

Berechenbarkeitstheorie: Teil II

Prof. Dr. David Sabel

LFE Theoretische Informatik



- Ackermannfunktion
- Primitiv rekursive Funktionen
- μ -rekursive Funktionen

Die Ackermannfunktion

- Von Wilhelm Ackermann in 1920er Jahre vorgeschlagen
- sehr schnell wachsende Funktion
- Variante von Rózsa Péter:

Ackermann-Funktion $a : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$

$$a(x, y) = \begin{cases} y + 1, & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

Die Ackermannfunktion

- Von Wilhelm Ackermann in 1920er Jahre vorgeschlagen
- sehr schnell wachsende Funktion
- Variante von Rózsa Péter:

Ackermann-Funktion $a : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$

$$a(x, y) = \begin{cases} y + 1, & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

Unser nächstes Ziel:

Die Ackermannfunktion ist WHILE-berechenbar, aber:
die Ackermannfunktion ist **nicht** LOOP-berechenbar
(obwohl sie total ist)

Einige Werte der Ackermannfunktion

Tabelle mit $a(x, y)$ -Einträgen:

$y \backslash x$	0	1	2	3	4
0	1	2	3	5	13
1	2	3	5	13	65533
2	3	4	7	29	$2^{65536} - 3$
3	4	5	9	61	$a(3, 2^{65536} - 3)$
4	5	6	11	125	$a(3, a(4, 3))$

$$a(x, y) = \begin{cases} y + 1, & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{falls } x > 0 \text{ und } y > 0 \end{cases}$$

Mehrmaliges Entfalten der dritten Zeile zeigt:

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

Daher sind alle rekursiven Aufrufe echt kleiner und daher terminiert die Ackermannfunktion stets.

Lemma

Die Ackermannfunktion ist total.

Berechenbarkeit der Ackermannfunktion

Intuitiv klar: Mit jeder modernen Programmiersprache ist die Ackermannfunktion implementierbar. Daher ist sie auch (intuitiv) berechenbar.

Berechenbarkeit der Ackermannfunktion

Intuitiv klar: Mit jeder modernen Programmiersprache ist die Ackermannfunktion implementierbar. Daher ist sie auch (intuitiv) berechenbar.

Satz

Die Ackermannfunktion ist WHILE-berechenbar.

Berechenbarkeit der Ackermannfunktion

Intuitiv klar: Mit jeder modernen Programmiersprache ist die Ackermannfunktion implementierbar. Daher ist sie auch (intuitiv) berechenbar.

Satz

Die Ackermannfunktion ist WHILE-berechenbar.

Beweis: Erstelle ein WHILE-Programm, welches die rekursive Berechnung durchführt mit einem Stack.

Daher ist der erste Schritt im Beweis:

- Darstellung des Stacks
- Implementierung von Operationen auf dem Stack als WHILE-Programm.

Danach wird das Programm angegeben, dass die Stackoperationen durchführt.

Darstellung von Tupelfolgen mit fester Länge k als eine einzige Zahl:

- $\langle n_1, \dots, n_k, 0 \rangle = c(n_1, c(n_2, (\dots, c(n_k, 0) \dots)))$
- wobei $c(x, y)$ eine Bijektion $(\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ ist
- Wir nehmen an, dass *left* und *right* existieren mit $\text{left}(c(x, y)) = x$ und $\text{right}(c(x, y)) = y$
- Genauere Details zu c , *left*, *right* werden später nochmal erörtert

Ackermannfunktion mit WHILE berechnen

Stack:

- Stack: Folge von Zahlen $n_1, \dots, n_k, 0$ sodass n_1 ganz oben liegt. Leerer Stack: 0 markiert Kellerboden.
- Stack im WHILE-Programm: Variable *stack*, die Zahl $\langle n_1, \dots, n_k, 0 \rangle = c(n_1, c(n_2, (\dots, c(n_k, 0) \dots)))$ speichert, und *stacksize*, die Größe des Stacks speichert

