2 Berechenbarkeit

Dieses Kapitel entspricht im Wesentlichen dem Kapitel 2 (*Berechenbarkeitstheorie*) in [9].

Jeder, der programmieren kann, weiß, dass es so etwas wie einen *intuitiven Berechenbarkeitsbegriff* gibt. Man hat eine Vorstellung davon, welche Funktionen *berechenbar* sind.

Will man allerdings zeigen, dass gewisse Funktionen *nicht* berechenbar sind, reicht dieser intuitive Berechenbarkeitsbegriff nicht aus. Es ist deshalb notwendig, den intuitiven Begriff der Berechenbarkeit formal sauber, d. h. mathematisch korrekt zu definieren.

Es tritt jedoch das Problem auf, dass man zwar einen formalen Berechenbarkeitsbegriff hat, allerdings wiederum nicht zeigen kann, dass er genau dem intuitiven Begriff entspricht. Deshalb werden wir verschiedene Berechenbarkeitsbegriffe definieren (siehe Kapitel 2.2 und 2.3). Letztlich werden wir zeigen, dass alle diese verschiedenen Berechenbarkeitsbegriffe äquivalent sind und somit annnehmen, dass wir wahrscheinlich den intuitiven Begriff getroffen haben (siehe Kapitel 2.4). Beweisen können wir es aber nach wie vor *nicht*!

Wir beginnen mit der Diskussion des intuitiven Berechenbarkeitsbegriffes.

2.1 Intuitiver Berechenbarkeitsbegriff

Eine (evtl. partielle Funktion) $f : \mathbb{N}^k \to \mathbb{N}$ soll als *intuitiv berechenbar* angesehen werden, falls es ein Rechenverfahren, einen *Algorithmus*[2] gibt, das/der f *berechnet*, d. h. gestartet mit $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$ als Eingabe soll der Algorithmus nach endlich vielen Schritten mit der Ausgabe $f(n_1, n_2, \ldots, n_k) \in \mathbb{N}$ stoppen.

Im Falle einer partiellen Funktion (also einer Funktion, die an manchen Stellen undefiniert sein kann) soll der Algorithmus bei der entsprechenden Eingabe nicht stoppen (z. B. durch eine unendliche Schleife).

Jedem Algorithmus wird also eine Funktion, die durch ihn berechnet wird, zugeordnet.

Beispiel 2.1 Der Algorithmus

```
INPUT(n);
REPEAT UNTIL FALSE
```

„berechnet“ die total undefinierte Funktion $\Omega\colon \mathbb{N} \to \mathbb{N}$ vermöge $n \mapsto f(n) = \textit{nicht definiert}$.

Beispiel 2.2 Die Funktion $f\colon \mathbb{N} \to \mathbb{N}$, definiert durch

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

(zum Beispiel $f(314) = 1$, $f(314158) = 0$, $f(31415926535897932384626433832795028841) = 1$) ist berechenbar, denn es gibt Näherungsverfahren für die Zahl π, mit denen man π beliebig genau bestimmen kann (auch wenn es für sehr große n sehr lange dauern kann, bis man $f(n)$ kennt).

Beispiel 2.3 Die Funktion $g\colon \mathbb{N} \to \mathbb{N}$, definiert durch

$$g(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der Dezimalbruchentwicklung von } \pi \text{ vorkommt,} \\ 0 & \text{sonst} \end{cases}$$

ist möglicherweise nicht berechenbar. Wir können zwar zum Beispiel $g(141) = 1$ und $g(3589) = 1$ angeben. Falls aber ein n in der bis heute bekannten Dezimaldarstellung von π nicht vorkommt, können wir momentan keine Aussage treffen, da wir ja nicht wissen, wo in der Dezimaldarstellung n auftaucht.

Unser bisheriges Wissen über die Zahl π reicht nicht aus, um eine Entscheidung über Berechenbarkeit oder Nicht-Berechenbarkeit zu treffen.

[2] Benannt nach Mohamed Ibn Musa Alchwarizmi, geboren in Chiwa (Chorism), gestorben in Bagdad nach 846.

Berechenbarkeit ist also eng mit dem Begriff eines *Algorithmus* verbunden. Intuitiv stellen wir dabei folgende Forderungen an einen Algorithmus.

Ein Algorithmus

- überführt Eingabedaten in Ausgabedaten (wobei die Art der Daten vom Problem, das durch den Algorithmus gelöst werden soll, abhängig ist),
- besteht aus einer endlichen Folge von Anweisungen mit folgenden Eigenschaften:
  - es gibt eine eindeutig festgelegte Anweisung, die als erste auszuführen ist,
  - nach Abarbeitung einer Anweisung gibt es eine eindeutig festgelegte Anweisung, die als nächste abzuarbeiten ist, oder die Abarbeitung des Algorithmus ist beendet und hat eindeutig bestimmte Ausgabedaten geliefert,
  - die Abarbeitung einer Anweisung erfordert keine Intelligenz (ist also prinzipiell durch eine Maschine realisierbar).

Mit diesem intuitiven Konzept lässt sich leicht feststellen, ob ein Verfahren ein Algorithmus ist. Betrachten wir als Beispiel die schriftliche Addition. Als Eingabe fungieren die beiden gegebenen zu addierenden Zahlen; das Ergebnis der Addition liefert die Ausgabe. Der Algorithmus besteht im Wesentlichen aus der sukzessiven Addition der entsprechenden Ziffern unter Beachtung des jeweils entstehenden Übertrags, wobei mit den „letzten“ Ziffern angefangen wird. Zur Ausführung der Addition von Ziffern ist keine Intelligenz notwendig (obwohl wir in der Praxis dabei das scheinbar Intelligenz erfordernde Kopfrechnen benutzen), da wir eine Tafel benutzen können, in der alle möglichen Additionen von Ziffern enthalten sind (und wir davon ausgehen, dass das Ablesen eines Resultats aus einer Tafel oder Liste ohne Intelligenz möglich ist). In ähnlicher Weise kann man leicht überprüfen, dass z. B.

- der Gaussssche[3] Algorithmus zur Lösung von linearen Gleichungssystemen (über den rationalen Zahlen),
- Kochrezepte (mit Zutaten und Kochgeräten als Eingabe und dem fertigen Gericht als Ausgabe),
- Bedienungsanweisungen für Geräte,
- PASCAL-Programme

Algorithmen sind.

Jedoch ist andererseits klar, dass dieser Algorithmenbegriff nicht ausreicht, um zu klären, ob es für ein Problem einen Algorithmus zur Lösung gibt. Falls man einen Algorithmus zur Lösung hat, so sind nur obige Kriterien zu testen. Um jedoch zu zeigen, dass es keinen Algorithmus gibt, ist es erforderlich, eine Kenntnis aller möglichen Algorithmen zu haben; und dafür ist der obige intuitive Begriff zu unpräzise. Folglich wird es unsere Aufgabe sein, eine Präzisierung des Algorithmenbegriffs vorzunehmen, die es gestattet, in korrekter Weise Beweise führen zu können.

