Inhaltsverzeichnis

Kapitel 3

Ergänzungen I : Weitere Modelle der Berechenbarkeit

3.1 Rekursive Funktionen

Im ersten Teil der Vorlesung haben wir berechenbare Funktionen als von Programmen bzw. TURING-Maschinen induzierte Funktionen eingeführt. In diesem Abschnitt gehen wir einen Weg, der etwas direkter ist. Wir werden Basisfunktionen definieren und diese als berechenbar ansehen. Mittels Operationen, bei denen die Berechenbarkeit nicht verlorengeht, werden dann weitere Funktionen erzeugt.

Wir geben nun die formalen Definitionen. Als *Basisfunktionen* betrachten wir:

- die nullstellige Funktion $Z_0 : \mathbf{N}_0 \to \mathbf{N}_0$, die den (konstanten) Wert 0 liefert,
- die Funktion $S : \mathbf{N}_0 \to \mathbf{N}_0$, bei der jeder natürlichen Zahl ihr Nachfolger zugeordnet wird,
- die Funktion $P : \mathbf{N}_0 \to \mathbf{N}_0$, bei der jede natürliche Zahl $n \geq 1$ auf ihren Vorgänger und die 0 auf sich selbst abgebildet wird,
- die Funktionen $P_i^n : \mathbf{N}_0^n \to \mathbf{N}_0$, die durch

$$P_i^n(x_1, x_2, \ldots, x_n) = x_i$$

  definiert sind.

Anstelle von $S(n)$ schreiben wir zukünftig auch - wie üblich - $n + 1$. P_i^n ist die übliche Projektion eines n-Tupels auf die i-te Komponente (Koordinate).

Als *Operationen* zur Erzeugung neuer Funktionen betrachten wir die beiden folgenden Schemata:

- *Kompositionsschema*: Für eine m-stellige Funktion g und m n-stellige Funktionen $f_1, f_2, \ldots, f_m$ definieren wir die n-stellige Funktion f vermöge

$$f(x_1, x_2, \ldots, x_n) = g(f_1(x_1, x_2, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_m(x_1, x_2, \ldots, x_n)).$$

- *Rekursionsschema*: Für fixierte natürliche Zahlen $x_1, x_2 \ldots, x_n$, eine n-stellige Funktion g und eine $(n+2)$-stellige Funktion h definieren wir die $(n+1)$-stellige Funktion f vermöge

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n), \\ f(x_1, x_2, \ldots, x_n, y+1) &= h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)). \end{aligned}$$

Zuerst erwähnen wir, dass das Kompositionsschema eine einfache Formalisierung des „Einsetzens“ ist.
Wir merken an, dass das gegebene Rekursionsschema eine parametrisierte Form der klassischen Rekursion

$$\begin{aligned} f(0) &= c \\ f(y+1) &= h(y, f(y)), \end{aligned}$$

wobei c eine Konstante ist, mit den Parametern $x_1, x_2, \ldots, x_n$ ist.
Für das Kompositionsschema ist sofort einzusehen, dass ausgehend von festen Funktionen $g, f_1, f_2, \ldots, f_m$ genau eine Funktion f definiert wird. Wir zeigen nun, dass dies auch für das Rekursionsschema gilt, wobei wir (um die Bezeichnungen einfach zu halten) dies nur für die klassische parameterfreie Form durchführen. Zuerst einmal ist klar, dass durch das Schema eine Funktion definiert wird (für $y = 0$ ist der Wert festgelegt, für $y \geq 1$ lässt er sich schrittweise aus der Rekursion berechnen). Wir zeigen nun die Eindeutigkeit. Seien dazu f_1 und f_2 zwei Funktionen, die den Gleichungen des Rekursionsschemas genügen. Mittels vollständiger Induktion beweisen wir nun $f_1(y) = f_2(y)$ für alle natürlichen Zahlen y. Laut Schema gilt für $y = 0$ die Beziehung

$$f_1(0) = f_2(0) = c,$$

womit der Induktionsanfang gezeigt ist. Sei die Aussage schon für alle natürlichen Zahlen $x \leq y$ bewiesen. Dann gilt

$$f_1(y+1) = h(y, f_1(y)) = h(y, f_2(y)) = f_2(y+1),$$

wobei die erste und letzte Gleichheit aus dem Rekursionsschema und die zweite Gleichheit aus der Induktionsvoraussetzung folgen.

Definition 3.1 *Eine Funktion $f : \mathbf{N}_0^n \to \mathbf{N}_0$ heißt primitiv-rekursiv, wenn sie mittels endlich oft wiederholter Anwendung von Kompositions- und Rekursionsschema aus den Basisfunktionen erzeugt werden kann.*

Wir lasssen dabei auch die 0-malige Anwendung, d. h. keine Anwendung, als endlich oftmalige Anwendung zu; sie liefert stets eine der Basisfunktionen.

Beispiel 3.2 a) Ausgehend von der Basisfunktion S gewinnen wir mittels Kompositionsschema die Funktion f mit $f(n) = S(S(n))$, aus der durch erneute Anwendung des Kompositionsschemas f' mit $f'(n) = S(f(n)) = S(S(S(n)))$ erzeugt werden kann. Offenbar ordnen f bzw. f' jeder natürlichen Zahl ihren zweiten bzw. dritten Nachfolger zu. Beide Funktionen sind nach Definition primitiv-rekursiv.

b) Wegen $x = P(S(x))$ ist die identische Funktion $id : \mathbf{N}_0 \to \mathbf{N}_0$ mit $id(x) = x$ ebenfalls primitiv-rekursiv.

c) Die nullstellige konstante Funktion Z_0 gehört zu den Basisfunktionen und ist daher primitiv-rekursiv. Wir zeigen nun, dass auch die n-stellige Funktion Z_n mit $Z_n(x_1, x_2, \ldots, x_n) = 0$ für alle $x_1, x_2, \ldots, x_n$ ebenfalls primitiv-rekursiv ist.
Es sei $n = 1$. Dann betrachten wir das Rekursionsschema

$$Z_1(0) = Z_0 \quad \text{und} \quad Z_1(y+1) = P_2^2(y, Z_1(y))\,.$$

Wir zeigen mittels vollständiger Induktion, dass Z_1 die einstellige konstante Funktion mit dem Wertevorrat 0 ist. Offensichtlich gilt $Z_1(0) = 0$, da Z_0 den Wert 0 liefert. Sei nun schon $Z_1(y) = 0$ gezeigt. Dann ergibt sich aus der zweiten Rekursionsgleichung sofort $Z_1(y+1) = P_2^2(y, 0) = 0$, womit der Induktionsschritt vollzogen ist.
Nehmen wir nun an, dass wir bereits die n-stellige konstante Funktion Z_n mit dem Wert 0 als primitiv-rekursiv nachgewiesen haben, so können wir analog zu Obigem zeigen, dass das Rekursionsschema

$$\begin{aligned} Z_{n+1}(x_1, x_2, \ldots x_n, 0) &= Z_n(x_1, x_2, \ldots x_n)\,, \\ Z_{n+1}(x_1, x_2, \ldots x_n, y+1) &= P_{n+2}^{n+2}(x_1, x_2, \ldots, x_n, y, Z_{n+1}(x_1, x_2, \ldots x_n, y)) \end{aligned}$$

die $(n+1)$-stellige konstante Funktion Z_{n+1} mit dem Wert 0 liefert.

d) Die Addition und Multiplikation natürlicher Zahlen lassen sich mittels der Rekursionsschemata

$$\begin{aligned} add(x, 0) &= id(x), \\ add(x, y+1) &= S(P_3^3(x, y, add(x, y))) \end{aligned}$$

und

$$\begin{aligned} mult(x, 0) &= Z_1(x), \\ mult(x, y+1) &= add(P_1^3(x, y, mult(x, y)), P_3^3(x, y, mult(x, y))) \end{aligned}$$

definieren. Da die Identität, S, Z und die Projektionen bereits als primitiv-rekursiv nachgewiesen sind, ergibt sich damit die primitive Rekursivität von *add* und aus der dann die von *mult*.
Entsprechend unserer obigen Bemerkung ist klar, dass durch diese Schemata eindeutige Funktionen definiert sind. Durch einfaches „Nachrechnen“ überzeugt man sich davon, dass es sich wirklich um Addition und Multiplikation handelt, z.B. bedeutet die letzte Relation mit der üblichen Notation $add(x, y) = x + y$ und $mult(x, y) = x \cdot y$ nichts anderes als das bekannte Distributivgesetz

$$mult(x, y+1) = x \cdot (y+1) = x + x \cdot y = add(x, mult(x, y)).$$

d) Durch das Rekursionsschema

$$sum(0) = 0 \quad \text{und} \quad sum(y+1) = S(add(y, sum(y)))$$

wird die Funktion

$$sum(y) = \sum_{i=0}^{y} i = \frac{y(y+1)}{2}$$

definiert, wovon man sich leicht mittels vollständiger Induktion überzeugen kann.

Wir betrachten nun die folgende rekursive Definition der Fibonacci-Folge:

$$\begin{aligned} f(0) &= 1, \quad f(1) = 1, \\ f(y+2) &= f(y+1) + f(y). \end{aligned}$$

Für diese Rekursion ist nicht offensichtlich, dass sie durch das obige Rekursionsschema realisiert werden kann, da nicht nur auf den Wert $f(y)$ rekursiv zurückgegriffen wird. Die Rekursion für die Fibonacci-Folge lässt sich aber unter Verwendung von zwei Funktionen so umschreiben, dass jeweils nur die Kenntnis der Werte an der Stelle y erforderlich ist. Dies wird durch das Schema

$$\begin{aligned} &f_1(0) = 1, && f_2(1) = 1, \\ &f_1(y+1) = f_2(y), && f_2(y+1) = f_1(y) + f_2(y) \end{aligned}$$

geleistet. Hiervon ausgehend führen wir die folgende Verallgemeinerung des Rekursionsschemas, simultane Rekursion genannt, ein: Für n-stellige Funktionen g_i und die $(n+m+1)$-stelligen Funktionen h_i, $1 \leq i \leq m$, definieren wir simultan die $(n+1)$-stelligen Funktionen f_i, $1 \leq i \leq m$, durch

$$\begin{aligned} f_i(x_1, \ldots, x_n, 0) &= g_i(x_1, \ldots, x_n), \ 1 \leq i \leq m, \\ f_i(x_1, \ldots, x_n, y+1) &= h_i(x_1, \ldots, x_n, y, f_1(x_1, \ldots, x_n, y), \ldots, f_m(x_1, \ldots, x_n, y)). \end{aligned}$$

Wir wollen nun zeigen, dass die simultane Rekursion auch nur die Erzeugung primitiv-rekursiver Funktionen gestattet. Um die Notation nicht unnötig zu verkomplizieren werden wir die Betrachtungen nur für den Fall $n = 1$ und $m = 2$ durchführen.
Seien die Funktionen C, E, D_1 und D_2 mittels der Funktionen $\ominus$ und div aus Übungsaufgabe 10 von Abschnitt 1 durch

$$\begin{aligned} C(x_1, x_2) &= sum(x_1 + x_2) + x_2, \\ E(0) &= 0, \quad E(n+1) = E(n) + (n \ div \ sum(E(n)+1)), \\ D_1(n) &= E(n) + sum(E(n)) \ominus n, \\ D_2(n) &= E(n) \ominus D_1(n) \end{aligned}$$

definiert. Entsprechend der Konstruktion und Übungsaufgabe 10 von Abschnitt 1 sind alle diese Funktionen primitiv-rekursiv. Weiterhin rechnet man nach, dass die folgenden Bedingungen erfüllt sind: Für alle natürlichen Zahlen n, n_1 und n_2 gilt

$$C(D_1(n), D_2(n)) = n, \ D_1(C(n_1, n_2)) = n_1, \ D_2(C(n_1, n_2)) = n_2.$$

Zur Veranschaulichung betrachte man Abbildung 3.1 und prüfe nach, dass durch $E(n)$ die Nummer der Diagonalen in der n steht, durch $x_1 + x_2$ die Nummer der Diagonalen, in der sich die Spalte von x_1 und die Zeile von x_2 kreuzen, durch $C(x_1, x_2)$ das im Kreuzungspunkt der Spalte zu x_1 und der Zeile zu x_2 stehende Element, durch D_1 und D_2 die Projektionen von einem Element gegeben werden.
Dann definieren wir für die gegebenen Funktionen g_i und h_i, $1 \leq i \leq m$, die Funktionen g und h durch

$$\begin{aligned} g(x) &= C(g_1(x), g_2(x)), \\ h(x, y, z) &= C(h_1(x, y, D_1(z), D_2(z)), h_2(x, y, D_1(z), D_2(z))) \end{aligned}$$

x_1 / x_2	0	1	2	3	4	...
0	0	1	3	6	10	...
1	2	4	7	11	...	
2	5	8	12	...		
3	9	13	...			
4	14	...				
...	...					

Abbildung 3.1:

die Funktion f durch das Rekursionsschema

$$\begin{aligned} f(x,0) &= g(x), \\ f(x,y+1) &= h(x,y,f(x,y)) \end{aligned}$$

und die Funktionen f_1 und f_2, die durch das simultane Rekursionsschema erzeugt werden sollen, durch

$$f_1(x,y) = D_1(f(x,y)) \text{ und } f_2(x,y) = D_2(f(x,y)).$$

Wegen

$$f_i(x,0) = D_i(f(x,0)) = D_i(g(x)) = D_i(C(g_1(x), g_2(x)) = g_i(x)$$

für $i \in \{1,2\}$, sind die Ausgangsbedingungen des verallgemeinerten Rekursionsschemas erfüllt, und analog zeigt man, dass die Rekursionsbedingungen befriedigt werden. Diese Konstruktion von f_1 und f_2 erfordert nur das ursprüngliche Rekursionsschema (für die Funktion f) und das Kompositionsschema, womit gezeigt ist, dass diese beiden Funktionen primitiv-rekursiv sind.

Aufgrund der eben gezeigten Äquivalenz von Rekursionsschema und simultanem Rekursionsschema werden wir zukünftig auch von der simultanen Rekursion Gebrauch machen, um zu zeigen, dass gewisse Funktionen primitiv-rekursiv sind.

Satz 3.3 *Eine Funktion f ist genau dann primitiv-rekursiv, wenn sie* **LOOP***-berechenbar ist.*

Beweis: Wir zeigen zuerst mittels Induktion über die Anzahl k der Operationen zur Erzeugung der primitiv-rekursiven Funktion f, dass f auch **LOOP**-berechenbar ist.
Sei $k = 0$. Dann muss die zu betrachtende Funktion f eine Basisfunktion sein. Die Tabelle in Abbildung 3.2 gibt zu jeder Basisfunktion f ein **LOOP**-Programm Π mit $\Phi_{\Pi,1} = f$. Damit ist der Induktionsanfang gesichert.
Wir führen nun den Induktionsschritt durch. Sei dazu f eine Funktion, die durch k-malige, $k \geq 1$, Anwendung der Operationen erzeugt wurde. Dann gibt es eine Operation, die als letzte angewendet wurde. Hiernach unterscheiden wir zwei Fälle, welche der beiden Operationen dies ist.
Fall 1. Kompositionsschema. Dann gilt

$$f(x_1, x_2, \ldots, x_n) = g(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)),$$

f	Π
Z	$x_1 := 0$
S	$x_1 := S(x_1)$
P	$x_1 := P(x_1)$
P_i^n	$x_1 := x_i$

Abbildung 3.2:

wobei die Funktionen $g, f_1, f_2, \ldots, f_m$ alle durch höchstens $(k-1)$-malige Anwendung der Operationen entstanden sind. Nach Induktionsannahme gibt es also Programme $\Pi, \Pi_1, \Pi_2, \ldots, \Pi_m$ derart, dass

$$\Phi_{\Pi,1} = g \text{ und } \Phi_{\Pi_i,1} = f_i \text{ für } 1 \leq i \leq m$$

gelten. Nun prüft man leicht nach, dass das Programm

$x_{n+1} := x_1; x_{n+2} := x_2; \ldots; x_{2n} := x_n;$
$\Pi_1; x_{2n+1} := x_1; x_1 := x_{n+1}; x_2 := x_{n+2}; \ldots; x_n := x_{2n};$
$\Pi_2; x_{2n+2} := x_1; x_1 := x_{n+1}; x_2 := x_{n+2}; \ldots; x_n := x_{2n};$
...
$\Pi_m; x_{2n+m} := x_1;$
$x_1 := x_{2n+1}; x_2 := x_{2n+2}; \ldots; x_m := x_{2n+m}; \Pi$

die Funktion f berechnet (die Setzungen $x_{n+i} := x_i$ stellen ein Abspeichern der Eingangswerte für die Variablen x_i dar; durch die Anweisungen $x_i := x_{n+i}$ wird jeweils gesichert, dass die Programme Π_j mit der Eingangsbelegung der x_i arbeiten, denn bei der Abarbeitung von Π_{j-1} kann die Belegung der x_i geändert worden sein; die Setzungen $x_{2n+j} := x_1$ speichern die Werte $f_j(x_1, x_2, \ldots, x_n)$, die durch die Programme Π_j bei der Variablen x_1 entsprechend der berechneten Funktion erhalten werden; mit diesen Werten wird dann aufgrund der Anweisungen $x_j := x_{2n+j}$ das Programm Π gestartet und damit der nach Kompositionsschema gewünschte Wert berechnet).
Fall 2. Rekursionsschema. Die Funktion f werde mittels Rekursionsschema aus den n- bzw. $(n+2)$-stelligen Funktionen g (an der Stelle $y = 0$) und h (für die eigentliche Rekursion) erzeugt. Da sich diese beiden Funktionen durch höchstens $(k-1)$-malige Anwendung der Schemata erzeugen lassen können, gibt es für sie Programme Π über den Variablen $x_1, x_2, \ldots x_n$ und Π' über den Variablen $x_1, x_2, \ldots, y, z$ mit $\Phi_{\Pi,1} = g$ und $\Phi_{\Pi',1} = h$ (wobei wir zur Vereinfachung nicht nur Variable der Form x_i, wie in Abschnitt 1.1.1 gefordert, verwenden). Wir betrachten das folgende Programm:

$y := 0; x_{n+1} := x_1; x_{n+2} := x_2; \ldots x_{2n} := x_n; \Pi; z := x_1;$
LOOP y' **BEGIN** $x_1 := x_{n+1}; \ldots x_n := x_{2n}; \Pi'; z := x_1; y := S(y)$ **END**;
$x_1 := z$

und zeigen, dass dadurch der Wert $f(x_1, x_2, \ldots, x_n, y')$ berechnet wird.
Erneut wird durch die Variablen x_{n+i} die Speicherung der Anfangsbelegung der Variablen x_i gewährleistet. Ist $y' = 0$, so werden nur die erste und dritte Zeile des Programms realisiert. Daher ergibt sich der Wert von Π bei der ersten Variablen, und weil Π die Funktion g berechnet, erhalten wir $g(x_1, x_2, \ldots, x_n)$, wie es das Rekursionsschema für

$f(x_1, x_2, \ldots, x_n, 0)$ erfordert. Ist dagegen $y' > 0$, so wird innerhalb der **LOOP**-Anweisung mit $z = f(x_1, x_2, \ldots, x_n, y)$ der Wert $f(x_1, x_2, \ldots, y+1)$ berechnet und die Variable y um Eins erhöht. Da dies insgesamt von $y = 0$ und $f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n)$ (aus der ersten Zeile) ausgehend, y'-mal zu erfolgen hat, wird tatsächlich $f(x_1, x_2, \ldots, x_n, y')$ als Ergebnis geliefert.
Damit ist der Induktionsbeweis vollständig.