Ackermannfunktion mit WHILE berechnen

Stack:

- Stack: Folge von Zahlen $n_1, \dots, n_k, 0$ sodass n_1 ganz oben liegt. Leerer Stack: 0 markiert Kellerboden.
- Stack im WHILE-Programm: Variable *stack*, die Zahl $\langle n_1, \dots, n_k, 0 \rangle = c(n_1, c(n_2, (\dots, c(n_k, 0) \dots)))$ speichert, und *stacksize*, die Größe des Stacks speichert

Stack-Operationen:

- *push*($x, stack$) legt Zahl x oben auf den Stack.
WHILE-Programm dazu:
 $stack := c(x, stack); stacksize := stacksize + 1.$
- $x := pop(stack)$ entfernt das oberste Element vom Stack und setzt x auf dessen Wert.
WHILE-Programm: $x := left(stack); stack = right(stack);$
 $stacksize := stacksize - 1$
- $stacksize \neq 1$: WHILE-Programm dazu existiert

WHILE-Programm zur Berechnung von $a(x, y)$

```
stack := 0;
stacksize := 0;
push(x, stack);
push(y, stack);
WHILE size(stack)  $\neq$  1 DO
    y := pop(stack);
    x := pop(stack);
    IF x = 0
        THEN push(y + 1, stack);
        ELSE IF y = 0 THEN push(x - 1, stack); push(1, stack)
            ELSE push(x - 1, stack); push(x, stack), push(y - 1, stack);
        END
    END
END
result := pop(stack);
```

Lemma

Die folgenden Monotonie-Eigenschaften gelten für die Ackermannfunktion a :

- 1 $y < a(x, y)$
- 2 $a(x, y) < a(x, y + 1)$
- 3 $a(x, y + 1) \leq a(x + 1, y)$
- 4 $a(x, y) < a(x + 1, y)$
- 5 Falls $x \leq x'$ und $y \leq y'$, dann gilt auch $a(x, y) \leq a(x', y')$

Beweise im Skript (Induktion über x bzw. y , bzw. (x, y))

Maximale LOOP-berechenbare Zahlen

- Sei P ein LOOP-Programm
- Seien x_0, \dots, x_k die in P vorkommenden Variablen.
- Sei

$$f_P(n) = \max \left\{ \sum_{i=0}^k \rho'(x_i) \mid \sum_{i=0}^k \rho(x_i) \leq n, (\rho, P) \xrightarrow[\text{LOOP}]{}^* (\rho', \varepsilon) \right\}$$

- $f_P(n)$ ist die maximale Zahl, die als Summe aller Endbelegungen ρ' der Variablen x_0, \dots, x_k zustande kommt, über alle initialen Variablenbelegung ρ , die in der Summe der Werte $\rho(x_0), \dots, \rho(x_k)$ den Wert n nicht überschreiten.

Maximale LOOP-berechenbare Zahlen (2)

Satz

Für jedes LOOP-Programm P gibt es eine Konstante k , so dass $f_P(n) < a(k, n)$ für alle $n \in \mathbb{N}$.

Beweis: Normalisiere P zunächst:

- Ersetze Zuweisungen $x_i := x_j + c$ durch $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \dots x_i := x_i + 1}_{c-1 \text{ mal}}$.
- Für **LOOP** x_i **DO** Q **END** und x_i kommt in Q vor:
Ersetze **LOOP** x_i **DO** Q **END** durch $x'_i := x_i; \mathbf{LOOP} x'_i \mathbf{DO} Q \mathbf{END}; x'_i := 0$,
wobei x'_i eine neue Variable ist.
- Beides verändert f_P nicht!

Zeige Behauptung für normalisierte Programme.

Maximale LOOP-berechenbare Zahlen (3)

Zu zeigen: Für jedes normalisierte LOOP-Programm P gibt es eine Konstante k , so dass $f_P(n) < a(k, n)$ für alle $n \in \mathbb{N}$.

