Turingmaschinen

Hans Jürgen Ohlbach

Keywords: Einfache Turingmaschine, Mehrbandmaschine, universelle Turingmaschine

Empfohlene Vorkenntnisse: Endliche Automaten

Inhaltsverzeichnis

1 Einführung **2**

2 Informelle Beschreibung **2**

3 Formale Definition **4**

4 Mehrband-Turingmaschine **6**

5 Höhere Programmierkonstrukte **8**

5.1 Hintereinander-Ausführung 8

5.2 if . . . then . . . else 8

5.3 While 9

6 Universelle Turingmaschine **9**

1 Einführung

Turingmaschinen wurden 1936 von dem englischen Mathematiker Alan Turing erfunden, als extrem einfaches Modell, mit dem man Begriffe wie *Algorithmus* oder *Berechenbarkeit* mathematisch exakt erfassen kann, und dann auch darüber Beweise führen kann. Heutige Computer sind zwar viel schneller, und auch viel einfache zu programmieren als Turingmaschinen. Es hat sich aber gezeigt, dass man das was man mit heutigen Computern berechnen kann, im Prinzip auch mit Turingmaschinen berechenbar ist. Daher lassen sich viele Resultate über Turingmaschinen auch auf heutige Computer übertragen.

In diesem Text wird nur das Konzept der Turingmaschinen selbst eingeführt. Was man damit anfangen kann, wird in den jeweilig anderen Texten angesprochen.

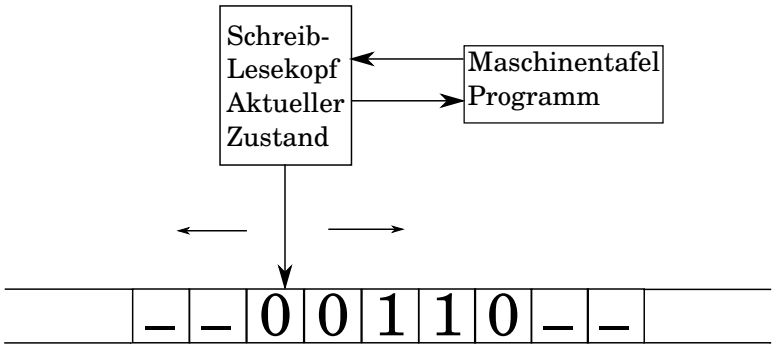
2 Informelle Beschreibung

Eine Turingmaschine kann man als Verallgemeinerung von endlichen Automaten und Kellerautomaten sehen. Endliche Automaten haben keinen Speicher, Kellerautomaten sind endliche Automaten mit einem Stack als Speicher, und Turingmaschinen haben statt dem Stack ein beliebig großes, beliebig beschreibbares Band.

Das Band besteht aus einzelnen Speicherzellen, in denen jeweils ein Buchstabe eines Alphabets stehen kann. Diese Zellen können von einem Schreib-Lesekopf gelesen und beschreiben werden. Der Schreib-Lesekopf kann jedoch nicht beliebige Zellen ansteuern, sondern von der aktuellen Zelle nur zur linken oder rechten Nachbarzelle gehen. Darin unterscheidet sich das Band von den Hauptspeichern moderner Rechner, deren Zellen über ihre Adressen beliebig angesprochen werden können.

Das Band wird oft als *unendlich groß* definiert. Das ist nicht wörtlich zu verstehen. Es bedeutet nur, dass es *beliebig groß* ist, d.h. es gibt keine endliche Grenze. Egal wieviel Platz ein Algorithmus braucht, es ist immer genügend viel davon da. Zu jedem endlichen Zeitpunkt sind aber nur endlich viele Zellen beschrieben.

Schematisch sieht das folgendermaßen aus:



Zweiseitiges unendliches Band

Wie ein endlicher Automat hat die Turingmaschine Zustände und Übergänge. Die Übergänge hängen

ab vom aktuellen Zustand und vom Buchstaben, der gerade gelesen wird. Dann kann die Maschine in einen neuen Zustand übergehen, den gerade gelesen Buchstaben überschreiben, und dann den Schreib-Lesekopf nach links oder rechts bewegen, oder stehen lassen.