Wir zeigen nun die umgekehrte Richtung. Wir gehen analog vor, werden vollständige Induktion über die Programmtiefe t benutzen und sogar zeigen, dass jede von einem **LOOP**-Programm Π berechnete Funktion $\Phi_{\Pi,j}$, $j \geq 1$ eine primitiv-rekursive Funktion ist.
Es sei $t = 1$. Dann bestehen die Programme aus den Wertzuweisungen. Wenn wir die im ersten Teil dieses Beweises gegebenen Tabelle von rechts nach links lesen, finden wir zu jeder derartigen Wertzuweisung die zugehörige primitiv-rekursive Funktion, die identisch mit der vom Programm berechneten Funktion ist. Damit ist der Induktionsanfang gesichert.
Sei nun Π ein Programm der Tiefe $t > 1$. Dann gilt $\Pi = \Pi_1; \Pi_2$ oder Π=**LOOP** y **BEGIN** Π' **END** für gewisse Programme Π_1, Π_2, Π' mit einer Tiefe $\leq t - 1$. Nach Induktionsannahme sind dann alle Funktionen $\Phi_{\Pi_1,j}, \Phi_{\Pi_2,j}, \Phi_{\Pi',j}$ primitiv-rekursiv.
Ist Π als Nacheinanderausführung von Π_1 und Π_2 gegeben, so ergeben sich für die von Π berechneten Funktionen die Beziehungen

$$\Phi_{\Pi,j}(x_1, \ldots x_n) = \Phi_{\Pi_2,j}(\Phi_{\Pi_1,1}(x_1, \ldots, x_n), \Phi_{\Pi_1,2}(x_1, \ldots, x_n), \ldots, \Phi_{\Pi_1,m}(x_1, \ldots, x_n)).$$

Damit entstehen die von Π berechneten Funktionen mittels des Kompositionsschems aus primitiv-rekursiven Funktionen und sind daher selbst primitiv-rekursiv.
Sei nun Π=**LOOP** y **BEGIN** Π' **END**, wobei wir ohne Beschränkung der Allgemeinheit annehmen, dass y nicht unter den Variablen $x_1, x_2, \ldots, x_n$ des Programms Π' vorkommt (siehe Übungsaufgabe 5). Dann werden die von Π berechneten Funktionen durch das folgende simultane Rekursionsschema bestimmt:

$$\begin{aligned} \Phi_{\Pi,j}(x_1, \ldots, x_n, 0) &= P^n_j(x_1, \ldots, x_n), \\ \Phi_{\Pi,j}(x_1, \ldots, x_n, y+1) &= \Phi_{\Pi',j}(\Phi_{\Pi,1}(x_1, \ldots, x_n, y), \ldots, \Phi_{\Pi,n}(x_1, \ldots, x_n, y)) \end{aligned}$$

(die erste Gleichung legt den Wert der Variablen vor Abarbeitung des Programms fest; um zum Wert für $y > 0$ zu kommen, wird das Programm Π' entsprechend der zweiten Gleichung stets wieder ausgeführt, wobei die im vorhergehenden Schritt erhaltenen Funktionswerte als Eingaben dienen (siehe Kompositionsschema); wie beim **LOOP**-Programm ist y-malige Nacheinanderausführung zur Gewinnung von $\Phi_{\Pi,j}(x_1, \ldots, x_n, y)$ notwendig.
Damit ist der Induktionsbeweis auch für diese Richtung geführt. □

Wir wollen nun eine weitere Operation zur Erzeugung von Funktionen einführen, die es uns gestattet, auch partielle Funktionen zu erhalten (mittels Kompositions- und Rekursionsschema erzeugte Funktionen sind offenbar total).

- *μ-Operator*: Für eine $(n+1)$-stellige Funktion h definieren wir die n-stellige Funktion f wie folgt. $f(x_1, x_2, \ldots, x_n) = z$ gilt genau dann, wenn die folgenden Bedingungen erfüllt sind:

- $h(x_1, x_2 \ldots, x_n, y)$ ist für alle $y \leq z$ definiert,
- $h(x_1, x_2, \ldots, x_n, y) \neq 0$ für $y < z$,
- $h(x_1, x_2, \ldots, x_n, z) = 0$.

Wir benutzen die Bezeichnungen

$$f(x_1, \ldots, x_n) = (\mu y)[h(x_1, \ldots, x_n, y) = 0] \quad \text{bzw.} \quad f = (\mu y)[h].$$

Intuitiv bedeutet dies, dass für die festen Parameter $x_1, x_2, \ldots, x_n$ der kleinste Wert von z bestimmt wird, für den $h(x_1, x_2, \ldots, x_n, z) = 0$ gilt (wobei bei nicht überall definierten Funktionen zusätzlich verlangt wird, dass für alle kleineren Werte y als z das Tupel $(x_1, x_2, \ldots x_n, y)$ im Definitionsbereich liegt, sonst ist f an dieser Stelle nicht definiert).

Beispiel 3.4 a) Es gilt

$$(\mu y)[add(x, y)] = \begin{cases} 0 & \text{für } x = 0 \\ nicht\ definiert & \text{sonst} \end{cases}$$

(für $x = 0$ ist wegen $0 + 0 = 0$ offenbar $z = 0$ der gesuchte minimale Wert; für $x > 0$ gilt auch $x + y > 0$ für alle y, und daher existiert kein z mit der dritten Eigenschaft aus der Definition des μ-Operators).

b) Es sei

$$h(x, y) = |9x^2 - 10xy + y^2|.$$

Durch Anwendung des μ-Operators auf h entsteht die Identität, d.h. f mit $f(x) = x$ für alle x.

Dies ist wie folgt leicht zu sehen. Für einen fixierten Wert von x ist $(\mu y)[h(x, y)]$ die kleinste natürliche Nullstelle des Polynoms $9x^2 - 10xy + y^2$ in der Unbestimmten y. Eine einfache Rechnung ergibt die Nullstellen x und $9x$. Somit gilt

$$f(x) = (\mu y)[h(x, y)] = x.$$

Wir wollen nun eine Erweiterung der primitiv-rekursiven Funktionen mittels μ-Operator entsprechend Definition 3.1 vornehmen. Da jedoch durch die Anwendung des μ-Operators Funktionen entstehen können, deren Definitionsbereiche echte Teilmengen von $\mathbf{N}_0^n$ sind, müssen wir zuerst Kompositions- und Rekursionsschema auf diesen Fall ausdehnen.

Beim Kompositionsschema ist $f(x_1, x_2, \ldots, x_n)$ genau dann definiert, wenn für $1 \leq i \leq n$ die Funktionen f_i auf dem Tupel $\underline{x} = (x_1, x_2, \ldots, x_n)$ und g auf $(f_1(\underline{x}), f_2(\underline{x}), \ldots, f_m(\underline{x}))$ definiert sind. In ähnlicher Weise kann das Rekursionsschema erweitert werden; die Details dazu überlassen wir dem Leser.

Definition 3.5 *Eine Funktion $f : \mathbf{N}_0^n \to \mathbf{N}_0$ heißt partiell-rekursiv, wenn sie mittels endlich oft wiederholter Anwendung von Kompositionsschema, Rekursionsschema und μ-Operator aus den Basisfunktionen erzeugt werden kann.*

Satz 3.6 *Eine Funktion ist genau dann partiell-rekursiv, wenn sie* **LOOP/WHILE**-*berechenbar ist.*

Beweis: Wir gehen wie beim Beweis von Satz 3.3 vor.
Daher reicht es in der ersten Richtung zusätzlich zu den dortigen Fakten zu zeigen, dass jede partiell-rekursive Funktion f, die durch Anwendung des μ-Operators auf h entsteht, **LOOP**/**WHILE**-berechenbar ist. Nach Induktionsannahme ist h **LOOP**/**WHILE**-berechenbar, also $h = \Phi_{\Pi,1}$ für ein Programm Π. Um den minimalen Wert z zu berechnen, berechnen wir der Reihe nach die Werte y' an den Stellen mit $0, 1, 2, \ldots$ für die Variable y und testen jeweils, ob der aktuelle Wert von y' von Null verschieden ist. Formal ergibt sich folgendes Programm für f:

$y := 0; x_{n+1} := x_1; \ldots x_{2n} := x_n; \Pi; y' := x_1;$
WHILE $y' \neq 0$ **BEGIN** $y := S(y); x_1 := x_{n+1}, \ldots, x_n := x_{2n}; \Pi; y' := x_1$ **END**;
$x_1 := y$

In der umgekehrten Richtung ist noch die **WHILE**-Anweisung zusätzlich zu betrachten. Sei also $\Pi'' =$ **WHILE** $x_k \neq 0$ **BEGIN** Π' **END**, wobei nach Induktionsannahme alle Funktionen $\Phi_{\Pi',j}$ partiell-rekursiv sind. Wir konstruieren zuerst die gleichen Funktionen $\Phi_{\Pi,j}$ wie bei der Umsetzung der **LOOP**-Anweisung im Beweis von Satz 3.3. Nach den dortigen Überlegungen gibt $\Phi_{\Pi,j}(x_1, x_2, \ldots x_n, y)$ den Wert der Variablen x_j nach y-maliger Hintereinanderausführung von Π' an. Die $\Phi_{\Pi,j}$ sind partiell-rekursive Funktionen, da sie durch Anwendung des simultanen Rekursionsschemas auf partiell-rekursive Funktionen entstanden sind. Wir betrachten nun die Funktion

$$w(x_1, \ldots, x_n) = (\mu y)[\Phi_{\Pi,k}(x_1, \ldots, x_n, y)],$$

die nach Definition die kleinste Zahl von Durchläufen von Π' liefert, um den Wert 0 bei der Variablen x_k zu erreichen. Entsprechend der Semantik der **WHILE**-Anweisung bricht diese genau nach $w(x_1, \ldots, x_n)$ Schritten ab. Folglich gilt

$$\Phi_{\Pi'',i}(x_1, \ldots, x_n) = \Phi_{\Pi,i}(x_1, \ldots, x_n, w(x_1, \ldots, x_n))$$

(da die rechten Seiten den Wert der Variablen x_i nach $w(x_1, \ldots, x_n)$ Hintereinanderausführungen von Π' und damit bei Abbruch der **WHILE**-Anweisung angeben). Entsprechend dieser Konstruktion ist damit jede von Π'' berechnete Funktion partiell-rekursiv. □

Wir bemerken, dass die Beweise der Sätze 3.3 und 3.6 eine enge Nachbarschaft zwischen dem Berechenbarkeitsbegriff auf der Basis von **LOOP**/**WHILE**-Programmen einerseits und partiell-rekursiven Funktionen andererseits ergibt, da die Wertzuweisungen den Basisfunktionen, die Hintereinanderausführung von Programmen dem Kompositionsschema, die **LOOP**-Anweisung dem Rekursionsschema und die **WHILE**-Anweisung dem μ-Operator im wesentlichen entsprechen.

Durch Kombination der Sätze 1.10 und 3.6 erhalten wir die folgende Aussage.

Folgerung 3.7 *Es gibt eine totale Funktion, die nicht partiell-rekursiv ist.* □

3.2 Registermaschinen

Wir wollen nun einen Berechenbarkeitsbegriff behandeln, der auf einer Modellierung der realen Rechner basiert.

Definition 3.8 *i) Eine Registermaschine besteht aus den Registern*

$$B,\ C_0, C_1, C_2, \ldots, C_n, \ldots$$

und einem Programm.
B heißt Befehlszähler, C_0 heißt Arbeitsregister oder Akkumulator, und jedes der Register C_i, $i \geq 1$, heißt Speicherregister.
Jedes Register enthält als Wert eine natürliche Zahl.
ii) Unter einer Konfiguration der Registermaschine verstehen wir das unendliche Tupel

$$(b, c_0, c_1, \ldots, c_n, \ldots),$$

wobei

- *das Register B die Zahl $b \in \mathbf{N}_0$ enthält,*
- *für $n \geq 0$ das Register C_n die Zahl $c_n \in \mathbf{N}_0$ enthält.*

iii) Das Programm ist eine endliche Folge von Befehlen. Durch die Anwendung eines Befehls wird die Konfiguration der Registermaschine geändert. Die folgende Liste gibt die zugelassenen Befehle und die von ihnen jeweils bewirkte Änderung der Konfiguration $(b, c_o, c_1, \ldots, c_n, \ldots)$ in die Konfiguration $(b', c'_0, c'_1, \ldots, c'_n, \ldots)$ an, wobei für die nicht angegebenen Komponenten $u' = u$ gilt:

Ein- und Ausgabebefehle:

`LOAD` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = c_i$
`ILOAD` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = c_{c_i}$
`CLOAD` i ,	$i \in \mathbf{N}_0$	$b' = b + 1$	$c'_0 = i$
`STORE` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_i = c_0$
`ISTORE` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_{c_i} = c_0$

Arithmetische Befehle:

`ADD` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = c_0 + c_i$
`CADD` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = c_0 + i$
`SUB` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = \begin{cases} c_0 - c_i & \text{für } c_0 \geq c_i \\ 0 & \text{sonst} \end{cases}$
`CSUB` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = \begin{cases} c_0 - i & \text{für } c_0 \geq i \\ 0 & \text{sonst} \end{cases}$
`MULT` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = c_0 * c_i$
`CMULT` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = c_0 + i$
`DIV` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = \begin{cases} \lfloor c_0/c_i \rfloor & \text{für } c_i > 0 \\ \textit{nicht definiert} & \text{sonst} \end{cases}$
`CDIV` i ,	$i \in \mathbf{N}$	$b' = b + 1$	$c'_0 = \lfloor c_0/i \rfloor$

Sprungbefehle:

`GOTO` i ,	$i \in \mathbf{N}$	$b' = i$
`IF` $c_0 = 0$ `GOTO` i ,	$i \in \mathbf{N}$	$b' = \begin{cases} i & \textit{falls } c_0 = 0 \\ b+1 & \textit{sonst} \end{cases}$

Stopbefehl:

`END`

Eine Registermaschine lässt sich entsprechend Abb. 3.3 veranschaulichen.

