

### 3. LOOP-, WHILE- und GOTO-Berechenbarkeit

#### 3.1 LOOP-Programme

Komponenten:	Variablen:	$x_0, x_1, x_2, \dots, y, z, \dots$
	Konstanten:	0, 1, 2, ...
	Trennsymbole:	;        :=
	Operationszeichen:	+, -
	Schlüsselwörter:	LOOP, DO, END

Def.: Die Menge der LOOP-Programme ist induktiv wie folgt definiert:

1. jede Wertzuweisung der Form  $x_i := x_j + c$  oder  $x_i := x_j - c$  ist ein LOOP-Programm, wobei  $c$  Konstante ist;
2. falls  $P_1$  und  $P_2$  LOOP-Programme sind, so ist  $P_1; P_2$  ein LOOP-Programm;
3. falls  $P$  ein LOOP-Programm ist, so ist  $\text{LOOP } x_i \text{ DO } P \text{ END}$  ein LOOP-Programm.

Semantik:

- LOOP-Programme berechnen Funktionen über  $\mathbb{N}$
- Eingabewerte  $n_1, \dots, n_k$  bei Beginn der Programmausführung an Variablen  $x_1, \dots, x_k$  gebunden.
- Alle anderen Variablen haben Anfangswert 0.
- Ausgabewert ist Wert von  $x_0$  nach Beendigung des Programms.
- Wertzuweisung: wie üblich, "-" ist modifizierte Subtraktion:  $x_j - c = 0$ , falls  $c > x_j$ .
- $P_1; P_2$  Hintereinanderausführung von  $P_1$  und  $P_2$ .
- $\text{LOOP } x_i \text{ DO } P \text{ END}$ :  $P$  wird  $x_i$  mal ausgeführt. Zu beachten: der Wert von  $x_i$  wird vor (!) der ersten Ausführung von  $P$  ermittelt und die Schleife entsprechend oft ausgeführt. Zuweisungen an  $x_i$  in  $P$  haben keine Auswirkung auf die Zahl der Schleifendurchläufe!

Def.: Eine Funktion  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  heißt LOOP-berechenbar, falls es ein LOOP-Programm gibt, das  $f$  berechnet, d.h.  $P$  gestartet mit der Variablenbelegung  $n_1, \dots, n_k$  der Variablen  $x_1, \dots, x_k$  (alle anderen Variablen initialisiert mit 0) stoppt mit  $f(n_1, \dots, n_k)$  als Wert der Variable  $x_0$ .

Anmerkung:

offensichtlich sind nur totale Funktionen berechenbar (aber nicht alle totalen  $\rightarrow$  später).

Abkürzungen:	$x_i := x_j$	statt $x_i := x_j + 0$
	$x_i := c$	statt $x_i := x_j + 0$ , wobei $x_j$ eine Variable mit Wert 0 ist.

Modellierung von Fallunterscheidungen:

IF  $x = 0$  THEN  $P$  END                       $y := 1$ ;                      (y Hilfsvariable)  
  LOOP  $x$  DO  $y := 0$  END;  
  LOOP  $y$  DO  $P$  END

IF  $x = 0$  THEN  $P_1$  ELSE  $P_2$  END         $y := 1$ ;  $z := 0$ ;                      (y,z Hilfsvariablen)  
  LOOP  $x$  DO  $y := 0$ ;  $z := 1$  END;  
  LOOP  $y$  DO  $P_1$  END;  
  LOOP  $z$  DO  $P_2$  END

Weitere Beispiele:

Addition:

```
x0:= x1;  
LOOP x2 DO x0 := x0 + 1 END
```

Multiplikation:

```
x0:= 0;  
LOOP x2 DO x0 := x0 + x1 END
```

ausführlich:

```
x0:= 0;  
LOOP x2 DO LOOP x1 DO x0 := x0 + 1 END END
```

$f(n_1, n_2) = 1$  falls  $n_1 = n_2$ , = sonst:

```
x0 := 1;  
x3 := x1;  
LOOP x2 DO x3 := x3 - 1 END;  
LOOP x3 DO x0 := 0 END;  
x3 := x2;  
LOOP x1 DO x3 := x3 - 1 END;  
LOOP x3 DO x0 := 0 END
```

### 3.2 WHILE-Programme

Def.: Die Menge der WHILE-Programme ist induktiv wie folgt definiert:

1. jede Wertzuweisung der Form  $x_i := x_j + c$  oder  $x_i := x_j - c$  ist ein WHILE-Programm, wobei  $c$  Konstante ist;
2. falls  $P_1$  und  $P_2$  WHILE-Programme sind, so ist  $P_1; P_2$  ein WHILE-Programm;
3. falls  $P$  ein WHILE-Programm ist, so ist  $\text{LOOP } x_i \text{ DO } P \text{ END}$  ein WHILE-Programm;
4. falls  $P$  ein WHILE-Programm ist, so ist  $\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$  ein WHILE-Programm.

Semantik von WHILE-Schleife:  $P$  wird so lange ausgeführt, bis  $x_i$  den Wert 0 hat.