Indem man diese Übergänge spezifiziert, programmiert man die Turingmaschine.

Ein erstes Beispiel zeigt die Arbeitsweise. Die Turingmaschine soll das Einerkomplement einer binären Zahl berechnen. Dafür muss einfach nur 0 mit 1 vertauscht werden, z.B.
$Einerkomplement(00110) = 11001$.

Wir nehmen an, die Zahl steht auf dem Band, und der Schreib-Lesekopf steht über dem linkesten Bit (so wie in dem Bild oben). Alle Zellen vor und hinter der Zahl sind leer, was man meist durch ein Blank _ kennzeichnet. Die Maschine muss jetzt nur testen, ob unter dem Schreib-Lesekopf eine 0 steht, dann eine 1 schreiben, und eine Zelle nach rechts gehen. Wenn stattdessen eine 1 steht, muss sie eine 0 schreiben, und eine Zelle nach rechts gehen. Das Verfahren terminiert wenn eine leere Zelle erreicht wird.

Wir schreiben das folgendermaßen auf:

$$\begin{array}{lrcl} 1. & (z_0, 0) & \mapsto & (z_0, 1, R) \\ 2. & (z_0, 1) & \mapsto & (z_0, 0, R) \\ 3. & (z_0, \_) & \mapsto & (z_e, \_, N) \end{array}$$

In diesem Beispiel gibt es zwei Zustände für die Maschine, den Startzustand z_0 und den Endzustand z_e.

Regel 1 besagt: Wenn im Zustand z_0 eine 0 gelesen wird, dann bleibe im Zustand z, ersetze die 0 durch eine 1, und gehe eine Zelle nach rechts (R).

Regel 2 besagt: Wenn im Zustand z_0 eine 1 gelesen wird, dann bleibe im Zustand z, ersetze die 1 durch eine 0, und gehe eine Zelle nach rechts (R).

Regel 2 besagt: Wenn im Zustand z_0 ein Blank _ gelesen wird (das steht für die leere Zelle), dann gehe in den Endzustand z_e, und bleibe an der Zelle stehen (N). Das Blank wird nicht verändert.

Für die Zahl 01100 würde die Maschine jetzt folgende Schritte machen:

$\overset{\downarrow}{0}\ 1\ 1\ 0\ 0$	$1\ \overset{\downarrow}{1}\ 1\ 0\ 0$	$1\ 0\ \overset{\downarrow}{1}\ 0\ 0$	$1\ 0\ 0\ \overset{\downarrow}{0}\ 0$	$1\ 0\ 0\ 1\ \overset{\downarrow}{0}$	$1\ 0\ 0\ 1\ 1\ \overset{\downarrow}{\_}$

Am Ende steht der Schreib-Lesekopf über dem ersten Blank, und die Maschine hält an. Oft fordert man jedoch, dass der Schreib-Lesekopf zum Schluss wieder über dem linkesten Bit steht. Im Beispiel müsste er dann wieder ganz nach links fahren, ohne etwas zu verändern.

3 Formale Definition

Mathematisch kann man Turingmaschinen folgendermaßen definieren:

Definition 1 (Turingmaschine)
Eine Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \_, E)$ besteht aus

- *einer Menge Z von Zuständen,*
- *dem Eingabealphabet Σ,*
- *dem Arbeitsalphabet $\Gamma \supseteq \Sigma$*
  (es können also mehr Buchstaben benutzt werden, als in der Eingabe)
- *der Überführungsfunktion δ: $Z \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$*
- *einem Startzustand z_0,*
- *dem Blank $\_$, und*
- *einer Menge E von Endzuständen.*

Die Überführungsfunkion ist folgendermaßen zu verstehen: Eine Regel $(z, a) \mapsto (z', b, X)$ bedeutet folgende *Aktion*: Wenn im Zustand z der Schreib-Lesekopf ein a liest, dann gehe in den Zustand z', ersetze das a durch b und gehe entweder nach links $(X = L)$, oder nach rechts $(X = R)$ oder bleibe stehen $(X = N)$.