Maximale LOOP-berechenbare Zahlen (3)

Zu zeigen: Für jedes normalisierte LOOP-Programm P gibt es eine Konstante k , so dass $f_P(n) < a(k, n)$ für alle $n \in \mathbb{N}$.

Beweis durch strukturelle Induktion über normalisiertes Programm.

Maximale LOOP-berechenbare Zahlen (3)

Zu zeigen: Für jedes normalisierte LOOP-Programm P gibt es eine Konstante k , so dass $f_P(n) < a(k, n)$ für alle $n \in \mathbb{N}$.

Beweis durch strukturelle Induktion über normalisiertes Programm.

Fälle:

- 1 Zuweisung $x_i := x_j + c$ mit $c \in \mathbb{Z}$ und $c \leq 1$
- 2 Sequenz $P_1; P_2$
- 3 LOOP-Schleife **LOOP** x_i **DO** Q **END**,
wobei x_i nicht in Q vorkommt.

Maximale LOOP-berechenbare Zahlen (4)

Fall: Zuweisung $x_i := x_j + c$ mit $c \in \mathbb{Z}$ und $c \leq 1$

Dann gilt $f_P(n) \leq 2n + 1$, da im maximalen Fall:

- $\rho(x_k) = 0$ für $k \neq j$
- $\rho(x_j) = n$
- $c = 1$ und
- $\rho'(x_i) = n + 1$,
- $\rho'(x_j) = n$ und
- $\rho'(x_k) = 0$ für $k \neq j$ und $k \neq i$

Mit $a(2, y) = 2y + 3$ (s. Skript) folgt

$$f_P(n) \leq 2n + 1 < 2n + 3 = a(2, n)$$

D.h. die Aussage $f_P(n) \leq a(k, n)$ gilt mit $k = 2$.

Maximale LOOP-berechenbare Zahlen (5)

Fall: Sequenz $P_1; P_2$

Induktionsannahme: $f_{P_i}(n) < a(k_i, n)$ für $i = 1, 2$.

Es gilt:

$$\begin{aligned}f_P(n) &= f_{P_2}(f_{P_1}(n)) \\ &< a(k_2, a(k_1, n)) \\ &\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\}, n)) && (1) \\ &\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\} + 1, n + 1)) && (1) \\ &= a(\max\{k_1, k_2\} + 1, n + 1) && (\text{Definition von } a) \\ &\leq a(\max\{k_1, k_2\} + 2, n) && (2)\end{aligned}$$

(1) Monotonie: $x \leq x', y \leq y' \implies a(x, y) \leq a(x', y')$

(2) Monotonie: $a(x, y + 1) \leq a(x + 1, y)$

Daher gilt $f_P(n) < a(k, n)$ für $k = \max\{k_1, k_2\} + 2$.

Maximale LOOP-berechenbare Zahlen (6)

Fall: **LOOP** x_i **DO** Q **END** und x_i kommt nicht in Q vor

Induktionsannahme: $f_Q(n) < a(k_Q, n)$ für ein k_Q .

$f_P(n)$ berechnet Maximum. Sei ρ so, dass $f_P(n)$ maximal ist.

- Wenn $\rho(x_i) = 0$, dann ist $f_P(n) = n < a(0, n)$.
- Wenn $\rho(x_i) = 1$, dann ist $f_P(n) = f_Q(n) < a(k_Q, n)$
- Sei $\rho(x_i) \geq 2$

Maximale LOOP-berechenbare Zahlen (7)

