

Berechenbarkeit und Komplexität

3. Vorlesung



Prof. Dr. Dietrich Kuske



FG Automaten und Logik, TU Ilmenau

Sommersemester 2024

Argument (K^-) gegen die Loop-Vermutung: Die Ackermann-Funktion

1. Teil der heutigen Vorlesung:

Wilhelm Ackermann (1886-1962) konstruierte 1928 eine Funktion, die intuitiv berechenbar ist, und bewies, daß sie nicht primitiv-rekursiv (d.h. loop-berechenbar) ist. Damit waren die Loop- und die Hilbertsche Vermutung zu verwerfen.

Rózsa Péter (1905-1977) vereinfachte 1955 Ackermanns Funktion zur „Ackermann-Péter-Funktion“ (die heute „Ackermann-Funktion“ heißt).

Wir werden im 2. Teil der Vorlesung dann zeigen, daß auch keine Erweiterung der Loop-Programme geeignet ist, die intuitiv berechenbaren Funktionen zu beschreiben (solange wir nur totale Funktionen betrachten).

Grundidee: Die folgenden Funktionen sind loop-berechenbar:

$$0. (m, n) \mapsto m + 1$$

$$1. (m, n) \mapsto m + n = m + \underbrace{1 + 1 \cdots + 1}_{n \text{ mal}}$$

$$2. (m, n) \mapsto m \cdot n = \underbrace{m + m \cdots + m}_{n \text{ mal}}$$

$$3. (m, n) \mapsto m^n = \underbrace{m \cdot m \cdots \cdot m}_{n \text{ mal}}$$

$$4. (m, n) \mapsto m^{m^{\cdots^m}} \left. \vphantom{m^{m^{\cdots^m}}} \right\} n \text{ mal}$$

5. usw. usf.

Loop-berechenbare Funktionen können also sehr schnell wachsen, die Ackermann-Funktion wächst aber noch schneller!

Konstruktion

- Für $f: \mathbb{N} \rightarrow \mathbb{N}$ sei $F(f) = g: \mathbb{N} \rightarrow \mathbb{N}$ definiert durch

$$g(y) = \begin{cases} f(1) & \text{falls } y = 0 \\ f(g(y-1)) & \text{falls } y > 0 \end{cases}$$

Also ist $F: \mathbb{N}^{\mathbb{N}} \rightarrow \mathbb{N}^{\mathbb{N}}$ Funktion, die numerische Funktionen auf numerische Funktionen abbildet.

Wir definieren nun eine Folge von Funktionen $\text{ack}_x: \mathbb{N} \rightarrow \mathbb{N}$ für $x \in \mathbb{N}$:

- $\text{ack}_0: \mathbb{N} \rightarrow \mathbb{N}: y \mapsto y + 1$
- $\text{ack}_{x+1} = F(\text{ack}_x)$, d.h.

$$\text{ack}_{x+1}(y) = \begin{cases} \text{ack}_x(1) & \text{falls } y = 0 \\ \text{ack}_x(\text{ack}_{x+1}(y-1)) & \text{falls } y > 0 \end{cases}$$

	0	1	2	3	y
ack_0	1	2	3	4	$y + 1$
ack_1	2	3	4	5	$y + 2$
ack_2	3	5	7	9	$2y + 3$
ack_3	5	13	29	61	$2^{y+3} - 3$
ack_4	13	65533	$2^{65536} - 3$	$2^{2^{65536} - 6} - 3$	
ack_5	65533		

$$\text{ack}_4(y) = 2^{2^{\dots^2}} \left. \vphantom{2^{2^{\dots^2}}} \right\} y+3 \text{ mal} - 3$$

Definition

Die Funktion $\text{ack}: \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $\text{ack}(x, y) = \text{ack}_x(y)$ heißt **Ackermann-Funktion**.

Lemma

Ist $x > 0$ und $y \in \mathbb{N}$, so gilt $\text{ack}_x(y) = (\text{ack}_{x-1})^{y+1}(1)$.

Beweis siehe z.B. Uwe Schöning: Theoretische Informatik - kurzgefasst. Spektrum 2001, Seiten 116 ff.