Die Überführungsfunktion kann *deterministisch* oder *nichtdeterministisch* sein. Wenn sie deterministisch ist, dann gibt es für jeden Zustand und jeden gelesenen Buchstaben genau eine Aktion der Maschine. Wenn sie nichtdeterministisch ist, dann kann es für einen Zustand und einen gelesenen Buchstaben mehrere alternative Aktionen geben.

Die Turingmaschine startet im Zustand z_0 mit der Eingabe auf dem Band. Alle Zellen links und rechts von der Eingabe sind leer, bzw. mit Blanks gefüllt. Der Schreib-Lesekopf steht auf dem linkesten Zeichen der Eingabe.

Die Maschine liest dann das Zeichen unter dem Schreib-Lesekopf und führt dann entsprechend der Überführungsfunktion eine Aktion aus. Wenn es mehrere alternative Aktionen gibt, muss sie dann eine aussuchen. Sobald sie in einen Endzustand kommt, stoppt sie.

Wenn eine nichtdeterministische Maschine realistisch arbeiten sollte, und ein reales Problem lösen sollte, dann müsste sie bei mehreren alternativen Aktionen systematisch eine nach der anderen ausprobieren, und eventuell backtracken. Mit dieser Vorgehensweise würde sie dann deterministisch arbeiten. Auf diese Weise lassen sich nichtdeterministische Turingmaschinen immer deterministisch machen.

Definition 2 (Linear beschränkte Turingmaschine)
Eine Turingmaschine heißt linear beschränkt *falls sie während der Arbeit nicht mehr Platz auf dem Band benötigt als die Eingabe. (Man könnte auch einen festen „Faktor x * Platz der Eingabe“ zulassen.)*

Linear beschränkte Turingmaschinen können natürlich auch nichtdeterministisch sein. Es ist aber noch ein ungelöstes Problem, ob man eine nichtdeterministische linear beschränkte Turingmaschine deterministisch machen kann, und sie dann immer noch linear beschränkt ist (erstes LBA-Problem). Eine Turingmaschine mit unbeschränktem Band hat immer genügend Platz, um sich für das Backtracking die noch zu probierenden Alternativen zu merken. Genau dieser Platz fehlt bei der linear beschränkten Turingmaschine. Es ist aber noch unklar, ob man auch ohne diesen Platz auskommen kann.

Als nächstes Beispiel programmieren wir eine Turingmaschine, die zu einer Binärzahl eine 1 addiert. Also $110+1 = 111$ oder $111+1 = 1000$. Die Maschine muss zunächst ganz nach rechts laufen. Dann läuft sie von rechts nach links, addiert eine 1, und muss sich den Übertrag merken. Dazu benötige man zunächst einen Startzustand z_r, in dem die Maschine soweit nach rechts läuft, bis sie das erste Blank findet. Dann geht sie einen Schritt nach links und kommt in den Zustand z_1 (für 1 addieren). Steht da eine 0, dann macht sie daraus eine 1 und geht in den Zustand z_0 (für 0 addieren). Trifft sie das Blank, dann wird da eine 1 hingeschrieben, und sie geht in den Endzustand z_e. Im Zustand z_0 wird einfach nur weiter nach links gelaufen bis zum Blank. Dann geht sie wieder einen Schritt nach rechts, und bleibt stehen.

Die Überführungsfunktion sieht dann so aus.

$$
\begin{array}{lrcl}
1. & (z_r, 0) & \mapsto & (z_r, 0, R) \\
2. & (z_r, 1) & \mapsto & (z_r, 1, R) \\
3. & (z_r, \_) & \mapsto & (z_1, \_, L) \\
4. & (z_1, 1) & \mapsto & (z_1, 0, L) \\
5. & (z_1, 0) & \mapsto & (z_0, 1, L) \\
6. & (z_1, \_) & \mapsto & (z_e, 1, N) \\
7. & (z_0, 1) & \mapsto & (z_0, 1, L) \\
8. & (z_0, 0) & \mapsto & (z_0, 0, L) \\
9. & (z_0, \_) & \mapsto & (z_e, \_, R)
\end{array}
$$

Für die Zahl 101 ergibt sich folgender Durchlauf durch die Maschine:

$\overset{\downarrow}{1}\ 0\ 1$	$1\ \overset{\downarrow}{0}\ 1$	$1\ 0\ \overset{\downarrow}{1}$	$1\ 0\ 1\ \overset{\downarrow}{\_}$	$1\ 0\ \overset{\downarrow}{1}$	$1\ \overset{\downarrow}{0}\ 0$	$\overset{\downarrow}{1}\ 1\ 0$	$\overset{\downarrow}{\_}\ 1\ 1\ 0$	$\overset{\downarrow}{1}\ 1\ 0$
$z_r : 2$	$z_r : 1$	$z_r : 2$	$z_r : 3$	$z_1 : 4$	$z_1 : 5$	$z_0 : 7$	$z_0 : 9$	z_e

In der unteren Zeile steht immer der aktuelle Zustand, und die als nächstes anzuwendende Regel.

Sobald man eine Operation $x + 1$ programmieren kann, kann man auch eine Funktion $x + y$ programmieren, indem man die $+1$-Operation y mal durchführt. Danach kann man eine Operation $x * y$ programmieren, indem man die $+$-Operation y mal durchführt, usw. Damit lassen sich beliebige arithmetische Funktionen auf *natürlichen Zahlen* programmieren.

Definition 3 (Turing-Berechenbarkeit) *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$ *ist* Turing-berechenbar *falls es eine* deterministische *Turingmaschine gibt, die folgendes tut:*

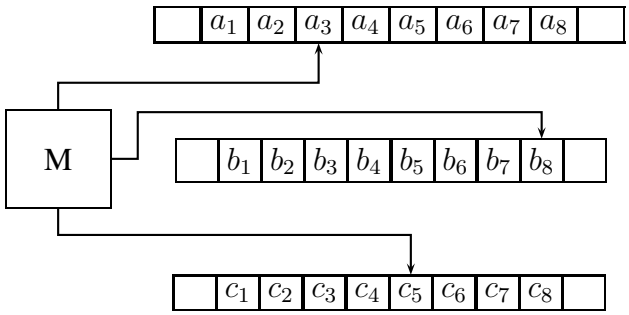
- *die Eingabewerte* $n_1, \ldots, n_k$ *stehen nacheinander in Binärdarstellung auf dem Band.*
- *Der Schreib-Lesekopf steht auf dem linkesten Bit.*
- *Die Turingmaschine hält irgendwann, und das Funktionsergebnis* $f(n_1, \ldots, n_k)$ *steht binär auf dem Band.*
- *Der Schreib-Lesekopf steht wieder auf dem linkesten Bit.*

Partialiät *einer Funktion äußert sich so, dass für die Eingaben, für die die Funktion nicht definiert ist, die Turingmaschine in eine Endlosschleife gerät.*

4 Mehrband-Turingmaschine

In den heutigen Prozessoren hat man die physikalisch möglichen Beschleunigungen soweit ausgereizt, dass man mehr Rechengeschwindigkeit fast nur noch durch Parallelverarbeitung bekommen kann. Parallelität kann man in Turingmaschinen durch Mehrband-Turingmaschinen modellieren. Damit kann man die Frage beantworten, ob Parallelität die Berechnungskraft signifikant erhöht.

Eine Mehrband-Turingmaschine hat mehrere, aber endlich viele, Bänder. Jedes Band hat einen eigenen Schreib-Lesekopf, so wie in dem Bild unten.



Für die Maschine gibt es aber *einen globalen Zustand*, der die Operationen in den verschiedenen Bändern steuert. Hätte jedes Band seinen eignen Zustand, dann gäbe es keine Kommunikation zwischen den Bändern. Die Mehrband Maschine würde dann wie mehrere unabhängige Maschinen arbeiten.

