Berechenbarkeit und Komplexität
3. Vorlesung

Prof. Dr. Dietrich Kuske

FG Automaten und Logik, TU Ilmenau

Sommersemester 2023

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.

Rósza 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. $(m, n) \mapsto m^{m^{\cdot^{\cdot^{\cdot^{m}}}}}\Big\}$n mal
5. usw. usf.

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

Konstruktion

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

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

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

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

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

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

	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	$\ldots$	$\ldots$		

$$\mathrm{ack}_4(y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\}\text{y+3 mal} - 3$$

Definition

Die Funktion $\mathrm{ack}\colon \mathbb{N}^2 \to \mathbb{N}$ mit $\mathrm{ack}(x,y) = \mathrm{ack}_x(y)$ heißt Ackermann-Funktion.

Lemma

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

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 $\mathrm{ack}_x(y) \leq \mathrm{ack}_{x'}(y')$.

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

Definition

Sei P Loop-Programm mit Variablen $x_1, x_2, \ldots, x_\ell$.
Für Anfangswerte $(n_i)_{1 \leq i \leq \ell}$ seien $(n'_i)_{1 \leq i \leq \ell}$ die Werte der Variablen bei Programmende.

$$f_P : \mathbb{N} \to \mathbb{N} : n \mapsto \max\left\{ \sum_{1 \leq i \leq \ell} n'_i \;\middle|\; \sum_{1 \leq i \leq \ell} n_i \leq n \right\}$$

Beschränkungslemma

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

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 $f_p(n) \leq n + n + 1 < 2n + 3 = \mathrm{ack}_2(n)$. 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: $f_{P_1}(n) \leq \mathrm{ack}_{k_1}(n)$ und $f_{P_2}(n) \leq \mathrm{ack}_{k_2}(n)$. Setze $k = \max(k_1, k_2) + 2$.
  Dann gilt

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

- 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: $f_Q(n) \leq \mathrm{ack}_h(n)$. Um $f_P(n)$ zu berechnen, interessieren wir uns für die Abläufe von P, in denen insbesondere die Schleife höchstens n mal durchlaufen wird.

$$
\begin{aligned}
f_P(n) &\leq f_Q^n(n) \leq \mathrm{ack}_h^n(n) \\
&= \mathrm{ack}_h^{n-1}\big(\mathrm{ack}_h(n)\big) \leq \mathrm{ack}_h^{n-1}\big(\mathrm{ack}_{h+1}(n)\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{nach Monotonielemma} \\
&= \mathrm{ack}_h^{n-2}\Big(\mathrm{ack}_h\big(\mathrm{ack}_{h+1}(n)\big)\Big) = \mathrm{ack}_h^{n-2}\big(\mathrm{ack}_{h+1}(n+1)\big) \\
&\;\;\vdots \\
&= \mathrm{ack}_{h+1}(2n-1) \\
&\leq \mathrm{ack}_{h+1}\big(\mathrm{ack}_{h+2}(n-1)\big) \text{ nach Monotonielemma} \\
&= \mathrm{ack}_{h+2}(n)
\end{aligned}
$$

Mit $k = h + 2$ gilt also $f_P(n) \leq \mathrm{ack}_k(n)$ 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 $Q = \big(x_2 := x_1; P; x_1 := x_1 + 1\big)$

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

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

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

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

n	$\mathrm{ack}_0(n)$	$\mathrm{ack}_1(n)$	$\mathrm{ack}_2(n)$	$ack_3(n)$	$\mathrm{ack}_4(n)$ $\cdots$
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	$\mathrm{ack}_0(n)$	$\mathrm{ack}_1(n)$	$\mathrm{ack}_2(n)$	$ack_3(n)$	$\mathrm{ack}_4(n)$ ⋯
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	$2^{2^{65536}-6} - 3$
⋮					

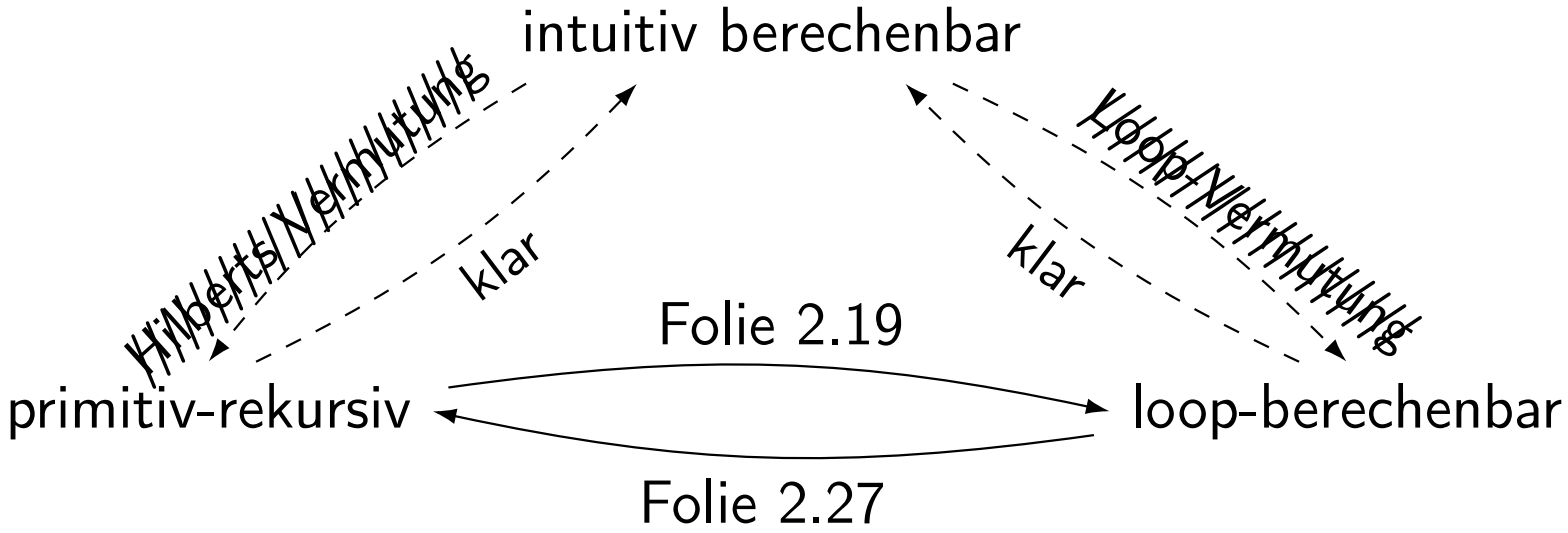
Veranschaulichung: trage in eine Tabelle die Funktionen ack_k für $k \in \mathbb{N}$ ein

