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

**3.1 LOOP-Programme**

Komponenten: Variablen: x0, x1, x2, …, y, z, …

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 xi:= xj + c oder xi:= xj - c ist ein LOOP-Programm, wobei c Konstante ist;
2. falls P1 und P2 LOOP-Programme sind, so ist P1; P2 ein LOOP-Programm;
3. falls Pein LOOP-Programm ist, so ist LOOP xi DO P END ein LOOP-Programm.

Semantik:

* LOOP-Programme berechnen Funktionen über IN.
* Eingabewerte n1, ..., nk bei Beginn der Programmausführung an Variablen x1, ..., xk gebunden.
* Alle anderen Variablen haben Anfangswert 0.
* Ausgabewert ist Wert von x0 nach Beendigung des Programms.
* Wertzuweisung: wie üblich, "-" ist modifizierte Subtraktion: xj - c = 0, falls c > xj.
* P1; P2 Hintereinanderausführung von P1 und P2.
* LOOP xi DO P END: P wird xi mal ausgeführt. Zu beachten: der Wert von xi wird vor (!) der ersten Ausführung von P ermittelt und die Schleife entsprechend oft ausgeführt. Zuweisungen an xi in P haben keine Auswirkung auf die Zahl der Schleifendurchläufe!

**Def.:** Eine Funktion f: IN k -> IN heißt LOOP-berechenbar, falls es ein LOOP-Programm gibt, das f berechnet, d.h. P gestartet mit der Variablenbelegung n1, ..., nk der Variablen x1, ..., xk (alle anderen Variablen initialisiert mit 0) stoppt mit f(n1, ..., nk) als Wert der Variable x0.

Anmerkung:

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

Abkürzungen: xi:= xj statt xi:= xj + 0

xi:= c statt xi:= xj + c, wobei xj 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 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

Weitere Beispiele:

Addition:

x0:= x1;

LOOP x2 DO x0 := x0 + 1END

Multiplikation:

x0:= 0;

LOOP x2 DO x0 := x0 + x1 END

ausführlich:

x0:= 0;

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

f(n1,n2) = 1 falls n1 = n2, 0 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 xi:= xj + c oder xi:= xj - c ist ein WHILE-Programm, wobei c Konstante ist;
2. falls P1 und P2 WHILE-Programme sind, so ist P1; P2 ein WHILE-Programm;
3. falls Pein WHILE-Programm ist, so ist LOOP xi DO P END ein WHILE-Programm;
4. falls Pein WHILE-Programm ist, so ist WHILE xi ≠ 0 DO P END ein WHILE-Programm.

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

Anmerkung: LOOP-Konstrukt nicht mehr unbedingt nötig, da

LOOP xi DO P END

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

y := xi;

WHILE y ≠ 0 DO y := y-1; P END

**Def.:** Eine Funktion f: IN k -> IN heißt WHILE-berechenbar, falls es ein WHILE-Programm gibt, das f berechnet, d.h. P gestartet mit der Variablenbelegung n1, ..., nk der Variablen x1, ..., xk (alle anderen Variablen initialisiert mit 0) stoppt mit f(n1, ..., nk) als Wert der Variable x0, falls f(n1, ..., nk) 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:

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 Ai, die Marke Mi besitzen:

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

Zulässige Anweisungen:

Wertzuweisungen: 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 Ak ist entweder ein unbedingter Sprung oder HALT.

Semantik:

Wertzuweisung wie bisher,

GOTO legt nächste auszuführende Programmanweisung fest,

Stopanweisung bedeutet Terminieren,

Ausgabe ist Wert von x0.

Beispiel: ganzzahlige Division (ohne Rest, undefiniert falls x2 = 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 M1, M2 geeignete Marken sind.

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

Umgekehrte Richtung:

Gegeben GOTO-Programm M1:A1; M2:A2; ... Mk:Ak

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:

Ai' = xi:= xj ± c; count := count + 1 falls Ai = xi:= xj ± c

count := n falls Ai = GOTO Mn

IF xj = c THEN count := n

ELSE count := count + 1 END falls Ai = IF xj = c THEN GOTO Mn

count := 0 falls Ai = 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.

(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):

GOTO WHILE TM

LOOP

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

Sei M = (Z, , , , z1, , E) TM zur Berechnung von f.

