3. LOOP-, WHILE- und GOTO-Berechenbarkeit

3.1 LOOP-Programme

Komponenten:	Variablen:	$x_0, x_1, x_2, \ldots, y, z, \ldots$
	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 LOOP x_i DO P END ein LOOP-Programm.

Semantik:

- LOOP-Programme berechnen Funktionen über $\mathbb{N}$.
- Eingabewerte $n_1, ..., n_k$ bei Beginn der Programmausführung an Variablen $x_1, ..., x_k$ 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.
- LOOP x_i DO P 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 $f: \mathbb{N}^k \to \mathbb{N}$ heißt LOOP-berechenbar, falls es ein LOOP-Programm gibt, das f berechnet, d.h. P gestartet mit der Variablenbelegung $n_1, ..., n_k$ der Variablen $x_1, ..., x_k$ (alle anderen Variablen initialisiert mit 0) stoppt mit $f(n_1, ..., n_k)$ als Wert der Variable x_0.

Anmerkung:
offensichtlich sind nur totale Funktionen berechenbar (aber nicht alle totalen -> später).

Abkürzungen:	$x_i := x_j$	statt $x_i := x_j + 0$
	$x_i := c$	statt $x_i := x_j + c$, 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:
x_0:= x_1;
LOOP x_2 DO x_0 := x_0 + 1 END

Multiplikation:
x_0:= 0;
LOOP x_2 DO x_0 := x_0 + x_1 END

ausführlich:
x_0:= 0;
LOOP x_2 DO LOOP x_1 DO x_0 := x_0 + 1 END END

$f(n_1,n_2)$ = 1 falls $n_1 = n_2$, 0 sonst:

x_0 := 1;
x_3 := x_1;
LOOP x_2 DO x_3 := x_3 – 1 END;
LOOP x_3 DO x_0 := 0 END;
x_3 := x_2;
LOOP x_1 DO x_3 := x_3 – 1 END;
LOOP x_3 DO x_0 := 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 LOOP x_i DO P END ein WHILE-Programm;
4. falls P ein WHILE-Programm ist, so ist WHILE $x_i \neq 0$ DO P 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: $\mathbb{N}^k$ -> $\mathbb{N}$ heißt WHILE-berechenbar, falls es ein WHILE-Programm gibt, das f berechnet, d.h. P gestartet mit der Variablenbelegung n_1, ..., n_k der Variablen x_1, ...,

x_k (alle anderen Variablen initialisiert mit 0) stoppt mit $f(n_1, ..., n_k)$ als Wert der Variable x_0, falls $f(n_1, ..., n_k)$ 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

$x_i := 1;$
WHILE $x_i \neq 0$ DO $x_i := 1$ 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:

Wertzuweisung, 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:

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

Zulässige Anweisungen:

Wertzuweisungen:	$x_i := x_j + c$ oder $x_i := x_j - c$
unbedingter Sprung:	GOTO M_i
bedingter Sprung:	IF $x_i = c$ THEN GOTO M_j
Stopanweisung:	HALT

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

Semantik:
Wertzuweisung wie bisher,
GOTO legt nächste auszuführende Programmanweisung fest,
Stopanweisung bedeutet Terminieren,
Ausgabe ist Wert von x_0.

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

M_0: $x_3 := x_2$;
M_1: IF $x_3 = 0$ THEN GOTO M_2;
IF $x_1 = 0$ THEN GOTO M_3
$x_1 := x_1 - 1$;
$x_3 := x_3 - 1$;
GOTO M_1;
M_2: $x_0 := x_0 + 1$;
GOTO M_0:
M_3: HALT

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

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

WHILE $x_i \neq 0$ DO P END

wird simuliert durch:

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

wobei M_1, M_2 geeignete Marken sind.