Anmerkung: LOOP-Konstrukt nicht mehr unbedingt nötig, da

```
LOOP x_i DO P END
```

simuliert werden kann durch ( $y$  ist Variable, die sonst nicht im Programm verwendet wird):

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

Def.: Eine Funktion  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  heißt WHILE-berechenbar, falls es ein WHILE-Programm gibt, das  $f$  berechnet, d.h.  $P$  gestartet mit der Variablenbelegung  $n_1, \dots, n_k$  der Variablen  $x_1, \dots,$

$x_k$  (alle anderen Variablen initialisiert mit 0) stoppt mit  $f(n_1, \dots, n_k)$  als Wert der Variable  $x_0$ , falls  $f(n_1, \dots, n_k)$  definiert ist. Ansonsten terminiert das Programm nicht.

Da jedes LOOP-Programm auch ein WHILE-Programm ist, ist klar, dass alle LOOP-berechenbaren Funktionen auch WHILE-berechenbar sind. Es ist auch offensichtlich, dass WHILE-Programme mehr Funktionen berechnen können: WHILE-Programme können endlos laufen und damit echt partielle Funktionen berechnen. Z.B. berechnet das Programm

```
xi := 1;  
WHILE xi ≠ 0 DO xi := 1 END
```

die für jede Eingabe undefinierte Funktion, die nicht LOOP-berechenbar ist.

Wir werden später zeigen, dass es auch totale berechenbare Funktionen gibt, die nicht LOOP-berechenbar sind.

Simulation von WHILE-Programmen mit Turing-Maschinen:

Wertzuzuweisung, Hintereinanderschaltung von Programmen sowie WHILE-Schleifen durch Mehrband-TMs simulierbar (jeder Variable des WHILE-Programms entspricht ein Band). Außerdem Mehrband-TMs durch 1-Band TM simulierbar. Deshalb gilt:

Satz: Jede WHILE-berechenbare Funktion ist Turing-berechenbar.

Umkehrung wird nach Einführung von GOTO-Programmen bewiesen.

### 3.3 GOTO-Programme

Sequenzen von Anweisungen  $A_i$ , die Marke  $M_i$  besitzen:

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

Zulässige Anweisungen:

```
Wertzuzuweisungen:  xi := xj + c oder xi := xj - c  
unbedingter Sprung:  GOTO Mi  
bedingter Sprung:   IF xi = c THEN GOTO Mj  
Stopanweisung:     HALT
```

Bemerkungen: Marken, die nicht angesprungen werden können, dürfen entfallen. Die letzte Anweisung  $A_k$  ist entweder ein unbedingter Sprung oder HALT.

Semantik:

Wertzuzuweisung wie bisher,  
GOTO legt nächste auszuführende Programmanweisung fest,  
Stopanweisung bedeutet Terminieren,  
Ausgabe ist Wert von  $x_0$ .

Beispiel: ganzzahlige Division (ohne Rest, undefiniert falls  $x_2 = 0$ ):

```

M0:  x3 := x2;
M1:  IF x3 = 0 THEN GOTO M2;
      IF x1 = 0 THEN GOTO M3
      x1 := x1 - 1;
      x3 := x3 - 1;
      GOTO M1;
M2:  x0 := x0 + 1;
      GOTO M0;
M3:  HALT

```

Def. von GOTO-Berechenbarkeit analog zu WHILE-Berechenbarkeit.

Jedes WHILE-Programm kann durch GOTO-Programm simuliert werden:

```

WHILE xi ≠ 0 DO P END

```

wird simuliert durch:

```

M1:  IF xi = 0 THEN GOTO M2;
      P;
      GOTO M1;
M2:  ...

```

wobei M<sub>1</sub>, M<sub>2</sub> geeignete Marken sind.

Satz: Jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

Umgekehrte Richtung:

Gegeben GOTO-Programm M<sub>1</sub>:A<sub>1</sub>; M<sub>2</sub>:A<sub>2</sub>; ... M<sub>k</sub>:A<sub>k</sub>

Das Programm wird simuliert durch folgendes WHILE-Programm:

```

count := 1;
WHILE count ≠ 0 DO
  IF count = 1 THEN A1' END;
  IF count = 2 THEN A2' END;
  ...
  IF count = k THEN Ak' END;
END

```

Hierbei gilt:

A <sub>i</sub> ' =	x <sub>i</sub> := x <sub>j</sub> ± c; count = count + 1	falls A <sub>i</sub> =	x <sub>i</sub> := x <sub>j</sub> ± c
	count := n	falls A <sub>i</sub> =	GOTO M <sub>n</sub>
	IF x <sub>j</sub> = c THEN count := n		
	ELSE count := count + 1 END	falls A <sub>i</sub> =	IF x <sub>j</sub> = c THEN GOTO M <sub>n</sub>
	count := 0	falls A <sub>i</sub> =	HALT

Anmerkung: IF ... THEN ... ELSE modellierbar durch LOOP, wie bereits früher gezeigt:

```

IF x = 0 THEN P1 ELSE P2 END   y:= 1; z:= 0;           (y,z Hilfsvariablen)
                                LOOP x DO y := 0; z:= 1 END;
                                LOOP y DO P1 END;
                                LOOP z DO P2 END

