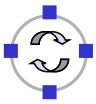
Programmierung und Algorithmen

Kapitel 4
Algorithmenparadigmen



Überblick

Einführung

Applikative Algorithmen

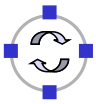
Imperative Algorithmen

Deduktive Algorithmen



Einführung

- Begriff **Algorithmenparadigma**
  - Allgemein: *„Paradigma“ [griechisch] das: Beispiel, Muster.* (Brockhaus)
  - Informatik: „Muster“ für Entwurf und Formulierung von Algorithmen
- Grundlegende Arten zur Notation von Algorithmen
  - Applikativ: Verallgemeinerung der Funktionsauswertung
  - Imperativ: basierend auf einem einfachen Maschinenmodell
    mit gespeicherten und änderbaren Werten
  - Deduktiv: basierend auf logischen Aussagen und Schlussfolgerungen
  - Weitere Paradigmen (objektorientiert, ...)
- Java: objektorientiert, imperativ, Elemente von applikativ



Applikative Algorithmen

Grundidee:

- Definition zusammengesetzter Funktionen durch Terme:

$$f(x) = 5x + 1$$

- Unbestimmte:
  - $x, y, z, \ldots$ vom Typ `int`
  - $q, p, r, \ldots$ vom Typ `bool`
- Terme mit Unbestimmten:
  - Terme vom Typ `int`

$$x, \quad x - 2, \quad 2x + 1, \quad (x + 1)(y - 1)$$

- Terme vom Typ `bool`

$$p, \quad p \wedge \texttt{true}, \quad (p \vee \texttt{true}) \Rightarrow (q \vee \texttt{false})$$



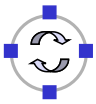
Funktionsdefinitionen

Sind v_1, ..., v_n Unbestimmte vom Typ T_1, .., T_n (**`bool`** oder **`int`**) und ist $t\ (v_1, ..., v_n)$ ein Term vom Typ T, so heißt

$$f\ (v_1, ..., v_n) = t\ (v_1, ..., v_n)$$

eine Funktionsdefinition vom Typ T .

- f : *Funktionsname*
- v_1, ..., v_n : *formale Parameter*
- $t\ (v_1, ..., v_n)$: *Funktionsausdruck*
- Zur Definition von Termen siehe Kapitel 2, Folie 48 zu int-Termen



Beispiele für Funktionsdefinitionen

1. f (p, q, x, y) = **`if`** p V q **`then`** $2x + 1$ **`else`** $3y - 1$ **`fi`**
2. $g(x)$ = **`if`** **`even`**(x) **`then`** $x \div 2$ **`else`** $3x - 1$ **`fi`**
3. $h(p, q)$ = **`if`** p **`then`** q **`else false fi`**



Beispiele für Funktionsdefinitionen /2

1. f (p, q, x, y) = **if** $p \vee q$ **then** $2x + 1$ **else** $3y - 1$ **fi**

   f : **bool** × **bool** × **int** × **int** → **int**

   f (**true**, **true**, 3, 4) = 7

2. g(x) = **if even**(x) **then** $x \div 2$ **else** $3x - 1$ **fi**

   g : **int** → **int**

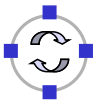
   $g(2) = 1$, $g(3) = 8$

3. $h(p, q)$ = **if** p **then** q **else false fi**

   h: **bool** × **bool** → **bool**

   h(**false**, **false**) = **false**

   Bemerkung: $h(p, q) = p \wedge q$



Erweiterung der Funktionsdefinition

- Erweiterung der Definition von Termen
- Neu: Aufrufe definierter Funktionen sind Terme
- Beispiel für Einsatz in Funktionsdefinitionen:

$f(x, y)$ = **if** $g(x, y)$ **then** $h(x + y)$ **else** $h(x - y)$ **fi**
$g(x, y)$ = $(x = y) \vee$ **odd**(y)
$h(x)$ = $j(x + 1) \cdot j(x - 1)$
$j(x)$ = $2x - 3$



Beispiel für Auswertung

$f(1, 2) \mapsto$ **if** $g(1, 2)$ **then** $h(1 + 2)$ **else** $h(1 - 2)$ **fi**

$\mapsto$ **if** $1 = 2 \vee$ **odd**(2) **then** $h(1 + 2)$ **else** $h(1 - 2)$ **fi**

$\mapsto$ **if false then** $h(1 + 2)$ **else** $h(1 - 2)$ **fi**

$\mapsto h(1 - 2)$

$\mapsto h(-1)$

$\mapsto j(-1 + 1) \cdot j(-1 - 1)$

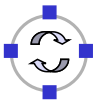
$\mapsto j(0) \cdot j(-1 - 1)$

$\mapsto j(0) \cdot j(-2)$

$\mapsto (2 \cdot 0 - 3) \cdot j(-2)$

$\mapsto^* (-3) \cdot (-7)$

$\mapsto 21$



Rekursive Funktionsdefinitionen

$f(x, y) =$ **if** $x = 0$ **then** y
else if $x > 0$ **then** $f(x - 1, y) + 1$
else $-f(-x, -y)$ **fi fi**

Auswertungen

$f(0, y) \mapsto y$ für alle y

$f(1, y) \mapsto f(0, y) + 1 \mapsto y + 1$

$f(2, y) \mapsto f(1, y) + 1 \mapsto (y + 1) + 1 \mapsto y + 2$

...

$f(n, y) \mapsto n + y$ für alle $n \in$ **int**, $n > 0$

$f(-1, y) \mapsto -f(1, -y) \mapsto -(1 - y) \mapsto y - 1$

...

