Berechenbarkeitstheorie

Hans Jürgen Ohlbach

Keywords: Was kann man berechnen, Turing-Berechenbarkeit, Church-Turing These, Loop-Berechenbarkeit, Ackermann-Funktion, Unentscheidbarkeit des Halteproblems, Unmöglichkeitsbeweise, Beweis durch Diagonalisierung

Empfohlene Vorkenntnisse: Turingmaschinen

Inhaltsverzeichnis

1 Einführung

Es geht um die Frage, was können Computer berechnen, und was nicht? Gibt es vielleicht lösbare Problem, die heutige Computer nicht lösen können, und mit welchen Techniken könnte man sie in Zukunft lösen? Wo sind die Grenzen dessen, was man automatisieren kann?

Wir Menschen haben eine intuitive Vorstellung davon, welche Art von Problemen man automatisch lösen kann. Diese Vorstellung gerät aber sehr schnell an ihre Grenzen. Den größten gemeinsamen Teil zweier Zahlen kann man definitiv automatisch lösen. Auch den kürzesten Weg zwischen zwei Städten kann man automatisch finden. Das macht jedes Navi.

Aber wie steht es z.B. darum, automatisch festzustellen, das ein Programm keine Endlosschleifen hat, und für jede Eingabe terminiert? Hier versagt die Intuition. Es lässt sich aber beweisen, dass das nicht möglich ist.

In der Berechenbarkeitstheorie untersucht man die Fragen, was „berechenbar“ überhaupt bedeutet, und dann, was berechenbar ist, und was nicht. Insbesondere lernt man an diesen Problemen Beweistechniken für Unmöglichkeitsbeweise kennen.

2 Gegenstände der Berechnung

Grundsätzlich kann man unterscheiden zwischen Digitalrechnern und Analogrechner. Analogrechner führen ihre Berechnungen mit Hilfe von *kontinuierlich* mechanischen oder elektronischen Vorgängen durch. Die Betonung liegt dabei auf *kontinuierlichen*, so kontinuierlich wie die reellen Zahlen. Ein Beispiel für einen mechanischen Analogrechner ist der nur noch den Älteren bekannte Rechenschieber, dessen Läufer kontinuierlich verschoben werden kann.

In den Anfangszeiten der Computerentwicklung gab es auch elektronische Analogrechner, die mit kontinuierlichen Spannungsverläufen rechnen konnten. Aus vielerlei Gründen sind die aber inzwischen alle durch die viel genaueren, vielseitigeren und verlässlicheren Digitalrechner ersetzt worden. Im Gegensatz zu Analogrechnern rechnen Digitalrechner mit *diskreten* Werten, heute fast nur noch mit Bits. Es gab auch welche, die mit Zahlen $0 \ldots 9$ rechneten. Für die Theorie spielt das keine Rolle (wohl aber für die Technik dahinter). Auch die Zahlen $0 \ldots 9$ kann man wiederum mit Bits kodieren.

Man kann also feststellen, was Digitalrechner tun wenn sie rechnen, ist Bitfolgen zu manipulieren. Bitfolgen kann man wiederum als Repräsentation von ganzen Zahlen auffassen, und zwar reichen sogar die nicht-negativen natürlichen Zahlen $\mathbb{N}_0$ als Interpretation der Bitfolgen. Aus mathematischer Sicht berechnen Digitalrechner nichts anders als Funktionen

$$f : \mathbb{N}_0^n \mapsto \mathbb{N}_0.$$

Für uns heute ist das nichts überraschendes, denn es ist jedem klar, dass alles, was Computer verarbeiten sollen, Daten, Töne, Bilder, Videos, usw. erst mal in Bitfolgen umgewandelt werden muss.

Für Texte bietet sich z.B. die ASCII-Kodierung an.

