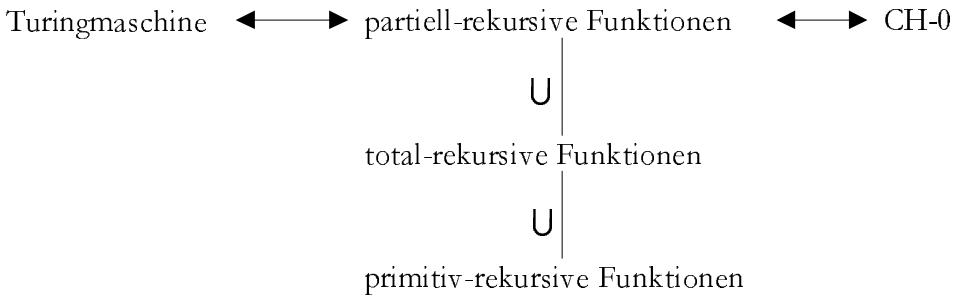
Mitschrift zur Vorlesung Theoretische Informatik II
vom 12.01.2001

von Jean Gressmann

Zusammenfassung der Vorlesungsmitschrift:

- die Menge der primitiv-rekursiven Funktionen ist eine echte Teilmenge der μ-rekursiven Funktionen (Satz und Beweis)
- Definition der charakteristischen Funktion um Sprachen und REG/TM/partiell-rekursiven Funktionen in Beziehung zu setzen
- Die Chomsky-0 Sprachen lassen sich durch Turingmaschinen erkennen (Satz und „=>“-Richtung des Beweises).

Aus Kleenes Normalformtheorem ergibt sich der folgende Zusammenhang zwischen der Turingmaschine und den partiell-rekursiven Funktionen:



Motivation/Erläuterung bezüglich der Programmiersprachen

- die primitive Rekursion entspricht in den Programmiersprachen der for-Schleife
- alle primitiv-rekursiven Funktionen, die ohne den Operator „primitive Rekursion“ auskommen, haben die Schachtelungstiefe null
- jede primitiv-rekursive Funktion, die das prim.-rek. Schema (die Vorschrift zur Erzeugung der Funktion) mit Funktionen der Schachtelungstiefe i zur Definition verwendet, erhält die Schachtelungstiefe i+1

Beispiel:

Operation	Schachtelungstiefe (Anzahl der ineinander geschachtelten for-Schleifen)	Funktionsterm
Add	1	x+y = ((x+y)-1)+1
Sub 1	1	
Mult	2	x*y = x(y-1) + x
Sub	2	
Exp	3	

Aus Sicht von Pascal bedeutet dies: Wenn man von den Grundfunktionen (Schachtelungstiefe null) ausgeht, so müssen zur Realisierung der Addition *eine* for-Schleife, zur Realisierung der Multiplikation bereits *zwei* ineinander geschachtelte for-Schleifen verwendet werden. Durch die Anzahl der Verschachtelungen die notwendig sind, um einen Funktion zu realisieren, wird die Klasse F_{prim} hierarchisch geordnet. Die Hierachiestufe einer Funktion ist dabei immer abhängig von den Grundoperationen. Diesen Zusammenhang soll das nächste Beispiel verdeutlichen.

Beispiel:

Sei die Addition (x+y) als Grundfunktion (Stufe null) gegeben. Die Exponentialfunktion x^y wird über die Multiplikation definiert. Die Multiplikation basiert ihrerseits auf der Addition. Die Multiplikation ist also eine Funktion der Stufe 1 und die Exponentialfunktion eine Funktion der Stufe 2.

Ist allerdings die Multiplikation als Grundfunktion gegeben, so ist die Exponentialfunktion lediglich eine Funktion der Stufe 1.

Ein Programm, welches nur aus for-Schleifen besteht, terminiert nach einer maximalen Laufzeit. Eine Funktion, die durch ein for-Schleifenprogramm berechnet wird, ist daher immer total. Im Gegensatz dazu ist die Laufzeit eins while-Schleifenprogramms potenziell unendlich. Es gilt also, dass F_{prim} (totale Funktionen) $\subset F_\mu \supset R_\mu$ (total-rekursive Funktionen), da ja F_{prim} und R_μ totale Funktionen beinhalten wohingegen F_μ auch partiell definierte Funktionen enthält. Um das Kleene'sche Normalformtheorem zu beweisen, muß jetzt nur noch $F_{prim} \subset R_\mu$ gezeigt werden.

