Proseminar Theoretische Informatik 01.12.2015

Alternative Berechnungsmodelle

Andreas Berg, Pascal Müller, Marius Schidlack *Wolfgang Mulzer*

1 Der Berechenbarkeitsbegriff

1.1 intuitive Berechenbarkeit

Als Menschen mit Programmiererfahrung haben wir ein Gefühl dafür, welche Funktionen auf den natürlichen Zahlen berechenbar sind. Diese Intuition wollen wir formalisieren, damit wir nachweisen können, dass eine Funktion *nicht* berechenbar ist. Dass diese Formalisierung den intuitiven Berechenbarkeitsbegriff erfasst, kann nicht bewiesen werden, da die Intuition eines jeden subjektiv ist. Wir versuchen uns also daran eine Formalisierung anzugeben, auf die sich möglichst viele Menschen einigen können.

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ soll als *(intuitiv) berechenbar* angesehen werden, falls es ein Rechenverfahren gibt, das f berechnet. Genauer, soll das Rechenverfahren mit $n_1, \ldots, n_k \in \mathbb{N}^k$ als Eingabe starten und nach endlich vielen Schritten mit der Ausgabe von $f(n_1, \ldots, n_k)$ stoppen, falls die Funktion für diese Eingabe definiert ist. Ist die Funktion für eine Eingabe nicht definiert, so soll das Rechenverfahren nicht stoppen.

Beispiel:

$$f(n) = \begin{cases} 1\,, & \text{falls } n \text{ ein Anfangsabschnitt der Dezimalbruchentwicklung von } \pi \text{ ist} \\ 0\,, & \text{sonst} \end{cases}$$

ist berechenbar. Als Rechenverfahren verwenden wir ein Näherungsverfahren für π bis zur gewünschten Genauigkeit.

Kann jede reelle Zahl berechnet werden?

1.2 Churchsche These

Die Einführung von Berechnungsmodellen in Form von einfachen Programmiersprachen, wie zum Beispiel die WHILE-Programme, ermöglichen die Angabe von formalen Berechenbarkeitsbegriffen. Zwei dieser Definitionen von Turing und Church (Turingmaschine und Lamda-Kalkül) gehen auf 1936 zurück. Dabei lassen sich Äquivalenzen zwischen den Berechnungsmodellen feststellen.

Bisher hat niemand einen umfassenderen Berechenbarkeitsbegriff erfunden, als den der WHILE-Berechenbarkeit (und äquivalenten Berechenbarkeitsbegiffen; Definition weiter unten). Daraus ergibt sich folgende allgemein akzeptierte Überzeugung (*kein* Satz):

Churchsche These

Die durch die formale Definition der WHILE-Berechenbarkeit (und äquivalenten Berechenbarkeitsmodellen) erfasste Klasse von Funktionen stimmt genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.

Jemand der die Churchsche These vertritt und beweist, dass eine Funktion nicht WHILE-Berechenbar ist, glaubt also, dass diese Funktion überhaupt nicht berechenbar ist.

1.3 Zusammenhang zwischen Funktionen und Sprachen

Bisher haben wir uns mit Sprachen beschäftigt. Wenn Σ ein Alphabet ist, dann ist eine Sprache A eine Menge von Wörtern $A \subseteq \Sigma^*$. Ist ein beliebiges Wort $w \in \Sigma^*$ gegeben, so wollen wir entscheiden, ob $w \in A$. Hier finden wir die Brücke zu Funktionen.

Definition 1 (entscheidbar). *Eine Menge $A \subseteq \Sigma^*$ heißt* entscheidbar, *falls die* charakteristische Funktion *von A, nämlich $\chi_A : \Sigma^* \to \{0,1\}$,* ***berechenbar*** *ist. Hierbei ist für alle $w \in \Sigma^*$:*

$$\chi_A(w) = \begin{cases} 1\,, & w \in A \\ 0\,, & w \notin A \end{cases}$$

Ist die charakteristische Funktion berechenbar, so verfügen wir also über einen immer stoppenden Algorithmus, mit dem wir das Entscheidungsproblem für A lösen (Abbildung Entscheidungsproblem).