Behauptung

Die Ackermann-Funktion ack ist intuitiv berechenbar.

Begründung:

der folgende rekursive Algorithmus tut es:

```
function A(x,y: integer): integer
  if x = 0 then return y + 1;
  h := 1;
  for (i := 1, i <= y + 1, i++) do
    h := A(x - 1, h);
  return h;
```

Monotonielemma

Für alle $x, y, x', y' \in \mathbb{N}$ mit $x \leq x'$ und $y \leq y'$ gilt $\text{ack}_x(y) \leq \text{ack}_{x'}(y')$.

Beweis siehe z.B. Uwe Schöning: Theoretische Informatik - kurzgefasst. Spektrum 2001, Seiten 116 ff.

Definition

Sei P Loop-Programm mit Variablen x_1, x_2, \dots, x_ℓ .

Für Anfangswerte $(n_i)_{1 \leq i \leq \ell}$ seien $(n'_i)_{1 \leq i \leq \ell}$ die Werte der Variablen bei Programmende.

$$f_P: \mathbb{N} \rightarrow \mathbb{N}: n \mapsto \max \left\{ \sum_{1 \leq i \leq \ell} n'_i \mid \sum_{1 \leq i \leq \ell} n_i \leq n \right\}$$

Beschränkungslemma

Sei P ein Loop-Programm. Es gibt $k \in \mathbb{N}$, so daß für alle $n \in \mathbb{N}$ gilt: $f_P(n) \leq \text{ack}_k(n)$.

Beweis: Induktion über Aufbau von P

- P hat Form $x_i := c$ oder $x_i := x_j \pm c$ mit $c \in \{0, 1\}$
dann gilt $f_P(n) \leq n + n + 1 < 2n + 3 = \text{ack}_2(n)$. Also setze $k = 2$.
- P hat Form $P_1; P_2$ und es gibt $k_1, k_2 \in \mathbb{N}$, so daß für alle $n \in \mathbb{N}$ gilt:
 $f_{P_1}(n) \leq \text{ack}_{k_1}(n)$ und $f_{P_2}(n) \leq \text{ack}_{k_2}(n)$. Setze $k = \max(k_1, k_2) + 2$.
Dann gilt

$$\begin{aligned}
 f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \leq \text{ack}_{k_2}(f_{P_1}(n)) \\
 &\leq \text{ack}_{k_2}(\text{ack}_{k_1}(n)) \text{ nach Monotonielemma} \\
 &\leq \text{ack}_{k-2}(\text{ack}_{k-1}(n)) \text{ nach Monotonielemma} \\
 &= \text{ack}_{k-1}(n+1) \\
 &\leq \text{ack}_k(n) \text{ nach Schönig, Lemma C in Kap. 2.5}
 \end{aligned}$$

- P hat Form „loop x_i do Q end“ und es gibt $h \in \mathbb{N}$, so daß für alle $n \in \mathbb{N}$ gilt: $f_Q(n) \leq \text{ack}_h(n)$. Um $f_P(n)$ zu berechnen, interessieren wir uns für die Abläufe von P , in denen insbesondere die Schleife höchstens n mal durchlaufen wird.

$$\begin{aligned}
 f_P(n) &\leq f_Q^n(n) \leq \text{ack}_h^n(n) \\
 &= \text{ack}_h^{n-1}(\text{ack}_h(n)) \leq \text{ack}_h^{n-1}(\text{ack}_{h+1}(n)) \\
 &\hspace{20em} \text{nach Monotonielemma} \\
 &= \text{ack}_h^{n-2}(\text{ack}_h(\text{ack}_{h+1}(n))) = \text{ack}_h^{n-2}(\text{ack}_{h+1}(n+1)) \\
 &\quad \vdots \\
 &= \text{ack}_{h+1}(2n-1) \\
 &\leq \text{ack}_{h+1}(\text{ack}_{h+2}(n-1)) \text{ nach Monotonielemma} \\
 &= \text{ack}_{h+2}(n)
 \end{aligned}$$

Mit $k = h + 2$ gilt also $f_P(n) \leq \text{ack}_k(n)$ für alle $n \in \mathbb{N}$



Satz (vgl. Behauptung auf Folie 3.2)

Die Ackermann-Funktion ist nicht loop-berechenbar.

