Rekursions- und Lerntheorie

WiSe 2010/11; Univ. Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Rekursions- und Lerntheorie Gesamtübersicht

1. Einführung: Grundsätzliche Betrachtungen
2. Berechenbarkeitstheorie: Die Churchsche These (3-4 VL)
3. Ausblicke auf weitere Ergebnisse der Rekursionstheorie
4. Lerntheorie: Modelle und Aussagen

μ-Rekursion Neues Erzeugungsschema: *μ-Operator*, $\mu : f \mapsto g$

- gegeben: $(k+1)$-stellige (evtl. partielle!) Funktion f
- erzeugt: k-stellige Funktion g mit

$$\begin{aligned} g(x_1, ..., x_k) \quad = \quad & \min\{n \in \mathbb{N} \mid f(n, x_1, ..., x_k) = 0 \text{ und} \\ & \text{für alle } m < n \text{ ist } f(m, x_1, ..., x_k) \text{ definiert und} \\ & \text{für alle } m < n \text{ ist } f(m, x_1, ..., x_k) > 0\} \end{aligned}$$

oder kürzer

$$g(x_1, ..., x_k) = (\mu n)[f(n, x_1, ..., x_k) = 0]$$

Anmerkungen zur Nullstellensuche mit μ:

- Schreibweise: $g := \mu f$
- Mimimum der leeren Menge ist undefiniert, damit:
- g ist evtl. nicht total:
  - wenn $f(n, x_1, ..., x_k)$ nie Null wird, ist $g(x_1, ..., x_k)$ undefiniert;
  - wenn $f(n, x_1, ..., x_k) = 0$, aber $f(m, x_1, ..., x_k)$ undefiniert ist für ein $m < n$, so ist $g(x_1, ..., x_k)$ ebenfalls undefiniert.
- Zur Auswertung von g: Werte $f(n, x_1, ..., x_k)$ für $n = 0, 1, 2, ...$ der Reihe nach ausrechnen...

Zentrale Definition dieser Vorlesung

Die Menge der μ-*rekursiven* (auch: *partiell rekursiven*) Funktionen besteht genau aus den Funktionen, die sich aus der Menge der Grundfunktionen durch endlich oft wiederholte Anwendung des Kompositionsschemas, des Rekursionsschemas und des μ-Operators bilden lassen.

Bsp.: $(\mu y)[|2^y - x|]$ berechnet ... ?

Satz: Die Klasse der μ-rekursiven Funktionen stimmt mit der Klasse der WHILE- (GOTO-, TURING-) berechenbaren Funktionen überein.

Anmerkungen

Kompositionsschema und Rekursionsschema müssen streng genommen in ihrer Definition modifiziert werden.

Üblicherweise betrachtet man dabei z.B. das Ergebnis eines Kompositionsschemas nur dann als definiert, wenn alle beteiligten Funktionen an den relevanten Positionen definiert sind, obwohl das u.U. auch anders sinnvoll wäre.

Bsp.: Ist f_1 total, f_2 jedoch partiell, so ist $pr_1^2(f_1(x), f_2(x))$ nur für den Definitionsbereich von f_2 definiert.

Beweis: analog zum Beweis 'LOOP-berechenbar $\Leftrightarrow$ primitiv rekursiv', erweitert um μ-Operator und WHILE-Schleife.
Alter Beweis also nur noch um Fall (d) erweitert:

Fall (d) für '$\Rightarrow$':
$g = \mu f$ für μ-rekursive Funktion f, also

$$g(x_1, ..., x_k) = (\mu n)[f(n, x_1, ..., x_k) = 0]$$

Induktiv: WHILE-Programm Q zur Berechnung von f existiert...

Passendes WHILE-Programm für g:

$x_0 := 0;$
$y := f(0, x_1, ..., x_k);$
`WHILE` $y \neq 0$ `DO`
  $x_0 := S(x_0);$
  $y := f(x_0, x_1, ..., x_k);$
`END;`

(Berechnung von f über Q wieder mit Sicherung der Variablen...)

Fall (d) für '$\Leftarrow$':
WHILE-Programm P habe die Form `WHILE` $x_i \neq 0$ `DO` Q `END`

In P vorkommende Variablen seien x_0, x_1,..., x_k mit $k \geq r$.