Fall: **LOOP** x_i **DO** Q **END** und x_i kommt nicht in Q vor

- Sei $\rho(x_i) \geq 2$ Da x_i nicht in Q vorkommt, gilt:

$$\begin{aligned} f_P(n) &= \underbrace{f_Q(\dots (f_Q(n - \rho(x_i)) \dots))}_{\rho(x_i) \text{ mal}} + \rho(x_i) \quad (\text{da } x_i \notin Q, f_Q \text{ unabh. von } \rho(x_i)) \\ &\leq a(k_1, \dots, a(k_1, n - \rho(x_i)) \dots) \quad (\rho(x_i)\text{-mal gilt } < \text{ (für jedes } f_Q)). \\ &\hspace{20em} \text{Daher } \leq \text{ und } \rho(x_i) \text{ fällt weg)} \\ &< \underbrace{a(k_1, \dots, a(k_1, a(k_1 + 1, n - \rho(x_i)) \dots))}_{\rho(x_i) - 1 \text{ mal}} \quad (1) \\ &= a(k_1 + 1, n - \rho(x_i) + \rho(x_i) - 1) \quad (\text{Definition von } a) \\ &= a(k_1 + 1, n - 1) \leq a(k_1 + 1, n) \quad (2) \end{aligned}$$

(1) Monotonie: $a(x, y) < a(x + 1, y)$ (2) Monotonie: $a(x, y) < a(x, y + 1)$

Daher gilt $f_P(n) < a(k, n)$ für $k = k_1 + 1$

Theorem

Die Ackermannfunktion ist nicht LOOP-berechenbar.

Beweis:

- Annahme: a ist LOOP-berechenbar.
- Dann ist auch $f(x_1) = a(x_1, x_1)$ LOOP-berechenbar.
- Sei P LOOP-Programm, das f berechnet.
- Dann gilt $f(x_1) \leq f_P(x_1)$ (da $\rho'(x_0) = f(x_1)$ nach Ausführung von P)
- Es gibt Konstante k , sodass $f_P(n) < a(k, n)$
- Starte P mit $\rho = \{x_1 \mapsto k\}$.
- Dann gilt $f(k) \leq f_P(k) < a(k, k) = f(k)$
- Widerspruch!
- a ist nicht LOOP-berechenbar.

Theorem

Es gibt totale WHILE-berechenbare (bzw. GOTO-berechenbare, Turingberechenbare) Funktionen, die nicht LOOP-berechenbar sind.

Rekursive Funktionen

- Weiterer Formalismus zur Definition der Berechenbarkeit
- Primitiv rekursive Funktionen und
- μ -rekursive Funktionen

Wir werden sehen

Primitiv rekursive Funktionen entsprechen genau den LOOP-berechenbaren Funktionen

μ -rekursive Funktionen entsprechen genau den Turingberechenbaren (WHILE-, GOTO-berechenbaren) Funktionen

Primitiv rekursive Funktionen

Definition

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist **primitiv rekursiv**, wenn sie der folgenden induktiven Definition genügt:

- Jede **konstante Funktion** $f(x_1, \dots, x_k) = c \in \mathbb{N}$ ist primitiv rekursiv.
- Die **Projektionsfunktionen** $\pi_i^k(x_1, \dots, x_k) = x_i$ sind primitiv rekursiv.
- Die **Nachfolgerfunktion** $\text{succ}(x) = x + 1$ ist primitiv rekursiv.
- **Komposition / Einsetzung**: Wenn $g : \mathbb{N}^m \rightarrow \mathbb{N}$ und für $i = 1, \dots, m$: $h_i : \mathbb{N}^k \rightarrow \mathbb{N}$ primitiv rekursiv sind, dann ist auch f mit $f(x_1, \dots, x_k) = g(h_1(x_1, \dots, x_k), \dots, h_m(x_1, \dots, x_k))$ primitiv rekursiv.
- **Primitive Rekursion**: Wenn $g : \mathbb{N}^{k-1} \rightarrow \mathbb{N}$ und $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ primitiv rekursiv, dann ist auch f mit

$$f(x_1, \dots, x_k) = \begin{cases} g(x_2, \dots, x_k), & \text{wenn } x_1 = 0 \\ h(f(x_1 - 1, x_2, \dots, x_k), x_1 - 1, x_2, \dots, x_k), & \text{sonst} \end{cases}$$

primitiv rekursiv.