2.2 Turing-Berechenbarkeit

Wir beginnen mit Turings[4] Vorschlag zu einer formalen Definition des Berechenbarkeitsbegriffs, der auf der nach ihm benannten Turingmaschine basiert. Er ging hierbei von der Idee aus nachzuempfinden, wie ein Mensch eine systematische Berechnung, wie etwa eine schriftliche Multiplikation nach der Schulmethode, durchführt. Er verwendet hierzu ein Rechenblatt – in Felder eingeteilt – auf dem die Rechnung, samt aller Zwischenergebnisse, notiert wird. Zur Verfügung stehen ihm hierbei ein Schreibwerkzeug und eventuell ein Radierer, um Zeichen auf Felder zu notieren bzw. wieder zu löschen.

[3] Carl Friedrich Gauss, 1777–1855, deutscher Mathematiker.

[4] Alan Mathison Turing (1912–1954), britischer Mathematiker, Logiker, Kryptoanalytiker und Computerkonstrukteur.

Die jeweilige Aktion hängt nur von wenigen (endlich vielen) Symbolen ab, die sich im Umfeld der aktuellen Position des Schreibwerkzeuges befinden. Die Rechnung wird hierbei gesteuert von einem endlichen Programm.

Die formale Definition der Turingmaschine ist eine gewisse Vereinfachung obiger Ideen.

- Das zweidimensionale Rechenblatt wird reduziert zu einem eindimensionalen, beidseitig (potentiell) unendlichen Band, das in Felder eingeteilt ist. Jedes Feld kann ein einzelnes Zeichen des sogenannten Bandalphabets der Maschine enthalten. Zum Bandalphabet gehört auch das Leer- oder Blankzeichen $\square$, das in allen Feldern steht, in denen kein anderes Zeichen des Bandalphabets steht.
- Das Schreibwerkzeug und der Radierer verschmelzen zu einem einzigen Schreib-Lesekopf. Dieser kann sich auf dem Band bewegen. Nur solche Zeichen, auf denen sich dieser Kopf gerade befindet, können in einem momentanen Rechenschritt gelesen und verändert werden. Der Kopf kann in einem Rechenschritt dann um maximal eine Position nach links oder rechts bewegt werden.
- Die Steuerung übernimmt die endliche Kontrolle.

In Abbildung 2.1 ist diese Modell veranschaulicht.

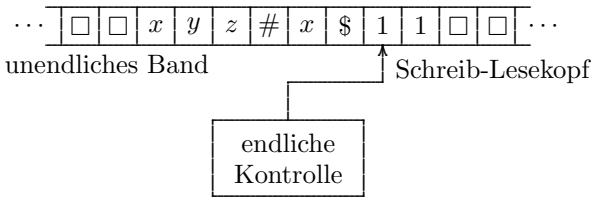

Abbildung 2.1: Veranschaulichung einer Turingmaschine

Formalisiert ergibt dieses Konzept folgende Definition.

Definition 2.4 *Eine (deterministische) Turingmaschine (kurz: TM) ist gegeben durch ein 7-Tupel*

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E).$$

Hierbei sind

- Z *eine endliche Menge (Zustandsmenge),*
- Σ *ein Alphabet (Eingabealphabet),*
- Γ *ein Alphabet (Bandalphabet) mit* $\Sigma \subseteq \Gamma$*,*
- $\delta\colon Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ *eine Funktion (Überführungsfunktion),*
- $z_0 \in Z$ *(Anfangszustand),*
- $\square \in \Gamma \setminus \Sigma$ *(Leerzeichen, Blank),*
- $E \subseteq Z$ *(Menge der Endzustände).*

Zur Interpretaion der Arbeitsweise der Turingmaschine: Falls

$$\delta(z, a) = (z', b, r)$$

gilt, bedeutet das: Wenn sich die TM M im Zustand z befindet und sie auf dem Band unter dem Schreib-Lesekopf das Zeichen a liest, so geht M in den Zustand z' über, schreibt auf das Band ein b (genau dorthin, wo der Kopf steht, sie also das a gelesen hat, d. h. a wird durch b überschrieben) und führt danach eine Kopfbewegung $r \in \{R, L, N\}$ aus. Hiebei bedeuten L: ein Schritt nach links, R: ein Schritt nach rechts, N: keine Kopfbewegung.

Die Turingmaschine beginnt ihre Arbeit, wenn das Eingabewort w auf dem Band steht, in allen anderen Zellen steht das Blankzeichen $\Box$, und sie sich im Anfangszustand z_0 befindet. Sie beendet ihre Arbeit, wenn sie einen Stopzustand, also einen Zustand aus E erreicht. Dabei wird vereinbart, dass auf dem Band dann das Ausgabewort steht, sonst nur Blankzeichen und der Kopf der Turingmaschine wiederum über dem ersten Symbol der Ausgabe steht.

Um die Arbeitsweise der Turingmaschine formalisiert darstellen zu können, benötigen wir eine Beschreibung der augenblicklichen Situation einer Turingmaschine, das geschieht mit der Konfiguration.

Definition 2.5 *Eine Konfiguration k einer Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ ist ein Wort $k \in \Gamma^* Z \Gamma^*$.*

Dabei soll $k = \alpha z \beta$ folgendermaßen interpretiert werden:

- $\alpha\beta$ steht auf dem Eingabeband, des weiteren stehen nur noch Blankzeichen $\Box$ auf dem Band,
- die Turingmaschine befindet sich im Zustand z und
- der Kopf der Turingmaschine befindet sich über dem ersten Symbol von β.

Eine Startkonfiguration ist somit $k_0 = z_0 w$ mit $w \in \Sigma^*$. Eine Endkonfiguration $k_e = \Box q w'$ mit $q \in E$, $w' \in \Sigma^*$.

Jetzt können wir die Arbeitsweise formalisieren.

Definition 2.6 *Wir definieren in der Menge der Konfigurationen einer Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ die binäre Relation „$\vdash$“ wie folgt. Es gilt*

$$a_1 \dots a_m z b_1 \dots b_n \vdash \begin{cases} a_1 \dots a_m z' c b_2 \dots b_n & \text{falls } \delta(z, b_1) = (z', c, N),\ m \geq 0,\ n \geq 1, \\ a_1 \dots a_m c z' b_2 \dots b_n & \text{falls } \delta(z, b_1) = (z', c, R),\ m \geq 0,\ n \geq 2, \\ a_1 \dots a_{m-1} z' a_m c b_2 \dots b_n & \text{falls } \delta(z, b_1) = (z', c, L),\ m \geq 1,\ n \geq 1, \end{cases}$$

$$a_1 \dots a_m z b_1 \vdash a_1 \dots a_m c z \Box \text{ falls } \delta(z, b_1) = (z', c, R),\ m \geq 0,$$

$$z b_1 \dots b_n \vdash z \Box c b_2 \dots b_n \text{ falls } \delta(z, b_1) = (z', c, L),\ n \geq 1.$$

Die Definition ist so gestaltet, dass die Konfigurationsbeschreibungen bei Bedarf verlängert werden, wenn die Maschine links oder rechts ein neues, bisher noch nicht besuchtes, Zeichen liest. Dieses kann natürlich nur das Blankzeichen $\Box$ sein.