```

$x = c$  testet man durch  $x - c = 0$  und  $c - x = 0$ .  
 (oberer Spezialfall sogar ganz ohne Schleifen modellierbar -> Übungen)

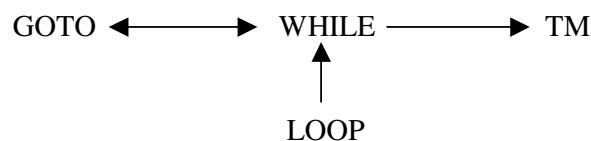
Satz: Jedes GOTO Programm kann durch ein WHILE-Programm simuliert werden. Damit ist jede GOTO-berechenbare Funktion WHILE-berechenbar.

Anmerkung: Obiges Programm hat eine einzige WHILE-Schleife:

Satz: (Kleenesche Normalform für WHILE-Programme)  
 Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden.

Bew.: P überführt in GOTO-Programm P' überführt in WHILE-Programm P''.

Bisher ergibt sich folgendes Bild ( $X \longrightarrow Y$  heißt X-berechenbar impliziert Y-berechenbar bzw. Y-Programm kann X-Programm simulieren):



Es fehlt noch  $\text{TM} \rightarrow \text{GOTO}$ , dann ist die Äquivalenz aller (bis auf LOOP) gezeigt.

Sei  $M = (Z, \Sigma, \Gamma, \delta, z_1, \square, E)$  TM zur Berechnung von  $f$ .  
 M wird simuliert durch ein GOTO-Programm

$M_1:P_1; M_2:P_2; M_3:P_3$

- P<sub>1</sub>: transformiert Anfangswerte der Variablen in Binärdarstellung  
 erzeugt Darstellung der Startkonfiguration von M  
 Codierung in 3 Variablen x,y,z
- P<sub>2</sub>: Schritt-für-Schritt-Simulation der Rechnung von M
- P<sub>3</sub>: erzeugt aus Zielkonfiguration Wert der Ausgabevariable  $x_0$ .

Seien  $Z = \{z_1, \dots, z_k\}$ ,  $\Gamma = \{a_1, \dots, a_m\}$ ,  $b > |\Gamma|$  (größer nötig, sonst evtl. führende Nullen)

TM-Konfiguration  $a_{i_1} \dots a_{i_p} z_r a_{j_1} \dots a_{j_q}$  wird durch Variablenwerte repräsentiert: Indizes der Symbole links von  $z_r$  werden als Zahl zur Basis  $b$  gelesen, entsprechend Indizes der Symbole rechts (in umgekehrter Reihenfolge):

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

$(i_1 \dots i_p)_b$  ist Wert der zur Basis  $b$  dargestellten Zahl  $i_1 \dots i_p$ :

$$x = \sum_{t=1 \dots p} i_t \times b^{p-t}$$

Analog für  $y$ , aber umgekehrte Reihenfolge.

Beispiel:  $\Gamma = \{a_1, \dots, a_3\}$ ,  $b = 4$

Konfiguration  $a_1 a_1 z_2 a_3 a_2$  wird repräsentiert durch

$$x = 1 \times 4^1 + 1 \times 4^0 = 5$$

$$y = 3 + 8 = 11$$

$$z = 2$$

$M_2:P_2$  hat folgende Struktur:

```
M2:  a := y MOD b;  
      IF (z = 1) AND (a = 1) THEN GOTO M11;  
      IF (z = 1) AND (a = 2) THEN GOTO M12;  
      ...  
      IF (z = k) AND (a = m) THEN GOTO Mkm;  
M11: P11;  
      GOTO M2;  
M12: P12;  
      GOTO M2;  
      ...  
Mkm: Pkm;  
      GOTO M2;
```

Wie sind Programme  $P_{ij}$  aufgebaut? Modellieren Konfigurationsübergänge der TM.

Beispiel: es gelte  $\delta(z_i, a_j) = (z_i', a_j', L)$

dann ist  $P_{ij}$ :

```
z := i';           % neuer Zustand  
y := y DIV b;     % erstes Symbol des rechten Wortes weg  
y := b × y + j';  % aj vor rechtes Wort fügen  
y := b × y + (x MOD b); % letztes Symbol linkes Wort rechts anfügen  
x := x DIV b;     % letztes Symbol linkes Wort entfernen
```

andere Fälle ähnlich. Falls  $z_i \in E$  so ist  $P_{ij}$ : GOTO  $M_3$ .

Satz: GOTO-Programme können Turingmaschinen simulieren. Damit ist jede Turing-berechenbare Funktion auch GOTO-berechenbar.

#### 4. Primitiv rekursive und $\mu$ -rekursive Funktionen

**Vorbemerkung: dieser Abschnitt enthält Teile, die in der Vorlesung nicht behandelt wurden. Sie wurden der Vollständigkeit halber im Skript belassen.**

Rekursion einer der ersten Ansätze, Berechenbarkeit zu präzisieren (parallel zu TM) basiert nicht auf Maschinenmodell (TM) oder Zustandsmodell (Bindung Wert an Variable, WHILE, GOTO).