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

Umgekehrte Richtung:

Gegeben GOTO-Programm $M_1:A_1$; $M_2:A_2$; ... $M_k:A_k$

Das Programm wird simuliert durch folgendes WHILE-Programm:

count := 1;
WHILE count $\neq 0$ DO
IF count = 1 THEN A_1' END;
IF count = 2 THEN A_2' END;
…
IF count = k THEN A_k' END;
END

Hierbei gilt:

A_i' =	$x_i := x_j \pm c$; count := count + 1	falls A_i =	$x_i := x_j \pm c$
	count := n	falls A_i =	GOTO M_n
	IF $x_j = c$ THEN count := n ELSE count := count + 1 END	falls A_i =	IF $x_j = c$ THEN GOTO M_n
	count := 0	falls A_i =	HALT

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

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

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

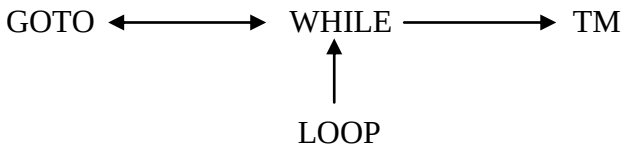
Satz: Jedes GOTO Programm kann durch ein WHILE-Programm simuliert werden. Damit ist jede GOTO-berechenbare Funktion WHILE-berechenbar.

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 ⟶ Y heißt X-berechenbar impliziert Y-berechenbar bzw. Y-Programm kann X-Programm simulieren):



Es fehlt noch TM → GOTO, dann ist die Äquivalenz aller (bis auf LOOP) gezeigt.

Sei M = (Z, Σ, Γ, δ, z_1, □, 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_1: transformiert Anfangswerte der Variablen in Binärdarstellung
erzeugt Darstellung der Startkonfiguration von M
Codierung in 3 Variablen x,y,z
P_2: Schritt-für-Schritt-Simulation der Rechnung von M
P_3: erzeugt aus Zielkonfiguration Wert der Ausgabevariable x_0.

Seien Z = {z_1, …, z_k}, Γ = {a_1, …, a_m}, b > | Γ | (größer nötig, sonst evtl. führende Nullen)

TM-Konfiguration $a_{i1}…a_{ip}z_ra_{j1}…a_{jq}$ 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):

$x = (i_1 \dots i_p)_b$
$y = (j_q \dots j_1)_b$
z = r

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

$$x = \Sigma_{t=1\ldots p}\, i_t \times b^{p-t}$$

Analog für y, aber umgekehrte Reihenfolge.

Beispiel: $\Gamma = \{a_1, \ldots, a_3\}$, b = 4
Konfiguration $a_1a_1z_2a_3a_2$ 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 μ-rekursive Funktionen

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 und 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 $\mathbf{c^j_k}$ mit $\mathbf{c^j_k}(n_1,...,n_j) = k$ für beliebige $n_1,...,n_j$ sind primitiv rekursiv.
2. Alle Projektionen $\boldsymbol{\pi}^{\mathbf{j}}_{\mathbf{i}}$ mit $\boldsymbol{\pi}^{\mathbf{j}}_{\mathbf{i}}(n_1,...,n_j) = n_i$ $(i \leq j)$ sind primitiv rekursiv.
3. Die Nachfolgerfunktion **s** mit $s(n) = n + 1$ ist primitiv rekursiv.
4. Wenn $g: \mathbb{N}^r \to \mathbb{N}$ und $h_1: \mathbb{N}^k \to \mathbb{N}$, …, $h_r: \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv sind, so ist auch die k-stellige Funktion $f=\mathbf{SUB(g,h_1,...,h_r)}$ mit
   $$f(n_1, ..., n_k) = g(h_1(n_1, ..., n_k),\ldots, h_r(n_1, ..., n_k))$$
   primitiv rekursiv.
5. Wenn $g: \mathbb{N}^{k-1} \to \mathbb{N}$ und $h: \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursive Funktion sind, dann auch die k-stellige Funktion $f=\mathbf{PR(g,h)}$: mit:

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

Beispiele:

Ansatz: übliche Rekursion:		nach Definition:		
add(0, x)	= x	=g(x)	also	$g=\pi^1_1$
add(n+1, x)	= s(add(n,x))	=h(add(n,x),n,x)	also	$h=SUB(s, \pi^3_1)$
		ergibt: $add=PR(\pi^1_1, SUB(s, \pi^3_1))$		

mult(0, x) = 0
mult(n+1, x) = add(mult(n,x),x)

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 $n_2, ..., n_k$ von f und g benutzt werden. Aus demselben Grund muss Rekursion in f nicht notwendiger Weise über das erste Argument laufen.

mult(0, x) = 0
mult(n+1, x) = add(mult(n,x),x)

sub(x,0) = x
sub(x,n+1) = sub_1(sub(x,n))

sub_1(0) = 0
sub_1(n+1) = n

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

$=_0(0) = 1$
$=_0(n+1) = 0$

$\neq_0(0) = 0$
$\neq_0(n+1) = 1$

$>(x,y) = \neq_0(sub(x,y))$
$=(x,y) = mult(sub(1, >(x,y)), sub(1, >(y,x)))$

$div(x,y) = div_3(x,y,0)$

$div_3(0,y,z) = =(y,z)$
$div_3(n+1,y,z) = >(y,z) * div_3(n,y,s(z)) + =(y,z) * s(div_3(n,y,1))$

Beispiel:
$div(5,2) = div_3(5,2,0) = div_3(4,2,1) = div_3(3,2,2)$
$= 1 + div_3(2,2,1) = 1 + div_3(1,2,2) = 1 + 1 + div_3(0,2,1) = 2$

$div(4,2) = div_3(4,2,0) = div_3(3,2,1) = div_3(2,2,2)$
$= 1 + div_3(1,2,1) = 1 + div_3(0,2,2) = 1 + 1 = 2$

$mod(x,y) = mod_3(x,y,0)$

$mod_3(0,y,z) = \neq(y,z) * z$
$mod_3(n+1,y,z) = >(y,z) * mod_3(n,y,s(z)) + =(y,z) * mod_3(n,y,1)$

Beispiel:
$mod(5,2) = mod_3(5,2,0) = mod_3(4,2,1) = mod_3(3,2,2)$
$= mod_3(2,2,1) = mod_3(1,2,2) = mod_3(0,2,1) = 1$

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

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

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

Zur Vorbereitung des Beweises primitiv rekursiv ⇔ 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) = 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 $x_1 \geq x_m$ 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 = Eingabe – Summe 1 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:

$$<n_0, n_1,...n_k> = c(n_0, c(n_1, ...,c(n_k,0)...))$$

Mit e_c und f_c kann man daraus auch die k+1 Umkehrfunktionen der k+1-stelligen Codierfunktion < > bekommen:

$d_0(n) = e_c(n)$
$d_1(n) = e_c(f_c(n))$
…
$d_k(n) = e_c(f_c(f_c(\ldots f_c(n)\ldots))$ k-mal f_c

Diese Funktionen sind primitiv rekursiv, falls e_c und f_c es sind. Letzteres zeigen wir jetzt.

Vorbemerkung:

gegeben n-stelliges Prädikat $P(x_1, ..., x_n)$ (Funktion, die Tupel den Wert 0 oder 1 zuordnet)

der beschränkte max-Operator für Argument i von P, $max^i_P(x_1, ..., x_n)$, liefert das größte $x \leq x_i$, für das $P(x_1, ..., x_{i-1}, x, x_{i+1}, ... x_n) = 1$.

$$max^i_P(x_1, ..., x_n) = max\{x \leq x_i \mid P(x_1, ..., x_{i-1}, x, x_{i+1}, ... x_n) = 1\}$$

Falls es kein solches x gibt, ist $max^i_P(x_1, ..., x_n) = 0$. max^i_P kann wie folgt definiert werden :

$$max^i_P(x_1, ..., x_{i-1}, 0, x_{i+1}, ... x_n) = 0$$

$$max^i_P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n) = \begin{cases} n+1 & \text{falls } P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n) = 1 \\ max^i_P(x_1, ..., x_{i-1}, n, x_{i+1}, ... x_n) & \text{sonst} \end{cases}$$

$$= P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n) * (n+1) + (1 - P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n)) * max^i_P(x_1, ..., x_{i-1}, n, x_{i+1}, ... x_n)$$

max^i_P ist damit primitiv rekursiv, falls P primitiv rekursiv ist.