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

n	$\mathrm{ack}_0(n)$	$\mathrm{ack}_1(n)$	$\mathrm{ack}_2(n)$	$ack_3(n)$	$\mathrm{ack}_4(n)$ ⋯
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 $[\![P]\!] : \mathbb{N} \to \mathbb{N}$.
2. Aus dem „Programmtext“ P und einer natürlichen Zahl n kann $[\![P]\!](n)$ berechnet werden, d.h. die folgende Abbildung ist intuitiv berechenbar:
$$I : L \times \mathbb{N} \to \mathbb{N} : (P, n) \mapsto [\![P]\!](n)$$
3. Einem Wort aus Γ^* „kann man ansehen“, daß es ein „Programmtext“ aus L ist, d.h. die folgende Funktion ist intuitiv berechenbar:
$$\chi_L : \Gamma^* \to \{\top, \bot\} : P \mapsto \begin{cases} \top & P \in L \\ \bot & \text{sonst} \end{cases}$$
4. Zu jeder intuitiv berechenbaren Fkt. $f : \mathbb{N} \to \mathbb{N}$ ex. $P \in L$ mit $[\![P]\!] = f$.

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 $I: L \times \mathbb{N} \to \mathbb{N}$. 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 $w_0, w_1, \ldots$ 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:

$$f\colon \mathbb{N} \to \mathbb{N}\colon n \mapsto [\![p(n)]\!](n) + 1$$

Wegen $[\![p(n)]\!](n) = I\big(p(n), n\big)$ ist sie aufgrund der 2. Eigenschaft intuitiv berechenbar.

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

Dann gilt

$$f(m) = [\![p(m)]\!](m) + 1 \neq [\![p(m)]\!](m) = [\![P]\!](m) = f(m)\,,$$

ein Widerspruch. □

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

Zum Beispiel:

n	$[\![p(0)]\!](n)$	$[\![p(1)]\!](n)$	$[\![p(2)]\!](n)$	$[\![p(3)]\!](n)$	$[\![p(4)]\!](n)$ $\cdots$
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 $[\![p(n)]\!]$ für $n \in \mathbb{N}$ ein

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

n	$[\![p(0)]\!](n)$	$[\![p(1)]\!](n)$	$[\![p(2)]\!](n)$	$[\![p(3)]\!](n)$	$[\![p(4)]\!](n)$ ⋯
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 $[\![p(n)]\!]$ für $n \in \mathbb{N}$ ein

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

n	$[\![p(0)]\!](n)$	$[\![p(1)]\!](n)$	$[\![p(2)]\!](n)$	$[\![p(3)]\!](n)$	$[\![p(4)]\!](n)$ ⋯
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 $f: D \to \mathbb{N}$ 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 $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ ist intuitiv berechenbar, wenn es einen Algorithmus gibt, der f berechnet, d.h.

- das Verfahren erhält $(n_1, \ldots, n_k) \in \mathbb{N}^k$ als Eingabe (nicht unbedingt aus dem Definitionsbereich),
- terminiert nach endlich vielen Schritten genau dann, wenn $f(n_1, \ldots, n_k)$ definiert ist
- und gibt in diesem Fall $f(n_1, \ldots, n_k)$ 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 $[\![P]\!] : \mathbb{N} \dashrightarrow \mathbb{N}$.
2. Aus dem „Programmtext“ P und einer natürlichen Zahl n kann $[\![P]\!](n)$ berechnet werden, d.h. die Abbildung $I : L \times \mathbb{N} \dashrightarrow \mathbb{N}$ mit
   - $I(P, n)$ definiert gdw. $[\![P]\!](n)$ definiert und
   - in diesem Fall gilt $I(P, n) = [\![P]\!](n)$

   ist intuitiv berechenbar.
3. Die folgende Funktion ist intuitiv berechenbar:

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

4. Zu jeder intuitiv berechenbaren partiellen Funktion $f : \mathbb{N} \dashrightarrow \mathbb{N}$ existiert $P \in L$ mit $[\![P]\!] = f$.

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

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

Dann erhält man wieder eine natürliche Zahl m mit $P = p(m)$ bzw. $[\![p(m)]\!] = f$. 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 die Programme u.U. nicht terminieren)
- Argumente der Form (K^+), (A^+) und (U^+) dafür, daß diese „Programmiersprachen“ den Begriff der intuitiven Berechenbarkeit korrekt beschreiben.