Berechenbarkeit und Komplexitätstheorie

WS17 — Müller — Universität Trier

Basis: i.W. Skripten zur Vorlesung Informatik-B2, Universität Duisburg-Essen
(W. Luther / P. Hertling, Überarbeitung durch N. Müller)

Stand: Oktober 2017

Inhaltsverzeichnis

Literatur

[1] Uwe Schöning: Theoretische Informatik - kurzgefasst, 4. Auflage. Spektrum, Heidelberg 2001

[2] John E. Hopcroft, Jeffrey D. Ullman: Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, Massachusetts, 1979 (Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, Oldenbourg-Verlag, München, 2000).

[3] C. Meinel, M. Mundhenk: Mathematische Grundlagen der Informatik. Mathematisches Denken und Beweisen, eine Einführung. Teubner, 2002.

[4] K. A. Ross, C. R. B. Wright: Discrete Mathematics. Prentice Hall, 1988.

[5] E. Kinber / C. Smith: Theory of Computing. A Gentle Introduction. Prentice Hall.

Es gibt zahlreiche gute Skripten im Internet, z.B.: Skripten zur Vorlesung Informatik-B2 an der Universität Duisburg-Essen (Prof. Luther, Prof. Hertling)

Es gibt ungezählte Einführungen in die Thematik; finden Sie am besten eine für Sie am besten geeignete heraus !

$\rightsquigarrow$ nicht nachprüfbare Hausaufgabe: gehen Sie in die Bibliothek oder in eine Buchhandlung und beginnen Sie, in verschiedenen Büchern zu lesen: wenigstens zwei davon sollten Sie über die Vorlesung begleiten...

1 Intuitiver Berechenbarkeitsbegriff und Church'sche These

Aus "Automaten und Formale Sprachen" ist die Chomsky-Hierarchie der formalen Sprachen bekannt. Dabei wurde angemerkt, dass es zu kontextfreien Grammatiken Fragestellungen gibt, die nicht entscheidbar sind. Um eine derartige negative Aussage zu begründen, muss man zuerst einmal einen präzisen Begriff der *Entscheidbarkeit* haben. Es ist eines der Ziele dieser Veranstaltung, nachzuweisen, dass es in der Tat einen präzise definierten Begriff der Entscheidbarkeit für Mengen von natürlichen Zahlen(vektoren) oder für Mengen von Wörtern gibt. Ebenso gibt es einen präzise definierten Begriff der Berechenbarkeit für Funktionen auf natürlichen Zahlen oder auf endlichen Wörtern. Wir werden zuerst auf verschiedene Weise Berechenbarkeit für Funktionen definieren und zeigen, dass ganz verschiedene Ansätze alle zu dem gleichen Begriff führen. Anschließend werden wir den Entscheidbarkeitsbegriff für Mengen auf den Berechenbarkeitsbegriff für Funktionen zurückführen.

Zuerst wollen wir diskutieren, welche Funktionen denn als berechenbar bezeichnet werden sollten. Wir verwenden dabei zunächst eine vereinfachte Sichtweise: der Zeitverbrauch und der Speicherplatzverbrauch interessieren uns jetzt (noch) nicht. Wir wollen nur sehen, ob eine Funktion im Prinzip berechnet werden kann, wenn man annimmt, dass für die Rechnung hinreichend viel Zeit und Speicherplatz zur Verfügung stehen.

Definition 1.1 (Intuitiver Berechenbarkeitsbegriff)**.** *Die von einem Algorithmus A (z.B. einem JAVA-Programm) berechnete Funktion $f_A : \mathbb{N}^k \dashrightarrow \mathbb{N}$ ist definiert wie folgt:*

$$f_A(n_1, ..., n_k) = \begin{cases} n, & \text{falls } A \text{ bei Eingabe von } (n_1, ..., n_k) \text{ das Resultat } n \\ & \text{berechnet und dann anhält,} \\ \text{undefiniert,} & \text{falls } A \text{ bei Eingabe von } (n_1, ..., n_k) \text{ endlos rechnet.} \end{cases}$$

Eine Funktion f heißt berechenbar, wenn es einen Algorithmus A gibt, der sie berechnet (also $f = f_A$). Insbesondere hält A damit genau dann bei der Eingabe $(n_1, ..., n_k)$ an, wenn $f(n_1, ..., n_k)$ definiert ist.

Beispiel 1.2. *Berechnung der Summe zweier natürlicher Zahlen m und n als Funktion $f : \mathbb{N}^2 \to \mathbb{N}$ mit $f(m, n) = n + m$:*

```
1  import java.util.Vector;
2  import java.math.BigInteger;
3
4  public class Addition {
5    public static void main( String args[] ){
6
7      BigInteger m = new BigInteger( args[0] );
8      BigInteger n = new BigInteger( args[1] );
9
10     while ( n.compareTo(BigInteger.ZERO) > 0 ) {
11       n = n.subtract( BigInteger.ONE );
12       m = m.add( BigInteger.ONE );
13     }
14     System.out.println( m.toString() );
15   }
16 }
```

Beispiel 1.3. *Berechnung einer partiellen Funktion $f : \mathbb{N} \dashrightarrow \mathbb{N}$ durch folgenden Algorithmus:*

```
1  import java.math.BigInteger;
2
3  public class Forever {
4    public static void main( String args[] ) {
5      BigInteger n = BigInteger.ONE;
6      while ( n.compareTo(BigInteger.ZERO) > 0 ) {
7        n = n.add(BigInteger.ONE);
8      }
9    }
10 }
```

Da die obige Schleife nicht abbricht, berechnet der Algorithmus die partielle Funktion f von $\mathbb{N}$ nach $\mathbb{N}$, die nirgends definiert ist.

Beispiel 1.4. *Berechnung einer totalen Funktion* $f : \mathbb{N} \to \mathbb{N}$ *mit*

- $f(n) = 1$, *falls die Dezimaldarstellung von* n *ein Anfangsabschnitt der Dezimalbruchentwicklung von* $\pi = 3,1415...$ *ist,*
- $f(n) = 0$ *sonst.*

Die Dezimalbruchentwicklung von π *kann man mit beliebiger Genauigkeit berechnen:*

$$\pi = 4 \cdot \lim_{t \to \infty} (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots \pm \frac{1}{2t-1})$$

Man braucht für größere Genauigkeit natürlich mehr Zeit und mehr Speicher, aber darum geht es uns im Moment nicht. Für jede Zahl n *kann man* $f(n)$ *in endlich vielen Schritten ausrechnen. Daher ist diese Funktion* f *berechenbar.*

Beispiel 1.5. *Berechnung einer totalen Funktion* $g : \mathbb{N} \to \mathbb{N}$ *mit*

- $g(n) = 1$, *falls die Dezimaldarstellung von* n *als Teilwort in der Dezimalbruchentwicklung von* $\pi = 3,1415...$ *vorkommt,*
- $g(n) = 0$, *sonst.*

Ob diese Funktion berechenbar ist oder nicht, ist nicht bekannt!

Dazu weiß man heute zu wenig über die Zahl π. *Tatsächlich gibt es Wissenschaftler, die vermuten, dass jedes aus den Dezimalziffern gebildete Wort in der Dezimalbruchentwicklung von* π *vorkommt. Wenn das stimmt, dann ist* g *die Funktion, die überall den Wert* 1 *annimmt.*

Beispiel 1.6. *Berechnung einer totalen Funktion* $h : \mathbb{N} \to \mathbb{N}$ *mit*

- $h(n) = 1$, *falls die* 7 *mindestens* n *mal hintereinander in der Dezimalbruchentwicklung von* π *vorkommt,*
- $h(n) = 0$, *sonst.*

Diese Funktion ist berechenbar: Entweder kommen beliebig lange Folgen von 7 *in der Dezimalentwicklung von* π *vor. Dann ist* h *die konstante Funktion, die überall den Wert* 1 *annimmt. Oder es gibt eine Zahl* n_0, *so dass die* 7 *zwar* n_0*-mal in der Dezimalbruchentwicklung von* π *vorkommt (dann natürlich auch* n*-mal für jedes* $n \leq 0$*), aber nicht* $n_0 + 1$*-mal (dann natürlich auch nicht* n*-mal für jedes* $n > n_0$*). In diesem Fall gilt* $h(n) = 1$ *für* $n \leq n_0$, $h(n) = 0$ *für* $n > n_0$.

In jedem der Fälle gibt es einen Algorithmus zur Berechnung von h. *Allerdings wissen wir heute nicht, welches der richtige Algorithmus ist...*

Anmerkung: Beispiel 1.4 darf nicht darüber hinweg täuschen, dass die analoge Funktion f_r für den Anfangsabschnitt der Länge n der Dezimalbruchentwicklung der reellen Zahl r nicht für jede reelle Zahl r berechenbar ist. Es gibt überabzählbar viele reelle Zahlen und auch überabzählbar viele Dezimalbruchentwicklungen , aber nur abzählbar viele zum Beispiel in JAVA formulierbare Algorithmen zur Berechnung von Dezimalbruchentwicklungen. Es gibt daher auch nur abzählbar viele reelle Zahlen, deren Dezimalbruchentwicklungen berechnet werden können. Die Dezimalbruchentwicklungen aller anderen reellen Zahlen (das sind immer noch überabzählbar viele) kann man nicht berechnen.

In den folgenden Abschnitten werden wir Berechenbarkeit für Funktionen auf natürlichen Zahlen(vektoren) bzw. auf endlichen Wörtern auf verschiedene Weise einführen, nämlich über Turingmaschinen, über WHILE-Programme . Wir werden sehen, dass all diese Berechenbarkeitsbegriffe äquivalent sind.

Es gibt einen schwächeren Berechenbarkeitsbegriff, nämlich die Klasse der über LOOP-Programme berechenbaren Funktionen. Einen dazu äquivalenten Begriff (primitiv-rekursive Funktionen) können wir hier nur erwähnen.