Satz O: $F_{prim} \subset R_\mu$

Vorbemerkung zum Beweis des Satzes O:

Um die Klasse F_{prim} zu verlassen, wird eine Funktion benötigt, deren Werte über alle Maßen wachsen. Idealerweise würde diese durch eine (Endlos-)while-Schleife realisiert. While-Schleifen lassen sich allerdings *nicht* mit for-Schliefen konstruieren.

Beispiel:

NICHT möglich:

```
for(int i = 0; i < 1; i--) oder
for(int i = 0; i < 1; )
```

Die Idee für die gesuchte Funktion ist nun die: Man gibt der Funktion eine Verschachtelungstiefe als zweiten Parameter mit. Die Funktion rechnet dann mit der angegeben Tiefe das Ergebnis für den ersten Parameter aus.

Ist zum Beispiel

A(x,y) eine Funktion. Für den Parameter x wird der Funktionswert von A berechnet. y gibt an, mit welcher Verschachtelungstiefe der Funktionswert von x ausgerechnet werden soll.

Da ein Programm nur über eine feste Anzahl von for-Schleifen verfügen kann (der Quelltext muss ja endlich sein) ist es nicht möglich, eine solche Funktion in der Klasse F_{prim} zu berechnen.

Beweisidee:

„$\subseteq$“ klar, siehe Definition von F_{prim} und R_μ.

Um „$\neq$“ zu zeigen, geht man über die Schachtelungstiefe und definiert eine Funktion A, die eine Verschachtelungstiefe als Eingabe erhält:

$$A: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$

Für A(x,y) besitzt das Programm, welches A realisiert, die Verschachtelungstiefe y und rechnet mit dem Wert x. A kann damit nicht prim.-rek. sein, da ein for-Schleifen-Programm eine feste Verschachtelungstiefe besitzt. Die bekannteste Fassung der Funktion A ist die Ackermannfunktion (eine Version):

$$\begin{aligned} A(x,0) &= x+1 \\ A(0,y) &= A(1,y-1), \quad \forall\, y \in \mathbb{N}_0 \\ A(x+1,y+1) &= A(A(x,y+1),y), \quad \forall\, y,x \in \mathbb{N}_0 \end{aligned}$$

Anschaulich kann man sich das Wachstum der Ackermannfunktion durch das folgenden Bild verdeutlichen:

$$2^{2^{2^{\cdot^{\cdot^{\cdot}}}}} = n \underbrace{\to}_{\text{eine Schachtelung mehr}} 2^{\overbrace{2^{2^{\cdot^{\cdot^{\cdot}}}}}^{n}}$$

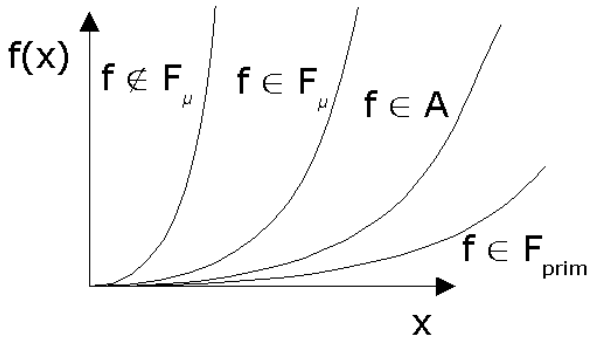
Einige Werte der Ackermannfunktion:

$$A(x,0) = x+1, \quad A(x,1) = x+2, \quad A(x,2) = 2x+3, \quad A(x,3) = 2^{x+3}, \quad A(x,4) > 10^{10^{21000}}$$

Zu zeigen bleibt:

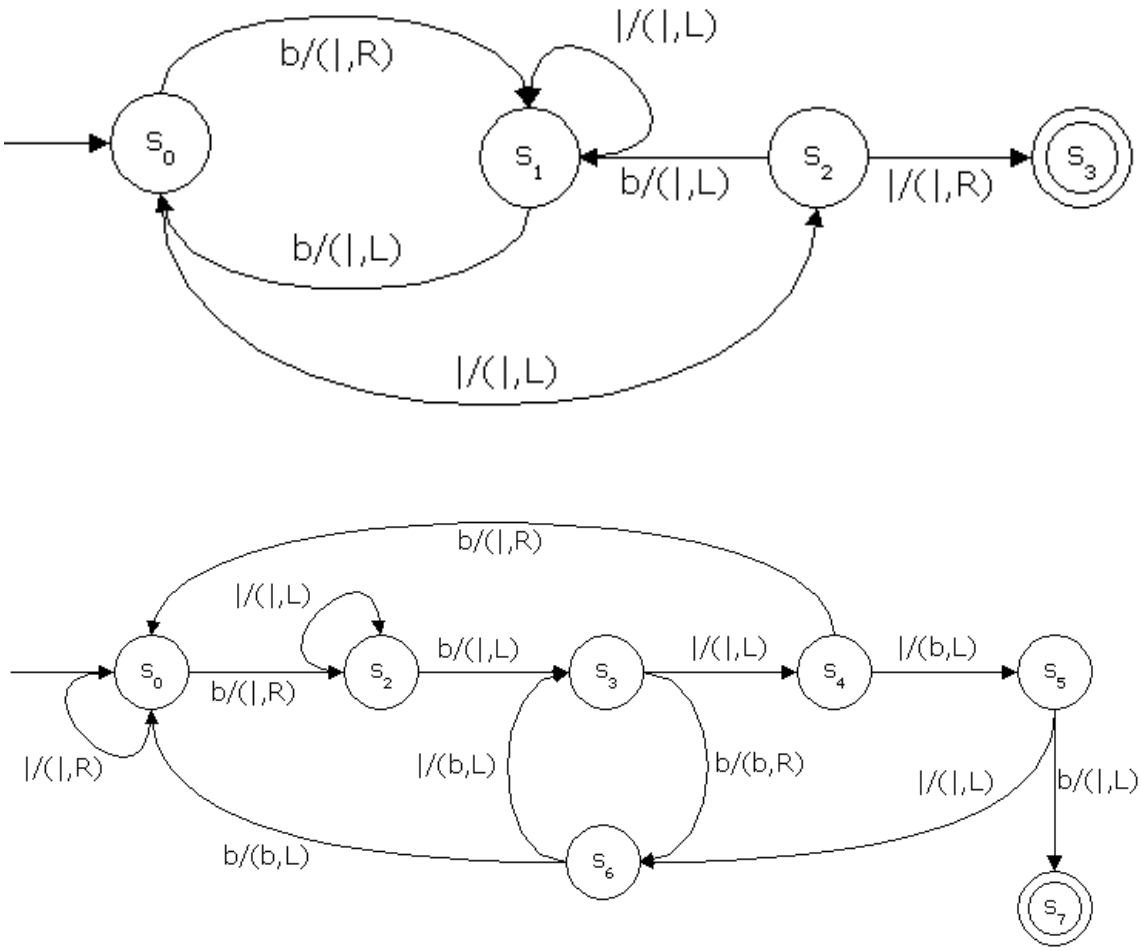
Zu jeder prim.-rek. Funktion $f: \mathbb{N}_0 \to \mathbb{N}_0$ gibt es ein $n_f \in \mathbb{N}_0$ und ein $y_f \in \mathbb{N}_0$ (passend zum n_f), so dass für alle $n > n_f$ gilt: $f(n) < A(n,y_f)$. y_f wird dabei größer als die Schachtelungstiefe von f gewählt, also z.B. y_f = Schachtelungstiefe(f) + 2.

Zur anschaulichen Darstellung von Funktionsklassen sind Wachstumsprozesse oft geeignet:



Funktionen, die selbst im Vergleich zu der Ackermannfunktion sehr schnell wachsen, sind die sogenannten Biber-Funktionen. Eine TM, die eine Biber-Funktion realisiert, versucht mit einer endlichen Anzahl von Zuständen maximal viele Striche „|“ auf das Band zu schreiben, ohne Endlosschleifen benutzen. Ein solcher Automat terminiert also nach einer *endlichen* Anzahl von Schritten. Es gibt bereits Automaten, die mit fünf Zuständen (davon ist einer Endzustand) ca. 16 Millionen Striche schreiben.