Definition 2.7 *Mit $\vdash^*$ bezeichnen wir den reflexiven und transitiven Abschluss der binären Relation $\vdash$, also es gilt $k_0 \vdash^* k_e$ genau dann, wenn*

(i) *$k_0 = k_e$ ist, oder*

(ii) *eine Zahl $n \geq 1$ und Konfigurationen $k_1, k_2, \dots, k_n$ existieren, so dass*

$$k_0 \vdash k_1 \vdash k_2 \vdash \dots \vdash k_n \vdash k_e.$$

Betrachten wir ein Beispiel.

Beispiel 2.8 Gegeben sei die Turingmaschine

$$M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \Box\}, \delta, z_0, \Box, \{z_e\}),$$

wobei δ wie folgt definiert ist. Wir geben dabei δ in einer Tabelle an, wobei im Kreuzungspunkt der Zeile mit der Bezeichnung a und der Spalte mit der Bezeichnung z der Funktionswert $\delta(z, a)$

steht.

δ	z_0	z_1	z_2
$\square$	$(z_1, \square, L)$	$(z_e, 1, N)$	$(z_e, \square, R)$
0	$(z_0, 0, R)$	$(z_2, 1, L)$	$(z_2, 0, L)$
1	$(z_0, 1, R)$	$(z_1, 0, L)$	$(z_2, 1, L)$

Wenn diese Maschine mit der Eingabe 101 gestartet wird, so stoppt sie schließlich mit 110 auf dem Band, wobei der Schreib-Lesekopf wieder auf dem ersten Zeichen der Ausgabe steht.

Präzisiert, haben wir im Einzelnen folgende Überführungsschritte:

$$z_0101 \vdash 1z_001 \vdash 10z_01 \vdash 101z_0\square \vdash 10z_11\square \vdash 1z_100\square \vdash z_2110\square \vdash z_2\square110\square \vdash \square z_e110\square$$

Nachdem wir die Turingmaschine und ihre Arbeitsweise eingeführt und definiert haben, sind wir jetzt in der Lage, den Begriff der *Turingberechenbaren Funktion* zu definieren. Dazu wird zunächst der Begriff für Funktionen $f : \Sigma^* \to \Sigma^*$ eingeführt und danach für Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ für ein $k \in \mathbb{N}$ erweitert, wobei $\text{bin} : \mathbb{N} \to \{0,1\}^*$ eine Codierung der natürlichen Zahlen in der Menge $\{0,1\}^*$ darstellt, nämlich die übliche Binärdarstellung der natürlichen Zahlen, wobei wir allerdings keine führenden Nullen zulassen (die Zahl „0“ wird also durch das leere Wort dargestellt).

Definition 2.9 *Eine Funktion $f : \Sigma^* \to \Sigma^*$ heißt Turingberechenbar, falls es eine Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ gibt, so dass für alle $x, y \in \Sigma^*$ gilt:*

$$f(x) = y \quad \textit{genau dann, wenn} \quad z_0x \vdash^* \square \ldots \square z_e y \square \ldots \square \textit{ für } z_e \in E.$$

Definition 2.10 *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ für ein $k \in \mathbb{N}$ heißt Turingberechenbar, falls es eine Turingmaschine $M = (Z, \{0,1\}, \Gamma, \delta, z_0, \square, E)$ gibt, so dass für alle $n_1, n_2, \ldots, n_k, m \in \mathbb{N}$ gilt:*

$$f(n_1, n_2, \ldots, n_k) = m \quad \textit{genau dann, wenn}$$
$$z_0\text{bin}(n_1)\#\text{bin}(n_2)\#\ldots\#\text{bin}(n_k) \vdash^* \square \ldots \square z_e \text{bin}(m) \square \ldots \square \textit{ für } z_e \in E.$$

Man beachte, dass bei beiden Definitionen implizit ausgedrückt wird, dass im Falle $f(x) =$ *undefiniert* die Maschine M *keinen* Stopzustand erreicht, also zum Beispiel in eine unendliche Schleife gerät.

Beispiel 2.11 Die Nachfolgerfunktion $f : \mathbb{N} \to \mathbb{N}$ vermöge $n \mapsto f(n) = n + 1$ ist Turingberechenbar, da die Turingmaschine aus Beispiel 2.8 die Eingabe $\text{bin}(n)$ in die Ausgabe $\text{bin}(n + 1)$ transformiert.

Beispiel 2.12 Die für alle Wörter aus $\{a, b\}^*$ nicht-definierte Funktion $\Omega : \{a, b\}^* \to \{a, b\}^*$ vermöge $w \mapsto \Omega(w) =$ *nicht definiert* ist Turingberechenbar, da sie von der Turingmaschine $M = (\{z_0, z_e\}, \{a, b\}, \{a, b, \square\}, \delta, z_0, \square, \{z_e\})$ mit $\delta(z_0, x) = (z_0, x, N)$ für alle $x \in \{a, b, \square\}$ berechnet wird.

Eine *Mehrband-Turingmaschine* kann auf k Bändern, $k \geq 1$, unabhängig voneinander operieren, d. h. sie hat k Schreib-Leseköpfe, die in jedem Schritt lesen, schreiben und sich unabhängig voneinander bewegen können. Formal kann eine solche Maschine erfasst werden, indem wir δ als eine Funktion von $Z \times \Gamma^k$ in $Z \times \Gamma^k \times \{R, L, N\}^k$ ansetzen. Die Begriffe *Konfiguration*, *direkter Überführungsschritt* sowie *berechnete Funktion* können dementsprechend erweitert werden. Wir wollen es aber an dieser Stelle nicht tun, sondern es mit der informellen Darstellung belassen. In der Literatur, z. B. in [5] kann der interessierte Leser die Formalisierungen finden.

Wir wollen uns nun der Frage widmen, ob die Mehrband-Turingmaschine mehr *Berechnungskraft* besitzt als die einfache Turingmaschine, wobei wir den Beweis folgender Aussage nur ganz grob skizzieren und wieder auf die Literatur verweisen.

