Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

Algorithmen und Berechnungskomplexität II

Skript SS 2016

Nach Aufzeichnungen von Rolf Klein und Elmar Langetepe

Bonn, Mai 2016

Inhaltsverzeichnis

1 Einführung **1**

2 Theoretische Berechenbarkeit **3**

2.1 Conways Spiel des Lebens 3
2.2 μ-rekursive Funktionen 4
2.3 Turingmaschinen 12
2.4 Entscheidbarkeit 31

3 Paktische Berechenbarkeit **38**

3.1 Die Random Access Machine 38

Literaturverzeichnis **40**

Kapitel 1

Einführung

Im vergangenen Semester haben wir verschiedene klassische algorithmische Fragestellungen (Sortieren, Flussprobleme, Graphenprobleme, Bin-Packing, Knapsack, etc.) untersucht, Lösungspläne dafür entworfen und die Effizienz dieser Pläne untersucht. Die Lösungspläne beruhten dabei auf sehr unterschiedlichen Vorgehensweisen (Divide-and-Conquer, Inkrementelle Konstruktion, Greedy, Dynamische Programmierung, Sweep, etc.). Diese Paradigmen haben wir für verschiedene Fragestellungen zur Anwendung gebracht. Für die Analyse der Lösungen wurden dann die wesentlichen Berechnungsschritte asymptotisch abgeschätzt. Neben der formalen Beschreibung dieser Abschätzungen (O- und Ω-Notation) haben wir auch verschiedene Methoden der Analyse (Rekursionsgleichungen, Induktionsbeweise, strukturelle Eigenschaften, Rekursionstiefe, etc.) kennengelernt und angewendet. Außerdem haben wir gesehen, dass zur rechnergestützten Umsetzung der Lösungspläne eine geeignete, effiziente Repräsentation der Daten (Datenstrukturen) im Rechner notwendig ist. Ein wesentlicher Bestandteil der Vorlesung war die formale Analyse der vorgestellten Techniken und Strukturen. Dadurch konnten wir Gütegarantien angeben. Die Formulierung der Algorithmen wurde im Wesentlichen durch Pseudocode dargestellt, der sich im Prinzip in jeder höheren Programmiersprache umsetzen lässt.
In diesem Semester beschäftigen wir uns mit der theoretischen und praktischen *Berechenbarkeit* von (mathematischen) Problemen. Dazu gehen wir in Kapitel 2 der Frage nach, welche Probleme bzw. Funktionen wir mit Algorithmen und modernen Computern überhaupt lösen können. Eine mathematische Funktion nennen wir *intuitiv berechenbar*, falls es einen Algorithmus gibt, der zu jeder Eingabe aus dem Definitionsbereich eine Ausgabe produziert. Terminiert der Algorithmus nicht, so war die Eingabe nicht in der Definitionsmenge enthalten. Dabei verstehen wir unter einem Algorithmus ein Rechenverfahren mit den folgenden Eigenschaften:

1. Die Rechenvorschrift besteht aus einem endlichen Text; der Ablauf ist eine Folge von (potenziell unendlich vielen) Rechenschritten.

2. Zu jedem Zeitpunkt ist eindeutig der nächste Rechenschritt bestimmt.
3. Jeder Rechenschritt hängt nur von der Eingabe und den bisher berechneten Zwischenergebnissen ab.

Intuitiv haben wir bereits eine Vorstellung, wann eine mathematische Funktionen berechenbar ist. In Kapitel 2 formalisieren wir diesen intuitiven Berechenbarkeitsbegriff. Wie wir sehen werden, gibt es tatsächlich auch Problemstellungen, die nicht berechenbar sind. Das heißt wir können sie mit modernen Computern nicht lösen. Im zweiten Teil dieser Vorlesung konzentrieren wir uns auf die *praktische Berechenbarkeit*. Obwohl ein Problem theoretisch berechenbar ist, kann es sein, dass die Berechnung des Ergebnisses viele Jahre dauern würde. Aus diesem Grund führen wir in Kapitel 3 eine Charakterisierung von Schwere-Klassen der Problemstellungen je nach Komplexität ein.

Kapitel 2

Theoretische Berechenbarkeit

2.1 Conways Spiel des Lebens

Zur Motivation betrachten wir das *Spiel des Lebens*, welches J. H. Conway 1970 entwarf. Das Spielfeld besteht aus einem Gitter quadratischer Zellen. Wir nehmen an, dass die Anzahl an Zellen in jede Richtung unbeschränkt ist. Jede der Zellen kann genau einen der Zustände *lebendig* bzw. *tot* annehmen. Das Spiel findet in Zeitschritten statt. Zu Beginn des Spiels, d.h. zum Zeitpunkt $t = 0$ wird eine erste *Generation* lebendiger Zellen auf dem Gitter platziert, die übrigen Zellen sind tot. Zu jedem Zeitpunkt $t \geq 0$ ist die Generation zum Zeitpunkt $t + 1$ durch die folgenden Spielregeln bestimmt:

1. Eine tote Zelle zum Zeitpunkt t lebt zum Zeitpunkt $t + 1$, falls genau drei (der acht direkten) Nachbarn zum Zeitpunkt t leben.

2. Eine lebende Zelle zum Zeitpunkt t überlebt, falls sie zum Zeitpunkt t zwei oder drei lebende Nachbarn hat.

Mit diesen Spielregeln können wir leicht einen Algorithmus angeben, der zu jedem Zeitpunkt die folgende Generation berechnet. Es kann jedoch keinen Algorithmus geben, der für zwei beliebige Generationen prüft, ob die zweite Generation zu irgend einem Zeitpunkt aus der ersten hervorgeht. Dies kann man auch formal beweisen. Offensichtlich haben wir hier eine Grenze der Berechenbarkeit erreicht. Solche Grenzen werden wir in Abschnitt 2.4 untersuchen.
Zunächst sehen wir uns an, was überhaupt *berechnet* werden kann. Intuitiv haben wir davon eine Vorstellung. Zur Formalisierung des Berechenbarkeitsbegriffs wurden einige Konzepte (bzw. Modelle) entwickelt. In dieser Vorlesung lernen wir zwei der Konzepte näher kennen. In Abschnitt 2.2 betrachten wir den Ansatz der μ-rekursiven Funktionen. Anschließend untersuchen

wir die von Turing eingeführten Turingmaschinen. Beide Konzepte definieren den Begriff der Berechenbarkeit auf ganz verschiedene Art und Weise. Dennoch sind beide Konzepte gleich mächtig, was wir auch formal beweisen werden. Diese Tatsache nehmen wir als Indiz für die Gültigkeit der folgenden These, die die Fähigkeiten moderner Rechenmaschinen beschreibt.

Church-Turing-These *(Churchsche These) Die Klasse der durch Turingmaschinen (bzw. μ-rekursiven Funktionen) berechenbaren Funktionen umfasst alle intuitiv berechenbaren Funktionen.*

Da wir den Begriff *intuitiv berechenbarer* Funktionen nicht exakt formalisieren können, lässt sich diese These auch nicht beweisen. Es gibt jedoch eine Vielzahl weiterer Formalisierungen, die alle hinsichtlich ihrer Berechnungsstärke äquivalent sind.

2.2 μ-rekursive Funktionen

Wir konstruieren uns zunächst eine Klasse (bzw. Menge) von berechenbaren Funktionen. Die Idee besteht darin, einige wenige einfache **Grundfunktionen** zu definieren. Aus diesen lassen sich dann mittels einfacher **Operationen** (bzw. Schemata) neue berechenbare Funktionen gewinnen.

(A) Grundfunktionen (für $r, s \in \mathbb{N}_0$)

(i) **Konstante Funktionen**

$$\begin{aligned} c_s^r : \quad & \mathbb{N}_0^r && \to \mathbb{N}_0 \\ & \mathfrak{x} = (x_1, x_2, \ldots, x_r) && \mapsto s \end{aligned}$$

(ii) **Nachfolgefunktion**

$$\begin{aligned} N : \quad & \mathbb{N}_0 && \to \mathbb{N}_0 \\ & x && \mapsto x + 1 \end{aligned}$$

(iii) **Projektionen**

$$\begin{aligned} P_i^r : \quad & \mathbb{N}_0^r && \to \mathbb{N}_0 \\ & \mathfrak{x} = (x_1, x_2, \ldots, x_r) && \mapsto x_i \end{aligned}$$

(B) Operationen (für $r, s \in \mathbb{N}_0$)

(i) **Substitution** (simultanes Einsetzen) Für

$$\begin{aligned} g_1, \ldots, g_r : \quad & \mathbb{N}_0^m && \to \mathbb{N}_0 \\ g : \quad & \mathbb{N}_0^r && \to \mathbb{N}_0 \end{aligned}$$

definiere

$$\begin{aligned} h: \quad \mathbb{N}_0^m &\to \mathbb{N}_0 \\ \mathfrak{x} &\mapsto g\left(g_1(\mathfrak{x}), \ldots, g_r(\mathfrak{x})\right) \end{aligned}$$

(ii) **Primitive Rekursion** Für

$$\begin{aligned} g: \quad \mathbb{N}_0^r &\to \mathbb{N}_0 \\ f: \quad \mathbb{N}_0^{r+2} &\to \mathbb{N}_0 \end{aligned}$$

definiere

$$\begin{aligned} h: \quad \mathbb{N}_0^{r+1} &\to \mathbb{N}_0 \\ h(\mathfrak{x}, 0) &:= g(\mathfrak{x}) \\ h(\mathfrak{x}, y+1) &:= f(\mathfrak{x}, y, h(\mathfrak{x}, y)) \end{aligned}$$

Aus diesen Grundfunktionen und Operationen können wir nun die Klasse der primitiv rekursiven Funktionen wie folgt erzeugen.

Definition 1 (Primitiv rekursive Funktionen)**.** Die Klasse $\mathcal{P}$ der primitiv rekursiven Funktionen ist die kleinste Klasse von Funktionen, die

1. die Grundfunktionen enthält und
2. abgeschlossen ist unter den Operationen Substitution und primitive Rekursion.

Eine Funktion f heißt primitiv rekursiv genau dann wenn $f \in \mathcal{P}$ gilt.

Für eine primitiv rekursive Funktion können wir also eine Folge von Grundfunktionen und Operationen angeben, aus denen sie sich herleitet. Um zu beweisen, dass eine gegebene Funktion f tatsächlich primitiv ist ($f \in \mathcal{P}$), müssen wir für sie eine solche Herleitung finden. Wir illustrieren dies anhand folgender Beispiele.

Beispiel 1. 1. Addition $a(x, y) = x + y$

$$\begin{aligned} a(x, 0) &:= P_1^1(x) \\ x + y + 1 = a(x, y+1) &:= N\left(P_3^3(x, y, a(x, y))\right) \end{aligned}$$

2. Multiplikation $m(x, y) = x \cdot y$

$$\begin{aligned} m(x, 0) &:= c_0^1(x) \\ m(x, y+1) &:= a\left(P_3^3(x, y, m(x, y)), P_1^3(x, y, m(x, y))\right) \\ &= x \cdot y + x \end{aligned}$$

Folglich ist die Multiplikation primitiv rekursiv, da wir bereits gezeigt haben, dass die Addition primitiv rekursiv ist und $\mathcal{P}$ unter Substitution abgeschlossen ist.

3. Potenz $h(x,y) = x^y$

$$\begin{aligned} h(x,0) &:= c_1^1(x) \\ h(x,y+1) &:= m\left(P_3^3(x,y,h(x,y)), P_1^3(x,y,h(x,y))\right) \\ &= x^y \cdot x \end{aligned}$$

Hier verwenden wir, dass die Multiplikation primitiv rekursiv ist.

4. Vorgängerfunktion $V(y) = \begin{cases} 0 & , y = 0 \\ y-1 & , y > 0 \end{cases}$

$$\begin{aligned} V(0) &:= c_0^0 \\ V(y+1) &:= P_1^2(y, V(y)) \end{aligned}$$

5. Modifizierte Differenz $d(x,y) = \begin{cases} x-y & , x \geq y \\ 0 & , \text{sonst} \end{cases}$

$$\begin{aligned} d(x,0) &:= P_1^1(x) \\ d(x,y+1) &:= V\left(P_3^3(x,y,d(x,y))\right) \end{aligned}$$

Statt $d(x,y)$ werden wir im Folgenden auch $x \dot{-} y$ schreiben.

Um komplexere Funktionen beschreiben zu können, verwenden wir Prädikate. Ein $r-$stelliges *Prädikat* P über $\mathbb{N}_0$ ist eine Teilmenge von $\mathbb{N}_0^r$. Zu diesem Prädikat P gehört außerdem eine *charakteristische Funktion*

$$\begin{aligned} \chi_P : &\quad \mathbb{N}_0^r \to \{0,1\} \\ \chi_P(\mathfrak{x}) &= \begin{cases} 1 & , \mathfrak{x} \in P \\ 0 & , \mathfrak{x} \notin P. \end{cases} \end{aligned}$$

Statt $\mathfrak{x} \in P$ schreiben wir kurz $P(\mathfrak{x})$. Wir nennen das Prädikat P genau dann primitiv rekursiv, wenn die zugehörige charakteristische Funktion $\chi_P \in P$ primitiv rekursiv ist.

Lemma 1. *Mit P und Q sind auch die Prädikate*

$$\begin{aligned} P \wedge Q &:= P \cap Q \\ P \vee Q &:= P \cup Q \\ \neg P &:= \mathbb{N}_0^r \setminus P \end{aligned}$$

primitiv rekursiv.

Beweis. Es gilt

$$\begin{aligned}\chi_{P\wedge Q} &= \chi_P \cdot \chi_Q \\ \chi_{P\vee Q} &= \operatorname{sg}(\chi_P + \chi_Q) \\ \chi_{\neg P} &= 1 \dot{-} \chi_P = 1 - \chi_P\end{aligned}$$

wobei die Funktion $\operatorname{sg}(x) := \begin{cases} 1 & , x > 0 \\ 0 & , x = 0 \end{cases} \in \mathcal{P}$ ist. (Übungsaufgabe) □

Mit Hilfe von Prädikaten lässt sich bequem beweisen, dass die Definition von Funktionen mit einer endlichen Anzahl an Fallunterscheidungen nicht aus $\mathcal{P}$ herausführt. Jede so neu definierte Funktion ist also ebenfalls primitiv rekursiv, wie wir nun zeigen.

