# “Theorie der Informatik” (CS206) Ackermann-Fkt, GOTO-Programme, Bezug WHILE-Prog., $\mu$-Rekursion

27. März 2013

Proff Malte Helmert und Christian Tschudin

Departement Mathematik und Informatik, Universität Basel

## Wiederholung/Einstieg

1. Wie können Turingmaschinen “komponiert” (zusammengehängt) werden?
2. Wie sieht eine WHILE-Maschine aus?
3. Was ist ein LOOP-Programm?
4. Terminiert jedes LOOP-Programm?
5. Was ist eine “primitiv-rekursive Funktion” ?
6. Warum gibt es zu jedem (LOOP-) Programm eine Funktion, die “das gleiche berechnet”?

# LOOP-Programm für `monus`

Programmiere zuerst die Vorgängerfunktion $pred(x)$, die wie folgt (primitiv-rekursiv!) definiert ist:

$pred(0) = 0$
$pred(n+1) = n$

```
pred(x):
  a = 0
  result = 0
  LOOP x DO
    result = a
    a = SUCC(a)
  OD
```

```
monus(x,y):
  result = x
  LOOP y DO
    result = pred(result)
  OD
```



# Ackermann-Funktion

Die Ackermann-Funktion war eines der ersten Beispiele (1928) einer arithmetischen Funktion, die *nicht* LOOP-berechenbar ist.

Rekursive Definition:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Siehe Demo: viele Berechnungsschritte, *sehr* schnelles Wachstum

# Ackermann-Funktion (Forts)

| m\n | 0 | 1 | 2 | 3 | 4 | n |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | $n+1$ |
| 1 | 2 | 3 | 4 | 5 | 6 | $n+2$ |
| 2 | 3 | 5 | 7 | 9 | 11 | $2n+3$ |
| 3 | 5 | 13 | 29 | 61 | 125 | $2^{(n+3)}-3$ |
| 4 | 13 | 65533 | $2^{65536}-3$ | $2^{2^{65536}}-3$ | $A(3, A(4, 3))$ | $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3 \text{ twos}}-3$ |
| 5 | 65533 | $A(4, 65533)$ | $A(4, A(5, 1))$ | $A(4, A(5, 2))$ | $A(4, A(5, 3))$ | $A(4, A(5, n-1))$ |
| 6 | $A(5, 1)$ | $A(5, A(5, 1))$ | $A(5, A(6, 1))$ | $A(5, A(6, 2))$ | $A(5, A(6, 3))$ | $A(5, A(6, n-1))$ |



# Uebersicht

Bisher:

- WHILE-Maschine, WHILE-Programm
- LOOP-Programme,
  und primitiv-rekursive Funktionen

Folgend:

- GOTO-Programme, Aequivalenz zu WHILE-Programmen

Anschliessend:
$\mu$-Rekursion, Aequivalenz zu WHILE-Programmen

# GOTO-Programm – Definition

Grundidee: Programmierung einer einfachen (Register)-CPU

- **Programm = Sequenz von Zeilen** mit
  `MARKE : ANWEISUNG`
- Anweisungen sind:
  - Wert-Zuweisung $x_i$ := $x_j$ + c
  - unbedingter Sprung: GOTO $M_i$
  - bedingter Sprung: IF $x_i$ == c THEN GOTO $M_i$
  - Stopanweisung: HALT

Bemerkung – Möglichkeit unendlicher Schleifen: `M1 : GOTO M1`



# GOTO-Programm – Beispiel

Berechnung der Fakultätsfunktion $n!$

```
M0: n = ...    # Parameter n
M1: f = 1
M2: IF (n <= 1) THEN GOTO M6
M3: f = f * n
M4: n = n - 1
M5: GOTO M2
M6: HALT       # Resultat f
```

```
    n = ...
    f = 1
L1: IF (n <= 1) THEN GOTO L2
    f = f * n
    n = n - 1
    GOTO L1
L2: HALT
```

(fakultät() kann durchaus als LOOP-Program formuliert werden)

# WHILE- und GOTO-Programme

- Jedes GOTO-Programm kann als WHILE-Programm umgeschrieben werden
- Jedes WHILE-Programm kann als GOTO-Programm umgeschrieben werden