Grundidee: Angabe Basisfunktionen + Prinzipien, um aus Funktionen neue zu generieren. Rekursion bedeutet: Zurückführen eines Problems auf ein kleineres: hier Berechnung für  $n+1$  auf die Berechnung für  $n$ .

Def.: Die Klasse der primitiv rekursiven Funktionen über  $\mathbb{N}$  ist induktiv so definiert:

1. Alle konstanten Funktionen  $c_k^j$  mit  $c_k^j(n_1, \dots, n_j) = k$  für beliebige  $n_1, \dots, n_j$  sind primitiv rekursiv.
2. Alle Projektionen  $\pi_i^j$  mit  $\pi_i^j(n_1, \dots, n_j) = n_i$  ( $i \leq j$ ) sind primitiv rekursiv.
3. Die Nachfolgerfunktion  $s$  mit  $s(n) = n + 1$  ist primitiv rekursiv.
4. Wenn  $g: \mathbb{N}^r \rightarrow \mathbb{N}$  und  $h_1: \mathbb{N}^k \rightarrow \mathbb{N}, \dots, h_r: \mathbb{N}^k \rightarrow \mathbb{N}$  primitiv rekursiv sind, so ist auch die  $k$ -stellige Funktion  $f = \text{SUB}(g, h_1, \dots, h_r)$  mit
 
$$f(n_1, \dots, n_k) = g(h_1(n_1, \dots, n_k), \dots, h_r(n_1, \dots, n_k))$$
 primitiv rekursiv.
5. Wenn  $g: \mathbb{N}^{k-1} \rightarrow \mathbb{N}$  und  $h: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  primitiv rekursive Funktionen sind, dann auch die  $k$ -stellige Funktion  $f = \text{PR}(g, h)$ : mit:

$$\begin{aligned} f(0, n_2, \dots, n_k) &= g(n_2, \dots, n_k) \\ f(n+1, n_2, \dots, n_k) &= h(f(n, n_2, \dots, n_k), n, n_2, \dots, n_k). \end{aligned}$$

Beispiele:

Ansatz: über Rekursion:

$$\begin{aligned} \text{add}(0, x) &= x \\ \text{add}(n+1, x) &= s(\text{add}(n, x)) \end{aligned}$$

nach Definition:

$$\begin{aligned} &= g(x) && \text{also } g = \pi_1^1 \\ &= h(\text{add}(n, x), n, x) && \text{also } h = \text{SUB}(s, \pi_1^3) \\ \text{ergibt: } \text{add} &= \text{PR}(\pi_1^1, \text{SUB}(s, \pi_1^3)) \end{aligned}$$

$$\begin{aligned} \text{mult}(0, x) &= 0 \\ \text{mult}(n+1, x) &= \text{add}(\text{mult}(n, x), x) \end{aligned}$$

Anmerkung: da beliebige Projektionen primitiv rekursiv sind, kommt es auf die Reihenfolge der Argumente von  $g$  und  $h$  nicht an. Es müssen auch nicht alle Argumente  $n_2, \dots, n_k$  von  $f$  und  $g$  benutzt werden. Aus demselben Grund muss Rekursion in  $f$  nicht notwendiger Weise über das erste Argument laufen.

$$\begin{aligned} \text{mult}(0, x) &= 0 \\ \text{mult}(n+1, x) &= \text{add}(\text{mult}(n, x), x) \end{aligned}$$

$$\begin{aligned} \text{sub}(x, 0) &= x \\ \text{sub}(x, n+1) &= \text{sub}_1(\text{sub}(x, n)) \end{aligned}$$

$$\begin{aligned} \text{sub}_1(0) &= 0 \\ \text{sub}_1(n+1) &= n \end{aligned}$$

Wir können auch Prädikate definieren (jeweils 1 "wahr", 0 "falsch"):

$$\begin{aligned} =_0(0) &= 1 \\ =_0(n+1) &= 0 \end{aligned}$$

$$\begin{aligned} \neq_0(0) &= 0 \\ \neq_0(n+1) &= 1 \end{aligned}$$

$$\begin{aligned} >(x,y) &= \neq_0(\text{sub}(x,y)) \\ =(x,y) &= \text{mult}(\text{sub}(1, >(x,y)), \text{sub}(1, >(y,x))) \end{aligned}$$

$$\text{div}(x,y) = \text{div}_3(x,y,0)$$

$$\begin{aligned} \text{div}_3(0,y,z) &= =(y,z) \\ \text{div}_3(n+1,y,z) &= >(y,z) * \text{div}_3(n,y,s(z)) + =(y,z) * s(\text{div}_3(n,y,1)) \end{aligned}$$