Theorem 1. *Seien $P_1, \ldots P_k$ paarweise disjunkte r-stellige primitiv rekursive Prädikate, und seien $f_1, \ldots, f_k : \mathbb{N}_0^r \to \mathbb{N}_0$ primitiv rekursive Funktionen. Dann ist auch folgende Funktion primitiv rekursiv:*

$$g(\mathfrak{x}) := \begin{cases} f_1(\mathfrak{x}) & , \textit{falls}\, P_1(\mathfrak{x}) \\ \vdots & \\ f_k(\mathfrak{x}) & , \textit{falls}\, P_k(\mathfrak{x}) \\ 0 & , \textit{sonst.} \end{cases}$$

Beweis. Wegen der Disjunktheit der Prädikate ist g wohldefiniert. Wir können g ausdrücken durch

$$\begin{aligned} g(\mathfrak{x}) &= \sum_{i=1}^{k} \chi_{P_i}(\mathfrak{x}) \cdot f_i(\mathfrak{x}) \\ &= \underbrace{a(\chi_{P_i}(\mathfrak{x}) \cdot f_1(\mathfrak{x}), a(\cdots))}_{k-\text{mal verschachtelt}} \end{aligned}$$

□

Nun stellt sich die Frage, ob wir mit der Klasse der primitiv rekursiven Funktionen bereits eine Beschreibung aller intuitiv berechenbaren Funktionen gefunden haben. Dass dem nicht so ist lässt sich mittels Diagonalisierung beweisen. W. Ackermann gab sogar explizit eine mathematische Funktion an, die nicht primitiv rekursiv ist.
Da die Klasse der primitiv rekursiven Funktionen nicht alle intuitiv berechenbaren Funktionen abdeckt wollen wir jetzt eine entsprechende Erweiterung von $\mathcal{P}$ vornehmen. Wir führen einen Operator ein, der auf der Ganzheit der natürlichen Zahlen nach der kleinsten Nullstelle sucht.

Definition 2 ($\mu-$Operator)**.** Sei $f : \mathbb{N}_0^{r+1} \to \mathbb{N}_0$ eine Funktion. Dann wird $\mu f : \mathbb{N}_0^r \to \mathbb{N}_0$ definiert durch

$$\mu f(\mathfrak{x}) := \begin{cases} \text{das kleinste } y \text{ mit } f(\mathfrak{x}, y) = 0 \text{ und} & \\ f(\mathfrak{x}, 0), \ldots, f(\mathfrak{x}, y-1) \text{ ist definiert} & \text{, falls } y \text{ existiert} \\ \text{undefiniert} & \text{, sonst.} \end{cases}$$

Falls der zweite Fall nie eintritt, d.h. falls gilt $\forall \mathfrak{x} \exists y : f(\mathfrak{x}, y) = 0$, sagt man: μf entsteht aus f durch Anwendung des μ-Operators *im Normalfall*. Insbesondere ist $\mu f(\mathfrak{x})$ undefiniert, falls es kein y gibt mit $f(\mathfrak{x}, y) = 0$ oder $f(\mathfrak{x}, i)$ undefiniert ist für mindestens ein $0 \leq i \leq y - 1$. Wir erweitern nun die Klasse der primitiv rekursiven Funktionen, indem wir die Anwendung des μ Operators erlauben.

Definition 3 ($\mu-$rekursive Funktionen)**.** Die Klasse $\mathcal{F}_\mu^{tot}$ der $\mu-$rekursiven Funktionen ist die kleinste Klasse, die

1. die Grundfunktionen enthält,
2. abgeschlossen unter den Operationen Substitution und primitiver Rekursion ist und
3. abgeschlossen ist unter Verwendung des $\mu-$Operators im Normalfall.

Wir definieren $\mathcal{F}_\mu^{par}$ als die Klasse der partiellen $\mu-$rekursiven Funktionen analog, fordern jedoch die Abgeschlossenheit bzgl des μ-Operators generell.

Schauen wir uns zunächst ein paar Beispiele für die Leistungsfähigkeit des μ-Operators an:

Beispiel 2. 1. Sei $f(x, y) = x \dot{-} 13y$, dann ist $\mu f(x) = \lceil x/13 \rceil$ total.

2. Sei $g(x, y) = (x \dot{-} 13y) + (13y \dot{-} x)$. Dann ist $\mu g(x) = x/13$, falls x durch 13 teilbar ist, sonst undefiniert.
3. Es gilt folgende Folgerung: $f \in \mathcal{F}_\mu^{tot}$ bijektiv $\Rightarrow f^{-1} \in \mathcal{F}_\mu$.
   Zum Beweis setze $h(v, w) := (v \dot{-} f(w)) + (f(w) \dot{-} v)$. Dann ist h μ-rekursiv und und $h(v, w) = 0$ genau dann, wenn $f(w) = v$. Setzen wir nun $g(v) := \mu h(v)$, dann folgt

   $$\begin{aligned} g(v) &= \text{kleinstes } w \text{ mit } f(w) = v \\ &\underset{f \text{ bij.}}{=} \text{das } w \text{ mit } f(w) = v. \end{aligned}$$

   Damit haben wir jedoch bereits f^{-1} gefunden, denn $g = f^{-1}$. Man könnte auch sagen, dass der μ-Operator nach $f^{-1}(v)$ *sucht*.

Nun müssen wir zeigen, dass wir mit Hilfe des μ-Operators tatsächlich die Klasse der primitiv rekursiven Funktionen erweitert haben.

Theorem 2.

$$\mathcal{P} \subsetneq \mathcal{F}_\mu^{tot} \subsetneq \mathcal{F}_\mu^{par}$$

Beweis. Die zweite Inklusion ist offensichtlich. Es ist jedoch zu beachten, dass sich nicht jedes $f \in \mathcal{F}_\mu^{par}$ zu einem $\hat{f} \in \mathcal{F}_\mu^{tot}$ fortsetzen lässt!
Die erste Inklusion kann mittels Diagonalisierung bewiesen werden. Der Beweis findet sich beispielsweise in [1]. Alternativ können wir auch eine intuitiv berechenbare Funktion konstruieren, die nicht primitiv rekursiv ist. Die Ackermannfunktion erfüllt genau diese Anforderungen, wie wir im folgenden zeigen. □

Wir konstruieren nun eine μ-rekursive Funktion A, die nicht in $\mathcal{P}$ liegt. Idee: extrapoliere $N, +, \cdot, x^y, \ldots$

$$\begin{aligned}
f_1(x,y) &= x+y \\
f_2(x,y) &= x \cdot y \\
f_3(x,y) &= x^y \\
f_4(x,y) &= \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{y-\text{mal}} \\
&\vdots \\
f_{n+1}(x,y+1) &= f_n(x, f_{n+1}(x,y))
\end{aligned}$$

Wir wenden also bei jedem Folgenglied die Operation des vorigen Folgenglieds y-mal auf x an. Das schnelle Wachstum der Funktion hängt nicht so sehr von x ab (schon für $x = 2$ extrem). Deshalb eine kompaktere Definition:

$$\begin{aligned}
f_0(y) &:= y+1 \\
f_{n+1}(0) &:= f_n(1) \\
f_{n+1}(y+1) &:= f_n(f_{n+1}(y))
\end{aligned}$$

Per Induktion lässt sich zeigen, dass jede Funktion f_n in $\mathcal{P}$ liegt. Es handelt sich um eine primitive Rekursion. Ab jetzt betrachten wir den Index n als Variable angesehen und erhalten die *Ackermannfunktion*

$$A(x,y) := f_x(y).$$

Also

$$\begin{aligned}
A(0,y) &= y+1 \\
A(x+1,0) &= A(x,1) \\
A(x+1,y+1) &= A(x, A(x+1,y)).
\end{aligned}$$

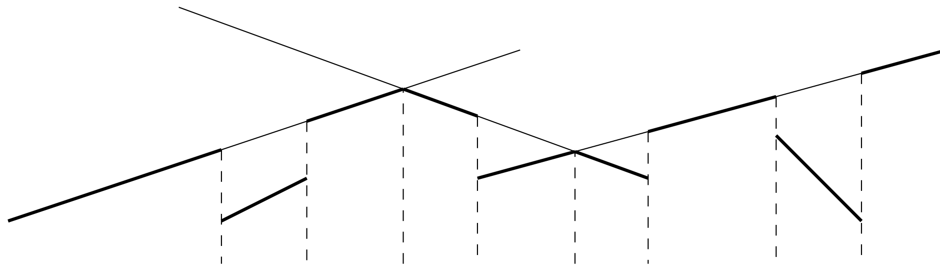


Abbildung 2.1: Zur Komplexität der unteren Kontur von Liniensegmenten.

Diese Funktion ist offensichtlich intuitiv berechenbar. Man rechnet leicht nach, dass

$$\begin{aligned} A(0,y) &= y+1 \\ A(1,y) &= y+2 \\ A(2,y) &= 2y+3 \\ A(3,y) &= 2^y \cdot 8 - 3 \\ A(4,y) &= \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{y+3\text{-mal}} - 3 \end{aligned}$$

Außerdem kommt diese *künstlich* erzeugte Funktion (bzw. ihr Inverses) tatsächlich *in der Natur* vor! Für

$$\alpha(n) := \mu x : A(x,x) \geq n$$

kann die untere Kontur (siehe Abbildung 2.1) von n Liniensegmenten aus $\Theta(n\alpha(n))$ vielen Stücken bestehen! Diese Tatsache können wir in der Algorithmischen Geometrie zur Laufzeitanalyse verwenden.
Wir werden nun im folgenden zeigen, dass die Ackermannfunktion nicht primitiv rekursiv, aber μ-rekursiv ist. Es gilt also tatsächlich $\mathcal{P} \subsetneq \mathcal{F}_\mu^{tot}$. Dazu zeigen wir:

1. A ist nicht primitiv rekursiv, d.h. $A \notin \mathcal{P}$. (Denn A wächst schneller als jede Funktion in $\mathcal{P}$.)
2. A ist μ-rekursiv, d.h. $A \in \mathcal{F}_\mu^{tot}$.

Der Beweis, dass A nicht primitiv rekursiv ist stützt sich auf folgendes Lemma, welches wir hier nicht beweisen werden.

Lemma 2. *Für jede primitiv rekursive Funktion $f \in \mathcal{P}$ gibt es ein k, sodass*

$$f(\mathfrak{x}) \leq A(k, \underbrace{\sum \mathfrak{x}}_{x_1+x_2+\cdots+x_r})$$

für alle $\mathfrak{x}$.

Mit Hilfe von Lemma 2 können wir folgenden Widerspruchsbeweis führen. Angenommen A wäre primitiv rekursiv, dann wäre auch $f(x) := A(x,x)+1$ primitiv rekursiv. Damit folgt aus dem Lemma 2, dass stets ein k existiert, sodass

$$f(x) = A(x,x) + 1 \leq A(k,x).$$

Mittels Diagonalisierung, also der Wahl $x := k$ folgt

$$A(k,k) + 1 \leq A(k,k),$$

was ein Widerspruch ist. Die Ackermannfunktion kann also nicht primitiv rekursiv sein. Intuitiv ist klar, dass die Ackermannfunktion μ-rekursiv ist. Auf den Beweis verzichten wir jedoch an dieser Stelle und verweisen auf Abschnitt 2.3.

Exkurs Schließlich betrachten wir noch folgende nützliche, schwächere Version des μ-Operators. Sie heißt *beschränkter μ-Operator* und wir werden sie später benötigen. Im Gegensatz zum μ-Operator führt sie nicht über $\mathcal{P}$ hinaus.

Theorem 3. *Mit $f : \mathbb{N}_0^{r+1} \to \mathbb{N}_0$ ist auch der beschränkte μ-Operator*

$$\begin{aligned} \mu_b f : \quad \mathbb{N}_0^{r+1} &\to \mathbb{N}_0 \\ (\mathfrak{x}, y) &\mapsto \begin{cases} \textit{das kleinste } x \leq y & \\ \textit{mit } f(\mathfrak{x}, x) = 0, & \textit{falls ein solches existiert} \\ 0, & \textit{sonst} \end{cases} \end{aligned}$$

primitiv rekursiv.

Beweis. Der Suchraum des Operators ist beschränkt und wir können primitive Rekursion anwenden:

$$\begin{aligned} \mu_b f(\mathfrak{x}, 0) &= 0 \\ \mu_b f(\mathfrak{x}, y+1) &= \begin{cases} y+1, & \text{falls } f(\mathfrak{x}, y+1) = 0 \text{ und} \\ & \mu_b f(\mathfrak{x}, y) = 0 \text{ und } f(\mathfrak{x}, 0) > 0 \\ \mu_b f(\mathfrak{x}, y), & \text{sonst.} \end{cases} \end{aligned}$$

Da wir eine endliche Fallunterscheidung verwendet haben und die zweistelligen Prädikate $\vee, =, >$ (Übungsaufgabe) und auch f primitiv rekursiv sind, folgt nach Theorem 1, dass auch $\mu_b f$ primitiv rekursiv ist. □

Rekapitulation

Zur Beschreibung intuitiv berechenbarer Funktionen haben wir zwei Klassen von Funktionen definiert:

$\mathcal{P}$	primitiv rekursive Funktionen	Konstante c_i^r, Nachfolger N, Projektionen P_i^r, abgeschlossen gegen Substitution und Primitive Rekursion
$\mathcal{F}_\mu$	$\mu-$rekursive Funktionen	$\mathcal{P} + \mu-$Operator im Normalfall

Zum Nachweis, dass $\mathcal{P} \subsetneq \mathcal{F}_\mu$ haben wir gezeigt der $\mu-$Operator nicht entbehrlich ist. Dafür haben wir die Ackermannfunktion A konstruiert und gezeigt, dass diese nicht in $\mathcal{P}$ liegt. Dass A tatsächlich μ-rekursiv ist werden wir erst später exakt beweisen.
Die allgemeine Frage, der wir nachgehen ist, ob alle intuitiv berechenbaren Funktionen in $\mathcal{F}_\mu^{tot}$ liegen. Mit dem Ansatz der μ-rekursiven Funktionen haben wir eine *Stütze* für die Church-Turing-These gefunden. Diese lässt sich zwar nicht beweisen, jedoch durch verschiedene Ansätze erhärten. Im folgenden Kapitel betrachten wir einen zweiten Ansatz.