Beispiel: ASCII-Codierung

Text:	M	a	i
	01001101	01100001	01101001
	$2^{22} + 2^{19} + 2^{18} + 2^{16}$	$+ 2^{14} + 2^{13} + 2^{8}$	$+2^{6} + 2^{5} + 2^{3} + 2^{0}$
Zahl:	5 071 209		

Die Zahl kann wieder eindeutig in den ASCII Text zurückübersetzt werden!

Gödel-Kodierung: eine schon viel früher von Kurt Gödel entwickelte Abbildung von Texten auf Zahlen ist die nach ihm benannte *Gödel-Kodierung*:

Zunächst werden die Buchstaben durchnummeriert:

a	b	...	i	...	z	A	B	...	M	...	Z
1	2	...	9	...	26	27	28	...	39	...	52

Die Nummern werden als Exponenten für eine aufsteigende Folge von Primzahlen 2,3,5,7,11,13,... benutzt. Ein Text wird mit diesen exponierten Primzahlen kodiert:

Text:	M	a	i	
	2^{39}	$\cdot 3^{1}$	$\cdot 5^{9}$	(2, 3, 5 sind die ersten Primzahlen)
Zahl:	3 221 225 472 000 000 000			

Wegen der Eindeutigkeit der Primfaktorzerlegung kann die Zahl wieder eindeutig in den Text zurück übersetzt werden.

Wie man Töne, Bilder, Videos usw. in Bitfolgen übersetzt, lernen die Medieninformatiker.

Wir halten also fest:

Aus Sicht der Berechenbarkeitstheorie ist das, was Digitalrechner tun:
Sie berechnen Funktionen auf natürlichen Zahlen.

3 Funktionen auf natürlichen Zahlen

Nicht alle alle Funktionen auf natürlichen Zahlen sind überall definiert. Z.B. ist die Divisionsfunktion nicht für den Divisor 0 definiert. Funktionen, die nicht überall definiert sind, nennt man *partielle Funktionen*.

Für Algorithmen, die diese Funktionen berechnen sollen, nehmen wir zur Vereinfachung und, dass die Algorithmen für die Werte, wo die Funktion nicht definiert ist, *nicht terminiert*, d.h. in eine Endlosschleife geraten. (Dann braucht man keine Theorie mit Exception Handling).

Für viele Funktionen auf natürlichen Zahlen ist intuitiv einleuchtend, dass es Algorithmen gibt, die diese berechnen, auch wenn die Algorithmen vielleicht auch kompliziert sind, wie in folgendem Beispiel:

$$f_\pi(n) = \begin{cases} 1 & \text{falls } n \text{ Anfangsabschnitt der Dezimaldastellung von } \pi \text{ ist} \\ 0 & \text{falls nicht} \end{cases}$$

Dafür muss man die Dezimalbruchentwicklung von π berechnen, was aber möglich ist.

Z.B. ist $f_\pi(314) = 1$ weil $\pi = 3.1415\ldots$, und $f_\pi(315) = 0$

Schon schwieriger wird es bei folgender Funktion

$$g_\pi(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der Dezimaldastellung von } \pi \text{ vorkommt} \\ 0 & \text{falls nicht} \end{cases}$$

Zur Berechnung von $g_n(1234567)$ müsste man die Dezimalbruchentwicklung von π soweit fortführen, bis die Folge 1234567 auftaucht. Wenn sie aber nirgends auftaucht?? Es könnte auch sein, dass ein zukünftiger Mathematiker beweisen kann, dass jede Zahl irgendwo in der Dezimalbruchentwicklung von π vorkommt. Dann wäre $g_\pi(n) = 1$ für alle n. Und das ist ganz leicht zu implementieren.

Eine andere Funktion, die ganz leicht zu implementieren wäre, wenn man nur wüsste wie, ist die folgende:

$$h_\pi(n) = \begin{cases} 1 & \text{falls in der Dezimaldarstellung von } \pi \text{ die Zahl 7 mindestens } n \text{ mal hintereinander vorkommt} \\ 0 & \text{falls nicht} \end{cases}$$

Es wäre $h_\pi(5) = 1$ wenn irgendwo in der Dezimaldarstellung von π 77777 vorkommt.