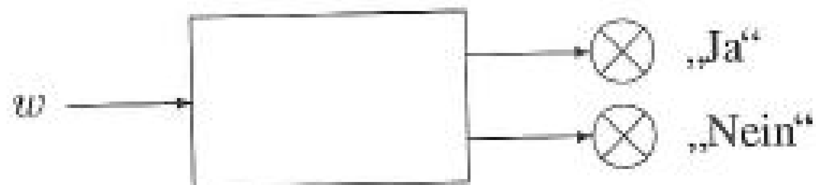

Abbildung 1: Entscheidungsproblem

2 LOOP-, WHILE- und GOTO-Programme

2.1 LOOP-Programme

LOOP ist eine Programmiersprache bestehend aus den folgenden Komponenten:

Variablen: $x_0\ x_1\ x_2 \ldots$
Konstanten: 0 1 2...
Wertzuweisungen: $x_i\ :=\ x_j \pm\ c$ (Es kann ggf. x_j oder c weggelassen werden)
Sequenzen: $P_1;\ P_2$ (P_1 und P_2 werden nacheinander ausgeführt)
Schleifen: *LOOP x DO P END* (P wird so oft ausgeführt, wie es zu Beginn der Schleife in x stand)

Variablen können nur positive Zahlen enthalten und werden auf 0 gesetzt, falls bei einer Subtraktion negative Zahlen entstehen würden. Es kann davon ausgegangen werden, dass die Eingaben für ein Programm zu Beginn in $x_1 \ldots x_n$ stehen. Das Ergebnis wird in x_0 gespeichert.

Definition 2. *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt* ***LOOP-berechenbar****, wenn es ein LOOP-Programm P gibt für das die Eingaben in $x_1 \dots x_n$ stehen und für das im Falle der Terminierung gilt $x_0 = f(x_1, \dots, x_n)$*

Beispiel:

1. Die Addition zweier Variablen $x_i := x_j + x_k$ lässt sich wie folgt definieren

$$x_i := x_j + 0$$
$$LOOP\ x_k\ DO\ x_i := x_i + 1\ END$$

2. Ein Konstrukt $IF\ x = 0\ THEN\ A\ ELSE\ B$

$$a := 1;\ b := 0;$$
$$LOOP\ x\ DO\ a := 0;\ b := 1\ END$$
$$LOOP\ a\ DO\ A\ END$$
$$LOOP\ b\ DO\ B\ END$$

2.2 WHILE-Programme

Wir betrachten nun eine andere Programmiersprache namens WHILE, die über dieselben Komponenten wie LOOP Verfügt, außer dass wir den LOOP-Befehl durch den WHILE-Befehl ersetzen:

$$WHILE\ x \neq 0\ DO\ P\ END$$

Hierbei ist x eine Variable und P ein anderes WHILE-Programm. Dieser Programmabschnitt führt P solange aus, wie $x \neq 0$ ist. Dies wird bei jedem neuen Schleifendurchlauf geprüft. Hierdurch erhalten wir die Klasse der WHILE-berechenbaren Funktionen, deren Definition Analog zu der LOOP-Berechenbarkeit erfolgt.

Satz 3. *Jedes LOOP-Programm kann durch ein WHILE-Programm simuliert werden*

Beweis des Satzes. Ersetze jeden LOOP-Befehl in dem Programm der Form

$$LOOP\ x\ DO\ P\ END$$

durch einen WHILE-Befehl folgender Gestalt

$$y := x$$
$$WHILE\ y \neq 0\ DO\ y := y - 1;\ P\ END$$

Die WHILE-Schleife wird dann genau x-mal ausgeführt und ist, da wir x in y Kopiert haben unabhängig von nachträglichen Änderungen von x. Dies entspricht der Definition der LOOP Schleife. □

2.3 GOTO-Programme

Erneut führen wir eine neue Programmiersprache ein, genannt GOTO. In GOTO existieren weiterhin Variablen, Konstanten und Wertzuweisungen. Jedoch wird nun vor jeder Programmsequenz A_i eine Sprungmarke M_i gesetzt:

$$M_0 : A_0; M_1 : A_1; \ldots$$

Anstatt WHILE oder LOOP Verwenden wir nun folgende Konstrukte:

$$\begin{aligned}&GOTO\ M_i\\&IF\ x_i = c\ THEN\ GOTO\ M_j\\&HALT\end{aligned}$$

GOTO springt in dem Programm zu der angegebenen Sprungmarke und führt das Programm von dort aus weiter aus, ggf. unter der Bedingung, dass $x_i = c$. HALT lässt das Programm terminieren. Vom Programm nicht verwendete Sprungmarken können daher auch weggelassen werden.