M wird simuliert durch ein GOTO-Programm

M1:P1; M2:P2; M3:P3

P1: transformiert Anfangswerte der Variablen in Binärdarstellung

erzeugt Darstellung der Startkonfiguration von M

Codierung in 3 Variablen x,y,z

P2: Schritt-für-Schritt-Simulation der Rechnung von M

P3: erzeugt aus Zielkonfiguration Wert der Ausgabevariable x0.

Seien Z = {z1, …, zk},  = {a1, …, am}, b > |  | (größer nötig, sonst evtl. führende Nullen)

TM-Konfiguration ai1…aipzraj1…ajq wird durch Variablenwerte repräsentiert: Indizes der Symbole links von zr werden als Zahl zur Basis b gelesen, entsprechend Indizes der Symbole rechts (in umgekehrter Reihenfolge):

x = (i1 … ip)b

y = (jq … j1)b

z = r

(i1 … ip)b ist Wert der zur Basis b dargestellten Zahl i1 … ip :

x = t=1…p it × bp-t

Analog für y, aber umgekehrte Reihenfolge.

Beispiel:  = {a1, …,a3}, b = 4

Konfiguration a1a1z2a3a2 wird repräsentiert durch

x = 1 × 41 + 1 × 40  = 5

y = 3 + 8 = 11

z = 2

M2:P2 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 Pij aufgebaut? Modellieren Konfigurationsübergänge der TM.

Beispiel: es gelte (zi,aj) = (zi’,aj’,L)

dann ist Pij:

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 zi ∈ E so ist Pij: GOTO M3.

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

**4. Primitiv rekursive und -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 IN ist induktiv so definiert:

1. Alle konstanten Funktionen **cjk** mit **cjk**(n1,...,nj) = k für beliebige n1,...,nj sind primitiv rekursiv.
2. Alle Projektionen **ji** mit **ji**(n1,...,nj) = ni (i ≤ j) sind primitiv rekursiv.
3. Die Nachfolgerfunktion **s** mit s(n) = n +1 ist primitiv rekursiv.
4. Wenn g: IN r -> IN und h1: IN k -> IN, …, hr: IN k -> IN primitiv rekursiv sind, so ist auch die k-stellige Funktion f=**SUB(g,h1,...,hr)** mit

f(n1, ..., nk) = g(h1(n1, ..., nk),…, hr(n1, ..., nk))

primitiv rekursiv.

1. Wenn g: IN k-1 -> IN und h: IN k+1 -> IN primitiv rekursive Funktion sind, dann auch die k-stellige Funktion f=**PR(g,h)**: mit:

f(0, n2, ..., nk) = g(n2, ..., nk)

f(n+1, n2, ..., nk) = h(f(n, n2, ..., nk), n, n2, ..., nk).

Beispiele:

Ansatz: übliche Rekursion: nach Definition:

add(0, x) = x =g(x) also g=11

add(n+1, x) = s(add(n,x)) =h(add(n,x),n,x) also h=SUB(s,31)

ergibt: add=PR(11, SUB(s,31))

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 n2, ..., nk 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) = sub1(sub(x,n))

sub1(0) = 0

sub1(n+1) = n

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

=0(0) = 1

=0(n+1) = 0

≠0(0) = 0

≠0(n+1) = 1

>(x,y) = ≠0(sub(x,y))

=(x,y) = mult(sub(1, >(x,y)), sub(1, >(y,x)))

div(x,y) = div3(x,y,0)

div3(0,y,z) = =(y,z)

div3(n+1,y,z) = >(y,z) \* div3(n,y,s(z)) + =(y,z) \* s(div3(n,y,1))

Beispiel:

div(5,2) = div3(5,2,0) = div3(4,2,1) = div3(3,2,2)

= 1 + div3(2,2,1) = 1 + div3(1,2,2) = 1 + 1 + div3(0,2,1) = 2

div(4,2) = div3(4,2,0) = div3(3,2,1) = div3(2,2,2)

= 1 + div3(1,2,1) = 1 + div3(0,2,2) = 1 + 1 = 2

mod(x,y) = mod3(x,y,0)

mod3(0,y,z) = ≠(y,z) \* z

mod3(n+1,y,z) = >(y,z) \* mod3(n,y,s(z)) + =(y,z) \* mod3(n,y,1)