# Ein WHILE-Prog in ein GOTO-Prog umformen

Aus

```
WHILE x <> 0 DO  P  OD
```

wird

```
M1 : IF x = 0 THEN GOTO M4
M2 : P
M3 : GOTO M1
M4 : ...
```

# Ein GOTO-Prog in ein WHILE-Prog umformen

Ein GOTO-Programm

```
M1 : A1;    M2 : A2;  ...   Mk : Ak
```

wird abgearbeitet durch

```
count = 1
WHILE count <> 0 DO
  IF count = 1 THEN A1';
  IF count = 2 THEN A2';
  ...
OD
```

wobei $A_i$' wie folgt definiert ist (siehe nächstes Slide)



# GOTO $\rightarrow$ WHILE (Forts)

$A_i$' ist wie folgt definiert:

- falls $A_i$ die Form $x_j = x_l + c$ hat:
  $x_j$ = $x_l$ + c; count = count + 1
- falls $A_i$ die Form $GOTO\ M_n$ hat:
  count = n
- falls $A_i$ die Form $IF\ x_j = c\ THEN\ GOTO\ M_n$ hat:
  IF $x_j$=c THEN count=n ELSE count=count+1
- falls $A_i$ die Form $HALT$ hat:
  count = 0

# Stellenwert von GOTO für die Programmierpraxis

- Heute gängige CPUs sind GOTO-Maschinen:
  - Register (inkl Hauptspeicher)
  - sequenzielles Abarbeiten von Maschinenbefehlen (Marke = Speicheradresse)
  - explizite Sprungbefehle: `jump, jump-conditional`
- GOTO ist heute auf Assembler-Bereich beschränkt
  - Programmier-Hochsprachen vermeiden GOTOs (siehe nächstes Slide)
  - GOTO ist in C noch möglich ...



# Stellenwert von GOTO (Fortsetzung)

Edsger W. Dijkstra: ***Go To Statement Considered Harmful*** (CACM 1968)

- Entwurf von Programmiersprachen bringt Einsicht, dass GOTO zu schlechten Programmen führt. ($\rightarrow$ "strukturiertes Programme" statt "Spaghetti-Code")
- Titel des Beitrags ist in Computer Science zu einer Redewendung geworden: *"XYZ Considered Harmful"*

*<< For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so. >>*

# Zwischenstand:

- LOOP $\approx$ primitiv-rekursiv
- LOOP ist weniger mächtig als WHILE
- GOTO $\approx$ WHILE $\approx$ ??
  $\Rightarrow \mu$-Rekursion



# $\mu$-Rekursion – Definition

Echte Erweiterung der primitiv-rekursiven Funktionen mit einem $\mu$-Operator (Minimierung)

Sei $f$ eine $k+1$-stellige Funktion. Nach Anwendung des $\mu$-Operators entsteht die Funktion $g : \mathbb{N} \to \mathbb{N}$ wie folgt:

$$g(x_1, \ldots, x_k) := min\{n | f(n, x_1, \ldots, x_k) = 0 \\ und\ f(m, x_1, \ldots, x_k)\ definiert\ \forall\, m < n\}$$

D.h.: $g()$ ergibt das kleinste $n$, für welches $f() = 0$

Vorsicht: $g()$ kann partielle Funktion sein, da $min\{\}$ leer sein kann. $g(x_1, \ldots, x_n)$ ist dann undefiniert!

# Beweis, dass WHILE Programme $\approx$ $\mu$-rekursiv

(Aufbauend auf LOOP $\approx$ primitive-rekursiv)

Vorbereitung für die Richtung "WHILE $\Rightarrow$ $\mu$-rekursiv"

- Was ist die Funktion zu `WHILE x`$_i$`<>0 DO Q OD;` ?
- Sei $h(n, \bar{x})$ := Konfiguration (der Register) nach der $n$-ten Ausführung von `Q` ($\bar{x}$ ist Registerwertvektor $< x_1, \ldots x_k >$)
- Sei $d_i(m)$ das $i$-te Register der Konfiguration $m$, d.h. $d()$ "dekodiert" eine Konfiguration