Satz 4. *Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden*

Beweis des Satzes. Ersetze jeden WHILE-Befehl in dem Programm der Form

$$WHILE\ x \neq 0\ DO\ P\ END$$

durch eine Reihe an GOTO-Befehlen folgender Gestalt

$$\begin{aligned}&M_i :\ IF\ x \neq 0\ THEN\ GOTO\ M_j;\\&P;\\&GOTO\ M_i\\&M_j :\ \ldots\end{aligned}$$

An der Sprungmarke M_i wird geprüft, ob die Abbruchbedingung gilt. Wenn dies der Fall ist, wird hinter den Code gesprungen, der sich in der WHILE-Schleife befand. □

Wir werden nun durch den Beweis des umgekehrten Falls die Äquivalenz der beiden Programmiersprachen zeigen

Satz 5. *Jedes GOTO-Programm kann durch ein WHILE-Programm (mit nur einer Schleife) simuliert werden*

Beweis des Satzes. Gegeben sei ein GOTO-Programm der Form

$$M_1 : A_1; M_2 : A_2; M_3 : A_3; \ldots M_k : A_k;$$

Man kann dieses Programm durch folgende WHILE-Schleife ersetzen:

$$\begin{aligned}
&zeile := 1\\
&WHILE\ zeile \neq 0\ DO\\
&\quad IF\ zeile = 1\ THEN\ A'_1\\
&\quad IF\ zeile = 2\ THEN\ A'_2\\
&\quad IF\ zeile = 3\ THEN\ A'_3\\
&\quad \ldots\\
&\quad IF\ zeile = k\ THEN\ A'_k\\
&END
\end{aligned}$$

Dabei ergeben sich die A'_i wie folgt

$$A'_i = \begin{cases} x_j := x_l \pm c;\ zeile := zeile + 1 & \text{falls } A_i = x_j := x_l \pm c \\ zeile := n; & \text{falls } A_i = GOTO\ M_n \\ IF\ x_j = c\ THEN\ zeile := n\ ELSE\ zeile := zeile + 1\ END & \text{falls } A_i = IF\ x_j = c\ THEN\ GOTO\ M_n \\ zeile := 0 & \text{falls } A_i = HALT \end{cases}$$

□

Aus dieser Äquivalenz lässt sich auch die Existenz einer Normalform für WHILE-Programme ableiten.

Definition 6. *Ein WHILE-Programm ist in Kleenescher Normalform, wenn sie nur eine WHILE-Schleife enthält.*

Satz 7. *Für jedes WHILE-Programm existiert eine Kleenesche Normalform desselben*

Beweis. Wandle das WHILE-Programm mit genannter Methode in ein GOTO-Programm um, und das entstandene GOTO-Programm zurück in ein WHILE-Programm. Dieses hat nur eine WHILE-Schleife. □

3 Äquivalenz von TM, WHILE und GOTO

3.1 Definitionen

Definition 8 (Deterministische Turingmaschine)**.** *Eine deterministische Turingmaschine M ist definiert als 7-Tupel M=($Q,\Sigma,\Gamma,\delta,q_0,\square,q_f$) und Startet mit dem Eingabewort auf dem Band. Dabei ist Q: Menge aller Zustände; Σ: Eingabealphabet; Γ: Bandalphabet (es gilt auch $\Sigma \subset \Gamma$); δ: Die Übergangsfunktion; q_0: Der Startzustand der TM; $\square$: Das Zeichen für das leere Bandfeld; q_f: Der akzeptierende Endzustand.*
(Dies soll nur eine wiederholende Definition für eine einfache TM sein, keine genaue Defnition aller TM's die in diesem Text benutzt werden)

3.2 WHILE → TM

Satz 9. *Turingmaschinen können WHILE-Programme simulieren. D.h. jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.*

Beweis des Satzes. Jede Eigenschaft eines WHILE-Programms kann durch Mehrband-TM simuliert werden.
A. Variable: Entspricht je einem eigenen Band in der TM (SL-Kopf steht auf aktuellem Wert).
B. Wertezuweisung/Variablenabfragen: Schreiben/Lesen der Werte mit δ auf den den Variablen entsprechenden Bändern.
C. Sequenzierung: Auch TM's können hintereinander geschaltet werden.
D. WHILE-Schleife: Die abzufragende Schleifenbedingung liegt auf dem der Variable entsprechendem Band. Die Übergänge in δ für Wahr/Falsch führen entsprechend zum Schleifenkörper bzw. hinter die Schleife. Der Schleifenkörper kann wie jeder andere Programmteil über δ realisiert werden.