Beweis: Indirekt: Angenommen, P wäre Loop-Programm, das ack berechnet.

betrachte das Loop-Programm $Q = (x_2 := x_1; P; x_1 := x_1 + 1)$

es berechnet die Funktion $\mathbb{N} \rightarrow \mathbb{N}: n \mapsto \text{ack}(n, n) + 1 = \text{ack}_n(n) + 1$.

nach Beschränkungslemma existiert $k \in \mathbb{N}$ mit $f_Q(m) \leq \text{ack}_k(m)$ f.a. $m \in \mathbb{N}$

damit: $\text{ack}_k(k) < \text{ack}_k(k) + 1 = f_Q(k) \leq \text{ack}_k(k)$, ein Widerspruch. \square

Veranschaulichung: trage in eine Tabelle die Funktionen ack_k für $k \in \mathbb{N}$ ein

n	$\text{ack}_0(n)$	$\text{ack}_1(n)$	$\text{ack}_2(n)$	$\text{ack}_3(n)$	$\text{ack}_4(n) \dots$
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	$2^{2^{65536} - 6} - 3$
\vdots					

Veranschaulichung: trage in eine Tabelle die Funktionen ack_k für $k \in \mathbb{N}$ ein

Alle Zahlen auf der **Diagonale** um eins erhöhen. Dadurch erhält man f .

n	$\text{ack}_0(n)$	$\text{ack}_1(n)$	$\text{ack}_2(n)$	$\text{ack}_3(n)$	$\text{ack}_4(n) \dots$
0	1 2	2	3	5	13
1	2	3 4	5	13	65533
2	3	4	7 8	29	$2^{65536} - 3$
3	4	5	9	61 62	$2^{2^{65536} - 6} - 3$
⋮					

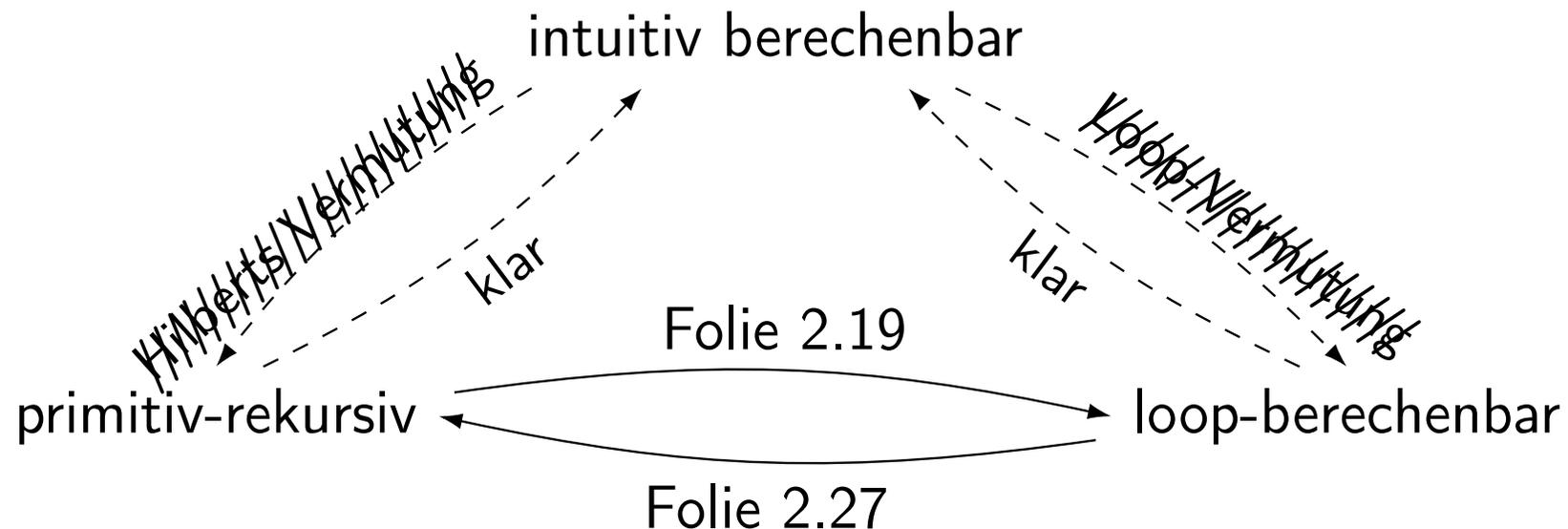
Veranschaulichung: trage in eine Tabelle die Funktionen ack_k für $k \in \mathbb{N}$ ein