Satz 2.13 *Zu jeder Mehrband-Turingmaschine M gibt es eine (Einband-) Turingmaschine M', die dieselbe Funktion berechnet wie M.*

Beweis. Hier nur die Beweisidee:

Sei k die Anzahl der Bänder und Γ das Bandalphabet von M. Das Arbeitsband von M' soll dann $2k$ *Spuren* enthalten, k davon werden genutzt, um die Inhalte der k Bänder von M zu speichern und die restlichen k Bänder werden nur benutzt, um die jeweilige Kopfposition über dem simulierten Band zu markieren. M' simuliert dann einen Arbeitsschritt von M, indem sukzessive alle Inhalte der Zellen der Spuren gelesen werden, die durch die Kopfposition markiert sind, dann gemäß der Überführungsfunktion von M alle diese Inhalte wieder sukzessive neu beschrieben werden und anschließend gemäß der Überführungsfunktion wiederum nacheinander alle Marker für die Kopfpositionen verschoben werden. Genaueres lese der geneigte Leser bitte in der Literatur nach. □

Dieser Satz ermöglicht es uns, gewisse Funktionen durch Mehrband-Turingmaschinen berechnen zu lassen, was oftmals wesentlich „günstiger“ ist als die Nutzung von Einband-Turingmaschinen. Wir aber wissen, dass es dann natürlich auch Einband-Turingmaschinen gibt, die diese Funktionen realisieren.

Wir werden davon im folgenden Gebrauch machen und zur einfacheren Darstellung folgende Notationen einführen.

Notation 2.14 Wenn M eine 1-Band-Turingmaschine ist, so bezeichnet $M(i,k)$, $i \leq k$, diejenige k-Band-Turingmaschine, die wir aus M dadurch erhalten, dass die Aktionen von M auf dem Band i ablaufen und alle anderen Bänder unverändert bleiben. Falls die Gesamtzahl der Bänder k „genügend“ groß ist, also eigentlich keine Rolle spielt, so schreiben wir einfach $M(i)$ statt $M(i,k)$.

Notation 2.15 Die Turingmaschine aus Beispiel 2.8 (die zu einer Zahl 1 dazuaddiert) bezeichnen wir mit „Band := Band + 1“ und anstelle von „Band := Band + 1 (i)“ scheiben wir „Band i := Band $i+1$“. In ähnlicher Weise können wir auch Mehrband-Turingmaschinen erhalten, die die Operationen „Band i := Band $i \dot{-} 1$“, „Band $i := 0$“ sowie „Band i := Band j“ ausführen. Dabei bedeutet „ $\dot{-}$ “ die modifizierte Subtraktion $\dot{-} : \mathbb{N}^2 \to \mathbb{N}$ vermöge

$$(n_1, n_2) \mapsto n_1 \dot{-} n_2 = \begin{cases} n_1 - n_2 & \text{falls } n_1 \geq n_2, \\ 0 & \text{sonst.} \end{cases}$$

Als nächstes wollen wir Turingmaschinen „hintereinanderschalten“.

Notation 2.16 Seien M_1 und M_2 zwei Turingmaschinen, so wollen wir durch

$$\text{start} \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \text{stop}$$

oder auch durch

$$M_1; M_2$$

diejenige Turingmaschine verstehen, die zuerst wie die Turingmaschine M_1 arbeitet und wenn M_1 einen Stopzustand erreichen würde in den Anfangszustand von M_2 übergeht und jetzt wie die Turingmaschine M_2 arbeitet. Sie stoppt dann, wenn M_2 einen Stopzustand erreichen würde.

Beispiel 2.17 Betrachten wir das Diagramm im Bild 2.2. So erkennen wir, dass dort das schematische Flussbild einer Turingmaschine steht, welche dreimal nacheinander zur Zahl auf dem Band 1 addiert, also es sich um die Turingmaschine „Band := Band + 3“ handelt.

Notation 2.18 Analog soll die Turingmaschine im Bild 2.3 nach Simulation der Turingmaschine M die Turingmaschine M_1 simulieren, falls sie bei der Simulation von M im Zustand z_{e1} stoppt. Analog soll sie die Turingmaschine M_2 abarbeiten, falls sie bei der Simulation von M im Zustand z_{e2} stoppt.

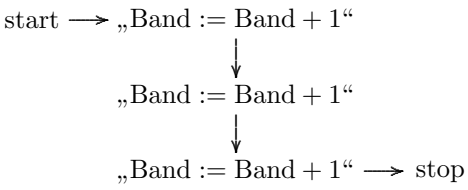

Abbildung 2.2: Die Turingmaschine „Band := Band + 3“

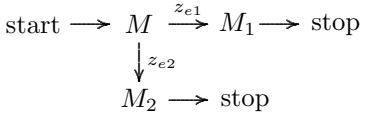

Abbildung 2.3: Eine sich verzweigende Turingmaschine

Betrachten wir in folgendem Beispiel noch eine spezielle Turingmaschine.

Beispiel 2.19 Es sei $M = (\{z_0, z_1, ja, nein\}, \Sigma, \Gamma, \delta, z_0, \square, \{ja, nein\})$ mit $0 \in \Sigma,\ 0, \square \in \Gamma$ sowie mit der Überführungsfunktion δ, gegeben durch

$$\begin{aligned}
\delta(z_0, a) &= (nein, a, N) \quad \text{für } a \neq 0,\\
\delta(z_0, 0) &= (z_1, 0, R),\\
\delta(z_1, a) &= (nein, a, L) \quad \text{für } a \neq \square,\\
\delta(z_1, \square) &= (ja, \square, L).
\end{aligned}$$

Diese Turingmaschine testet, ob die Eingabe genau das Wort 0 ist. Falls ja, stoppt sie im Zustand *ja*, falls nein, stoppt sie im Zustand *nein*. Wir wollen diese Turingmaschine mit „Band = 0 ?“ bezeichnen.

Anstatt von „Band = 0 ? (i)“ schreiben wir „Band $i = 0$?“

Beispiel 2.20 Betrachten wir noch ein weiteres Beispiel einer Turingmaschine. Sei M eine beliebige Turingmaschine, dann bezeichnen wir die Turingmaschine, die durch das Diagramm in der

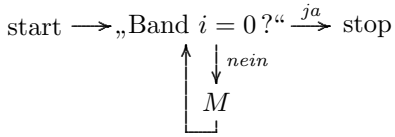

Abbildung 2.4: Die Turingmaschine „WHILE Band $i \neq 0$ DO M“

Abbildung 2.4 gegeben ist, mit „WHILE Band $i \neq 0$ DO M“. Die Arbeitsweise ist einfach zu erkennen.

Wie wir aus obigem Bespiel erkennen, können wir durch unsere eingeführten Bezeichnungen bereits verschiedene einfache Programmiersprachen-ähnliche Konzepte mit einer Mehrband-Turingmaschine simulieren. Dabei können die Bandinhalte als Variablenwerte angesehen werden. Es gibt einfache Wertzuweisungen, Hintereinanderreihung von Programmteilen ist möglich, und einfache Abfrage und WHILE-Schleifen können wir „programmieren“. Wir möchten daran erinnern, dass wir all dieses natürlich auch mit Einband-Turingmaschinen simulieren können.

2.3 LOOP-, WHILE- und GOTO-Berechenbarkeit

Nachdem wir im Kapitel 2.2 die Turingmaschine zur Annäherung an den Begriff *berechenbare Funktion* eingeführt haben, betrachten wir in diesem Kapitel einfache *Programmiersprachen.*

Zunächst führen wir LOOP-Programme in drei Stufen ein.