□

3.3 TM → GOTO

Satz 10. *GOTO-Programme können Turingmaschinen simulieren. Also ist jede Turing-berechenbare Funktion auch GOTO-berechenbar.*

Beweis des Satzes. Wir leiten aus einer Einband-TM ein GOTO-Programm ab, das diese simulieren kann.

Diese Simulierung können wir in drei Schritte, I, II, und III unterteilen. Dabei sind I und III lediglich für das Formatieren der Start- und Endkonfiguration der TM zuständig. In II werden wir die eigentliche Logik hinter der Turingmaschine in ein GOTO-Programm überführen, welche dann die Aktionen der Turingmaschine bei entsprechender Eingabe simulieren soll.

Zusatz: Seien Bandalphabet Γ und Zustandsmenge Q unserer TM durchnummeriert.
D.h. $\Gamma=\{a_1,\ldots,a_m\}$ sowie $Q=\{z_1,\ldots,z_k\}$.

I: Formatierung der Startkonfiguration
Die Startkonfiguration unserer TM besteht aus dem Startzustand q_0 und dem initialen Band B. Unser GOTO-Programm besitzt 4 globale Variablen. Zur besseren Übersicht bezeichnen wir diese mit den Buchstaben b, l, r und z.
Die Variable b ist eine Konstante, die beliebig gewählt werden kann, solange b > |Γ| gilt.
Um das Band und die SL-Kopf-Position darzustellen, benutzen wir l und r. Zum Beispiel stellen wir die Konfiguration

$$\left[a_{l1}\ldots a_{lj}\; z_w\; a_{r1}\ldots a_{rp}\right]$$
$$\text{wobei } a_{xy}\in\Gamma \text{ und } z_w\in Q$$

dar mit

$$l=(l_1\ldots l_j)_b$$
$$r=(r_p\ldots r_1)_b$$
$$z=w$$

Hierbei bedeutet $(i_1\ldots i_p)_b$ die Zahl $i_1\ldots i_p$ in b-närer Darstellung, also:

$$x=\sum_{n=1}^{p} i_n\cdot b^{p-n}$$

Hierbei ist zu beachten dass in r die Zeichen des rechten Bandteiles verkehrt herum in die Ziffern übernommen wurden. Somit werden die Bandsymbole links, unter und rechts vom SL-Kopf jeweils dargestellt durch die b^0-Stelle von l, die b^0-Stelle von r und die b^1-Stelle von r.

Die Variable z wird hier auf die Zahl-Darstellung des Startzustandes (w) gesetzt.

II: Logik der TM simulieren

$$
\begin{aligned}
M_0 : \quad & a := \ r\ MOD\ b\ ; \\
& IF\ (z = 1)\ AND\ (a = 1)\ THEN\ GOTO\ M_{11} \\
& IF\ (z = 1)\ AND\ (a = 2)\ THEN\ GOTO\ M_{12} \\
& \vdots \\
& IF\ (z = k)\ AND\ (a = m)\ THEN\ GOTO\ M_{km} \\
M_{11} : \ & \star \\
& GOTO\ M_0\ ; \\
M_{12} : \ & \star \\
& GOTO\ M_0\ ; \\
& \vdots \\
M_{km} : \ & \star \\
& GOTO\ M_0\ ;
\end{aligned}
$$

In der Variablen a steht offensichtlich die Ziffer für das Symbol, über dem sich der SL-Kopf befindet.
An den mit $\star$ markierten Programmstellen simulieren wir nun, was δ tun würde. Bei aktueller Zustand z=1 (bzw. z_1) und aktuellem Bandsymbol a=1 (bzw. a_1) zum Beispiel gehen wir nach M_{11} und führen die drei Aktionen von δ aus. 1. Ziffer für neuen Zustand z':

$$z := \ z'$$

2. Ziffer für neues Zeichen unter dem SL-Kopf a':

$$
\begin{aligned}
r &:= \ r\ DIV\ b \\
r &:= \ r \cdot b\ +\ a'
\end{aligned}
$$