2.3 Turingmaschinen

In diesem Abschnitt führen wir als zweites Berechnungsmodell Turingmaschinen ein. Wir werden außerdem zeigen, dass beide Modell äquivalent sind. Das Modell wurde 1936, also noch vor der Entwicklung erster Computer, von Alan M. Turing beschrieben. Seine Abstraktion eines Computers wird ihm zu Ehren *Turingmaschine* genannt.
Ganz allgemein nehmen wir an, dass der Speicher aus k Halbbändern besteht. Wie wir später in Lemma 7 zeigen, reicht streng genommen ein einziges solches Halbband aus. In einigen Beweisen werden wir jedoch auch sehen, dass Turingmaschinen mit beliebig vielen Halbbändern deutlich einfacher zu beschreiben sind. Jedes Halbband besteht aus einer Folge von *Bandquadraten* und ist nach rechts unbeschränkt groß. In jedem Bandquadrat steht genau ein Zeichen aus einem endlichen Bandalphabet Σ. Das Bandalphabet enthält die *Sonderzeichen* $\$, \#$, und $\sqcup$. Das Zeichen \$ markiert den Beginn des Bandes, $\sqcup$ eine leere Zelle und # wird als Trennzeichen verwendet. Die Turingmaschine verwendet eine sogenannte *endliche Kontrolle*. Diese besteht aus je einem Lese-/Schreibkopf für jedes der Halbbänder. Dieser zeigt stets auf ein Bandquadrat des Bands und ermöglicht das Auslesen und Verändern des jeweiligen Zeichens. Der Kopf kann schrittweise über das jeweilige Band bewegt werden. Diese Bewegung kodieren wir durch die Zahlen -1, 1 und 0. Dabei bedeutet -1 eine Bewegung des Kopfs um ein Bandquadrat nach links, $+1$ nach rechts und 0 keine Bewegung. Die Kontrolle handelt

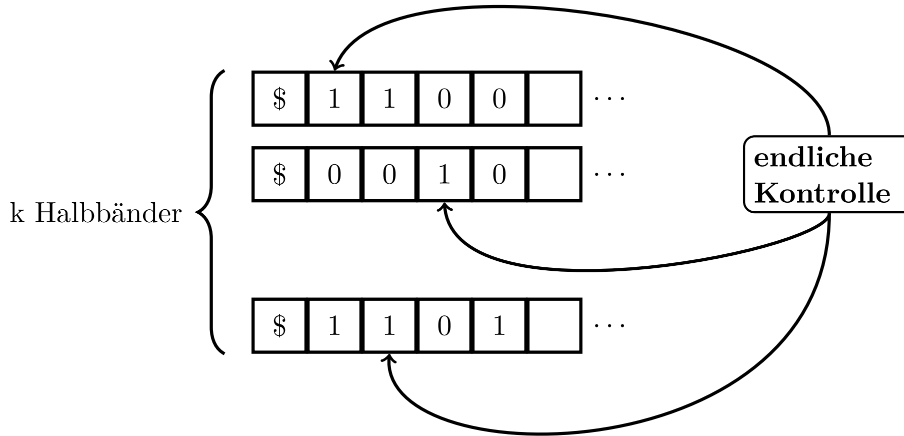


Abbildung 2.2: Darstellung einer Turingmaschine.

deterministisch und die Maschine befindet sich zu jedem Zeitpunkt in einem Zustand q aus einer endlichen Zustandsmenge Q. Mittels einer (ggf. partiellen) Zustandsübergangsfunktion $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{-1, 0, +1\}^k$ regelt die endliche Kontrolle die Arbeit der Turingmaschine. Diese Funktion ordnet einem aktuellen Zustand q, unter Berücksichtigung der von allen Köpfen gelesenen Zeichen $s_1, \ldots s_k$, einen neuen Zustand q' zu:

$$(q, s_1, \ldots s_k) \mapsto (q', s'_1, \ldots, s'_k, m_1, \ldots, m_k).$$

Dabei legt sie auch fest, welche Zeichen des Bandalphabets in die ausgelesenen Bandquadrate geschrieben werden und welche Bewegung jeder Kopf nach dem Schreiben ausführt. Vorstellen können wir uns eine Turingmaschine wie in Abschnitt 2.3 dargestellt.
Turingmaschinen können wir auch als Erweiterung von endlichen Automaten in drei Kriterien interpretieren: Bewegung, Schreiben, Speicherplatz. Der Kopf darf sich sowohl nach rechts als auch nach links bewegen. Die Turingmaschine darf ein Eingabesymbol an der Position des Kopfes auch verändern. Außerdem ist der Arbeitsbereich (bzw. Speicher) nach rechts über die Eingabe hinaus unbeschränkt. Damit endet eine Berechnung nicht mehr nach dem Lesen des letzten Eingabezeichens sondern in speziellen *Endzuständen*. Formal wird die Maschine wie folgt definiert.

Definition 4. Eine (deterministische) k-Band Turingmaschine (DTM) M ist ein 5-Tupel $M = (\Sigma, Q, \delta, q_0, F)$, wobei

1. Σ ein endliches Bandalphabet mit $\{0, 1, \#, \$, \sqcup\} \subseteq \Sigma$,

2. Q eine endliche Zustandsmenge mit $Q \cap \Sigma = \emptyset$,

3. $q_0 \in Q$ der Startzustand,
4. $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{-1, 1, 0\}^k$ die Zustandsübergangsfunktion und
5. $F := \{q \in Q \mid \forall \mathfrak{x} \in \Sigma^k : \delta(q, \mathfrak{x}) \text{ ist undefiniert}\}$ die Menge der Endzustände

sind.

Die Maschine arbeitet schrittweise wie folgt. Das Zeichen $a \in \Sigma$ unter dem Lese-/Schreibkopf wird gelesen und wenn sich die Maschine im Zustand $q \in Q$ befindet, wird $\delta(q, a) = (q', b, d) \in Q \times \Sigma \times \{-1, 1, 0\}$ ausgewertet:

- Die Maschine schreibt $b \in \Sigma$ auf die aktuelle Zelle.
- Die Maschine bewegt den Lese-/Schreibkopf nach links ($d = -1$), nach rechts ($d = 1$) oder verbleibt auf der aktuellen Zelle ($d = 0$).
- Die Maschine wechselt in den Zustand q'.

Weiterhin legen wir zunächst folgende **Regeln** fest. Das Zeichen \$ darf nie überschrieben werden. Wenn das Zeichen \$ gelesen wird, dann darf die Maschine im nächsten Schritt den Lese-/Schreibkopf nicht nach links bewegen. Der Kopf bewegt sich gemäß der oben angegebenen Aktion. Ein einzelner solcher Schritt wird als *Rechenschritt* bezeichnet. Die Anzahl der Rechenschritte, bis die Maschine in einem Endzustand landet, wird als *Laufzeit* bezeichnet. Die Anzahl der insgesamt aktiv verwendeten Zellen ergibt den benötigten *Speicherplatz*. Die Maschine *terminiert*, wenn ein Endzustand erreicht wird. Zu Beginn steht jeder Lese-/Schreibkopf auf dem ersten Zeichen hinter dem Bandanfang \$. Die Eingabe der Länge n steht in den ersten n Bandquadraten des ersten Bandes. Alle übrigen Bandquadrate aller Bänder, bis auf die jeweils ersten, welche \$ enthalten, sind mit $\sqcup$ beschrieben.
Wir möchten nun spezifizieren, was es bedeutet, dass eine Turingmaschine M eine Funktion $f : \mathbb{N}_0^r \to \mathbb{N}_0$ berechnen kann. Dafür stellen wir die Eingaben mit der Abbildung

$$\begin{aligned} \text{bin} : \qquad\qquad \mathbb{N}_0 &\to \{0,1\}^+ = \bigcup_{n \geq 1} \{0,1\}^n \\ n = \sum_{i=1}^{s} w_i \cdot 2^{s-i} &\mapsto w_1 w_2 \dots w_s \end{aligned}$$

binär dar, wobei $w_i \in \{0, 1\}$ gilt.

Definition 5 (turingberechenbar)**.** Sei $M = (Q, \Sigma, \delta, q_0, F)$ eine k-Band Turingmaschine und $f : \mathbb{N}_0^r \to \mathbb{N}_0$ eine partielle Funktion. Wir sagen M *berechnet* f, falls für alle $\mathfrak{x} = (x_1, \dots, x_k) \in \mathbb{N}_0^k$ gilt:
M, angesetzt auf die Eingabe $\text{bin}(x_1)\#\,\text{bin}(x_2)\# \dots \#\,\text{bin}(x_k)$ stoppt genau dann nach endlich vielen Schritten, wenn $f(\mathfrak{x})$ definiert ist.

Da nun klar ist, was es bedeutet, dass eine (partielle) Funktion von einer deterministischen Turingmaschine berechnet wird, führen wir zwei Klassen zur Beschreibung turingberechenbarer Funktionen ein.

Definition 6 (turingberechenbare Funktionen)**.** Die Klasse der totalen turingberechenbaren Funktionen ist

$$\mathcal{F}_{\text{TM}}^{tot} := \{f : \mathbb{N}_0^r \to \mathbb{N}_0 \mid r \geq 0 \land \exists \text{ DTM M, die f berechnet}\}.$$

Die Klasse der partiellen turingberechenbaren Funktionen ist

$$\mathcal{F}_{\text{TM}}^{par} := \{f : \mathbb{D}_f \to \mathbb{N}_0 \mid r \geq 0 \land \exists \text{ DTM M, die f berechnet}\},$$

wobei $\mathbb{D}_f \subseteq \mathbb{N}_0^r$ den Definitionsbereich der Funktion f bezeichne.

Beispiel 3. Konstruktion einer 2-Band Turingmaschine M zur Berechnung der Nachfolgefunktion N. Dabei verwenden wir folgendes Programm:

- Kopiere Band 1 auf Band 2; (kurz Band 2 := Band 1)
- Lösche Band 1;
- Addiere binär 1 zum Inhalt von Band 2 von rechts nach links. Speichere Übertrag in Zuständen q_2 und q_3.
- Falls am Ende q_2, schreibe 1 auf Band 1 und bewege Kopf nach rechts;
- Hänge Inhalt von Band 2 an Band 1;
- Lösche Band 2;
- Köpfe nach vorne.

Zur konkreten Beschreibung der Übergangsfunktion δ verwenden wir eine etwas andere Tabellenschreibweise.

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_0	0/1	$\sqcup$	q_0	$\sqcup$	0/1	1	1
q_0	$\sqcup$	$\sqcup$	q_1	$\sqcup$	$\sqcup$	-1	-1

Band 2 := Band 1;
Band 1 löschen;

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_1	$\sqcup$	0/1	q_1	$\sqcup$	0/1	-1	0
q_1	\$	0/1	q_2	\$	0/1	1	0

Kopf 1 nach links;
Kopf 2 bleibt rechts;

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_2	$\sqcup$	1	q_2	$\sqcup$	0	0	-1
q_2	$\sqcup$	0	q_3	$\sqcup$	1	0	-1
q_3	$\sqcup$	0/1	q_3	$\sqcup$	0/1	0	-1
q_3	$\sqcup$	\$	q_4	$\sqcup$	\$	0	1

Addiere 1 zu Band 2 von rechts nach links;
q_2 Übertrag,
q_3 kein Übertrag.

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_2	$\sqcup$	\$	q_4	1	\$	1	1

Falls am Ende q_2, schreibe 1 auf Band 1;
Kopf 1 nach rechts;

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_4	$\sqcup$	0/1	q_4	0/1	$\sqcup$	1	1
q_4	$\sqcup$	$\sqcup$	q_5	$\sqcup$	$\sqcup$	-1	-1

Hänge Band 2 an Band 1;
Lösche Band 2.

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_5	0/1	$\sqcup$	q_5	0/1	$\sqcup$	-1	0
q_5	\$	$\sqcup$	q_6	\$	$\sqcup$	1	0
q_6	0/1	$\sqcup$	q_6	0/1	$\sqcup$	0	-1
q_6	0/1	\$	q_7	0/1	\$	0	1

Kopf 1 nach links;
Kopf 2 nach links.

$M = (\{\$, 0, 1, \sqcup\}, \{q_0, \dots, q_7\}, \delta, q_0, \{q_7\})$ ist die gesuchte Turingmaschine.

2.3.1 Äquivalenz der Berechnungsmodelle

Auf den ersten Blick scheint der Ansatz der Turingmaschinen zur Beschreibung des Berechenbarkeitsbegriffs ganz anders als der Ansatz der μ-rekursiven Funktionen zu sein. Wie wir im folgenden Abschnitt zeigen, sind die Klasse der μ-rekursive Funktionen und die Klasse der turingberechenbaren Funktionen jedoch äquivalent. Dass beiden Modelle gleich mächtig sind, nehmen wir als weiteres Indiz für die Gültigkeit der Church-Turing-These.
Wir zeigen die Äquivalenz in zwei Schritten und beginnen mit dem Beweis, dass jede μ-rekursive Funktion auch turingberechenbar ist.