Die Funktion f kann aber aufgrund dieser Konstruktion mit keiner der anderen Funktionen übereinstimmen.

n	$\text{ack}_0(n)$	$\text{ack}_1(n)$	$\text{ack}_2(n)$	$\text{ack}_3(n)$	$\text{ack}_4(n) \dots$
0	1 2	2	3	5	13
1	2	3 4	5	13	65533
2	3	4	7 8	29	$2^{65536} - 3$
3	4	5	9	61 62	$2^{2^{65536} - 6} - 3$
⋮					

Dies ist schon wieder ein Diagonalisierungsbeweis (vgl. 6. Vorlesung Automaten und Formale Sprachen und 1. Vorlesung Berechenbarkeit und Komplexität).

Zusammenfassung Loop-Programme



- (U⁺) zwei äquivalente Vorschläge, den Begriff „intuitiv berechenbar“ zu formalisieren: loop-berechenbar und primitiv-rekursiv
- (K⁺) viele Funktionen sind loop-berechenbar bzw. primitiv-rekursiv
- (A⁺) die beiden Klassen erfüllen viele Abschlußeigenschaften
- (K⁻) es gibt intuitiv berechenbare Funktionen, die nicht loop-berechenbar sind.

Was nun?

Wir sollten die „Programmiersprache“ der Loop-Programme so erweitern, daß alle intuitiv berechenbaren Funktionen darstellbar sind.

Idealerweise erhalten wir eine „Programmiersprache“ $L \subseteq \Gamma^*$ für ein gewisses Alphabet Γ mit folgenden Eigenschaften:

1. Jedes „Programm“ P aus L definiert eine Funktion $\llbracket P \rrbracket : \mathbb{N} \rightarrow \mathbb{N}$.
2. Aus dem „Programmtext“ P und einer natürlichen Zahl n kann $\llbracket P \rrbracket (n)$ berechnet werden, d.h. die folgende Abbildung ist intuitiv berechenbar:

$$I: L \times \mathbb{N} \rightarrow \mathbb{N}: (P, n) \mapsto \llbracket P \rrbracket (n)$$

3. Einem Wort aus Γ^* „kann man ansehen“, daß es ein „Programmtext“ aus L ist, d.h. die folgende Funktion ist intuitiv berechenbar:

$$\chi_L: \Gamma^* \rightarrow \{\top, \perp\}: P \mapsto \begin{cases} \top & P \in L \\ \perp & \text{sonst} \end{cases}$$

4. Zu jeder intuitiv berechenbaren Fkt. $f: \mathbb{N} \rightarrow \mathbb{N}$ ex. $P \in L$ mit $\llbracket P \rrbracket = f$.

Beispiel

Die „Programmiersprachen“ der Loop-Programme und die der rek-Programme haben die ersten drei Eigenschaften, aber nicht die 4. Eigenschaft (denn die Ackermann-Funktion ist weder loop- noch rek-berechenbar).

Wir werden jetzt zeigen, daß die „ideale Programmiersprache“ nicht existiert, genauer:

Behauptung

Sei $L \subseteq \Gamma^*$ und $I: L \times \mathbb{N} \rightarrow \mathbb{N}$. Gelten die ersten drei Eigenschaften, so gilt die vierte Eigenschaft nicht.

