
5. Varianten des Turingmaschinen-Konzeptes

I: Varianten der Programmstruktur

In der Literatur findet sich eine Vielzahl von Varianten des Turingmaschinen-Konzeptes, die sich alle als äquivalent zum Grundkonzept erweisen. Untersucht hat man diese Varianten zum Teil um die Robustheit des TM-Konzeptes zu zeigen und um die Church-Turing-These zu untermauern. Ziel war es jedoch auch, effizientere (d.h. schnellere) Maschinen zu finden bzw. Maschinen mit besser strukturierbaren Programmen. In diesem Abschnitt betrachten wir Varianten des Programmformats, im nächsten Abschnitt dann Varianten der Basismaschine.

Die Unterscheidung von Operations- und Testschritten bei mathematischen Maschinen dient der klaren Darstellung der logischen Struktur und der Funktionsweise der (Basis-) Maschine. Die Programme werden hierdurch jedoch oft recht umständlich. Bei geeignetem Testsatz TEST verschmilzt man daher Test- und Operationsanweisungen zu *bedingten Anweisungen* (i, t, f, j) mit der Bedeutung, dass, falls im Programmzustand i der Test t positiv ausgeht, die Operation f ausgeführt und der Nachfolgezustand j angenommen wird (*if-then-Anweisung*).

Bedingte Anweisungen benutzt man meist bei *eindeutigen* und *vollständigen* Testmengen, d.h. Mengen TEST mit den Eigenschaften

$$\begin{aligned} \forall s \in S \exists! t \in \text{TEST} (t(s) = 1) & \quad (\text{Eindeutigkeit}) \\ \forall s \in S \exists t \in \text{TEST} (t(s) = 1) & \quad (\text{Vollständigkeit}) \end{aligned}$$

(dies gilt offensichtlich für die Tests der Basis-Turingmaschinen.) In diesem Fall erhält man (wegen der Eindeutigkeit) deterministische Programme $P = \{I_i : i \leq n\}$ aus bedingten Anweisungen, indem man fordert, dass sich je zwei verschiedene Instruktionen von P in ihren beiden ersten Komponenten unterscheiden:

$$(i, t, f, j), (i', t', f', j') \in P \ \& \ (i, t, f, j) \neq (i', t', f', j') \Rightarrow i \neq i' \vee t \neq t'.$$

Durch die weitere Forderung, dass es auch umgekehrt zu jeder mit i, t beginnenden Anweisung für jedes $t' \in \text{TEST}$ eine mit i, t' beginnende Anweisung gibt, stellt die Vollständigkeit von TEST sicher, dass die Maschine wie bisher genau dann stoppt, wenn sie erstmals einen Stoppzustand erreicht. Erlaubt die Basismaschine zusätzlich eine den Speicherinhalt nicht verändernde Transformation, so kann man die Äquivalenz von Programmen und Programmen aus bedingten Anweisungen zeigen. Wir zeigen dies hier nur für den Fall der Turingmaschinen. Dabei fassen wir zusätzlich Druck- und Bewegungsbefehle zusammen und schreiben statt des Tests t_a nur den Buchstaben a selbst. Die bedingte Anweisung

$$(i, a, a', B, j)$$

ist also zu lesen als: Liest man im Zustand i den Buchstaben a auf dem Arbeitsfeld, so überdrückt man diesen mit a' , führt die Kopfbewegung B aus und geht in den Zustand j .

5.1 LEMMA. *Das Turingmaschinenkonzept und das Konzept der Turingmaschine mit bedingten Anweisungen sind äquivalent. D.h. zu jeder TM $M = (B, P)$ gibt es eine äquivalente TM $M' = (B, P')$, deren Programm P' aus bedingten Anweisungen besteht, und umgekehrt.*

BEWEIS. Sei die TM $M = (B, P)$ gegeben. Ersetzt man jede Operationsanweisung $(i, a, j) \in P$ ($a \in \Gamma$) bzw. $(i, B, j) \in P$ ($B \in \{L, R, S\}$) durch die bedingten Anweisungen (i, a', a, S, j) bzw. (i, a', a', B, j) für alle $a' \in \Gamma$ und jede Testanweisung $(i, t_a, j, k) \in P$ durch die bedingten Anweisungen (i, a, a, S, k) und (i, a', a', S, j) für alle $a' \in \Gamma - \{a\}$, dann erhält man ein Programm P' aus bedingten Anweisungen, so dass $M = (B, P)$ und $M' = (B, P')$ die selbe Funktion berechnen.