$f(x, y) \mapsto x + y$ für alle x, $y \in$ **int**

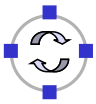


Applikative Algorithmen

Ein applikativer Algorithmus ist eine Menge von Funktionsdefinitionen

$$
\begin{aligned}
f_1(v_{1,1}, \ldots, v_{1,n_1}) &= t_1(v_{1,1}, \ldots, v_{1,n_1}),\\
&\vdots\\
f_m(v_{m,1}, \ldots, v_{m,n_m}) &= t_m(v_{m,1}, \ldots, v_{m,n_m}).
\end{aligned}
$$

Die erste Funktion f_1 wird wie beschrieben ausgewertet und bestimmt die „Bedeutung“ (Semantik) des Algorithmus.



Definiertheit

$f(x) =$ **`if`** $x = 0$ **`then`** 0 **`else`** $f(x - 1)$ **`fi`**

- Auswertungen:

$$
\begin{aligned}
f(0) &\mapsto 0\\
f(1) &\mapsto f(0) \mapsto 0\\
f(x) &\mapsto 0 \text{ für alle } x \geq 0\\
f(-1) &\mapsto f(-2) \mapsto \ldots \text{ Auswertung terminiert nicht!}
\end{aligned}
$$

- Also gilt:

$$
f(x) = \begin{cases} 0 & \text{falls } x \geq 0 \\ \bot & \text{sonst} \end{cases}
$$



Beispiel Fakultät

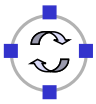
$x! = x(x - 1)(x - 2) \ldots 2 \cdot 1$ für $x > 0$

- Bekannte Definition: $0! = 1$, $x! = x \cdot (x - 1)!$
- Problem: negative Eingabewerte
- 1. Lösung:

$fac(x) =$ **`if`** $x = 0$ **`then`** 1 **`else`** $x \cdot fac(x - 1)$ **`fi`**

- Bedeutung:

$$fac(x) = \begin{cases} x! & \text{Falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$$



Beispiel Fakultät /2

- 2. Lösung:
  $fac(x) =$ **`if`** $x \leq 0$ **`then`** 1 **`else`** $x \cdot fac(x - 1)$ **`fi`**

- Bedeutung:

$$fac(x) = \begin{cases} x! & \text{Falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

  - Nachteil dieser Lösung: mathematisch nicht korrekt bei negativen Werten
  - Vorteil: Algorithmus terminiert immer und liefert korrektes Ergebnis bei Einhalten des Definitionsbereiches

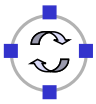


Fibonacci-Zahlen

- Zahlenreihe
- Progression bei Vermehrung von Kaninchen



1. Am Anfang gibt es ein Kaninchenpaar.
2. Jedes Paar wird im zweiten Monat zeugungsfähig.
3. Danach zeugt es jeden Monat ein weiteres Paar.
4. Kein Kaninchen stirbt.



Fibonacci-Zahlen /2

$$f_0 = f_1 = 1, \quad f_i = f_{i-1} + f_{i-2} \text{ für } i > 0$$

- Dabei ist f_{i-1} die Anzahl der Kaninchenpaare im letzten Monat und f_{i-2} die Anzahl der Kaninchenpaare im vorletzten Monat (i bezeichnet dabei den aktuellen Monat)

$$fib(x) = \texttt{if } (x = 0) \vee (x = 1) \texttt{ then } 1 \texttt{ else } fib(x - 2) + fib(x - 1) \texttt{ fi}$$

- Bedeutung:

$$fib(x) = \begin{cases} x\text{-te Fibonacci-Zahl} & \text{falls } x > 0 \\ \perp & \text{falls } x < 0 \\ 1 & \text{sonst} \end{cases}$$

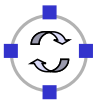


Produkt nur unter Verwendung der Addition

- Ausnutzung folgender Regeln:

$$
\begin{aligned}
0 \cdot y &= 0 \\
x \cdot y &= (x - 1) \cdot y + y \quad \text{für } x > 0 \\
x \cdot y &= -((-x) \cdot y) \quad \text{für } x < 0
\end{aligned}
$$

```
prod(x, y) = if x = 0  then 0 else
             if x > 0  then prod(x - 1, y) + y
                       else - prod( - x, y) fi  fi
```

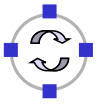


Größter gemeinsamer Teiler ggT

- Für $0 < x$ und $0 < y$ gilt:

$$
\begin{aligned}
ggT(x, x) &= x \\
ggT(x, y) &= ggT(y, x) \\
ggT(x, y) &= ggT(x, y - x) \quad \text{für } x < y
\end{aligned}
$$

```
ggT(x, y) = if (x ≤ 0) V (y ≤ 0)
              then ggT(x, y)
              else if x = y
                     then x
                     else if x > y
                            then ggT(y, x)
                            else ggT(x, y - x) fi  fi  fi
```

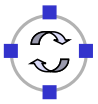


Größter gemeinsamer Teiler /2

- *ggT* korrekt für positive Eingaben, bei negativen Eingaben undefiniert (Berechnung terminiert nicht – der ggT kann hierfür jedoch auch sinnvoll definiert werden)

ggT(39, 15) ↦ *ggT*(15, 39) ↦ *ggT*(15, 24) ↦
ggT(15, 9) ↦ *ggT*(9, 15) ↦ *ggT*(9, 6) ↦
ggT(6, 9) ↦ *ggT*(6, 3) ↦ *ggT*(3, 6) ↦
ggT(3, 3) ↦ 3

- Berechnungsschema ist Formalisierung des Originalverfahrens von Euklid
  (Euklid: Elemente, 7. Buch, Satz 2; ca 300 v.Chr.).