Begründung: indirekt, sei also L eine Sprache, die die Eigenschaften 1-4 hat.

Sei w_0, w_1, \dots die längen-lexikographische Aufzählung aller Wörter aus Γ^* (d.h. die kurzen Wörter kommen erst, gleichlange werden lexikographisch geordnet).

Die Funktion p , die einer natürlichen Zahl $n > 0$ das n -te Element von L in dieser Aufzählung zuordnet, ist intuitiv berechenbar:

```
 $m := 0; i := -1;$   
while  $m < n$  do  
   $i := i + 1;$   
  if  $\chi_L(w_i) = \top$  (d.h.  $w_i \in L$ ) then  $m := m + 1;$   
endwhile;  
return  $w_i$ 
```

Betrachte die folgende Funktion:

$$f: \mathbb{N} \rightarrow \mathbb{N}: n \mapsto \llbracket p(n) \rrbracket (n) + 1$$

Wegen $\llbracket p(n) \rrbracket (n) = l(p(n), n)$ ist sie aufgrund der 2. Eigenschaft intuitiv berechenbar.

Aufgrund der 4. Eigenschaft existiert ein $P \in L$ mit $\llbracket P \rrbracket = f$. Da $(w_i)_{i \in \mathbb{N}}$ alle Programmtexte aus L enthält, existiert eine natürliche Zahl $m > 0$ mit $P = p(m)$.

Dann gilt

$$f(m) = \llbracket p(m) \rrbracket (m) + 1 \neq \llbracket p(m) \rrbracket (m) = \llbracket P \rrbracket (m) = f(m),$$

ein Widerspruch. □

Veranschaulichung: trage in eine Tabelle die Funktionen $\llbracket p(n) \rrbracket$ für $n \in \mathbb{N}$ ein

Zum Beispiel:

n	$\llbracket p(0) \rrbracket (n)$	$\llbracket p(1) \rrbracket (n)$	$\llbracket p(2) \rrbracket (n)$	$\llbracket p(3) \rrbracket (n)$	$\llbracket p(4) \rrbracket (n) \dots$
0	7	20	33	0	12
1	12	33	94	2	17
2	99	101	17	11	22
3	2	0	14	99	42
4	17	5	77	7	11
\vdots					

Veranschaulichung: trage in eine Tabelle die Funktionen $\llbracket p(n) \rrbracket$ für $n \in \mathbb{N}$ ein

Zum Beispiel: Alle Zahlen auf der **Diagonale** um eins erhöhen. Dadurch erhält man f .

n	$\llbracket p(0) \rrbracket (n)$	$\llbracket p(1) \rrbracket (n)$	$\llbracket p(2) \rrbracket (n)$	$\llbracket p(3) \rrbracket (n)$	$\llbracket p(4) \rrbracket (n) \dots$
0	7 8	20	33	0	12
1	12	33 34	94	2	17
2	99	101	17 18	11	22
3	2	0	14	99 100	42
4	17	5	77	7	11 12
⋮					

Veranschaulichung: trage in eine Tabelle die Funktionen $\llbracket p(n) \rrbracket$ für $n \in \mathbb{N}$ ein

Die Funktion f kann aber aufgrund dieser Konstruktion mit keiner der anderen Funktionen übereinstimmen.

n	$\llbracket p(0) \rrbracket (n)$	$\llbracket p(1) \rrbracket (n)$	$\llbracket p(2) \rrbracket (n)$	$\llbracket p(3) \rrbracket (n)$	$\llbracket p(4) \rrbracket (n) \dots$
0	7 8	20	33	0	12
1	12	33 34	94	2	17
2	99	101	17 18	11	22
3	2	0	14	99 100	42
4	17	5	77	7	11 12
⋮					

Dies ist schon wieder ein Diagonalisierungsbeweis (vgl. ...).

Ausweg: partielle Funktionen

Wir werden sehen, daß die obige Argumentation nicht funktioniert, wenn wir **partielle** Funktionen als intuitiv berechenbar zulassen. Damit könnte es sein, daß es eine „Programmiersprache“ für die intuitiv berechenbaren partiellen Funktionen gibt - zumindest können wir diese Hoffnung nicht mit obiger Argumentation widerlegen.