Beispiel:

$$\begin{aligned} \text{div}(5,2) &= \text{div}_3(5,2,0) = \text{div}_3(4,2,1) = \text{div}_3(3,2,2) \\ &= 1 + \text{div}_3(2,2,1) = 1 + \text{div}_3(1,2,2) = 1 + 1 + \text{div}_3(0,2,1) = 2 \end{aligned}$$

$$\begin{aligned} \text{div}(4,2) &= \text{div}_3(4,2,0) = \text{div}_3(3,2,1) = \text{div}_3(2,2,2) \\ &= 1 + \text{div}_3(1,2,1) = 1 + \text{div}_3(0,2,2) = 1 + 1 = 2 \end{aligned}$$

$$\text{mod}(x,y) = \text{mod}_3(x,y,0)$$

$$\begin{aligned} \text{mod}_3(0,y,z) &= \neq(y,z) * z \\ \text{mod}_3(n+1,y,z) &= >(y,z) * \text{mod}_3(n,y,s(z)) + =(y,z) * \text{mod}_3(n,y,1) \end{aligned}$$

Beispiel:

$$\begin{aligned} \text{mod}(5,2) &= \text{mod}_3(5,2,0) = \text{mod}_3(4,2,1) = \text{mod}_3(3,2,2) \\ &= \text{mod}_3(2,2,1) = \text{mod}_3(1,2,2) = \text{mod}_3(0,2,1) = 1 \end{aligned}$$

$$\text{sum}(n) = \sum_{i \leq n} i = n(n+1)/2$$

$$\begin{aligned} \text{sum}(0) &= 0 \\ \text{sum}(n+1) &= \text{sum}(n) + n + 1 \end{aligned}$$

### Exkurs: Codierung von Paaren/Tupel natürlicher Zahlen als natürliche Zahl

Zur Vorbereitung des Beweises primitiv rekursiv  $\Leftrightarrow$  LOOP-berechenbar zeigen wir, dass eine beliebige Anzahl natürlicher Zahlen eindeutig als natürliche Zahl codiert und wieder decodiert werden kann, und dass die dazu nötigen Funktionen primitiv rekursiv sind.

Betrachte die Funktion  $c(x,y) = \text{sum}(x + y) + x$

y\x	0	1	2	3	3
0	0	2	5	9	14
1	1	4	8	13	
2	3	7	12		
3	6	11			
4	10				

Beispiel:  $c(2,1) = 1+2+3+2 = 8$

Bijektion zwischen  $\mathbb{N}^2$  und  $\mathbb{N}$ . Offensichtlich primitiv rekursiv/LOOP-berechenbar:

Auch die Umkehrfunktionen für  $c$  sind LOOP-berechenbar:

$e_c$  liefert erstes Argument von  $c(x,y)$  zurück:  $e_c(c(x,y)) = x$   
 $f_c$  zweites Argument:  $f_c(c(x,y)) = y$

LOOP Programm:

```

e_c :   x_m := 1 ;
        LOOP x_1 do
            IF x_1 ≥ x_m THEN
                x_1 := x_1 - x_m;
                x_m := x_m + 1;
            ELSE x_0 := x_1 END;

```

[für  $f_c$ : Anweisung in letzter Zeile  $x_0 := (x_m - 1) - x_1$ ; außerdem muss Schleife mindestens 2 mal durchlaufen werden, damit der Fall  $n = 1$  richtig behandelt wird]

$x_m$  wird in jedem Schritt erhöht (1, 2, 3, etc.) und von der Eingabe abgezogen, solange der verbleibende Wert noch groß genug ist (nach  $k$  Schleifendurchläufen, in denen THEN ausgeführt wird, ist  $x_1 = \text{Eingabe} - \text{Summe } 1 \text{ bis } k$ ); ist der verbleibende Wert zu klein, so wird nur noch ELSE ausgeführt und nach der ersten Ausführung von ELSE nichts mehr verändert. Gesuchte Werte können dann aus dem „Rest“ berechnet werden.

Wir zeigen gleich, dass diese Funktionen auch primitiv rekursiv sind.

Verallgemeinerung auf  $k+1$ -Tupel natürlicher Zahlen:

$$\langle n_0, n_1, \dots, n_k \rangle = c(n_0, c(n_1, \dots, c(n_k, 0) \dots))$$

Mit  $e$  und  $f$  kann man daraus auch die  $k+1$  Umkehrfunktionen der  $k+1$ -stelligen Codierfunktion  $\langle \rangle$  bekommen:

$$\begin{aligned}
 d_0(n) &= e_c(n) \\
 d_1(n) &= e_c(f_c(n)) \\
 &\dots \\
 d_k(n) &= e_c(f_c(f_c(\dots f_c(n) \dots))) \quad k\text{-mal } f_c
 \end{aligned}$$

Diese Funktionen sind primitiv rekursiv, falls  $e$  und  $f$  es sind. Letzteres zeigen wir jetzt.

Vorbemerkung:

gegeben einstelliges Prädikat  $P(x)$  (Funktion, die  $x$  den Wert 0 oder 1 zuordnet)

der beschränkte max-Operator  $q_P(n)$  liefert das größte  $x \leq n$ , für das  $P(x) = 1$ .

$$\max\{x \leq n \mid P(x) = 1\}$$

Falls es kein solches  $x$  gibt ist der Wert von  $q_P(n)$  0.

$$\begin{aligned} q_P(0) &= 0 \\ q_P(n+1) &= n+1 \quad \text{falls } P(n+1) \\ &= q_P(n) \quad \text{sonst} \\ &= P(n+1) * (n+1) + (1 - P(n+1)) * q_P(n) \end{aligned}$$

$q_P$  ist primitiv rekursiv, falls  $P$  primitiv rekursiv.