3. SL-Kopf-Bewegung (R/L):

$$
\begin{aligned}
& r := \ r \cdot b\ +\ (l\ MOD\ b)\ und \\
& l := \ l\ MOD\ b\ \rightarrow\ falls\ Linksbewegung, \\
& l := \ l \cdot b\ +\ (r\ MOD\ b)\ und \\
& r := \ r\ MOD\ b\ \rightarrow\ falls\ Rechtsbewegung
\end{aligned}
$$

(4). Falls z' ein akzeptierender Endzustand ist fügen wir noch an:

$$GOTO\ M_E;$$

III: Formatierung der Endkonfiguration
Wenn M_E erreicht wurde, hat die Turingmaschine terminiert. Wir können nun den Schritt I Rückgängig machen, indem wir aus l und r das Band mit den richtigen Bandsymbolen rekonstruieren. Wie das funktioniert sollte offensichtlich sein.

□

4 Die Ackermannfunktion

Die Ackermannfunktion ist ein Beispiel für eine Funktion, die WHILE-berechenbar, aber nicht LOOP-berechenbar ist. Ursprünglich wurde sie 1928 von Ackermann angegeben. Wir betrachten eine vereinfachte Version von Hermes.

Definition 11.

$$a : \mathbb{N}^2 \to \mathbb{N} \quad \textit{ist die Ackermannfunktion mit}$$

$$a(0,y) = y+1 \tag{1}$$

$$a(x,0) = a(x-1,1), \qquad x > 0 \tag{2}$$

$$a(x,y) = a(x-1,a(x,y-1)), \qquad x,y > 0 \tag{3}$$

Diese Funktion wächst sehr schnell an. Ein Gefühl dafür kann man bekommen, wenn man ein paar Funktionswerte betrachtet.

$A(m,n)$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$m=0$	1	2	3	4	5	6
$m=1$	2	3	4	5	6	7
$m=2$	3	5	7	9	11	13
$m=3$	5	13	29	61	125	253
$m=4$	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	$A(3,2^{2^{65536}}-3)$	$A(3,A(4,4))$
$m=5$	65533	$A(4,65533)$	$A(4,A(4,65533))$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,4))$
$m=6$	$A(4,65533)$	$A(5,A(4,65533))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,4))$

Abbildung 2: Funktionswerte der Ackermannfunktion [1]

Auf der Website von Gary Fredericks [2] kann man animierte Berechnungen von Funktionswerten der Ackermannfunktion betrachten. Dabei kommt der häufig wiederholte rekursive Aufruf der Funktion besonders deutlich zur Geltung.

Unser großes Ziel ist es zu zeigen, dass die Ackermannfunktion a nicht LOOP-berechenbar ist. Wenn wir das geschafft haben, bleibt noch zu zeigen, dass a WHILE-berechenbar ist. Dazu führen wir ersteinmal eine Betrachtung einiger Eigenschaften von a durch.

Satz 12. *Die Ackermannfunktion a ist eine* totale *Funktion von* $\mathbb{N}^2 \to \mathbb{N}$. *D. h. a ist für alle* $(x,y) \in \mathbb{N}^2$ *definiert.*

Beweis des Satzes. Induktion nach x mit $y \in \mathbb{N}$ beliebig:
IA: $x = 0$, $a(0,y) = y+1$, a ist definiert.

IV: Sei $a(x-1,y)$ mit $x > 1$ definiert.
IS: Zu zeigen: Dann ist auch $a(x,y)$ definiert.

$$\begin{aligned} a(x,y) &\stackrel{(3)}{=} a(x-1,a(x,y-1)) \\ &= \underbrace{a(x-1,a(x-1,\ldots,a(x-1}_{\text{y-mal}},a(x,0)\ldots)) \\ &\stackrel{(2)}{=} \underbrace{a(x-1,a(x-1,\ldots,a(x-1}_{\text{(y+1)-mal}},1)\ldots)) \quad \text{ist nach } IV \text{ definiert.} \end{aligned}$$