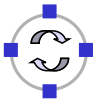


Test, ob eine Zahl gerade ist: *even*()

even(0) = **true**
odd(0) = **false**
even(*x* + 1) = *odd*(*x*)
odd(*x* + 1) = *even*(*x*)

```
even(x) = if x = 0 then true
                   else if x > 0 then odd(x - 1)
                                 else odd(x + 1) fi fi
odd(x)  = if x = 0 then false
                   else if x > 0 then even(x - 1)
                                 else even(x + 1) fi fi
```

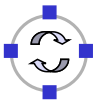
Algorithmus für *odd* durch Vertauschen der Reihenfolge der Funktionsdefinitionen



Primzahltest

$$
\begin{aligned}
prim(x) = \ & \texttt{if } x \leq 1 \texttt{ then false} \\
& \texttt{else } pr\,(2, x) \texttt{ fi} \\
pr\,(x, y) = \ & \texttt{if } x \geq y \texttt{ then true} \\
& \texttt{else } (y \texttt{ mod } x) \neq 0 \\
& \qquad \wedge\, pr\,(x + 1, y) \texttt{ fi}
\end{aligned}
$$

- Hilfsfunktion *pr (x, y)* ist wahr, genau dann wenn *y* durch keine Zahl *z*, mit $x \leq z < y$, teilbar ist
- $prim(x) \Leftrightarrow (x > 1) \wedge pr\,(2, x)$



Primzahltest /2

- Lösung nicht sehr effizient. . .
- Verbesserungen:
  - Ersetze $x \geq y$ durch $x \cdot x \geq y$ (kleinster Teiler muss kleiner oder gleich als Quadratwurzel sein!).
  - Ersetze *pr (x + 1, y)* durch

    if $x = 2$ **then** *pr* $(3, y)$ **else** *pr* $(x + 2, y)$ **fi**.



Äquivalenz applikativer Algorithmen

- McCarthys 91-Funktion

  $f(x) =$ **`if`** $x > 100$ **`then`** $x - 10$ **`else`** $f(f(x + 11))$ **`fi`**

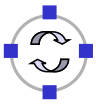
- Beispielberechnungen:

  $f(100) \mapsto f(f(111)) \mapsto f(101) \mapsto 91$
  $f(99) \mapsto f(f(110)) \mapsto f(100) \mapsto \ldots \mapsto 91$
  $\ldots \mapsto \ldots \mapsto 91$

- Gilt die folgende Äquivalenz?

  $f(x) =$ **`if`** $x > 100$ **`then`** $x - 10$ **`else`** 91 **`fi`**

- **Ja!** — *Beweis später in der Vorlesung.*



Äquivalenz applikativer Algorithmen /2

$f(x) =$ **`if`** $x = 1$ **`then`** 1 **`else`** $f(g(x))$ **`fi`**
$g(x) =$ **`if`** **`even`**(x) **`then`** $x \div 2$ **`else`** $3x + 1$ **`fi`**

- Für negative Zahlen und 0:

  $f(x) = \bot$ für $x \leq 0$

- Für positive Zahlen ist als definierter Wert nur 1 möglich

$f(1) \mapsto 1$
$f(2) \mapsto f(1) \mapsto 1$
$f(3) \mapsto f(10) \mapsto f(5) \mapsto f(16) \mapsto f(8) \mapsto f(4) \mapsto f(2) \mapsto 1$
$f(4) \mapsto \ldots \mapsto 1$
$\ldots$



Äquivalenz applikativer Algorithmen /3

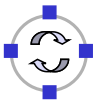
$f(x) = \texttt{if } x = 1 \texttt{ then } 1 \texttt{ else } f(g(x)) \texttt{ fi}$

$g(x) = \texttt{if even}(x) \texttt{ then } x \div 2 \texttt{ else } 3x + 1 \texttt{ fi}$

Verdacht: Es gilt folgende Äquivalenz:

$f(x) = \texttt{if } x > 0 \texttt{ then } 1 \texttt{ else } \bot \texttt{ fi}$

... aber noch unbewiesen.



Ackermann-Funktion

$$f(x, y) = \texttt{if } x \leq 0 \texttt{ then } y + 1 \texttt{ else if } y \leq 0 \texttt{ then } f(x - 1, 1) \texttt{ else } f(x - 1, f(x, y - 1)) \texttt{ fi fi}$$

- Werte wachsen unglaublich schnell an:

$f(0, 0) \mapsto 1$

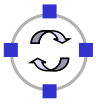
$f(1, 0) \mapsto f(0, 1) \mapsto 2$

$f(1, 1) \mapsto f(0, f(1, 0)) \mapsto f(0, f(0, 1)) \mapsto f(0, 2) \mapsto 3$

$f(2, 2) \mapsto f(1, f(2, 1)) \mapsto f(1, f(1, f(2, 0))) \mapsto f(1, f(1, f(1, 1)))$
$\ldots \mapsto f(1, 5) \mapsto \ldots \mapsto f(0, 6) \mapsto 7$

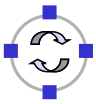
$f(3, 3) \mapsto 61$

$f(4, 4) \mapsto 2^{2^{2^{2^{2^{2}}}}} - 3 = 2^{2^{2^{65536}}} - 3$



Imperative Algorithmen

- Verbreitetste Art, Algorithmen für Computer zu formulieren
- Basiert auf den Konzepten Anweisung und Variablen
- Wird durch Programmiersprachen Java, C, PASCAL, FORTRAN, COBOL, . . . realisiert
- Hier zunächst nur Darstellung der Grundprinzipien



Variablen

- Eine Variable besteht aus einem Namen (z. B. X) und einem veränderlichen Wert.
- Ist *t* ein Term ohne Unbestimmte und *w*(*t*) sein Wert, dann heißt das Paar *x* := *t* eine Wertzuweisung.
  Ihre Bedeutung ist festgelegt durch
  Nach Ausführung von x := t gilt x = w(t).