Zunächst werden die *syntaktischen Komponenten*, der *Zeichensatz* festgelegt, bevor die Syntax induktiv und schließlich die Semantik definiert wird.

Definition 2.21 LOOP-*Programme bestehen aus folgenden Zeichen* (*syntaktischen Komponenten*) :

- *Variablen:* $x_0 \quad x_1 \quad x_2 \quad \ldots$
- *Konstanten:* $0 \quad 1 \quad 2 \quad \ldots$
- *Operationssymbole:* $+ \quad -$
- *Trennsymbole:* $; \quad :=$
- *Schlüsselwörter:* LOOP DO END

Definition 2.22 *Die Syntax von* LOOP-*Programmen wird wie folgt induktiv definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c \qquad \textit{bzw.} \qquad x_i := x_j - c$$

ist ein LOOP-*Programm, wobei c eine Konstante ist.*

(ii) *Sind* P_1, P_2 LOOP-*Programme, dann sind auch*

$$P_1; P_2$$

sowie

$$\text{LOOP } x_i \text{ DO } P_1 \text{ END}$$

LOOP-*Programme.*

Definition 2.23 *Die Semantik von* LOOP-*Programmen ist wie folgt definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c$$

wird wie „üblich“ interpretiert: der neue Wert der Variablen x_i berechnet sich als Summe des Wertes der Variablen x_j und der Konstanten c, wobei der Wert in der Variablen x_j erhalten bleibt.
Die Wertzuweisung

$$x_i := x_j - c$$

wird analog interpretiert, wobei sich aber die Werte nach der sogenannten modifizierte Differenz „$\dot{-}$“, die wie folgt definiert ist

$$n_1 \dot{-} n_2 = \begin{cases} n_1 - n_2 \textit{ falls } n_1 \geq n_2, \\ 0 \qquad \textit{sonst,} \end{cases}$$

berechnen.

(ii) *Ein* LOOP*-Programm der Form* $P_1; P_2$ *soll die Hintereinanderausführung der Programme* P_1 *und* P_2 *bedeuten, also zuerst wird das Programm* P_1*, dann das Programm* P_2 *ausgeführt. Ein* LOOP*-Programm der Form* LOOP x_i DO P_1 END *bedeutet, dass das Programm* P_1 *sooft ausgeführt wird, wie der Wert der Variablen* x_i *zu Beginn angibt. Änderungen des Wertes der Variablen* x_i *haben also keinen Einfluss auf die Anzahl der Wiederholungen.*

Wir können jedem LOOP-Programm P eine Funktion zuordnen, nämlich die von diesem LOOP-Programm P *berechnete Funktion.* Präzisiert wird das mit dem Begriff der LOOP-berechenbaren Funktion.

Definition 2.24 *Eine Funktion* $f\colon \mathbb{N}^k \to \mathbb{N}$*,* $k \in \mathbb{N}$*, heißt* LOOP*-berechenbar, falls es ein* LOOP*-Programm* P *gibt, das* f *in dem Sinne berechnet, dass* P*, gestartet mit* $n_1, n_2, \ldots, n_k$ *in den Variablen* $x_1, x_2, \ldots, x_k$ *und* 0 *in den restlichen Variablen, mit dem Wert* $f(n_1, n_2, \ldots, n_k)$ *in der Variablen* x_0 *stoppt.*

Betrachten wir dazu einige Beispiele.

Beispiel 2.25 Gegeben sei das LOOP-Programm

$x_0 := x_1 + 0;$
LOOP x_2 DO $x_0 := x_0 + 1$ END

Man erkennt leicht, dass das Programm mit dem Wert der Summe der Anfangsbelegungen der Variablen x_1 und x_2 in der Variablen x_0 stoppt. Es berechnet also die Addition $+ : \mathbb{N}^2 \to \mathbb{N}$ vermöge $(x_1, x_2) \mapsto +(x_1, x_2) = x_1 + x_2$.

Also ist die Addition LOOP-berechenbar.

Bemerkung 2.26 Im obigen Beispiel 2.25 lautet die erste Programmzeile

$x_0 := x_1 + 0$

Normalerweise würde man dafür

$x_0 := x_1$

schreiben. Das wollen wir in Zukunft auch machen. D. h., wir werden die enge Definition der Wertzuweiseung in der Definition der Syntax von LOOP-Programmen etwas aufweiten, indem wir auch Wertzuweisungen der Form

$x_i := x_j$ und
$x_i := c$

zulassen wollen, da wir natürlich diese durch die Programme

$x_i := x_j + 0$

bzw.

$x_k = 0;$
$x_i := x_k + c$

simulieren können.

Beispiel 2.27 Gegeben sei das LOOP-Programm

LOOP x_2 DO
LOOP x_1 DO $x_0 := x_0 + 1$ END
END

Eine genaue Betrachtung des Programms zeigt, dass damit die Funktion $\cdot : \mathbb{N}^2 \to \mathbb{N}$ vermöge $(x_1, x_2) \mapsto \cdot(x_1, x_2) = x_1 \cdot x_2$, also die Multiplikation zweier Zahlen berechnet wird, die damit also LOOP-berechenbar ist.

Man beachte, dass die Anfangsbelegung der Variablen x_0 natürlich laut Definition 0 ist. Das wird hier gebraucht und verwendet.

Bemerkung 2.28 Wir haben es im Beispiel 2.27 mit zwei ineinander verschachtelten LOOP-Schleifen zu tun. Man erkennt, dass die innere Schleife eigentlich die Addition aus Beispiel 2.25 ist. Wir könnten also das Programm im Beispiel 2.27 im Prinzip auch als

LOOP x_2 DO $x_0 := x_0 + x_1$ END

schreiben, wobei das natürlich kein „LOOP-Programm“ im strengen Sinne der Definition ist. Da wir allerdings einmal nachgewiesen haben, dass die Addition „$x_0 := x_0 + x_1$“ durch ein LOOP-Programm berechnet werden kann, wollen wir im Folgenden solche Konstruktionen verwenden und sind uns dabei bewusst, dass wir eigentlich dafür die vollständigen LOOP-Programme benutzen müssten, wobei wir natürlich auf den korrekten Gebrauch der einzelnen Variablen achten müssen und uns immer bewusst sein müssen, dass es bei LOOP-Programmen ausschließlich „globale“ Variable gibt.

Die Syntax der LOOP-Programme ist sehr „kurz“, d. h., viele Kontrukte höherer Programmiersprachen wie z. B. IF-THEN-ELSE stehen nicht zur Verfügung. Ist das nun eine Einschränkung der LOOP-Programme?

Betrachten wir etwa die Anweisung

IF $x_1 = 0$ THEN A ELSE B END

wobei A und B LOOP-Programme sein sollen. Können wir dann diese Anweisung durch ein LOOP-Programm simulieren? Die Antwort gibt folgendes Beispiel.