Ähnlich können wir ein Prädikat $ex^i_P(x_1, ..., x_n)$ definieren, das Wert 1 hat, wenn es $y \leq x_i$ gibt, für das $P(x_1, ..., x_{i-1}, y, x_{i+1}, ... x_n) = 1$:

$$ex^i_P(x_1, ..., x_n) = 1 \text{ gdw. } \exists\, y \leq x_i : P(x_1, ..., x_{i-1}, y, x_{i+1}, ... x_n) = 1.$$

$$ex^i_P(x_1, ..., x_{i-1}, 0, x_{i+1}, ... x_n) = P(x_1, ..., x_{i-1}, 0, x_{i+1}, ... x_n)$$

$$ex^i_P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n) = P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n) + ex^i_P(x_1, ..., x_{i-1}, n, x_{i+1}, ... x_n) - P(x_1, ..., x_{i-1}, n+1, x_{i+1}, ... x_n) * ex^i_P(x_1, ..., x_{i-1}, n, x_{i+1}, ... x_n)$$

(wenn beide gelten, 1 abziehen)

ex^i_P ist offensichtlich primitiv rekursiv, falls P primitiv rekursiv ist.

Betrachten wir nun das primitiv rekursive 3-stellige Prädikat P mit

$$P(x_1, x_2, x_3) = 1 \text{ gdw. } c(x_1, x_2) = x_3$$

Für dieses P ist $ex^2_P(x_1, x_2, x_3) = 1$ gdw. $\exists\, y \leq x_2 : c(x_1, y) = x_3$.

Der Einfachheit halber nennen wir das neue Prädikat Q2, also $Q2(x_1, x_2, x_3) = ex^2_P(x_1, x_2, x_3)$. Wenden wir nun den beschränkten max-Operator auf das erste Argument von Q2 an, so ergibt sich :

$$max^1_{Q2}(x_1, x_2, x_3) = max\{x \leq x_1 \mid \exists\, y \leq x_2 : c(x, y) = x_3\}$$

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

$$e_c(n) = max^1_{Q2}(n, n, n) = max\{x \leq n \mid \exists\, y \leq n : c(x, y) = n\}$$

Analog definieren wir das dreistellige Prädikat $Q1 = ex^1_P$ sowie max^2_{Q1}. Dann ist

$$f_c(n) = max^2_{Q1}(n, n, n) = max\{y \leq n \mid \exists\, x \leq n : c(x,y) = n\}.$$

Damit sind die Umkehrfunktionen e_c und f_c auch primitiv rekursiv. Dieses Ergebnis wird für den Beweis des folgenden Satzes benötigt:

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

Beweis: LOOP-berechenbar => 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

$$g_P(<a_0,...,a_k>) = <b_0,...,b_k>$$

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

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

$$g_P(n) = <d_0(n),... d_{i-1}(n), d_j(n) \pm c, d_{i+1}(n), \ldots, d_k(n)>$$

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 $g_P(n) = h_Q(d_i(n),n)$, wobei

$h_Q(0,x) = x$
$h_Q(n+1,x) = g_Q(h_Q(n,x))$

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

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

$f(x_1,...,x_r) = d_0(g_P(<0, n_1,...,n_r,0,...,0>)$ % rechts k-r Nullen

Primitiv rekursiv => 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

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

Berechnung von $f(n, x_1, ..., x_r)$ durch folgendes LOOP-Programm:

$y := g(x_1, ..., x_r)$; $z := n$; $k := 0$;
LOOP z DO $y := h(y, k, x_1, ..., x_r)$; $k := k+1$ END

Anmerkung: das entsprechende LOOP-Programm in Schöning funktioniert nicht:
Gegenbeispiel: $f(0) = 0$; $f(x+1) = h(f(x),x)$ mit $h(x,y) = x^2 + y + 1$)