Andere Ansätze wie GOTO-Programme und die μ-rekursiven (partiell rekursiven) Funktionen führen zu einen Berechbarkeitsdefinition, die ebenfalls zur Turing-Berechenbarkeit äquivalent ist

Offen ist die Frage, ob es auch eine stärkere, praktisch verwendbare Definition berechenbarer Funktionen gibt. Es ist allerdings bis heute niemandem gelungen, überzeugend zu demonstrieren, dass es Funktionen auf den natürlichen Zahlen gibt, die in einem intuitiven, praktisch realisierbaren Sinne 'berechenbar' sind. die aber trotzdem nicht bereits durch Turingmaschinen berechenbar sind. Dies hat zu der von den meisten Wissenschaftlern akzeptierten These geführt, die Church bereits 1936 formuliert hat:

Church'sche These

Die durch Turingmaschinen berechenbaren Funktionen (d.h. genau die durch WHILE-Programme berechenbaren Funktionen) auf natürlichen Zahlen sind genau die im intuitiven Sinn berechenbaren Funktionen auf natürlichen Zahlen.

Das heißt, wenn von einer Funktion nachgewiesen ist, dass sie in einem der angegebenen Sinne nicht berechenbar ist, dann ist die (allerdings nicht streng bewiesene) Überzeugung, dass die Funktion gar nicht berechenbar ist.

Wir werden zeigen, dass es Funktionen gibt, die nicht berechenbar sind.

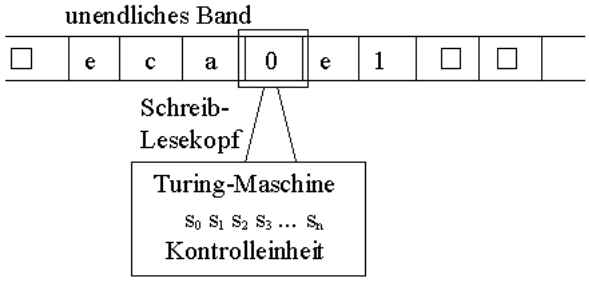
Da wir den Entscheidbarkeitsbegriff für Mengen auf diesen Berechenbarkeitsbegriff begründen werden, haben wir dann auch einen präzise definierten Entscheidbarkeitsbegriff für Mengen. Wir werden sehen, dass es Probleme gibt, die nicht entscheidbar sind; zum Beispiel ist das Wortproblem nicht für jede Typ-0-Sprache entscheidbar.

Wichtige verwendete Schreibweisen (i.W. aus "Automaten und Formale Sprachen"):

- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$: natürliche Zahlen (inkl. 0)
- totale Funktion $f\colon A \to B$
- partielle Funktion $f\colon A \dashrightarrow B$
- Potenzmenge $\mathcal{P}(X) := \{U \mid U \subseteq X\}$ einer Menge X
- Alphabet Σ: endliche Menge von Zeichen, z.B. $\Sigma = \{a, b\}$
- Worte/Zeichenketten über Σ: endliche Folgen von Zeichen aus Σ (vgl. Java-`String`)
- $|w|$ Länge des Wortes w (vgl. Java-`length()`)
- $\#_a w$ gibt an, wie oft das Zeichen a in w vorkommt
- Σ^k: Worte der Länge $k \in \mathbb{N}$,
- ε: leeres Wort, String mit Länge 0
- Σ^*: Worte beliebiger Länge, $\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k$

2 Berechenbarkeit mittels Turingmaschinen

Turingmaschinen sind evtl. schon bereits als Akzeptoren für die Sprachen der Chomsky Typ-0-Klasse bekannt.



Definition 2.1 (Turingmaschine)**.** *Eine (nichtdeterministische) Turingmaschine (TM) ist durch ein 7-Tupel beschrieben:*

$$\boldsymbol{TM = (S, E, A, \delta, s_0, \square, F)}$$

Dabei bedeuten

- $\boldsymbol{S = \{s_0, s_1, ..., s_n\}}$ *die Menge der Zustände,*
- $\boldsymbol{E = \{e_1, e_2, ..., e_r\}}$ *das endliche Eingabealphabet,*
- $\boldsymbol{A = \{a_0, a_1, ..., a_m\}}$ *das endliche Arbeitsalphabet (auch Bandalphabet genannt), es sei dabei* $\boldsymbol{E \subset A}$*,*
- $\boldsymbol{s_0}$ *der Startzustand,*
- $\boldsymbol{a_0 = \square}$ *das Blank, das zwar dem Arbeitsalphabet, aber nicht dem Eingabealphabet angehört,*
- $\boldsymbol{F \subseteq S}$ *die Menge der Endzustände*
- $\boldsymbol{\delta}$ *sei die (totale)* Überführungsfunktion *mit (im deterministischen Fall)*

  $$\boldsymbol{\delta : (S \setminus F) \times A \to S \times A \times \{L, R, N\}}$$

  bzw. (im nichtdeterministischen Fall)

  $$\boldsymbol{\delta : (S \setminus F) \times A \to \mathcal{P}(S \times A \times \{L, R, N\})}$$

  Hier bedeutet: $\boldsymbol{L}$*= links,* $\boldsymbol{R}$ *= rechts,* $\boldsymbol{N}$*= neutral (nicht bewegen).*

Die Überführungsfunktion wird folgendermaßen interpretiert:
Die Definition

$$\boldsymbol{\delta(s, a) = (s', b, x)}$$

bzw. im nichtdet. Fall

$$\boldsymbol{(s', b, x) \in \delta(s, a)}$$

mit $\boldsymbol{x \in \{L, R, N\}}$, beschreibt folgendes Verhalten(bzw. folgendes mögliche Verhalten im nichtdet. Fall):

Wenn sich der Automat im Zustand s befindet und unter dem Kopf das Zeichen a steht, so schreibt der Automat b und geht ein Feld nach rechts (R), bzw. links (L), bzw. bewegt sich nicht (N) und geht in den Zustand s' über.

Jede Turingmaschine kann auch als Tabelle geschrieben werden in Form einer Matrix mit den Zuständen als Zeilenmarkierung und den Zeichen aus dem Bandalphabet als Spaltenindizes. In jedem Kästchen steht dann ein Tripel bestehend aus neuem Zustand, geschriebenem Zeichen und Positionszeichen. Im Fall des nichtdeterministischen Automaten enthalten die Kästchen Mengeneinträge.

Definition 2.2 (Konfiguration)**.** *Eine Konfiguration einer Turingmaschine* $TM = (S, E, A, \delta, s_0, \square, F)$ *ist ein Tripel* (u, s, v) *aus* $A^* \times S \times A^+$*:*

- *Das Wort* uv *ist die aktuelle Bandinschrift (im schon beschriebenen oder besuchten Teil des Bandes)*
- s *ist der aktuelle Zustand,*
- *Der Schreib-Lesekopf steht auf dem ersten Zeichen von* $v\square$*, weshalb* $v \neq \varepsilon$ *gelten muss.*
- *Beim Start der Maschine enthält das Band ein Wort* $v = w\square$*, wobei* w *die Eingabe sei. Damit ist* v *aus* $E^* \circ \{\square\}$*, ferner sei* $s = s_0$ *und* u *sei leer. Bei Eingabe* w *startet die Maschine also mit der Konfiguration* $(\varepsilon, s_0, w\square)$.

O.B.d.A. gilt $S \cap A = \emptyset$*, daher schreiben wir eine Konfiguration* (u, s, v) *meist abgekürzt als* usv *auf.*

Definition 2.3 (Übergangsrelation $\vdash$)**.** *Zu einer (nicht)deterministischen Turingmaschine*

$$TM = (S, E, A, \delta, s_0, \square, F)$$

wird die Übergangsrelation $\vdash_{TM}$ *(oder kurz* $\vdash$*) aus* $(A^* \times S \times A^+)^2$ *folgendermaßen definiert:*

- $$a_1 \ldots a_m s b_1 \ldots b_n \vdash a_1 \ldots a_m s' c b_2 \ldots b_n$$
  falls: $(s', c, N) \in \delta(s, b_1), m \geq 0, n \geq 1$
- $$a_1 \ldots a_m s b_1 \ldots b_n \vdash a_1 \ldots a_m c s' b_2 \ldots b_n$$
  falls: $(s', c, R) \in \delta(s, b_1), m \geq 0, n \geq 2$
- $$a_1 \ldots a_m s b_1 \ldots b_n \vdash a_1 \ldots a_{m-1} s' a_m c b_2 \ldots b_n$$
  falls: $(s', c, L) \in \delta(s, b_1), m \geq 1, n \geq 1$

Wir schreiben noch zwei Sonderfälle auf:

- $$a_1 \ldots a_m s b_1 \vdash a_1 \ldots a_m c s' \square$$
  falls $(s', c, R) \in \delta(s, b_1)$ *,* $m \geq 0, n = 1$
- $$s b_1 \ldots b_n \vdash s' \square c b_2 \ldots b_n$$
  falls $(s', c, L) \in \delta(s, b_1)$ *,* $m = 0, n \geq 1$

Bei Überschreiten der Grenzen der Bandinschrift werden also Blanks an die Teilwörter v bzw. u angefügt.

Dabei bedeutet das Symbol $\vdash$, dass die Konfiguration links in die jeweilige Konfiguration rechts übergehen kann (bzw. muss, wenn es genau eine Möglichkeit gibt).

Weiterhin sei $\vdash^*$ wieder die reflexiv-transitive Hülle der Relation $\vdash$.