Definition

Seien $k \in \mathbb{N}$ und $D \subseteq \mathbb{N}^k$. Eine Funktion $f: D \rightarrow \mathbb{N}$ heißt **partielle Funktion** von \mathbb{N}^k nach \mathbb{N} . Wir schreiben hierfür $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$.

Intuitiver Berechenbarkeitsbegriff - Erweiterung

Eine partielle Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ ist **intuitiv berechenbar**, wenn es einen Algorithmus gibt, der f berechnet, d.h.

- das Verfahren erhält $(n_1, \dots, n_k) \in \mathbb{N}^k$ als Eingabe (nicht unbedingt aus dem Definitionsbereich),
- terminiert nach endlich vielen Schritten genau dann, wenn $f(n_1, \dots, n_k)$ definiert ist
- und gibt in diesem Fall $f(n_1, \dots, n_k)$ aus.

Wir nehmen zusätzlich an, daß es ein Alphabet Γ gibt, so daß jeder Algorithmus als Wort über Γ beschrieben werden kann.

Idealerweise erhalten wir eine „Programmiersprache“ $L \subseteq \Gamma^*$ für ein gewisses Alphabet Γ mit folgenden Eigenschaften:

1. Jedes „Programm“ P aus L definiert eine partielle Funktion $\llbracket P \rrbracket : \mathbb{N} \rightarrow \mathbb{N}$.
2. Aus dem „Programmtext“ P und einer natürlichen Zahl n kann $\llbracket P \rrbracket (n)$ berechnet werden, d.h. die Abbildung $I : L \times \mathbb{N} \rightarrow \mathbb{N}$ mit
 - $I(P, n)$ definiert gdw. $\llbracket P \rrbracket (n)$ definiert und
 - in diesem Fall gilt $I(P, n) = \llbracket P \rrbracket (n)$
 ist intuitiv berechenbar.
3. Die folgende Funktion ist intuitiv berechenbar:

$$\chi_L : \Gamma^* \rightarrow \{\top, \perp\} : P \mapsto \begin{cases} \top & P \in L \\ \perp & \text{sonst} \end{cases}$$

4. Zu jeder intuitiv berechenbaren partiellen Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ existiert $P \in L$ mit $\llbracket P \rrbracket = f$.

Wir betrachten die Begründung der Behauptung auf Folie 3.13. Die für den Beweis wesentliche Funktion f wird jetzt partiell (denn $\llbracket p(n) \rrbracket (n)$ könnte ja undefiniert sein), d.h. $f: \mathbb{N} \rightarrow \mathbb{N}$ mit

- $f(n)$ ist genau dann definiert, wenn $\llbracket p(n) \rrbracket (n)$ definiert ist und
- in diesem Fall gilt $f(n) = \llbracket p(n) \rrbracket (n) + 1$.

Dann erhält man wieder eine natürliche Zahl m mit $P = p(m)$ bzw. $\llbracket p(m) \rrbracket = f$. Jetzt müssen aber zwei Fälle unterschieden werden:

- $f(m)$ ist definiert: dann folgt der Widerspruch wie gehabt.
- $f(m)$ ist undefiniert: dann kann der Widerspruch nicht gezeigt werden.

Zusammenfassung 3. Vorlesung

in dieser Vorlesung neu

- zwei große Enttäuschungen:
 - Ackermann-Funktion intuitiv berechenbar, aber nicht loop-berechenbar
 - es gibt keine „Programmiersprache“ für die totalen intuitiv berechenbaren Funktionen
- wir müssen auch partielle Funktionen betrachten

kommende Vorlesung

- drei „Programmiersprachen“, die
 - die Ackermann-Funktion zu berechnen erlauben
 - und partielle Funktionen beschreiben (da deren Programme u.U. nicht terminieren)
- Argumente der Form (K^+) , (A^+) und (U^+) dafür, daß diese „Programmiersprachen“ den Begriff der intuitiven Berechenbarkeit korrekt beschreiben.