Zwei Fälle sind denkbar:

1. Jede 7er-Folge kommt irgendwo in der Dezimaldarstellung von π vor. Dann ist $h_\pi(n) = 1$ für alle n.
2. 7er-Folgen mit maximal k 7er kommen in π vor. Dann ist
$$h_\pi(n) = \begin{cases} 1 & \text{falls } n \leq k \\ 0 & \text{falls nicht} \end{cases}$$

In beiden Fällen ist h_π extrem leicht zu implementieren, aber niemand weiß, welche Version die richtige ist. Vielleicht wird es auch nie jemand wissen.

3.1 Wieviele berechenbare Funktionen auf $\mathbb{N}_0$ gibt es?

Jedes Programmiersystem benutzt eine endliche Menge Σ von Buchstaben für ihre Programme. Damit kann man abzählbar viele Texte Σ^* bilden, die man auch lexikographisch ordnen kann. Lässt man die syntaktisch falschen Texte weg, bleiben immer noch *abzählbar* viele Programme übrig.

Damit kann man also abzählbar viele Funktionen implementieren. Eine interessante Konsequenz daraus ist:

Es gibt offensichtlich reelle Zahlen, deren Dezimalbruchentwicklung nicht berechenbar ist.

Weil es überabzählbare viele reelle Zahlen gibt, es aber nur abzählbar viele Programme gibt, kann es nicht für jede reelle Zahl ein Programm für dessen Dezimalbruchentwicklung geben.

4 Turing-Berechenbarkeit

Es gibt inzwischen ganz viele unterschiedliche Prozessoren, und noch viel mehr unterschiedliche Programmiersprachen. Da stellt sich dann die Frage, ob die alle gleichmächtig sind, d.h. die gleiche Menge von berechenbaren Funktionen berechnen können, oder ob es da Unterschiede in der Berechnungskraft gibt.

Um diese Frage zu beantworten, geht man auf die von Alan Turing 1936 vorgeschlagene Maschine, eben die Turingmaschine, zurück und definiert zunächst den Begriff der Turing-Berechenbarkeit.

Definition 1 (Turing-Berechenbarkeit) *Eine Funktion $f : \mathbb{N}^k \mapsto Nat$ ist Turing-berechenbar falls es eine deterministische Turingmaschine gibt, die folgendes tut:*

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

Anstelle von Bitfolgen über $\{0,1\}$ kann man auch beliebige Wörter über beliebige Zeichen Σ erlauben.

Die Turingmaschine ist recht primitiv. Daher wäre es nicht verwunderlich, wenn unsere modernen Rechner stärker wären, d.h. Funktionen berechnen könnten, die die Turingmaschine prinzipiell nicht berechnen kann. Dem ist aber nicht so. Nach jahrzehntelanger intensiven Suche und Entwicklung ist es bisher noch nicht gelungen, ein stärkeres Berechnungsmodell als die Turingmaschine zu finden.

Das haben Alan Turing (1912 - 1954) selbst, sowie Alonzo Church (1903-1995) schon vermutet und die berühmte **Church-Turing-These** aufgestellt:

Church-Turing-These:
Jede intuitiv berechenbare Funktion auf $\mathbb{N}_0$ kann auch mit einer Turingmaschine berechnet werden.

Es ist eine These, kein Theorem. Um sie als Theorem zu beweisen, müsste man den Begriff *intuitiv berechenbar* mathematisch definieren, was wohl sehr schwierig wird. Man könnte sie aber widerlegen,

indem man ein Berechnungsmodell findet, und eine Funktion, die mit diesem Berechnungsmodell berechnet werden kann, und dann nachweist, dass die Turingmaschine diese Funktion nicht berechnen kann. Das ist bisher noch niemand gelungen.