Beispiel 2.4 (Inkrementieren einer Binärzahl)**.** *Wir wollen eine Turingmaschine angeben, die die Bandinschrift als eine Binärzahl interpretiert und $\mathbf{1}$ hinzuaddiert. Als Beispiel wählen wir die Zahl $\mathbf{1010}$ und erhalten $\mathbf{1011}$. Interessant wird die Frage, wie die Maschine mit der Zahl $\mathbf{111}$ oder $\square$ umgeht, da hier die Stellenzahl wächst.*

$$TM = (\{s_0, s_1, s_2, s_f\}, \{0, 1\}, \{0, 1, \square\}, \delta, s_0, \square, \{s_f\})$$

mit

$\delta(s, a)$	0	1	$\square$
s_0	$(s_0, 0, R)$	$(s_0, 1, R)$	$(s_1, \square, L)$
s_1	$(s_2, 1, L)$	$(s_1, 0, L)$	$(s_f, 1, N)$
s_2	$(s_2, 0, L)$	$(s_2, 1, L)$	$(s_f, \square, R)$
s_f	$-$	$-$	$-$

Beispiel:

$$\begin{aligned} s_0111\square \;&\vdash 1s_011\square \vdash 11s_01\square \vdash 111s_0\square \\ &\vdash 11s_11\square \vdash 1s_110\square \vdash s_1100\square \vdash s_1\square000\square \\ &\vdash s_f1000\square \end{aligned}$$

Man überlegt sich, dass die Maschine zunächst mittels der Angaben der Zeile 1 solange nach rechts läuft, bis sie das Ende der Zahl (niederwertigste Ziffer) erreicht hat. Ein weiterer Schritt führt auf ein Blank (wenn zu Beginn nichts auf dem Band steht, steht der Lese-Schreibkopf schon auf einem Blank). Dann erfolgt wieder ein Schritt nach links und Übergang nach s_1.

Nunmehr ist die Maschine in Arbeitsposition. Trifft sie auf eine Null, so invertiert sie sie (Addition von $\mathbf{1}$) und geht nach s_2. Trifft sie vorher auf Einsen, müssen diese invertiert werden. Ein Spezialfall tritt auf, wenn es keine $\mathbf{0}$ in der Zahldarstellung gibt. Dann wächst die Stellenzahl um $\mathbf{1}$ und die Maschine schreibt eine $\mathbf{1}$ anstelle des ersten linken Blanks. Dies ist die führende Stelle, die Maschine geht in den Endzustand über und der Kopf bleibt hier stehen.

Die vorletzte Zeile der Tabelle betrifft den Fall, dass die Zahldarstellung eine Null enthält. Hier läuft die Maschine noch schrittweise im Zustand s_2 bis ans linke Ende der Zahl. Trifft sie auf ein Blank, kehrt sie um und geht mit dem Schreibkopf über der führenden Stelle in den Endzustand über.

Definition 2.5 (Initialkonfiguration, Finalkonfiguration, akzeptierte Sprache)**.**

- *Initialkonfiguration beim Start der Turingmaschine mit Eingabe $w \in E^*$ ist $s_0w\square$.*
- *Finalkonfigurationen sind alle Konfigurationen us_fv mit $s_f \in F$. Hier kann die Berechnung nicht mehr fortgesetzt werden.*
- *Weiter ist*
$$L(TM) := \{w \in E^* \mid s_0w\square \vdash^* us_fv, s_f \in F, u, v \in A^*\}$$
die von der Turingmaschine akzeptierte Sprache L.

Turingmaschinen lassen sich nutzen, um sowohl die Typ-0-Sprachen als auch die kontextsensitiven Sprachen charakterisieren (letzte über Beschränkung der Größe des Arbeitsbandes). Außerdem ist einfach zu erkennen, das sie mächtiger als endliche Automaten und Kellerautomaten sind.

Ein endlicher Automat ist ein Spezialfall einer Turingmaschine, der Schreib-Lesekopf nutzt ausschließlich die Lese-Funktion und der Kopf bewegt sich nur nach rechts. Formal bedeutet das, dass die Übergangsfunktion die Form $\delta(s, a) = (s', a, R)$ für alle $a \in E$ hat.

Die Simulation von Kellerautomaten ist nach folgender Idee möglich. Das Schreib-Leseband der Turingmaschine wird "zweigeteilt": Rechtsseitig steht das Eingabewort. Die Maschine trägt auf dem ersten freien Platz links ein Zeichen ein, das den Boden des Kellers markiert und simuliert in späteren Schritten den Kellerspeicher. Dabei müssen die Übergänge unter der Übergangsfunktion des Kellerautomaten, die ja mehrere Zeichen an die Kellerspitze schreiben können, durch mehrere Schritte der Turingmaschine simuliert werden. Diese kann sich auch merken, an welcher Stelle sie zuletzt die Eingabe gelesen hat, indem sie die schon verarbeiteten Zeichen durch ein Zeichen aus dem Arbeitsalphabet ersetzt. Das oberste Kellersymbol ist leicht zu finden, da links davon ein Blank steht. Ist der Keller leer, so geht die Turingmaschine in einen Endzustand über.

Damit ist gezeigt, das die Turingmaschine mindestens so leistungsfähig ist wie die bisherigen Automaten. Wir wollen nun zeigen, dass man mittels einer Turingmaschine auch die Sprache $L = \{a^n b^n c^n \mid n > 0\}$ erkennt.

Beispiel 2.6 (Turingmaschine zu $L = \{a^n b^n c^n \mid n > 0\}$)**.** *Wir wollen die Grundidee der Konstruktion nach Sander, Stucky und Herschel S. 194f. erläutern.*

Dazu definieren wir $TM := (S, E, A, \delta, s_0, \square, F)$ *wie folgt*

$$\begin{aligned} S &= \{s_0, s_1, s_2, s_3, s_4, s_f\} \\ A &= \{a, b, c, 0, 1, 2, \square\} \\ F &= \{s_f\} \end{aligned}$$

Die Übergangsfunktion definieren wir später in einer Tabelle.

Die Maschine steht über dem ersten Zeichen des Wortes. Sodann ersetzt sie das am weitesten links stehende a *durch* 0 *und geht in den Folgezustand über. Dasselbe geschieht mit dem am weitesten links stehenden* b *(wird ersetzt durch* 1*) bzw.* c *(wird ersetzt durch* 2*).*

Dann beginnt durch Zurückspringen in den Ausgangszustand die Suche von neuem. Wird kein a *mehr gefunden, darf auch kein* b *oder* c *mehr auftreten; ist noch ein* a *vorhanden, muss das gleiche auch für* b *und* c *gelten.*

Interessant ist der Zustand s_3*. Hier läuft der Automat zurück bis zur ersten Null von rechts und geht dann einen Schritt nach rechts. Kommen keine* b *und* c *mehr vor, geht der Kopf bis zum rechten Rand und beim ersten Blank in den Endzustand.*

	a	b	c	0	1	2	$\square$
s_0	$(s_1, 0, R)$				$(s_4, 1, R)$		
s_1	(s_1, a, R)	$(s_2, 1, R)$			$(s_1, 1, R)$		
s_2		(s_2, b, R)	$(s_3, 2, L)$			$(s_2, 2, R)$	
s_3	(s_3, a, L)	(s_3, b, L)		$(s_0, 0, R)$	$(s_3, 1, L)$	$(s_3, 2, L)$	
s_4					$(s_4, 1, R)$	$(s_4, 2, R)$	$(s_f, \square, N)$
s_f							

Ist die Übergangsfunktion nicht definiert, bleibt der Automat stehen.

Als Beispiel soll das Wort $abcc$ geteset werden:

$s_0abcc \vdash 0s_1bcc \vdash 01s_2cc \vdash 0s_312c \vdash s_3012c \vdash 0s_012c \vdash 01s_42c \vdash 012s_4c$ halt

Also wird $abcc$ nicht akzeptiert). Wäre das letzte c nicht vorgekommen, wäre der Kopf auf dem Blank in den Endzustand übergegangen. Das Wort abc wäre daher akzeptiert worden.

Jetzt wollen wir Turingmaschinen auch zur Berechnung von Funktionen $f : E^* \to E^*$ und von Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ einsetzen. Für die Notation der Zahleneingaben bieten sich verschiedene Möglichkeiten an. Naheliegend sind natürlich die dezimale oder auch die binäre Notation $bin(n)$. Eine andere Möglichkeit besteht darin, die Variable durch die unär zu beschreiben. Hier schreibt man für n z.B. dann n-mal ein Zeichen |. Außerdem müssen wir bei der Eingabe eines Vektors zwischen den einzelnen Komponenten ein Trennsymbol haben, z.B. ein Blank. Der Kopf steht wiederum ganz links über dem ersten Zeichen. Nach Ausführung der Berechnung enthält das Band den Ergebniswert und der Kopf steht wiederum über dem ersten Zeichen.

Definition 2.7. *Eine Funktion $f : E^* \dashrightarrow E^*$ heißt Turing-berechenbar, falls es eine deterministische Turingmaschine TM gibt derart, dass für $x, y \in E^*$ genau dann $f(x) = y$ gilt, wenn es einen Endzustand $s' \in F$ und Worte u, v gibt mit*

$$s_0 x \quad \vdash_{TM}^* \quad us'v$$

und y das längste Präfix von v über E ist.

Eine Funktion $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ heißt Turing-berechenbar, falls es eine deterministische Turingmaschine TM gibt derart, dass für $n_1, ..., n_k, m$ aus $\mathbb{N}$ genau dann $f(n_1, ..., n_k) = m$ gilt, wenn es einen Endzustand $s' \in F$ und Worte u, v gibt mit

$$s_0 bin(n_1)\square bin(n_2)\square ...\square bin(n_k)\square \quad \vdash_{TM}^* \quad us'v$$

und $bin(m)$ das längste Präfix von v über $\{0, 1\}$ ist. Dabei sei $bin(n)$ die Binärdarstellung von n.

Beispiel 2.8. *a) Wir haben bereits in Beispiel 2.4 gesehen, dass die folgende Funktion f (bezüglich der Binärdarstellung) berechenbar ist:*

$$f(n) = S(n) = n + 1$$

b) Die nirgends definierte Funktion $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ ist ebenfalls berechenbar.

Sie wird zum Beispiel von einer Turingmaschine mit einem Startzustand s_0, einem davon verschiedenen Endzustand s_1 und den Übergängen $\delta(s_0, a) = (s_0, a, N)$ berechnet.