Beispiel:



6.4 Grammatiken

Frage: Wie ordnen sich Grammatiken in die Berechnungsmodelle ein?

Problem:

Wie lassen sich Beziehungen zwischen Sprachen (erzeugt durch Grammatiken) und TM/Reg.-Maschinen/part.-rek. Funktionen (die Funktionen berechnen) herstellen?

Lösung:

Man geht über zu charakteristischen Funktionen, die als Wert (Bild) nur noch zwei Symbole haben

(für w ∈ L(G) und w ∉ L(G)). Dieser Übergang entspricht dem Übergang von Turingmaschinen zu Akzeptoren.

Definition P:

Sei T ein Alphabet und t ∈ T ein beliebiges aber fest gewähltes Zeichen. Sei L ⊆ T* eine Sprache.

1.) Die (totale) Abbildung $X_L : T^* \rightarrow T^*$ mit $X_L = \begin{Bmatrix} t & falls\, w \notin L \\ tt & falls\, w \in L \end{Bmatrix}$ heißt charakteristische Funktion von L.

2.) Die (partielle) Abbildung $\Psi_L : T^* \rightarrow T^*$ mit

$\Psi_L = \begin{Bmatrix} \perp & falls\, w \notin L \\ tt & falls\ w \in L \end{Bmatrix}$, $\perp$ = *nicht definiert* heißt partiell-charakteristische Funktion von L.

Reguläre Sprachen ◄——► DA

Kontextfreie Sprachen ◄——► NPDA

CH-0 ◄·····► Turingmaschine

◄——► werden akzeptiert durch

◄·····► einseitige Akzeptanz (siehe Text)

Chomsky-0 Sprachen werden akzeptiert in dem Sinne, dass sich eine TM konstruieren läßt, die alle Wörter der jeweiligen CH-0 Sprache akzeptiert. Wird dieser Maschine ein Wort vorgesetzt, das nicht Element der Sprache ist, so terminiert die TM nicht. Das Wort wird nicht erkannt.

Man kann das Halteproblem auf diese Situation übertragen: Das Nichthalten der TM beschreibt die Klasse der part.-charakteristischen Funktionen, die nicht berechenbar sind.

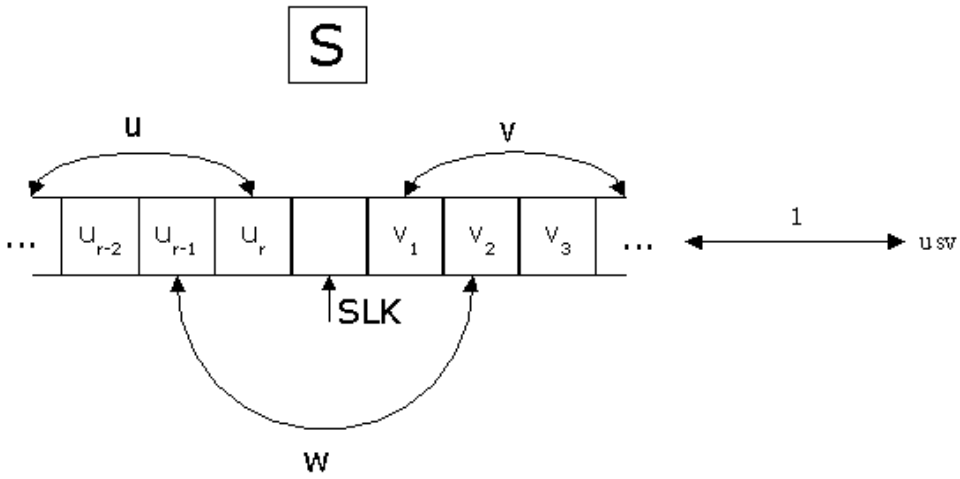
Satz Q:

Eine Sprache L ⊆ T* kann genau dann von einer Grammatik erzeugt werden, wenn die part.-ch. Funktion $\left(\Psi_L : T^* \rightarrow T^*\right)$ Turing-berechenbar ist. (Chomsky-0 Sprachen können durch Turingmaschinen erkannt werden)