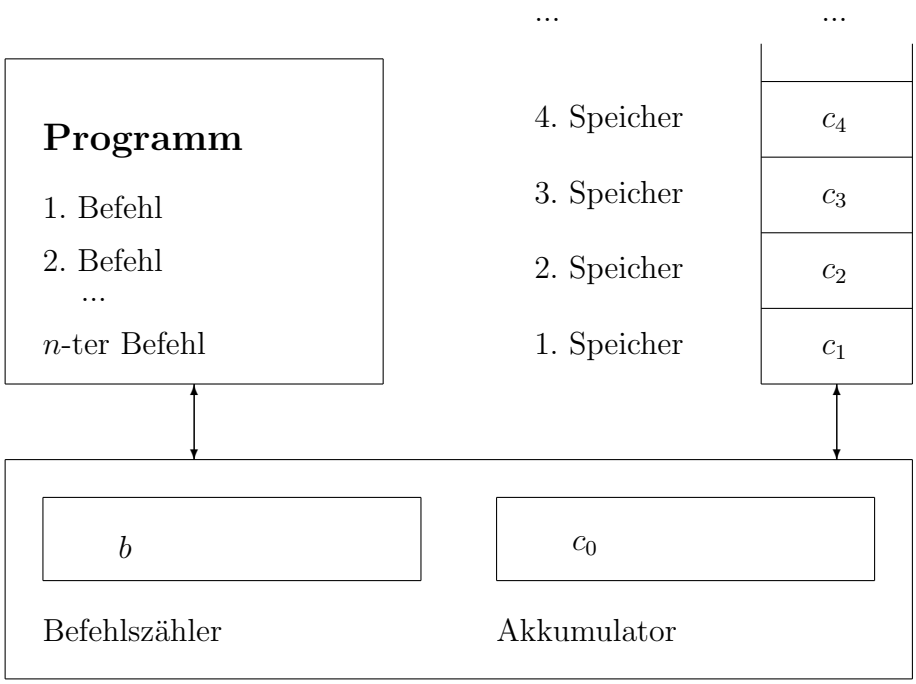

Abbildung 3.3: Registermaschine

Bei den Eingabebefehlen `LOAD` i bzw. `CLOAD` i wird der Wert des i-ten Registers bzw. die Zahl i in den Akkumulator geladen; bei `STORE` i wird der Wert des Akkumulators in das i-te Speicherregister eingetragen. Es sei j der Inhalt des i-ten Registers (d.h. $c_i = j$); dann werden durch die Befehle `ILOAD` i bzw. `ISTORE` i mit indirekter Adressierung der Inhalt des Registers j in den Akkumulator geladen bzw. der Inhalt des Akkumulators in das j-te Register gespeichert.
Bei den Befehlen `ADD` i, `SUB` i, `MULT` i und `DIV` i erfolgt eine Addition, Subtraktion, Multiplikation und Division des Wertes des Akkumulators mit dem Wert des i-ten Speicherregisters. Da die Operationen nicht aus dem Bereich der natürlichen Zahlen herausführen sollen, wird die Subtraktion nur dann wirklich ausgeführt, wenn der Subtrahend nicht kleiner als der Minuend ist und sonst 0 ausgegeben; analog erfolgt die Division nur ganzzahlig.
Die Befehle `CADD` i, `CSUB` i, `CMULT` i und `CDIV` i arbeiten analog, nur dass anstelle des Wertes des i-ten Registers die natürliche Zahl i benutzt wird. Dadurch werden auch arithmetische Operationen mit Konstanten möglich.

In all diesen Fällen wird der Wert des Befehlsregisters um 1 erhöht, d.h. der nächste Befehl des Programms wird abgearbeitet. Dies ist bei den Sprungbefehlen grundsätzlich anders. Bei `GOTO` i wird als nächster Befehl der i-te Befehl des Programms festgelegt, während bei der `IF`-Anweisung in Abhängigkeit von dem Erfülltsein der Bedingung $c_0 = 0$ der nächste Befehl der i-te bzw. der im Programm auf die `IF`-Anweisung folgende Befehl des Programms ist.
Der Befehl `END` ist ein Stopbefehl.

Definition 3.9 *Es sei M eine Registermaschine wie in Definition 3.8. Die von M induzierte Funktion $f_M : \mathbf{N}_0^n \longrightarrow \mathbf{N}_0$ ist wie folgt definiert: $f_M(x_1, x_2, \ldots x_n) = y$ gilt genau dann, wenn M ausgehend von der Konfiguration $(1, 0, x_1, x_2, \ldots, x_n, 0, 0, \ldots)$ die Konfiguration $(b, c_0, y, c_2, c_3, \ldots)$ für gewisse $b, c_0, c_2, c_3, \ldots$ erreicht und der b-te Befehl des Programm* `END` *ist.*

Entsprechend dieser Definition gehen wir davon aus, dass zu Beginn der Arbeit der Registermaschine die ersten n Speicherregister die Werte $x_1, x_2, \ldots, x_n$ und der Akkumulator und alle anderen Speicherregister den Wert 0 enthalten, die Abarbeitung des Programms mit dem ersten Befehl begonnen wird und bei Erreichen eines `END`-Befehls beendet wird und dann das Ergebnis y im ersten Speicherregister abgelegt ist (die Inhalte der anderen Register interessieren nicht).

Wir geben nun drei Beispiele.

Beispiel 3.10 Wir betrachten die Registermaschine M_1 mit dem Programm aus Abb. 3.4.

```
 1  CLOAD 1
 2  STORE 3
 3  LOAD 2
 4  IF c0 = 0 GOTO 12
 5  LOAD 3
 6  MULT 1
 7  STORE 3
 8  LOAD 2
 9  CSUB 1
10  STORE 2
11  GOTO 4
12  LOAD 3
13  STORE 1
14  END
```

Abbildung 3.4: Programm der Registermaschine aus Beispiel 3.10

Das Programm geht davon aus, dass im ersten und zweiten Register Werte stehen (Befehle 3 bzw. 6), so dass wir davon ausgehen, dass M_1 eine zweistellige Funktion $f_{M_1}(x, y)$ berechnet, wobei x und y zu Beginn im ersten bzw. zweiten Speicherregister stehen.

M_1 verhält sich wie folgt: Mittels der ersten zwei Befehle wird der Wert 1 in das dritte Register geschrieben. Die Befehle 4 – 11 bilden eine Schleife, die sooft durchlaufen wird, wie y angibt, denn bei jedem Durchlauf wird y um 1 verringert (Befehle 8 – 10). Ferner erfolgt bei jedem Durchlauf der Schleife eine Multiplikation des Inhalts des dritten Registers mit x (Befehle 5 – 7). Abschließend wird der Inhalt des dritten Registers in das erste umgespeichert, weil dort nach Definition das Ergebnis zu finden ist. Folglich induziert diese Registermaschine die Funktion

$$f_{M_1}(x, y) = 1 \cdot \underbrace{x \cdot x \cdot \ldots \cdot x}_{y \ mal} = x^y.$$

M_1 berechnet also die Potenzfunktion.

Beispiel 3.11 Wir betrachten die Registermaschine M_2 mit dem Programm aus Abb. 3.5 und zu Beginn der Arbeit stehe nur im ersten Register ein möglicherweise von Null verschiedener Wert (in den anderen Registern steht also eine Null).

```
 1  LOAD 1
 2  IF c_0 = 0 GOTO 12
 3  LOAD 2
 4  CADD 1
 5  STORE 2
 6  ADD 3
 7  STORE 3
 8  LOAD 1
 9  CSUB 1
10  STORE 1
11  GOTO 1
12  LOAD 3
13  STORE 1
14  END
```

Abbildung 3.5: Programm der Registermaschine aus Beispiel 3.11

Steht im ersten Register eine Null, so werden wegen des zweiten Befehls die Befehle 12–14 abgearbeitet, durch die die Ausgabe 0 erzeugt wird. Anderenfalls erfolgt ein Durchlaufen der Befehle 3–5, durch die der Inhalt des Registers 2 um 1 erhöht wird, und anschließend der Befehle 6–7, durch die eine Addition der Werte der Register 2 (nach der Erhöhung) und 3 erfolgt, deren Resultat wieder in Register 3 abgelegt wird. Danach wird der Wert des Registers 1 um 1 erniedrigt. Diese Befehle werden solange durchgeführt, wie in Register 1 keine 0 steht, d.h. diese Schleife wird n mal durchlaufen, wenn n zu Beginn in Register 1 steht, da dieses Register bei jedem Durchlauf um 1 verringert wird. In Register 2 stehen während der Durchläufe nacheinander die Zahlen 1, 2, $\ldots n$, die in Register 3 aufaddiert werden. Da der Inhalt des dritten Registers das Resultat liefert, erhalten wir

$$f_{M_2}(n) = \sum_{i=1}^{n} i\,.$$

Beispiel 3.12 Wir gehen jetzt umgekehrt vor. Wir geben uns eine Funktion vor und wollen eine Registermaschine konstruieren, die diese Funktion induziert. Dazu betrachten wir die auf einem Feld (oder Vektor) $(x_1, x_2, \ldots, x_n)$ und seiner Länge n durch definierte Funktion

$$f(n, x_1, x_2, \ldots, x_n) = \begin{cases} \sum_{i=1}^{n} x_i & n \geq 1 \\ 0 & n = 0 \end{cases} \tag{3.1}$$

definierte Funktion.
Wir konstruieren eine Registermaschine, bei der zu Beginn n im ersten Register, x_i im $i+1$-ten Register und 0 in allen anderen Registern steht. Die Addition der Elemente des Feldes realisieren wir, indem wir zum Inhalt des zweiten Registers der Reihe nach die Werte $x_n, x_{n-1}, \ldots, x_2$ aus den Registern $n+1, n, \ldots, 3$ addieren. Hierbei greifen wir durch indirekte Adressierung immer auf das entsprechende i-te Register zu, indem wir das erste Register zuerst auf $n+1$ setzen und dann bei jeder Addition um 1 verringern. Die Addition erfolgt solange, wie die Registernummer (im ersten Register), auf die wir zugreifen wollen, mindestens 3 ist. Für diesen ganzen Prozess konstruieren wir eine Schleife.
Die beiden Sonderfälle, $n = 0$ und $n = 1$ (bei denen eigentlich keine Addition erfolgt) lassen sich einfach dadurch realisieren, dass der Inhalt des zweiten Registers (0 bei $n = 0$ und x_1 bei $n = 1$) direkt in das Ergebnisregister 1 umgespeichert wird. Diese beiden Fällen werden wir außerhalb der Schleife vorab erledigen.
Abschließend speichern wir das Ergebnis, das in jedem Fall im zweiten Register steht, in das erste Register um.
Formal ergibt dies das Programm aus Abb. 3.6.

1	`LOAD` 1	
2	`CSUB` 1	Befehle 1–3 testen, ob Sonderfall vorliegt
3	`IF` $c_0 = 0$ `GOTO` 15	
4	`LOAD` 1	
5	`CADD` 1	Befehle 4–5 setzen Registernummer für Addition auf $n+1$
6	`STORE` 1	
7	`ILOAD` 1	
8	`ADD` 2	Befehle 7–9 addieren $c_{i+1} = x_i$, $n \geq i \geq 2$, zu c_2
9	`STORE` 2	
10	`LOAD` 1	
11	`CSUB` 3	Befehle 10–12 testen, ob $c_3 = x_2$ schon addiert wurde
12	`IF` $c_0 = 0$ `GOTO` 15	
13	`CADD` 2	Verringern der Registernummer $i+1$ um 1
14	`GOTO` 6	
15	`LOAD` 2	
16	`STORE` 1	Befehle 15 und 16 speichern das Ergebnis in das erste Register
17	`END`	

Abbildung 3.6: Programm einer Registermaschine zur Berechnung von (3.1)

Wir wollen nun zeigen, dass Registermaschinen die Funktionen, die von **LOOP/WHILE**-Programmen induziert werden, berechnen können.

Satz 3.13 *Zu jedem* **LOOP/WHILE**-*Programm* Π *gibt es eine Registermaschine* M *derart, dass* $f_M = \Phi_{\Pi,1}$ *gilt.*

Beweis. Wir beweisen den Satz mittels Induktion über die Tiefe der **LOOP/WHILE**-Programme.
Induktionsanfang $k = 1$. Dann ist die gegebene Funktion eine der Wertzuweisungen.
Ist $x_i := 0$ die Anweisung, so liefert die Registermaschine mit dem Programm

```
1  CLOAD 0
2  STORE i
3  END
```

bereits das gewünschte Verhalten.
Ist $x_i := S(x_j)$, so leistet die Registermaschine mit dem Programm

```
1  LOAD j
2  CADD 1
3  STORE i
4  END
```

die gewünschte Simulation.
Für $x_i := P(x_j)$ und $x_i := x_j$ geben wir analoge Konstruktionen.
Induktionsschritt von $< k$ *auf* k. Wir haben zwei Fälle zu unterscheiden, nämlich ob als letzte Operation beim Aufbau der Programme eine Hintereinanderausführung oder eine **WHILE**-Schleife angewendet wurde (auf die Betrachtung der **LOOP**-Schleife können wir wegen der Bemerkung am Ende des Abschnitts 1.1.1 verzichten).
Hintereinanderausführung. Es sei $\Pi = \Pi_1; \Pi_2$, und für $i \in \{1, 2\}$ sei M_i die nach Induktionsvoraussetzung existierende Registermaschine mit $f_{M_i} = \Phi_{\Pi_i,1}$. Ferner habe M_i das Programm P_i, das aus r_i Befehlen bestehen möge. Weiterhin bezeichnen wir mit $p_{j,i}$ den j-ten Befehl von P_i. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass jedes der Programme P_i nur einen `END`-Befehl enthält, der am Ende des Programms steht. Wir modifizieren nun die Befehle des Programms P_2 dahingehend, dass wir alle Befehlsnummer j in einem Sprungbefehl oder einem bedingten Befehl durch $j + r_1 - 1$ ersetzen. Dadurch entstehe q_j aus $p_{j,2}$ für $1 \leq j \leq r_2$. Dann berechnet die Registermaschine mit dem Programm

$$\begin{array}{rl} 1 & p_{1,1} \\ 2 & p_{2,1} \\ \ldots & \ldots \\ r_1 - 1 & p_{r_1-1,1} \\ r_1 & q_1 \\ r_1 + 2 & q_2 \\ \ldots & \ldots \\ r_1 + r_2 - 1 & q_{r_2} \end{array}$$

die Funktion $\Phi_{\Pi,1}$.

WHILE-*Schleife* Sei $\Pi' = \textbf{WHILE}x_i \neq 0\textbf{BEGIN}\Pi\textbf{END}$, und seien $p_1, p_2, \ldots p_r$ die Befehle einer Registermaschine M mit $f_M = \Phi_{\Pi,1}$, wobei wiederum $p_r = \texttt{END}$ gelte. Für $1 \leq i \leq m$ sei q_i der Befehl, der aus p_i entsteht, indem jede in ihm vorkommende Befehlsnummern um 2 erhöht werden. Dann berechnet das Programm

1	$\texttt{LOAD}i$
2	$\texttt{IF}c_0 = 0\texttt{GOTO}r + 3$
3	q_1
4	q_2
...	...
$r+1$	q_{r-1}
$r+2$	$\texttt{GOTO}1$
$r+3$	$\texttt{END}$

die von Π' indizierte Funktion. □

Um zu zeigen, dass auch umgekehrt jede von einer Registermaschine induzierte Funktion **LOOP/WHILE**-berechenbar ist, geben wir nun eine Simulation von Registermaschinen durch Mehrband-TURING-Maschinen an. Dies ist hinreichend, weil wir aus dem ersten Teil der Vorlesung wissen, dass bis auf eine Kodierung von TURING-Maschinen induzierte Funktionen **LOOP/WHILE**-berechenbar sind.
Da die von TURING-Maschinen induzierten Funktionen Wörter in Wörter abbilden, während die von Registermaschinen berechneten Funktionen Tupel natürlicher Zahlen auf natürliche Zahlen abbilden, müssen wir natürliche Zahlen durch Wörter kodieren. Dies kann z.B. durch die Binär- oder Dezimaldarstellung der Zahlen erfolgen.
Für eine natürliche Zahl m bezeichne $dec(m)$ die Dezimaldarstellung von n.
Der folgende Satz besagt nun, dass es zu jeder Registermaschine M eine mehrbändige TURING-Maschine gibt, die im wesentlichen dasselbe wie M leistet, d.h. auf eine Eingabe der Dezimaldarstellungen von $m_1, m_2, \ldots m_n$ liefert die TURING-Maschine die Dezimaldarstellung von $f_M(m_1, m_2, \ldots, m_n)$, also die Dezimaldarstellung des Ergebnisses der Berechnung von M.

Satz 3.14 *Es seien M eine Registermaschine M mit $f_M : \mathbf{N}^n \to \mathbf{N}$. Dann gibt es eine 3-Band-*TURING*-Maschine M', deren Eingabealphabet außer den Ziffern 0,1,2,... 9 noch das Trennsymbol # und das Fehlersymbol F enthält und deren induzierte Funktion*

$$f_{M'}(w) = \begin{cases} dec(f_M(m_1, m_2, m_3, \ldots, m_n)) & w = dec(m_1)\#dec(m_2)\#dec(m_3)\ldots\#dec(m_n) \\ F & sonst \end{cases}$$

gilt (auf einer Eingabe, die einem Zahlentupel entspricht, verhält sich M' wie M und gibt bei allen anderen Eingaben eine Fehlermeldung).