Beispiel 2.9. *Wir hatten festgestellt, dass die Typ-0-Sprachen genau diejenigen Sprachen sind, die von nichtdeterministischen Turingmaschinen erkannt werden. Außerdem hatten wir bemerkt, dass man die nichtdeterministischen Turingmaschinen durch deterministische Turingmaschinen simulieren kann. Also ist eine Sprache genau dann vom Typ 0, wenn es eine deterministische Turingmaschine gibt, die sie erkennt.*

Ein Wort x gehört genau dann zu der von einer Turingmaschine erkannten Sprache, wenn man von der Anfangskonfiguration $s_0 x$ aus eine Konfiguration der Form $us_f v$ mit beliebigen Wörtern und einem Endzustand s_f erreichen kann. Da die Maschine ein Eingabewort also akzeptiert, sobald sie in einen Endzustand gelangt, ist das Verhalten einer Maschine nach Erreichen eines Endzustandes für das Akzeptieren oder Nicht-Akzeptieren also ohne Belang. Daher hatten wir Turingmaschinen so definiert, dass sie nach Erreichen eines Endzustandes nicht weiterrechnen (können).

Wir können die Maschine auch zum Beispiel so modifizieren, dass sie bei Erreichen eines Endzustandes erst zwei Blanks schreibt und den Kopf auf das rechte dieser Blanks setzt, d.h. sie hält mit $...\square s_f \square...$ (wenn sie anhält)

Damit haben wir gezeigt, dass eine Sprache L genau dann vom Typ 0 ist, wenn die Funktion $g : E^* \to E^*$ mit $g(x) = \varepsilon$ für x aus L und $g(x)$ undefiniert für alle anderen Wörter x, berechenbar ist.

Schließlich wollen wir noch einmal überlegen, was passieren kann, wenn die Maschine ein Wort x *nicht* akzeptiert. Dann wird sie niemals eine Konfiguration $us_f v$ mit beliebigen Wörtern und einem Endzustand s_f erreichen. Dafür kann es zwei Gründe geben: Entweder sie hört nie auf zu rechnen, erreicht aber niemals einen Endzustand. Oder sie bleibt in einer Konfiguration, die keine Endkonfiguration ist, stecken, weil es zu dem gerade aktuellen Zustand und dem gerade unter dem Lese-Schreibkopf befindlichen Symbol keinen Übergang gibt. Diesen zweiten Fall können wir nun ausschließen, indem wir einen neuen Zustand s_l einführen, und zu jedem Nicht-Endzustand s und jedem Arbeitssymbol a derart, dass es keinen Übergang von (s, a) aus gibt, einen Übergang $\delta(s, a) = (s_l, a, N)$ einführen und zu jedem Arbeitssymbol a einen Übergang $\delta(s_l, a) = (s_l, a, N)$ einführen.

Dann führt jedes Wort, das nicht akzeptiert wird, zu einer nie endenden Berechnung. Also ist eine Sprache genau dann vom Typ 0, wenn es eine deterministische Turingmaschine gibt, die genau bei den Wörtern aus L (als Eingabe) anhält.

All diese Überlegungen kann man natürlich auch auf Mengen von Zahlen übertragen. Wir kommen später darauf zurück.

Wir wollen nun eine Mehrband-Turingmaschine MTM mit $k \geq 1$ Bändern und Schreibköpfen definieren, die unabhängig voneinander operieren. Formal wird man die Übergangsfunktion $\delta : S \times A^k \to S \times A^k \times \{L, R, N\}^k$ ansetzen und den Begriff der Konfiguration und Übergangsrelation anpassen.

Man kann nun zeigen, dass diese Mehrband-Turingmaschine, die ja zunächst stärker als eine Einbandmaschine erscheint und diese durch Stilllegung von $k-1$ Bändern simulieren kann, umgekehrt auch durch eine Einbandmaschine simuliert werden kann.

Satz 2.10. *Zu jeder Mehrbandmaschine MTM gibt es eine Einbandmaschine TM, die dieselbe Sprache akzeptiert, bzw. dieselbe Funktion berechnet.*

Beweis: Sei eine Mehrbandmaschine MTM mit k Bändern gegeben, die wir uns mit 1 bis k durchnummeriert denken. Sei A ihr Arbeitsalphabet. Dann wollen wir eine Turingmaschine TM konstruieren, die die MTM simuliert.

Dazu wird das Schreib-Leseband der TM in $2k$ Spuren unterteilt, die wir uns mit 1 bis $2k$ durchnummeriert denken. Dabei soll die Spur $2i-1$ den Inhalt von Band i der MTM enthalten. Die Spur $2i$ soll nur an einer Stelle das Zeichen $*$ und sonst nur Blanks enthalten. Das Zeichen $*$ soll anzeigen, dass der Lese-Schreibkopf des i-ten Bandes der MTM gerade an dieser Stelle steht.

Der Kopf der Einbandmaschine muss jetzt die Kopfpositionen der MTM sukzessive abfahren. Daher wählen wir für die TM das Arbeitsalphabet $A' = A \cup (A \cup \{*\})^{2k}$.

Die TM simuliert die MTM nun folgendermaßen: Gestartet wird mit der Eingabe $a_1...a_n$ aus E^*. Dann erzeugt TM zunächst die Startkonfiguration von MTM in der Spurendarstellung.

Die Startkonfiguration sieht so aus, dass das Eingabewort $a_1...a_n$ auf Spur 1 steht, alle anderen ungeraden Spuren leer sind, und alle geraden Spuren an der ersten Position unter dem Lese-Schreibkopf einen Stern haben und ansonsten leer sind.

Durch Laufen nach rechts bis zu dem am weitesten rechts stehenden $*$ oder evtl. eine Position darüber hinaus kann dann der Lese-Schreibkopf der TM alle Änderungen vornehmen, die die Mehrbandmaschine in einem Schritt vornimmt. Dazu muss er in den Tupeln lesen, schreiben und eventuell die Sternmarken löschen und neu setzen (dazu muss er eventuell bei einer Linksbewegung eines Lese-Schreibkopfes noch einen Extraschritt nach links und rechts machen).

Danach kehrt der Kopf wieder auf die am weitesten links stehende Marke zurück, und es kann der nächste Schritt der MTM simuliert werden.

Die Verwendung mehrerer Bänder bringt dennoch wesentlich Vorteile: Jedes Band stellt im Prinzip ein unendlich langes Register dar. Besitzt die Maschine nur eines dieser Register, wird sie Rechnungen, die n Eingaben erfordern, nur in der angegebenen Form mit Hintereinander-Eintrag ausführen können.

Hier werden die einfachsten Beispiele schon sehr unelegant in der Formulierung, da die TM die meiste Zeit damit beschäftigt ist, zwischen den einzelnen Eingaben hin- und herzufahren.

Bei einer Mehrbandmaschine werden die Eingaben untereinander geschrieben und können quasi gleichzeitig verarbeitet werden.

Oft möchte man das Verhalten einer Einbandmaschine in einer k-Band-Turingmaschine auf Band i simulieren. Die anderen Bänder bleiben unverändert. Dies wird dann durch $MTM(i,k)$ beschrieben. Es ist damit so etwas wie eine Projektionsmaschine definiert mit einer Projektion auf die i-te Koordinate. Die Übergangsfunktion sieht dann z.B. folgendermaßen aus: $\delta(s, a_1, ..., c, ..., a_k) = (s', a_1, ..., d, ..., a_k, N, ..., R, ..., N)$.

Zur Erleichterung der Bezeichnung wird k oft einfach groß genug gewählt und dann in der Schreibweise weggelassen.

Wir wollen nun zeigen, dass man einige Grundoperationen in Turingmaschinen implementieren kann und mit Turingmaschinen 'programmieren' kann.

Wir hatten in Beispiel 2.4 bereits gesehen, dass man mit einer Einbandmaschine Zahlen inkrementieren kann. Ein Mehrbandmaschine, die dies auf Band i macht, wollen wir mit

Band i := Band $i+1$

beschreiben. Man kann auch leicht eine Maschine angeben, die Zahlen dekrementiert, d.h. die n auf $n-1$ abbildet, wenn $n > 0$, und auf 0, wenn $n = 0$. Ein Mehrbandmaschine, die dies auf Band i macht, sei mit

Band i := Band $i - 1$

beschrieben. Weiterhin möchten wir Elementaroperationen wie

Band $i := 0$

und

Band $i :=$ Band j

einführen. Wir weisen darauf hin, dass eine formale Behandlung all dieser Implementierungen es eigentlich nötig macht, die Turingtabellen anzugeben. Da die Operationen und ihre Verwirklichung jedoch einsichtig sind, wollen wir es unterlassen.

Ein weiterer wichtiger Aspekt ist die Ausführung mehrerer Operationen hintereinander und damit die Hintereinanderschaltung von Turingmaschinen.

Zu zwei (Einband- oder Mehrband-)Turingmaschinen

$$M_1 = (S_1, E, A_1, \delta_1, s_{01}, \square, F_1)$$

und

$$M_2 = (S_2, E, A_2, \delta_2, s_{02}, \square, F_2)$$

von denen wir annehmen wollen, dass die Zustandsmengen S_1 und S_2 keine gemeinsamen Elemente haben, schreiben wir

oder

$M_1; M_2$

und meinen damit die Turingmaschine

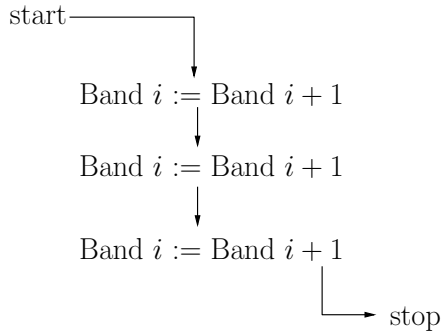
$$M = (S_1 \cup S_2, E, A_1 \cup A_2, \delta, s_{01}, \square, F_2)$$

mit

$$\delta = \delta_1 \cup \delta_2 \cup \{((s_f, a), (s_{02}, a, N)) \mid s_f \in F_1, a \in A_1\}$$

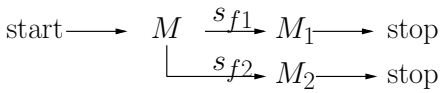
Dabei sind die Übergangsfunktionen als Relationen über kartesischen Produkten geschrieben.