Beispiel 2.29 Das Konstrukt

IF $x_1 = 0$ THEN A ELSE B END

wird durch das LOOP-Programm

$x_2 := 1;\ x_3 := 0;$
LOOP x_1 DO $x_2 := 0;\ x_3 := 1$ END;
LOOP x_2 DO A END;
LOOP x_3 DO B END

simuliert. Dabei sind die Variablen x_2 und x_3 natürlich nicht in den Programmen A und B enthalten.

Bemerkung 2.30 Wie wir eben gesehen haben, können wir IF-THEN-ELSE Anweisungen durch LOOP-Programme simulieren. Ist die Abfragebedingung komplizierter, so müssen wir natürlich die Simulation anpassen, aber man erkennt nach einiger Übung, dass man dieses entsprechend formulieren kann.

Deshalb werden wir auch bei der Angabe von speziellen LOOP-Programmen, die wir noch benutzen werden, solche IF-THEN-ELSE-Konstrukte verwenden, um Schreibarbeit zu sparen. Dabei haben wir natürlich das Wissen, dass wir daraus problemlos „reine“ LOOP-Programme konstruieren können.

Eine genaue Betrachtung der LOOP-Programme zeigt, dass bei der Abarbeitung jede LOOP-Schleife nur endlich oft durchlaufen werden kann. Insgesamt gibt es natürlich in einem LOOP-Programm auch nur endlich viele LOOP-Schleifen, also stoppt jedes LOOP-Programm. Das aber heißt, die von LOOP-Programmen berechneten Funktionen enthalten keine undefinierten Stellen. Also:

Lemma 2.31 *Jede von einem* LOOP*-Programm berechnete Funktion ist total.* □

Das aber wiederum hat natürlich folgende Konsequenz.

Folgerung 2.32 *Es gibt Funktionen, die nicht* LOOP*-berechenbar sind.* □

Etwas schwieriger ist folgendes Lemma zu zeigen, das wir hier deshalb auch nur nennen wollen.

Lemma 2.33 *Es gibt totale Funktionen, die nicht* LOOP*-berechenbar sind.* □

Dem interessierten Leser sei gesagt, dass zum Beispiel die sogenannte *Ackermannfunktion* eine solche ist. Einen Beweis dafür kann man zum Beispiel in [9] finden.

Wir wollen wegen oben genannter Unzulänglichkeiten der LOOP-Programme dieses Konzept erweitern, indem wir WHILE-Schleife einführen und so zu den sogenannten WHILE-Programmen kommen.

Wir geben hier noch einmal die vollständige Definition von WHILE-Programmen an, wobei es sich immer nur um kleinere Erweiterungen der LOOP-Programme handelt.

Definition 2.34 WHILE*-Programme bestehen aus folgenden Zeichen (syntaktischen Komponenten) :*

- *Variablen:* $x_0 \;\; x_1 \;\; x_2 \;\; \ldots$
- *Konstanten:* $0 \;\; 1 \;\; 2 \;\; \ldots$
- *Operationssymbole:* $+ \;\; -$
- *Trennsymbole:* $; \;\; := \;\; \neq$
- *Schlüsselwörter:* LOOP WHILE DO END

Definition 2.35 *Die Syntax von* WHILE*-Programmen wird wie folgt induktiv definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c \qquad \textit{bzw.} \qquad x_i := x_j - c$$

ist ein WHILE*-Programm, wobei c eine Konstante ist.*

(ii) *Sind* P_1, P_2 WHILE*-Programme, dann sind auch*

$$P_1; P_2$$

sowie

$$\text{LOOP } x_i \text{ DO } P_1 \text{ END}$$

und

$$\text{WHILE } x_i \neq 0 \text{ DO } P_1 \text{ END}$$

WHILE*-Programme.*

Definition 2.36 *Die Semantik von* WHILE*-Programmen ist wie folgt definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c$$

wird wie „üblich“ interpretiert: der neue Wert der Variablen x_i *berechnet sich als Summe des Wertes der Variablen* x_j *und der Konstanten c, wobei der Wert in der Variablen* x_j *erhalten bleibt.*

Die Wertzuweisung

$$x_i := x_j - c$$

wird analog interpretiert, wobei sich aber die Werte nach der sogenannten modifizierte Differenz „ $\dot{-}$ “, die wie folgt definiert ist

$$n_1 \dot{-} n_2 = \begin{cases} n_1 - n_2 & \textit{falls } n_1 \geq n_2, \\ 0 & \textit{sonst,} \end{cases}$$

berechnen.

(ii) *Ein* WHILE*-Programm der Form* $P_1; P_2$ *soll die Hintereinanderausführung der Programme* P_1 *und* P_2 *bedeuten, also zuerst wird das Programm* P_1*, dann das Programm* P_2 *ausgeführt.*
Ein WHILE*-Programm der Form* LOOP x_i DO P_1 END *bedeutet, dass das Programm* P_1 *sooft ausgeführt wird, wie der Wert der Variablen* x_i *zu Beginn angibt. Änderungen des Wertes der Variablen* x_i *haben also keinen Einfluss auf die Anzahl der Wiederholungen.*
Ein WHILE*-Programm der Form* WHILE $x_i \neq 0$ DO P_1 END *bedeutet, dass das Programm* P_1 *solange ausgeführt wird, wie der Wert der Variablen* x_i *ungleich Null ist. Es findet also vor jedem erneuten Durchlauf des Programms* P_1 *eine Abfrage der Variablen* x_1 *statt.*

Wir können wiederum jedem WHILE-Programm P eine Funktion zuordnen, nämlich die von diesem WHILE-Programm P *berechnete Funktion*. Präzisiert wird das mit dem Begriff der WHILE-berechenbaren Funktion.

Definition 2.37 *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$, $k \in \mathbb{N}$, *heißt* WHILE*-berechenbar, falls es ein* WHILE*-Programm* P *gibt, das* f *in dem Sinne berechnet, dass* P*, gestartet mit* $n_1, n_2, \ldots, n_k$ *in den Variablen* $x_1, x_2, \ldots, x_k$ *und* 0 *in den restlichen Variablen, mit dem Wert* $f(n_1, n_2, \ldots, n_k)$ *in der Variablen* x_0 *stoppt. Ist* $f(n_1, n_2, \ldots, n_k)$ *dagegen nicht definiert, so stoppt* P *nicht.*

Da die Definition der WHILE-Programme die LOOP-Programme auch enthalten, haben wir

Folgerung 2.38 *Jede* LOOP*-berechenbare Funktion ist* WHILE*-berechenbar.* □

Betrachten wir ein Beispiel.