Ist umgekehrt $M' = (B, P')$ eine TM mit Programm P' aus bedingten Anweisungen, so erhält man eine äquivalente übliche TM $M = (B, P)$ durch Definition von P wie folgt: Sei $z : Z_{P'} \times \Gamma \times \{+, -, \times\} \rightarrow \mathbb{N} - \{Z_{P'}\}$ eine injektive Abbildung und sei $\Gamma = \{a_0, \dots, a_n\}$. Dann enthält P für jeden Zustand $i \in Z_{P'}$ und für $0 \leq l < n$ die Testanweisungen

$$\begin{aligned} & (i, t_{a_0}, z(i, a_0, -), z(i, a_0, +)) \\ & (z(i, a_l, -), t_{a_{l+1}}, z(i, a_{l+1}, -), (z, i, a_{l+1}, +)) \end{aligned}$$

Diese überführen den Zustand i in den Zustand $z(i, a, +)$, wobei a der Buchstabe auf dem Arbeitsfeld ist, ohne hierbei den Speicher zu verändern. Jede bedingte Anweisung (i, a, a', B, j) aus P' wird also durch Hinzunahme von $(z(i, a, +), a', z(i, a, \times))$ und $(z(i, a, \times), B, j)$ zu P korrekt simuliert. \square

5.2 BEISPIEL. Die im letzten Abschnitt beschriebene Idee für ein Programm zur Berechnung der Addition zweier natürlicher Zahlen führt zu folgendem Programm P aus bedingten Anweisungen¹:

$$P = \{(0, b, b, R, 1), (1, 1, 1, R, 1), (1, b, 1, L, 2), (2, 1, 1, L, 2), (2, b, b, R, 3), (3, 1, b, R, 4), (4, 1, b, S, 5)\} \quad (5.1)$$

Üblicherweise stellen wir solch ein Programm übersichtlicher in Form einer Tabelle dar:

Z	\times	Γ	\rightarrow	Γ	\times	Bew	\times	Z
0		b		b		R		1
1		1		1		R		1
1		b		1		L		2
2		1		1		L		2
2		b		b		R		3
3		1		b		R		4
4		1		b		S		5

¹Wie üblich geben wir nur die relevanten Instruktionen von P an. Um P zu vervollständigen, hat man für jeden Nicht-Stoppzustand i von P und jedes $a \in \Gamma$, für die keine mit i, a beginnende Instruktion in P vorkommt, die Instruktion (i, a, a, S, i) hinzuzufügen. (Dies führt in durch i, a gekennzeichneten Konfigurationen zu einer Endlosschleife.)

Die beiden bislang betrachteten Programmformate benutzen als Kontrollstrukturen *Sprünge* (= *goto*; hier durch die Nachfolgeadresse realisiert) und *Fallunterscheidungen* (= *if-then*; hier durch Tests in Form von bedingten Sprüngen bzw. allgemeinen bedingten Anweisungen realisiert). Diese sind für Maschinensprachen üblich (und reichen – nach der Church-Turing-These – aus, um alle anderen (effektiven) Kontrollstrukturen zu simulieren). Höhere Programmiersprachen basieren wesentlich auf *Schleifen* als Kontrollstrukturen, insbesondere den *while-Schleifen*. Hierbei wird eine Anweisung(sfolge) so lange iteriert, wie eine gegebene Schleifenbedingung erfüllt ist. (Die Anzahl der Schleifendurchläufe ist also dynamisch bestimmt, und es kann passieren, dass die Schleife unendlich oft durchlaufen wird, also die Rechnung nicht terminiert.) Eine Variante der *while-Schleife* ist die *repeat-until-Schleife*, bei der die Schleife bei erstmaliger Erfüllung der Schleifenbedingung verlassen wird.