Beispiel:

mod(5,2) = mod3(5,2,0) = mod3(4,2,1) = mod3(3,2,2)

= mod3(2,2,1) = mod3(1,2,2) = mod3(0,2,1) = 1

sum(n) = i ≤ 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 IN 2 und IN.. Offensichtlich primitiv rekursiv/LOOP-berechenbar:

Auch die Umkehrfunktionen für c sind LOOP-berechenbar:

ec liefert erstes Argument von c(x,y) zurück: ec(c(x,y)) = x

fc zweites Argument: fc(c(x,y)) = y

LOOP Programm:

ec : xm := 1 ;

LOOP x1 do

IF x1 ≥ xm THEN

x1 := x1 – xm;

xm := xm + 1;

ELSE x0 := x1 END;

[für fc: Anweisung in letzter Zeile x0 := (xm – 1) – x1; außerdem muss Schleife mindestens 2 mal durchlaufen werden, damit der Fall n = 1 richtig behandelt wird]

xm 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 x1 = 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:

<n0, n1,...nk> = c(n0, c(n1, ...,c(nk,0)...))

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

d0(n) = ec(n)

d1(n) = ec(fc(n))

…

dk(n) = ec(fc(fc(…fc(n)…)) k-mal fc

Diese Funktionen sind primitiv rekursiv, falls ec und fc es sind. Letzteres zeigen wir jetzt.

Vorbemerkung:

gegeben n-stelliges Prädikat P(x1, ..., xn) (Funktion, die Tupel den Wert 0 oder 1 zuordnet)

der beschränkte max-Operator für Argument i von P, maxiP(x1, ..., xn), liefert das größte x ≤ xi, für das P(x1, ..., xi-1, x ,xi+1,... xn) = 1.

maxiP(x1, ..., xn) = max{x ≤ xi | P(x1, ..., xi-1, x ,xi+1,... xn) = 1}

Falls es kein solches x gibt, ist maxiP(x1, ..., xn) = 0. maxiP kann wie folgt definiert werden :

maxiP(x1, ..., xi-1, 0 ,xi+1,... xn) = 0

maxiP(x1, ..., xi-1, n+1 ,xi+1,... xn) = n+1 falls P(x1, ..., xi-1, n+1 ,xi+1,... xn) = 1

maxiP(x1, ..., xi-1, n, xi+1,... xn) sonst

= P(x1, ..., xi-1, n+1 ,xi+1,... xn) \* (n+1) +

(1 - P(x1, ..., xi-1, n+1 ,xi+1,... xn)) \* maxiP(x1, ..., xi-1, n, xi+1,... xn)

maxiP ist damit primitiv rekursiv, falls P primitiv rekursiv ist.

Ähnlich können wir ein Prädikat exiP(x1, ..., xn) definieren, das Wert 1 hat, wenn es y ≤ xi gibt, für das P(x1, ..., xi-1, y ,xi+1,... xn) = 1:

exiP(x1, ..., xn) = 1 gdw. ∃ y ≤ xi : P(x1, ..., xi-1, y ,xi+1,... xn) = 1.

exiP(x1, ..., xi-1, 0 ,xi+1,... xn) = P(x1, ..., xi-1, 0 ,xi+1,... xn)

exiP(x1, ..., xi-1, n+1 ,xi+1,... xn) =

P(x1, ..., xi-1, n+1 ,xi+1,... xn) + exiP(x1, ..., xi-1, n,xi+1,... xn) –

P(x1, ..., xi-1, n+1 ,xi+1,... xn) \* exiP(x1, ..., xi-1, n, xi+1,... xn)

(wenn beide gelten, 1 abziehen)

exiP ist offensichtlich primitiv rekursiv, falls P primitiv rekursiv ist.

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

P(x1,x2, x3) = 1 gdw. c(x1,x2)= x3

Für dieses P ist ex2P(x1, x2, x3) = 1 gdw. ∃ y ≤ x2 : c(x1,y) = x3.

Der Einfachheit halber nennen wir das neue Prädikat Q2, also Q2(x1,x2, x3) = ex2P(x1, x2, x3).