Beispiel: Die durch:



beschriebene Maschine realisiert die Operation ‘Band i := Band $i + 3$’.

Weiteres Beispiel: Die Maschine



realisiert eine Verzweigung über die Endzustände s_{f1} und s_{f2} der Maschine M und ein Fortfahren mit entweder M_1 oder M_2.

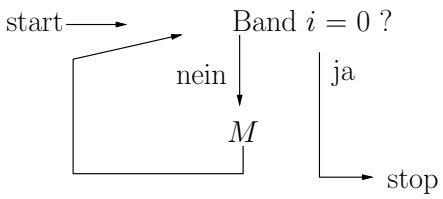
Nun wollen wir eine bedingte Verzweigung simulieren, bei der getestet wird, ob 0 (als Zahl) auf dem Band steht. Wir nennen die folgende Maschine “Band = 0?”. Dazu wählen wir die Zustandsmenge $S := \{s_0, s_1, ja, nein\}$, wobei die Endzustände ja und $nein$ seien. Das Arbeitsalphabet sei hier $A = \{0, 1, \square\}$.

Die Übergangsfunktion wird in der folgenden Tafel angegeben:

δ	1	0	$\square$
s_0	$(nein, 1, N)$	$(s_1, 0, R)$	$(ja, \square, N)$
s_1	$(nein, 1, L)$	$(s_1, 0, R)$	$(ja, \square, L)$

Ähnlich wie in dem vorigen Beispiel kann man damit eine bedingte Verzweigung programmieren. Die gleiche Operation kann auch in der Form ‘Band$i = 0$?’ mit einem entsprechenden Test auf Band i vorgenommen werden.

Schließlich können wir eine WHILE-Schleife simulieren:



Dies wollen wir mit ‘WHILE Band $i \neq 0$ DO M’ abkürzen.

3 LOOP- und WHILE-Berechenbarkeit

In diesem Kapitel wollen wir uns mit anderen möglichen Definitionen von berechenbaren 'Zahlenfunktionen' (= hier benutzt für Funktionen, die natürliche Zahlen oder Tupel davon auf natürliche Zahlen abbilden) beschäftigen. Zuerst führen wir sogenannte LOOP-Programme ein, etwas später dann WHILE-Programme. Mit beiden Programmenarten kann man Zahlenfunktionen berechnen. Alle durch LOOP-Programme berechenbaren Funktionen können auch durch WHILE-Programme berechnet werden. In einem späteren Kapitel werden wir aber sehen, dass die Umkehrung nicht einmal dann gilt, wenn man nur totale Funktionen betrachtet. Aber wir werden zeigen, dass die durch Turingmaschinen berechenbaren Zahlenfunktionen genau mit den durch WHILE-Programme berechenbaren Funktionen übereinstimmen. Damit haben wir eine weitere einfache Möglichkeiten kennen gelernt, die berechenbaren Zahlenfunktionen zu charakterisieren.

Zuerst führen wir die LOOP-Programme ein. Die Idee ist, dass man mit diesen Programmen auf Variablen rechnen darf, die als Werte natürliche Zahlen annehmen, und dass einem als Programmkonstrukt LOOP-Schleifen zur Verfügung stehen, die oft auch for-Schleifen genannt werden.

Die syntaktischen Grundbausteine von LOOP-Programmen sind:

- Variablen: $x_0\ x_1\ x_2\ \ldots$
- Konstanten: $0\ 1\ 2\ \ldots$ (also für *jede* natürliche Zahl eine Konstante)
- Trennsymbole und Operationszeichen: $;\quad :=\quad +\quad -$
- Schlüsselwörter: `LOOP DO END`

LOOP-Programme sind dann:

- Jede Wertzuweisung $x_i := x_j + c$ und $x_i := x_j - c$ ist ein LOOP-Programm.
- Sind P_1 und P_2 LOOP-Programme, dann ist auch $P_1; P_2$ ein LOOP-Programm.
- Ist x Variable und P LOOP-Programm, dann ist auch `LOOP` x `DO` P `END` ein LOOP-Programm.
- Weitere Konstruktionen sind nicht zugelassen.

Die Semantik (Bedeutung) von LOOP-Programmen ergibt sich aus:

- Die Werte der Variablen x_i sind (beliebig) aus $\mathbb{N}$.
- Bei $x_i := x_j + c$ erhält x_i den Wert $x_j + c$ analog zu üblichen Programmiersprachen...
- Bei $x_i := x_j - c$ erhält x_i den Wert $x_j - c$, falls $x_j \geq c$, ansonsten den Wert 0. (Damit bleiben alle Werte in $\mathbb{N}$.)
- Bei $P_1; P_2$ wird erst P_1 ausgeführt, dann P_2.
- Bei `LOOP` x `DO` P `END` wird das Programm P so oft ausgeführt, wie der Wert von x *zu Beginn* der LOOP-Anweisung angibt.

Mit LOOP-Programmen kann man Zahlenfunktionen berechnen:

Definition 3.1 (LOOP-Berechenbarkeit)**.** *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt LOOP-berechenbar, falls es ein LOOP-Programm P gibt, so dass P gestartet mit der beliebigen Eingangsbelegung $n_1, ..., n_k$ aus $\mathbb{N}$ in den Variablen $x_1, ..., x_k$ und 0 in allen anderen Variablen mit dem Wert $f(n_1, ..., n_k)$ in der Variablen x_0 stoppt.*

Zwei wichtige Eigenschaften der LOOP-berechenbaren Funktionen müssen hervorgehoben werden. Erstens sind alle LOOP-berechenbaren Funktionen total, d.h. auf allen Zahlenvektoren aus $\mathbb{N}^k$ (für ein festes k natürlich) definiert, da jedes LOOP-Programm nach endlich vielen Schritten ('Schritt' = Ausführung einer Wertzuweisung) anhält, wie man per Induktion über den Aufbau der Programme formal beweisen kann: Die Aussage gilt offenbar für Programme der Form $x_i := x_j \pm c$. Gilt sie für P_1 und für P_2, dann auch für $P_1; P_2$. Gilt sie für P, dann auch für `LOOP`x_i`DOPEND`. Letzteres gilt dank der speziellen Semantik der `LOOP`-Anweisung. `for`-Anweisungen wie sie z.B. Programmiersprachen wie `C` und `Java` erlauben, haben eine andere Semantik — hier kann man mühelos Endlosschleifen produzieren, etwa durch `for (x=0;x<1;x=x) x=x-1;`.

Zweitens stellt sich die Frage, ob alle totalen Turing-berechenbaren Funktionen auch LOOP-berechenbar sind. Diese Frage muss verneint werden, wie wir später sehen werden.

Wir wollen nun zeigen, dass einige einfache Funktionen und Konstrukte in der LOOP-Sprache realisierbar sind. Wir wollen zunächst dazu äquivalente LOOP-Programm angeben und dann diese Funktionen/Konstrukte dann auch als Abkürzungen in anderen LOOP-Programmen verwenden.

- Eine Kopierzuweisung '$x_i := x_k$' ergibt sich zum Beispiel durch die Zuweisung '$x_i := x_k + 0$' mit der Konstanten $c = 0$.
- Eine Zuweisung '$x_i := c$' erhält man durch '$x_i := x_k + c$', wenn man dabei eine Variable x_k nutzt, die immer den Wert 0 hat.
- Bedingte Ausführungen '`IF` $x_k = 0$ `THEN` Q `END`' sind implementierbar durch

  $y := 1;$
  `LOOP` x_k `DO` $y := 0$ `END;`
  `LOOP` y `DO` Q `END`

  mit einer ansonsten nicht benutzten Variable y
- Als Übungsaufgabe: '`IF` $x_k = 0$ `THEN` Q `ELSE` R `END`'
- Additionen '$x_i := x_j + x_k$' mit $i \neq j$:

  $x_i := x_k;$
  `LOOP` x_j `DO` $x_i := x_i + 1$ `END`
- Der Spezialfall '$x_0 := x_1 + x_2$' berechnet damit die Additionsfunktion $+ : \mathbb{N}^2 \to \mathbb{N}$ im Sinne von Definition 3.1
- Der Fall $i = j$, d.h. '$x_i := x_i + x_k$', ist sogar noch einfacher:

  `LOOP` x_k `DO` $x_i := x_i + 1$ `END`
- Die Subtraktion '$x_i := x_j - x_k$' (wieder mit der Konvention, dass der Wert 0 herauskommen soll, wenn $x_j < x_k$ ist) kann ähnlich berechnet werden.
- Multiplikationen '$x_i := x_j \cdot x_k$'

  $x_i := 0;$
  `LOOP` x_j `DO` $x_i := x_i + x_k$ `END`

  was wiederum nur eine Abkürzung ist für

  $x_i := 0;$
  `LOOP` x_j `DO`
  `LOOP` x_k `DO` $x_i := x_i + 1$ `END`
  `END`

- Die 'Signum'-Funktion $sg : \mathbb{N} \to \mathbb{N}$ mit

$$sg(x) := \begin{cases} 1 \text{ falls } x > 0 \\ 0 \text{ falls } x = 0 \end{cases}$$

ist realisierbar über

$x_0 := 1;$
IF $x_1 = 0$ THEN $x_0 := 0$ END

- Die inverse Signum-Funktion $\overline{sg}(x) := 1 - sg(x)$ kann ähnlich berechnet werden.

- Die 'Gleichheits'-Funktion $se(x, y)$ mit

$$se(x, y) := \begin{cases} 1 \text{ falls } x = y \\ 0 \text{ falls } x \neq y \end{cases}$$

kann über die Funktionen sg und $\overline{sg}$ realisiert werden.

