

- Ist P ein WHILE-Programm, x_i Variable, dann ist auch

WHILE $x \neq 0$ DO P END;

ein WHILE-Programm.

Semantik.

P wird solange ausgeführt, wie $x_i \neq 0$ gilt.

Man kommt in WHILE-Programmen ohne LOOP aus:

LOOP x DO P END;

wird simuliert durch

$y := x;$
WHILE $y \neq 0$ DO $y := y - 1; P$ END;

Definition.

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das gestartet mit $n_1, \dots, n_k$ in $x_1, \dots, x_k$ (0 sonst) mit dem Wert $f(n_1, \dots, n_k)$ in x_0 stoppt.

Sonst stoppt P nicht.

Satz.

TM können WHILE-Programme simulieren.

D.h. jede WHILE-berechenbare Funktion ist auch TM-berechenbar.

Beweis.

Man kann die Wertzuweisungen, Sequenzen und WHILE-Schleifen mit einer Mehrband TM simulieren.

(Wobei das i -te Band der i -ten Variablen (binär) entspricht.)

Man kann eine Mehrband TM mit einer 1-Band TM simulieren. ■

Beweis der Umkehrung:

Betrachte GOTO-Programme.

GOTO-Programme bestehen aus Folgen von markierten Anweisungen

$M_1 : A_1 ; M_2 : A_2 ; \dots ; M_k : A_k$

Als Anweisungen sind zugelassen:

Wertzuweisungen: $x_i := x_j \pm c$

unbedingter Sprung: GOTO M_i

bedingter Sprung: IF $x_i = c$ THEN GOTO M_i

Stopanweisung: HALT

Vereinbarung: Marken von nicht angesprungen Anweisungen werden weggelassen.

Die Semantik ist klar.

Definition.

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt *GOTO-berechenbar*, falls es ein GOTO-Programm P gibt, das gestartet mit $n_1, \dots, n_k$ in $x_1, \dots, x_k$ (0 sonst) mit dem Wert $f(n_1, \dots, n_k)$ in x_0 stoppt. Sonst stoppt P nicht.

Satz.

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

Beweis.

WHILE $x_i \neq 0$ DO P END

kann simuliert werden durch

M_1 : IF $x_i = 0$ THEN GOTO M_2 ;
 P ;
 GOTO M_1 ;
 M_2 : ...

■

Korollar

Jede WHILE-berechenbare Funktion ist GOTO-berechenbar.

Satz.

Jedes GOTO-Programm kann durch ein WHILE-Programm (mit nur einer WHILE-Schleife) simuliert werden.

Beweis.

Gegeben sei ein GOTO-Programm

$M_1 : A_1 ; M_2 : A_2 ; \dots ; M_k : A_k$

wird simuliert durch folgendes WHILE-Programm mit nur einer WHILE-Schleife:

count:=1;
 WHILE count $\neq$ 0 DO
 IF count = 1 THEN A'_1 END;
 IF count = 2 THEN A'_2 END;

$\vdots$
 IF $count = k$ THEN A'_k END;
 END

wobei

$$A'_i = \begin{cases} x_j := x_i \pm c; count := count + 1 & \text{falls } A_i = x_j \pm c \\ count := n & \text{falls } A_i = \text{GOTO } M_n \\ \text{IF } x_j = c \text{ THEN } count := n \text{ ELSE} & \\ \quad count := count + 1 \text{ END} & \text{falls } A_i = \text{THEN GOTO } M_n \\ count := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

■

Korollar

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist GOTO-berechenbar, falls f WHILE-berechenbar ist.

Satz. (Kleenesche Normalformsatz für WHILE-Programme)

Jede WHILE-berechenbare Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden.

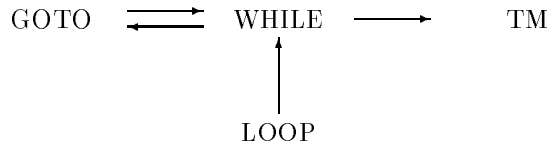
Beweis.

Sei P ein beliebiges WHILE-Programm für f .

P wird simuliert durch das GOTO-Programm P' .

P' wird simuliert durch das WHILE-Programm P'' mit nur einer WHILE-Schleife. ■

Bisher ist gezeigt worden:



Satz.

GOTO-Programme können TM simulieren.

Beweis.

Sei $M = (Q, \Sigma, \Gamma, \delta, q_1, \square, F)$ eine TM die f berechnet.

Das simulierende GOTO-Programm hat folgende Gestalt:

$M_1 : A_1 ; M_2 : A_2 ; M_3 : A_3$

wobei:

- P_1 : transformiert die gegebenen Anfangswerte in Binärdarstellung und erzeugt eine Darstellung der Starkonfiguration von M in die Variablen x, y, z .
- P_2 : simuliert die Rechnung von M Schritt-für-Schritt durch Veränderung der Variablenwerte x, y, z .
- P_3 : erzeugt aus der kodierten Form der Endkonfiguration in x, y, z die eigentliche Ausgabe in der Ausgabevariable x_0 .