Indem man zeigt, wie man höhere Konstrukte wie Parallelität, if-then-else, while usw. in einer Turingmaschine implementieren kann (siehe Miniskript über Turingmaschinen) wird es plausibler, dass auch höhere Programmiersprachen nicht berechnungsstärker sind als Turingmaschinen.

Um das etwas formaler zu machen, hat man andere Programmiersysteme mit solchen höheren Konstrukten definiert, die aber immer noch einfach genug sind, um sie mathematisch zu erfassen, und dann gezeigt, dass jedes Programm in einem dieser Programmiersysteme in ein gleichwertiges Programm in den anderen Programmiersystemen transformiert werden kann. Also gibt es bisher nichts berechnungsstärkeres als Turingmaschinen.

Man sagt daher auch, das Programmiersystem X ist *Turing-äquivalent* wenn es die gleichen Funktionen berechnen kann wie Turingmaschinen.

5 Loop-Berechenbarkeit

Bei diesen Untersuchungen hat sich ein System, die *Loop-Programmiersprache* als echt schwächer als die Turingmaschine herausgestellt, wobei sie trotzdem noch erstaunlich viele Berechnungsprobleme lösen kann. Daher soll sie hier kurz angesprochen werden.

Definition 2 (Loop-Programme) *Die Sprache hat eine endliche Menge von Variablen $x_0, \ldots, x_n$. Ein Loop-Programm ist damit folgendermaßen aufgebaut:*

- *es gibt Zusweisungen: $x_i = x_j + c$; und $x_i = x_j - c$;, wobei c eine Zahl ist,*
- *es gibt die Hintereinanderausfühung: Loop-Programm$_1$;Loop-Programm$_2$,*
- *und es gibt das Loop-Konstrukt: Loop x_i DO Loop-Programm END;*

Zuweisung und Hintereinanderausführung funktionieren wie erwartet. Wichtig ist die Ausführung des Loop-Konstrukts.
Loop x_i DO Loop-Programm END
führt das Loop-Programm genau so oft aus, wie der Wert von x_i beim Eintritt in die Schleife war. Wenn das x_i in der Schleife verändert wird, hat das keinen Einfluss auf die Anzahl der Schleifendurchläufe.

Wenn zu Beginn $x_1 = 5$ ist, dann wird die Schleife
Loop x_1 DO $x_1 = x_1 + 1$ END trotzdem nur 5 mal durchlaufen, auch wenn x_1 dann einen anderen Wert hat.

Die Ein-Ausgabe-Konventionen sind dabei: $x_1, \ldots, x_k$ enthalten zu Beginn die Eingabewerte, und x_0 enthält am Ende das Ergebnis.

Beobachtung: **Ein Loop-Programm terminiert immer.**

Das folgt unmittelbar aus der Tatsache, dass die Anzahl der Loop-Durchläufe beim Eintritt in die Schleife festliegt.

Das ist anders als bei einem while-Konstrukt *while Bedingung DO Programm END*, wo die Bedingung u.U. nie falsch wird und daher die Schleife nicht terminiert.

Loop-Programme können keine partiellen Funktionen implementieren.
Allein daher ist Loop-Berechenbarkeit schwächer als Turing-Berechenbarkeit.

6 Die Ackermann-Funktion

Loop-Programme können keine partiellen Funktionen implementieren. Aber wie steht es mit totalen Funktionen? Gibt es welche, die eine Turingmaschine berechnen kann, aber kein Loop-Programm? Die gibt es in der Tat, aber nur wenn die Funktionen ungeheuer komplexe Berechnungen notwendig machen.

Das „klassische“ Beispiel dazu ist die *Ackermann Funktion*. Wilhelm Ackermann (1896-1962) hatte folgende Idee:

Von der einfachsten arithmetische Operation kann man immer kompliziertere definieren:

1. $x + 1$ (einfach 1 dazu addieren).
2. $x + n$ (n dazu addieren) ist wiederholtes +1.
3. $x * n$ (n mal multiplizieren) ist wiederholtes addieren.
4. x^n (n facher Exponent) ist wiederholtes multiplizieren.
5. $x^{\cdot^{\cdot^{x}}}\Big\}\, n$ mal (Hyperexponieren) ist wiederholtes exponieren, usw.