*. . . Fortsetzung . . .*



# Beweis, dass WHILE Programme $\approx$ $\mu$-rekursiv (Forts)

Betrachte $d_i(h(n, m_0))$, wobei $m_0$ die Startkonfiguration ist:

- $d_i(h(n, m_0))$ ist der Wert von $x_i$ nach $n$ Ausführungen
- Wende $\mu$-Operator an:
  $\mu(d_i\, h)(m_0)$ ist das kleinste $n$ wo $x_i = 0$
- Nun einsetzen:
  $m_{resultat} := h(\mu(d_i\, h)(m_0),\, m_0)$

D.h. wir konnten das Programm in eine Funktion übersetzen (dank dem $\mu$-Operator, der die notwendige Anzahl Durchgänge der WHILE-Schleife "berechnet")

# Beweis, dass WHILE Programme $\approx$ $\mu$-rekursiv

Richtung "$\mu$-rekursiv $\Rightarrow$ WHILE"

- Sei $f()$ eine Funktion, welche ein WHILE-Programm berechnet.
- Zu zeigen: für $\mu f$ gibt es auch ein WHILE-Programm
- Programm:

```
x0 := 0;
y  := f(0, x1, ... xn);
WHILE  y <> 0  DO
   x0 := x0 + 1;
   y := f(x0, x1, ... xn);
OD
```



# Bewertung von $\mu$-Rekursiv

WHILE-Programme können "beliebig" lange laufen:

- Für endliche Laufzeit (= Abbruchbedingung wird irgend einmal erfüllt) gibt uns der $\mu$-Operator die Anzahl Schleifendurchläufe an.

Der Unterscheid von LOOP-Programmen zu WHILE-Programmen kann auf diese "Vorhersage"-Funktion zurückgeführt werden:

– Falls diese Vorhersage (für alle Inputwerte) berechenbar ist, bleibt ein Programm in der LOOP-Klasse.

Für Programme die nie abbrechen, darf diese Vorhersage kein Resultat liefern, d.h. selbst auch nicht abbrechen.

# Alonzo Church

- Alonzo Church, 1903 – 1995, USA
- Unentscheidbarkeit der Peano-Arithmetik
- 1936 “Lambda Calculus”, ist für (funktionale) Programmiersprachen zentral, siehe etwa LISP.
- Doktorvater von Stephen Kleene und Alan Turing



# Church–Turing These

*Die Klasse der “intuitiv berechenbaren” Funktionen stimmt genau mit der Klasse der TM-berechenbaren Funktionen überein.*

Wird nie beweisbar sein (weil “intuitiv” nicht formaliserbar ist), ist aber allgemein akzeptiert!

Was genau heisst “berechenbar”? Zwei Sichten:

- Funktion/Programm: Kann ein Resultat berechnen
- Grammatik: Kann Ableitung eines Wortes einer Sprache finden.

# Vorschau verwandter Konzepte mit unterschiedl. Stärken

Wort ableiten

Wort akzeptieren

semi-entscheidbar

rekursiv aufzählbar

---

charakteristische Funktion, Berechenbarkeit

---

entscheidbar

Vollständigkeit



# Turing–Berechenbarkeit – Definition

Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ heisst Turing-**berechenbar**, falls es eine Turingmaschine $M$ gibt, so dass für $n_1, \ldots n_k, m \in \mathbb{N}$ gilt:

$$f(n_1, \ldots n_k) = m$$
$$\Longleftrightarrow$$
$$z_0\, bin(n_1)\# \ldots\ \#bin(n_k) \quad \vdash^* \quad \Box z_e\, bin(m) \Box$$

$z_0 \ldots$ ist die Anfangskonfiguration
$z_0$ ist der Anfangszustand (der TM)
# ist das Trennzeichen, $\Box$ das Leerzeichen
$z_e$ ein Endzustand $\in E$

# Turing–Maschine: “Undefiniert = nicht-abbrechend”

Vorherige Definition impliziert:

- Sobald die Maschine anhält, liegt ein Resultat vor:
  es wurde etwas “berechnet”.
- Wenn eine Funktion nicht berechenbar ist,
  darf keine Endkonfiguration erreicht werden,
  d.h. die dazugehörige TM gerät in eine Endlos-Schleife.
- Wenn eine TM nicht abbricht, dann ist die dazugehörige Funktion
  (auf dem gegebenen Input) nicht berechenbar/undefiniert.