Beispiel 2.39 Das WHILE-Programm

$x_3 := x_1 - 5$;
WHILE $x_3 \neq 0$ DO $x_1 := x_1 + 1$ END;
LOOP x_1 DO $x_0 := x_0 + 1$ END;
LOOP x_2 DO $x_0 := x_0 + 1$ END

berechnet die Funktion $f\colon \mathbb{N}^2 \to \mathbb{N}$ vermöge

$$f(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{falls } x_1 \leq 5, \\ \textit{nicht definiert} & \text{sonst.} \end{cases}$$

Aus dem Beispiel, das zeigt, dass WHILE-Programme auch echt partielle Funktionen berechnen können haben wir sofort:

Folgerung 2.40 *Es gibt* WHILE*-berechenbare Funktionen, die nicht* LOOP*-berechenbar sind.* □

Bemerkung 2.41 Sei LOOP x_i DO P END ein LOOP-Programm. Sei weiterhin y eine Variable, die in dem Programm P nicht vorkommt. Dann wird offensichtlich diese LOOP-Schleife durch das WHILE-Programm

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

simuliert. Also benötigen wir in der Programmiersprache WHILE die LOOP-Schleife eigentlich nicht mehr.

Im Kapitel 2.2 wurde angedeutet, dass Wertzuweisungen, Hintereinanderschalten von Turingmaschinen und WHILE-Schleifen auf einer Mehrband-Turingmaschine simulierbar sind. Hierbeir entspricht dem i-ten Band der Turingmaschine gerade die Variable x_1 des WHILE-Programms, wobei der Wert einer Variablen auf dem Band als Binärzahl dargestellt wird.

Schließlich kann noch jede Mehrband-Turingmaschine durch eine „normale“ Turingmaschine (also 1-Band-Turingmaschine) simuliert werden.

Damit erhalten wir:

Satz 2.42 *Jede* WHILE*-berechenbare Funktion ist Turingberechenbar.* □

Wir wollen jetzt versuchen, die Umkehrung zu zeigen, dabei ist es günstig, wenn wir noch einen Zwischenschritt einfügen, nämlich die GOTO-*Berechenbarkeit.*

Definition 2.43 GOTO*-Programme bestehen aus folgenden Zeichen (syntaktischen Komponenten)* :

- *Variablen:* x_0 x_1 x_2 ...
- *Konstanten:* 0 1 2 ...
- *Operationssymbole:* $+$ $-$
- *Trennsymbole:* : ; := =
- *Marken:* M_1 M_2 M_3 ...
- *Schlüsselwörter:* GOTO IF THEN HALT

Definition 2.44 *Die Syntax von* GOTO*-Programmen wird wie folgt definiert. Ein* GOTO*-Programm besteht aus einer endlichen Folge von Anweisungen A_i, die jeweils durch eine Marke M_i eingeleitet werden:*

$M_1 : A_1$;
$M_2 : A_2$;
$\vdots$
$M_k : A_k$

Dabei ist jede Anweisung A_i, $1 \leq i \leq k$ von folgender Form:

- Wertzuweisung: $x_i := x_j + c$ *oder* $x_i := x_j - c$*, wobei c eine Konstante ist,*
- unbedingter Sprung: GOTO M_i*,*
- bedingter Sprung: IF $x_i = c$ THEN GOTO M_i *oder*
- Stopanweisung: HALT.

Definition 2.45 *Die Semantik von* GOTO*-Programmen ist wie „üblich“ definiert. Lautet eine Programmzeile*

- M_i: $x_j := x_k + c$, *so wird die Wertanweisung* $x_j := x_k + c$ *gemäß der Semantik der* LOOP- *oder* WHILE-*Programme ausgeführt. Danach wird die nächste Programmzeile abgearbeitet.*
- M_i: GOTO M_j, *wird als nächste Programmzeile die mit der Marke* M_j *ausgeführt.*
- M_i: IF $x_j = c$ THEN GOTO M_k, *wird als nächste Programmzeile die mit der Marke* M_k *ausgeführt, falls der Wert der Variablen* x_j *gleich* c *ist, sonst wird die folgende Programmzeile abgearbeitet.*
- M_i: HALT, *stoppt das Programm.*

Die von GOTO-Programmen berechnete Funktionen werden analog den WHILE-Programmen definiert, dabei ist klar, dass GOTO-Programme auch in eine unendliche Schleife (M_i: GOTO M_i) geraten können.

Definition 2.46 *Eine Funktion* $f: \mathbb{N}^k \to \mathbb{N}$, $k \in \mathbb{N}$, *heißt* GOTO-*berechenbar, falls es ein* GOTO-*Programm* P *gibt, das* f *in dem Sinne berechnet, dass* P, *gestartet mit* $n_1, n_2, \ldots, n_k$ *in den Variablen* $x_1, x_2, \ldots, x_k$ *und* 0 *in den restlichen Variablen, mit dem Wert* $f(n_1, n_2, \ldots, n_k)$ *in der Variablen* x_0 *stoppt. Ist* $f(n_1, n_2, \ldots, n_k)$ *dagegen nicht definiert, so stoppt* P *nicht.*

Beispiel 2.47 Eine WHILE-Schleife

WHILE $x_i \neq 0$ DO P END

kann durch folgendes GOTO-Programm simuliert werden.

M_1: IF $x_i = 0$ THEN GOTO M_4;
M_2: P;
M_3: GOTO M_1;
M_4: ...

Bemerkung 2.48 Oft werden in GOTO-Programmen nur die Marken aufgeschrieben, die wirklich gebraucht werden, nämlich durch einen Sprung. Das GOTO-Programm im obigen Beispiel würde damit folgende Gestalt haben:

M_1: IF $x_i = 0$ THEN GOTO M_4;
P;
GOTO M_1;
M_4: ...

Mit dem Beispiel 2.47 erhalten wir sofort:

Satz 2.49 *Jede* WHILE-*berechenbare Funktion ist auch* GOTO-*berechenbar.* □

Wir wollen jetzt die Umkehrung zeigen.

Beispiel 2.50 Gegeben sei das GOTO-Programm

M_1: A_1;
M_2: A_2;
⋮
M_k: A_k

Wir simulieren dies durch ein WHILE-Programm mit *nur einer* WHILE-Schleife wie folgt:

$count := 1;$
WHILE $count \neq 0$ DO
IF $count = 1$ THEN A'_1 END;
IF $count = 2$ THEN A'_2 END;
$\vdots$
IF $count = k$ THEN A'_k END
END

wobei die Anweisungen A'_i folgendermaßen definiert werden:

$$A'_i = \begin{cases} x_j := x_\ell + c;\ count := count + 1 & \text{falls } A_i = x_j := x_\ell + c \\ x_j := x_\ell - c;\ count := count + 1 & \text{falls } A_i = x_j := x_\ell - c \\ count := n & \text{falls } A_i = \text{GOTO } M_n \\ \text{IF } x_j = c \text{ THEN } count := n & \\ \qquad \text{ELSE } count := count + 1 \text{ END} & \text{falls } A_i = \text{IF } x_j = c \text{ THEN GOTO } M_n \\ count := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

Aus Beispiel 2.50 erhalten wir sofort folgenden Satz.