  Vor der Ausführung der ersten Wertzuweisung gilt $x = \perp$ (undefiniert).

- Beispiele:
  X := 7 *F* := *true*
  X := (3 - 7) · 9 *Q* := ¬ (*true* V ¬ *false*) V ¬¬*true*



Zustände

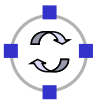
- Ist $\underline{X} = \{ X_1, X_2, \ldots \}$ eine Menge von Variablen(-namen), von denen jede Werte aus der Wertemenge W haben kann (alle Variablen vom gleichen Typ), dann ist der Zustand Z eine partielle Abbildung

$$Z : \underline{X} \rightarrow W \text{ (Zuordnung des momentanen Wertes)}$$

- Ist $Z : \underline{X} \rightarrow W$ Zustand und wählt man eine Variable X und einen Wert w, so ist der transformierte Zustand wie folgt definiert:

$$Z \quad \langle X \leftarrow w \rangle \quad : \quad \underline{X} \rightarrow W \text{ mit}$$

$$Y \mapsto \begin{cases} w, & \text{falls } X = Y \\ Z(Y), & \text{sonst} \end{cases}$$



Ausdrücke

- Ausdrücke entsprechen im Wesentlichen den Termen einer applikativen Sprache, jedoch stehen an der Stelle von Unbestimmten nun Variablen
- Auswertung von Termen ist zustandsabhängig:
  - Gegebener Term $2X + 1$
  - Wert im Zustand Z ist $2 \cdot Z(X) + 1$



Wert eines Ausdrucks

- Wert eines Ausdrucks $t\ (X_1, \ldots, X_n)$ wird mit $Z(t\ (X_1, \ldots, X_n))$ bezeichnet:

$$Z(2 \cdot X + 1) = 2 \cdot Z(X) + 1$$

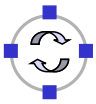
- Damit auch möglich:

$$X := t\ (X_1, \ldots, X_n)$$

- Transformierter Zustand ist

$$[\![\ X := t\ (X_1, \ldots, X_n)\ ]\!]\ (Z) = Z \quad \langle X \leftarrow Z(t\ (X1\ ,\ldots,Xn))\ \rangle$$

- "Semantikklammern": Bedeutung einer Anweisung als Funktion auf Zuständen



Zuweisungen als Anweisungen /1

Beispielszuweisung:

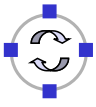
$$\alpha_1 = (X := 2 \cdot Y + 1)$$

- Transformation in Funktion auf Zuständen

$$[\![\ \alpha_1\ ]\!]\ (Z) = Z \quad \langle X \leftarrow 2Z(Y)+1\ \rangle$$

- Zuweisung berechnet neuen Zustand
  - Alter Zustand Z

    Neuer Zustand $[\![\ \alpha_1\ ]\!]\ (Z)$



Zuweisungen als Anweisungen /2

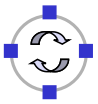
Beispielzuweisung mit der selben Variable auf beiden Seiten:

$$\alpha_2 = (X := 2 \cdot X + 1)$$

- Transformation in

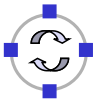
$$[\![\alpha_2]\!] (Z) = Z_{\langle X \leftarrow 2Z(X)+1 \rangle}$$

- Bei der letzten Anweisung handelt es sich *nicht* um eine rekursive Gleichung!



Komplexe Anweisungen

- Bisher: Anweisungen als Funktionen auf Zuständen
- Konstrukte zum Aufbau von imperativen Algorithmen
  1. Sequenz
  2. Auswahl
  3. Iteration
- Semantik wird wiederum durch Konstruktion von Funktionen festgelegt



Sequenz

- Folge (Sequenz): Sind α_1 und α_2 Anweisungen, so ist

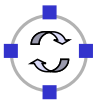
  $\alpha_1; \alpha_2$

  auch eine Anweisung.

- Zustandstransformation

$$[\![\alpha_1; \alpha_2]\!] (Z) = [\![\alpha_2]\!] ([\![\alpha_1]\!] (Z))$$

- Semantik: Schachteln der Funktionsaufrufe



Selektion

- Auswahl (Selektion): Sind α_1 und α_2 Anweisungen und B ein Boolscher Ausdruck, so ist

  `if` B **`then`** $\boldsymbol{\alpha_1}$ **`else`** $\boldsymbol{\alpha_2}$ **`fi`**

  eine Anweisung.

$$[\![\textbf{if } B \textbf{ then } \boldsymbol{\alpha_1} \textbf{ else } \boldsymbol{\alpha_2} \textbf{ fi}]\!] (Z) = \begin{cases} [\![\alpha_1]\!] (Z) \text{ falls } Z(B) = \textbf{true} \\ [\![\alpha_2]\!] (Z) \text{ falls } Z(B) = \textbf{false} \end{cases}$$

- Voraussetzung: $Z(B)$ definiert — sonst Bedeutung der Auswahlanweisung undefiniert.



Iteration

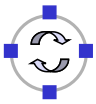
- Wiederholung (Iteration): Ist Anweisung und *B* Boolscher Ausdruck, so ist

  `while` B **`do`** ***α*** **`od`**

  eine Anweisung.

$$[\![\textbf{while } B \textbf{ do } \alpha \textbf{ od}]\!](Z) = \begin{cases} Z, & \text{falls } Z(B) = \textbf{false} \\ [\![\textbf{while } B \textbf{ do } \alpha \textbf{ od}]\!]([\![\alpha]\!](Z)), & \text{sonst} \end{cases}$$

- Ist $Z(B)$ undefiniert, so ist die Bedeutung dieser Anweisung ebenfalls undefiniert.