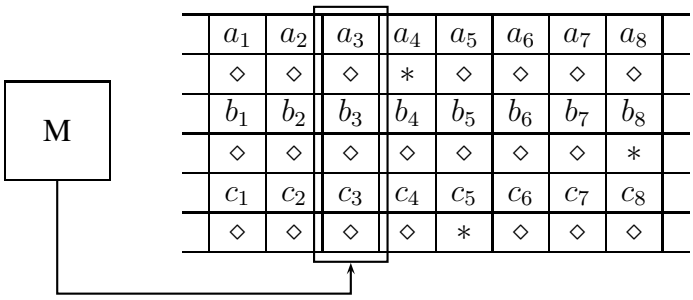
Die Regeln der Überführungsfunktion sähen dann etwas so aus:

$$(z, a_1, \ldots, a_n) \mapsto (z', b_1, X_1, \ldots . b_n, X_n)$$

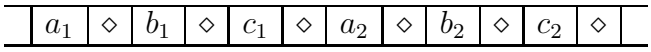
mit der Bedeutung: wenn die Maschine im Zustand z' ist, und im Band 1 a_1 gelesen wird, ..., und im Band n a_n gelesen wird, dann gehe in den Zustand z', schreibe auf Band 1 b_1 und bewege den Schreib-Lesekopf von Band 1 nach X_1 (R,L,N) ... und bewege den Schreib-Lesekopf von Band n nach X_n (R,L,N).

Von modernen Prozessoren kennt man es, dass Parallelität mit seriellen Prozessoren simuliert werden kann, indem die parallelen Prozesse abwechselnd für eine gewisse Zeit im seriellen Prozesser arbeiten dürfen, und dann wieder warten müssen, bis sie wieder dran sind. Daher ist es nicht überraschend, dass man auch Mehrband-Turingmaschinen durch eine Einband-Turingmaschine simuliert werden kann.

Das folgende Bild zeigt die Idee:



An jeder Bandposition kann man einen vertikalen Schnitt machen. In dem Schnitt sind die jeweilgen Zelleninhalte und die Hilfszeichen zusehen. Diese vertikalen Schnitte stellt man sich jetzt um 90 Grad gekippt vor, und alle hintereinander liegend. Jetzt hat man ein einziges Band.



Die Einband-Maschine, die eine n-Band-Maschine simuliert, muss jetzt folgendermaßen arbeiten: Der Lesekopf läuft soweit nach rechts, bis er n mal auf einen Stern getroffen ist. Dann liegt fest, wo die Leseköpfe der n-Band-Maschine wären. In der Zelle vor den Sternen kann die Einband-Maschine erkennen, was die Mehrbandmaschine gelesen hätte. Dann läuft die Einband-Maschine wieder an den Anfang zurück. Jetzt kann sie die Aktionen simulieren, die die Mehrbandmaschine machen würde, indem sie wieder zu den Zellen mit den Sternen läuft, dort den Zellinhalt vor dem Stern verändert, und dann auch den Stern verschiebt, falls nötig.

Die Konsequenz ist, dass die Mehrbandmaschine auch nicht mehr berechnen kann, als die Einband-Maschine. Sie kann aber etwas schneller rechnen. Für jeden Rechenschritt der Mehrband-Maschine muss die Einband-Maschine im schlimmsten Fall viermal durch den ganzen Bandinhalt der Einband-Maschine laufen. Die Größe des Bandinhalts der Einband-Maschine ergibt sich aus zweimal Anzahl n der Bänder der Mehrband-Maschine mal die Länge m des jeweiligen Bandinhalts, d.h. $\mathcal{O}(n * m)$.

Übertragen auf unsere heutigen Parallelprozessoren heißt das, dass diese auch nicht mehr Berechnungskraft haben als die seriellen Prozessoren, sind sind nur etwas (oder auch sehr viel) schneller.

Für manche theoretischen Untersuchungen sind Mehrband-Turingmaschinen allerdings praktischer als Einband-Turingmaschinen.

5 Höhere Programmierkonstrukte

Moderne Programmiersprachen haben eine Reihe von Programmierkonstrukten, die die Programmierung sehr erleichtern. Wir zeigen, dass man diese auch mit Turingmaschinen realisieren kann. Damit könnte man eine Art höhere Turing-Programmiersprache definieren.