Beweis. Wir geben hier keinen vollständigen formalen Beweis; wir geben nur die Idee des Beweises wider; der formale Beweis lässt sich unter Verwendung der Konstruktionen und Ideen aus den Beweisen der Lemmata 1.21 und 1.22 erbringen.
Wir konstruieren eine 3-Band-TURING-Maschine M', die schrittweise die Arbeit von M simuliert:

Auf dem ersten Arbeitsband speichern wir im Wesentlichen die Konfiguration der Registermaschine. Da diese ein unendliches Tupel ist, kann dies nicht direkt geschehen. Wir geben dort im wesentlichen die Folge der Nummern und Inhalte der Register an, die im Laufe der schon simulierten Schritte belegt worden sind. Formal steht auf dem ersten Band das Wort

$$\#\#0\#dec(c_0)\#\#dec(k_1)\#dec(c_{k_1})\#\#dec(k_2)\#dec(c_{k_2})\ldots\#\#dec(k_s)\#dec(c_{k_s})\#\#$$

(d.h. durch ## werden die verschiedenen Register voneinander getrennt; es wird stets die Nummer 0 bzw. k_i des Registers, $1 \leq i \leq s$, und der Inhalt c_o bzw. c_{k_i} angegeben die durch ein # getrennt sind; in allen anderen Registern steht eine 0; da stets nur endlich viele Register einer Registermaschine mit von Null verschiedenen Werten belegt werden, enthält das erste Band stets nur endlich viele Symbole).
Die gegebene Registermaschine M habe ein Programm mit r Befehlen. Für $1 \leq i \leq r$ konstruieren wir eine TURING-Maschine M_i, die eine Änderung des ersten Bandes entsprechend dem i-ten Befehl vornimmt. M_i arbeitet nur auf den drei Arbeitsbändern. Nach der eigentlichen Simulation des Befehls der Registermaschine werden das zweite und dritte Arbeitsband stets geleert und auf dem ersten Arbeitsband der Kopf zum Anfang bewegt (d.h. er steht über dem ersten #). $z_{i,0}$ sei der Anfangszustand und q_i der Stoppzustand von M_i. Um festzuhalten, welcher Befehl gerade simuliert wird, haben die Zustände von M' die Form (i, z), wobei $1 \leq i \leq r$ gilt und z ein Zustand von M_i ist. Für eine Anfangsphase werden noch weitere Zustände benötigt.
M' arbeitet nun wie folgt: Zuerst testet M', ob die Eingabe die Form $w_1\#w_2\#\ldots\#w_n$, wobei für $1 \leq i \leq n$ entweder $w_i = 0$ oder $w_i = x_iy_i$ mit $x_i \in \{1, 2, \ldots, 9\}$ und $y_i \in \{0, 1, \ldots, 9\}^*$ gilt, d.h. ob die Eingabe die Kodierung eines n-Tupels natürlicher Zahlen ist.
Ist dies nicht der Fall, so schreibt M' das Fehlersymbol F auf das Ausgabeband und stoppt.
Im anderen Fall schreibt M' auf das erste Arbeitsband die entsprechend modifizierte Eingabe

$$\#\#0\#0\#\#1\#dec(m_1)\#\#2\#dec(m_2)\ldots\#\#dec(n)\#dec(m_n)\#\#,$$

geht in den Zustand $(1, z_{1,0})$ über, und M_1 beginnt mit der Simulation des ersten Befehls. M_i, $1 \leq i \leq r$, beendet seine Simulation im Zustand (i, q_i), da während der Simulation nur die zu M_i gehörende zweite Komponente geändert wird. M' geht nun in den Zustand $(j, z_{j,0})$, wobei j der nach dem i-ten Befehl abzuarbeitende Befehl ist.

Wir geben nun die Arbeitsweise einiger TURING-Maschinen zu Befehlen der Registermaschine an, wobei wir nur die Änderung des Inhalts des ersten Arbeitsbandes angeben (es folgen dann noch die Löschungen auf den anderen Arbeitsbändern und die Kopfbewegung zum Anfang des ersten Arbeitsbandes).
a) Sei `LOAD t` der i-te Befehl. Dann schreibt M_i zuerst $dec(t)$ auf das zweite Arbeitsband und testet dann, ob im t-ten Register schon etwas gespeichert ist. Dazu läuft sie über das Wort auf dem ersten Arbeitsband und vergleicht stets ob nach zwei aufeinanderfolgenden # das Wort $dec(t)$ folgt. Hinter $dec(t)$ steht dann $\#dec(c_t)\#$ auf dem ersten Band. M_i leert das zweite Band und kopiert $dec(c_t)$ auf das zweite Band. Dann ersetzt M_i den Inhalt

des Akkumulators (zwischen dem dritten und vierten # auf dem Band) durch den Inhalt $dec(c_t)$ des t-ten Registers. Findet M_i keinen Eintrag im t-ten Register (ist bei Erreichen eines $*$ gegeben), so wird eine 0 in den Akkumulator geschrieben.
b) Im Fall der indirekten Adressierung `ILOAD` t schreiben wir zuerst den Inhalt des t-ten Registers in Dezimaldarstellung auf das zweite Band (dieser wird analog zu a) gesucht) und mit diesem Wert anstelle von $dec(t)$ verfahren wir wie bei a).
c) Ist der i-te Befehl `CLOAD` t, so schreiben wir gleich $dec(t)$ auf das zweite Band und kopieren dies in den Akkumulator (zwischen dem dritten und dem vierten Vorkommen von #).
d) Ist `STORE` t der i-te Befehl, so kopiert M_i den Inhalt $dec(c_0)$ des Akkumulators auf das dritte Band, schreibt $dec(t)$ auf das zweite Band, sucht die Stelle, wo der Inhalt des t-Registers steht, (dies steht hinter $\#\#dec(t)\#$) und ersetzt diesen durch den Inhalt des dritten Bandes. Wird $\#\#dec(t)\#$ nicht gefunden (d.h. es erfolgte noch kein Eintrag in dieses Register, so wird $dec(t)\#dec(c_0)\#\#$ an das Wort auf dem ersten Band angefügt.
e) Ist der i-te Befehl `ADD` t, so wird zuerst der Inhalt des t-ten Registers gesucht und auf das zweite Band geschrieben. Durch ein # getrennt schreibt M_i den Inhalt des Akkumulators dahinter und addiert beide Zahlen, wobei das Ergebnis auf das dritte Band geschrieben wird. Dieses Ergebnis schreiben wir in den Akkumulator.
f) Beim Befehl `GOTO` t ändern wir direkt den Zustand $(i, z_{i,0})$ zu $(t, z_{t,0})$.
g) Beim Befehl `IF` $c_o \neq 0$ `GOTO` t testet M_i, ob zwischen dem dritten und vierten # genau eine 0 steht. Ist dies der Fall geht M' in den Zustand $(t, z_{t,0})$, andererfalls in $(i+1, z_{i+1,0})$.
h) Beim Befehl `END` wird der Inhalt des ersten Registers (zwischen dem sechsten und siebenten #) auf das Ausgabeband geschrieben und gestoppt.
Die Konstruktionen für die anderen Fälle sind analog.

Aus diesen Erklärungen folgt sofort, dass M' bei Eingabe von der Kodierung eines Zahlentupels schrittweise die Befehle der Registermaschine simuliert und am Ende die Kodierung des Inhalts des ersten Registers, in dem das Ergebnis der Berechnung der Registermaschine steht, ausgibt. □

Fassen wir unsere Ergebnisse über Beziehungen zwischen Berechenbarkeitsbegriffen zusammen, so ergibt sich folgender Satz.

Satz 3.15 *Für eine Funktion f sind die folgenden Aussagen gleichwertig:*

- *f ist durch ein* **LOOP/WHILE**-*Programm berechenbar.*
- *f ist partiell-rekursiv.*
- *f ist durch eine Registermaschine berechnenbar.*
- *f ist bis auf Konvertierung der Zahlendarstellung durch eine* Turing-*Maschine berechenbar.*
- *f ist bis auf Konvertierung der Zahlendarstellung durch eine k-Band-*Turing-*Maschine, $k \geq 1$, berechenbar.* □

3.3 Komplexitätstheoretische Beziehungen

Im vorhergehenden Abschnitt haben wir nur gezeigt, dass eine gegenseitige Simulation von TURING-Maschinen und Registermaschinen möglich ist. In diesem Abschnitt wollen eine genauere Analyse durchführen, indem wir noch zusätzlich die Komplexität der Berechnungen betrachten. Daher definieren wir zuerst die Komplexität bei Registermaschinen. Hierbei nehmen wir eine Beschränkung der (indirekten) arithmetischen Befehle auf Addition und Subtraktion vor, um etwas realistischer zu sein, da die Komplexität der Berechnung einer Multiplikation bzw. Division erheblich höher ist als die von Addition und Subtraktion. Werden Multiplikation und Division dagegen auf Addition und Subtraktion zurückgeführt, entstehen realistischere Maße.

Definition 3.16 *Es sei M eine Registermaschine. Die (uniforme) Komplexität von M auf $(x_1, x_2, \ldots, x_n)$, bezeichnet durch $t_M(x_1, x_2, \ldots, x_n)$ ist als Anzahl der Schritte definiert, die M bis zum Erreichen des* `END`*-Befehls auf der Eingabe $(x_1, x_2, \ldots, x_n)$ ausführt.*

Definition 3.17 *Es sei M eine Registermaschine. Wir sagen, dass M $t(n)$-zeitbeschränkt ist, wenn*

$$t_M(x_1, x_2, \ldots, x_n) \leq t(n)$$

für jede Eingabe $(x_1, x_2, \ldots, x_n)$ gilt.

Satz 3.18 *i) Jede $t(n)$-zeitbeschränkte Registermaschine kann durch eine $O((t(n))^3)$-zeitbeschränkten 3-Band-*TURING*-Maschine simuliert werden.*
ii) Jede $t(n)$-zeitbeschränkte Registermaschine kann durch eine $O((t(n))^6)$-zeitbeschränkten TURING*-Maschine simuliert werden.*

Beweis. i) Wir analysieren die im Beweis von Satz 3.14 gegebene TURING-Maschine. Da in jedem Schritt der Registermaschine ein Registerinhalt hinsichtlich der Länge höchstens um 1 erhöht wird (z.B. bei einer Addition einer Zahl mit sich selbst)[1]. Daher ist die Länge der Registerinhalte durch $O(t(n))$ beschränkt. Außerdem können in $t(n)$ höchstens $O(t(n))$ Register gefüllt werden. Folglich ist jeder Teil $dec(k)\#dec(c_k)$ in der Länge durch $O(t(n))$ beschränkt. Damit steht auf dem ersten Band höchstens ein Wort der Länge $O(t(n)^2)$. Folglich erfordert die Arbeit einer jeder Maschine, die einen Befehl simuliert, höchstens $O(t(N)^2)$ Schritte (Lesen der Konfiguration auf Band 1, umspeichern, vergleichen etc. von Wörtern der Länge $\leq O(t(n))$). Da insgesamt $t(n)$ mal Befehle simuliert werden, benötigt die TURING-Maschine insgesamt höchstens $O(t(n)^3)$ Schritte.
ii) Die Aussage ergibt sich aus dem ersten Teil unter Verwendung von Satz **??** □

Wir wollen nun zeigen, dass auch die Simulation in der umgekehrten Richtung polynomiale Berechenbarkeit erhalten bleibt. Wir machen dabei aber darauf aufmerksam, dass wir damit sogar eine direkte Simulation von TURING-Maschinen durch Registermaschinen erhalten (im vorhergehenden Abschnitt haben wir einen Umweg über **LOOP**/**WHILE**-Berechenbarkeit gewählt.
Wir benötigen das folgende Lemma.

[1] Man beachte, dass diese Aussage bei Verwendung der Multiplikation nicht mehr gilt. Die Multiplikation kann zur Verdopplung der Länge führen.

Lemma 3.19 *Es sei $k \geq 2$. Jede $t(n)$-zeitbeschränkte k-Band-*Turing*-Maschine kann durch eine $O(t(n))$-zeitbeschränkte k-Band-*Turing*-Maschine simuliert werden, die keine Zelle links der Eingabe betritt/verändert.*

Beweis. Wir geben keinen vollständigen formalen Beweis sondern nur die Idee der Konstruktion. Es sei $M = (k, X, Z, z_0, Q, \delta)$ eine k-Band-Turing-Maschine, bei der wir die Zellen eines jeden Bandes entsprechend den natürlichen Zahlen durchnummerieren. Wir konstruieren eine k-Band-Turing-Maschine, bei der wir ebenfalls eine Nummerierung der Zellen eines jeden Bandes vornehmen. In jede Zelle des Bandes k mit der Nummer i, wobei $i > 0$ ist, speichern wir jetzt aber ein Paar von Buchstaben aus $X \cup \{*\}$ ab. Das Paar (a, b) in der Zelle mit Nummer i des Bandes k bei M' beschreibt die Situation, dass bei M in der Zelle des Bandes k mit der Nummer i das Symbol a und in der Zelle mit der Nummer $-i$ das Element b steht. Daher muss sich M' nie auf Zellen mit nicht positiven Nummern begeben. Die Maschine M' muss sich aber im Zustand merken, ob es sich über dem negativen oder positiven Bereich von M befindet. Dies kann dadurch erreicht werden, dass die Zustände die Form $(u_1, u_2, \ldots u_k, q)$ mit $u_i \in \{+, -\}$, $1 \leq i \leq k$, und $Q \in Z$ haben. Dabei gibt u_i, $1 \leq i \leq k$, an, ob sich der Kopf des Bandes i über dem positiven oder negativen Teil des Bandes befindet. Da die Zelle mit der Nummer 0 nur ein Element aus $X \cup \{*\}$ (und kein Paar) enthält, ist es einfach den Wechsel von $+$ zu $-$ und umgekehrt zu vollziehen.
Offensichtlich braucht M' genau die gleiche Anzahl von Schritten wie M. □

Satz 3.20 *Jede $t(n)$-zeitbeschränkte k-Band-*Turing*-Maschine mit $k \geq$ kann durch eine $O(t(n))$-zeitbeschränkte Registermaschine simuliert werden.*

Beweis. Es sei $M = (k, X, Z, z_0, Q, \delta)$ eine k-Band-Turing-Maschine. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass $X = \{1, 2, \ldots, n\}$ und $Z = \{0, 1, 2, \ldots, m\}$ gelten, wobei $z_0 = 0$ gilt. Außerdem wollen wir anstelle von $*$ die Zahl 0 verwenden. Wir nehmen nach Lemma 3.19 an, dass sich die Köpfe nur über Zellen mit den Nummern aus $\mathbf{N}_0$ befinden.
Wir konstruieren nun eine Registermaschine, die wie folgt aufgebaut ist und arbeitet: Im Register i, $1 \leq i \leq k$, steht die Position, in der sich der Kopf des Bandes i befindet, im Register $k + 1$ steht die Position, in der sich der Kopf des Eingabebandes befindet, im Register $k+2$ steht der Zustand der Turing-Maschine, im Register $C+(k+1)p$ steht der Inhalt der Zelle mit Nummer p vom Eingabeband, im Register $C+kp+j$ steht der Inhalt der Zelle mit Nummer p vom Band j, wobei C eine hinreichend große Konstante ist, so dass die notwendigen Zwischenrechnungen in den Registern $k+3$ – $C-1$ durchgeführt werden können. Wir illustrieren dies durch ein Beispiel mit $k = 2$. Wenn sich M im Zustand 5 befindet, auf dem Eingabeband 1232, auf dem ersten Arbeitsband 221 und auf dem zweiten Arbeitsband 1133 stehen (der erste Buchstabe stehe stets in der Zelle mit der Nummer 0), und der Kopf des Eingabebandes, ersten und zweiten Arbeitsbandes auf den Positionen 2, 3, und 1 stehen, so ergibt sich die folgende Konfiguration der Registermaschine

$$(b, c_0, 3, 1, 2, 5, c_5, c_6, \ldots, c_{C-1}, 1, 2, 1, 2, 2, 1, 3, 2, 3, 2, 0, 3, 0, 0, 0, \ldots).$$

Das Programm der Registermaschine besteht aus „kleinen“ Programmen, die nacheinander abgearbeitet werden. Ein dieser Programme überträgt die Eingabe der Turing-Maschine in die entsprechenden Register. Die restlichen Programme simulieren einen Arbeitsschritt der **Turing-Maschine**, d.h. sie berechnen jeweils die Funktionen δ, wobei sie die notwendigen Eingaben aus den entsprechenden Registern holen, genauer den Zustand aus Register $k+2$, das Symbol des Eingabebandes aus Register $C + c_{k+1}(k+1)$, das Symbol des j-ten Arbeitsbandes aus Register $C + c_j(k+1) + j$, $1 \leq j \leq k$, und die ermittelten Werte (einschließlich des neuen Zustandes und der neuen Kopfpositionen) wieder entsprechend ablegen. Die Durchführung einer derartigen Berechnung von δ geschieht in konstanter Zeit (man beachte, dass k fest gewählt ist). Damit erfordert die Simulation nur $O(t(n)))$ Schritte der Registermaschine. □

Entsprechend den vorstehenden Sätzen ist es hinsichtlich der Zugehörigkeit zur Klasse **P** nicht wichtig, ob wir mitTuring-Maschinen oder Registermaschinen arbeiten.

Kapitel 4

Ergänzung II: Abschluss- und Entscheidbarkeitseigenschaften formaler Sprachen

4.1 Abschlusseigenschaften formaler Sprachen

Im ersten Teil der Vorlesung haben wir das Verhalten von regulären Sprachen hinsichtlich der Operationen Vereinigung, Durchschnitt, Komplement, Produkt und Kleene-Abschluss untersucht. Daraus haben wir eine Charakterisierung der regulären sprachen (durch reguläre Ausdrücke gewonnen.
In diesem Abschnitt wollen wir untersuchen, wie sich die Sprachen der anderen Familien der Chomsky-Hierarchie gegenüber diesen Operationen verhalten. Weiterhin werden wir weitere Operationen einführen, mittels derer uns eine weitere Charakterisierung der regulären Sprachen gelingen wird.
Wir geben zuerst die hierfür grundlegende Definition.