Zur Präzisierung dieser Schleifenkonzepte sei $f : S \rightarrow S$ eine Speichertransformation, $t : S \rightarrow \{0, 1\}$ ein Test und $\bar{t} : S \rightarrow \{0, 1\}$ der zu t *duale Test* (d.h. $\bar{t}(s) = 1 - t(s)$). Wir definieren die *n-fache Iteration* $\text{Iter}(f, n) = f^n$ von f durch

$$\begin{aligned} \text{Iter}(f, 0)(s) &= f^0(s) = s \\ \text{Iter}(f, n+1)(s) &= f^{n+1}(s) = f(f^n(s)) = f(\text{Iter}(f, n)(s)), \end{aligned}$$

und die *Iteration von f nach t* durch

$$\text{Iter}_t(f)(s) = f^{n_s}(s),$$

wobei n_s das kleinste n mit $t(f^n(s)) = 1$ ist, falls solch ein n existiert und

$$\text{Iter}_t(f)(s) \uparrow$$

andernfalls. Hier wird also die Funktion f so lange iteriert, bis der Test t erstmals gilt. Dies aber ist gerade die Bedeutung der *repeat-Anweisung*²:

$$\text{repeat } \{f\} \text{ until } t \equiv \text{Iter}_t(f).$$

Betrachtet man $\text{Iter}_{\bar{t}}(f)$, so wird f so lange iteriert, wie t gilt, d.h.

$$\begin{aligned} \text{while } t \text{ do } \{f\} &\equiv \text{Iter}_{\bar{t}}(f) \equiv \text{repeat } \{f\} \text{ until } \bar{t} \\ \text{repeat } \{f\} \text{ until } t &\equiv \text{while } \bar{t} \text{ do } \{f\} \end{aligned}$$

Hieraus ergibt sich sogleich die Äquivalenz dieser beiden Schleifen-Konzepte, falls wir es mit einer komplementierten Testmenge TEST zu tun haben, d.h. einer Testmenge, die mit jedem Test auch dessen dualen Test enthält:

$$\forall t \in \text{TEST} \quad (\bar{t} \in \text{TEST}) \quad (\text{Komplementiertheit})$$

Man beachte, dass die Testmenge einer Turingmaschine jedoch nur für das binäre Bandalphabet $\Gamma = \{b, 1\}$ komplementiert ist.

Als nächstes betrachten wir *repeat-until-Programme* für Turingmaschinen, die wir *Turingoperatoren* nennen. Der Programmablauf ist hier bis auf die durch Schleifen bedingte iterative Ausführung von Programmteilen sequentiell, weshalb eine Adressierung der Instruktionen überflüssig ist. Turingoperatoren sind (möglicherweise partielle) Speichertransformationen. Die von einem Turingoperator berechnete Funktion erhält man durch Hinzunahme eines geeigneten Ein- und Ausgabemechanismus (s. weiter unten).

²Warnung: Unsere Notation weicht hier von der üblichen ab. In der Regel wird in einer Repeat-Anweisung zuerst die Anweisung f ausgeführt und dann getestet, sodass f zumindest einmal ausgeführt wird. Diese übliche Interpretation entspricht also unserer Befehlsfolge $f; \text{repeat } \{f\} \text{ until } t$.

5.3 DEFINITION. Sei B eine Turing-Basismaschine mit Bandalphabet Γ und Speicher $S = BI \times \mathbb{Z}$. Die Turingoperatoren (TO) $P : S \rightarrow S$ über B sind induktiv wie folgt definiert:

1. R, L, S und alle $a \in \Gamma$ sind Turingoperatoren über B , wobei

$$\begin{aligned} R(f, z) &= (f, z + 1) \\ L(f, z) &= (f, z - 1) \\ S(f, z) &= (f, z) \\ a(f, z) &= (f_{(a,z)}, z) \end{aligned}$$

Hierbei unterscheidet sich die bereits im letzten Abschnitt verwendete Funktion $f_{(a,z)}$ von f höchstens an der Stelle z , wobei $f_{(a,z)}(z) = a$. Die Turingoperatoren dieser Gruppe entsprechen gerade den elementaren Operationen.