Theorem 4. *Jede μ-rekursive Funktion ist turingberechenbar, d.h. es gilt*

$$\mathcal{F}_\mu^{par} \subseteq \mathcal{F}_{\mathrm{TM}}^{par} \text{ und } \mathcal{F}_\mu^{tot} \subseteq \mathcal{F}_{\mathrm{TM}}^{tot}.$$

Beweis. Zunächst überlegen wir uns in Lemma 3, dass alle Grundfunktionen turingberechenbar sind. Anschließend zeigen wir mit je einem Lemma, dass wir auch die Operationen Substitution, primitive Rekursion und die Anwendung des μ-Operators durch Turingmaschinen realisieren können. □

Lemma 3. *Die konstanten Funktionen c_i^r, die Projektionen P_i^r und die Nachfolgefunktion N sind turingberechenbar.*

Dass die Grundfunktionen von $\mathcal{P}$ turingberechenbar sind, kann durch Angabe der jeweiligen Turingmaschine leicht gezeigt werden. Für die Nachfolgefunktion haben wir diese bereits in Beispiel 3 konstruiert. Die Konstruktion von Turingmaschinen zur Berechnung der anderen Funktionen ist eine Übungsaufgabe.
Etwas aufwändiger ist die Konstruktion von Turingmaschinen, welche die Operationen Substitution, primitive Rekursion und die Anwendung des μ-Operators simulieren. Hervorzuheben ist, dass jede dieser Turingmaschinen mit einer sehr kleinen (d.h. zwei, drei oder vier) Anzahl an zusätzlichen Halbbändern auskommt.

Lemma 4 (Substitution)**.** *Sei* $h : \mathbb{N}_0^m \to \mathbb{N}_0, \mathfrak{x} \mapsto f(g_1(\mathfrak{x}), \ldots, g_r(\mathfrak{x}))$. *Sind* $f, g_1, \ldots g_r$ *durch die* k*-Band Turing-Maschinen* $F, G_1, \ldots G_r$ *berechenbar, dann lässt sich* h *durch eine* $(k+2)$*-Band Turingmaschine* H *berechnen.*

Beweis. Für den Beweis beschreiben wird schematisch das Programm der Turingmaschine H. Zunächst beschreiben wir zu jeder Turing-Maschine G_i eine Maschine G_i':

G_i': Band 3 := Band 1;
lasse G_i auf den Bändern $3, 4, \ldots k+2$ laufen;
für $i = 1$: Band 2 := Band 3;
für $2 \leq i \leq r$: Band 2 := Band 2# Band 3;
lösche Band 3;
(alle Köpfe nach vorne;)

Damit können wir nun die Turingmaschine H beschreiben:

H: $G_1'; G_2'; \ldots; G_r'$;
lösche Band 1; (auf Band 2 ist jetzt $g_1(\mathfrak{x})\#g_2(\mathfrak{x})\# \ldots \#g_r(\mathfrak{x})$)
lasse F auf den Bändern $2, 3, \ldots, k+1$ laufen;
Band 1:= Band 2;
lösche Band 2;
(alle Köpfe nach vorne;)

Dass zwei zusätzliche Bänder zur Verfügung stehen, haben wir also folgendermaßen genutzt. Auf einem Band wird zunächst die ursprüngliche Eingabe während der Simulation der Turingmaschinen G_i' unverändert gespeichert. Auf dem zweiten Band wird derweil die Ausgabe aller G_i's zusammengesetzt. Dies stellt dann wiederum die Eingabe für die Turingmaschine F dar. Neben diesen beiden Bändern sind natürlich k-Halbbänder nötig gewesen, um F und die G_i''s auszuführen. Mit zwei zusätzlichen Bändern kommen wir also bequem aus. □

Lemma 5 (primitive Rekursion)**.** *Sei* $h : \mathbb{N}_0^{r+1} \to \mathbb{N}_0$ *mit*

$$\begin{array}{rcl} (0, \mathfrak{x}) & \mapsto & g(\mathfrak{x}), \\ (n+1, \mathfrak{x}) & \mapsto & f(n, h(n, \mathfrak{x}), \mathfrak{x}), \end{array}$$

und seinen G, F zwei k-Band Turingmaschinen zu Berechnung von g und f. Dann gibt es eine $(k+4)$-Band Turingmaschine H zur Berechnung von h.

Beweis. Wieder beweisen wir die Aussage durch Beschreibung des Programms der Turingmaschine H. Diesmal verwenden wir zwei der vier zusätzlichen Bänder (Band 1 und Band 4) als Zähler. Auf Band 1 zählen wir die Anzahl der noch durchzuführenden Berechnungen von f, während auf Band 4 die Anzahl der bereits durchgeführten Berechnungen von f steht. Band 2 dient zur Speicherung der Eingabe $\mathfrak{x}$. Auf Band 3 speichern wir schließlich das bisher berechnete (Teil-)Ergebnis, welches *bottom-up* berechnet wird:

H: kopiere den hinter dem ersten # stehenden Inhalt von Band 1 nach Band 2;
lösche diesen Inhalt auf Band 1;
Band 3 := Band 2;
lasse G auf den Bändern $3, 4 \ldots, k+2$ laufen;
Band 4 := 0;
WHILE Band 1 $\neq$ 0
DO
Band 5 := Band 4 # Band 3 # Band 2;
lasse F auf den Bändern $5, 6, \ldots, k+4$ laufen;
Band 3 := Band 5;
lösche Band 5;
Band 1 := Band 1 - 1;
Band 4 := Band 4 + 1;
OD
Band 1 := Band 3;
lösche Bänder 2,3,4;
(alle Köpfe nach vorne;)

Bemerkenswert ist die Tatsache, dass die **while**-Schleife im Programm von H auch durch eine Zählschleife ersetzen werden kann. Dies bedeutet, dass primitiv rekursive Funktionen generell mit Zählschleifen auskommen. (Das kann bspw. im Beweis, dass die Klasse der sogenannten *LOOP*-Programme und $\mathcal{P}$ äquivalent sind, verwendet werden.) □

Lemma 6 ($\mu-$Operator)**.** *Sei $h = \mu f : \mathbb{N}_0^r \to \mathbb{N}_0$ mit*

$$\mathfrak{x} \mapsto \begin{cases} \textit{das kleinste } n \textit{ mit } f(n, \mathfrak{x}) = 0 \textit{ und} & \\ f(0, \mathfrak{x}), \ldots, f(n-1, \mathfrak{x}) \textit{ ist definiert} & \textit{, falls } n \textit{ existiert} \\ \textit{undefiniert} & \textit{, sonst.} \end{cases}$$

Sei F eine k-Band Turingmaschine zur Berechnung von f. Dann existiert eine $(k+3)$-Band Turingmaschine H zur Berechnung von h.

Beweis. Zum Beweis geben wir wieder das Programm der Turingmaschine H an und verwenden die zusätzlichen Bänder wie folgt. Auf Band 2 wird

die Eingabe $\mathfrak{x}$ dauerhaft gespeichert. Band 3 zählt beginnend bei 0 bis n, falls ein solches existiert. Auf Band 1 speichern wir stets das Ergebnis der Simulation von F und prüfen in einer while-Schleife, ob es bereits 0 wurde.

```
H:  Band 2 := Band 1;
    Band 1 := 1;
    Band 3 := 0;
    WHILE Band 1 ≠ 0
      DO
            Band 4 := Band 3 # Band 2;
            lasse F auf den Bändern 4, 5, ..., k + 3 laufen;
            Band 1 := Band 4; (Ergebnis)
            lösche Band 4;
            Band 3 := Band 3 + 1;
      OD
    Band 1 := Band 3 - 1;
    lösche Band 3;
    (alle Köpfe nach vorne;)
```

Im Beweis wird deutlich, dass die while-Schleife nicht einfach durch eine Zählschleife ersetzt werden kann. Tatsächlich entspricht die Anwendung des μ-Operators der (potentiell unendlich langen) Durchführung einer while-Schleife. (Tatsächlich lässt sich beweisen, dass die Klasse der sogenannten *WHILE*-Programme und $\mathcal{F}_{\mu}^{tot}$ äquivalent sind. Später werden wir mit Korollar 1 sogar zeigen, dass jede beliebige Turingmaschine mit einer einzigen while-Schleife auskommt.) □

Wir haben nun gezeigt, dass jede μ-rekursive Funktion auch turingberechenbar ist. Dass umgekehrt auch die Rechengänge einer Turingmaschine durch μ-rekursive Funktionen simuliert werden können ist schwieriger zu zeigen. Zur Vorbereitung und Vereinfachung zeigen wir zunächst, dass jede $k-$Band Turingmaschine streng genommen mit nur einem einzigen Band auskommt.

Lemma 7. *Sei $f : \mathbb{N}_0^r \to \mathbb{N}_0 \in \mathcal{F}_{\mathrm{TM}}^{par}$ auf einer k-Band Turingmaschine berechenbar. Dann lässt sich f auch auf einer 1-Band Turingmaschine $\tilde{F}$ berechnen.*

Beweis. Um die Arbeit von F mit nur einem Band zu simulieren teilen wir das Halbband von $\tilde{F}$ in $2k$ “Spuren” ein, wie Abbildung 2.3 verdeutlicht. Je zwei Spuren von $\tilde{F}$ dienen zur Simulation eines Bandes von F. Eine davon speichert die Position des Kopfes, die andere den Bandinhalt des entsprechenden Bands von F. $\tilde{F}$ simuliert nun die Rechengänge von F wie folgt:

- Suche nach den k Feldern, in denen $\downarrow$ vorkommt und merke die zugehörigen Bandinhalte im Zustand.

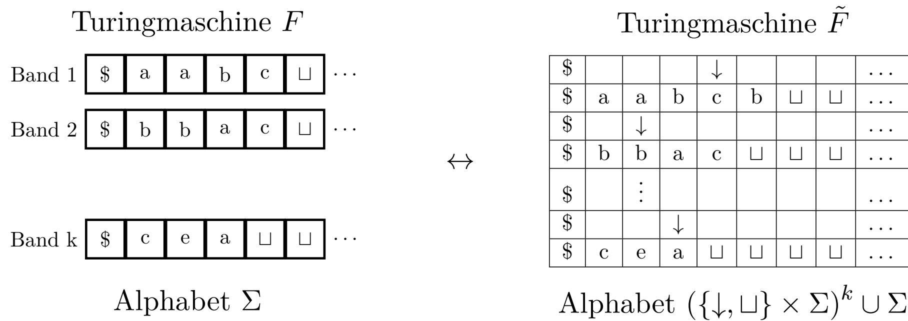


Abbildung 2.3: Transformation der k-Band DTM F zu 1-Band DTM $\tilde{F}$.

- Verwende Übergangsfunktion δ von F, um den neuen Zustand, die neuen Bandsymbole und die Bewegungen der k Köpfe zu berechnen.
- Besuche nochmals alle Felder mit $\downarrow$ und ändere die Bandinschrift und die Kopfmarkierung entsprechend.

Sobald die Simulation der Berechnungen von F auf den k Bändern beendet ist, steht die Ausgabe in der zweiten Spur. Deshalb muss abschließend der Inhalt für jedes Feld rechts von \$ durch den Inhalt der zweiten Spur ersetzt werden. Dann kann $\tilde{F}$ den Kopf nach vorne bewegen und in einen Endzustand wechseln. □

Nun stellt sich die Frage, wie viel Mehraufwand die 1-Band Turingmaschine für diese Simulation betreiben muss. Erstaunlicherweise ist dieser Mehraufwand beschränkt, wie folgendes Lemma zeigt.

Lemma 8. *Die Anzahl der Schritte von $\tilde{F}$ ist quadratisch in der Anzahl der Schritte von F.*

Beweis. Um den i-ten Schritt von F zu simulieren, muss $\tilde{F}$ bis zur Position des am weitesten rechts stehenden Kopfes von F vorlaufen. Da F anfangs ganz links startet, können die Köpfe im i-ten Schritt höchstens i Felder nach rechts gewandert sein. □

Wir haben immer noch nicht die Äquivalenz der Klasse der μ-rekursiven Funktionen und der turingberechenbaren Funktionen gezeigt. Dazu fehlt uns noch folgende Beziehung.

Theorem 5. *Jede turingberechenbare Funktion ist μ-rekursiv, d.h. es gilt*

$$\mathcal{F}_{\mathrm{TM}}^{par} \subseteq \mathcal{F}_{\mu}^{par} \text{ bzw. } \mathcal{F}_{\mathrm{TM}}^{tot} \subseteq \mathcal{F}_{\mu}^{tot}.$$

Bei dem Beweis des Theorems wissen wir bereits wegen Lemma 7, dass jede turingberechenbare Funktion durch eine 1-Band Turingmaschine berechnet

wird. Die Konfiguration einer 1-Band Turingmaschinen können wir auch einfach als Zeichenkette

$$K = \$a_1 \dots a_{k-1}\ q\ a_k \dots a_t$$

schreiben. Dabei entsprechen die a_i der Bandinschrift, q dem Zustand und a_k die Position des L/S-Kopfes. Wir nennen K genau dann eine *Endkonfiguration*, wenn q ein akzeptierender Zustand ist, d.h. wenn $q \in F$. Andernfalls gibt es für K eine eindeutige *Nachfolgekonfiguration*, die wir mit $\Delta(K)$ bezeichnen. Ist beispielsweise $\delta(q, a_k) = (q', a'_k, +1)$, so können wir die Nachfolgekonfiguration durch

$$\begin{array}{ccc} K & \overset{\Delta}{\longmapsto} & K' \\ \| & & \| \\ \$a_1 \dots a_{k-1}\ q\ a_k \dots a_t & & \$a_1 \dots a_{k-1} a'_k\ q'\ a_{k+1} \dots a_t \end{array}$$

leicht bestimmen. Der Beweis von Theorem 5 stellt uns jedoch vor eine Schwierigkeit, denn Turingmaschinen operieren auf Zeichenketten, während μ-rekursive Funktionen auf Zahlen operieren. Wir gehen daher in zwei Schritten vor:

(i) Konfigurationen (=Zeichenketten) durch Zahlen darstellen und

(ii) Übergangsfunktion Δ durch die Funktion $\tilde{\Delta}$ auf Zahlen simulieren.

Dies wird uns ermöglichen für jede turingberechenbare Funktion eine äquivalente μ-rekursive Funktion anzugeben, wie in Abbildung 2.4 skizziert.