Bemerkung

P_1, P_3 hängen nicht von M ab, lediglich von P_2 .

Kodierung:

Sei

$$\begin{aligned} Q &= \{q_1, \dots, q_k\} \\ \Gamma &= \{a_1, \dots, a_m\}, \quad b > \#\Gamma \end{aligned}$$

Kodierung der TM –Konfiguration:

$$a_{i_1} \dots a_{i_p} q_l a_{j_1} \dots a_{j_q}$$

durch die folgenden Werte von x, y, z :

$$\begin{aligned} x &= (i_1 \dots i_p)_b \\ y &= (j_q \dots j_1)_b \\ z &= l \end{aligned}$$

GOTO-Programmstück $M_2 : P_2$:

$$\begin{aligned} M_2: & a := y \text{ MOD } b; \\ & \text{IF } (z = 1) \text{ AND } (a = 1) \text{ THEN GOTO } M_{11} \\ & \text{IF } (z = 1) \text{ AND } (a = 2) \text{ THEN GOTO } M_{12} \\ & \vdots \\ & \text{IF } (z = k) \text{ AND } (a = m) \text{ THEN GOTO } M_{km} \\ M_{11}: & \otimes \\ & \text{GOTO } M_2 \\ M_{12}: & \otimes \\ & \text{GOTO } M_2 \\ & \vdots \\ M_{km}: & \otimes \\ & \text{GOTO } M_2 \end{aligned}$$

wobei $\otimes$ bedeutet:

Man nimmt das Programmstück das mit M_{ij} markiert ist

$$\text{Sei } \delta(q_i, a_j) = (q_i, a_j, L)$$

und simuliert den Übergang durch:

$$\begin{aligned} z &:= i'; \\ y &:= y \text{ DIV } b; \\ y &:= b * y + j'; \\ y &:= b * y + (x \text{ MOD } b); \\ x &:= x \text{ DIV } b; \end{aligned}$$

ist q_i Endzustand, kann man

für $\otimes$ GOTO M_3 setzen.

Der Rest ist klar. ■

Korollar

Eine Funktion ist

TM -berechenbar $\Leftrightarrow$ GOTO-berechenbar $\Leftrightarrow$ WHILE-berechenbar

1.6 Die Ackermann-Funktion

Ackermann gab 1925 eine Funktion an, die intuitiv berechenbar (also WHILE-berechenbar), jedoch nicht primitiv rekursiv (also LOOP-berechenbar) ist.

$$\begin{aligned} \text{ack}(0, y) &= y + 1 \\ \text{ack}(x, 0) &= \text{ack}(x - 1, 1) \\ \text{ack}(x, y) &= \text{ack}(x - 1, \text{ack}(x, y - 1)) \end{aligned}$$

z.B.:

$$\text{ack}(x, y) = \underbrace{\text{ack}(x - 1, \text{ack}(x - 1, \dots \text{ack}(x - 1, 1) \dots))}_{y\text{-mal}}$$

Betrachte eine ähnlich definierte Funktion a :

$$\begin{aligned} a(0, y) &= a(x, 0) = 1 \\ a(1, y) &= 3y + 1 \\ a(x, y) &= \underbrace{a(x - 1, a(x - 1, \dots a(x - 1, y) \dots))}_{y\text{-mal}} \end{aligned}$$

- a ist total definiert (vollständige Induktion)

MODULA-Prozedur für die berechnung von a :

```

PROCEDURE  $a(x, y: \text{CARDINAL}): \text{CARDINAL}$ ;
VAR  $i, s : \text{CARDINAL}$ ;
BEGIN
  IF  $(x = 0)$  OR  $(y = 0)$  THEN RETURN 1
  ELSIF  $x = 1$  THEN RETURN  $3 * y + 1$ 
  ELSE  $s := y$ 
        FOR  $i := 1$  TO  $y$  DO
           $s = a(x - 1, s)$ 
        END;
  RETURN  $s$ 
END;
END  $a$ ;

```

a ist nicht LOOP-berechenbar, denn

Sei P ein LOOP-Programm.
 Ordne P die Funktion $f_P : \mathbb{N} \rightarrow \mathbb{N}$ zu:
 Seien $x_0, \dots, x_n$ alle in P vorkommenden Variable.

n_i sei der Startwert von x_i .

n'_i sei der Endwert von x_i nach Ablauf von P .

$$f_P(n) := \max\{\sum_{i=0}^k n'_i \mid \sum_{i=0}^k n_i \leq n\}$$

(größtmögliche Summe aller Variablenendwerte, falls P mit dem Startwert der Summe $\leq n$ gestartet wird.)

Lemma

Für jedes LOOP-Programm P gibt es eine Konstante k mit $f_P(n) < a(k, n)$ für alle $n \geq k$.

Beweis.

Induktion über den Aufbau von P

- Habe P die Form $x_i := x_j \pm c$:

$$\Rightarrow f_P(n) \leq n + n + c = 2n + c$$

$$\Rightarrow f_P(n) < 3n + 1 = a(1, n) \leq a(k, n) \text{ für alle } k \geq 1, n \geq c$$

Wähle $k := c + 1$.