□

Lemma 13. Lemma A $\quad y < a(x,y)$

Beweis von Lemma A. Wir beginnen mit einer Induktion nach x.

$$\begin{aligned} IA_x : &\quad x = 0, \quad y < a(0,y) = y+1 \\ IV_x : &\quad \text{gelte } y < a(x,y) \text{ für } x \in \mathbb{N} \text{ fest und } y \text{ beliebig} \\ IS_x : &\quad \text{zu zeigen: } y < a(x+1,y)) \text{ für alle } y \in \mathbb{N} \end{aligned}$$

Dazu fahren wir mit einer Induktion nach y fort:

$$\begin{aligned} IA_y : &\quad y = 0, \quad 0 < 1 \stackrel{IV_x}{<} a(x,1) \stackrel{(2)}{=} a(x+1,0) \\ IV_y : &\quad \text{gelte} \quad y < a(x+1,y) \\ IS_y : &\quad \text{zu zeigen:} \quad y+1 < a(x+1,y+1) \\ &\quad \text{nach } IV_x \text{ gilt} \quad y < a(x,y) \\ &\quad \text{mit} \quad y := a(x+1,y) \quad \text{(a totale Funktion)} \\ &\quad \text{folgt} \quad a(x+1,y) < a(x,a(x+1,y)) \stackrel{(3)}{=} a(x+1,y+1) \\ &\quad IV_y : \quad y < a(x+1,y) \end{aligned}$$

⇒ In der Ungleichung erscheint zweimal <. Also können wir die linke Seite um eins erhöhen und dabei < beibehalten.

$$\Rightarrow y+1 < a(x+1,y)$$

□

Lemma 14. Lemma B $\quad a(x,y) < a(x,y+1)$

Beweis von Lemma B. Wir zeigen, dass die Behauptung für alle $x \in \mathbb{N}$ gilt.

$$\begin{aligned} x = 0 : &\quad a(0,y) = y+1 < y+2 = a(0,y+1) \\ x > 0 : &\quad \text{Lemma A:} \quad y < a(x,y) \\ &\quad \text{setze ein für y:} \quad a(x,y) \quad \text{(a totale Funktion)} \\ &\quad \text{setze ein für x:} \quad x-1 \quad \text{(x > 0)} \\ &\quad \Rightarrow a(x,y) < a(x-1,a(x,y)) \\ &\quad \stackrel{(3)}{\Leftrightarrow} a(x,y) < a(x,y+1) \end{aligned}$$

$\square$

Lemma 15. Lemma C $\quad a(x,y+1) \leq a(x+1,y)$

Beweis von Lemma C. Induktion nach y.

$$\begin{aligned}
IA: &\quad y=0, \quad a(x,1) \leq a(x,1) \overset{(2)}{=} a(x+1,0)\\
IV: &\quad \text{gelte} \quad a(x,y+1) \leq a(x+1,y)\\
IS: &\quad \text{zu zeigen:} \quad a(x,y+2) \leq a(x+1,y+1)\\
&\quad \text{Lemma A:} \quad y+1 < a(x,y+1)\\
&\quad \Rightarrow y+2 \leq a(x,y+1) \overset{IV}{\leq} a(x+1,y)\\
&\quad \Rightarrow a(x,y+2) \overset{\text{L B}}{\leq} a(x,a(x+1,y)) \overset{(3)}{=} a(x+1,y+1)
\end{aligned}$$

$\square$

Lemma 16. Lemma D $\quad a(x,y) < a(x+1,y)$

Beweis von Lemma D.

$$a(x,y) \overset{\text{L B}}{<} a(x,y+1) \overset{\text{L C}}{\leq} a(x+1,y)$$

$\square$

Satz 17 (Monotonie)**.** *Die Ackermannfunktion a besitzt eine allgemeine Monotonieeigenschaft in folgendem Sinn:*
Seien x, y, x', y' $\in \mathbb{N}$. *Es gilt:*

$$\forall x \leq x',\ \forall y \leq y' \ : \quad a(x,y) \leq a(x',y').$$

Beweis des Satzes.