Zu (i) Um die Konfiguration einer Turingmaschine als Zahl zu repräsentieren, wählen wir die folgende injektive Abbildung $\Psi : (Q \cup \Sigma)^* \to \mathbb{N}_0$. Für $Q \cup \Sigma = \{b_1, b_2, \dots, b_p\}$ ist (die $(p+1)$-adische Zahldarstellung)

$$\begin{array}{rcl} \epsilon & \longmapsto & 0 \\ b_i & \longmapsto & i \\ v_1 \dots v_s & \longmapsto & \sum_{j=1}^{s} \Psi(v_j)(p+1)^{s-j}, \end{array}$$

für $v_1, \dots, v_s \in Q \cup \Sigma$.

Zu (ii) Zentral ist die Konstruktion der Funktion $\tilde{\Delta}$, die für eine beliebige Kodierung die Nachfolgekodierung bestimmt. Um für eine beliebige Zahl feststellen zu können, ob sie der Kodierung einer Endkonfiguration entspricht, benötigen wir eine weitere Funktion. Das folgende Lemma liefert uns beide Funktionen.

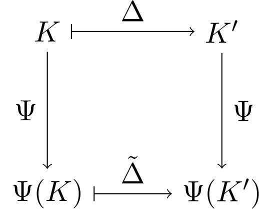


Abbildung 2.4: $\tilde{\Delta}$ macht das Diagramm kommutativ.

Lemma 9. *Zu einer Turingmaschine M gibt es primitiv rekursive Funktionen* $\tilde{\Delta}$ *und* END *mit*

$$\tilde{\Delta}(x) = \begin{cases} \Psi(K'), & \text{falls } x = \Psi(K) \text{ und } K' \text{ die Folgekonfiguration von } K \text{ ist} \\ 0, & \text{sonst .} \end{cases}$$

$$\mathrm{END}(x) = \begin{cases} 0, & \text{falls } x = \Psi(K) \text{ und } K \text{ Endkonfiguration ist} \\ 1, & \text{sonst.} \end{cases}$$

Das heißt, $\tilde{\Delta}$ *macht das Diagramm in Abbildung 2.4 kommutativ.*

Bevor wir Lemma 9 beweisen, fassen wir nochmal alle Schritte zusammen und führen den Beweis von Theorem 5 zu Ende.

Beweis von Theorem 5. Sei $f \in \mathcal{F}_{\mathrm{TM}}^{tot}$ eine r-stellige Funktion. Dann folgt aus Lemma 7, dass es eine 1-Band Turingmaschine $M = (Q, \Sigma, \delta, q_0, F)$ gibt, die f berechnet. Unter Verwendung der Übergangsfunktion $\tilde{\Delta}$ konstruieren wir mittels primitiver Rekursion folgende Funktion $D : \mathbb{N}_0^2 \to \mathbb{N}_0$, wobei

$$\begin{aligned} D(0, x) &:= x \\ D(n+1, x) &:= \tilde{\Delta}(D(n, x)) \end{aligned}$$

für $n \geq 0$. Folglich ist D primitiv rekursiv. Die Funktion liefert uns die Kodierung der Konfiguration K', die nach genau n Schritten aus der Konfiguration K entsteht:

$$D(n, \Psi(K)) = \begin{cases} \text{Kodierung } \Psi(K') \text{ derjenigen Konfiguration K', die entsteht, wenn M in K startet und n Rechenschritt macht,} & \text{falls das in K geht,} \\ 0, & \text{sonst.} \end{cases}$$

Mit Hilfe von D können wir in den Folgekonfigurationen von K nach der *ersten* auftretenden Endkonfiguration *suchen.* Dies können wir mit der μ-

rekursiven Funktion $A : \mathbb{N}_0 \to \mathbb{N}_0$

$$A(\Psi(K)) = \begin{cases} \text{(minimale) Anzahl an Rechenschritten, die } M \text{ gestartet in Konfiguration } K \text{ macht, bis sie in Endkonfiguration gerät,} & \text{falls sie das tut, d.h. hält} \\ \text{undefiniert} & \text{sonst, d.h. sie hält nicht.} \end{cases}$$

A ist μ-rekursiv für $A(x) := \mu g(x)$ mit $g(n,x) := \text{END}(D(n,x))$. Wir nehmen an, dass M angesetzt auf Konfiguration K nach $A(\Psi(K))$ Schritten in Endkonfiguration $K' = \$q\,\text{bin}(f(\mathfrak{x}))$ hält. Dann benötigen wir noch eine Funktion F, die uns aus der Endkonfiguration den Funktionswert $f(\mathfrak{x})$ extrahiert. Außerdem muss auch die Startkonfiguration $\$q_0\,\text{bin}(x_1)\#bin(x_2) \ldots \#\,\text{bin}(x_r)$ mit $\mathfrak{x} = (x_1, \ldots, x_r)$ durch eine Funktion E kodiert werden. Dass es diese primitiv rekursiven Funktionen gibt, zeigt das folgende (technische) Lemma, welches wir erst später beweisen.

Lemma 10. *Es gibt primitiv rekursive Funktionen E, F*

$$E(x_1, x_2, \ldots x_r) = \Psi\left(\$q_0\,\text{bin}(x_1)\#\,\text{bin}(x_2)\#\ldots\#\,\text{bin}(x_r)\right) \text{ und}$$

$$F\Big(\Psi(\$q\,\text{bin}(x))\Big) = x.$$

Für die Funktion f wissen wir also, dass bei einer Eingabe gilt

$$\begin{aligned} f(x_1, \ldots, x_r) \text{ ist definiert} \quad &\Leftrightarrow \quad M \text{, angesetzt auf Konfiguration } \$q_0\,\text{bin}(x_1)\#\ldots\#\,\text{bin}(x_r) \text{, hält in Endzustand} \\ &\Leftrightarrow \quad A\Big(E(x_1, \ldots x_r)\Big) \text{ ist definiert.} \end{aligned}$$

Mit Lemma 10 gilt in diesem Fall:

$$\begin{aligned} f(x_1, \ldots x_r) &= F\Big(\Psi(\text{Endkonfiguration})\Big) \\ &= F\Bigg(D\Big(\underbrace{A(E(x_1, \ldots, x_r))}_{\text{Anzahl der Rechenschritte.}}, \underbrace{E(x_1, \ldots, x_r)}_{\Psi(\text{Startkonfiguration})}\Big)\Bigg) \end{aligned}$$

Damit ist f μ-rekursiv, womit Theorem 5 bewiesen wäre. □

Nun müssen noch die Beweise von Lemma 9 und Lemma 10 nachgeholt werden. Zur Konstruktion dieser primitiv rekursiven Funktionen benötigen wir ein paar Hilfsmittel. Wir simulieren einfache Operationen wie Längenbestimmung, Konkatenation, Präfixbildung, usw. auf $(Q \cup \Sigma)^*$ durch primitiv rekursive Funktionen.

Lemma 11. *Es gibt primitiv-rekursive Funktionen,*

$$\mathrm{L}, \textsc{Concat}, \textsc{Prefix}, \textsc{Suffix}, \textsc{First}, \textsc{Last}, \textsc{Select}$$

sodass für alle $b, c \in (Q \cup \Sigma)^*$ *und alle* i *mit* $0 \leq i \leq |B|$ *gilt:*

$$\begin{array}{lcl}
L\Big(\Psi(b)\Big) & = & |b| \\
\textsc{Concat}\Big(\Psi(b), \Psi(c)\Big) & = & \Psi(bc) \\
\textsc{Prefix}\Big(\Psi(b), i\Big) & = & \begin{cases} \Psi(b_1 b_2 \ldots b_i), & \textit{falls } i > 0 \\ \Psi(\epsilon), & \textit{falls } i = 0 \end{cases} \\
\textsc{Suffix}\Big(\Psi(b), i\Big) & = & \begin{cases} \Psi(b_1 b_2 \ldots b_i), & \textit{falls } i > 0 \\ \Psi(\epsilon), & \textit{falls } i = 0 \end{cases} \\
\textsc{First}\Big(\Psi(b)\Big) & = & \Psi(b_1) \\
\textsc{Last}\Big(\Psi(b)\Big) & = & \Psi(b_{|b|}) \\
\textsc{Select}\Big(\Psi(b), i\Big) & = & \begin{cases} \Psi(b_i), & \textit{falls } i > 0 \\ \Psi(\epsilon), & \textit{falls } i = 0 \end{cases}
\end{array}$$

Beweis. Zum Beweis geben wir explizit die Funktionen an:

$$\begin{array}{rcl}
\mathrm{L}(x) & := & \min\{m; m \leq x \text{ und } (p+1)^m > x\} \\
& = & \text{niedrigste Potenz von } p+1 \text{ die in x nicht vorkommt}
\end{array}$$

Denn sei $Q = \{(x, m); (p+1)^m \leq x\}$, dann ist $\mathrm{L}(x) = h(x, x) + 1$ für $h(z, x) = \mu_b \chi_Q(z, x) =$ kleinstes $m \leq x : \chi_Q(z, m) = 0$.

$$\begin{array}{rl}
\textsc{Concat}(x, y) = & x(p+1)^{\mathrm{L}(y)} + y \\
\textsc{Prefix}(x, i) = & x \text{ div } (p+1)^{\mathrm{L}(x)-i}
\end{array}$$

Denn $\Psi(b_1 \ldots b_{|b|}) = \Psi(b_1)(p+1)^{|b|-1} + \Psi(b_2)(p+1)^{|b|-2} + \cdots + \Psi(b_i)(p+1)^{|b|-i}$ + niedrigere Terme, wobei $|b| = L(\Psi(b))$ ist.

$$\textsc{Suffix}(x, i) = \begin{cases} x \mod (p+1)^{\mathrm{L}(x)-i+1}, & \text{falls } i > 0 \\ 0, & \text{falls } i = 0 \end{cases}$$

Denn $\Psi(b_1 \ldots b_{|b|}) =$ höhere Terme $+\Psi(b_{i-1})(p+1)^{|b|-i+1} + \Psi(b_i)(p+1)^{|b|-i} +$

$\Psi(b_{i+1})(p+1)^{|b|-i-1} + \cdots + \Psi(b_{|b|})$

$$\begin{aligned}
\textsc{First}(x) &= \textsc{Prefix}(x,1)\\
\textsc{Last}(x) &= \textsc{Suffix}(x,\mathrm{L}(x))\\
\textsc{Select}(x,i) &= \begin{cases}\textsc{First}(\textsc{Suffix}(x,i)), & \text{falls } i>0\\ 0 & \text{falls } i=0\end{cases}
\end{aligned}$$

□

Im Folgenden werden wir einige Abkürzungen verwenden:

$$\begin{aligned}
x_{(i)} &= \textsc{Select}(x,i)\\
[x,y] &= \textsc{Concat}(x,y)\\
[x,y,z] &= \textsc{Concat}((x,y),z)
\end{aligned}$$

Nun holen wir die Beweise von Lemma 9 und Lemma 10 nach. Wir beginnen mit dem Beweis von Lemma 9:

Beweis zu Lemma 9. Zu zeigen ist, dass es Funktionen $\tilde{\Delta}, \mathrm{END} \in \mathcal{P}$ gibt, sodass

1. das Diagramm in Abbildung 2.4 kommutativ ist und
2. $\mathrm{END}\Big(\Psi(K)\Big) = \begin{cases} 0, & \text{falls K Endkonfiguration von M}\\ 1, & \text{sonst.}\end{cases}$

Da die Menge Q der Zustände der Turingmaschine M endlich ist, so ist auch $\Psi_Q := \{\Psi(q), q \in Q\}$ endlich und nach Theorem 1 folgt, dass die charakteristische Funktion χ_{Ψ_Q} primitiv rekursiv. Damit können wir in einer Konfiguration der Turingmaschine nach der Position des Kopfes suchen. Wir setzen dazu $q(x) := \min\{i; i \leq \mathrm{L}(x) \text{ und } x_{(i)} \in \Psi_Q\}$. Dann ist q nach Theorem 1 primitiv rekursiv und wenn x die Kodierung $\Psi(K)$ der Konfiguration $K = \$a_1 \ldots a_{k-1} \underset{\uparrow}{q}\, a_k \ldots a_t$ ist, dann ist

$$\begin{aligned}
q\Big(\Psi(K)\Big) &= k+1\\
&= \text{die Stellung des L/S-Kopfs auf dem Band.}
\end{aligned}$$

Mit den primitiv rekursiven Stringfunktionen aus Lemma 11 können wir uns nun primitiv rekursive Funktionen definieren, die eine Kodierung x in drei Teile $u(x)\ w(x)\ v(x)$ zerlegen:

$$\begin{aligned}
u(x) &:= \mathrm{PREFIX}(x, q(x) \dot{-} 2)\\
v(x) &:= \mathrm{SUFFIX}(x, q(x)+2)\\
w(x) &:= \Big[x_{(q(x)\dot{-}1)}, x_{(q(x))}, q_{(q(x)+1)}\Big].
\end{aligned}$$

Diese Teile sind die interessanten Stellen, da wir auf dem Teil $w(x)$ (lokal) die Folgekonfiguration berechnen können. Dabei verwenden wir die Eigenschaften der Turingmaschine M. Sei beispielsweise $K = \$a_1 \dots a_{k-1}qa_ka_{k+1}\dots a_t$ eine Konfiguration der Turingmaschine M. Dann zerlegen wir K in

$$\begin{aligned} u\Big(\Psi(K)\Big) &= \Psi(\$a_1 \dots a_{k-2}) \\ w\Big(\Psi(K)\Big) &= \Psi(a_{k-1}\ q\ a_k) \\ v\Big(\Psi(K)\Big) &= \Psi(a_{k+1} \dots a_t) \end{aligned}$$

und berechnen die Folgekonfiguration von $y = w(\Psi(K))$ mit

$$\tilde{\delta}(y) = \begin{cases} \Psi(a_{k-1}a'_kq'), & \text{falls } y = \Psi(a_{k-1}qa_k) \text{ und } \delta(q,a_k) = (q', a'_k, +1) \\ \Psi(a_{k-1}q'q'_k), & \text{falls } y = \Psi(a_{k-1}qa_k) \text{ und } \delta(q,a_k) = (q', a'_k, 0) \\ \Psi(q', a_{k-1}a'_k), & \text{falls } y = \Psi(a_{k-1}qa_k) \text{ und } \delta(q,a_k) = (q', a'_k, -1) \\ 0 & \text{sonst.} \end{cases}$$

Nach Theorem 1 ist $\tilde{\delta}$ primitiv rekursiv, da für a_{k-1}, a_k, q nur endlich viele Möglichkeiten existieren und $\tilde{\delta}$ durch endlich viele Fallunterscheidungen definiert ist. Damit erhalten wir die globale Übergangsfunktion auf ganzen Konfiguration

$$\tilde{\Delta} = \Big[u(x), \tilde{\delta}(w(x)), v(x)\Big].$$

Der Test auf Endkonfiguration ergibt sich durch

$$\mathrm{END}(x) = \begin{cases} 0, & \text{falls } \chi_{(q(x))} \in \Big\{\Psi(e); e \in F\Big\} \\ 1, & \text{sonst.} \end{cases}$$

Beide Funktionen sind somit primitiv rekursiv und leisten das Verlangte. □

Als letztes ist noch der Beweis von Lemma 10 zu zeigen.