Man könnte also eine Funktion $f(x, y, m)$ definieren, wo m angibt, welche Operation durchgeführt werden soll: m = 1: +1, m = 2: addieren, m = 3: multiplizieren usw. Es gelang ihm tatsächlich diese Funktion rekursiv zu definieren.

Die recht komplizierte Funktion wurde auf verschiedene Weise vereinfacht, z.B. indem man $y = 2$ festlegt. Eine Version davon ist die ***Hermes Funktion*** $a(n, m)$.

$$\begin{array}{lcll} a(0,m) & = & m+1 & \\ a(n,0) & = & a(n-1,1) & (n>0) \\ a(n,m) & = & a(n-1,a(n,m-1)) & (n,m>0) \end{array}$$

wobei der erste Parameter n die Art der Operation steuert.

Welches schreckliche Wachstumsverhalten diese Funktion hat, zeigt die Tabelle:

$n\backslash m$	0	1	2	3	4	m
0	1	2	3	4	5	$m+1$
1	2	3	4	5	6	$m+2$
2	3	5	7	9	11	$2m+3$
3	5	13	29	61	125	$8 \cdot 2^m - 3$
4	13	65533	$2^{65536} - 3 \sim 2 \cdot 10^{19728}$	$a(3, 2^{65536} - 3)$	$a(3, a(4, 3))$	$8 \cdot 2^{2^{\cdot^{\cdot^{2}}}} - 3$
5	65533	$a(4, 65533)$	$a(4, a(5, 1))$	$a(4, a(5, 2))$	$a(4, a(5, 3))$	

Die Funktion ist auf jeden Fall total, aber man kann zeigen, dass die Zahlen schneller wachsen, als jedes Loop-Programm produzieren kann.

Eine bessere Intuition, warum die Ackermann-Funktion nicht Loop-implementierbar ist, bekommt man, wenn man Ackermanns ursprünglichen Anzatz versucht, in ein Loop-Programm umzusetzen, wo man außer dem Loop-Konstrukt nur Addition mit Konstanten hat.
Für $x + 1$ braucht man gar keine Loop-Schleife,
Für $x + n$ braucht man eine Loop-Schleife,
Für $x * n$ braucht man zwei geschachtelte Loop-Schleifen.
Für x^n braucht man drei geschachtelte Loop-Schleifen.
Für $\left.x^{\cdot^{\cdot^{x}}}\right\} n$ mal braucht man vier geschachtelte Loop-Schleifen, usw.

Wenn man jetzt den Steuerparameter m einlesen will, müsste man damit entscheiden, wieviel geschachtelte Loop-Schleifen notwendig sind. Dafür bräuchte man einen Präprozessor, der ein Loop-Programm mit m geschachtelten Loop-Schleifen erzeugt, compiliert und ausführt. Das ist mit Loop-Programmierung aber nicht möglich. Nur die universelle Turingmaschine kann sowas.

Die Ackermann-Funktion ist ein Extrembeispiel, das, außer vielleicht für Belastungstests von Prozessoren, kaum konkrete Anwendungen hat. Bei konkreten Anwendungen, auch bei ziemlich komplexen, kann man i.A. die Anzahl der maximal notwendigen Schleifendurchgänge aus der Eingabe abschätzen, und dann reicht Loop-Programmierung aus.

Für die Programmierpraxis hat das aber keine Konsequenzen, denn alle unsere Programmiersprachen und Prozessoren sind Turing-äquivalent, und damit von vornherein stärker als Loop-Programmierung.

7 Das Halteproblem

Das *Halteproblem* ist das Problem: ist es möglich, ein Programm zu schreiben, welches beliebige andere Programme auf Endlosschleifen testet, und immer das richtige Resultat liefert?