Komponenten eines Tupels entfernen / vertauschen / vervielfachen

Wenn $g : \mathbb{N}^4 \rightarrow \mathbb{N}$ primitiv rekursiv, dann ist auch $f : \mathbb{N}^3 \rightarrow \mathbb{N}$ mit

$$f(n_1, n_2, n_3) = g(n_2, n_3, n_3, n_2),$$

denn:

$$f(n_1, n_2, n_3) = g(\pi_2^3(n_1, n_2, n_3), \pi_3^3(n_1, n_2, n_3), \pi_3^3(n_1, n_2, n_3), \pi_2^3(n_1, n_2, n_3))$$

Konstruktionen (2)

Rekursion durch das i . Argument

Für $1 \leq i \leq k$ kann man

$$f(x_1, \dots, x_k) = \begin{cases} g(x_1, \dots, x_{i-1}, x_{i+1}, x_k), & \text{falls } x_i = 0 \\ h(f(x_1, \dots, x_{i-1}, x_i - 1, x_{i+1}, \dots, x_k), x_1, \dots, \\ \quad x_{i-1}, x_i - 1, x_{i+1}, \dots, x_k), & \text{sonst} \end{cases}$$

durch $f(x_1, \dots, x_k) = f'(x_i, x_1 \dots, x_{i-1}, x_{i+1}, \dots, x_k)$ darstellen, wobei

$$\begin{aligned} f'(x_1, \dots, x_k) &= \begin{cases} g'(x_2, \dots, x_k), & \text{falls } x_1 = 0 \\ h'(f'(x_1 - 1, x_2 \dots, x_k), x_1 - 1, x_2, \dots, x_k), & \text{sonst} \end{cases} \\ g'(x_1, \dots, x_{k-1}) &= g(x_2, \dots, x_i, x_1, x_{i+1}, \dots, x_k) \\ h'(x_1, \dots, x_{k+1}) &= h(x_1, x_2, \dots, x_i, x_1 - 1, x_{i+1}, \dots, x_k) \end{aligned}$$

Additionsfunktion

$add(x_1, x_2) = x_1 + x_2$ ist primitiv rekursiv:

$$add(x_1, x_2) = \begin{cases} x_2, & \text{falls } x_1 = 0 \\ succ(add(x_1 - 1, x_2)), & \text{sonst} \end{cases}$$

Bemerkung:

Die verwendeten Funktionen g und h aus der Definition der primitiv rekursiven Funktionen sind hier:

- $g = \pi_1^1$
- $h(x_1, x_2, x_3) = succ(\pi_1^3(x_1, x_2, x_3))$

Multiplikationsfunktion

$$\mathit{mult}(x_1, x_2) = \begin{cases} 0, & \text{falls } x_1 = 0 \\ \mathit{add}(\mathit{mult}(x_1 - 1, x_2), x_2), & \text{sonst} \end{cases}$$

Idee: x_1 -mal x_2 zu 0 addieren:

Beispiele (3)

Differenz

Allgemein $x_1 - x_2$ nicht primitiv rekursiv, undefinierter Fall
 $x_1 < x_2$ nicht darstellbar.

Angepasste Differenz

liefert 0 falls $x_1 < x_2$, ist primitiv rekursiv:

$$\text{sub}(x_1, x_2) = \begin{cases} x_1, & \text{falls } x_2 = 0 \\ \text{pred}(\text{sub}(x_1, x_2 - 1)) & \text{sonst} \end{cases}$$

wobei

$$\text{pred}(x_1) = \begin{cases} 0, & \text{falls } x_1 = 0 \\ x_1 - 1, & \text{sonst} \end{cases}$$

Wir wollen zeigen:

Primitiv rekursive Funktionen sind genau die LOOP-berechenbaren Funktionen

Benötigt:

- Darstellung der Variablenbelegung ρ als **eine einzige Zahl**, um sie der primitiv rekursiven Funktion als Argument zu übergeben.
- D.h. eindeutige Darstellung eines Tupels natürlicher Zahlen als eine einzige Zahl.
- Operationen zum Konvertieren in beide Richtungen