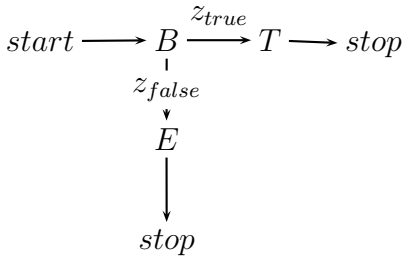
5.1 Hintereinander-Ausführung

Alle guten Programmiersprachen können Anweisungen hintereinander ausführen. In Java z.B. trennt man die Anweisungen durch ; und schreibt dann z.B. A1;A2; In analoger Weise kann man Turingmaschinen hintereinander schalten: M1;M2; Die Maschine M1 verarbeitet die Eingabe, schreibt das Ergebnis auf das Band und fährt den Schreib-Lesekopf wieder nach ganz links. Dann überführt man den Endzustand von M1 in den Anfangszustand von M2, und M2 arbeitet auf dem Ergebnis von M1 weiter. Als Beispiel, wenn M1 zu einer Binärzahl 1 addiert, dann kann man M1 n mal hintereinander schalten und erhält eine Maschine, die n addiert:

$$Band + n = \underbrace{Band + 1; Band + 1; ...; Band + 1}_{n \text{ mal}}$$

5.2 if . . . then . . . else . . .

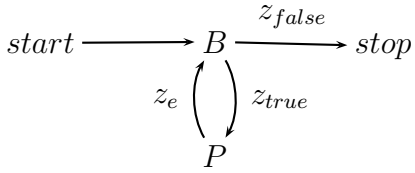
Ohne dieses Konstrukt kommt kaum eine Programmiersprache aus. Wir simulieren das durch drei Maschinen, die Bedingungsmaschine B, die Then-Maschine T und die Else-Maschine E. Diese werden folgendermaßen miteinander verbunden:



Die Eingabe steht wie immer auf dem Band. Zuerst wird die Bedingungsmaschine B ausgeführt. Diese endet entweder im Endszustand z_{true} oder z_{false}. Der Zustand z_{true} wird in den Startzustand von T überführt, und der Zustand z_{false} in den Startzustand von E.

5.3 While

Genauso wichtig wie ein if-Konstrukt ist ein while-Konstrukt. Dafür brauchen wir wieder eine Bedingungsmaschine B und eine Programm-Maschine P. Um ein whileBP zu realisieren, verschalten wir sie wie folgt.



Der Endzustand z_{false} der B-Maschine hält die Schleife an. Der Endzustand z_{true} wird in den Startzustand der P-Maschine überführt, und der Endzustand z_e der P-Maschine wird in den Startzustand der B-Maschine überführt.

Diese Beispiele verdeutlichen, dass Turing-Programmierung gar nicht so weit weg ist von der heutigen Art der Programmierung. Man bräuchte nur einen entsprechenden Compiler.

Es bleibt aber noch ein nicht so ganz trivialer Unterschied zwischen der Turingmaschine und den heutigen Computern. In diesen ist der Speicher frei adressierbar. Der Zugriff auf eine Speicherzelle ist damit in konstanter Zeit möglich. Das Band der Turingmaschine kann dagegen nur sequenziell abgelaufen werden. Für konkrete Algorithmen kann das einen linearen ($\mathcal{O}(n)$) Mehraufwand im Vergleich zum frei adressierbaren Speicher bedeuten.

6 Universelle Turingmaschine

Heute ist keine aufregende Sache mehr, dass jeder moderne Prozessor jeden anderen simulieren kann. Z.B. ist die Java Virtual Machine ein Simulator für eine Java-Maschine. Diesen Simulator gibt es für fast jeden Prozessortyp.

Diese Idee geht letztlich zurück auf Alan Turnings Idee einer universellen Turingmaschine. Er zeigte, dass man eine Turingmaschine U so programmieren kann, dass sie ein Beschreibung einer beliebig anderen Turingmaschine M vom Band lesen kann, zusätzlich eine Eingabe für diese andere Maschine, und dann die Aktionen von M auf dieser Eingabe simulieren kann, so dass am Ende auf dem Band das Ergebnis von M steht.

Die Maschine U ist also ein Turingmaschinen-Simulator.

Diese Erkenntnis hilft beim Beweis von sehr tiefgehenden Ergebnissen, u.a. der Unentscheidbarkeit des Halteproblems: Man kann kein Programm schreiben, welches andere Programme auf Endlosschleifen testet, und immer das richtige Ergebnis liefert.