Beweis:

Der Beweis wird ein sogenannter Simulationsbeweis sein. Die Turingmaschine simuliert dabei CH-0-Grammatiken:

Konfiguration der TM ◄——1——► Wort aus der Sprache



Zustandsübergang $\overset{2}{\longleftrightarrow}$ Grammatikregel

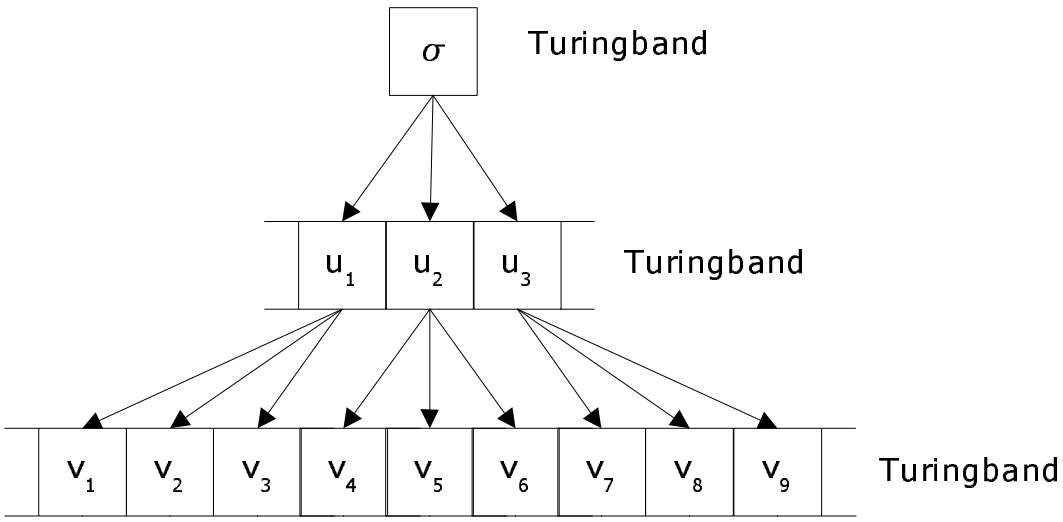
$$\delta(s,x) = (s',x',\begin{matrix}L\\R\\H\end{matrix}) \overset{2}{\longleftrightarrow} us\underbrace{xv'}_{v} \rightarrow ux's'v'$$

„=>“ Sei G = (N,T,P,σ) eine Grammatik, die L erzeugt (L(G) = L). Man betrachte die Mengen (M_i) der in i Ableitungsschritten ableitbaren Wörter für i = 0,1,2,...

$$M_0 = \{\sigma\},\ M_{i+1} = \{u \in (N \cup T^*) \mid \exists w \in M_i \ \ mit\, w \underbrace{\rightarrow}_{G} u\}$$

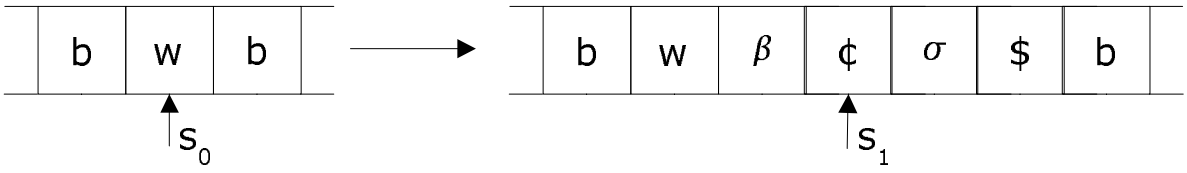
Idee:

Die TM erzeugt $M_0 \rightarrow M_1 \rightarrow M_2 \ldots$ in Form eines Breitendurchlaufs durch den Ableitungsbaum.