Wieder gesucht: μ-rekursive Funktion $g_P : \mathbb{N} \text{—}\circ\!\!\rightarrow \mathbb{N}$ mit

$$g_P(<a_0, a_1, ..., a_k>) = <b_0, b_1, ..., b_k>$$

für gegebene Anfangswerte a_0, a_1,..., a_k und Endwerte b_0, b_1,..., b_k (nach Ablauf von P) der Variablen x_0, x_1,..., x_k

Induktiv: Entsprechende (μ-rekursive!) Funktion g_Q für Q existiert...

Definiere (mit primitiver Rekursion) zweistellige Funktion h:

$$h(0, x) = x; \qquad h(n+1, x) = g_Q(h(n, x))$$

D.h. h beschreibt wieder n Anwendungen von Q

Dann: $t(x) := (\mu n)[d_{i+1}^{(k+1)}(h(n, x)) = 0]$ ist minimale Wiederholungszahl von Q bis zum Erreichen einer Nullstelle, also setze

$$g_P(x) = h(\quad t(x)\ ,\ x\ )$$

(Rest des Beweises aus letzter VL wird unverändert übernommen!)

Die Ackermann-Funktion

Geschichtliches: D. Hilbert vermutete 1926, dass der Begriff der Berechenbarkeit sich mit Hilfe der primitiven Rekursion formalisieren lässt.
Dies widerlegte W. Ackermann noch im gleichen Jahr!
R. Péter veröffentlichte 1955 eine vereinfachte Fassung:

$$\begin{aligned} a(0,m) &= m+1 \\ a(n+1,0) &= a(n,1) \\ a(n+1,m+1) &= a(n,a(n+1,m)) \end{aligned}$$

Aufgabe: Formalisieren Sie diese Definition weitest möglich mit Hilfe der μ-Rekursion.

Die Ackermann-Funktion Werte von $a(n, m)$:

n/m	0	1	2	3	4	m
0	1	2	3	4	5	$m+1$
1	2	3	4	5	6	$m+2$
2	3	5	7	9	11	$2m+3$
3	5	13	29	61	125	$8 \cdot 2^m - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$	$a(3, 2^{65536} - 3)$	$a(3, a(4, 3))$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}} - 3$ (m + 3 Terme im Turm)

$a(n, n)$ wächst schneller als jede LOOP-berechenbare Funktion.

Die Inverse α hiervon ist “praktisch konstant”, z.B. für UNION/FIND-Algorithmus

Abschätzung: $\mathcal{O}(n\alpha(n))$.

Originalarbeit von W. Ackermann siehe `doi:10.1007/BF01459088`.

Beweisskizze zu: Die Ackermannfunktion ist nicht primitiv-rekursiv

— Als erstes definiert man zu jeder primitiv-rekursiven Funktion g eine Funktion

$$f_g(n) := \max \left\{ g(n_1, \ldots, n_k) : \sum_{i=1}^{k} n_i \leq n \right\}$$

Diese Funktion gibt das Maximum an, das man mit g erreichen kann, wenn die Summe der Argumente n nicht überschreitet.

— Dann zeigt man durch strukturelle Induktion über den induktiven Aufbau der primitiv-rekursiven Funktionen, dass es zu jeder primitiv-rekursiven Funktion g eine natürliche Zahl k gibt, sodass für alle $n \geq k$ gilt: $f_g(n) < a(k, n)$.

Anschaulich zeigt dies, dass die Ackermannfunktion stärker wächst als jede primitiv-rekursive Funktion.

— Damit beweist man dann die Behauptuing:

Angenommen, $a(k, n)$ wäre primitiv-rekursiv, dann auch $g(n) := a(n, n)$.
Nach der Vorbemerkung gibt es aber ein k, sodass für alle $n \geq k$ gilt: $g(n) < a(k, n)$. Setzt man hier $n = k$, so erhält man den Widerspruch:

$$g(k) \leq f_g(k) < a(k, k) = g(k).$$

Einzelheiten siehe Uwe Schöning: Theoretische Informatik — kurzgefasst. Spektrum Akademischer Verlag, Heidelberg 2001, S. 108–113.