2. Sind P_1 und P_2 Turingoperatoren über B , so ist auch P_1P_2 ein Turingoperator über B , wobei

$$P_1P_2(f, z) = P_2(P_1(f, z))$$

(Sprechweise: „erst P_1 , dann P_2 “)

3. Ist P ein Turingoperator über B und $a \in \Gamma$, so ist auch $[P]_a$ ein Turingoperator über B , wobei

$$[P]_a(f, z) = \text{Iter}_{t_a}(P)(f, z).$$

(Sprechweise: „iteriere P so lange, bis a erstmals auf dem Arbeitsfeld steht“)

5.4 BEISPIEL. Das in den vorhergehenden Beispielen beschriebene Vorgehen zur Addition zweier natürlicher Zahlen wird durch den Turingoperator

$$P = R[R]_b 1[L]_b RbRb$$

beschrieben.

BEMERKUNG. Unsere Schreibweise für TOs ist sehr kompakt, weicht aber von der üblicher Programmiersprachen ab. Üblich ist

$$\begin{aligned} &P_1; \\ &P_2 \end{aligned}$$

statt P_1P_2 zu schreiben und – wie schon oben ausgeführt –

$$\text{repeat } \{P\} \text{ until } a$$

statt $[P]_a$. Der Operator aus Beispiel 5.4 liest sich dann als

$$\begin{aligned} &R; \\ &\text{repeat } \{R\} \text{ until } b; \\ &1; \\ &\text{repeat } \{L\} \text{ until } b; \\ &R; \\ &b; \\ &R; \\ &b. \end{aligned}$$

Für den späteren Gebrauch definieren wir eine Reihe von Turingoperatoren, die gängige Speicheroperationen beschreiben.

Der *Rechtsoperator*

$$R_a = [R]_a$$

($a \in \Gamma$) verschiebt das Arbeitsfeld nach rechts bis dieses erstmals den Buchstaben a enthält. Bei der Variante

$$R_{aa} = R_a R [R_a R]_a L$$

wird das Arbeitsfeld nach rechts auf das erste Feld verschoben, sodass dieses und das rechte Nachbarfeld beide den Buchstaben a enthalten. Die dualen *Linksoperatoren* sind

$$L_a = [L]_a \text{ und } L_{aa} = L_a L [L_a L]_a R,$$

wobei L_{aa} auf dem rechten a des ersten Vorkommens von aa links der ursprünglichen Kopfposition stehen bleibt. (Wir werden beim Entwurf von Programmen darauf achten, dass die relevante Bandinschrift nie zwei aufeinander folgende Blanks (bb) enthält. Der Operator R_{bb} verlegt dann das Arbeitsfeld an das „rechte Bandende“, d.h. das erste Blank rechts der relevanten Inschrift. Entsprechend führt L_{bb} zum „linken Bandende“.)

Für ein Bandalphabet $\Gamma \supseteq \{b, 0, 1\}$ wird das durch

$$K(\dots v \underset{\uparrow}{\underline{m}} \underline{b} \bar{\underline{b}} \dots) = (\dots v \underset{\uparrow}{\underline{m}} \underline{b} \bar{\underline{b}} \underline{m} \underline{b} \dots)$$

beschriebene Kopieren der ersten Komponente eines am rechten Bandende stehenden Tupels von Zahlen (genauer: Zahldarstellungen) an dessen Ende, wobei der links hier von stehende Bandteil nicht tangiert wird und das Arbeitsfeld hinter die kopierte Zahl verlegt wird, durch den folgenden *Kopieroperator* realisiert:

$$K = ROR_{bb}R1L_01R[OR_{bb}1L_01R]_b$$

(Der erste Befehl 0 ersetzt die erste zu kopierende 1 in \underline{m} durch eine 0, die durch den ersten Befehl 1 am rechten Bandende kopiert wird. Man läuft dann zu der gesetzten Marke 0 zurück, wandelt diese wieder in eine 1 um (zweiter Befehl 1) und geht zum rechten Nachbarfeld. Man iteriert dann diese Schrittfolge solange, bis man hinter \underline{m} , d.h. auf ein b gelangt ist ($[\dots]_b$). Hierbei ist jetzt allerdings am rechten Bandende die zu kopierende 1 direkt (ohne Zwischen-Blank) hinter die letzte 1 zu schreiben, weshalb wir den ersten Kopierschritt gesondert beschreiben mussten.) Wie sich der Kopieroperator bei einer Speicherbelegung verhält, die nicht der oben beschriebenen Situation entspricht, interessiert uns nicht.

Der *Löschoperator*

$$E_r = R[bR]_b$$

löscht eine rechts des Arbeitsfeldes stehende Zahl und bleibt hinter dieser stehen:

$$E_r(\dots v \underset{\uparrow}{\underline{n}} \underline{b} w \dots) = (\dots v \underset{\uparrow}{b} b^{n+1} \underline{b} w \dots)$$

Die Wirkung von

$$E_l = L[bL]_b$$

ist dual.

Schließlich betrachten wir einen *Transportoperator* T , der eine (z.B. durch Löschen entstandene) Lücke zwischen zwei am rechten Bandende stehenden Zahlen schließt, d.h.

$$T(\dots v \underline{b} m \underline{b}^l \underline{b} \underline{n} b \dots) = (\dots v \underline{b} m \underline{b} \underline{n} b \dots) \quad (l \geq 0)$$

leistet, wobei wir wiederum von $\{b, 0, 1\} \subseteq \Gamma$ ausgehen:

$$T = ROL_1 RR1R_0 bR[OL_1 R1R_0 bR]_b L_1 L_b$$

Wir haben noch nicht die von einem Turingoperator P über einer Basis-Turingmaschine $B = TB(\Sigma, T, \Gamma, n)$ berechnete Funktion definiert. Wir können dies tun, indem wir den üblichen Ein- und Ausgabemechanismus für Turingmaschinen verwenden. Dann ist die von P berechnete partielle Funktion $\text{resp} : (\Sigma^*)^n \rightarrow T^*$ durch

$$\text{resp}(w_1, \dots, w_n) = \text{out}(P(\text{in}(w_1, \dots, w_n)))$$

gegeben, wobei in und out die Ein- und Ausgabefunktionen der Basismaschine B sind. Die Funktionsklassen $F(\text{TO})$ und $F^{(n)}(\text{TO})$ sind dann in Entsprechung zu $F(\text{TM})$ und $F^{(n)}(\text{TM})$ definiert.

Beschränken wir uns auf die Berechnung arithmetischer Funktionen, so sind das Turingmaschinen- und das Turingoperatorenkonzept äquivalent, d.h. es gilt $F(\text{TM}) = F(\text{TO})$. Zum Nachweis hiervon werden wir benutzen, dass wir uns hier auf TMs über dem Bandalphabet $\Gamma = \{b, 1\}$ beschränken können, für die – wie oben bemerkt – die Testmenge komplementiert ist. Wir zeigen hier zunächst nur die eine Richtung und verschieben den Beweis der anderen Richtung auf später.

5.5 SATZ. $F(\text{TO}) \subseteq F(\text{TM})$.

Da TOs und TMs denselben Ein/Ausgabemechanismus besitzen, genügt es zu zeigen, dass jeder TO durch eine TM wie folgt simuliert werden kann:

5.6 LEMMA. Zu jedem TO P über B gibt es eine Turingmaschine $M = (B, P')$ mit Startzustand α_M und einzigem Stoppzustand ω_M , sodass M die Startkonfiguration $(\alpha_M, (f, z))$ in die Stoppkonfiguration $(\omega_M, P(f, z))$ überführt, falls $P(f, z) \downarrow$, und sonst nicht terminiert.

BEWEIS. Das Programm P' der Maschine M wird durch Induktion nach dem Aufbau von P definiert.