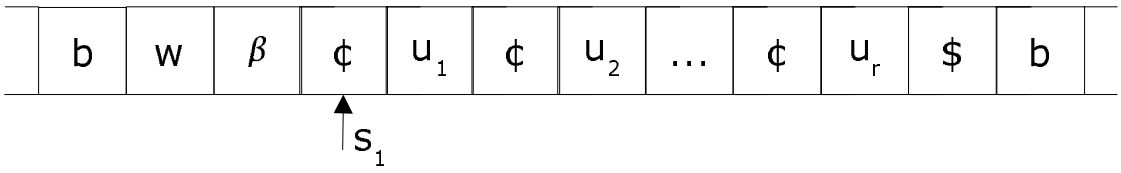


Es gilt $L(G) = \left(\bigcup_{i=9}^{\infty} M_i \right) \cap T^*$. Die Arbeitsweise einer Turingmaschine zur Berechnung von Ψ_L sei im folgenden skizzenhaft dargestellt. Eine vollständige Darstellung ist sehr umfangreich, da die Zustandsmenge riesig ist. Sie enthält das Wissen um die Produktionen der Grammatik.

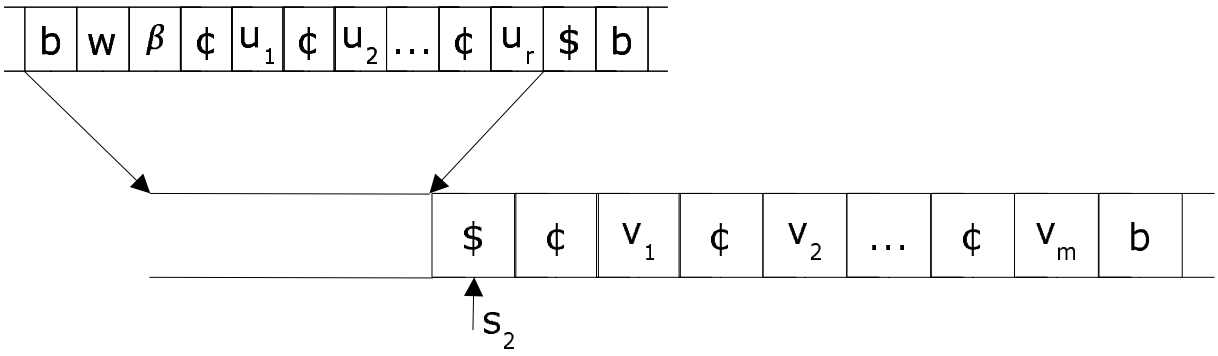
1. Die TM schreibt die Menge M_0 hinter das eingegebene Wort. Als Trennsymbol wird $\beta \notin N \cup T$ verwendet. Die Zeichen \$ und ¢ dienen zur Abgrenzung der Worte aus M_i und M_{i+1} und untereinander.



2. Wenn die Simulation



mit $M_i = \{u_1, \ldots, u_r\}$ vorliegt, dann schreibt die TM alle aus u_1 ableitbaren Wörter auf das Band, danach alle aus u_2 ableitbaren usw. bis die Menge $M_{i+1} = \{v_1, \ldots, v_r\}$ rechts hinter dem \$-Zeichen auf dem Band steht.



Danach werden die $v_1, \ldots, v_r$ nach links verschoben und treten an die Stelle der $u_1, \ldots, u_r$.

3. Vergleiche jedes v_k mit dem Wort w. Falls ein w = v_k existiert, lösche das Band und schreibe tt (Wort wurde akzeptiert).

Falls kein v_k mit w übereinstimmt, so gehe in die Konfiguration mit dem Zustand s_1 über und prüfe M_{i+1} gemäß 2. durch.

Die Schritte zwei und drei werden so lange wiederholt, bis das Wort erkannt wurde.

Man sieht:

Ist w in der Sprache, so muss das Verfahren terminieren, denn $\sigma \rightarrow^{s} w$ und $w \in M_s$. Die Turingmaschine hält an und liefert tt. Es folgt $\Psi_L(w) = tt$. Wenn das Wort w nicht in der Sprache enthalten ist $(w \notin L)$, dann gilt $\forall i \in \mathbb{N}_0 : w \notin M_i$. Die TM stoppt dann nicht, und $\Psi_L(w) = \perp$. Die Funktion Ψ_L wird von der TM berechnet.

Der Beweis in die Richtung „=>“ ist damit abgeschlossen.

Quellen

1.) http://grail.cba.csuohio.edu/~somos/bb.html