Beweis von Lemma 10. Dazu konstruieren wir zwei primitiv rekursive Funktionen E, F, sodass

a) $E(x_1, x_2, \dots x_r) = \Psi\left(\$q_0 \operatorname{bin}(x_1)\# \operatorname{bin}(x_2)\# \dots \# \operatorname{bin}(x_r)\right)$ zur Kodierung der Startkonfiguration und

b) $F\Big(\Psi\left(\$q \operatorname{bin}(y)\right)\Big) = y$ zum Auslesen des Funktionswerts in der Endkonfiguration

verwendet werden kann. Um Teil a) zu zeigen konstruieren wir zunächst eine primitiv rekursive Funktion $B : \mathbb{N}_0^2 \to \mathbb{N}_0$, die für $i, x \in \mathbb{N}_0$ das i-letzte Bit in der Binärdarstellung von x berechnet.

$$\begin{aligned} B(i,x) &= (x \mod 2^i) \text{ div } 2^{i-1} \\ x &= w_1 2^{s-1} + w_2 2^{s-2} + \cdots + w_{s-i}2^i + w_{s-i+1}2^{i-1} \\ &\quad + w_{s-i+2}2^{i-2} + \cdots + w_{s-1}s^1 + w_s \\ P(n,x) &= \sum_{i=1}^{n} \Psi\Big(\underbrace{B(i,x)}_{0 \text{ oder } 1}\Big)(p+1)^{i-1} \end{aligned}$$

Sowohl B als auch P sind primitiv rekursiv (Übungsaufgabe) und $P(L(x),x) = \Psi\Big(\text{ bin}(x)\Big)$. Damit erhalten wir E aus P durch endlich viele Anwendungen von Concat

$$E(x_1, \ldots x_r) = [\Psi(\$), \Psi(q_0), P(\text{L}(x_1), x_1), \Psi(\#), \ldots, \Psi(\#), P(\text{L}(x_r), x_r)]\,.$$

Da P, L und Concat primitiv rekursiv sind, gilt dies auch für E.

Nun zu Teil b). Hier müssen wir die primitiv rekursive Funktion F konstruieren, welche die $(p+1)$-adische Darstellung des Ergebnisses aus der Kodierung der Endkonfiguration liest und in die Zahl selbst transformiert. Dazu lesen wir zunächst die $(p+1)$-adische Darstellung aus der Kodierung der Endkonfiguration mit

$$G(x) = \text{Suffix}(x, q(x) + 1).$$

G ist primitiv rekursiv und wie gewünscht ist $G(\Psi(\$q\,\text{bin}(y))) = \Psi(\text{bin}(y))$. Mit der primitiv rekursiven Funktion $H : \mathbb{N}_0 \to \mathbb{N}_0$

$$H(x) = \sum_{i=1}^{\text{L}(x)} \Big(\Big(x \mod (p+1)^i\Big) \text{ div } (p+1)^{i-1} - 1\Big)\, 2^{i-1}$$

gilt schließlich $H(\Psi(\text{bin}(y))) = y$, wenn wir o.E. annehmen, dass $\Psi(0) = 1$ und $\Psi(1) = 2$. Wir können nun $F(x) = H(G(x))$ explizit angeben

$$F\Big(\Psi(\$q\,\text{bin}(y))\Big) = H\Big(\Psi(\,\text{bin}(y))\Big) = y.$$

Nach Konstruktion ist F primitiv rekursiv und leistet das Verlangte. □

Damit ist schließlich der Beweis vollständig erbracht, dass die Klasse der μ-rekursiven Funktionen und die Klasse der turingberechenbaren Funktionen äquivalent sind. Dies interpretieren wir als Indiz für die Gültigkeit der Church-Turing-These. Wir sehen uns nun weitere Anwendungen an.

2.3.2 Folgerungen und Ausblick

Die Ackermannfunktion ist μ-rekursiv

In Abschnitt 2.2 haben wir noch nicht bewiesen, dass die Ackermannfunktion tatsächlich μ-rekursiv ist. Dies können wir nun sehr einfach nachholen. Die Ackermannfunktion war wie folgt definiert.

$$A(0,y) = y+1 \tag{2.1}$$

$$A(x+1,0) = A(x,1) \tag{2.2}$$

$$A(x+1,y+1) = A(x,A(x+1,y)) \tag{2.3}$$

Es reicht nun zu beweisen, dass die Ackermannfunktion turingberechenbar ist. Dazu schreiben wir $A(x,y)$ zunächst als Zeichenfolge:

$$\underbrace{1\dots1}_{x\text{-mal}}\ A\ \underbrace{1\dots1}_{y\text{-mal}}$$

Wir beschreiben nun eine Turing-Maschine M, die A berechnet:

Vorbereitung Wir übersetzen die Binärdarstellung von x und y, trennen x und y durch das Zeichen A und setzen den Lese-/Schreibkopf darauf.

$$\$\overset{\downarrow}{\mathrm{bin}}(x)\#\mathrm{bin}(y)\sqcup\dots \Rightarrow \$\underbrace{1\dots1}_{x}\overset{\downarrow}{A}\underbrace{1\dots1}_{y}\sqcup\dots$$

Solange ein A auf dem Band steht, führt M folgende Makros aus, abhängig von der Umgebung des am weitesten rechts stehenden A.

(2.1) Falls links vom rechtesten A ein \$ steht:

$$\$\overset{\downarrow}{A}\underbrace{1\dots1}_{y}\sqcup\dots \quad\Rightarrow\quad \$\overset{\downarrow}{\underbrace{11\dots1}_{y+1}}\sqcup\dots$$

und wir sind fertig. Falls links vom rechtesten A ein weiteres A steht:

$$\dots A\overset{\downarrow}{A}\underbrace{1\dots1}_{y}\sqcup\dots \quad\Rightarrow\quad \dots\overset{\downarrow}{A}\underbrace{11\dots1}_{y+1}\sqcup\dots$$

(2.2) Falls links vom rechtesten A eine 1 steht und rechts $\sqcup$:

$$\dots1\overset{\downarrow}{A}\sqcup\dots \quad\Rightarrow\quad \dots\overset{\downarrow}{A}1\sqcup\dots$$

(2.3) Falls links und rechts vom rechtesten A eine 1 steht:

$$\dots\underbrace{L}_{\in\{A,\$\}}\underbrace{1\dots11}_{x+1}\overset{\downarrow}{A}\underbrace{1\dots11}_{y+1}\sqcup\dots \quad\Rightarrow\quad \dots A\underbrace{1\dots1}_{x}A\underbrace{1\dots11}_{x+1}\overset{\downarrow}{A}\underbrace{1\dots1}_{y}\sqcup\dots$$

Die $y+1$ Zeichen werden also weit genug nach rechts verschoben (kopiert) um Platz für eine Kopie von $x+1$ zu schaffen.

Schließlich wird das Ergebnis wieder in die binäre Darstellung umgewandelt:

$$\$\overset{\downarrow}{1}\ldots 1\sqcup\ldots \Rightarrow \$\mathrm{bin}\left(A\left(x,y\right)\right)\sqcup\ldots$$

Beispiel $A\left(1,2\right) = ?$

Band	Makro
$\$1\overset{\downarrow}{A}11\sqcup\ldots$	(initial)
$\$A1\overset{\downarrow}{A}1\sqcup\ldots$	(2.3)
$\$AA1\overset{\downarrow}{A}\sqcup\ldots$	(2.3)
$\$AA\overset{\downarrow}{A}1\sqcup\ldots$	(2.2)
$\$A\overset{\downarrow}{A}11\sqcup\ldots$	(2.1)
$\$\overset{\downarrow}{A}111\sqcup\ldots$	(2.1)
$\$\overset{\downarrow}{1}111\sqcup\ldots \Rightarrow A\left(1,2\right) = 4$	(2.1)

Einmalige Anwendung des μ-Operators

Interessanterweise lässt sich zeigen, dass die einmalige Anwendung des μ-Operators ausreichend ist, um eine beliebige μ-rekursive Funktion zu definieren. Zunächst müssen wir definieren, wann zwei (partielle) Funktionen gleich heißen sollen.

Definition 7 (Gleichheit von Funktionen)**.** Zwei r-stellige, partielle, μ-rekursive Funktionen $f^{(r)}, g^{(r)} \in \mathcal{F}_{\mu}^{par}$ heißen *gleich*, wenn sie den selben Definitionsbereich $D = D\left(f\right) = D\left(g\right)$ besitzen und für alle $\mathfrak{x} \in D$ gilt, dass $f\left(\mathfrak{x}\right) = g\left(\mathfrak{x}\right)$. Wir schreiben dann $f \mathrel{\tilde{=}} g$.

Aus dem Beweis von Theorem 5 folgern wir nun, dass jede beliebige μ-rekursive Funktion durch einmalige Anwendung des μ-Operators dargestellt werden kann. Eine solche Darstellung nennen wir Kleene'sche Normalform.

Korollar 1 (Kleene'sche Normalform)**.** *Zu jedem $f^{(r)} \in \mathcal{F}_{\mu}^{(par)}$ gibt es primitiv rekursive Funktionen $p^{(r+1)}, q^{(r+1)}$, sodass*

$$\forall \mathfrak{x} : f\left(\mathfrak{x}\right) \mathrel{\tilde{=}} q\left(\mu p\left(\mathfrak{x}\right), \mathfrak{x}\right).$$

Beweis. Im Beweis von Theorem 5 haben wir gezeigt, dass wir zu jeder turingberechenbaren Funktion f eine äquivalente μ-rekursive Funktion konstruieren können. Dazu haben wir Funktionen F, D, A, E konstruiert, sodass

$$f\left(\mathfrak{x}\right) = F\left(D\left(A\left(E\left(\mathfrak{x}\right)\right), E\left(\mathfrak{x}\right)\right)\right).$$

Dabei waren F, D, E primitiv rekursive Funktionen. Lediglich zur Definition der Funktion $A\left(x\right) = \mu g\left(x\right)$ haben wir den μ-Operator einmalig auf die primitiv rekursive Funktion $g(n,x) = \mathrm{END}(D(n,x))$ angewendet. □

Dass die Anwendung des μ-Operators der Durchführung von while-Schleifen entspricht, haben wir bereits im Beweis von Lemma 6 gesehen. Folglich impliziert das obige Korollar, dass es für jede turingberechenbare Funktion eine Turingmaschine gibt, die mit einer einzigen while-Schleife auskommt. Auch beim Programmieren käme man deshalb rein theoretisch mit einer einzigen while-Schleife aus.

Normierte Registermaschinen

Abgesehen von Turingmaschinen, wurden noch viele andere Maschinenmodelle eingeführt und untersucht. So zum Beispiel *normierte Registermaschinen*. Eine solche besteht aus m Registern, in welchen jeweils eine natürliche Zahl gespeichert werden kann. Folgende Operationen stehen zur Verfügung:

$$\begin{array}{rcl} a_i & : & \text{addiere 1 im } i-\text{ten Register} \\ s_i & : & \text{subtrahiere 1 im } i-\text{ten Register} \\ (M)_i & : & \text{iteriere } M \text{ so oft, bis im } i-\text{ten Register 0 steht} \\ M_1 M_2 & : & \text{führe nacheinander } M_1, M_2 \text{ aus} \end{array}$$

Beispiel 4. Das Programm $(s_1 a_2 a_2)_1\,(s_2 a_1)_2$ führt auf der Eingabe $(x, 0, \ldots)$ (also x in Register 1 und 0 in Register 2) zu der Ausgabe $(2x, 0, \ldots)$.

Auch für normierte Registermaschinen kann man beweisen, dass die Klasse der zugehörigen berechenbaren Funktionen

$$\mathcal{F}_{\text{NRM}} := \{f : f \text{ berechenbar durch norm. Registermaschine}\}$$

mit der Klasse der μ-rekursiven (bzw. turingberechenbaren) Funktionen übereinstimmt, d.h. $\mathcal{F}_{\text{NRM}} = \mathcal{F}_{\mu}^{tot}$. Man erhält also eine weitere Stütze für die These von Church-Turing.

Universelle Turingmaschinen

Mit dem Normalformtheorem (Korollar 1) haben wir gezeigt, dass es zu jeder μ-rekursiven Funktion f zwei primitiv rekursive Funktionen p, q gibt, sodass f und $q(\mu p(\mathfrak{x}), \mathfrak{x})$ gleich sind, d.h.

$$\forall \mathfrak{x} : \; f(\mathfrak{x}) \;\tilde{=}\; q(\mu p(\mathfrak{x}), \mathfrak{x}).$$

Hierbei hingen die Funktionen p und q jedoch von der Funktion f (bzw. von der Turingmaschine M, die f berechnet) ab. Statt die Struktur von M in die Funktionen p und q „einzubauen“ könnte man auch eine spezielle Kodierung $k = \langle M \rangle$ (Gödelisierung) der Turingmaschine als Argument übergeben. Dieser Trick führt zu einer verstärkten Form des Normalformtheorems.