Umsetzung in Programmiersprachen

- In realen imperativen Programmiersprachen gibt es fast immer diese Anweisungen, meistens jedoch viel mehr.
- **`while`**-Schleifen sind rekursiv definiert, sie brauchen nicht zu terminieren.
- Die Verwendung von **`if-fi`** ist wegen der klaren Klammerung sauberer.
- Bereits Programmiersprachen mit diesen Sprachelementen sind *universell* (d.h. man kann alle berechenbaren Funktionen mit ihnen berechnen; Beweise werden in Vorlesungen zur Theorie behandelt).
- Wir beschränken uns im Folgenden auf die Datentypen **`bool`** und **`int`**.



Syntax imperativer Algorithmen

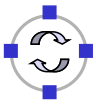
< Programmname >:

var $X, Y, \ldots :$ **int**; $P, Q, \ldots :$ **bool**; ⇐ Variablen-Deklaration

input $X_1, \ldots, X_n$; ⇐ Eingabe-Variablen

α; ⇐ (Anweisung(en)

output $Y_1, \ldots, Y_m$. ⇐ (Ausgabe-Variablen



Semantik imperativer Algorithmen

- Die Bedeutung (Semantik) eines imperativen Algorithmus ist eine partielle Funktion:

$$
\begin{array}{lll}
[\![\text{PROG}]\!] : & W_1 \times \ldots \times W_n & \mapsto \; V_1 \times \ldots \times V_m \\
[\![\text{PROG}]\!] & (w_1, \ldots, w_n) & = \; (Z(Y_1), \ldots, Z(Y_m)) \\
\text{wobei} & Z & = \; [\![\alpha]\!] (Z_0), \\
 & Z_0(X_i) & = \; w_i \,, \; i = 1, \ldots, n, \\
\text{und} & Z_0(Y) & = \; \perp \text{ für alle Variablen } Y \neq X_i \,, \; i = 1, \ldots, n
\end{array}
$$

wobei gilt

PROG	Programmname
$W_1, \ldots, W_n$	Wertebereiche der Typen von $X_1, \ldots, X_n$
$V_1, \ldots, V_m$	Wertebereiche der Typen von $Y_1, \ldots, Y_m$



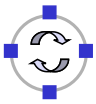
Kurze Charakterisierung

Die Algorithmenausführung imperativer Algorithmen besteht aus einer Folge von Basisschritten, genauer von Wertzuweisungen.

Diese Folge wird mittels Selektion und Iteration basierend auf booleschen Tests über dem Zustand konstruiert.

Jeder Basisschritt definiert eine elementare Transformation des Zustands.

Die Semantik des Algorithmus ist durch die Kombination all dieser Zustandstransformationen zu einer Gesamttransformation festgelegt.



Beispiele für imperative Algorithmen

- Fakultätfunktion $0! = 1$, $x! = x \cdot (x - 1)!$ für $x > 0$

```
FAC:var X,Y: int;
        input X;
        Y:=1;
        while X>1 do Y:=Y · X; X:=X-1 od
        output Y.
```

Hier ist $[\![FAC]\!](x) = \begin{cases} x! & \text{für } x \geq 0 \\ 1 & \text{sonst} \end{cases}$

- Falls Bedingung für die **`while`**-Schleife „$X \neq 0$“:

$$[\![FAC]\!](x) = \begin{cases} x! & \text{für } x \geq 0 \\ \bot & \text{sonst} \end{cases}$$



Beispiel für Semantik imperativer Algorithmen

- Gesucht ist das Ergebnis des Aufrufs *FAC*(3)
  - Abkürzung **`while`** *β* für die Zeile

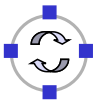
    `while` `X>1` **`do`** `Y:=Y * X; X:=X-1` **`od`**
  - Signatur der Semantikfunktion:

    $[\![FAC]\!] : \texttt{int} \rightarrow \texttt{int}$
  - Funktion ist durch Lesen von *Y* im Endzustand *Z* definiert:

    $[\![FAC]\!](w) = Z(Y)$
  - Endzustand *Z* ist definiert durch folgende Gleichung:

    $Z = [\![\alpha]\!](Z_0)$

    wobei α die Folge aller Anweisungen des Algorithmus ist.
  - Initialer Zustand Z_0 definiert als $Z_0 = (X = w, Y = \perp)$.
  - Zustände abkürzend ohne Variablennamen: $Z_0 = (w, \perp)$.



Beispiel für Semantik imperativer Algorithmen /2

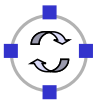
$$
\begin{aligned}
Z &= [\![\alpha]\!](Z_0) \\
&= [\![\alpha]\!](3, \perp) \\
&= [\![Y := 1; \text{while } \beta]\!](3, \perp) \\
&= [\![\text{while } \beta]\!]([\![Y := 1]\!](3, \perp)) \\
&= [\![\text{while } \beta]\!]((3, \perp)\ \langle Y \leftarrow 1 \rangle) \\
&= [\![\text{while } \beta]\!](3, 1) \\
&= \begin{cases} Z, \text{ falls } Z(B) = \texttt{false} \\ [\![\texttt{while } B \texttt{ do } \alpha' \texttt{ od}]\!]([\![\alpha']\!](Z)), \text{ sonst} \end{cases} \\
&= \begin{cases} (3, 1), \text{ falls } Z(X > 1) = (3 > 1) = \texttt{false} \\ [\![\texttt{while } \beta]\!]([\![Y := Y * X; X := X - 1]\!](Z)), \text{ sonst} \end{cases} \\
&= [\![\text{while } \beta]\!]([\![Y := Y * X; X := X - 1]\!](3, 1)) \\
&= [\![\text{while } \beta]\!]([\![X := X - 1]\!]([\![Y := Y * X]\!](3, 1))) \\
&= [\![\text{while } \beta]\!]([\![X := X - 1]\!](3, 3)) \\
&= [\![\text{while } \beta]\!](2, 3)
\end{aligned}
$$

... wird fortgesetzt...