- Habe P die Form $P_1; P_2$:

Nach Voraussetzung gibt es k_1, k_2 mit

$$f_{P_1} < a(k_1, n)$$

$$f_{P_2} < a(k_2, n) \text{ für allen } n \geq \max\{k_1, k_2\}$$

Setze $k_3 := \max\{k_1, k_2\}$

Es gilt:

$$\begin{aligned} f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\ &\leq f_{P_2}(a(k_1, n)) \\ &< a(k_2, a(k_1, n)) \\ &\leq a(k_3, a(k_3, n)) \\ &\leq \underbrace{a(k_3, a(k_3, \dots a(k_3, n) \dots))}_{n\text{-mal}} \text{ falls } n \geq 2 \\ &= a(k_3 + 1, n) \end{aligned}$$

Wähle $k := \max\{k_3, 2\}$

- Habe P die Form LOOP x_i DO P' END:

Nach Induktionvoraussetzung gibt es ein k' mit

$$f'_P < a(k', n) \text{ für alle } n \geq k'$$

Es gilt:

$$\begin{aligned} f_P(n) &= \underbrace{f'_P(f'_P(\dots f'_P(n) \dots))}_{n_i\text{-mal}} \\ &< \underbrace{a(k', a((k', \dots a(k', n) \dots))}_{n_i\text{-mal}} \\ &\leq \underbrace{a(k', a(k', \dots a(k', n) \dots))}_{n\text{-mal}} \\ &= a(k' + 1, n) \end{aligned}$$

wähle $k := k' + 1$

■

Satz.

Die Ackermann-Funktion a ist nicht LOOP-berechenbar.

Beweis.

Annahme: a ist LOOP-berechenbar

$\Rightarrow g(n) := a(n, n)$ ist LOOP-berechenbar

Sei P ein LOOP-Programm für g

$\Rightarrow g(n) \leq f_P(n)$

wähle k in P mit: $f_P(n) < a(k, n)$

Für $n = k$ gilt:

$$g(k) \leq f_P(k) < a(k, k) = g(k) \quad (\text{Widerspruch})$$

■

- a ist auch formal berechenbar.

WHILE-Programm für a :

Zunächst Angabe eines Programmes zur Berechnung von a , daß mit den Stackoperationen PUSH und POP operiert:

```

INPUT( $x, y$ );
INIT( $stack$ );
PUSH( $x, stack$ );
PUSH( $y, stack$ );
WHILE size( $stack$ )  $\neq$  1 DO
     $y :=$ POP( $stack$ );
     $x :=$ POP( $stack$ );
    IF ( $x = 0$ ) OR ( $y = 0$ ) THEN PUSH(1,  $stack$ );

```

```

    ELSIF  $x = 1$  THEN PUSH( $3 * y + 1, stack$ );
    ELSE LOOP  $y$  DO PUSH( $k - 1, stack$ ) END;
    PUSH( $y, stack$ )
END;
 $result := POP(stack)$ ;
OUTPUT( $result$ );

```

Man kann die einzelnen Stackoperationen durch WHILE-Programme simulieren:

Betrachte: $c : \mathbb{N} \rightarrow \mathbb{N}$ mit $c(x, y) := 2^{x+y} + x$
 c ist WHILE berechenbar und injektiv;

Beispiel: Werte für c :

	0	1	2	3	4
0	1	3	6	11	20
1	2	5	10	19	36
2	4	9	18	35	68
3	8	17	34	67	132
4	16	33	66	131	260

Man benutzt c zur Kodierung von Paaren.

Man kann aus $c(x, y) = n$ das Paar (x, y) zurückgewinnen:

Sei k max. mit $2^k < n$
 $\Rightarrow x := n - 2^k$
 $y := k - x$

Definiere:

$$c_1(n) := \begin{cases} x & \text{falls } n \in c(\mathbb{N}^2) \\ 0 & \text{sonst} \end{cases}; c_2(n) := \begin{cases} y & \text{falls } n \in c(\mathbb{N}^2) \\ 0 & \text{sonst} \end{cases}$$

c_1, c_2 sind WHILE- (sogar LOOP-) berechenbar.

Nun zur Simulation des Stacks durch ein WHILE-Programm:

Sei $(n_1, \dots, n_k)$ der Inhalt des Stacks,
 n_1 das Head-Element.

Kodierung von $(n_1, \dots, n_k)$ mit Hilfe von c :
 $n := c(n_1, c(n_2, \dots, c(n_k, 0) \dots))$

INIT(<i>stack</i>)	wird simuliert durch	$n := 0$
PUSH($a, stack$)	wird simuliert durch	$n := c(a, n)$
POP(<i>stack</i>)	wird simuliert durch	$result := c_1(n)$
		$n := c_2(n)$
		RETURN $result$
size($stack \neq 1$)	wird simuliert durch	$c_2(n) \neq 0$

Lemma

Die Ackermannfunktion ist WHILE-berechenbar.