# Turing-Maschine für eine unberechenbare Funktion

Fall: die total undefinierte Funktion $\Omega$

- Konstruiere aus $f(x, y) = 1$ (konstante Funktion) $\Omega$ wie folgt:
  $\Omega = \mu f$ (das kleinste $x$ wo $f(x, y) = 0$)
- Mögliche Implementierung: eine TM, die nie anhält
- TM für $\Omega$:
  Transitionsfunktion $\delta(z_0, *) = (z_0, a, R)$
  d.h. diese Maschine füllt das Band mit ‘$a$’s

# Entscheidbare Menge – Definition

Eine Menge $A \subseteq \Sigma^*$ heisst entscheidbar, falls die charakteristische Funktion $\chi_A : \Sigma^* \to \{0, 1\}$ berechenbar ist, wobei

$\chi_A(w) := 1$ falls $w \in A$
$\chi_A(w) := 0$ falls $w \notin A$



# Semi-Entscheidbare Menge – Definition

Eine Menge $A \subseteq \Sigma^*$ heisst **semi-**entscheidbar, falls die Funktion $\chi'_A : \Sigma^* \to \{0, 1\}$ berechenbar ist, wobei

$\chi'_A(w) := 1$ falls $w \in A$
$\chi'_A(w) := undefiniert$ falls $w \notin A$

Das heisst:
Aus “berechenbare Menge” folgt nicht “entscheidbare Menge” ($\chi'$ ist nämlich berechenbar!).

# (Semi-) Entscheidbar und Akzeptieren

Bezug zwischen Wort-Problem und charakteristische Funktion:

- Betrachte Sprachen, d.h. eine Menge von Wörten, $A$
- Das Wortproblem für eine Sprache ist entscheidbar, wenn die charakteristische Funktion berechenbar ist mit:

  $\chi_A(w) := 1$ falls $w \in A$
  $\chi_A(w) := 0$ falls $w \notin A$

Gibt uns das Konzept “Wort akzeptieren” schon “entscheidbar”?
**Nein!**
*Nur falls* wir eine Ableitung finden, wissen wir dass es eine Ableitung gibt. Aber solange keine gefunden ist, wissen wir i.A. nichts!



# Turing–Berechenbarkeit – “Wort in einer Sprache”

Betrachte $f : \Sigma^* \to \Sigma^*$

Für $x, y \in \Sigma^*$ gilt

$$f(x) = y$$
$$\Longleftrightarrow$$
$$z_0\, x \quad \vdash^* \quad \Box z_e y \Box$$

wobei $z_e$ ein Endzustand $\in E$

Wortableitung $f$:
setze $x = S$ (Startsymbol), $y = w$ (ein Wort)
z.B. Linksableitung etc

# Semi-entscheidbar $\approx$ “rekursiv-aufzählbar”

Grundidee:

- Ich kann beginnen, alle möglichen Resultatwerte aufzuzählen
- Wenn ich auf das gesuchte Resultat treffe, halte ich an.

Anwendung auf Wort-Problem:
– Gegeben: Wort $w$. Gesucht: “Gilt $w \in L$?”
– alle Wörter von $L$ generieren (zuerst der Länge 0, der Länge 1 etc)
– wir halten an, wenn $w$ gefunden wird

“Akzeptieren” $\approx$ semi-entscheidbar



# “Entscheiden” (volle Entscheidbarkeit)

Eine Sprache $L$ ist entscheidbar, wenn es eine TM gibt, die für ein beliebiges Wort $w$ entscheidet, ob $w \in L$ **oder ob** $w \notin L$.
(siehe auch Definition der charakteristischen Funktion)

- Andere Formulierung: falls eine Menge entscheidbar ist, gibt es eine TM, die in *jedem* Fall (beliebiges $w$) anhält und den Entscheid bekanntgibt.
- Entscheidbarkeit (einer Sprache oder Menge) **ist eine stärkere** d.h. einschränkendere Eigenschaft als Akzeptierbarkeit.