- Ganzzahldivision $x_i := x_j$ DIV x_k mit $x_k > 0$:
Dazu zunächst 'IF $x_k \leq x_j$ THEN Q END' über

$x_0 := x_k - x_j;$
$x_1 := \overline{sg}(x_0);$
LOOP x_1 do Q END

Damit erhalten wir '$x_i := x_j$ DIV x_k' durch:

$x_i := 0;$
LOOP x_j DO
  IF $x_k \leq x_j$ THEN $x_i := x_i + 1$ END;
  $x_j := x_j - x_k;$
END

- Zur Übung realisieren Sie z.B. die Modulo-Funktion MOD als LOOP-Programm (wird später noch benötigt...).

- Weitere Übung: Cantor'sche Bijektion $\langle \cdot, \cdot \rangle : \mathbb{N}^2 \to \mathbb{N}$

$$\langle x, y \rangle \;:=\; y + \sum_{i \leq x+y} i \;=\; y + \frac{(x+y)(x+y+1)}{2}$$

$\langle x, y \rangle$	0	1	2	3	4	5	...
0	0	2	5	9	14	20	...
1	1	4	8	13	19	26	...
2	3	7	12	18	25	33	...
3	6	11	17	24	32	41	...
4	10	16	23	31	40	50	...
5	15	22	30	39	49	60	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Sowohl die Bijektion von $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ als auch die einzelnen Komponenten $p_1, p_2 : \mathbb{N} \to \mathbb{N}$ der Umkehrfunktion (d.h. $\langle p_1(z), p_2(z) \rangle = z$) sind LOOP-berechenbar:

- Berechnung von $y + \sum_{i \leq x+y} i$ benötigt i.W. LOOP-Schleife mit Additionen

- Berechnung der Komponenten $p_1(z), p_2(z)$:
  - Suche (von $k = 0, ..., z$) nach größtem k mit $\sum_{i \leq k} i \leq z$.
  - Dann $p_2(z) = z - \sum_{i \leq k}$ und $p_1(z) = z - k$.

Nun wollen wir die LOOP-Programme um ein weiteres Programmkonstrukt erweitern, die WHILE-Schleife, und erhalten damit WHILE-Programme.

Die syntaktischen Grundbausteine von WHILE-Programmen sind dabei:

- unverändert wie bei Loop: Variablen $x_0\ x_1\ x_2 \ldots$, Konstanten $0\ 1\ 2 \ldots$, Trennsymbole und Operationszeichen
  ; := + −
- Schlüsselwörter: `LOOP DO END`
- neues, zusätzliches Schlüsselwort: `WHILE`

WHILE-Programme sind dann wie folgt definiert:

- Jede Wertzuweisung $x_i := x_j + c$ und $x_i := x_j - c$ ist ein WHILE-Programm.
- Sind P_1 und P_2 WHILE-Programme, dann ist auch $P_1; P_2$ ein WHILE-Programm.
- Ist x Variable und P WHILE-Programm, dann sind auch `LOOP` x `DO` P `END` und `WHILE` x `DO` P `END` WHILE-Programme
- Weitere Konstruktionen sind nicht zugelassen.

Jedes LOOP-Programm ist damit auch ein WHILE-Programm.

Die Semantik von WHILE-Programmen wird analog zu LOOP definiert.

Wichtig ist dabei die Semantik der neuen WHILE-Schleife:

- Bei `WHILE` x `DO` P `END` wird P ausgeführt, bis der (sich evtl. ändernde!) Inhalt von x zu Beginn des Schleifenrumpfes P den Wert. 0 hat.
- Bei `LOOP` x `DO` P `END` wurde P so oft ausgeführt, wie der Wert von x *zu Beginn* der LOOP-Anweisung angibt.

Also:

- LOOP-Programme müssen im Voraus wissen, wie oft eine Schleife ausgeführt wird.
- WHILE-Programme können hingegen unbeschränkt ‘suchen’...

Auch mit WHILE-Programmen kann man Zahlenfunktionen berechnen:

Definition 3.2 (WHILE-Berechenbarkeit)**.** *Eine Funktion* $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ *heißt WHILE-berechenbar, falls es ein WHILE-Programm* P *gibt, so dass* P *gestartet mit der beliebigen Eingangsbelegung* $n_1, ..., n_k$ *aus* $\mathbb{N}$ *in den Variablen* $x_1, ..., x_k$ *und* 0 *in allen anderen Variablen mit dem Wert* $f(n_1, ..., n_k)$ *in der Variablen* x_0 *nach endlich vielen Schritten stoppt, falls* $f(n_1, ..., n_k)$ *definiert ist, und nie stoppt, falls* $f(n_1, ..., n_k)$ *nicht definiert ist.*

Bei WHILE-Programmen kann man auf die LOOP-Anweisung sogar komplett verzichten, 'LOOP x_j DO P END' wird simuliert durch

$x_k := x_j;$
WHILE x_k DO $x_k := x_k - 1;$ P END

Dabei sei x_k eine Variable, die im Programm P nicht verwendet wird.

Alle LOOP-berechenbaren Funktionen sind WHILE-berechenbar.

Die Umkehrung gilt nicht, z.B. für die partielle Funktion f mit

$$f(x) := \begin{cases} 42 & \text{falls } x = 0 \\ undefiniert & \text{falls } x > 0 \end{cases}$$

f ist sicher nicht LOOP-, aber WHILE-berechenbar:

WHILE x_1 DO $x_1 := x_1 + 1$ END; $x_0 := 42$

Es gibt sogar *totale* Funktionen, die nicht LOOP-, aber WHILE-berechenbar sind (z.B. Ackermann-Funktion)!

Wir haben im letzten Kapitel gesehen, dass man alle Programmkonstrukte, die in WHILE-Programmen vorkommen, durch Mehrband-Turingmaschinen realisieren kann. Zum Beispiel kann man den Inhalt der i-ten Variablen x_i binär auf Band i der Maschine speichern. Außerdem haben wir gesehen, dass man Mehrband-Turingmaschine durch Einband-Turingmaschinen, also gewöhnliche Turingmaschinen simulieren kann. Daher gilt der folgende Satz:

Satz 3.3. *Jede WHILE-berechenbare Zahlenfunktion ist Turing-berechenbar.*

Schließlich möchten wir noch den Äquivalenzbeweis vollenden, indem wir den folgenden Satz beweisen.

Satz 3.4. *Turing-berechenbare Zahlenfunktionen sind WHILE-berechenbar.*

Zum Beweis starten wir mit einer Turingmaschine $TM = (S, E, A, \delta, s_0, \square, F)$ zur Berechnung einer Zahlenfunktion f. Wir simulieren sie durch ein WHILE-Programm aus drei Teilen.

Der erste Teil transformiert die Zahlen $n_1, ..., n_l$ der Eingabe von f in Binärdarstellung und stellt die Startkonfiguration der TM mittels dreier Zahlenvariablen x, y, z dar, die gleich erklärt werden.

Der zweite Programmteil simuliert die Berechnung der Turingmaschine Schritt für Schritt durch entsprechende Änderung der drei Zahlenvariablen x, y, z. Dabei wird die Konfiguration der TM zu jedem Zeitpunkt komplett in den drei Zahlenvariablen gespeichert.

Der dritte Teil erzeugt schließlich am Ende der Rechnung aus den drei Zahlenvariablen, die dann die Endkonfiguration der TM codieren, den Ausgabewert (eine Zahl) in der Variablen x_0.

Wir beschreiben dies etwas genauer.

Sei die Zustandsmenge $S = \{s_0, ..., s_k\}$ und das Arbeitsalphabet $A = \{a_1, ..., a_m\}$, sowie b eine Zahl mit $b > m$.

Dann wird eine Konfiguration

$$a_{i_1} ... a_{i_p} s_l a_{j_1} ... a_{j_q}$$

der Turingmaschine dadurch beschrieben, dass die Variablen die Werte $x = (i_1 ... i_p)_b, y = (j_q ... j_1)_b$ und $z = l$ annehmen.

Dabei sei hier

$$(i_1 ... i_p)_b = \sum_{l=1}^{p} i_l \cdot b^{p-l}$$

also ist x eine Codierung der Ziffern $i_1, \ldots i_p$ in einer einzigen Zahl. Bei y sei nur die Reihenfolge der Ziffern andersherum. Das heißt, in x speichern wir die Inschrift des Bandes links vom Lese-Schreibkopf. In y speichern

wir die Inschrift des Bandes ab der Position unter dem Lese-Schreibkopf und rechts davon. In z speichern wir den aktuellen Zustand.

Das mittlere, zweite Programmstück simuliert nun eine Schleife mit einer einzigen großen Verzweigung:

```
WHILE Zustand s ist kein Endzustand DO
    Bestimme Zeichen a unter dem Kopf;
    IF (s = s_0) und (a = a_1):
       simuliere δ(s_0, a_1) in x, y, z;
    ELSE IF...
       ...
    ELSE IF (s = s_k) und (a = a_m):
       simuliere δ(s_k, a_m) in x, y, z;
END WHILE;
```

Die Bestimmung des *Zeichens* a können wir durch die Berechnung der *Zahl* 'y MOD b' simulieren; damit entspricht der Vergleich '$a = a_j$' dem Test 'y MOD $b = j$'. Der Vergleich '$s = s_i$' zweier Zustände entspricht dem Zahlenvergleich '$z = i$'.

Es gibt insgesamt $(k+1) * m$ verschiedene Fälle, die abhängig von den Werten von s (es gibt $k+1$ Zustände s) und a (es gibt m Symbole a im Arbeitsalphabet A) auszuwählen sind.

Bei der Simulation der Übergangsfunktion $\delta(s_i, a_j) = (s'_i, a'_j, B)$ muss die Folgekonfiguration in den Variablen x, z, y erstellt werden:

Im Folgenden betrachten als Beispiel wir den Fall $B = L$. Hier wird der Zustand angepasst, also die Variable z entsprechend geändert, y wird zunächst um den letzten Eintrag j verkürzt, dann um j' verlängert und schließlich weiter um den x-Anteil verlängert, was die Kopfbewegung nach links simuliert. Schließlich wird x entsprechend verkürzt.