$$a(x,y) \overset{\text{L B, L D}}{\leq} a(x',y')$$

$\square$

Beobachtung 18 (Beobachtung zu LOOP-Programmen)**.** *Sei P ein LOOP-Programm. Seien* $x_0, x_1, \ldots, x_k$ *die in P vorkommenden Variablen. Seien* n_i *die Startwerte der* x_i*. Seien* n_i' *die Endwerte der* x_i *nach Ablauf von P. Sei* $\quad f_P : \mathbb{N} \to \mathbb{N} \quad$ *eine Funktion. Setze*

$$f_P(n) = max\{\sum_{i=0}^{k} n_i' \mid \sum_{i=0}^{k} n_i \leq n\}.$$

Dann ist $f_P(n)$ *die größtmögliche Summe von Endwerten von P bei Anfangswerten, die in der Summe* $\leq n$ *sind. (Nicht verwendete Variablen haben den Wert 0.)*

Lemma 19. Lemma E $\quad$ *Für jedes LOOP-Programm P gibt es eine Konstante k, so dass für alle n gilt:*

$$f_P(n) < a(k,n) \qquad \textit{(IV)}$$

Beweis von Lemma E. Induktion über den Aufbau von P:
1. P habe die Form $x_i := x_j \pm c$.
o. B. d. A. c $\in \{0,1\}$
$\Rightarrow f_P(n) \leq 2n+1 \qquad (x_i + x_j \leq n)$
Es gilt $a(1,y) = y+2$, $a(2,y) = 2y+3$.
$\Rightarrow k := 2, \quad f_P(n) < a(2,n)$

2. P habe die Form $P_1; P_2$.
Nach IV gilt: $\exists k_1, k_2 \in \mathbb{N}$, so dass gilt:

$$f_{P1}(n) < a(k_1,n), \quad f_{P2}(n) < a(k_2,n)$$

Wähle $k_3 = max\{k_1 - 1, k_2\}$. Es gilt dann:

$$\begin{aligned} f_P(n) &\leq f_{P2}(f_{P1}(n)) \\ &< a(k_2, a(k_1,n)) \qquad \text{(nach IV)} \\ &\overset{Monotonie}{\leq} a(k_3, a(k_3+1,n)) \\ &\overset{(3)}{=} a(k_3+1, n+1) \\ &\overset{\text{L C}}{\leq} a(k_3+2,n) \end{aligned}$$

$$\Rightarrow k := k_3 + 2 \qquad \Rightarrow f_P(n) < a(k_3+2,n)$$

3. P habe die Form LOOP x_i DO Q END.
Nach IV gilt: $\exists k_1 \in \mathbb{N}$ mit $f_Q(n) < a(k_1,n)$.
o. B. d. A. x_i kommt nicht in Q vor. (Die Anzahl der Loops wird einmalig zu Beginn bestimmt. Würde x_i in Q verändert, so hätte dies keinen Einfluss auf die Anzahl der Loops.) Sei $m \leq n$ eine Wahl für den Wert von x_i, bei dem das Maximum bezüglich der Bildung von $f_P(n)$ eingenommen wird. (Wir führen Q m-mal aus.)
3.1 $m = 0 \quad \Rightarrow f_P(n) = n < a(0,n)$
3.2 $m = 1 \quad \Rightarrow f_P(n) \leq f_Q(n) \overset{\text{IV}}{<} a(k_1,n)$
3.3 $m \geq 2$, Abschätzung:

$$
\begin{aligned}
f_P(n) &\leq \underbrace{f_Q(f_Q(\dots f_Q(}_{\text{m-mal}} \underbrace{n-m}_{x_i \text{ nicht in Q}})\dots)) \underbrace{+m}_{x_i} \\
&\overset{\text{IV}}{<} a(k_1, \underbrace{f_Q(f_Q(\dots f_Q}_{\text{(m-1)-mal}}(n-m)\dots)) + m \\
&\vdots \\
&\overset{\text{IV}}{<} \underbrace{a(k_1, a(k_1, \dots a}_{\text{m-mal}}(k_1, n-m)\dots)) + m
\end{aligned}
$$

m-mal $<$ und $+m \quad \Rightarrow$

$$
\begin{aligned}
f_P(n) &\leq a(k_1, a(k_1, \dots a(k_1, n-m)\dots)) \\
&\overset{\text{L D}}{<} \underbrace{a(k_1, a(k_1, \dots a(k_1}_{\text{(m-1)-mal}} a(k_1+1, n-m)\dots)) \\
&\overset{(3)}{=} a(k_1+1, n-1) \\
&\overset{\text{L B}}{<} a(k_1+1, n)
\end{aligned}
$$