Satz 2.51 *Jede* GOTO-*berechenbare Funktion ist auch* WHILE-*berechenbar.* □

Wir heben den Aspekt, dass die Simulation mit nur einer WHILE-Schleife auskommt, im folgenden Satz besonders hervor.

Satz 2.52 (Kleene Normalform für WHILE-Programme) *Jede* WHILE-*berechenbare Funktion kann durch ein* WHILE-*Programm mit nur einer* WHILE-*Schleife berechnet werden.*

Beweis. Sei P ein beliebiges WHILE-Programm zur Berechnung der Funktion f. Wir simulieren P zunächst durch ein GOTO-Programm P' gemäß Beispiel 2.47. Dann simulieren wir das GOTO-Programm P' durch ein WHILE-Programm P'' gemäß Beispiel 2.50. Offensichtlich berechnet das WHILE-Programm P'' dann die Funktion f und besitzt nur eine WHILE-Schleife. □

Jetzt zeigen wir noch, dass Turingmaschinen durch GOTO-Programme simuliert werden können.

Satz 2.53 *Jede Turingberechenbare Funktion ist auch* GOTO-*berechenbar.*

Beweis. Sei $M = (Z, \Sigma, \Gamma, \delta, z_1, \square, E)$ eine Turingmaschine zur Berechnung einer Funktion f. Wir simulieren M durch ein GOTO-Programm, das folgendermaßen aufgebaut ist:

$M_1 : P_1;$
$M_2 : P_2;$
$M_3 : P_3$

Hierbei transformiert P_1 die eingegebenen Anfangswerte der Variablen in Binärdarstellung und erzeugt eine Darstellung der Startkonfiguration von M, die sich in den Variablenwerten dreier Variablen x, y, z widerspiegelt. Wir geben diese Codierung von Turingmaschinen-Konfigurationen in drei natürliche Zahlen gleich im Detail an.

P_2 führt eine Schritt-für-Schritt-Simulation der Rechnung von M durch – durch entsprechendes Verändern der Variablenwerte von x, y und z.

P_3 schließlich erzeugt aus der codierten Form der Endkonfiguration in x, y, z die eigentliche Ausgabe in der Ausgabevariablen x_0

Man beachte, dass P_1 und P_3 gar nicht von M abhängen, sondern nur P_2.

Betrachte wir jetzt die schon erwähnte Codierung einer Konfiguration einer Turingmaschine M. Seien die Mengen

$$Z = \{z_1, z_2, \ldots, z_k\} \text{ und}$$
$$\Gamma = \{a_1, a_2, \ldots, a_k\}$$

durchnumeriert. Sei außerdem b eine Zahl mit $b > |\Gamma|$. Dann repräsentieren wir eine Turingmaschinen-Konfiguration

$$a_{i_1} a_{i_2} \ldots a_{i_p} z_\ell a_{j_1} a_{j_2} \ldots a_{j_q}$$

dadurch, dass die drei Programmvariablen x, y, z die Werte

$$x = (i_1 i_2 \ldots i_p)_b$$
$$y = (j_q j_{q-1} \ldots j_1)_b$$
$$z = \ell$$

annehmen, dabei bedeutet $(i_1 i_2 \ldots i_p)_b$ die Zahl $i_1 i_2 \ldots i_p$ in b-närer Darstellung, also

$$x = \sum_\mu = 1^p i_\mu \cdot b^{p-\mu}.$$

Analoges gilt für y (die Ziffern stehen hier jedoch in umgekehrter Reihenfolge).

Das GOTO-Programmstück $M_2 : P_2$ hat nun folgende Form:

M_2 : $a := y \bmod b$;
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} : $\circledast$;
GOTO M_2;
M_{12} : $\circledast$;
GOTO M_2;
$\vdots$
M_{km} : $\circledast$;
GOTO M_2

Wir beschreiben nun, was an den mit $\circledast$ bezeichneten Stellen passiert. Greifen wir repräsentativ das Programmstück, das mit der Marke M_{ij} beginnt, heraus. Nehmen wir an, dass die entsprechende δ-Anweisung

$$\delta(z_i, a_j) = (z_{i'}, a_{j'}, L)$$

lautet. Dies kann durch folgende Anweisungen simuliert werden:

$z := i'$;
$y := y \text{ div } b$;
$y := b * y + j'$;
$y := b * y + (x \bmod b)$;
$x := x \text{ div } b$

Entsprechend kann man sich die anderen Fälle vorstellen.

Falls z_i Endzustand ist, so setzen wir für $\circledast$ einfach

GOTO M_3

Die Konstruktionen von P_1 und P_3 sind nun entsprechend auszuführen und überlassen wir dem geneigten Leser. □

2.4 Die Churchsche These

Wir stellen die Ergebnisse aus dem vorangegangenen Kapitel noch mal im Folgenden zusammenfassenden Satz dar.

Satz 2.54 *Sei $f\colon \mathbb{N}^k \to \mathbb{N}$ eine Funktion, dann sind folgende Aussagen äquivalent:*

(i) *f ist* WHILE-*berechenbar.*

(ii) *f ist Turing-berechenbar.*

(iii) *f ist* GOTO-*berechenbar.* □

Außerdem erinnern wir, dass jede LOOP-berechenbare Funktion zwar WHILE-berechenbar ist, aber die Umkehrung nicht gilt.

Es gibt neben den von uns hier betrachteten Berechenbarkeitsbegriffen noch viele andere Begriffe, die in den letzten Jahrzehnten betrachtet wurden. Dazu zählen z. B. Berechenbarkeitsbegriffe, die auf *Flussdiagramme*, *Registermaschinen*, *partiell-rekursive Funktionen* zurückgeführt werden. Erstaunlicherweise musste man erkennen, dass all diese Begriffe nicht über den Begriff der Turingberechenbarkeit hinausführen. Das führt zu der Tatsache, dass man heute davon ausgeht, dass man mit dem Begriff der Turingberechenbarkeit genau den eingangs besprochenen Begriff der intuitiven Berechbarkeit getroffen hat.

Diese Überzeugung fasst man unter dem Namen *Churchche*[5] These zusammen:

Churchsche These *Jede intuitiv berechenbare Funktion ist Turing-berechenbar.*

Diese These ist *nicht* beweisbar, da der intuitive Berechenbarkeitsbegriff eben nicht formal mathematisch erfasst werden kann. Allerdings könnte man sie sofort widerlegen, indem man einen Berechenbarkeitsbegriff angibt, der eben nicht durch Turingmaschinen simuliert werden kann – das aber wurde in den vergangenen Jahrzehnten (die historisch ersten Definitionen von Turing und Church gehen auf 1936 zurück) immer wieder versucht, allerdings bis heute vergeblich, so dass man, wie oben schon ausgeführt, heute allgemein von der Richtigkeit der These ausgeht.

[5] A. Church, Amerikanischer Mathematiker