Bei $x \neq 0$, d.h. links vom Kopf der Turingmaschinen stehen noch Zeiochen, geschieht mit den folgenden Anweisungen, die durch WHILE-Programme simuliert werden können:

$z := i'$;
$y := y$ DIV b;
$y := b * y + j'$;
$y := b * y + (x$ MOD $b)$;
$x := x$ DIV b;

Im Fall $x = 0$, also keinen weiteren Zeichen links vom Kopf der Maschine, muss ein Blank 'eingefügt' werden, was durch $y := b * y + j_\square$ an Stelle von $y := b * y + (x$ MOD $b)$ möglich ist (wobei $j_\square$ der Index von $\square$ in A sei).

Ist die Maschine im Endzustand angelangt, so wird in den dritten Teil des WHILE-Programmes verzweigt. Die Konstruktionen dieses und des ersten Teils hängen nicht von der Übergangsfunktion d der Turingmaschine ab.

Damit haben wir auch Satz 3.3 bewiesen.

Als Folgerung aus der Konstruktion ergibt sich:

Satz 3.5. *Jedes Turing-Programm kann durch ein WHILE-Programm mit nur einer einzigen WHILE-Schleife berechnet werden.*

Achtung: es wird in der Simulation nur eine einzige WHILE-Schleife gebraucht, aber viel möglicherweise viele LOOP-Schleifen. Dies führt zu dem folgenden Satz.

Satz 3.6 (Kleenesche Normalform für WHILE-Programme)**.** *Zu jedem WHILE-Programm gibt es ein äquivalentes WHILE-Programm mit nur einer einzigen WHILE-Schleife.*

Zum Beweis transformiere man erst ein WHILE-Programm für die jeweilige WHILE-berechenbare Funktion in ein Turing-Programm (wir haben oben gesehen, wie das geht) und dieses dann wieder in ein WHILE-Programm mit nur einer WHILE-Schleife.

4 Primitiv rekursive und μ-rekursive Funktionen

Wir haben die berechenbaren Zahlenfunktionen bisher durch Turingmaschinen und durch WHILE-Programme charakterisiert. Außerdem haben wir die Klasse der durch LOOP-Programme berechenbaren Funktionen eingeführt. In diesem Abschnitt wollen wir eine weitere Charakterisierung der berechenbaren Zahlenfunktionen kennenlernen. Wir werden die Klasse der sogenannten primitiv rekursiven Funktionen und die Klasse der sogenannten μ-rekursiven Funktionen definieren und beweisen, dass die primitiv rekursiven Funktionen genau die LOOP-berechenbaren Funktionen sind und die μ-rekursiven genau die WHILE-berechenbaren. Beide Funktionenklassen werden dadurch definiert, dass wir von einer Menge von Grundfunktionen ausgehen und eine Zahlenfunktion primitiv rekursiv bzw. μ-rekursiv nennen werden, wenn man sie aus den Grundfunktionen durch endlich häufige Anwendung einiger einfacher Operatoren bzw. Erzeugungsschemata gewinnen kann. Dabei lassen wir bei den μ- rekursiven Funktionen einen Operator mehr zu als bei den primitiv rekursiven Funktionen.

Wir werden im Folgenden aus formalen Gründen auch nullstellige Zahlenfunktionen betrachten, also Funktionen $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ mit $k = 0$. Dies sind einfach Funktionen ohne Eingabe, die einen festen Wert liefern, also Konstanten. Man beachte, dass es einen formalen Unterschied macht, ob man die nullstellige Funktion mit Wert 7 betrachtet, also die Konstante 7, oder die zweistellige Funktionen, die bei jeder Eingabe (x, y) den Wert 7 ausgibt.

Die folgenden Funktionen sind unsere Grundfunktionen, die wir von vornherein primitiv rekursiv nennen wollen:

- $Z : \mathbb{N}^0 \to \mathbb{N}$, die nullstellige Funktion (=Konstante) mit $Z() = 0$
- $S : \mathbb{N}^1 \to \mathbb{N}$, die einstellige Nachfolgerfunktion mit $S(n) = n + 1$
- $pr^k_j : \mathbb{N}^k \to \mathbb{N}$, die k-stellige Projektion auf die j-te Komponente, also $pr^k_j(x_1, ..., x_k) = x_j$, definiert für $k \geq 1$ und $1 \leq j \leq k$.

Jetzt führen wir zwei Erzeugungsschemata ein, mittels derer man aus gegebenen Zahlenfunktionen neue Zahlenfunktionen erhält.

- Das Kompositionsschema $Komp : (g_1, ..., g_m, h) \mapsto f$ erzeugt aus m k-stelligen Funktionen g_1,...,g_m und einer m-stelligen Funktion h eine k-stellige Funktion f mit

  $$f(x_1, ..., x_k) := h(g_1(x_1, ..., x_k), ..., g_m(x_1, ..., x_k))$$

  Dabei ist $m \geq 1$ und $k \geq 0$.

- Das Rekursionsschema $PrRek : (g, h) \mapsto f$ der primitiven Rekursion erzeugt aus einer k-stelligen Verankerungsfunktion g und einer $(k+2)$-stelligen Funktion h eine neue (rekursiv definierte) $(k+1)$-stellige Funktion f mit

  $$\begin{aligned} f(x_1, ..., x_k, 0) &:= g(x_1, ..., x_k) \\ f(x_1, ..., x_k, y+1) &:= h(x_1, ..., x_k, y, f(x_1, ..., x_k, y)) \end{aligned}$$

  Dabei ist $k \geq 0$.

Damit können wir die Funktionsklasse der primitiv rekursiven Funktionen definieren:

Definition 4.1. *Die Menge der primitiv rekursiven Funktionen besteht genau aus den Funktionen, die sich aus der Menge der Grundfunktionen durch endlich oft wiederholte Anwendung des Kompositionsschemas und des Rekursionsschemas bilden lassen.*

Bemerkungen:

- 'Endlich oft wiederholbar' umfasst auch nullfache Wiederholung. Die Grundfunktionen selbst sind also auch primitiv rekursiv.
- Man kann die neue Klasse auch als kleinste Funktionenklasse definieren, die die Menge der Grundfunktionen enthält und unter den Anwendungen der beiden Schemata abgeschlossen ist.

Formaler kann man die Klasse auch folgendermaßen definieren:

1. Alle Grundfunktionen sind primitiv rekursiv.
2. Sind m k-stellige Funktionen g_1,...,g_m primitiv rekursiv und eine m-stellige Funktion h ebenfalls primitiv rekursiv, so ist auch die im Kompositionsschema definierte k-stellige Funktion $f = Komp(g_1, ..., g_m, h)$ primitiv rekursiv.
3. Sind eine k-stellige Funktion g und eine $(k+2)$-stellige Funktion h primitiv rekursiv, so ist auch die im Rekursionsschema definierte $(k+1)$-stellige Funktion $f = PrRek(g, h)$ primitiv rekursiv.
4. Weitere primitiv rekursive Funktionen gibt es nicht.

Wir ersparen es uns, formal nachzuweisen, dass durch das Rekursionsschema $PrRek$ wirklich eine Funktion definiert wird.

Da die Grundfunktionen alle total sind und das Kompositionsschema und das Rekursionsschema aus totalen Funktionen wieder totale Funktionen erzeugen, sind alle primitiv rekursiven Funktionen total.

Man kann primitive Rekursivität auch für charakteristische Funktionen und Prädikate definieren:

Eine Teilmenge T der natürlichen Zahlen kann durch ihre charakteristische Funktion $ch_T : \mathbb{N} \to \mathbb{N}$ charakterisiert werden:

$$ch_T(x) = \begin{cases} 1 \text{ für } x \in T \\ 0 \text{ sonst} \end{cases}$$

Dieses Konzept kann auf Teilmengen aus $\mathbb{N}^k$ ausgedehnt werden. Insofern kann man von primitiv rekursiven Mengen reden, wenn die zugehörigen charakteristischen Funktionen primitiv rekursiv sind.

Oftmals werden Mengen auch durch Prädikate definiert: Ein Prädikat P ist eine Funktion $P : \mathbb{N} \to \{\text{true}, \text{false}\}$.

Dann ist $\{x \mid P(x) = \text{true}\}$ eine Menge. Damit kann der Begriff 'primitiv rekurisv' auch auf Prädikate ausgedehnt werden:

P heißt primitiv rekursiv, wenn die folgende Funktion f_P primitiv rekusiv ist:

$$f_P(x) = \begin{cases} 1 \text{ falls } P(x) = \text{ true} \\ 0 \text{ falls } P(x) = \text{ false} \end{cases}$$

Mit Hilfe der *charakteristischen* Funktionen kann man auch einfach die Negation eines Prädikates, die Disjunktion und die Konjunktion ausdrücken, die in der Mengenlehre über die Vereinigung und den Durchschnitt analog definiert werden. Daher überträgt sich die primitive Rekursivität auch auf diese abgeleiteten Prädikate.

Beispiel 4.2. *(a) Die nullstellige Funktion Z mit Wert 0 ist eine Grundfunktion.*

Die nullstellige Funktion $c_1^{(0)}$ mit Wert $c_1^{(0)}() = 1$ erhält man durch Komposition der Nachfolgerfunktion und der nullstelligen Funktion mit Wert 0: es gilt $S(0) = 1$, d.h. $c_1^{(0)} = Komp(Z, S)$. Also die Funktion $c_1^{(0)}$ auch primitiv rekursiv.

Durch Induktion zeigt man ebenso, dass für jede natürliche Zahl m die nullstellige Funktion $c_m^{(0)}$ mit Wert m primitiv rekursiv ist.