Wir führen nun einen Operator ein, der es erlaubt, weitere (insbesondere echt partielle) Funktionen zu definieren:

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

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

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

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

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

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

(<=) 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 ⇔ LOOP-berechenbar). Dann ist

$$g_P(n) = h_Q(\mu(d_i\, h_Q)(n),n)$$

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

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

Folgendes Programm berechnet $\mu(f)$:

$x_0 := 0$; $y := f(0, x_1, ..., x_k)$;
WHILE $y \neq 0$ DO $x_0 := x_0 + 1$; $y := f(x_0, x_1, ..., x_k)$ END

5. Die Ackermannfunktion

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

ack(0,y) = y + 1
ack(x,0) = ack(x-1, 1) x > 0
ack(x,y) = ack(x-1, ack(x, y-1)) x,y > 0

Beispiele:

ack(2,0) = ack(1,1) = ack(0, ack(1,0)) = ack(0, ack(0,1)) = ack(0,2) = 3
ack(1,2) = ack(0,ack(1,1)) = ack(0,3) = 4
ack(2,1) = ack(1,ack(2,0)) = ack(1,3) = ack(0,ack(1,2)) = ack(0,4) = 5

Satz: Die Ackermannfunktion ist total.

Bew.: Induktion über 1. Argument
Induktionsanfang: für x = 0 und für alle y gilt ack(x,y) = y + 1
Induktionsschritt: für x-1 und alle y gelte: ack(x-1,y) ist definiert
ack(x,y) = ack(x-1, ack(x, y-1)) = ack(x-1, ack(x-1, … ack(x-1,1)…)) y+1 mal ack.
Da ack(x-1,y) für alle y definiert ist, muss auch ack(x,y) definiert sein.

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

ack(0,n) = n + 1
ack(1,n) = 2 + (n + 3) - 3
ack(2,n) = 2 * (n + 3) - 3
ack(3,n) = 2 ^ (n + 3) - 3
ack(4,n) = 2 ^ 2 ^ ... ^ 2 - 3 (wobei die Potenz (n+3)-mal vorkommt)

Exkurs: Knuths Up Arrow Notation

Multiplikation mit natürlicher Zahl ist iterierte Addition:

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

Potenzieren ist iterierte Multiplikation:

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

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

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

Nach dieser Definition ist etwa:

$$3 \uparrow\uparrow 2 = 3^3 = 27$$
$$3 \uparrow\uparrow 3 = 3^{3^3} = 3^{27} = 7625597484987$$
$$3 \uparrow\uparrow 4 = 3^{3^{3^3}} = 3^{7625597484987}$$
$$3 \uparrow\uparrow 5 = 3^{3^{3^{3^3}}} = 3^{3^{7625597484987}}$$

usw.

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

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

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

$$a \underbrace{\uparrow\uparrow\ldots\uparrow}_{n} b = \underbrace{a \underbrace{\uparrow\ldots\uparrow}_{n-1} a \underbrace{\uparrow\ldots\uparrow}_{n-1} a \ \ldots \ a \underbrace{\uparrow\ldots\uparrow}_{n-1} a}_{b \text{ copies of } a}$$

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 ≥ 0, n ≥ 1.

Mit dieser Notation lässt sich die Ackermannfunktion so formulieren:

ack(0,n) = n + 1
ack(1, n) = 2 + (n + 3) - 3
ack(2, n) = 2 (n + 3) - 3
ack(3, n) = 2 ↑ (n + 3) - 3
ack(4, n) = 2 ↑↑ (n + 3) - 3
ack(5, n) = 2 ↑↑↑ (n + 3) – 3 usw.

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

Mit den Codierungsfunktionen können wir den Stack als natürliche Zahl codieren und einer Variablen s zuweisen: s := < n_1,n_2,…,n_k>. 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↑ ↑ (y + 3) - 3
	5	65533											2↑ ↑ ↑ (y + 3) - 3

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