Theorem 6 (Starke Kleene'sche Normalform)**.** *Es gibt primitiv rekursive Funktionen U, T, φ, so dass zu jeder μ-rekursiven Funktion eine Kodierung k einer Turingmaschine existiert, so dass*

$$\forall \mathfrak{x} : f(\mathfrak{x}) \;\tilde{=}\; U(\mu T(k, \varphi(\mathfrak{x}))).$$

Das Theorem ermöglicht den Bau einer *universellen* Turingmaschine M^*, welche die Funktion

$$(k, \mathfrak{x}) \longmapsto U(\mu T(k, \varphi(\mathfrak{x})))$$

aus $\mathcal{F}_\mu^{par}$ berechnet. Bei Eingabe der *richtigen* Kodierung k kann die universelle Turingmaschine M^* also *jede* Funktion aus $\mathcal{F}_\mu^{tot}$ berechnen. Die Kodierungen (der Turingmaschinen) können als Programme interpretiert werden, welche nun nicht mehr selbst in der Turingmaschine gebunden sind sondern stattdessen Eingabedaten von M^* sind. Die universelle Turingmaschine führt diese Programme aus. Die Idee, Programme als Eingabe zu betrachten, liegt übrigens auch der Von-Neumann-Rechnerarchitektur zu Grunde.

2.4 Entscheidbarkeit

In Abschnitt 2.2 und Abschnitt 2.3 haben wir uns mit Modellen zur Beschreibung von Berechenbarkeit auseinander gesetzt. Nun möchten wir uns überlegen, was mit diesen Modellen prinzipiell ausgerechnet werden kann und wo wir die Grenzen dieser Modelle erreichen.
Für Conways Spiel des Lebens haben wir in Abschnitt 2.1 bereits eine Grenze kennengelernt. Es gibt keinen Algorithmus, der für zwei beliebige Generationen prüft, ob zu irgend einem Zeitpunkt des Spiels die eine aus der anderen hervorgeht. Es gibt jedoch viel einfachere und allgemeinere Probleme für die es keinen Algorithmus gibt. In Abschnitt 2.3 haben wir uns zum Schluss mit universellen Turingmaschinen befasst. Als Eingabe bekam die universelle Turingmaschine M^* ein Tupel $(k, \mathfrak{x})$, wobei k die Kodierung einer beliebigen Turingmaschine M und $\mathfrak{x}$ die Eingabe für M war. Die universelle Turingmaschine simulierte die Berechnungen von M auf der Eingabe $\mathfrak{x}$. Hier stellt sich natürlich die Frage, ob es einen Algorithmus gibt, der für eine beliebige (Kodierung einer) Turingmaschine M und einer beliebigen Eingabe $\mathfrak{x}$ feststellt, ob M bei ihren Berechnungen irgendwann anhält. Diese informelle Problemstellung ist als *Halteproblem* für Turingmaschinen bekannt. Wir werden für dieses Problem formal beweisen, dass es keinen solchen Algorithmus gibt.
Da Turingmaschinen auf beliebigen Zeichenketten operieren, betrachten wir zunächst keine Funktionen sondern Sprachen. Natürlich gilt

$$\begin{aligned} & f : \Sigma^* \to \mathbb{N}_0 \text{ ist turingberechenbar} \\ \Leftrightarrow \quad & \tilde{f} : \mathbb{N}_0 \xrightarrow{\pi^{-1}} \Sigma^* \xrightarrow{f} \mathbb{N}_0 \text{ ist } \mu\text{-rekursiv.} \end{aligned}$$

Dabei bezeichne $\pi : \Sigma^* \to \mathbb{N}_0$ eine geeignete berechenbare Gödelisierung. Für eine Sprache führen wir nun den Begriff der Entscheidbarkeit ein.

Definition 8 (Entscheidbar)**.** Sei Σ ein endliches Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar*, wenn ihre charakteristische Funktion $\chi_L : \Sigma^* \to$

$\{0,1\}$ mit

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{sonst.} \end{cases}$$

turingberechenbar ist. Die Sprache L heißt *semi-entscheidbar*, wenn die partielle Funktion

$$\chi_L^*(w) = \begin{cases} 1, & \text{falls } w \in L \\ \text{undefiniert}, & \text{sonst.} \end{cases}$$

turingberechenbar ist.

Für eine entscheidbare Sprache L können wir also mit einem Verfahren (Algorithmus) in endlich vielen Schritten feststellen, ob ein Wort $w \in \Sigma^*$ in L liegt oder nicht. Wir können also für χ_L eine Turingmaschine M konstruieren und auf w ansetzen. Die Turingmaschine muss nach endlich vielen Schritten halten (da χ_L total) und den Wert 0 oder 1 liefern. Liefert M (bzw. χ_L) den Wert 1, so sagen wir *M akzeptiert w*. In diesem Fall liegt w in L. Andernfalls liefert M die Ausgabe 0, d.h. $w \notin L$, und wir sagen auch *M verwirft w*. Entscheidet eine Turingmaschine M die Sprache L, d.h. sie berechnet χ_L, so schreiben wir auch $L(M) = L$.

2.4.1 Unentscheidbarkeit des Halteproblems

Um das Halteproblem formal zu beschreiben überlegen wir uns zunächst, wie wir eine Turingmaschine M als Eingabe über den Alphabet $\{0,1,\#\}$ ausdrücken können. Diese Darstellung bezeichnen wir als *Kodierung* $\langle M \rangle$ der Turingmaschine. Die Kodierung ist jedoch selbst eine Zeichenkette und daher von einer Gödelnummer zu unterscheiden. Sei $M = (\Sigma, Q, \delta, q_0, F)$ eine Turingmaschine. Wir nummerieren die Elemente von $Q \cup \Sigma$ wie folgt durch

$$\begin{aligned} Q &= \{q_0, q_1, \ldots, q_{p-1}\} && \text{Zustände,} \\ \Sigma &= \{a_p, a_{p+1}, \ldots, a_r\} && \text{Bandalphabet,} \\ F &= \left\{q_{p-|F|}, \ldots, q_{p-1}\right\} && \text{Endzustände } (F \subseteq Q)\,. \end{aligned}$$

Da so Zustände und Zeichen eindeutig einer natürlichen Zahl zugeordnet werden, können wir diese Zahl mit ihrer Binärdarstellung identifizieren und haben sie so mit dem Alphabet $\{0,1,\#\}$ dargestellt. Nun müssen wir nur noch die Übergangsfunktion in geeigneter Weise darstellen. Dazu kodieren wir $\delta(q_i, a_j) = (q_{i'}, a_{j'}, m)$ durch den String

$$\#\#\,\mathrm{bin}\,(i)\,\#\,\mathrm{bin}\,(j)\,\#\,\mathrm{bin}\,(i')\,\#\,\mathrm{bin}\,(j')\,\#\,\mathrm{bin}\,(c)\,,$$

wobei wir die Kopfbewegung $m \in \{-1, 0, +1\}$ ausdrücken durch

$$c = \begin{cases} 0, & \text{falls } m = -1 \\ 1, & \text{falls } m = 0 \\ 2, & \text{falls } m = 1. \end{cases}$$

Insgesamt ergibt sich die Kodierung der Turingmaschine M somit durch

$$\begin{array}{rcll}\langle M\rangle & = & \#\#\#\# & \overbrace{\text{bin}\,(0)\,\#\cdots\#\,\text{bin}\,(p-1)}^{Q} \\ & & \#\#\# & \underbrace{\text{bin}\,(p)\,\#\cdots\#\,\text{bin}\,(r)}_{\Sigma} \\ & & \#\#\# & \underbrace{\text{bin}\,(p-|F|)\,\#\cdots\#\,\text{bin}\,(p-1)}_{F} \\ & & \#\#\# & \underbrace{\delta_1\#\cdots\#\delta_k}_{\delta} \\ & & \#\#\#\# & \end{array}$$

Durch die Kodierung des Alphabets ergibt sich automatisch eine Kodierung der Eingabe $w = a_{i_1} a_{i_2} \cdots a_{i_m}$ durch

$$\langle w\rangle = \text{bin}\,(i_1)\,\#\,\text{bin}\,(i_2)\,\#\cdots\#\,\text{bin}\,(i_m)\,\#\#.$$

Das *allgemeine Halteproblem* für Turingmaschinen ist nun definiert durch die Sprache

$$H := \{x \in \{0,1,\#\}^* \mid x = \langle M\rangle\,\langle w\rangle \text{ und } M \text{ hält bei Eingabe } w\}\,.$$

Nun möchten wir beweisen, dass diese Sprache unentscheidbar ist. Dabei ist leicht zu testen, ob $\langle M\rangle$ die Kodierung einer Turingmaschine ist. Ob diese Turingmaschine jedoch auf der Eingabe w hält, ist unentscheidbar. Um dies zu beweisen betrachten wir zunächst eine eingeschränkte Version des Halteproblems. Dazu betrachten wir diejenigen Turingmaschinen, die halten, wenn sie auf ihre eigene Kodierung als Eingabe angesetzt werden. Es genügt, wenn wir uns auf diesen Spezialfall beschränken.

Theorem 7. *Das eingeschränkte Halteproblem*

$$H_e := \{x \in \{0,1,\#\}^* \mid x = \langle M\rangle \text{ und } M \text{ hält bei Eigabe } x\}$$

ist unentscheidbar.

Beweis. Wir beweisen die Aussage indirekt, indem wir annehmen, dass H_e entscheidbar wäre. Somit existiert eine Turingmaschine M, welche χ_{H_e} berechnet. Nun können wir eine Turingmaschine M' konstruieren, sodass $\forall x \in \{0,1,\#\}^*$ gilt

- M' hält bei Eingabe x, falls $x \notin H_e$,
- M' hält nicht bei Eingabe x, falls $x \in H_e$.

Nun betrachten wir das Verhalten von M' bei Eingabe der eigenen Kodierung $x = \langle M'\rangle$. Dann ist

$$\begin{array}{rcl} M' \text{ hält bei Eingabe } \langle M'\rangle & \Leftrightarrow & \langle M'\rangle \notin H_e \\ & \Leftrightarrow & M' \text{ hält nicht bei Eingabe } \langle M'\rangle \end{array}$$

ein Widerspruch und es folgt, dass H_e nicht entscheidbar ist. □

Aus dem Spezialfall folgt unmittelbar die Unentscheidbarkeit des allgemeinen Halteproblems.

Korollar 2. *Das allgemeine Halteproblem*

$$H := \{x \in \{0,1,\#\}^* \mid x = \langle M\rangle \langle w\rangle \text{ und } M \text{ hält bei Eingabe } w\}$$

ist unentscheidbar.

Beweis. Für die Eingabe $x = \langle M\rangle$ für eine Turingmaschine M gilt

$$x \in H_e \iff M \text{ hält bei Eingabe } \langle M\rangle \iff x' := \underbrace{\langle M\rangle \langle\langle M\rangle\rangle}_{x\langle x\rangle} \in H.$$

Da $x' = x\langle x\rangle$ leicht aus x berechnet werden kann, würde aus der Entscheidbarkeit von von H auch die Entscheidbarkeit von H_e folgen. □

Damit ist ein für die Informatik wichtiges Problem, die vollständige semantische Korrektheit von Programmen, als unentscheidbar erkannt. Es ist ebenfalls unentscheidbar, ob eine beliebige Turingmaschine eine vorgegebene Aufgabe löst. Dies ist Inhalt des folgenden Satzes von Rice, den wir hier ohne Beweis zitieren:

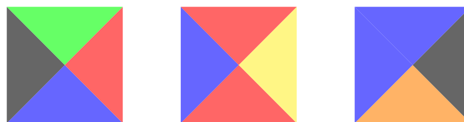
Theorem 8 (Satz von Rice)**.** *Sei* $\mathcal{P} \subsetneq F \subsetneq \mathcal{F}_\mu^{par}$*, dann ist*

$$\{\langle M\rangle\ ;\ M \text{ ist Turingmaschine, die eine Funktion aus } F \text{ berechnet }\}$$

unentscheidbar.

2.4.2 Weitere Beispiele unentscheidbarer Probleme

Im Folgenden gehen wir noch kurz auf weitere unentscheidbare Probleme ein. Conways Spiel des Lebens haben wir bereits in Abschnitt 2.1 erwähnt. Auch haben wir bereits in der Vorlesung das Problem der *Wang-Parkettierung* kennengelernt. Gegeben sind Gruppen von Quadraten einheitlicher Größe, deren Kanten mit bestimmten Farben markiert sind.



Das Problem besteht darin, zu entscheiden, ob die Ebene mit den gegebenen Kachel-Typen lückenlos parkettiert werden kann. Dabei dürfen die Kacheln nicht rotiert werden und nur gleichfarbige Kanten aneinander gelegt werden. Ein anderes unentscheidbares Problem ist *Hilberts zehntes Problem*. Gegeben ist ein Polynom $p(x_1, \dots, x_n)$ mit ganzzahligen Koeffizienten, für welches entschieden werden soll, ob es $a_1, \dots a_n \in \mathbb{Z}$ gibt mit $p(a_1, \dots a_n) = 0$. Die

Unentscheidbarkeit hängt hier von der Ganzzahligkeit der Lösungen ab. Ob es eine reelle Lösung gibt, ist (leicht) entscheidbar.
In vorigen Semestern haben wir uns bereits mit Grammatiken beschäftigt. Eine (unbeschränkte) Grammatik G war dabei gegeben durch ein 4-Tupel (T, N, S, R), wobei

$$\begin{aligned} T &: \text{die Menge der Terminalsymbole } (a, b, c, \ldots), \\ N &: \text{die Menge der Nichtterminalsymbole } (A, B, C, \ldots \notin T), \\ S &: \text{das ausgezeichnete Startsymbol aus } N \text{ und} \\ R &: \text{die Produktions- bzw. Ersetzungsregeln } \alpha \to \beta \text{ mit } \alpha, \beta \in \{N \cup T\}^* \end{aligned}$$

bezeichnen. Dabei sind T, N und R stets endliche Mengen. Die von G erzeugte Sprache $L(G)$ ist definiert als der transitive Abschluss der Relation $\to$, d.h. $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. Für die Grammatik G besteht das allgemeine *Wortproblem* darin, zu entscheiden, ob ein beliebiges Wort $w \in T^*$ in der Sprache $L(G)$ liegt. Dieses Problem ist ebenfalls unentscheidbar, da man das Halteproblem darauf reduzieren kann (Übungsaufgabe).
Schließlich betrachten wir noch das *Post'sche Korrespondenzproblem.* Für ein Alphabet Σ sind zwei Listen $v_1, \ldots, v_k$ und $w_1, \ldots, w_k$ von Wörtern in Σ^* gegeben. Die Frage ist, ob es eine nicht-leere Indexfolge $i_1, i_2, \ldots, i_m$ gibt, sodass $v_{i_1} v_{i_2} \cdots v_{i_m} = w_{i_1} w_{i_2} \cdots w_{i_m}$ gilt. Auch hier kann mittels Reduktion gezeigt werden, dass dieses Problem unentscheidbar ist.
Es sei noch einmal erwähnt, dass Unentscheidbarkeit bedeutet, dass es keinen allgemeinen Algorithmus für *alle* Fälle gibt. Das Lösen von Spezialfällen kann dagegen durchaus möglich sein, erfordert aber im Allgemeinen einige Anstrengung.

