Die Ackermannfunktion

Prof. Dr. Jasmin Blanchette

Lehr- und Forschungseinheit für
Theoretische Informatik

Stand: 20. Juni 2023
Folien ursprünglich von PD Dr. David Sabel

Die Ackermannfunktion

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

Ackermannfunktion $a : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

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

Die Ackermannfunktion

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

Ackermannfunktion $a : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

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

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	$a(3, a(4, 3))$

Terminierung

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

Mehrmaliges Entfalten der dritten Zeile zeigt:

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

Daher sind alle rekursiven Aufrufe echt kleiner bezüglich des ersten Arguments.
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-Programme.

Danach wird das Programm angegeben, dass die Stackoperationen durchführt.

Darstellung vom Stack

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

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

Darstellung vom Stack:

- Folge von Zahlen $\langle n_1, \ldots, n_k \rangle$ sodass n_1 ganz oben liegt.
  Leerer Stack: 0.
- Im WHILE-Programm:
  - Variable *stack*, die $\langle n_1, \ldots, n_k \rangle = c(n_1, c(n_2, (\ldots, c(n_k, 0)\ldots)))$ speichert
  - Variable *stacksize*, die die Größe des Stacks speichert
  - Invariante: $\langle n_1, \ldots, n_k \rangle$ stellt $a(n_k, a(n_{k-1}, \ldots a(n_2, n_1)\ldots))$ dar.

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 dazu:
  $x := left(stack)$;
  $stack := right(stack)$;
  $stacksize := stacksize - 1$

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

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

$stack := 0;$
$stacksize := 0;$
$push(x, stack);$
$push(y, stack);$
WHILE $stacksize \neq 1$ **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;
$result := pop(stack)$

□

Eigenschaften der Ackermannfunktion

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 (durch Induktion)

Maximale LOOP-berechenbare Zahlen (1)

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

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

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

Maximale LOOP-berechenbare Zahlen (2)

Satz

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

Maximale LOOP-berechenbare Zahlen (2)

Satz

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

Beweis: Normalisiere P zunächst:

- Ersetze Zuweisungen $x_i := x_j + c$ mit $c > 1$ durch $x_i := x_j + 1; \underbrace{x_i := x_i + 1; \ldots x_i := x_i + 1}_{(c-1)\text{-mal}}$.
- Für **LOOP** x_i **DO** Q **END** und x_i kommt in Q vor: Ersetze **LOOP** x_i **DO** Q **END** durch $x_i' := x_i$; **LOOP** x_i' **DO** Q **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, sodass $f_P(n) < a(k, n)$ für alle $n \in \mathbb{N}$.

Maximale LOOP-berechenbare Zahlen (3)

Zu zeigen: Für jedes normalisierte LOOP-Programm P gibt es eine Konstante k, sodass $f_P(n) < a(k, n)$ 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, sodass $f_P(n) < a(k, n)$ 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 $k \neq j$
- $\rho(x_j) = n$
- $c = 1$
- $\rho'(x_i) = n + 1$
- $\rho'(x_j) = n$
- $\rho'(x_k) = 0$ für $k \neq j$ und $k \neq i$

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

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

D.h. die Aussage $f_P(n) < a(k, n)$ 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:

$$
\begin{aligned}
f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\
&< a(k_2, f_{P_1}(n)) && \text{(Annahme)} \\
&\leq a(k_2, a(k_1, n)) && \text{(Annahme, 1)} \\
&\leq a(\max\{k_1, k_2\}, a(\max\{k_1, k_2\} + 1, n)) && \text{(1)} \\
&= a(\max\{k_1, k_2\} + 1, n + 1) && \text{(2)} \\
&\leq a(\max\{k_1, k_2\} + 2, n) && \text{(3)}
\end{aligned}
$$

(1) Monotonie: $x \leq x', y \leq y' \implies a(x, y) \leq a(x', y')$
(2) Definition von a: $a(x, y) = a(x - 1, a(x, y - 1))$ falls $x > 0$ und $y > 0$
(3) 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)

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

$$
\begin{aligned}
f_P(n) &\leq \underbrace{f_Q(\ldots(f_Q}_{\rho(x_i)\text{-mal}}(n-\rho(x_i))\ldots) + \rho(x_i) && \text{(da } x_i \notin Q,\ f_Q \text{ unabh. von } \rho(x_i)\text{)} \\
&\leq a(k_1, \ldots, a(k_1, n-\rho(x_i))\ldots) && \text{(}\rho(x_i)\text{-mal gilt } < \text{ (für jedes } f_Q\text{),} \\
& && \text{daher } \leq \text{ und } \rho(x_i) \text{ fällt weg} \\
& && \text{(mit Monotonie und I.H.))} \\
&< \underbrace{a(k_1, \ldots, a(k_1,}_{(\rho(x_i)-1)\text{-mal}} a(k_1+1, n-\rho(x_i))\ldots)) && \text{(Monotonie)} \\
&= a(k_1+1, n-1) && (1) \\
&< a(k_1+1, n) && \text{(Monotonie)}
\end{aligned}
$$

(1): Definition von a: $a(x,y) = a(x-1, a(x, y-1))$ falls $x > 0$ und $y > 0$

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

Ackermannfunktion nicht LOOP-berechenbar

Theorem

Die Ackermannfunktion ist nicht LOOP-berechenbar.

Ackermannfunktion nicht LOOP-berechenbar

Theorem

Die Ackermannfunktion ist nicht LOOP-berechenbar.

Beweis durch Widerspruch:

- Annahme: a ist LOOP-berechenbar.
- Dann ist auch $g(x_1) = a(x_1, x_1)$ LOOP-berechenbar.
- Sei P LOOP-Programm, dass g berechnet.
- Dann gilt $g(x_1) \leq f_P(x_1)$ (da $\rho'(x_0) = g(x_1)$ 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 $g(k) \leq f_P(k) < a(k, k) = g(k)$. Widerspruch.
- a ist nicht LOOP-berechenbar. □

Fazit

Theorem

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