1. $P \in \Gamma \cup \{R, L, S\}$. Dann ist $P' = \{(0, P, 1)\}$, also $\alpha_M = 0$ und $\omega_M = 1$.
2. $P = P_1 P_2$. Nach I.V. gibt es Programme P'_1 und P'_2 mit zugehörigen Startzuständen $\alpha_{M_1}, \alpha_{M_2}$ und Stoppzuständen $\omega_{M_1}, \omega_{M_2}$, die P_1 und P_2 wie gewünscht simulieren. Durch eventuelles Umbenennen der Zustände können wir o.B.d.A. annehmen, dass $Z_{P'_1} = \{\alpha_{M_1} = 0, 1, \dots, r = \omega_{M_1}\}$ und $Z_{P'_2} = \{\alpha_{M_2} = r, r+1, \dots, s = \omega_{M_2}\}$ gilt. Dann ist $P' = P'_1 \cup P'_2$ das gewünschte Programm mit $\alpha_M = 0$ und $\omega_M = s$.

3. $P = [P_1]_a$. Für das nach I.V. zu P_1 passende Programm P'_1 mit $\alpha_{M_1}, \omega_{M_1}$ und für neue Zustände $\alpha_M, \omega_M \notin Z_{P'_1}$ ist dann

$$P' = P'_1 \cup \{(\alpha_M, t_a, \alpha_{M_1}, \omega_M), (\omega_{M_1}, t_a, \alpha_{M_1}, \omega_M)\}$$

das gewünschte Programm. □

Bei dem von uns gewählten Maschinenmodell dürfen die Eingaben während der Rechnung im Speicher zerstört werden. Dies erweist sich als gravierender Nachteil, wenn man Maschinen, die auf dieselbe Eingabe zugreifen, hintereinanderschalten möchte.

Dieses Problem lässt sich vermeiden, wenn man *konservative Rechnungen* betrachtet, bei denen die Eingaben (sowie das Band links der Eingaben) erhalten bleiben und die Ausgabe hinter die Eingaben geschrieben wird. Wir präzisieren dies hier (für späteren Gebrauch) nur für den Fall von Turingoperatoren zur Berechnung arithmetischer Funktionen:

5.7 DEFINITION. Der Turingoperator P über der Basismaschine $TB(\Sigma_1, \Sigma_1, \Gamma, n)$ berechnet die partielle Funktion $\varphi: \mathbb{N}^n \rightarrow \mathbb{N}$ konservativ, falls für $\vec{m} \in Db(\varphi)$

$$P(\dots v \underset{\uparrow}{b \vec{m} b} \dots) = (\dots v \underset{\uparrow}{b \vec{m} b} \varphi(\vec{m}) b \dots)$$

und für $\vec{m} \notin Db(\varphi)$

$$P(\dots v \underset{\uparrow}{b \vec{m} b} \dots) \uparrow.$$

Man beachte, dass wir hier die Ausgabefunktion abweichend definieren: Am Ende der Rechnung steht der Kopf vor der ersten Eingabe, nicht vor der Ausgabe. Darüber hinaus löscht ein konservativer Operator den benutzten Speicher vor Abschluss der Rechnung, d.h. der rechte Bandteil enthält nur die Eingaben und die Ausgabe. Mit $F_{kon}(\text{TO})$ bezeichnen wir die Klasse der von TOs konservativ berechneten partiellen Funktionen. Diese stimmt mit der Klasse $F(\text{TO})$ überein. Wir zeigen hier zunächst nur die einfachere Richtung:

5.8 LEMMA. $F_{kon}(\text{TO}) \subseteq F(\text{TO})$.

BEWEIS. Sei P ein TO, der die n -stellige partielle Funktion φ konservativ berechnet. P unterscheidet sich dann von einer konventionellen Berechnung von φ nur dadurch, dass der Kopf nach Ausführung von P vor der 1. Eingabe und nicht vor der Ausgabe steht. Man muss nach Ausführung von P also lediglich den Kopf hinter die Eingaben bewegen, um einen TO P' zur konventionellen Berechnung von φ zu erhalten, also z.B. $P' = PR_{bb}LL_b$. □