Eine solches Verfahren nennt man auch „Gödelisierung“ (nach Kurt Gödel)

Gödelisierung (1)

- Tupel von natürlichen Zahlen (x_0, \dots, x_k) bijektiv in die natürlichen Zahlen abbilden
- Für festes k
- mit **primitiv rekursiven** Funktionen

$$c(x, y) = \binom{x + y + 1}{2} + x$$

Werte von $c(x, y)$ für $x, y \in \{0, \dots, 5\}$:

$y \backslash x$	0	1	2	3	4	5
0	0	1	3	6	10	15
1	2	4	7	11	16	22
2	5	8	12	17	23	30
3	9	13	18	24	31	39
4	14	19	25	32	40	49
5	20	26	33	41	50	60

Funktion c ist primitiv rekursiv, da

$$\binom{0}{2} = 0 \text{ und } \binom{n+1}{2} = \binom{n}{2} + n$$

Für $k+1$ -Tupel definieren wir:

$$\langle x_0, \dots, x_k \rangle = c(x_0, c(x_1, \dots, c(x_k, 0) \dots))$$

Beachte: $\langle \cdot \rangle$ ist primitiv rekursiv

(da c primitiv rekursiv und Komposition primitiv rekursiv)

Gödelisierung (3)

Rückgewinnung der Komponenten:

Seien *left* und *right* Funktionen mit

- $left(c(x, y)) = x$ und
- $right(c(x, y)) = y$.

Im Skript wird gezeigt:

- *left* und *right* existieren
- *left* und *right* sind primitiv rekursiv

Gödelisierung (5)

Zugriff auf beliebige Komponenten:

Programmiere $d_i(\langle x_0, \dots, x_k \rangle) = x_i$ durch:

$$\begin{aligned}d_0(x) &= \mathit{left}(x) \\d_1(x) &= \mathit{left}(\mathit{right}(x)) \\d_i(x) &= \mathit{left}(\underbrace{\mathit{right}(\mathit{right} \dots \mathit{right}(x) \dots)}_{i \text{ mal}})\end{aligned}$$

Damit sind auch die d_i -Funktionen primitiv rekursiv.

Von LOOP-Programm berechnete Funktion

- Sei P ein LOOP-Programm, ρ eine Variablenbelegung mit

$$(\rho, P) \xrightarrow[\text{LOOP}]{}^* (\rho', \varepsilon)$$

- Seien x_0, x_1, \dots, x_n alle vom Programm P verwendeten Variablen (auch solche, die nicht in ρ vorkommen)
- Die von P berechnete Funktion:

$$g_P(\langle x_0, \dots, x_n \rangle) = \langle x'_0, \dots, x'_n \rangle$$

(wobei $x'_i = \rho'(x_i)$)

Lemma

Für jedes LOOP-Programm P ist die zugehörige Funktion g_P primitiv rekursiv.

Beweis: Strukturelle Induktion über jedes Teilprogramm Q und die zugehörige Funktion g_Q .

- Basis: Q ist Zuweisung $x_i = x_j \pm c$.
Für g_Q muss gelten:

$$g_Q(\langle m_0, \dots, m_n \rangle) = \langle m_0, \dots, m_{i-1}, m_j \pm c, m_i, \dots, m_n \rangle$$

Primitiv rekursive Implementierung:

$$g_Q(x) = \langle d_0(x), \dots, d_{i-1}(x), d_j(x) + c, d_{i+1}(x), \dots, d_n(x) \rangle$$

LOOP-Programme berechnen primitiv rekursive Funktionen (2)

Induktionsschritt:

- Q ist eine Sequenz $Q_1; Q_2$.

Induktionshypothese: primitiv rekursive Funktionen g_{Q_1}, g_{Q_2} .

Funktion $g_Q(x) = g_{Q_2}(g_{Q_1}(x))$ ist primitiv rekursiv.

- Q ist **LOOP** x_i **DO** P **END**.

Induktionshypothese liefert primitiv rekursive Funktion g_P .