Beispiel für Semantik imperativer Algorithmen /3

$$
\begin{aligned}
Z \quad &= \quad [\![\alpha]\!] (Z_0) \\
&\ldots \\
&= \quad [\![\textbf{while } \beta]\!] (2, 3) \\
&= \quad \begin{cases} (2, 3), \text{ falls } Z(X > 1) = (2 > 1) = \textbf{false} \\ [\![\textbf{while } \beta]\!] ([\![\ Y := Y * X; X := X - 1 \ ]\!] (Z)), \text{ sonst} \end{cases} \\
&= \quad [\![\textbf{while } \beta]\!] ([\![\ Y := Y * X; \ X := X - 1 \ ]\!] (2, 3)) \\
&= \quad [\![\textbf{while } \beta]\!] ([\![\ X := X - 1 \ ]\!] ([\![\ Y := Y * X \ ]\!] (2, 3))) \\
&= \quad [\![\textbf{while } \beta]\!] ([\![\ X := X - 1 \ ]\!] (2, 6)) \\
&= \quad [\![\textbf{while } \beta]\!] (1, 6) \\
&= \quad \begin{cases} (1, 6), \text{ falls } Z(X > 1) = (1 > 1) = \textbf{false} \\ [\![\textbf{while } \beta]\!] ([\![\ Y := Y * X; X := X - 1 \ ]\!] (Z)), \text{ sonst} \end{cases} \\
&= \quad (1, 6)
\end{aligned}
$$



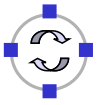
Beispiel für Semantik imperativer Algorithmen /4

- Der Übergang von der 3. auf die 4. Zeile folgt der Definition der Sequenz, indem der Sequenzoperator in einen geschachtelten Funktionsaufruf umgesetzt wird.
- Nur in der 5. Zeile wurde eine Wertzuweisung formal umgesetzt, später sind sie einfach verkürzt direkt ausgerechnet.
- In der 7. Zeile haben wir die Originaldefinition der Iteration eingesetzt (nur mit dem Kürzel α' statt α, da α bereits verwendet wurde). Im Beispiel gilt

$$\alpha' = \{ Y := Y * X; X : X - 1 \}$$

- Das Z in der 7. und 8. Zeile steht für den Zustand (3, 1) (in späteren Zeilen analog für den jeweils aktuellen Zustand).
- Bei diesem Beispiel sieht man Folgendes sehr deutlich: *Die Ausführung einer While-Schleife erfolgt analog einer rekursiven Funktionsdefinition!*
- Damit gilt:

$$[\![\ FAC \ ]\!] (3) = Z(Y) = (X = 1, \ Y = 6)(Y) = 6$$

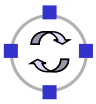


Fibonacci-Zahlen

```
FIB: var X,A,B,C: int;
     input X;
     A:=1, B:=1;
     while X>0 do
       C:=A+B; A:=B; B:=C; X:=X-1
     od
     output A.
```

$$[\![FIB]\!](x) = \begin{cases} \text{die x-te Fibonacci-Zahl,} & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

Hinweis: Die Zwischenspeicherung der beiden letzten Fibonacci-Zahlen in den Variablen A und B ist effizienter als der doppelt rekursive Aufruf der applikativen Version



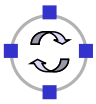
Größter gemeinsamer Teiler: Version 1

```
GGT1   var X,Y: int;
       input X,Y;
       while X ≠ Y do
         while X>Y do X:=X-Y od
         while X<Y do Y:=Y-X od
       od
       output X.
```



Größter gemeinsamer Teiler: Auswertung

X	Y
19	5
14	5
9	5
4	5
4	1
3	1
2	1
1	1



Größter gemeinsamer Teiler: Version 2

```
GGT2  var X,Y,R: int;
      input X,Y;
      R:=1;
      while R ≠ 0 do
            R:=X mod Y; X:=Y; Y:=R od
      output X.
```

Einige Berechnungen für GGT2:

X	*Y*	*R*
19	5	1
5	4	4
4	1	1
1	0	0

X	*Y*	*R*
2	100	1
100	2	1
2	0	0

ist X < Y, so wird erst vertauscht: X ↔ Y

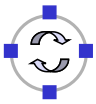


Größter gemeinsamer Teiler /3

- Wie ist das Verhalten von *GGT2* für negative X oder Y?

$$[\![GGT2]\!](x, y) = \begin{cases} ggT(x, y), & \text{falls } x, y > 0 \\ y, & \text{falls } x = y \neq 0 \text{ oder } x = 0, y \neq 0 \\ \bot & \text{falls } y = 0 \\ ggT(|x|,|y|), & \text{falls } x < 0 \text{ und } y > 0 \\ -ggT(|x|,|y|), & \text{falls } y < 0 \end{cases}$$

- Ist GGT2 effizienter als Version 1?