Wenden wir nun den beschränkten max-Operator auf das erste Argument von Q2 an, so ergibt sich :

max1Q2(x1, x2, x3) = max{x ≤ x1 | ∃ y ≤ x2 : c(x,y) = x3}

Da x ≤ c(x,y) und y ≤ c(x,y) lassen sich Umkehrfunktionen so definieren:

ec(n) = max1Q2(n, n, n) = max{x ≤ n | ∃ y ≤ n : c(x,y) = n}

Analog definieren wir das dreistellige Prädikat Q1 = ex1P sowie max2Q1. Dann ist

fc(n) = max2Q1(n, n, n) = max{y ≤ n | ∃ x ≤ n : c(x,y) = n}.

Damit sind die Umkehrfunktionen ec und fc 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 gP gibt, so dass

gP(<a0,...,ak>) = <b0,...,bk>

gdw. P gestartet mit Werten a0,...,ak der Variablen x0,...,xk terminiert mit Werten b0,...,bk dieser Variablen.

Induktionsanfang: Falls P Zuweisung der Form xi := xj ± c ist, so ist

gP(n) = <d0(n),... di-1(n), dj(n) ± c, di+1(n), …, dk(n)>

Induktionsschritt: Seien Q, R Programme, für die es die Funktionen gQ(n) und gR(n) gibt (Induktionsannahme).

Falls P die Form Q;R hat, so ist gP(n) = gR(gQ(n)).

Falls P die Form LOOP xi DO Q END hat, so ist gP(n) = hQ(di(n),n), wobei

hQ(0,x) = x

hQ(n+1,x) = gQ(hQ(n,x))

hQ(z,x) modelliert z-maliges Anwenden von gQ auf x.

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

f(x1,...,xr) = d0(gP(<0, n1,...,nr,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

f(0, x1, ..., xr) = g(x1, ..., xr)

f(n+1, x1, ..., xr) = h(f(n, x1, ..., xr), n, x1, ..., xr)

Berechnung von f(n, x1, ..., xr) durch folgendes LOOP-Programm:

y := g(x1, ..., xr); z:= n; k:= 0;

LOOP z DO y:= h(y, k, x1, ..., xr); 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) = x2 + 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 -Operators auf f entstehende Funktion (f): IN k -> IN ist definiert wie folgt:

(f)(n1, ..., nk) = min{n | f(n, n1, ..., nk) = 0 und für alle m < n ist f(m, n1, ..., nk) definiert}

Dabei gilt: min{} = undefiniert.

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

**Satz:** Die Klasse der -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 xi ≠ 0 DO Q END.

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

gP(n) = hQ((di hQ)(n),n)

(di h)(n) ist die kleinste Anzahl von Schleifendurchläufen, nach deren Ausführung die Variable xi im Programm P den Wert 0 bekommt.

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

Folgendes Programm berechnet  (f):

x0 := 0; y := f(0, x1, ..., xk);

WHILE y ≠ 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:

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

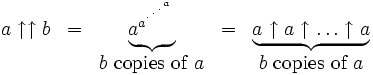
Multiplikation mit natürlicher Zahl ist iterierte Addition:



Potenzieren ist iterierte Multiplikation:



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



Nach dieser Definition ist etwa:

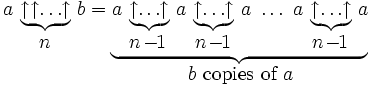


usw.

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



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



Die vollständige formale Definition ist wie folgt:



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 n1n2…nk (n1 Top-Element) bedeutet: ack(nk, ack(nk-1, ...ack(n2,n1)..)) zu berechnen.

Mit den Codierungsfunktionen können wir den Stack als natürliche Zahl codieren und einer Variablen s zuweisen: s := < n1,n2,…,nk>. Die Stack-Operationen lassen sich so modellieren:

INIT s := 0; size := 0

PUSH(x) s := c(x,s); size := size + 1

z:= POP z := ec(s); s := fc(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 | 2y + 3 - 3 |
| **4** | 13 | 65533 | [ack(4,2)](http://www.informatik.uni-freiburg.de/~zeisberg/ackermann/ack_4_2.txt) |  |  |  |  |  |  |  |  | 2↑ ↑ (y + 3) - 3 |
| **5** | 65533 |  |  |  |  |  |  |  |  |  |  | 2↑ ↑ ↑ (y + 3) - 3 |

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