Konstruktion von g_Q : x_i -mal wird g_P angewendet.

$$g_Q(x) = \text{run}(d_i(x), x)$$
$$\text{run}(n, x) = \begin{cases} x, & \text{falls } n = 0 \\ g_P(\text{run}(n-1, x)) & \text{sonst} \end{cases}$$

run ist primitiv rekursiv. Damit auch g_Q primitiv rekursiv.

Satz 11.1.6

Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

Beweis:

- Sei $f : \mathbb{N}^k \rightarrow \mathbb{N}$ LOOP-berechenbar.
- Sei P LOOP-Programm mit $\rho = \{x_1 \mapsto n_1, \dots, x_k \mapsto n_k\}$,
 $(\rho, P) \xrightarrow[\text{LOOP}]{}^* (\rho', \varepsilon)$ und $\rho'(x_0) = f(n_1, \dots, n_k)$
- Es gilt $f(n_1, \dots, n_k) = d_0(g_P(\langle 0, n_1, \dots, n_k, 0, \dots, 0 \rangle))$.
- Da $\langle \cdot \rangle$, d_0 und g_P primitiv rekursiv sind, ist auch f primitiv rekursiv.

Satz 11.1.7

Jede primitiv rekursive Funktion ist LOOP-berechenbar.

Beweis: Induktion über Struktur der primitiv rekursiven Funktion:

- Wenn $f(x) = c$, $f = succ$, oder $f = \pi_n^k$, dann gibt es auch LOOP-Programm dazu.
- Wenn $f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$:
Induktionshypothese liefert LOOP-Programme P_h und $P_{g,1}, \dots, P_{g,n}$, die h, g_1, \dots, g_n berechnen.

Konstruiere Programm für f nach dem **Schema**:

$$y_1 := g_1(x_1, \dots, x_k); \dots; y_n := g_n(x_1, \dots, x_k); x_0 := h(y_1, \dots, y_n)$$

Genauer: $P_{g,1}, \dots, P_{g,n}, P_h$ abändern, so dass sie auf disjunkten Variablenmengen arbeiten, entsprechende Variableninhalte für x_1, \dots, x_k verdoppeln.

- ...

Primitiv rekursive Funktionen sind LOOP-berechenbar (2)

- ...
- f ist primitiv rekursiv:

$$f(x_1, \dots, x_k) = \begin{cases} g(x_2, \dots, x_k) & \text{wenn } x_1 = 0 \\ h(f(x_1 - 1, x_2, \dots, x_k), x_1 - 1, x_2, \dots, x_k) & \text{sonst} \end{cases}$$

wobei $g : \mathbb{N}^{k-1} \rightarrow \mathbb{N}$ und $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ primitiv rekursiv sind.
Induktionshypothese liefert LOOP-Programme, die g, h berechnen.
Konstruiere LOOP-Programm für f nach dem **Schema**

```
 $y := 0;$   
 $x_0 := g(x_2, \dots, x_k);$   
LOOP  $x_1$  DO  $y := y + 1; x_0 := h(x_0, y, x_2, \dots, x_k)$  END
```

Theorem 11.1.8

Die primitiv rekursiven Funktionen sind genau die LOOP-berechenbaren Funktionen.

Definition μ -Operator

Sei $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ eine (partielle oder totale) Funktion. Dann ist $(\mu h) : \mathbb{N}^k \rightarrow \mathbb{N}$ definiert als

$$(\mu h)(x_1, \dots, x_k) = \begin{cases} n & \text{falls } h(n, x_1, \dots, x_k) = 0 \text{ und f\u00fcr} \\ & \text{alle } m < n: h(m, x_1, \dots, x_k) \text{ ist} \\ & \text{definiert und } h(m, x_1, \dots, x_k) > 0. \\ \text{undefiniert,} & \text{sonst} \end{cases}$$

- μ -Operator „sucht“ nach einer kleinsten Nullstelle von h .
- Wenn diese nicht existiert (entweder da h keine Nullstelle hat, oder da h undefiniert ist f\u00fcr Werte, die kleiner als die Nullstelle sind), dann ist auch der μ -Operator angewendet auf h undefiniert.