Definition 4.1 *Es seien $\mathcal{L}$ eine Menge von Sprachen und τ eine n-stellige Operation, die über Sprachen definiert ist. Dann heißt $\mathcal{L}$ abgeschlossen unter τ, wenn für beliebige Sprachen $L_1, L_2, \ldots, L_n \in \mathcal{L}$ auch*

$$\tau(L_1, L_2, \ldots, L_n) \in \mathcal{L}$$

gilt.

Wir untersuchen nun, ob die Sprachmengen der Chomsky-Hierarchie unter den üblichen mengentheoretischen Operationen Vereinigung, Durchschnitt und Komplement und den algebraischen Operationen Produkt und Kleene-Abschluss abgeschlossen sind. Dabei werden wir zwar der Vollständigkeit halber auch die Ergebnisse für die Menge der regulären Mengen angeben (und damit jeweils Satz **??** partiell wiederholen), aber den Beweis dafür nicht erneut geben.

Lemma 4.2 *$\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen unter der (üblichen) Vereinigung von Sprachen.*

Beweis. Wir zeigen die Aussage zuerst für $\mathcal{L}(CF)$. Seien L_1 und L_2 zwei kontextfreie Sprachen über dem Alphabet T. Wir haben zu zeigen, dass auch $L_1 \cup L_2$ eine kontextfreie Sprache (über T) ist.
Dazu seien

$$G_1 = (N_1, T_1, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T_2, P_2, S_2)$$

zwei kontextfreie Grammatiken mit

$$L(G_1) = L_1 \quad \text{und} \quad L(G_2) = L_2.$$

Offenbar können wir ohne Beschränkung der Allgemeinheit annehmen, dass

$$T_1 = T_2 = T \quad \text{und} \quad N_1 \cap N_2 = \emptyset$$

gelten (notfalls sind die Nichtterminale umzubenennen). Ferner sei S ein Symbol, das nicht in $N_1 \cup N_2 \cup T$ liegt. Wir betrachten nun die kontextfreie Grammatik

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S).$$

Offenbar hat jede Ableitung in G die Form

$$(*) \qquad S \Longrightarrow S_i \Longrightarrow^* w,$$

wobei $i = 1$ oder $i = 2$ gilt und $S_i \Longrightarrow^* w$ eine Ableitung in G_i ist (da wegen $N_1 \cap N_2 = \emptyset$ keine Symbole aus N_j, $j \neq i$ entstehen können und damit keine Regeln aus P_j anwendbar sind). Folglich gilt $w \in L(G_i)$. Hieraus folgt sofort

$$L(G) \subseteq L(G_1) \cup L(G_2) = L_1 \cup L_2.$$

Man sieht aber auch aus $(*)$ sofort, dass jedes Element aus $L(G_i)$, $i \in \{1, 2\}$, erzeugt werden kann, womit auch die umgekehrte Inklusion

$$L(G) \supseteq L(G_1) \cup L(G_2) = L_1 \cup L_2$$

gezeigt ist.
Da die bei der Konstruktion von G hinzugenommenen Regeln bei kontextabhängigen oder allgemeinen Regelgrammatiken zugelassen sind, kann mit dem gleichen Beweis auch gezeigt werden, dass $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ gegenüber Vereinigung abgeschlossen sind. □

Lemma 4.3 *$\mathcal{L}(REG)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen unter Durchschnitt, und $\mathcal{L}(CF)$ ist gegenüber Durchschnitt nicht abgeschlossen.*

Beweis. i) $\mathcal{L}(RE)$. Es seien $L_1 \in \mathcal{L}(RE)$ und $L_2 \in \mathcal{L}(RE)$ gegeben. Nach Satz 2.43 und 2.32 gibt es TURING-Maschinen

$$M_1 = (X, Z_1, z_{01}, Q_1, \delta_1, Q_1) \quad \text{und} \quad M_2 = (X, Z_2, z_{02}, Q_2, \delta_2, Q_2)$$

mit

$$T(M_1) = L_1 \quad \text{und} \quad T(M_2) = L_2,$$

wobei wir wieder annehmen können, dass $Z_1 \cap Z_2 = \emptyset$ gilt. Wir betrachten nun die TURING-Maschine M, die wie folgt arbeitet (die formale Beschreibung von M bleibt dem Leser überlassen). Zuerst ersetzt sie jeden Buchstaben x auf dem Band durch das Paar (x, x). Dann arbeitet sie wie M_1, wobei sie stets nur den Inhalt der ersten Komponente entsprechend δ_1 verändert und sich somit in der zweiten Komponente das ursprünglich auf dem Band befindliche Wort speichert (werden Leerzeichen gelesen, so sind wie ein Paar $(*, *)$ zu behandeln). Wird ein Zustand aus Q_1 erreicht, so ersetzt die Maschine alle Paare auf dem Band durch ihre zweite Komponente, womit das Ausgangswort wieder auf dem Band steht. Dann bewegt sie den Kopf zum ersten Buchstaben, geht in den Zustand z_{02} und beginnt nun wie M_2 zu arbeiten. Die Maschine stoppt, wenn sie einen Zustand aus Q_2 erreicht.
Entsprechend dieser Arbeitsweise wird mittels der ersten Komponente getestet, ob das Wort von M_1 akzeptiert wird; ist dies der Fall wird auch noch getestet, ob es in $T(M_2)$ liegt. Folglich erreicht M genau dann einen Stoppzustand, wenn das Wort sowohl in $T(M_1)$ als auch $T(M_2)$ liegt. Wenn wir alle Stoppzustände zur Akzeptanz verwenden, erhalten wir

$$T(M) = T(M_1) \cap T(M_2) = L_1 \cap L_2,$$

womit die Behauptung aus Satz 2.43 folgt.

ii) $\mathcal{L}(CS)$. Der Beweis kann genauso wie unter i) geführt werden, wobei wir von linear beschränkten Automaten ausgehen und Satz 2.45 benutzen.

iii) $\mathcal{L}(CF)$. Um diese Aussage zu beweisen, reicht es, zwei kontextfreie Sprachen L_1 und L_2 anzugeben, deren Durchschnitt keine kontextfreie Sprache ist. Dazu betrachten wir

$$L_1 = \{a^n b^n c^m : n \geq 1, m \geq 1\} \quad \text{und} \quad L_2 = \{a^m b^n c^n : n \geq 1, m \geq 1\}.$$

Es ist leicht zu zeigen, dass diese beiden Sprachen kontextfrei sind (wir überlassen die Konstruktion zugehöriger Grammatiken dem Leser). Dann gilt

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geq 1\},$$

und dies ist nach Lemma 2.28 keine kontextfreie Sprache. □

Lemma 4.4 *i) $\mathcal{L}(REG)$ und $\mathcal{L}(CS)$ sind abgeschlossen bezüglich Komplement.*
ii) $\mathcal{L}(CF)$ und $\mathcal{L}(RE)$ sind nicht abgeschlossen unter Komplement.

Beweis. Wir beweisen die Aussage nur für $\mathcal{L}(RE)$, $\mathcal{L}(REG)$ und $\mathcal{L}(CF)$, da der Beweis für $\mathcal{L}(CS)$ mit den bisher zur Verfügung stehenden Mitteln zu aufwändig ist (ein Beweis ist z.B. in [24] zu finden).

$\mathcal{L}(RE)$. Wäre $\mathcal{L}(RE)$ unter Komplementbildung abgeschlossen, so wäre wegen Satz 2.35 jede rekursiv-aufzählbare Sprache rekursiv. Dies steht aber im Widerspruch zu Satz 2.37.

$\mathcal{L}(CF)$. Wir nehmen an, dass $\mathcal{L}(CF)$ gegenüber Komplement abgeschlossen ist. Es seien nun zwei beliebige kontextfreie Sprachen L_1 und L_2 gegeben. Wir setzen

$$X = alph(L_1) \cup alph(L_2), \quad X_1 = X \setminus alph(L_1), \quad X_2 = X \setminus alph(L_2).$$

Ferner seien R_1 und R_2 die Mengen aller Wörter über X, die mindestens einen Buchstaben aus X_1 bzw. X_2 enthalten. Wir zeigen nun, dass diese beiden Sprachen regulär und damit auch kontextfrei sind. Hierfür reicht es festzustellen, dass für $i \in \{1,2\}$ die reguläre Grammatik

$$G_i = (\{S,A\}, X, \bigcup_{a \in alph(L_i)} \{S \to aS\} \cup \bigcup_{b \in X_i} \{S \to bA, S \to b\} \cup \bigcup_{x \in X} \{A \to xA, A \to x\}, S)$$

die Sprache R_i erzeugt. Dies folgt daher, dass ein Übergang zum Nichtterminal A oder zum direkten Terminieren aus S nur möglich sind, wenn mindestens ein Element aus X_i erzeugt wurde.
Aufgrund einfacher mengentheoretischer Fakten gilt

$$X^* \setminus L_i = ((alph(L_i))^* \setminus L_i) \cup R_i = \overline{L_i} \cup R_i$$

für $i \in \{1,2\}$. Nach unserer Annahme und Lemma 4.2 sind damit $X^* \setminus L_1$ und $X^* \setminus L_2$ kontextfreie Sprachen. Damit ist auch

$$R = (X^* \setminus L_1) \cup (X^* \setminus L_2)$$

eine kontextfreie Sprache. Wegen unserer Annahme und

$$L_1 \cap L_2 = X^* \setminus ((X^* \setminus L_1) \cup (X^* \setminus L_2)) = (alph(R))^* \setminus R$$

ist damit $L_1 \cap L_2$ als kontextfrei nachgewiesen. Dies steht im Widerspruch zu Lemma 4.3. Folglich muss die gemachte Annahme falsch sein, womit die Aussage des Lemmas gezeigt ist. □

Lemma 4.5 *$\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen unter Produkt.*

Beweis. $\mathcal{L}(CF)$. Erneut gehen wir von zwei kontextfreien Grammatiken

$$G_1 = (N_1, T, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T, P_2, S_2)$$

mit $N_1 \cap N_2 = \emptyset$ aus und zeigen, dass die Grammatik

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

die Sprache

$$L(G) = L(G_1) \cdot L(G_2)$$

erzeugt. Hierzu reicht zu bemerken, dass bis auf die Reihenfolge der Anwendung der Regeln jede Ableitung in G die Form

$$S \Longrightarrow S_1 S_2 \Longrightarrow^* w_1 S_2 \Longrightarrow^* w_1 w_2$$

hat, wobei für $i \in \{1,2\}$ die Ableitung $S_i \Longrightarrow^* w_i$ auch eine Ableitung in G_i ist, d.h. nur mittels Regeln aus P_i entsteht.
$\mathcal{L}(CS)$ und $\mathcal{L}(RE)$. Wir können den Beweis wie bei $\mathcal{L}(CF)$ führen, wenn wir voraussetzen, dass die Grammatiken in der Normalform aus Satz 2.16 sind (diese Voraussetzung sichert, dass sich die Ableitungen in G_1 und G_2 nicht über den Kontext beeinflussen können. Außerdem haben wir das Leerwort, falls es in $L(G_1)$ und/oder $L(G_2)$ liegt, gesondert zu behandeln (dies erfordert nur die Vereinigung einiger Sprachen). □

Lemma 4.6 *$\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen gegenüber der Bildung des (positiven) Kleene-Abschlusses.*

Beweis. Wir beweisen die Aussage zuerst für den positiven KLEENE-Abschluss.
$\mathcal{L}(CF)$. Es sei die kontextfreie Sprache L von der kontextfreien Grammatik $G = (N, T, P, S)$ erzeugt. Wir setzen dann

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to SS', S' \to S\}, S')$$

(wobei S' wieder ein zusätzliches Symbol ist). Bis auf die Reihenfolge der Anwendung der Regeln hat jede Ableitung in G' die Form

$$\begin{aligned} S' &\Longrightarrow SS' \Longrightarrow^* w_1S' \Longrightarrow w_1SS' \Longrightarrow^* w_1w_2S' \Longrightarrow w_1w_2SS' \Longrightarrow ... \\ &\Longrightarrow w_1w_2 \dots w_{n-1}S' \Longrightarrow w_1w_2 \dots w_{n-1}S \Longrightarrow^* w_1w_2 \dots w_{n-1}w_n, \end{aligned}$$

wobei die Ableitungen $S \Longrightarrow^* w_i$ für $1 \leq i \leq n$ stets nur Regeln aus P benutzen. Daher gilt $w_i \in L(G) = L$ für $1 \leq i \leq n$ und $w_1w_2 \dots w_n \in L^n$ ist bewiesen.
Umgekehrt ist klar, dass auf diese Weise jedes aus L^n, $n \geq 1$, erzeugt werden kann. Folglich gilt

$$L(G') = \bigcup_{n \geq 1} L^n = L^+,$$

womit L^+ als kontextfrei nachgewiesen ist.
$\mathcal{L}(CS)$ und $\mathcal{L}(RE)$. Wir gehen erneut wie bei $\mathcal{L}(CF)$ vor, benutzen die Normalform nach Satz 2.16 und ändern die zu P hinzugefügten Regeln, um zu sichern, dass sich die Kontexte nicht gegenseitig beeinflussen, d.h. dass die Ableitung des Wortes w_i erst beginnt, wenn die von w_{i-1} hierdurch nicht mehr beeinflusst werden kann. Dies geschieht durch Hinzufügen der Regeln

$$\begin{aligned} &S' \to S,\ S' \to SS'', \\ &xS'' \to xSS'',\ xS'' \to xS \quad \text{für } x \in T. \end{aligned}$$

Die Details des Beweises überlassen wir dem Leser.

Wir geben nun die Modifikationen für den KLEENE-$*$. Gilt $\lambda \in L$, so können wir wegen der dann gegebenen Gültigkeit von $L^* = L^+$ wie oben vorgehen. Ist $\lambda \notin L$, so haben wir nur noch das Leerwort zusätzlich zu L^+ zu erzeugen. Ist $G = (N, T, P, S)$ eine Grammatik mit $L(G) = L^+$, so ist dies einfach dadurch zu realisieren, dass wir zu N ein weiteres Nichtterminal S' und zu P die Regeln $S' \to \lambda$ und $S' \to S$ hinzufügen und S' als Axiom nehmen. □

Wir kommen nun zu Operationen, die aus der Algebra bekannt sind und dort eine wichtige Rolle spielen.

Definition 4.7 *Es seien X und Y zwei Alphabete. Ein Homomorphismus h von X^* in Y^* ist eine eindeutige Abbildung von X^* in Y^*, bei der $h(w_1w_2) = h(w_1)h(w_2)$ für beliebige Wörter w_1 und w_2 aus X^* gilt.*
Ein Homomorphismus heißt nichtlöschend, wenn für alle Wörter $w \neq \lambda$ auch $h(w) \neq \lambda$ gilt.

Wegen $h(w) = h(w \cdot \lambda) = h(w)h(\lambda)$ gilt für jeden Homomorphismus $h(\lambda) = \lambda$.
Es sei $w = a_1 a_2 \dots a_n \in X^+$ mit $a_i \in X$. Dann gilt $h(w) = h(a_1)h(a_2)\dots h(a_n)$. Daher ist $h(w)$ berechenbar, wenn die Werte $h(a_i)$, $1 \leq i \leq n$, bekannt sind. Folglich reicht es aus die Wörter $h(a)$ für $a \in X$ zu kennen, um für jedes Wort $w \in X^*$ das Bild $h(w)$ zu bestimmen.
Für einen nichtlöschenden Homomorphismus ist dann $h(a) \neq \lambda$ für alle $a \in X$.

Beispiel 4.8 Die Homomorphismen h_1 und h_2 von $\{a,b\}^*$ in $\{a,b,c\}^*$ seien durch

$$h_1(a) = ab,\ h_1(b) = bb \quad \text{und} \quad h_2(a) = ac,\ h_2(b) = \lambda$$

gegeben. Dann ergeben sich

$$h_1(abba) = abbbbbab,\ h_1(bab) = bbabbb,\ h_2(abba) = acac),\ h_2(bab) = ac\,.$$

Wir erweitern nun den Homomorphismusbegriff auf Sprachen durch die folgenden Festlegungen.