Ähnlich können wir ein Prädikat  $Q_P(n)$  definieren, das Wert 1 hat, wenn es  $x \leq n$  gibt, für das  $P(x) = 1$ :  $Q_P(n) = 1$  gdw.  $\exists y \leq n : P(y) = 1$ .

$$\begin{aligned} Q_P(0) &= P(0) \\ Q_P(n+1) &= P(n+1) + Q_P(n) - P(n+1) * Q_P(n) \quad \% \text{ wenn beide gelten 1 abziehen} \end{aligned}$$

$Q_P$  ist primitiv rekursiv, falls  $P$  primitiv rekursiv.

Da  $x \leq c(x,y)$  und  $y \leq c(x,y)$  lassen sich Umkehrfunktionen so definieren:

$$\begin{aligned} e'(n,m,k) &= \max\{x \leq n \mid \exists y \leq k : c(x,y) = m\} \\ e(n) &= e'(n,n,n) \end{aligned}$$

$$\begin{aligned} f'(n,m,k) &= \max\{y \leq n \mid \exists x \leq k : c(x,y) = m\} \\ f(n) &= f'(n,n,n) \end{aligned}$$

Da auch " $c(x,y) = m$ " primitiv rekursiv ist, sind  $e$  und  $f$  auch primitiv rekursiv (max und beschränkter Existenzquantor sind gerade  $q_{Q_P}$  und  $Q_P$  für  $P = "c(x,y) = m"$ ).

Satz: Die Klasse der primitiv rekursiven Funktionen stimmt genau mit der Klasse der LOOP-berechenbaren Funktionen überein.

Beweis: LOOP-berechenbar  $\rightarrow$  primitiv rekursiv

Falls  $r$ -stellige Funktion  $f$  LOOP-berechenbar, so gibt es LOOP-Programm  $P$  mit  $k+1$  ( $k+1 > r$ ) Variablen, das  $f$  berechnet. Wir zeigen durch Induktion über die Struktur von  $P$ , dass es einstellige primitiv rekursive Funktion  $g_P$  gibt, so dass

$$g_P(\langle a_0, \dots, a_k \rangle) = \langle b_0, \dots, b_k \rangle$$

gdw.  $P$  gestartet mit Werten  $a_0, \dots, a_k$  der Variablen  $x_0, \dots, x_k$  terminiert mit Werten  $b_0, \dots, b_k$  dieser Variablen.

Induktionsanfang: Falls  $P$  Zuweisung der Form  $x_i := x_j \pm c$  ist, so ist

$$g_P(n) = \langle d_0(n), \dots, d_{i-1}(n), d_j(n) \pm c, d_{i+1}(n), \dots, d_k(n) \rangle$$

Induktionsschritt: Seien Q, R Programme, für die es die Funktionen  $g_Q(n)$  und  $g_R(n)$  gibt (Induktionsannahme).

Falls P die Form Q;R hat, so ist  $g_P(n) = g_R(g_Q(n))$ .

Falls P die Form LOOP  $x_i$  DO Q END hat, so ist  $g_P(n) = h(d_i(n), n)$ , wobei

$$\begin{aligned} h(0, x) &= x \\ h(n+1, x) &= g_Q(h(n, x)) \end{aligned}$$

$h(z, x)$  modelliert z-maliges Anwenden von  $g_Q$  auf x.

Es gibt also für alle P ein entsprechendes  $g_P(n)$ . Jetzt gilt:

$$f(x_1, \dots, x_r) = d_0(g_P(\langle 0, n_1, \dots, n_r, 0, \dots, 0 \rangle)) \quad \% \text{ rechts k-r Nullen}$$

Primitiv rekursiv  $\rightarrow$  LOOP-berechenbar

Induktion über Struktur der primitiv rekursiven Funktion:

Basisfunktionen LOOP-berechenbar,

Komposition von Funktionen durch Hintereinanderausführen von Programmen und Speichern von Zwischenergebnissen in Variablen,

primitive Rekursion der Form

$$\begin{aligned} f(0, x_1, \dots, x_r) &= g(x_1, \dots, x_r) \\ f(n+1, x_1, \dots, x_r) &= h(f(n, x_1, \dots, x_r), n, x_1, \dots, x_r) \end{aligned}$$

Berechnung von  $f(n, x_1, \dots, x_r)$  durch:

$$\begin{aligned} y &:= g(x_1, \dots, x_r); z := n; k := 0; \\ \text{LOOP } z \text{ DO } y &:= h(y, k, x_1, \dots, x_r); k := k+1 \text{ END} \end{aligned}$$

Anmerkung: das entsprechende LOOP-Programm in Schöning funktioniert nicht:

Gegenbeispiel:  $f(0) = 0; f(x+1) = h(f(x), x)$  mit  $h(x, y) = x^2 + y + 1$

Def.: Sei  $f$  eine  $k+1$ -stellige Funktion. Die durch Anwendung des  $\mu$ -Operators auf  $f$  entstehende Funktion  $\mu(f): \mathbb{N}^k \rightarrow \mathbb{N}$  ist definiert wie folgt:

$$\mu(f)(n_1, \dots, n_k) = \min\{n \mid f(n, n_1, \dots, n_k) = 0 \text{ und für alle } m < n \text{ ist } f(m, n_1, \dots, n_k) \text{ definiert}\}$$

Dabei gilt:  $\min\{\} = \text{undefiniert}$ .