Deduktive Algorithmen

- Basierend auf logischen Aussagen
- Programmiersprachen wie PROLOG
- Logische Aussagen ergeben - streng genommen - allein kein Berechnungsmodell
- Deduktiver Algorithmus =
  - Logische Aussagen +
  - Interpretationsalgorithmus +
  - Anfrage



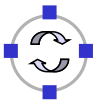
Aussagen und Aussageformen

- Aussage:
  Susi ist Tochter von Petra
- Aussageform mit Unbestimmten:
  X ist Tochter von Y
- Belegung:

$$X \mapsto \text{Susi};\ Y \mapsto \text{Petra}$$

- Atomare Formel:

$$\mathit{Tochter}(X, Y)$$



Logik der Fakten und Regeln

- Alphabet
  - Unbestimmte *X, Y, Z*, ...
  - Konstanten *a, b, c*, ...
  - Prädikatensymbole *P, Q, R*, ... mit Stelligkeit (analog der Signatur von ADT)
  - Logische Konnektive Λ und $\Rightarrow$
- Atomare Formeln: $P(t_1, .., t_n)$
- Fakten: atomare Formeln ohne Unbestimmte
- Regeln (α_i ist atomare Formel):

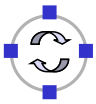
$$\alpha_1 \Lambda \alpha_2 \Lambda \cdots \Lambda \alpha_m \Rightarrow \alpha_0$$

Auch leere Prämissen möglich (entsprechen **`true`**)!



Beispiel

```
Fakten:
    1. TOCHTER(Susi,Petra)
    2. TOCHTER(Petra,Rita)
Regeln:
    3. TOCHTER(X,Y) ∧ TOCHTER(Y,Z)
          ⇒ ENKELIN(X,Z)
```



Ableitung neuer Fakten

Vereinfachte Darstellung:

- Finde Belegung der Unbestimmten einer Regel, so dass auf der linken Seite (Prämisse) bekannte Fakten stehen
- Rechte Seite ergibt dann neuen Fakt

- Logische Regel: *Modus Ponens*
  (lat. ponere "stellen, setzen": setzende Schlussfigur, d.h. Schlussfigur, bei der eine positive Aussage hergeleitet wird)

$$\frac{P \Rightarrow Q\,;\; P}{Q}$$



Beispiel II

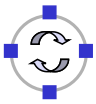
```
Fakten:  1. TOCHTER(Susi,Petra)
         2. TOCHTER(Petra,Rita)
Regeln:  3. TOCHTER(X,Y) ∧ TOCHTER(Y,Z)
            ⇒ ENKELIN(X,Z)
```

Belegung

$$X \mapsto Susi,\ Y \mapsto Petra,\ Z \mapsto Rita$$

ergibt Fakt

$$ENKELIN(Susi,Rita)$$



Deduktiver Algorithmus

- Ein deduktiver Algorithmus D ist eine Menge von Fakten und Regeln.
- $F(D)$ sind alle direkt oder indirekt aus D ableitbaren Fakten.
- Eine Anfrage γ ist eine Konjunktion von atomaren Formeln.

$$\gamma = \alpha_1 \wedge \ldots \wedge \alpha_m$$

- Eine Antwort ist eine Belegung der Unbestimmten in γ , bei der aus allen α_i Fakten aus $F(D)$ werden.



Beispiel Addition

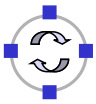
Fakten:

SUC(n, n + 1) für alle $n \in \mathbb{N}$

Regeln:

(1) $\Rightarrow$ ADD(X,0,X)

(2) ADD(X,Y,Z) $\wedge$ SUC(Y,V) $\wedge$ SUC(Z,W)
$\Rightarrow$ ADD(X,V,W)



Addition: Anfragen

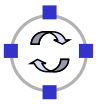
- ADD(3,2,5)? liefert **true**
  Ableitung:
  Regel (1) mit X=3;
  Regel (2) mit Belegung 3,0,3,1,4; // angegeben als x, y, z, v, w
  Regel (2) mit Belegung 3,1,4,2,5;
- ADD(3,2,X)? liefert $X \mapsto 5$
- ADD(3,X,5)? liefert $X \mapsto 2$
- ADD(X,Y,5)? liefert
  $(X, Y) \in \{ (0, 5), (1, 4), (2, 3), (3, 2), (1, 4), (5, 0) \}$
- ADD(X,Y,Z)? liefert unendliches Ergebnis



Auswertung logischer Anfragen

- Vollständiger Algorithmus würde Zeitrahmen dieser Vorlesungsstunde sprengen
- Idee:
  1. Starte mit γ als zu untersuchende Anfragemenge
  2. Untersuche Belegungen, die
     - einen Teil von γ mit Fakten gleichsetzen bzw.
     - einen Fakt aus γ mit einer rechten Seite einer Regel gleichsetzen
  3. Wende passende Regeln “rückwärts” an (ersetze Konklusion durch Prämissen in zu untersuchender Anfragemenge)
  4. Entferne gefundene Fakten aus der Anfragemenge

Letzte Schritte wiederholen solange bis γ leer ist



Auswertung logischer Anfragen II

Vollständige Lösung:

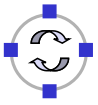
`BACKTRACKING !`

(d.h. man muss manchmal Schritte „geordnet“ zurücknehmen)