Wer schon einmal ein Programm compiliert und getestet hat, und dann ewig lange auf ein Ergebnis gewartet hat, ist mit diesem Problem konfrontiert. Oft hat man schnell herausgefunden, warum sich das Programm „aufgehängt“ hat, d.h. in eine Endlosschleife geraten ist. Manchmal ist es aber auch ganz schön verzwickt. Es wäre schön, wenn jeder Compiler einen „Endlosschleifentester“ hätte, der den Programmierer auf diese Fallen hinweist.

Leider kann man zeigen, dass so ein „Endlosschleifentester“ nicht existieren kann (sehr schade). Dieses Problem ist nur semientscheidbar. Wenn das Programm terminiert, wird man das irgendwann erfahren, aber wenn nicht, kann man zu keinem Zeitpunkt wissen, ob es vielleicht später doch terminiert, oder gar nie. Dazu braucht man gar keinen speziellen Algorithmus, man lässt das Programm einfach laufen.

Aber wie beweist man die Nichtexistenz eines Algorithmus?

Da die Beweistechnik dafür, *Beweis durch Diagonalisierung*, eine der grundlegenden Techniken für Unmöglichkeitsbeweise ist, wollen wir sie hier etwas ausführlicher darlegen. Wir starten mit einer Analogie aus dem echten Leben.

7.1 Das Unfehlbarer-Personalmanager-Problem

Dieses Beispiel soll die Beweistechnik mittels Diagonalisierung, wie sie zum Beweis der Unentscheidbarkeit des Halteproblems verwendet wird, illustrieren.

Wir beweisen: **Es kann keine unfehlbaren Personalmanager geben.**

Definition 3 (Das UPM-Problem) *Kann man anhand der Personalakte PA_A eines Angestellten/Bewerbers A entscheiden, ob A immer rechtzeitig seine Arbeit beendet?*

Wir betrachten zunächst das *spezielle UPM-Problem*

Definition 4 (Das SUPM-Problem) *Kann man anhand der Personalakte PA_A eines Angestellten/-Bewerbers A entscheiden, ob A, wenn er einmal am Tag seine eigene Personalakte zu bearbeiten hat, diese rechtzeitig bearbeitet?*

Beweis 1 *Wir führen einen Beweis durch Widerspruch:*
Angenommen es gibt einen unfehlbaren Personalmanager UPM, der anhand der Personalakte PA_A eines Angestellten/Bewerbers A entscheiden kann, ob A, wenn er einmal am Tag seine eigene Personalakte zu bearbeiten hat, diese rechtzeitig bearbeitet?

D.h.
$UPM(PA_A)$ = *ja* *falls A seine Personalakte PA_A rechtzeitig bearbeitet,*
$UPM(PA_A)$ = *nein* *falls A seine Personalakte PA_A nicht rechtzeitig bearbeitet.*

So ein unfehlbarer Personalmanager hat natürlich eine Sekretärin $UPMS$, die auch Personalakten bearbeiten kann. Sie macht das so, dass sie immer ihren Chef fragt, und dessen Entscheidung weitergibt. Nur einmal im Jahr, am 1. April macht sie einen Aprilscherz, und zwar jedes Jahr den gleichen, und das steht auch in ihrer Personalakte.

Falls an diesem Tag ihr Chef nein sagt, sagt sie ja, und falls ihr Chef ja sagt, versenkt sie die Personalakte gaaaanz tief in ihrer Schublade und geht erst einmal gaaaanz lange mit ihren Kolleginnen

Kaffee trinken. Ob sie die Personalakte jemals wiederfindet und fertig bearbeitet steht daher in den Sternen.

