

Grenzen der primitiven Rekursion: Die Ackermann-Funktion

Vermutung von Hilbert (1926):

Jede berechenbare Funktion ist primitiv-rekursiv.

Widerlegung durch Ackermann 1928 durch Angabe einer (totalen) Funktion, die nicht primitiv-rekursiv ist.

Beweisidee: Sie wächst schneller als alle primitiv-rekursiven Funktionen.

Definition:

$$\text{ack}(n, m) \Leftrightarrow \begin{cases} m + 1 & \leftarrow n = 0 \\ \text{ack}(n - 1, 1) & \leftarrow n > 0 \\ \text{ack}(n - 1, \text{ack}(n, m - 1)) & \leftarrow \text{sonst} \end{cases}$$

```
(define (ack n m)
  (cond
    ((zero? n) (1+ m))
    ((and (> n 0) (zero? m))
     (ack (- n 1) 1))
    (else
     (ack (- n 1) (ack n (- m 1))))))
```

Ackermann-Funktion: Ein Programmablauf

```
ack(3,2) = ack(2,ack(3,1))
           ack(2,ack(3,0))
           ack(2,1)
           ack(1,ack(2,0))
           ack(1,1)
           ack(0,ack(1,0))
           ack(0,1)
           ack(0,2)
           ack(1,3)
           ack(0,ack(1,2))
           ack(0,ack(1,1))
           ...
```

Zur Termination: (Lexikographische) Ordnung auf Tupeln

$(x_1, x_2, \dots, x_n) \prec (y_1, y_2, \dots, y_n)$
 $\Leftrightarrow \exists 1 \leq i \leq n. (x_i < y_i \wedge (\forall 1 \leq j < i. x_j = y_j))$

Noethersche Ordnung:

Es gibt keine unendlichen, absteigenden Folgen

Lösungsansatz zur Termination:

Suche eine Terminierungsfunktion, die in noethersche Ordnung abbildet.

Die lexikographische Ordnung

$(n, m) \prec (n', m') \Leftrightarrow n < n' \vee (n = n' \wedge m < m')$
 ist noethersch.

Hier:

$$\begin{aligned}(n-1, 1) &< (n, m) \\ (n-1, \text{ack}(n, m-1)) &< (n, m) \\ (n, m-1) &< (n, m)\end{aligned}$$

Die Ackermann-Funktion terminiert:

$\text{ack}(0,0)$ terminiert und

(Annahme:) $\text{ack}(n', m')$ terminiert für alle $(n', m') < (n, m)$

$\Rightarrow \text{ack}(n, m)$ terminiert,

weil $z = \text{ack}(n, m-1)$ terminiert

$$(n, m-1) < (n, m)$$

und $\text{ack}(n-1, z)$ terminiert

$$(n-1, z) < (n, m)$$

Variante der Ackermann-Funktion (2)

```
(define (times n m) ; m-fache Addition
  (if (zero? m)
      0
      (plus n (times n (sub1 m)))))
```

... und Exponentiation

```
(define (exponent n m)
  (if (zero? m)
      1
      (times n (exponent n (sub1 m)))))
```

Bem.: "Stark wachsende Funktionen"
(vgl. Springer/ Friedman)

Zum Wachstum: Eine Variante der Ackermann-Funktion

Ausgangspunkt: Nachfolger- und Vorgänger-Funktion

```
(define (add1 n) (+ n 1))
(define (sub1 n) (- n 1))
```

Damit kann man rekursiv die Addition und daraus die Multiplikation (als wiederholte Addition) definieren:

```
(define (plus n m)
  (if (zero? m)
      n
      (add1 (plus n (sub1 m)))))
```

Variante der Ackermann-Funktion (3)

Hyperexponentiation — nach demselben Schema:

```
(define (super n m)
  (if (zero? m)
      1
      (exponent n (super n (sub1 m)))))
```

z.B.

```
(super 2 3) => (exponent 2 (super 2 2))
=> (exponent 2 (exponent 2
  (super 2 1)))
=> (exponent 2 (exponent 2
  (exponent 2 (super 2 0))))
=> (exponent 2 (exponent 2
```

```

      (exponent 2 1))
=> (exponent 2 (exponent 2 2))
=> (exponent 2 4)
=> 16

```

Also: $(\text{super } 2 \ 3) = 2^{2^2}$
 $(\text{super } 2 \ 4) = 2^{2^{2^2}} = 65536$

Bildungsgesetz:

```

(define (f_k n m)
  (if (zero? m)
      1
      (f_k-1 n (f_k n (sub1 m)))))

```

Variante der Ackermann-Funktion (4)

Weiter

```

(define (superduper n m)
  (if (zero? m)
      1
      (super n (superduper n (sub1 m)))))

```

$(\text{superduper } 2 \ 3) = 65536$
 $(\text{superduper } 2 \ 4) = 2^{65536} \approx 10^{19500}$

Variante der Ackermann-Funktion (5)

Stattdessen abstrahieren wir und definieren *eine* Funktion höherer Ordnung `super-order` derart, daß

```

(super-order 1) = plus
(super-order 2) = times
(super-order 3) = exponent
...

```

```

(define (super-order n)
  (cond
    ((= n 1) plus)
    ((= n 2) times)
    (else
     (lambda (x y)

```

```

(if (zero? y) 1
    ((super-order (sub1 n))
     x
     ((super-order n) x (sub1 y))))))
((super-order 4) 2 3) => (super 2 3) => 16

```

Variante der Ackermann-Funktion (6)

Wir erhalten eine Variante der Ackermann-Funktion, wenn in `super-order` die Argumente `n`, `x` und `y` gleich sind:

```

(define (ack2 n)
  ((super-order n) n n))

(ack2 1) => (plus 1 1) => 2
(ack2 2) => (times 2 2) => 4
(ack2 3) => (exponent 3 3) => 27

```

Wie groß ist `(ack2 4)`? $4^4 = 256$

Sei $u = 4^{4^4} = 4^{256}$; $\log_{10} u = 256 \cdot \log_{10} 4 = 154.13$, also $4^{256} \approx 10^{154}$

Sei $v = 4^{4^4}$; $\log_{10} v \approx 10^{154} \log_{10} 4 \approx 0.602 \cdot 10^{154} \approx 6 \cdot 10^{153}$.

Damit: $v \approx 4^{10^{153}}$