$\Rightarrow k := k_1 + 1 \quad \Rightarrow f_P(n) < a(k_1+1, n)$

□

Satz 20 (LOOP-Berechenbarkeit)**.** *Die Ackermannfunktion a ist nicht LOOP-berechenbar.*

Beweis des Satzes durch Widerspruch. Annahme: a ist LOOP-berechenbar.
Sei g(n) = a(n,n)

$\overset{\text{Ann.}}{\Rightarrow}$ g(n) ist LOOP-berechenbar.
Sei P ein LOOP-Programm zu g.

Nach Definition von $f_P(n)$ gilt: $g(n) \leq f_P(n)$.
Nach Lemma E gibt es $k \in \mathbb{N}$, so dass für alle $n \in \mathbb{N}$ gilt:

$$f_P(n) < a(k, n)$$

Für $n = k$ ergibt das:

$$g(k) \leq f_P(k) < a(k,k) = g(k)$$

Widerspruch: $g(k) < g(k)$

$\Rightarrow$ a ist *nicht* LOOP-berechenbar.

□

Beobachtung 21. *Die Ackermannfunktion a ist intuitiv berechenbar. Denn wir können einen Algorithmus angeben, der a berechnet. Zum Beispiel in Python:*

```
def ack1(M, N):
    return (N + 1) if M == 0 else (
        ack1(M-1, 1) if N == 0 else ack1(M-1, ack1(M, N-1)))
```

Auf [3] sind 172 Algorithmen angegeben, die die Ackermannfunktion berechnen. Glaubt man der Churchschen These, so glaubt man nun auch, dass a Turing-berechenbar und damit wegen der oben gezeigten Äquivalenz auch WHILE-berechenbar ist.

Es gibt die Möglichkeit einen formalen Beweis zu führen um zu zeigen, dass a WHILE-berechenbar ist. Wir wollen hier nur die Idee skizieren:
- Schreibe ein Stack-Programm mit push und pop, welches a berechnet. (z. B. in [4] notiert) - Zeige, dass dieses Stack-Programm durch ein WHILE-Programm simuliert werden kann.
- c : $\mathbb{N}^2 \rightarrow \mathbb{N}$ Codierungsfunktion.
- Nutze c und ihre Umkehrfunktionen um den Inhalt des Stacks als einzelne Zahl darzustellen und auch um

$y \downarrow$ / $x \rightarrow$	0	1	2	3	4
0	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Abbildung 3: c-Funktion

die Stack-Operationen umzusetzen.
- c und ihre Umkehrfunktionen sind primitiv rekursiv. Damit sind sie ebenfalls LOOP-berechenbar und somit WHILE-berechenbar.

5 Zusammenfassung

Wir haben den intuitiven Berechenbarkeitsbegriff von Funktionen betrachtet. Dieser leitete uns zu der Churchschen These und zu formalen Berechnungsmodellen, welche auch den Berechenbarkeitsbegriff formalisieren. Von diesen Berechnungsmodellen wurden hier LOOP-, GOTO- und WHILE-Berechenbarkeit betrachtet. Wir haben die Äquivalenz zwischen GOTO-, WHILE- und Turing-Berechenbarkeit bewiesen und diese so einer einzigen Berechenbarkeitsklasse zugeordnet. Echt davon unterscheiden konnten wir den Begriff der LOOP-Berechenbarkeit. Dies gelang durch Untersuchung der Ackermannfunktion, welche nicht LOOP-Berechenbar, aber WHILE-berechenbar ist.

Literatur

[1] Susan Stepney, http://www-users.cs.york.ac.uk/ susan/cyc/a/ackermnn.htm, Zugriff 24.11.2015 11:45 Uhr

[2] Gary Fredericks, http://www.gfredericks.com/sandbox/arith/ackermann, Zugriff 24.11.2015 11:49 Uhr

[3] Rosetta Code wiki, 2007 von Mike Mol gegründet, http://rosettacode.org/wiki/Ackermann_function, Zugriff 24.11.2015 12:06 Uhr

[4] U. Schöning. Theoretische Informatik – kurz gefasst. Spektrum Akademischer Verlag, 5. Auflage, 2008