Am 1. April gilt also:
$UPMS(PA_A)$ = *ja* *falls* $UPM(PA_A)$ = *nein*
$UPMS(PA_A)$ = *Kaffeetrinken* *falls* $UPM(PA_A)$ = *ja.*

Die Fallunterscheidung ist vollständig, da der UPM ja unfehlbar ist, und auf jeden Fall für PA_A *terminiert.*

In diesem Jahr am 1. April bekommt sie ihre eigene Personalakte PA_{UPMS} *zu bearbeiten.*

Es gilt dann:

$UPMS(PA_{UPMS})$ *wird rechtzeitig fertig (und sagt ja)*
gdw. $UPM(PA_{UPMS})$ = *nein (sie hat ihren Chef gefragt)*
gdw. UPMS ihre eigene Personalakte nicht rechtzeitig bearbeitet.
(der UPM ist ja unfehlbar und hat immer recht).

Die Sekretärin wird also rechtzeitig fertig, genau dann wenn sie nicht rechtzeitig fertig wird. Das ist ein Widerspruch. Es kann daher keinen für dieses spezielle Problem unfehlbaren Personalmanager geben.

Der Beweis der Unentscheidbarkeit des UPM-Problems ist jetzt sehr einfach:

Beweis 2 *Falls es einen unfehlbaren Personalmanager gibt, der anhand der Personalakte* PA_A *eines Angestellten/Bewerbers* A *entscheiden kann, ob* A *immer rechtzeitig seine Arbeit beendet, dann könnte er auch entscheiden ob* A *insbesondere mit der Bearbeitung seiner eigenen Personalakte rechtzeitig fertig würde. Da das nicht geht, kann es auch keinen unfehlbaren Personalmanager für das UPM-Problem geben.*

Falls Sie auf eine Bewerbung eine Absage bekommen, können Sie jetzt dem Personalmanager z.B. sagen: „Es ist mathematisch bewiesen, dass Sie nicht unfehlbar sein können. Vielleicht haben Sie jetzt den größten Fehler Ihres Lebens gemacht, und den besten potentiellen Mitarbeiter Ihrer Firma abgewiesen. Vielleicht überlegen Sie sich Ihre Entscheidung doch noch mal“.

7.2 Unentscheidbarkeit des Halteproblems

Jetzt können wir die Beweisidee auf die Unentscheidbarkeit des Halteproblems übertragen. Der Originalbeweis wurde mit Turingmaschinen formuliert. Man kann ihn aber auch mit jeder anderen Programmiersprache, z.B. Java, formulieren.

Zunächst formuliert man das *spezielle Halteproblem*:

Definition 5 (Spezielles Halteproblem) *Kann man ein Programm SHT (Spezieller Haltetester) schreiben, welches für beliebige andere Programme P entscheiden kann, ob P angewendet auf seinen eigenen Code, d.h. $P(P)$, terminiert?*

Diese Programme P müssen also einen String als Eingabe erwarten. (Bei Turingmaschinen würde die andere Maschine P in eine Zahl n_P kodieren, und dann $P(n_P)$ aufrufen. Das ist aber für den eigentlichen Beweis unerheblich.)

Beweis 3 *Wir zeigen, dass so ein Spezieller Haltetester SHT nicht existieren kann.*

Angenommen SHT existiert. Er bekommt den Code eines Programms P als Eingabe und analysiert ihn. Falls er befindet, dass $P(P)$ terminiert, führt er `print(terminiert)` *aus, falls nicht, führt er* `print(terminiert nicht)` *aus.*

Jetzt modifizieren wir SHT etwas. SHT' ist fast wie SHT, nur aus der Stelle `print(terminiert nicht)` *machen wir* `print(terminiert)`*, und aus der Stelle* `print(terminiert)` *machen wir eine Endlosschleife* `while(true){}`.