Definition 4.9 *Es seien X und Y zwei Alphabete, $L \subseteq X^*$ und $L' \subseteq Y^*$ zwei Sprachen und h ein Homomorphismus von X^* in Y^*. Dann setzen wir*

$$h(L) = \{h(w) \mid w \in L\} \quad \textit{und} \quad h^{-1}(L') = \{w \mid w \in X^*,\ h(w) \in L'\}.$$

Beispiel 4.10 *(Fortsetzung von Beispiel 4.8)* Wir betrachten die Homomorphismen h_1 und h_2 aus Beispiel 4.8. Dann ergeben sich

$$\begin{aligned}
h_1(\{a\}^* \cup \{b\}^*) &= \{(ab)^n \mid n \geq 0\} \cup \{b^{2n} \mid n \geq 0\},\\
h_2(\{a\}^* \cup \{b\}^*) &= \{(ac)^n \mid n \geq 0\}\\
&\qquad \text{(die Potenzen von } b \text{ liefern nur das schon vorhandene Leerwort)},\\
h_1(\{a^n b^n \mid n \geq 1\}) &= \{(ab)^n b^{2n} \mid n \geq 1\},\\
h_2(\{a^n b^n \mid n \geq 1\}) &= \{(ac)^n \mid n \geq 1\},\\
h_1^{-1}(\{ab^n \mid n \geq 1\} &= \{ab^n \mid n \geq 0\},\\
h_2^{-1}(\{ac, acac\} &= \{b^i ab^j \mid i \geq 0, j \geq 0\} \cup \{b^r ab^s ab^t \mid r \geq 0, s \geq 0, t \geq 0\},\\
h_1^{-1}(\{a^n b^n \mid n \geq 1\}) &= \{a\},\\
h_2^{-1}(\{a^n b^n \mid n \geq 1\}) &= \emptyset.
\end{aligned}$$

Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter (nichtlöschenden) Homomorphismen abgeschlossen ist, wenn für beliebige Alphabete X und Y, beliebige Sprachen $L \subseteq X^*$ und jeden (nichtlöschenden) Homomorphismusn $h : X^* \to Y*$ aus $L \in \mathcal{L}$ auch $h(L) \in \mathcal{L}$ folgt.
Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter inversen Homomorphismen abgeschlossen ist, wenn für beliebige Alphabete X und Y, beliebige Sprachen $L \subseteq Y^*$ und beliebige Homomorphismen $h : X^* \to Y*$ aus $L \in \mathcal{L}$ auch $h^{-1}(L) \in \mathcal{L}$ folgt.

Lemma 4.11 *i) Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$ und $\mathcal{L}(RE)$ sind unter Homomorphismen abgeschlossen, $\mathcal{L}(CS)$ ist nicht unter Homomorphismen abgeschlossen.*
ii) iii) Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind unter nichtlöschenden Homomorphismen abgeschlossen.
iii) Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind unter inversen Homomorphismen abgeschlossen.

Beweis. Wir beweisen nur die Aussagen i) und ii).
a) $\mathcal{L}(RE)$. Es seien $L \in \mathcal{L}(RE)$ eine Sprache über T und $h : T^* \to Y^*$ ein beliebiger Homomorphismus. Dann gibt es eine Regelgrammatik $G = (N, T, P, S)$ mit $L = L(G)$. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass alle Regeln in P von der Form $A_1A_2 \dots A_n \to B_1B_2 \dots B_s$ mit $A_i \in N$ für $1 \leq i \leq n$ und $B_j \in N$ für $1 \leq j \leq s$ mit gewissen $n \geq 1$ und $s \geq 1$ oder von der Form $A \to a$ oder $A \to \lambda$ mit $A \in N$ und $a \in T$ sind (siehe Lemma 2.15. Wir konstruieren die Regelgrammatik $G' = (N, Y, P', S)$, wobei P' aus P entsteht, indem jede Regel der Form $A \to a$ durch $A \to h(a)$ ersetzt wird. Es ist leicht zu sehen, dass $L(G') = h(L(G))$ gilt. Wegen $h(L) = h(L(G))$ wird daher $h(L)$ durch eine Regelgrammatik erzeugt, womit bewiesen ist, dass $h(L) \in \mathcal{L}(RE)$ liegt.
b) $\mathcal{L}(CF)$. Wir geben den gleichen Beweis unter Verwendung der Chomsky-Normalform (siehe Satz 2.23).
c) $\mathcal{L}(REG)$. Es seien $L \in \mathcal{L}(RE)$ eine Sprache über T und $h : T^* \to Y^*$ ein beliebiger Homomorphismus. Dann gibt es eine reguläre Grammatik $G = (N, T, P, S)$ mit $L = L(G)$, bei der alle Regeln von der Form $A \to aB$, $A \to a$ oder $A \to \lambda$ mit $A, B \in N$ und $a \in T$ sind (siehe Satz 2.24) sind. Wir konstruieren die reguläre Grammatik $G' = (N, Y, P', S)$, wobei P' aus P entsteht, indem jede Regel der Form $A \to aB$ durch $A \to h(a)B$ und jede Regel der Form $A \to a$ durch $A \to h(a)$ ersetzt wird. Es ist leicht zu sehen, dass $L(G') = h(L(G))$ gilt. Wegen $h(L) = h(L(G))$ wird daher $h(L)$ durch eine reguläre Grammatik erzeugt, womit bewiesen ist, dass $h(L) \in \mathcal{L}(RE)$ liegt.
d) $\mathcal{L}(CS)$. Die obige Beweisidee kann für nichtlöschende Homomorphismen unter Verwendung der Normalform aus Satz 2.16 benutzt werden, da die neuen Regel $A \to h(a)$ in monotonen Grammatiken erlaubt sind. Ist der Homomorphismus dagegen löschend, so entsteht eine Regel $A \to \lambda$ aus mindestens einer Regel $A \to a$, die bei monotonen Grammatiken nicht erlaubt sind.
Es sei L eine Sprache aus $\mathcal{L}(RE) \setminus \mathcal{L}(CS)$ und $G = (N, T, P, S)$ eine Grammatik der in a) gegebenen Form mit $L = L(G)$. Wir konstruieren nun die monotone Grammatik $G' = (N, T \cup \{c\}, P', S)$, indem wir jede Regel $A \to \lambda$ aus P durch $A \to c$ ersetzen, wobei $c \notin N \cup T$ gilt. Die Sprache $L(G')$ unterscheidet sich von der Sprache $L(G)$ nur dadurch, dass in den Wörtern zusätzlich an einigen Stellen der Buchstabe c steht. Betrachten wir nun den Homomorphismus h mit $h(a) = a$ für $a \in T$ und $h(c) = \lambda$, so erhalten wir $h(L(G')) = L(G) = L \notin \mathcal{L}(CS)$. Somit ist $\mathcal{L}(CS)$ nicht abgeschlossen unter Homomorphismen. □

Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter Durchschnitten mit regulären Sprachen abgeschlossen ist, wenn für eine beliebige Sprache $L \in \mathcal{L}$ und eine beliebige reguläre Sprache R auch $L \cap R \in \mathcal{L}$ gilt.

Lemma 4.12 *Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind unter Durchschnitten mit regulären Sprachen abgeschlossen.*

Beweis. Da jede reguläre Sprache auch in $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ liegt, folgt die Behauptung für $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ aus Lemma 4.3.
Um die Aussage für $\mathcal{L}(CF)$ zu beweisen, konstruieren wir einen Kellerautomaten $\mathcal{M}$, der $L \cap R$ akzeptiert. Es seien $\mathcal{M}_1 = (X, Z_1, \Gamma, z_{0,1}, F_1, \delta_1)$ ein Kellerautomat, der L akzeptiert, und $\mathcal{A}_2 = (X, Z_2.z_{0,2}, F_2, \delta_2)$ ein deterministischer endlicher Automat, der R akzeptiert. Wir konstruieren nun den Kellerautomaten

$$\mathcal{M} = (X, Z_1 \times Z_2, \Gamma, (z_{0,1}, z_{0,2}), F_1 \times F_2, \delta)$$

mit

$$(z_1', z_2'), R, \beta) \in \delta((z_1, z_2), a, \gamma) \quad \text{falls} \quad (z_1', \beta) \in \delta_1(z_1, a, \gamma) \text{ und } \delta_2(z_2, a) = z_2',$$
$$(z_1', z_2), N, \beta) \in \delta((z_1, z_2), a, \gamma) \quad \text{falls} \quad (z_1', \beta) \in \delta_1(z_1, a, \gamma).$$

Entsprechend dieser Definition verhält sich $\mathcal{M}$ in der ersten Komponente der Zustände und hinsichtlich des Bandes wie $\mathcal{M}_1$ und in der zweiten Komponente der Zustände wie $\mathcal{A}_2$ (wobei von $\mathcal{A}_2$ nur dann ein Buchstabe gelesen wird, wenn auch $\mathcal{M}_1$ diesen Buchstaben liest und nach rechts geht). Damit erfolgte eine Akzeptanz nur, wenn sowohl $\mathcal{M}_1$ und cA_2 das Eingabewort akzeptieren. Damit wird $L \cap R$ akzeptiert. □

Wir fassen die Resultate zu Abschlusseigenschaften in der Tabelle aus Abb. 4.1 zusammen, wobei ein + bzw. – im Schnittpunkt der Spalte mit der Familie $\mathcal{L}$ von Sprachen und der Zeile mit der Operation τ bedeutet, dass $\mathcal{L}$ unter τ abgeschlossen bzw. nicht abgeschlossen ist.

	$\mathcal{L}(RE)$	$\mathcal{L}(CS)$	$\mathcal{L}(CF)$	$\mathcal{L}(REG)$
Vereinigung	+	+	+	+
Durchschnitt	+	+	–	+
Komplement	–	+	–	+
Produkt	+	+	+	+
(positiver) KLEENE-Abschluss	+	+	+	+
Homomorphismen	+	–	+	+
nichtlöschende Homomorphismen	+	+	+	+
inverse Homomorphismen	+	+	+	+
Durchschnitt mit regulären Mengen	+	+	+	+

Abbildung 4.1: Tabelle der Abschlusseigenschaften

Wir geben nun eine Bezeichnung für Sprachfamilien, die gegenüber allen bisher behandelten Operationen abgeschlossen sind, unter denen jede der Familien der Chomsky-Hierarchie abgeschlossen ist.

Definition 4.13 *Eine Menge $\mathcal{L}$ von Sprachen heißt abstrakte Familie von Sprachen (kurz AFL),*

- *wenn sie mindestens eine nichtleere Sprache enthält und*
- *wenn sie unter Vereingung, Produkt, Kleene-Abschluss, nichtlöschenden Homomorphismen, inversen Homomorphismen und Durchschnitten mit regulären Sprachen abgeschlossen ist.*

Die Menge $\mathcal{L}$ heißt volle AFL, wenn sie zusätzlich unter (beliebigen) Homomorphismen abgeschlossen ist.

Entsprechend der Tabelle aus Abb. 4.1 sind $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ AFLs, und $\mathcal{L}(REG)$, $\mathcal{L}(CF)$ und $\mathcal{L}(RE)$ sind sogar volle AFLs.
Wir wollen jetzt eine weitere Charakterisierung der Familie $\mathcal{L}(REG)$ geben. Sie wird sich als die kleinste volle AFL erweisen.

Satz 4.14 *Für jede volle AFL $\mathcal{L}$ gilt $\mathcal{L}(REG) \subseteq \mathcal{L}$.*

Beweis. Es sei $\mathcal{L}$ eine volle AFL und R eine beliebige reguläre Menge über dem Alphabet X. Dann enthält $\mathcal{L}$ eine nichtleere Sprache L. Es sei w ein Wort aus L. Da die endliche Menge $\{w\}$ eine reguläre Menge ist, liegt $L \cap \{w\} = \{w\}$ auch in $\mathcal{L}$. Wir definieren nun für jedes Element $a \in X$ den Homomorphismus h_a durch $h_a(a) = w$ und $h_a(b) = wa$ für jedes $b \in X$ mit $b \neq a$. Offenbar gilt dann $h_a^{-1}(\{w\}) = \{a\}$. Somit liegt jede Menge $\{a\}$ mit $a \in X$ in $\mathcal{L}$. Unter Verwendung des Homomorphismus h mit $h(a) = \lambda$ für alle $a \in X$ und der regulären Menge $\{b\}$, wobei $b \in X$ und $b \neq a$ gelte, erhalten wir, dass

$$\{\lambda\} = h(\{a\}) \text{ und } \emptyset = \{a\} \cap \{b\}$$

in $\mathcal{L}$ liegen. Aus diesen Mengen kann R nach dem Satz von Kleene durch (mehrfache, iterierte) Anwendung von Vereinigung, Produkt und Kleene-Abschluss erzeugt werden. Da $\mathcal{L}$ unter diesen Operationen abgeschlossen ist, ist $R \in \mathcal{L}$ gezeigt. □

Damit ergibt sich sofort die folgende Aussage.

Folgerung 4.15 *Die Menge $\mathcal{L}(REG)$ ist die (bez. der Inklusion) kleinste volle AFL.* □

Die AFLs wurden eingeführt, weil man bemerkte, dass die zu ihrer Definition verwendeten Abschlüsse unter gewissen Operationen den Abschluss unter weiteren Operationen nach sich ziehen. Daher brauchten diese neuen Abschlusseigenschaften nicht für jede Familie einzeln nachgewiesen werden, sondern es kann ein einheitlicher Beweis für alle AFLs gegeben werden. Wir demonstrieren dies an zwei Beispielen.
Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter mengentheoretischer Subtraktion regulärer Sprachen abgeschlossen ist, wenn für eine beliebige Sprache $L \in \mathcal{L}$ und eine beliebige reguläre Sprache R auch $L \setminus R \in \mathcal{L}$ gilt.

Satz 4.16 *Jede AFL ist unter mengentheoretischer Subtraktion regulärer Mengen abgeschlossen.*

Beweis. Es sei $\mathcal{L}$ eine AFL. Für eine Sprache $L \subseteq X^*$ aus $\mathcal{L}$ und eine reguläre Sprache $R \subseteq X^*$ gilt $L \setminus R = L \cap (X^* \setminus R)$. Da das Komplement $X^* \setminus R$ von R nach Lemma 4.4 regulär ist, ist $L \setminus R$ der Durchschnitt von einer Sprache in $\mathcal{L}$ und einer regulären Sprachen, d.h. $L \setminus R$ liegt in $\mathcal{L}$. □

Definition 4.17 *Es seien X und Y Alphabete. Für jeden Buchstaben $a \in X$ sei $\sigma(a)$ eine Sprache über Y. Für eine Sprache $L \subseteq X^*$ definieren die Sprache $\sigma(L) \subseteq Y^*$ durch*

$$\sigma(L) = \{w_1 w_2 \dots w_n \mid a_1 a_2 \dots a_n \in L,\ a_i \in X \text{ und } w_i \in \sigma(a_i) \text{ für } 1 \leq i \leq n,\ n \geq 0\}.$$

Beispiel 4.18 Für die Sprache $L = \{aba, aa\}$ und die Substitutionen σ_1 und σ_2, die durch

$$\sigma_1(a) = \{a^2\},\ \sigma_1(b) = \{ab\} \quad \text{und} \quad \sigma_2(a) = \{a, a^2\},\ \sigma_2(b) = \{b, b^2\}$$

gegeben sind, erhalten wir

$$\begin{aligned}
\sigma_1(L) &= \{a^2aba^2, a^2a^2\} = \{a^3ba^2, a^4\},\\
\sigma_2(L) &= \{aba, a^2ba, aba^2, a^2ba^2, ab^2a, a^2b^2a, ab^2a^2, a^2b^2a^2, aa, a^2a, aa^2, a^2a^2\}\\
&= \{aba, a^2ba, aba^2, a^2ba^2, ab^2a, a^2b^2a, ab^2a^2, a^2b^2a^2, a^2, a^3, a^4\}.
\end{aligned}$$

Wir bemerken, dass ein Homomorphismus $h : X^* \to Y^*$ als Substitution $\sigma : X^* \to Y^*$ aufgefasst werden kann, bei der die Mengen $\sigma(a)$ für $a \in X$ nur ein Wort enthalten.

Satz 4.19 *Jede volle AFL ist unter Substitutionen mit regulären Sprachen abgeschlossen.*

Beweis. Es seien $\mathcal{L}$ eine volle AFL, $L \subseteq X^*$ eine Sprache aus $\mathcal{L}$ und $\tau : X^* \to Y^*$ eine Substitution, bei der $\tau(a)$ für jedes $a \in X$ eine reguläre Sprache über Y ist. Es sei $X = \{a_1, a_2, \ldots, a_n\}$. Ferner gelte $\tau(a_i) = R_i$. Wir definieren nun

$$
\begin{aligned}
&X' = \{a' \mid a \in X\},\\
&h_1 : (X' \cup Y)^* \to X^* \text{ durch } h_1(x') = x \text{ für } x \in X \text{ und } h_1(y) = \lambda \text{ für } y \in Y,\\
&h_2 : (X' \cup Y)^* \to Y^* \text{ durch } h_2(x') = \lambda \text{ für } x \in X \text{ und } h_2(y) = y \text{ für } y \in Y,\\
&R = \bigcup_{i=1}^{n} a_i' R_i.
\end{aligned}
$$

Dann ergibt sich der Reihe nach

$$
\begin{aligned}
h_1^{-1}(L) &= \{u_0 x_1' u_1 x_2' u_2 \ldots x_r' u_r \mid x_1 x_2 \ldots x_r \in L,\ u_i \in Y^* \text{ für } 1 \le i \le r\},\\
h_1^{-1}(L) \cap R &= \{x_1' u_1 x_2' u_2 \ldots x_r' u_r \mid x_1 x_2 \ldots x_r \in L,\ u_i \in \tau(x_i) \text{ für } 1 \le i \le r\},\\
h_2(h_1^{-1}(L) \cap R) &= \{u_1 u_2 \ldots u_r \mid x_1 x_2 \ldots x_r \in L,\ u_i \in \tau(x_i) \text{ für } 1 \le i \le r\}.
\end{aligned}
$$

Damit gilt

$$\tau(L) = h_2(h_1^{-1}(L) \cap R),$$

und nach den für eine AFL geltenden Abschlußeigenschaften ist somit $\tau(L)$ in $\mathcal{L}$. □

Kapitel 5

Ergänzung III : Beschreibungskomplexität endlicher Automaten

5.1 Eine algebraische Charakterisierung der Klasse der regulären Sprachen

Ein (deterministischer) *endlicher Automat* ist ein Quintupel

$$\mathcal{A} = (X, Z, z_0, \delta, F),$$

wobei X und Z Alphabete sind, z_0 ein Element von Z ist, δ eine Funktion von $Z \times X$ in Z ist, und F eine nichtleere Teilmenge von Z ist. X ist die Menge der Eingabesymbole, und Z ist die Menge der Zustände. z_0 und F sind der Anfangszustand und die Menge der akzeptierenden Zustände.
δ ist die Überführungsfunktion von $\mathcal{A}$.
Durch

$$\begin{aligned} \delta^*(z, \lambda) &= z, \text{ für } z \in Z, \\ \delta^*(z, wa) &= \delta(\delta^*(z, w), a) \text{ für } w \in X^*,\ a \in X \end{aligned}$$

erweitern wir δ zu einer Funktion δ^* von $Z \times X^*$ in Z.
Die von $\mathcal{A}$ *akzeptierte* Sprache ist durch

$$T(\mathcal{A}) = \{w \in X^* \mid \delta^*(z_0, w) \in F\}$$

definiert.
Es ist bekannt, dass *eine Sprache L genau dann regulär ist, wenn es einen endlichen Automaten $\mathcal{A}$ mit $L = T(\mathcal{A})$ gibt.*
Wir wollen zuerst zwei algebraische Charakterisierungen regulärer Sprachen herleiten. Dazu benötigen wir die folgenden Definitionen.
Eine Äquivalenzrelation $\sim$ auf der Menge X^* heißt *Rechtskongruenz*, falls für beliebige Wörter $x, y \in X^*$ und $a \in X$ aus $x \sim y$ auch $xa \sim ya$ folgt. Bei Multiplikation von rechts werden äquivalente Wörter wieder in äquivalente Wörter überführt.