Alternativer Beweis

Verwende *Diagonalisierung* als Beweis-/Konstruktionsprinzip
Betrachte 10-elementiges Alphabet A:

$$A := \{ +, \ , -, :=, \ ;, \ , \texttt{LOOP}, \ \texttt{DO}, \ \texttt{END}, \ x, \ 0, \ 1 \}$$

$\Rightarrow$ jedes LOOP-Programm ist als Wort über A schreibbar!
(dabei Variable x_i durch 'x $bin(i)$' und Konstante c durch $bin(c)$)
Bilde Liste $P_0, P_1, P_2, P_3, \ldots$ *aller* LOOP-Programme durch

- Sortierung nach Länge des Programmes und
- bei gleicher Länge: Alphabetische Sortierung (mit beliebiger aber fester Ordnung auf A)

Dies liefert die *längenlexikographische Ordnung*, also NICHT wie im Wörterbuch.

Ein Hilfssatz Die folgende Funktion $g : \mathbb{N}^2 \to \mathbb{N}$ ist WHILE-(Turing-, GOTO-) berechenbar:

$g(i, n) :=$ der Wert der einstelligen Zahlenfunktion,
die von P_i berechnet wird, bei Eingabe von n.

Beweis: (1) Aus i kann P_i berechnet werden (mit einer Turingmaschine):

```
m := 0
FOR k := 0, 1, 2, ... DO
  FOR w ∈ A* mit |w| = k DO
    IF w codiert LOOP-Programm THEN
      IF m = i THEN RETURN w END
      m := m + 1
    END
  END
END
```

Syntaxüberprüfung, siehe Compilerbau-Vorlesung

Beweis des Hilfssatzes (Forts.)

(2) Wenn P_i und n gegeben sind: Die Berechnung von P_i auf $(0, n, 0, 0, ...)$ in den Variablen $x_0, x_1, ...$ kann simuliert werden (wieder mit einer Mehrband-Turingmaschine).
Dabei z.B. P_i auf einen Band gespeichert und alle Variablen, die P_i nutzt, auf einem anderen Band.
Es ist nicht möglich, für jede Variable ein eigenes Band vorzusehen (da die Zahl der Variablen von P_i abhängt).

Also: aus i und n kann das Resultat von P_i auf n berechnet werden
$\Rightarrow g$ ist berechenbare (totale) Funktion!

Wesentlicher Satz der VL

Satz: Die Funktion g aus vorigem Hilfssatz ist nicht LOOP-berechenbar.

Außerdem gibt es eine totale Funktion $f : \mathbb{N} \to \mathbb{N}$, die WHILE-berechenbar, aber nicht LOOP-berechenbar ist.

Beweis durch Diagonalisierung Benutze folgende 'Diagonalkonstruktion' einer Funktion f:

$$f(n) := g(n, n) + 1$$

Damit:

- g total, also auch f total
- g WHILE-berechenbar, also auch f WHILE-berechenbar
- Wäre g LOOP-berechenbar, so wäre auch f LOOP-berechenbar.

Annahme: f sei LOOP-berechenbar.

- Dann gibt es ein LOOP-Programm P_j, das f berechnet.
- Damit gilt $f(n) = g(j, n)$ für alle $n \in \mathbb{N}$
- Insbesondere auch $f(j) = g(j, j)$
- Nach Definition von f gilt jedoch $f(j) = g(j, j) + 1$
- Widerspruch...

Beweis durch Diagonalisierung

Damit: f nicht LOOP-berechenbar,
also auch g nicht LOOP-berechenbar

Idee der Diagonalisierung: Betrachte Diagramm aller Werte $g(i,n)$:

$i \backslash n$	0	1	2	3	$\cdots$
P_0	$g(0,0)$	$g(0,1)$	$g(0,2)$	$g(0,3)$	
P_1	$g(1,0)$	$g(1,1)$	$g(1,2)$	$g(1,3)$	
P_2	$g(2,0)$	$g(2,1)$	$g(2,2)$	$g(2,3)$	
P_3	$g(3,0)$	$g(3,1)$	$g(3,2)$	$g(3,3)$	
$\vdots$					

$f(n)$	$g(0,0)+1$	$g(1,1)+1$	$g(2,2)+1$	$g(3,3)+1$	$\cdots$

f wird so definiert, dass f sich in der Diagonale von jeder der Funktionen unterscheidet.