Jetzt lassen wir SHT' sich selbst untersuchen (hier ist die Diagonalisierung*), d.h. wir rufen SHT'(SHT') auf.*

Angenommen, SHT'(SHT') hält an. Das passiert nur an der Stelle, wo SHT' `print(terminiert)` *ausführt. Das ist aber genau die Stelle, wo SHT* `print(terminiert nicht)` *ausgeführt hätte. SHT(SHT') hätte also für die Analyse, ob SHT'(SHT') terminiert,* `terminiert nicht` *ausgedruckt, obwohl SHT'(SHT') in Wirklichkeit terminiert. Also hätte sich SHT geirrt.*

Es kann also keinen korrekten Speziellen Haltetester geben.

Daraus folgt die Unentscheidbarkeit des allgemeinen Halteproblems: Angenommen, es gäbe einen Allgemeinen Haltetester AHT, der für beliebige Programme P, angewandt auf beliebige Eingabe E entscheiden kann, ob $P(E)$ terminiert oder nicht. In diesem Fall könnte AHT auch entscheiden, ob $P(P)$ terminiert. Da wir gezeigt haben, dass das nicht geht, kann es auch keinen allgemeinen Haltetester geben.

Dieser letzte Beweisschritt ist ebenfalls ein Beispiel für eine universelle Beweisstrategie:

Um ein Problem A als unlösbar zu beweisen, finde eine Spezialisierung A' von A und zeige dessen Unlösbarkeit. Dann kann auch A nicht lösbar sein.

Der Vorteil hier ist, dass man bei spezielleren Problemen mehr Informationen hat, die man für den Beweis nutzen kann.

In der Literatur findet man weitere Probleme, deren Unentscheidbarkeit bewiesen ist, z.B. das *Postsche Korrespondenzproblem* (PCP), was wir aber hier nicht weiter thematisieren wollen.

Interessant dabei ist jedoch eine weitere Beweistechnik:
***Beweis durch Reduktion*: Man reduziert ein Problem A auf ein Problem B, indem man zeigt: wenn man Problem B lösen könnte, könnte man auch A lösen. Wenn man vorher A als unlösbar bewiesen hat, folgt damit, dass auch B unlösbar ist.**

Im konkreten Fall hat man das Halteproblem auf das Postsche Korrespondenzproblem reduziert, und gezeigt, wenn man das PCP-Problem lösen könnte, dann könnte man auch das Halteproblem lösen, was nicht geht. Daher kann man auch das PCP-Problem nicht lösen.

Unlösbarkeitsbeweise per Diagonalisierung bilden oft die Wurzel für eine ganze Hierarchie von Unlösbarkeitsresultaten, die man dann per Reduktion von der Wurzel her beweist.

Ein guter theoretischer Informatiker hat eine ganze Bibliothek von Unlösbarkeitsresultaten. Wenn er dann ein neues Unlösbarkeitsresultat beweisen will, dann sucht er das passende alte unlösbare Problem, und versucht es auf das neue zu reduzieren. Wenn ihm das gelingt, dann hat er auch die Unlösbarkeit des neuen Problems bewiesen.

8 Konsequenz für Formale Sprachen vom Typ 0

Mit der Unentscheidbarkeit des Halteproblems kann man jetzt auch die Unentscheidbarkeit des Wortproblems für Typ 0-Sprachen per Reduktion beweisen.

Angenommen, das Wortproblem für Typ 0-Sprachen wäre entscheidbar mit einem Entscheidungsverfahren EV.

Man kann jede Turingmaschine T in eine Typ 0-Grammatik G_T umwandeln, die genau die Wörter E $\in L(G_T)$ produziert, bei denen die Turingmaschine T(E) hält. Wie das geht, steht im Miniskript Typ01.

Um für eine Eingabe E zu überprüfen, ob T(E) hält, untersucht man mit dem Entscheidungsverfahren EV, ob E ein Wort der Sprache $L(G_T)$ ist. Damit könnte man also entscheiden, ob die Maschine T für die Eingabe E hält, oder nicht. Da das nicht möglich ist, kann es auch kein Entscheidungsverfahren für das Wortproblem von Typ 0-Sprachen geben.

Ergo: **Das Wortproblem für Typ 0-Sprachen ist unentscheidbar.**