Def.: Die Klasse der  $\mu$ -rekursiven Funktionen ist die kleinste Klasse von Funktionen, die die Basisfunktionen enthält und abgeschlossen ist unter Einsetzung, primitiver Rekursion und Anwendung des  $\mu$ -Operators.

Satz: Die Klasse der  $\mu$ -rekursiven Funktionen stimmt genau mit der Klasse der Turing-(WHILE-, GOTO-) berechenbaren Funktionen überein.

Beweis (Ergänzung zu Beweis für primitiv rekursiv  $\Leftrightarrow$  LOOP-berechenbar, nur WHILE noch zu behandeln):

( $\Leftarrow$ ) Sei P Programm der Form WHILE  $x_i \neq 0$  DO Q END.

Wieder sei  $h_Q(n,x)$  die Funktion, die den Zustand der Programmvariablen nach  $n$  Ausführungen von Q wiedergibt (siehe Beweis primitiv rekursiv  $\Leftrightarrow$  LOOP-berechenbar). Dann ist

$$g_P(n) = h_Q(\mu(d; h_Q)(n), n)$$

$\mu(d; h)(n)$  ist die kleinste Anzahl von Schleifendurchläufen, nach deren Ausführung die Variable  $x_i$  im Programm P den Wert 0 bekommt.

( $\Rightarrow$ ) Sei  $g = \mu(f)$ , WHILE-Programm für  $f$  existiert nach Induktionsvoraussetzung.

Folgendes Programm berechnet  $\mu(f)$ :

```
x0 := 0; y := f(0, x1, ..., xk);  
WHILE y  $\neq$  0 DO x0 := x0 + 1; y := f(x0, x1, ..., xk) END
```

## 5. Die Ackermannfunktion

Eingeführt 1928 von Ackermann, später von Hermes vereinfacht, dessen Version heute gebräuchlich ist:

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

Beispiele:

$$\begin{aligned} \text{ack}(2,0) &= \text{ack}(1,1) = \text{ack}(0, \text{ack}(1,0)) = \text{ack}(0, \text{ack}(0,1)) = \text{ack}(0,2) = 3 \\ \text{ack}(1,2) &= \text{ack}(0, \text{ack}(1,1)) = \text{ack}(0,3) = 4 \\ \text{ack}(2,1) &= \text{ack}(1, \text{ack}(2,0)) = \text{ack}(1,3) = \text{ack}(0, \text{ack}(1,2)) = \text{ack}(0,4) = 5 \end{aligned}$$

Satz: Die Ackermannfunktion ist total.

Bew.: Induktion über 1. Argument

Induktionsanfang: für  $x = 0$  und für alle  $y$  gilt  $\text{ack}(x,y) = y + 1$

Induktionsschritt: für  $x-1$  und alle  $y$  gelte:  $\text{ack}(x-1,y)$  ist definiert

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

Da  $\text{ack}(x-1,y)$  für alle  $y$  definiert ist, muss auch  $\text{ack}(x,y)$  definiert sein.

Ackermann Turing-berechenbar, aber wächst schneller als alle primitiv rekursiven Funktionen. alternative Definition, nach der ersten Zahl entwickelt:

$$\begin{aligned} \text{ack}(0,n) &= n + 1 \\ \text{ack}(1,n) &= 2 + (n + 3) - 3 \\ \text{ack}(2,n) &= 2 * (n + 3) - 3 \\ \text{ack}(3,n) &= 2 ^ (n + 3) - 3 \\ \text{ack}(4,n) &= 2 ^ 2 ^ \dots ^ 2 - 3 \text{ (wobei die Potenz } (n+3)\text{-mal vorkommt)} \end{aligned}$$

### Exkurs: Knuths Up Arrow Notation

Multiplikation mit natürlicher Zahl ist iterierte Addition:

$$ab = \underbrace{a + a + \dots + a}_{b \text{ copies of } a}$$

Potenzieren ist iterierte Multiplikation:

$$a \uparrow b = a^b = \underbrace{a \times a \times \dots \times a}_{b \text{ copies of } a}$$

Das inspirierte Knuth dazu, einen 'double arrow' Operator für iteriertes Potenzieren zu definieren:

$$a \uparrow\uparrow b = \underbrace{a^{a^{\dots^a}}}_{b \text{ copies of } a} = \underbrace{a \uparrow a \uparrow \dots \uparrow a}_{b \text{ copies of } a}$$