Der Satz von Kleene für μ-Rekursion

Satz: Für jede k-stellige μ-rekursive Funktion f
gibt es zwei $(k+1)$-stellige primitiv rekursive Funktionen p und q,
so dass sich f darstellen lässt als

$$f(x_1, ..., x_k) := p(x_1, ..., x_k, \mu q(x_0, x_1, ..., x_k))$$

Hierbei ist μq durch die Anwendung des μ-Operators auf q entstanden und steht abkürzend für

$$(\mu x_0)[q(x_0, x_1, ..., x_n) = 0]$$

Beweis: f μ-rekursiv
$\Rightarrow$ WHILE-Programm P für f
$\Rightarrow$ WHILE-Programm P' für P mit nur einer WHILE-Schleife
$\Rightarrow$ μ-rekursive Funktion f' für P' mit nur einem μ-Operator

Weitere Erinnerungen

— Umrechnungen zwischen Zahlen und Zeichenketten.
— Aufzählung berechenbarer Funktionen.
— Notationen als “Turingprogrammiersprachen”.
— smn- und utm-Eigenschaft.
— Äquivalenzsatz von Rogers: Eine Notation für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation h äquivalent, wenn sie die utm-Eigenschaft und die smn-Eigenschaft hat.
Details folgen!

Sei Σ ein Alphabet.
Eine *Notation* für die berechenbaren Funktionen $g : \Sigma^* \multimap \Sigma^*$ ist eine surjektive totale Funktion

$$h' : \{0,1\}^* \to \{ \text{ berechenbare Wortfunktionen über } \Sigma^* \}$$

- Jedem $w \in \{0,1\}^*$ wird eine berechenbare Wortfunktion zuordnet.

- Zu jeder berechenbaren Wortfunktion g gibt es ein Wort $w \in \{0,1\}^*$ derart, dass $h'(w)$ gerade die Funktion g ist.

Wir schreiben oft auch h'_w für die Funktion $h'(w)$.

Vergleichbarkeit von Notationen

- Eine Notation h'' ist in eine Notation h' übersetzbar, wenn es eine totale(!) berechenbare Wortfunktion

$$u : \{0,1\}^* \to \{0,1\}^*$$

gibt mit $h''_w(x) = h'_{u(w)}(x)$ für alle $w \in \{0,1\}^*$ und $x \in E^*$, d.h. $h''_w = h'_{u(w)}$.

- Jedes 'Programm' w der 'Programmiersprache' h'' kann also dann automatisch in ein 'Programm' der 'Sprache' h' übersetzt werden.
- u heißt *Übersetzungsfunktion*.

Notationen h' und h'' heißen *äquivalent*, wenn man jede in die andere übersetzen kann.

Satz: [Äquivalenzsatz von Rogers] Eine Notation für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten (Standard-)Notation h äquivalent, wenn sie die utm-Eigenschaft und die smn-Eigenschaft hat.

Beweis:

- Sei h unsere Standardnotation.
- Sei h' weitere Notation für berechenbaren Wortfunktionen.

'$\Rightarrow$': h und h' seien äquivalent, mit $h_w = h'_{u(w)}$ und $h'_w = h_{u'(w)}$

(a) utm-Eigenschaft für h: f_h mit $f_h(w\#x) := h_w(x)$ ist berechenbar.
Definiere f' durch $f'(w\#x) := f_h(u'(w)\#x)$
Dann ist f' berechenbar und

$$h'_w(x) = h_{u'(w)}(x) = f_h(u'(w)\#x) = f'(w\#x)$$

d.h. f' zeigt utm-Eigenschaft für h'.

(b) Sei $g : \Sigma^* \multimap \Sigma^*$ irgendeine berechenbare Wortfunktion.
smn-Eigenschaft für h: r mit $h_{r(w)}(x) := g(w\#x)$ ist berechenbar.
Dann ist $u \circ r$ berechenbar (und total) mit

$$g(w\#x) = h_{r(w)}(x) = h'_{u \circ r(w)}(x)$$

D.h. h' hat auch die smn-Eigenschaft.

'$\Leftarrow$': h' besitze ebenfalls die utm- und die smn-Eigenschaft:
utm-Eigenschaft für h: f_h mit $f_h(w\#x) = h_w(x)$ berechenbar.
Wende smn-Eigenschaft für h' auf f_h an, mit entsprechendem r':

$$h_w(x) = f_h(w\#x) = h'_{r'(w)}(x)$$

Damit Übersetzung von h nach h' mit r'

Analog: utm-E. für h' und smn-E. für $h \Rightarrow h'$ in h übersetzbar.

Damit: h und h' äquivalent!