Eine Äquivalenzrelation $\sim$ auf X^* heißt *Verfeinerung* der Menge $R \subseteq X^*$, wenn für beliebige Wörter $x, y \in X^*$ aus $x \sim y$ folgt, dass $x \in R$ genau dann gilt, wenn auch $y \in R$ gilt. Dies bedeutet, dass mit einem Wort $x \in R$ auch alle zu x äquivalenten Wörter in R liegen. Folglich liegt die zu $x \in R$ gehörige Äquivalenzklasse vollständig in R. Hieraus resultiert die Bezeichnung Verfeinerung für diese Eigenschaft.
Für eine Äquivalenzrelation $\sim$ bezeichnen wir die Anzahl der Äquivalenzklassen von $\sim$ als Index von $\sim$ und bezeichnen sie mit $Ind(\sim)$. Die Äquivalenzrelation $\sim$ hat *endlichen Index*, falls $Ind(\sim)$ endlich ist.
Wir nennen eine Äquivalenzrelation $\sim$ eine *R-Relation*, falls sie eine Rechtskongruenz mit endlichem Index ist und eine Verfeinerung von R ist.

Beispiel 5.1 Seien $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein endlicher Automat und $R = T(\mathcal{A})$. Wir definieren auf X^* die Relation $\sim_{\mathcal{A}}$ durch

$$x \sim_{\mathcal{A}} y \quad \text{genau dann, wenn} \quad \delta^*(z_0, x) = \delta^*(z_0, y)$$

gilt. Offenbar ist $\sim_{\mathcal{A}}$ eine Äquivalenzrelation. Wir zeigen nun, dass $\sim_{\mathcal{A}}$ eine R-Relation ist.
Sei $x \sim_{\mathcal{A}} y$. Dann gilt nach Definition $\delta^*(z_0, x) = \delta^*(z_0, y)$. Damit erhalten wir

$$\delta^*(z_0, xa) = \delta(\delta^*(z_0, x), a) = \delta(\delta^*(z_0, y), a) = \delta^*(z_0, ya)$$

und somit $xa \sim_{\mathcal{A}} ya$, womit nachgewiesen ist, dass $\sim_{\mathcal{A}}$ eine Rechtskongruenz ist.
Sei erneut $x \sim_{\mathcal{A}} y$. Ferner sei $x \in R$. Dies bedeutet $\delta^*(z_0, x) = \delta^*(z_0, y)$ und $\delta^*(z_0, x) \in F$. Folglich gilt $\delta^*(z_0, y) \in F$, woraus $y \in R$ folgt. Analog folgt aus $y \in R$ auch $x \in R$. Damit ist $\sim_{\mathcal{A}}$ Verfeinerung von R.
Sei $x \in X^*$. Ferner sei $\delta^*(z_0, x) = z$. Dann ergibt sich für die zu x bez. $\sim_{\mathcal{A}}$ gehörende Äquivalenzklasse $K_{\mathcal{A}}(x)$

$$\begin{aligned} K_{\mathcal{A}}(x) &= \{y \mid x \sim_{\mathcal{A}} y\} \\ &= \{y \mid \delta^*(z_0, x) = \delta^*(z_0, y)\} \\ &= \{y \mid \delta^*(z_0, y) = z\}. \end{aligned}$$

Sei nun $z \in Z$ ein von z_0 erreichbarer Zustand. Dann gibt es ein Wort x mit $\delta^*(z_0, x) = z$ und die Beziehungen

$$\begin{aligned} \{y \mid \delta^*(z_0, y) = z\} &= \{y \mid \delta^*(z_0, y) = \delta^*(z_0, x)\} \\ &= \{y \mid y \sim_{\mathcal{A}} x\} \\ &= K_{\mathcal{A}}(x) \end{aligned}$$

sind gültig. Daher gibt es eine eineindeutige Beziehung zwischen den Äquivalenzklassen von $\sim_{\mathcal{A}}$ und den Zuständen von $\mathcal{A}$, die vom Anfangszustand erreichbar sind. Dies bedeutet, dass die Anzahl der Zustände von $\mathcal{A}$ mit dem Index von $\sim_{\mathcal{A}}$ übereinstimmt. Wegen der Endlichkeit der Zustandsmenge Z ist also auch der Index von $\sim_{\mathcal{A}}$ endlich.

Beispiel 5.2 Für eine Sprache $R \subseteq X^*$ definieren wir die Relation $\sim_R$ wie folgt: $x \sim_R y$ gilt genau dann, wenn für alle $w \in X^*$ das Wort xw genau dann in R ist, falls dies auch für yw der Fall ist. Wir zeigen, dass $\sim_R$ eine Rechtskongruenz ist, die R verfeinert.

Seien dazu $x \sim_R y$, $a \in X$ und $w \in X^*$. Dann ist $aw \in X^*$ und nach Definition von $\sim_R$ gilt $xaw \in R$ genau dann, wenn $yaw \in R$ gültig ist. Dies liefert, da w beliebig ist, die Aussage $xa \sim_R ya$. Somit ist $\sim_R$ eine Rechtskongruenz.
Wählen wir $w =$
lambda, so ergibt sich aus $x \sim_R y$ nach Definition, dass $x \in R$ genau dann gilt, wenn auch $y \in R$ erfüllt ist. Deshalb ist $\sim_R$ eine Verfeinerung von R.
Wir bemerken, dass $\sim_R$ nicht notwendigerweise einen endlichen Index haben muss. Dazu betrachten wir

$$R = \{a^n b^n \mid n \geq 1\}$$

und zwei Wörter a^k und a^ℓ mit $k \neq \ell$. Wegen $a^k b^k \in R$ und $a^\ell b^k \notin R$ sind offensichtlich a^ℓ und a^k nicht äquivalent. Somit gibt es mindestens soviel Äquivalenzklassen wie Potenzen von a und damit soviel wie natürliche Zahlen. Dies zeigt die Unendlichkeit des Index.

Ziel dieses Abschnitts ist eine Charakterisierung der regulären Sprachen durch Äquivalenzrelationen mit den oben definierten Eigenschaften.

Satz 5.3 *Für eine Sprache $R \subseteq X^*$ sind die folgenden Aussagen gleichwertig:*
i) R ist regulär.
ii) Es gibt eine R-Relation.
iii) Die Relation $\sim_R$ (aus Beispiel 5.2) hat endlichen Index.

Beweis. i) $\Longrightarrow$ ii). Zu einer regulären Sprache R gibt es einen endlichen Automaten $\mathcal{A}$ mit $T(\mathcal{A}) = R$. Dann können wir entsprechend Beispiel 5.1 die Relation $\sim_{\mathcal{A}}$ konstruieren. Diese ist nach Beispiel 5.1 eine R-Relation.
ii) $\Longrightarrow$ iii) Nach Voraussetzung gibt es eine R-Relation $\sim$ mit endlichem Index. Wir beweisen nun $Ind(\sim_R) \leq Ind(\sim)$.
Dazu zeigen wir zuerst, dass aus $x \sim y$ auch $xw \sim yw$ für alle $w \in X^*$ folgt. Für $w =$
lambda ist dies klar. Für $w \in X$ ist es klar, da $\sim$ eine Rechtskongruenz ist. Sei die Aussage nun schon für $|w| < n$ bewiesen. Wir betrachten das Wort v der Länge n. Dann gibt es ein w der Länge $n-1$ und ein $a \in X$ mit $v = wa$. Nach Induktionvoraussetzung haben wir $xw \sim yw$. Nach der Definition der Rechtskongruenz folgt daraus $xwa \sim ywa$ und damit $xv \sim yv$.
Sei nun $x \sim y$ und $w \in X^*$. Dann gilt auch $xw \sim yw$. Da $\sim$ eine R-Relation ist, folgt $xw \in R$ gilt genau dann, wenn $yw \in R$ gilt. Nach Definition $\sim_R$ bedeutet dies aber $x \sim_R y$. Damit ist gezeigt, dass $x \sim y$ auch $x \sim_R y$ impliziert. Hieraus ergibt sich aber

$$\{y \mid y \sim x\} \subseteq \{y \mid y \sim_R x\}.$$

Jede Äquivalenzklasse von $\sim$ ist also in einer Äquivalenzklasse von $\sim_R$ enthalten. Damit gilt $Ind(\sim_R) \leq Ind(\sim)$. Da $Ind(\sim)$ endlich ist, hat auch $\sim_R$ endlichen Index.
iii) $\Longrightarrow$ i) Mit $K_R(x)$ bezeichnen wir die Äquivalenzklasse von $x \in X^*$ bez. $\sim_R$. Wir betrachten den endlichen Automaten

$$\mathcal{A} = (X, \{K_R(x) \mid x \in X^*\}, K_R(lambda), \delta, \{K_R(y) \mid y \in R\})$$

mit

$$\delta(K_R(x), a) = K_R(xa).$$

Wir bemerken zuerst, dass aufgrund der Voraussetzung, dass $\sim_R$ eine R-Relation ist, die Definition von $\mathcal{A}$ korrekt ist. (Die Endlichkeit der Zustandsmenge von $\mathcal{A}$ folgt aus der Endlichkeit des Index von $\sim_R$. Falls $K_R(x) = K_R(y)$, so ist auch $K_R(xa) = K_R(ya)$, denn $\sim_R$ ist eine Rechtskongruenz und daher gilt mit $x \sim_R y$, d. h. $K_R(x) = K_R(y)$, auch $xa \sim_R ya$.) Ferner beweist man mittels vollständiger Induktion über die Länge von x leicht, dass $\delta^*(K_R($
$lambda), x) = K_R(x)$ für alle $x \in X^*$ gilt. Damit folgt

$$T(\mathcal{A}) = \{x \mid \delta^*(K_R(lambda), x) \in \{K_R(y) \mid y \in R\}\} = \{x \mid K_R(x) \in \{K_R(y) \mid y \in R\}\} = R.$$

Damit ist R als regulär nachgewiesen. □

Im Beweisteil ii) $\Longrightarrow$ iii) wurde eigentlich die folgende Aussage bewiesen.

Folgerung 5.4 *Sei R eine beliebige Sprache. Dann gilt für jede R-Relation $\sim$ die Beziehung $Ind(\sim) \geq Ind(\sim_R)$.* □

5.2 Minimierung deterministischer endlicher Automaten

In diesem Abschnitt diskutieren wir als Komplexitätsmaß die Anzahl der Zustände eines endlichen Automaten. Dazu setzen wir für einen endlichen Automaten $\mathcal{A} = (X, Z, z_0, \delta, F)$ und eine reguläre Sprache R

$$\begin{aligned} z(\mathcal{A}) &= \#(Z), \\ z(R) &= \min\{z(\mathcal{A}) \mid T(\mathcal{A}) = R\}. \end{aligned}$$

Wir sagen, dass $\mathcal{A}$ *minimaler* Automat für R ist, falls $T(\mathcal{A}) = R$ und $z(\mathcal{A}) = z(R)$ gelten. Als erstes geben wir ein Resultat an, dass die Größe $z(R)$ für eine Sprache bestimmt.

Satz 5.5 *Für eine reguläre Sprache R gilt $z(R) = Ind(\sim_R)$.*

Beweis. Wir bemerken zuerst, dass aus dem Teil iii) $\Longrightarrow$ i) des Beweises von Satz 5.3 folgt, dass es einen Automaten $\mathcal{A}$ mit $z(\mathcal{A}) = Ind(\sim_R)$ gibt.
Sei nun $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein beliebiger endlicher Automat mit $T(\mathcal{A}) = R$. Dann ist nach Beispiel 5.1 die Relation $\sim_{\mathcal{A}}$ eine R-Relation, für die $z(\mathcal{A}) = Ind(\sim_{\mathcal{A}})$ gilt. Wegen Folgerung 5.4 erhalten wir daraus sofort $z(\mathcal{A}) \geq Ind(\sim_R)$.
Die Kombination dieser beiden Erkenntnisse besagt gerade die Behauptung. □

Wir beschreiben nun für einen Automaten $\mathcal{A}$ einen Automaten, der minimal für $T(\mathcal{A})$ ist. Sei $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein endlicher Automat. Für $z \in Z$ und $z' \in Z$ setzen wir $z \approx_{\mathcal{A}} z'$ genau dann, wenn für alle $x \in X^*$ genau dann $\delta^*(z, x)$ in F liegt, wenn auch $\delta^*(z', x)$ in F liegt. Falls der Automat $\mathcal{A}$ aus dem Kontext klar ist, schreiben wir einfach $\approx$ anstelle von $\approx_{\mathcal{A}}$

Lemma 5.6 *$\approx_{\mathcal{A}}$ ist eine Äquivalenzrelation auf der Menge der Zustände des Automaten $\mathcal{A}$.*

Beweis. Wir verzichten auf den einfachen Standardbeweis. □

Mit $K_\approx(z)$ bezeichnen wir die Äquivalenzklasse von $z \in Z$ bezüglich $\approx$ $(= \approx_{\mathcal{A}})$. Wir setzen

$$Z_\approx = \{K_\approx(z) \mid z \in Z\} \quad \text{und} \quad F_\approx = \{K_\approx(z) \mid z \in F\}$$

und konstruieren den Automaten

$$\mathcal{A}_\approx = (X, Z_\approx, K_\approx(z_0), \delta_\approx, F_\approx)$$

mit

$$\delta_\approx(K_\approx(z), a) = K_\approx(\delta(z, a)).$$

Wir zeigen, dass die Definition von $\mathcal{A}_\approx$ korrekt ist. Dazu haben wir nachzuweisen, dass die Definition von $\delta_\approx$ unabhängig von der Auswahl des Repräsentanten der Äquivalenzklasse ist. Seien daher z und z' zwei Zustände mit $K_\approx(z) = K_\approx(z')$ und a ein beliebiges Element aus X. Dann muss $z \approx z'$ gelten. Wir betrachten ein beliebiges Wort $w \in X^*$. Dann liegt $\delta^*(z, aw)$ in F genau dann wenn $\delta^*(z', aw)$ in F liegt. Wegen $\delta^*(z, aw) = \delta^*(\delta(z, a), w)$ und $\delta^*(z', aw) = \delta^*(\delta(z', a), w)$ folgt $\delta(z, a) \approx \delta(z', a)$ und damit auch $K_\approx(\delta(z, a)) = K_\approx(\delta(z', a))$, was zu zeigen war.
Wir zeigen nun, dass $\mathcal{A}_\approx$ minimal für die von $\mathcal{A}$ akzeptierte reguläre Sprache ist.