Nach dieser Definition ist etwa:

$$\begin{aligned}
 3 \uparrow\uparrow 2 &= 3^3 = 27 \\
 3 \uparrow\uparrow 3 &= 3^{3^3} = 3^{27} = 7625597484987 \\
 3 \uparrow\uparrow 4 &= 3^{3^{3^3}} = 3^{7625597484987} \\
 3 \uparrow\uparrow 5 &= 3^{3^{3^{3^3}}} = 3^{3^{7625597484987}} \\
 &\text{usw.}
 \end{aligned}$$

Entsprechend lässt sich ein 'triple arrow' Operator für iterierte Anwendung des 'double arrow' Operator definieren:

$$a \uparrow\uparrow\uparrow b = \underbrace{a \uparrow\uparrow a \uparrow\uparrow \dots \uparrow\uparrow a}_{b \text{ copies of } a}$$

und so weiter. Allgemein expandiert ein  $n$ -arrow Operator in eine Folge von  $(n - 1)$ -arrow Operatoren:

$$a \underbrace{\uparrow\uparrow\dots\uparrow}_n b = a \underbrace{\uparrow\uparrow\dots\uparrow}_{n-1} a \underbrace{\uparrow\uparrow\dots\uparrow}_{n-1} a \dots a \underbrace{\uparrow\uparrow\dots\uparrow}_{n-1} a$$

$b \text{ copies of } a$

Die vollständige formale Definition ist wie folgt:

$$a \uparrow^n b = \begin{cases} a^b, & \text{if } n = 1; \\ 1, & \text{if } b = 0; \\ a \uparrow^{n-1} (a \uparrow^n (b - 1)), & \text{otherwise} \end{cases}$$

für alle  $a, b, n$  mit  $a, b \geq 0, n \geq 1$ .

Mit dieser Notation lässt sich die Ackermannfunktion so formulieren:

$$\begin{aligned}
 \text{ack}(0, n) &= n + 1 \\
 \text{ack}(1, n) &= 2 + (n + 3) - 3 \\
 \text{ack}(2, n) &= 2 (n + 3) - 3 \\
 \text{ack}(3, n) &= 2 \uparrow (n + 3) - 3 \\
 \text{ack}(4, n) &= 2 \uparrow\uparrow (n + 3) - 3 \\
 \text{ack}(5, n) &= 2 \uparrow\uparrow\uparrow (n + 3) - 3 \quad \text{usw.}
 \end{aligned}$$

Man kann beweisen (Beweis siehe Schöning): die Ackermannfunktion wächst schneller als jede LOOP-berechenbare Funktion, also gilt:

Satz: Die Ackermannfunktion ist nicht LOOP-berechenbar.

Satz: Die Ackermannfunktion ist WHILE-berechenbar.

Beweis: Wir zeigen zunächst, wie die Berechnung mit Hilfe eines Stacks durchgeführt werden kann, dann wie Stacks mit WHILE-Programmen simuliert werden können.

PUSH(x) legt x auf den Stack ab,  
 POP entfernt oberstes Stackelement und liefert dessen Wert,  
 INIT initialisiert leeren Stack.

Folgendes Programm berechnet ack(x,y):

```

INIT;
PUSH(x);
PUSH(y);
WHILE size ≠ 1 DO                                % size: Größe des Stacks
  y := POP;
  x := POP;
  IF x = 0 THEN PUSH(y+1) ELSE
    IF y = 0 THEN PUSH(x-1); PUSH(1)
    ELSE PUSH(x-1); PUSH(x); PUSH(y-1) END
  END
END;
x0 := POP
  
```

Stackinhalt  $n_1 n_2 \dots n_k$  ( $n_1$  Top-Element) bedeutet:  $\text{ack}(n_k, \text{ack}(n_{k-1}, \dots \text{ack}(n_2, n_1) \dots))$  zu berechnen.

Mit den Codierungsfunktionen können wir den Stack als natürliche Zahl codieren und einer Variablen s zuweisen:  $s := \langle n_1, n_2, \dots, n_k \rangle$ . Die Stack-Operationen lassen sich so modellieren:

```

INIT      s := 0; size := 0
PUSH(x)   s := c(x,s); size := size + 1
z:= POP   z := e_c(s); s := f_c(s); size := size - 1
  
```

Daraus folgt also: Es gibt totale berechenbare Funktionen, die nicht LOOP berechenbar sind.

Wertetabelle:

ack(x,y)		y =												
		0	1	2	3	4	5	6	7	8	9	10	y	
x =	0	1	2	3	4	5	6	7	8	9	10	11	y+1	
	1	2	3	4	5	6	7	8	9	10	11	12	y+2	
	2	3	5	7	9	11	13	15	17	19	21	23	2y + 3	
	3	5	13	29	61	125	253	509	1021	2045	4093	8189	$2^{y+3} - 3$	
	4	13	65533	ack(4,2)										$2 \uparrow (y + 3) - 3$
	5	65533												$2 \uparrow \uparrow (y + 3) - 3$

ack(4,2) hat bereits 19809 Stellen in der Dezimaldarstellung.