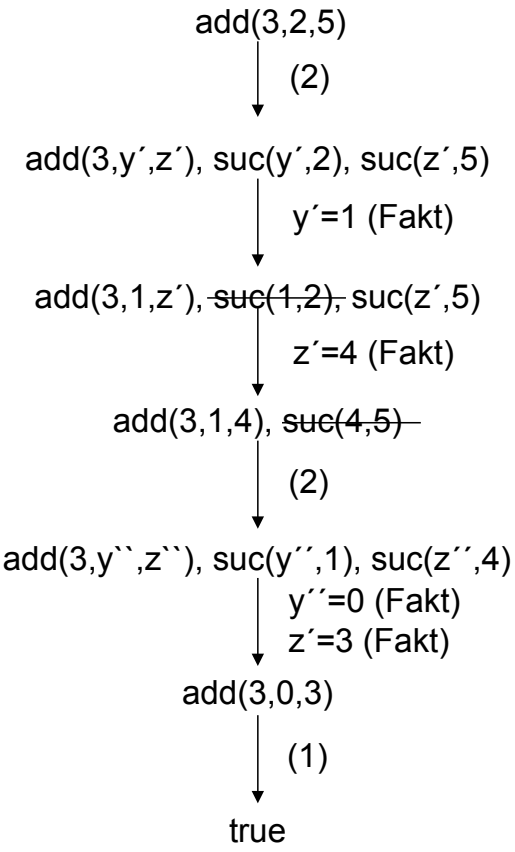
- Spezielle Form der Rekursion

Hier nur vereinfachter Formalismus:

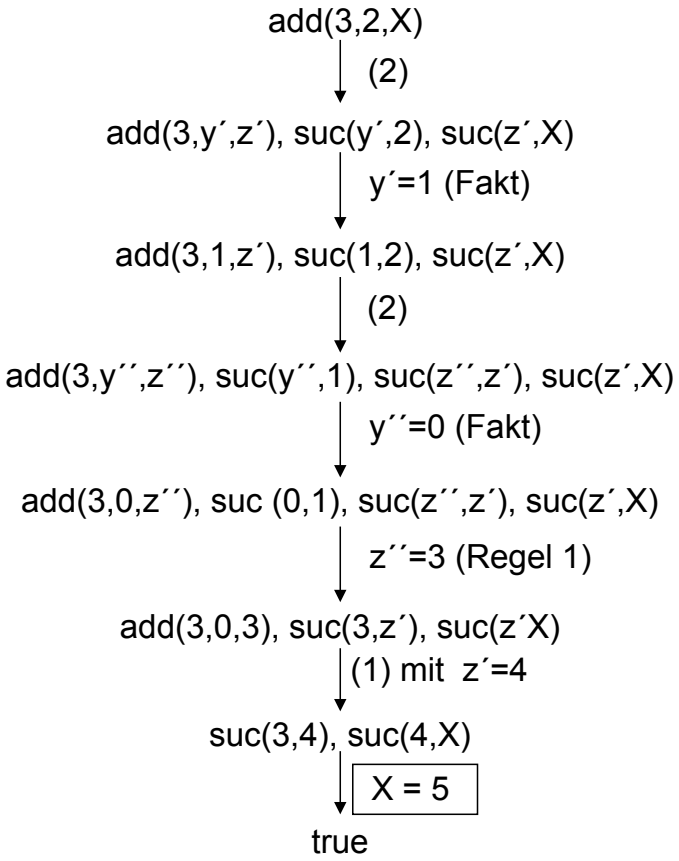
- keine Negation;
- kein Erkennen von Endlosrekursionen,
- kein Erkennen von unendlichen Belegungsmengen etc.



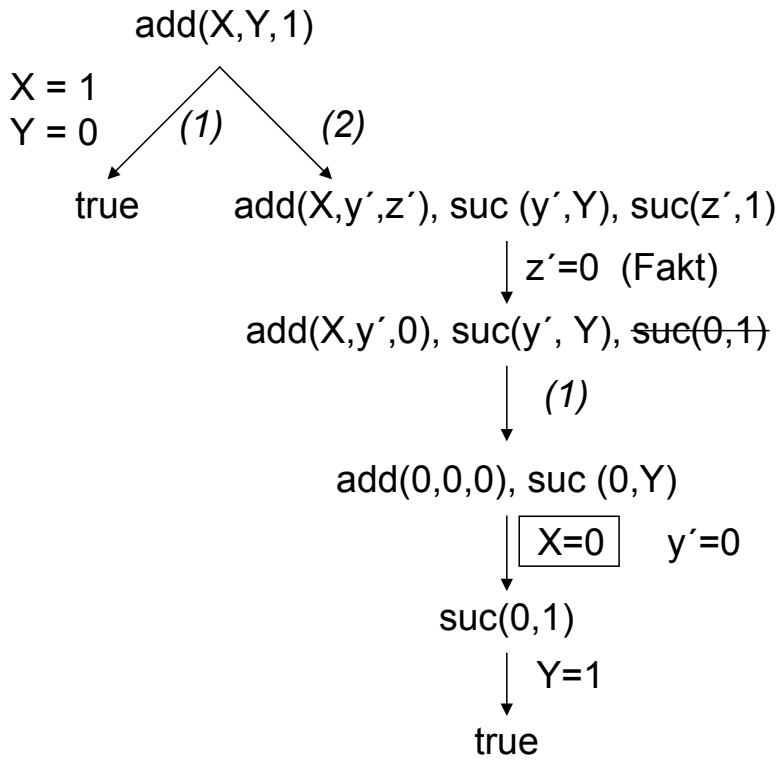
Auswertung: Beispiel 1

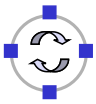


Auswertung: Beispiel 2



Auswertung: Beispiel 3





Weitere Paradigmen

- Weitere Paradigmen für Berechnungen
- Objektorientierung
  - Nicht nur für Algorithmen: Entwurf, Daten, Dokumente, etc.
  - Später im Java-Teil
- DNA Computing
- Quantenalgorithmen
- etc.



Zusammenfassung

- Unterschiedliche Algorithmenparadigmen
- Applikative Algorithmen:
  - Mengen von Funktionsdefinitionen
  - Rekursive Anwendung
- Imperative Algorithmen:
  - Anweisungen und Variablen
  - Syntax und Semantik
- Deduktive Algorithmen
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukture*n, Kapitel 3