2.4.3 Rekursive und rekursiv aufzählbare Prädikate

Analog zur Entscheidbarkeit (die wir über Turingberechenbarkeit definiert haben), können wir Eigenschaften von Prädikaten $P \subseteq \mathbb{N}_0$ untersuchen.

Definition 9. Sei P ein Prädikat, d.h. $P \subseteq \mathbb{N}_0$. P heißt genau dann *rekursiv*, wenn die charakteristische Funktion von P μ-rekursiv und total ist, d.h. $\chi_P \in \mathcal{F}_\mu^{tot}$. P heißt genau dann *rekursiv aufzählbar*, wenn gilt

- $P = \emptyset$ oder
- $P = \{f(x) \mid x \in \mathbb{N}_0\}$ für eine Funktion $f \in \mathcal{F}_\mu^{tot}$.

Mit dieser Definition sind die Begriffe *rekursiv* und *entscheidbar* gleichbedeutend. Wie das folgende Lemma zeigt, sind dann auch die Begriffe semientscheidbar und rekursiv aufzählbar äquivalent.

Lemma 12. *Ein Prädikat $P \subseteq \mathbb{N}_0$ ist genau dann rekursiv aufzählbar, wenn die Funktion c_P mit*

$$c_P(y) = \begin{cases} 1, & y \in P \\ \text{undefiniert}, & y \notin P \end{cases}$$

in $\mathcal{F}_\mu^{par}$ liegt.

Beweis. Sei zunächst P rekursiv aufzählbar. Für $P = \emptyset$ ist c_P nirgends definiert und damit in $\mathcal{F}_\mu^{par}$. Sei also $P = \{f(x) \mid x \in \mathbb{N}_0\}$ für eine μ-rekursive Funktion f. Dann ist

$$c_P(y) = 1 \dot{-} |y - f(\underbrace{\mu x (f(x) = y)}_{\text{definiert} \Leftrightarrow y \in P})|$$

in $\mathcal{F}_\mu^{par}$, da die verwendeten Funktionen (modifizierte Differenz, Betragsdifferenz) primitiv rekursiv sind.
Ist nun umgekehrt $c_P \in \mathcal{F}_\mu^{par}$, so können wir c_P nach Korollar 1 durch zwei primitiv rekursive Funktionen und einmalige Anwendung des μ-Operators darstellen, also $c_P(y) \; \hat{=} \; q(\mu s(y), y)$ mit $q, s \in \mathcal{P}$. Es gilt also

$$y \in P \Leftrightarrow c_P(y) \text{ ist definiert } \Leftrightarrow \exists z : s(y, z) = 0. \tag{2.4}$$

Wir wollen zeigen, dass P rekursiv aufzählbar ist. Dazu verwenden wir den Trick der Paarkodierung, beispielsweise mit der Cantorschen Paarungsfunktion oder mit der Funktion $k : \mathbb{N}_0^2 \to \mathbb{N}_0$ mit $k(v, w) = 2^v 3^w$. So ist k primitiv rekursiv und es gibt zwei Umkehrfunktionen $d_1, d_2 \in \mathcal{P}$, sodass $\forall v, w$

$$d_1(k(v, w)) = v$$
$$d_2(k(v, w)) = w$$

gilt. Ist $P = \emptyset$, so ist P nach Definition rekursiv aufzählbar. Sei also $P \neq \emptyset$, dann definieren wir für ein festes $p \in P$

$$f(x) := \begin{cases} d_1(x), & \text{falls } \exists v, w \in \mathbb{N}_0 : x = 2^v 3^w \text{ und } s(d_1(x), d_2(x)) = 0 \\ p, & \text{sonst.} \end{cases}$$

Die Funktion f ist selbst primitiv rekursiv und es gilt $P = \{f(x) \mid x \in \mathbb{N}_0\}$. Dabei gilt die Beziehung

- $\supseteq$: nach Definition von f und Gleichung 2.4;
- $\subseteq$: da für $y \in P$ nach Gleichung 2.4 gilt, dass $\exists z : s(y,z) = 0$; setze $x := k(y, z)$. Dann ist $s(d_1(x), d_2(x)) = s(y, z) = 0$, also $f(x) = d_1(x) = y$.

□

Wie der Beweis zeigt, lassen sich (nicht-leere) rekursiv aufzählbare Mengen sogar von einer primitiv rekursiven Funktion aufzählen. Wir halten noch folgende wichtige Beziehung fest.

Theorem 9. *Die Menge der rekursiven Prädikate ist echt enthalten in der Menge der rekursiv aufzählbaren Prädikate, d.h. es gilt*

$$\{P \subseteq \mathbb{N}_0 \mid P \textit{ rekursiv }\} \subsetneq \{P \subseteq \mathbb{N}_0 \mid P \textit{ rekursiv aufzählbar }\}.$$

Beweis. Um die echte Inklusion zu zeigen, betrachten wir erneut das Halteproblem

$$H = \{y \in \mathbb{N}_0 \mid y = \Psi(\langle M \rangle \langle w \rangle) \text{ und } M \text{ hält bei Eingabe } w\}$$

Nach Korollar 2 ist H nicht entscheidbar, also nicht rekursiv. Das Halteproblem ist jedoch rekursiv aufzählbar, denn

$$c_H(y) := 1 \dot{-} (1 \dot{-} (\#\text{Rechenschritte, nach denen } M \text{ bei Eingabe } w \text{ hält}))$$

ist in $\mathcal{F}_\mu^{par}$. (Das zeigt man wie im Beweis von Theorem 5.) □

Interessanterweise gilt für Sprachen und Grammatiken folgende Eigenschaft.

Theorem 10. *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann rekursiv aufzählbar, wenn es eine Grammatik G, die L erzeugt, d.h. es gilt $L = L(G)$.*

Allerdings gibt es auch Mengen, die nicht rekursiv aufzählbar sind. Zum Schluss zeigen wir noch einen nützlichen Zusammenhang zwischen rekursiven und rekursiv aufzählbaren Prädikaten.

Lemma 13. *Sei $P \subseteq \mathbb{N}_0$ ein Prädikat und $\mathbb{N}_0 \setminus P$ das Komplement. Sind P und $\mathbb{N}_0 \setminus P$ rekursiv aufzählbar, so ist P (bzw. das Komplement) rekursiv.*

Beweis. Da P und das Komplement rekursiv aufzählbar sind, gibt es $f, g \in \mathcal{F}_\mu^{tot}$, sodass

$$\begin{aligned} P &= \{f(x) \mid x \in \mathbb{N}_0\} \\ \mathbb{N}_0 \setminus P &= \{g(x) \mid x \in \mathbb{N}_0\}. \end{aligned}$$

Damit definieren wir die Funktion $h(y) := \mu x (f(x) = y \ \vee \ g(x) = y)$. Da $h \in \mathcal{F}_\mu^{tot}$ ist auch die charakteristische Funktion χ_P von P

$$\chi_P(y) = \begin{cases} 1, & \text{falls } f(h(y)) = y \\ 0, & \text{falls } g(h(y)) = y. \end{cases}$$

μ-rekursiv und total und P folglich rekursiv. □

Kapitel 3

Paktische Berechenbarkeit

3.1 Die Random Access Machine

Im letzten Semester haben wir die sogenannte Random Access Machine (RAM) kennen gelernt und diese als theoretische Grundlage für Laufzeitanalysen verwendet. Wir möchten uns in diesem Abschnitt damit beschäftigen, in welchem Verhältnis die Laufzeiten der Turingmaschinen zu den RAM-Laufzeiten stehen.

Zur Erinnerung Die RAM besteht aus folgenden Elementen:

- Abzählbar unendlich viele Speicherzellen, die mit den natürlichen Zahlen adressiert werden.
- Jede Speicherzelle kann eine beliebige reelle (approximiert, z.B. gemäß IEEE Standard) oder natürliche Zahl enthalten. Es ist sowohl eine direkter, als auch indirekter Speicherzugriff möglich
- Elemente Rechenoperationen: Addition, Subtraktion, Multiplikation, Division, Restbildung, Abrunden, Aufrunden
- Elementare Relationen: $\leq$, $\geq$, $=$, $\vee$, $\wedge$
- Kontrollierende Befehle: Verzweigung, Aufruf von Subroutinen, Rückgabe
- Datenbewegende Befehle: Laden, Speichern, Kopieren

Im logarithmischen Kostenmaß gehen wir davon aus, dass alle Zahlen binär codiert sind und der Zugriff auf ein bit in $\Theta(1)$ durchgeführt werden kann. Entsprechen benötigt z.B. der Zugriff auf eine natürliche Zahl n $O(\log_2 n)$ Zeit.
Programme, die von der RAM ausgeführt werden sollen, befinden sich in einem separaten Speicher, welcher nicht zur Laufzeit manipuliert werden

kann. Ein Programm darf nur aus endlich vielen Codezeilen bestehen und zu beginn dürfen nur endlich viele Speicherzellen mit der Eingabe belegt sein. Eine Programmzeile besteht stets aus einer Operation op_j, einer Menge von Speicherzellen $i_1, i_2, \ldots, i_k$, auf welche diese Operation zugreifen muss und einer Speicherzelle i_p in welche das Ergebnis geschrieben wird.

Ausgehend von einer RAM R mit Laufzeit $t(n)$ für eine Eingabe aus insgesamt n bits, möchten wir das Verhalten von R durch eine 2-Band Turingmaschine M simulieren. Dazu verwenden wir das erste Band als Speicher für die Belegten Speicherzellen der RAM.

$$\text{Band 1:} \#\#\# \operatorname{bin}(i_1) \# \operatorname{bin}(c(i_1)) \#\# \operatorname{bin}(i_2) \# \operatorname{bin}(c(i_2)) \#\# \cdots \\ \#\# \operatorname{bin}(i_m) \# \operatorname{bin}(c(i_m)) \#\#\#$$

wobei $i_1, i_2, \ldots, i_m$ die aktuell belegten Speicherzellen sind und $c(i_1)$, $c(i_2)$, ..., $c(i_m)$ deren Inhalt. Anfangs enthält Band 1 genau die Eingabe $\Rightarrow O(n)$ bits. Später enthält Band 1 höchstens $O(n + t(n))$ bits, da jedes erzeugte bit kostet auch einen der maximal $t(n)$ vielen Rechenschritte.
Die Turingmaschine M „merkt" sich in ihrem Zustand, welche Programmzeile j von R als nächstes auszuführen ist. Sie schreibt die zugehörigen Registerinhalte $c(i_1)$, $c(i_2)$, ..., $c(i_k)$ von Band 1 auf Band 2, führt darauf die Operation op_j aus und schreibt schließlich das Ergebnis in Zelle i_b auf Band 2 zurück. Existiert für i_b noch kein Eintrag auf Band 1, muss dieser erst erschaffen werden.

Betragen für eine Polynom g_1 die Kosten für das Ausführen einer Programmzeile $O(g_1(n + t(n)))$, so lassen sich die Gesamtkosten (bei $\leq t(n)$ auszuführenden Programmzeilen) polynomiell durch $g_2(n + t(n)) := t(n) \cdot g_1(n + t(n))$ abschätzen. Und da wir eine 2-Band TM durch eine 1-Band TM nur durch quadratischen Mehraufwand simulieren können, erhalten wir insgesamt folgendes Ergebnis:

Theorem 11. *Eine RAM mit laufzeit $t(n)$ im logarithmischen Kostenmaß lässt sich durch eine 1-Band-TM mit Laufzeit $O(g(n + t(n)))$ simulieren. Dabei hängt das Polynom g von der RAM ab.*

Umgekehrt lässt sich aber auch eine TM durch eine RAM simulieren.

Theorem 12. *Eine 1-Band-TM mit Laufzeit $t(n)$ lässt sich auf einer RAM im logarithmischen Kostenmaß in Zeit $O((n + t(n)) \log(n + t(n)))$ simulieren.*

Beweis. (Skizze)

- Speichere Zustand, Kopfstellung und Inhalte aller jemals besuchten Felder der TM in je einer Speicherzelle der RAM.

- Finde mit IF-Tests auf Zustand und Feldinhalt unterm Kopf heraus und welche Instruktion die TM ausführen würde.
- Aktualisiere entsprechend die Inhalte der Zellen, Zustand, Kopfstellung und Bandinhalt.

Woher kommt der Faktor $\log(n + t(n))$?
Die TM kann $n + t(n)$ Felder auf ihrem Band besuchen. Die Indizes der entsprechenden Zellen der RAM haben also die Länge $\log(n + t(n))$. □

Literaturverzeichnis

[1] Norbert Blum. Theoretische Informatik: Eine anwendungsorientierte Einführung. Oldenbourg Verlag, 2001.

[2] Uwe Schöning. Theoretische Informatik – kurzgefasst Spektrum, Akad. Verl., 2009.

[3] C. H. Papadimitriou. Computational Complexity Addison-Wesley 1994.

[4] Katrin Erk, Lutz Priese. Theoretische Informatik Springer, 2000.

[5] Ingo Wegener. Theoretische Informatik Teubner, 2005.