Satz 5.7 *Für jeden Automaten $\mathcal{A}$ ist $\mathcal{A}_\approx$ minimaler Automat für $T(\mathcal{A})$.*

Beweis. Wir zeigen zuerst mittels Induktion über die Wortlänge, dass sich die Definition von $\delta_\approx : Z_\approx \times X \to Z_\approx$ auch auf die Erweiterung $\delta^*_\approx : Z_\approx \times X^* \to Z_\approx$ übertragen lässt, d. h. dass $\delta^*_\approx(K_\approx(z), y) = K_\approx(\delta^*(z, y))$ für alle $y \in X^*$ gilt. Nach (genereller) Definition der Erweiterung haben wir

$$\begin{aligned} \delta^*_\approx(K_\approx(z), & \\ \quad lambda) &= K_\approx(z) = K_\approx(\delta^*(z, \\ \quad lambda)), & \\ \delta^*_\approx(K_\approx(z), a) &= \delta_\approx(K_\approx(z), a) = K_\approx(\delta(z, a)) = K_\approx(\delta^*(z, a)), \end{aligned}$$

womit der Induktionsanfang gesichert ist. Der Induktionsschluss ist durch

$$\begin{aligned} \delta^*_\approx(K_\approx(z), ya) &= \delta_\approx(\delta^*_\approx(K_\approx(z), y), a) \\ &= \delta_\approx(K_\approx(\delta^*(z, y)), a) \\ &= K_\approx(\delta(\delta^*(z, y), a)) \\ &= K_\approx(\delta^*(z, ya)) \end{aligned}$$

gegeben.
Hieraus erhalten wir sofort

$$\begin{aligned} T(\mathcal{A}_\approx) &= \{x \mid \delta^*_\approx(K_\approx(z_0), x) \in F_\approx\} \\ &= \{x \mid K_\approx(\delta^*(z_0, x)) \in F_\approx\} \\ &= \{x \mid \delta^*(z_0, x) \in F\} \\ &= T(\mathcal{A}). \end{aligned}$$

Damit bleibt nur noch $z(\mathcal{A}_{\approx}) = z(T(\mathcal{A}))$ zu zeigen. Wegen Satz 5.5 reicht es nachzuweisen, dass $z(\mathcal{A}_{\approx}) = Ind(\sim_{T(\mathcal{A})})$ erfüllt ist. Dazu betrachten wir die Relation $\sim_{\mathcal{A}_{\approx}}$ entsprechend Beispiel 5.1, für die $z(\mathcal{A}_{\approx}) = Ind(\sim_{\mathcal{A}_{\approx}})$ gilt, und beweisen $\sim_{\mathcal{A}_{\approx}} = \sim_{T(\mathcal{A})}$, womit der Beweis vollständig ist.
Seien zuerst $x \in X^*$ und $y \in X^*$ zwei Wörter mit $x \not\sim_{\mathcal{A}_{\approx}} y$. Dann gilt $\delta^*_{\approx}(K_{\approx}(z_0), x) \neq \delta^*_{\approx}(K_{\approx}(z_0), y)$. Hieraus erhalten wir $K_{\approx}(\delta^*(z_0, x)) \neq K_{\approx}(\delta^*(z_0, y))$ und deshalb $\delta^*(z_0, x) \not\approx \delta^*(z_0, y)$. Nach der Definition von $\approx$ heißt dies, dass es ein Wort $w \in X^*$ gibt, für das entweder

$$\delta^*(\delta^*(z_0, x), w) \in F \quad \text{und} \quad \delta^*(\delta^*(z_0, y), w) \notin F$$

oder

$$\delta^*(\delta^*(z_0, x), w) \notin F \quad \text{und} \quad \delta^*(\delta^*(z_0, y), w) \in F$$

erfüllt ist. Daher gilt entweder $\delta^*(z_0, xw) \in F$ und $\delta^*(z_0, yw) \notin F$ oder $\delta^*(z_0, xw) \notin F$ und $\delta^*(z_0, yw) \in F$ und folglich entweder $xw \in T(\mathcal{A})$ und $yw \notin T(\mathcal{A})$ oder $xw \notin T(\mathcal{A})$ und $yw \in T(\mathcal{A})$. Letzteres bedeutet aber gerade $x \not\sim_{T(\mathcal{A})} y$.
Analog beweist man, dass $x \sim_{\mathcal{A}_{\approx}} y$ für zwei Wörter $x \in X^*$ und $y \in X^*$ auch $x \sim_{T(\mathcal{A})} y$ zur Folge hat.
Somit stimmen die Äquivalenzrelationen $\sim_{\mathcal{A}_{\approx}}$ und $\sim_{T(\mathcal{A})}$ überein, was zu zeigen war. □

Leider geben die bisherigen Betrachtungen keine Hinweis, wie der Index von $\sim_R$ zu gegebenem R zu ermitteln ist, denn nach Definition von $\sim_R$ sind unendlich viele Wörter zu untersuchen, um $x \sim_R y$ festzustellen. Analog ist die Konstruktion von $\mathcal{A}_{\approx}$ nicht algorithmisch, denn auch die Feststellung, ob $x \approx y$ gilt, erfordert die Betrachtung von unendlich vielen Wörtern.
Deshalb beschreiben wir jetzt einen Algorithmus zur Bestimmung von $\approx = \approx_{\mathcal{A}}$ für einen endlichen Automaten $\mathcal{A}$. Dies versetzt uns dann in die Lage, zu $\mathcal{A}$ den minimalen Automaten $\mathcal{A}_{\approx}$ zu konstruieren. Dazu beginnen wir mit einer Liste aller Paare von Zuständen und markieren jeweils ein Paar, wenn sich aus dem Verhalten der beide Zustände des Paares entsprechend der Überführungsfunktion bzw. der Menge der akzeptierenden Zustände und der aktuellen Liste ergibt, dass die beiden Zustände des Paares nicht äquivalent sind.
Der *Reduktionsalgorithmus* besteht aus den folgenden Schritten:

1. Erstelle eine Liste aller Paare (z, z') von Zuständen $z \in Z$ und $z' \in Z$.

2. Streiche ein Paar (z, z') falls entweder $z \in F$ und $z' \notin F$ oder $z \notin F$ und $z' \in F$ gilt.

3. Führe die folgende Aktion solange aus, bis keine Markierungen mehr in der Liste möglich sind: Falls für ein Paar (z, z') und ein $a \in X$ das Paar $(\delta(z, a), \delta(z', a))$ nicht in der Liste ist, so streiche (z, z').

Beispiel 5.8 Wir betrachten den endlichen Automaten

$$\mathcal{A} = (\{a, b\}, \{0, 1, 2, 3, 4, 5\}, 0, \delta, \{1, 2, 5\}),$$

dessen Überführungsfunktion δ dem Zustandsgraphen aus Abb. 5.1 zu entnehmen ist. Wir

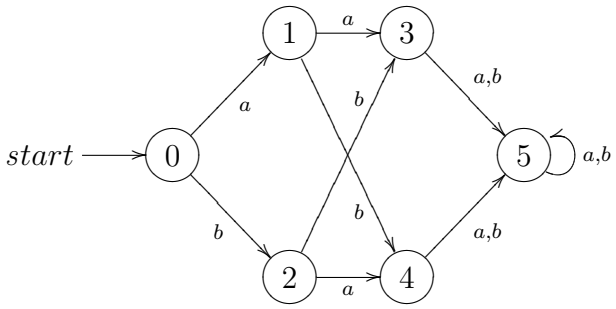

Abbildung 5.1: Zustandsgraph des Automaten $\mathcal{A}$ aus Beispiel 5.8

erhalten entsprechend Schritt 1 des Algorithmus zuerst die Liste

$$\begin{array}{llllll} (0,0), & (0,1), & (0,2), & (0,3), & (0,4), & (0,5), \\ (1,0), & (1,1), & (1,2), & (1,3), & (1,4), & (1,5), \\ (2,0), & (2,1), & (2,2), & (2,3), & (2,4), & (2,5), \\ (3,0), & (3,1), & (3,2), & (3,3), & (3,4), & (3,5), \\ (4,0), & (4,1), & (4,2), & (4,3), & (4,4), & (4,5), \\ (5,0), & (5,1), & (5,2), & (5,3), & (5,4), & (5,5). \end{array}$$

Hieraus entsteht nach Schritt 2 die Liste

$$\begin{array}{llllll} (0,0), & (0,3), & (0,4), & (1,1), & (1,2), & (1,5), \\ (2,1), & (2,2), & (2,5), & (3,0), & (3,3), & (3,4), \\ (4,0), & (4,3), & (4,4), & (5,1), & (5,2), & (5,5). \end{array}$$

Wir haben nun solange wie möglich Schritt 3 anzuwenden. Hierzu gehen wir stets die Liste durch und streichen die entsprechenden Paare und wiederholen diesen Prozess. Nach dem ersten Durchlauf streichen wir nur die Paare $(1,5)$ (da $(\delta(1,a),\delta(5,a)) = (3,5)$ nicht mehr in der Liste ist), $(2,5)$, $(5,1)$ und $(5,2)$. Beim zweiten Durchlauf werden die Paare $(0,3)$ (da das Paar $(\delta(0,a),\delta(3,a)) = (1,5)$ beim ersten Durchlauf gerade gestrichen wurde), $(0,4)$, $(3,0)$ und $(4,0)$ gestrichen. Beim dritten Durchlauf werden keine Streichungen mehr vorgenommen. Damit ergibt sich abschließend die Liste

$$(0,0),\ (1,1),\ (1,2),\ (2,1),\ (2,2),\ (3,3),\ (3,4),\ (4,3),\ (4,4),\ (5,5).$$

Wir zeigen nun, dass mittels des oben angegebenen Algorithmus wirklich die Relation $\approx_{\mathcal{A}}$ berechnet wird. Dies ist im Wesentlichen die Aussage des folgenden Lemmas.

Lemma 5.9 *Sei $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein endlicher Automat, und seien $z \in Z$ und $z' \in Z$ zwei Zustände des Automaten. Dann ist das Paar (z, z') genau dann in der durch den Reduktionsalgorithmus erzeugten Liste enthalten, wenn $z \approx_{\mathcal{A}} z'$ gilt.*

Beweis. Wir beweisen mittels Induktion über die Anzahl der Schritte des Reduktionsalgorithmus die folgende äquivalente Aussage: Das Paar (z, z') wird genau dann durch den Reduktionsalgorithmus aus der Liste gestrichen, wenn es ein Wort $x \in X^*$ gibt, für das entweder $\delta^*(z, x) \in F$ und $\delta^*(z', x) \notin F$ oder $\delta^*(z, x) \notin F$ und $\delta^*(z', x) \in F$ gelten.

Im Folgenden nehmen wir stets ohne Beschränkung der Allgemeinheit an, dass der erste dieser beiden Fälle eintritt.
Erfolgt ein Streichen des Paares (z, z') im zweiten Schritt, so besitzt wegen $\delta^*(z,$ $lambda) = z$ und $\delta^*(z',$
$lambda) = z'$ das leere Wort die gewünschte Eigenschaft. Hat umgekehrt das Leerwort die Eigenschaft, so wird das Paar im zweiten Schritt gestrichen.
Das Paar (z, z') werde nun im dritten Schritt gestrichen. Dann gibt es ein Element $a \in X$ so, dass das Paar $(\delta(z,a), \delta(z',a))$ bereits früher gestrichen wurde. Nach Induktionsannahme gibt es ein Wort $x \in X^*$ mit $\delta^*(\delta(z,a),x) \in F$ und $\delta^*(\delta(z',a),x) \notin F$. Damit haben wir auch $\delta^*(z,ax) \in F$ und $\delta^*(z',ax) \notin F$, womit die Behauptung bewiesen ist. Durch Umkehrung der Schlüsse zeigt man, dass aus der Existenz eines Wortes ax mit $\delta^*(z,ax) \in F$ und $\delta^*(z',ax) \notin F$ folgt, dass (z,z') gestrichen wird. □

Beispiel 5.8 (Fortsetzung) Wir erhalten aus der bereits oben erzeugten Liste die folgenden Äquivalenzklassen bez. $\approx$:

$$K_\approx(0) = \{0\},\ K_\approx(1) = K_\approx(2) = \{1,2\},\ K_\approx(3) = K_\approx(4) = \{3,4\},\ K_\approx(5) = \{5\}.$$

Hieraus resultiert nun entsprechend obiger Konstruktion der minimale Automat

$$\mathcal{A}_\approx = (X, \{K_\approx(0), K_\approx(1), K_\approx(3), K_\approx(5)\}, K_\approx(0), \delta_\approx, \{K_\approx(1), K_\approx(5)\}),$$

dessen Überführungsfunktion aus Abb. 5.2, in der wir $[i]$ anstelle von $K_\approx(i)$, $1 \le i \le 5$, schreiben, entnommen werden kann.

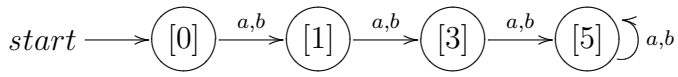

Abbildung 5.2: Zustandsgraph des minimalen Automaten zu $\mathcal{A}$ aus Beispiel 5.8

Wir bemerken, dass aus der Darstellung des minimalen Automaten sofort

$$T(\mathcal{A}) = T(\mathcal{A}_\approx) = X \cup X^3X^*$$

zu sehen ist.

Wir untersuchen nun die Komplexität des Reduktionsalgorithmus. Es gibt n^2 Paare von Zuständen. Bei Schritt 2 oder jedem Durchlauf entsprechend Schritt 3 streichen wir ein Paar oder stoppen. Das Durchmustern einer Liste der Länge r erfordert $O(r)$ Schritte. Damit ist durch

$$\sum_{i=1}^{n^2} O(i) = O\Big(\sum_{i=1}^{n^2} i\Big) = O(n^3)$$

eine obere Schranke für die Komplexität gegeben.
Wir merken hier ohne Beweis an, dass es erheblich effektivere Algorithmen zur Minimierung von endlichen Automaten gibt.

Satz 5.10 *Die Konstruktion des minimalen Automaten zu einem gegebenem endlichen Automaten mit n Zuständen ist in $O(n \cdot \log(n))$ Schritten möglich.* □

Wir wollen nun beweisen, dass der minimale Automat im Wesentlichen eindeutig bestimmt ist. Um das „im Wesentlichen“ exakt zu fassen, geben wir die folgende Definition.

Definition 5.11 *Zwei Automaten $\mathcal{A} = (X, Z, z_0, \delta, F)$ und $\mathcal{A}' = (X, Z', z_0', \delta', F')$ heißen isomorph, wenn es eine eineindeutige Funktion φ von Z auf Z' mit folgenden Eigenschaften gibt:*
– Es ist $\varphi(z_0) = z_0'$.
– Für $z \in Z$ gilt $z \in F$ genau dann, wenn auch $\varphi(z) \in F'$ gilt.
– Für alle $z \in Z$ und $a \in X$ gilt $\delta'(\varphi(z), a) = \varphi(\delta(z, a))$.

Intuitiv liegt Isomorphie zwischen zwei Automaten vor, wenn diese sich nur in der Bezeichnung der Zustände unterscheiden (und ansonsten das gleiche Verhalten zeigen).

Satz 5.12 *Sind $\mathcal{A}$ und $\mathcal{A}'$ zwei minimale Automaten für die reguläre Menge R, so sind $\mathcal{A}$ und $\mathcal{A}'$ isomorph.*

Beweis. Seien $\mathcal{A} = (X, Z, z_0, \delta, F)$ und $\mathcal{A}' = (X, Z', z_0', \delta', F')$ zwei minimale Automaten für R.
Für zwei Zustände $z \in Z$ und $z' \in Z$ von $\mathcal{A}$ ist $z \not\approx_{\mathcal{A}} z'$. Dies folgt daraus, dass aufgrund der Minimalität von $\mathcal{A}$ der Automat $\mathcal{A}_{\approx}$ genau so viel Zustände wie $\mathcal{A}$ hat. Also können keine zwei Zustände von $\mathcal{A}$ in einer Äquivalenzklasse bez. $\approx_{\mathcal{A}}$ liegen. Nun beweist man analog zum letzten Teil des Beweises von Satz 5.7, dass die Relationen $\sim_{\mathcal{A}}$ und $\sim_R$ übereinstimmen.
Entsprechend ergibt sich auch $\sim_{\mathcal{A}'} = \sim_R$ und damit $\sim_{\mathcal{A}} = \sim_{\mathcal{A}'}$.
Sei nun $z \in Z$. Dann gibt es ein Wort $x \in X^*$ mit $\delta^*(z_0, x) = z$. Wir setzen nun

$$\varphi(z) = (\delta')^*(z_0', x).$$

Diese Setzung liefert eine korrekte Definition von φ, denn der Wert von $\varphi(z)$ hängt nicht von der Wahl von x ab. Seien nämlich $x \in X^*$ und $y \in X^*$ mit $\delta^*(z_0, x) = \delta^*(z_0, y)$, so ergibt sich zuerst $x \sim_{\mathcal{A}} y$ und damit auch $x \sim_{\mathcal{A}'} y$, woraus $(\delta')^*(z_0', x) = (\delta')^*(z_0', y)$ folgt.
Wir zeigen, dass φ ein Isomorphismus ist.
Wegen $z_0 = \delta^*(z_0,$
lambda) ergibt sich $\varphi(z_0) = (\delta')^*(z_0', \lambda) = z_0'$.
Sei nun $z \in F$. Dann gibt es ein $x \in R$ mit $z = \delta^*(z_0, x)$. Dann gilt $\varphi(z) = (\delta')^*(z_0', x)$. Wegen $x \in R$ und $R = T(\mathcal{A}')$ erhalten wir $\varphi(z) \in F'$. Sei umgekehrt $\varphi(z) \in F'$. Dann liegt x mit $\varphi(z) = (\delta')^*(z_0', x)$ in R und somit ist $z = \delta(z_0, x) \in F$.
Außerdem gilt

$$\begin{aligned} \delta'(\varphi(z), a) &= \delta'((\delta')^*(z_0', x), a) = (\delta')^*(z_0', xa) \\ &= \varphi(\delta^*(z_0, xa)) = \varphi(\delta(\delta^*(z_0, x), a)) \\ &= \varphi(\delta(z, a)). \end{aligned}$$

□