Definition

Die Menge aller μ -rekursiven Funktionen sei die kleinste Menge, so dass gilt:

- Jede konstante Funktion $f(x_1, \dots, x_k) = c \in \mathbb{N}$ ist μ -rekursiv.
- Die Projektionsfunktionen $\pi_i^k(x_1, \dots, x_k) = x_i$ sind μ -rekursiv.
- Die Nachfolgerfunktion $\text{succ}(x) = x + 1$ ist μ -rekursiv.
- **Komposition / Einsetzung:** Wenn $g : \mathbb{N}^m \rightarrow \mathbb{N}$ und für $i = 1, \dots, m$: $h_i : \mathbb{N}^k \rightarrow \mathbb{N}$ μ -rekursiv sind, dann ist auch f mit $f(x_1, \dots, x_k) = g(h_1(x_1, \dots, x_k), \dots, h_m(x_1, \dots, x_k))$ μ -rekursiv.
- **Rekursion:** Wenn $g : \mathbb{N}^{k-1} \rightarrow \mathbb{N}$ und $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ μ -rekursiv, dann ist

$$f(x_1, \dots, x_k) = \begin{cases} g(x_2, \dots, x_k), & \text{wenn } x_1 = 0 \\ h(f(x_1 - 1, x_2, \dots, x_k), x_1 - 1, x_2, \dots, x_k), & \text{sonst} \end{cases}$$

auch μ -rekursiv.

- **μ -Operator:** Wenn $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ μ -rekursiv, dann auch $f = \mu h$ μ -rekursiv.

WHILE-berechenbare Funktionen sind μ -rekursiv

Satz 11.2.3

Jede WHILE-berechenbare Funktion ist μ -rekursiv.

Beweis: Analog zu Satz 11.1.6 über Struktur des WHILE-Programms P

Neuer Fall: P ist **WHILE** $x_i \neq 0$ **DO** Q **END**

Induktionshypothese liefert μ -rekursive Funktion g_Q für Q .

Konstruiere:

$$\begin{aligned}g_P(x) &= \text{run}(\mu(\text{run}_i)(x), x) \\ \text{run}_i(n, x) &= d_i(\text{run}(n, x)) \\ \text{run}(n, x) &= \begin{cases} x, & \text{falls } n = 0 \\ g_Q(\text{run}(n-1, x)) & \text{sonst} \end{cases}\end{aligned}$$

- $\text{run}(n, x)$ führt n -mal g_Q aus.
- $\mu(\text{run}_i)(x)$ berechnet, wie oft die Schleife minimal durchlaufen werden muss, bis x_i den Wert 0 hat .
- Divergiert die Schleife, so ist $\mu(\text{run}_i)$ undefiniert und g_P undefiniert.

Satz 11.2.4

Jede μ -rekursive Funktion ist WHILE-berechenbar.

Beweis: Analog zu Satz 11.1.7 über die Struktur der Funktion.

Neuer Fall: $f = \mu h$ für eine μ -rek. Funktion h .

Induktionshypothese liefert WHILE-Programm P_h , das h berechnet.

Konstruiere P_f nach dem **Schema**:

$x_0 := 0;$

$y := h(0, x_1, \dots, x_n);$

WHILE $y \neq 0$ **DO**

$x_0 := x_0 + 1; y := h(x_0, \dots, x_n)$

END

- **WHILE**-Schleife berechnet minimalen Wert für $h(x_0, \dots, x_n) = 0$
- Wenn dieser nicht existiert, terminiert die Schleife nicht.
- Entspricht der Berechnung von μh .

Theorem 11.2.5

Die μ -rekursiven Funktionen entsprechen genau den WHILE-berechenbaren (und damit auch den GOTO- und Turingberechenbaren) Funktionen.

Überblick: Berechenbarkeitsformalismen