(b) Sei m irgendeine natürliche Zahl. Mit der nullstelligen konstanten Funktion $g = c_m^{(0)}$ mit Wert m und $h = pr_2^2$ erhält man mit dem Rekursionsschema daraus die einstellige konstante Funktion $c_m^{(1)}$ mit Wert m: Bei $c_m^{(1)} := PrRek(g, h)$ ist stets $c_m^{(1)}(x) = m$

(c) *Sei m irgendeine natürliche Zahl und $k \geq 1$. Aus der einstelligen konstanten Funktion $c_m^{(1)}$ mit Wert m und der Funktion pr_1^k erhält man durch Substitution die k-stellige konstante Funktion mit Wert m.*

(d) *Die identische Funktion $id : \mathbb{N} \to \mathbb{N}$ ist primitiv rekursiv, denn sie ist nichts anderes als die Grundfunktion pr_1^1.*

(e) *Die Addition kann rekursiv durch $add(x,0) = x$ und $add(x,y+1) = S(add(x,y))$ definiert werden.*

Formal erhält man sie mit dem Rekursionsschema aus der einstelligen primitiv rekursiven Funktion $g = id$ und der 3-stelligen primitiv rekursiven Funktion $h = S(pr_3^3)$.

Also ist die Additionsfunktion primitiv rekursiv.

(f) *Ähnlich sieht man, dass auch die Multiplikationsfunktion primitiv rekursiv ist. Es gilt nämlich $mult(x,0) = 0$ und $mult(x,y+1) = add(x, mult(x,y))$.*

Also erhält man $mult$ durch Anwendung des Rekursionsschemas auf

- *g = die einstellige konstante Funktion mit Wert 0*
- *$h = add(pr_1^3, pr_3^3)$, eine 3-stellige Funktion, die man wiederum durch Komposition aus den Grundfunktionen pr_1^3,pr_3^3 und der schon als primitiv rekursiv erkannten Funktion add erhält.*

(g) *Die Vorgängerfunktion $V : \mathbb{N} \to \mathbb{N}$ ist durch $V(0) := 0$ und $V(n+1) := n$ definiert. Also erhält man sie mit dem Rekursionsschema aus $g = Z$ und $h = pr_1^2$.*

(h) *Die modifizierte Subtraktion $sub : \mathbb{N} \to \mathbb{N}$ mit*

$$sub(x,y) = \begin{cases} x - y, & \text{falls } x \geq y \\ 0, & \text{falls } x < y \end{cases}$$

erfüllt die Gleichungen $sub(x,0) = x$ und $sub(x,y+1) = V(sub(x,y))$. Also erhält man sie mit dem Rekursionsschema aus den primitiv rekursiven Funktionen id und $V(pr_3^3)$.

(i) *Die Vorzeichenfunktion $sg : \mathbb{N} \to \mathbb{N}$ ist primitiv rekursiv, denn $sg(0) = 0$ und $sg(y+1) = 1$ für alle y.*

Beispiel 4.3. *Wir werden etwas später ein weiteres Erzeugungsschema definieren, das zu den μ-rekursiven Funktionen führen wird. Man kann es als Realisierung einer unbeschränkten Nullstellensuche bezeichnen. Hier wollen wir zeigen, dass man eine beschränkte Nullstellensuche noch mit den ersten beiden Erzeugungsschemata realisieren kann.*

Sei $f : \mathbb{N}^{k+1} \to \mathbb{N}$ eine totale Funktion.

Wir definieren eine neue totale Funktion $g : \mathbb{N}^{k+1} \to \mathbb{N}$ durch

$$g(x_1,...,x_k,x_{k+1}) = \min(\{x_{k+1}\} \cup \{n \in \mathbb{N} \mid n < x_{k+1} \text{ und } f(n,x_1,...,x_k) = 0\} \ )$$

Dann gilt

$$\begin{aligned} g(x_1,...,x_k,0) &= 0 \\ g(x_1,...,x_k,y+1) &= g(x_1,...,x_k,y) \\ &\quad + sg(f(g(x_1,...,x_k,y),x_1,...,x_k)) \end{aligned}$$

Daher gilt: wenn f primitiv rekursiv ist, ist auch g primitiv rekursiv.

Beispiel zur Anwendung der beschränkten Minimalisierung:

- *Sei* $\boldsymbol{f : \mathbb{N}^2 \to \mathbb{N}}$ *definiert durch*

$$\boldsymbol{f(n,x) := sub(x, mult(n,n))} \quad = x - n^2$$

- *Sei* $\boldsymbol{g}$ *die nach dem eben beschriebenen Verfahren zu* $\boldsymbol{f}$ *definierte primitiv rekursive Funktion, d.h.*

$$\boldsymbol{g(x,y) = \min(\ \{y\} \cup \{n \in \mathbb{N} \mid n < y \text{ und } n^2 \geq x\}\ )}$$

- *Sei* $\boldsymbol{h : \mathbb{N}^2 \to \mathbb{N}}$ *definiert durch* $\boldsymbol{h(x) := g(x, x{+}1)}$.
  Dann ist auch $\boldsymbol{h}$ *primitiv rekursiv, und es gilt:*

$$\boldsymbol{h(x) = \min\{n \in \mathbb{N} \mid n^2 \geq x\}}$$

  Für eine Quadratzahl $\boldsymbol{x}$ *ist* $\boldsymbol{h(x) = \sqrt{x}}$*, generell:*

$$\boldsymbol{h(x) = \lceil \sqrt{x} \rceil}$$

Wir wollen jetzt zeigen, dass man Zahlenvektoren mit Hilfe einer primitiv rekursiven Funktion durch einzelne Zahlen codieren kann. Auch die Umkehrfunktionen dieser Codierung werden primitiv rekursiv sein.

Die einstellige Funktion $\boldsymbol{c_2(n) := \frac{n\cdot(n+1)}{2} = \sum_{i=0}^{n} i}$ gibt gerade die Summe der Zahlen von $\boldsymbol{0}$ bis $\boldsymbol{n}$ wieder. Sie ist primitiv rekursiv, denn es gilt $\boldsymbol{c_2(0) = 0}$ und $\boldsymbol{c_2(n{+}1) = c_2(n) + n + 1 = S(add(n, c_2(n)))}$

Man erhält $\boldsymbol{c_2}$ also mit dem Rekursionsschema aus den Funktionen $\boldsymbol{Z}$ und $\boldsymbol{S(add)}$.

Man kann dies nun für die zweistellige Funktion $\boldsymbol{c(x,y) := c_2(x+y) + x}$ nutzen, die man aus $\boldsymbol{c_2}$ und anderen primitiv rekursiven Funktionen durch Komposition erhält. Also ist diese Funktion selbst primitiv rekursiv. Sie ist außerdem eine Bijektion zwischen $\mathbb{N}^2$ und $\mathbb{N}$.

Die Umkehrung dieser Bijektion erhält man folgendermaßen: Sei $\boldsymbol{z}$ gegeben. Wir wollen $\boldsymbol{x}$ und $\boldsymbol{y}$ mit $\boldsymbol{z = \langle x, y\rangle}$ finden. Man bestimmt dazu $\boldsymbol{n}$ so, dass $\boldsymbol{c_2(n) \leq z < c_2(n{+}1)}$ gilt. Dann ist $\boldsymbol{x = z - c_2(n)}$ und $\boldsymbol{y = n - x}$.

Beispiel: $\boldsymbol{z = \langle x, y\rangle = 18 \Rightarrow n = 5, x = 3, y = 2}$.

Wir wollen nun beweisen, dass auch die Komponenten der Umkehrung primitiv rekursive Funktionen sind.

Dazu setzen wir $\boldsymbol{p_1(\langle x, y\rangle) := x}$, $\boldsymbol{p_2(\langle x, y\rangle) := y}$. Es muss also gelten $\boldsymbol{\langle p_1(n), p_2(n)\rangle = n}$.

Wir wollen zeigen, dass die so definierten Funktionen $\boldsymbol{p_1}$ und $\boldsymbol{p_2}$ primitiv rekursiv sind.

Dazu folgen wir der eben beschriebenen Methode, um aus $\boldsymbol{z = \langle x, y\rangle}$ die Zahlen $\boldsymbol{x = p_1(z)}$ und $\boldsymbol{y = p_2(z)}$ zu bestimmen.

Zuerst berechnet man die kleinste Zahl $\boldsymbol{n}$, so dass $\boldsymbol{z < c_2(n{+}1)}$ gilt. Dies geht mit einer beschränkten Nullstellensuche wie in Beispiel 4.4.3: Die durch $\boldsymbol{f(n,z) := sub(1, sub(c_2(n{+}1), z))}$ definierte zweistellige Funktion $\boldsymbol{f}$ ist primitiv rekursiv. Nach Beispiel 4.4.3 ist dann auch die durch

$$\boldsymbol{g(z,t) = \min(\{t\} \cup \{n \in \mathbb{N} \mid n < t \text{ und } f(n,z) = 0\})}$$

definierte zweistellige Funktion $\boldsymbol{g}$ primitiv rekursiv.

Da klar ist, dass die gesuchte Zahl $\boldsymbol{n}$ nicht größer als $\boldsymbol{z}$ sein kann, braucht man nur bis $\boldsymbol{z}$ zu suchen. Also ist die gewünschte Zahl $\boldsymbol{n}$ gegeben durch $\boldsymbol{n = g(z,z)}$.

Wir erhalten $\boldsymbol{x = p_1(z) = z - c_2(g(z,z))}$ und $\boldsymbol{y = p_2(z) = n - x = g(z,z) - (z - c_2(g(z,z)))}$. Also ist sowohl $\boldsymbol{p_1}$ als auch $\boldsymbol{p_2}$ primitiv rekursiv.

Man kann die Cantorsche Bijektion auch verwenden, um ein $\boldsymbol{k}$-Tupel von natürlichen Zahlen (mit $\boldsymbol{k \geq 1}$) in eine einzige nicht negative Zahl umzukodieren. Dazu setzt man:

$$\boldsymbol{\langle n_1, n_2, ..., n_k\rangle := \langle n_1, \langle n_2, ..., \langle n_{k-1}, n_k\rangle ...\rangle\rangle}$$

Die so definierte Funktion ist also eine Bijektion von $\mathbb{N}^k$ nach $\mathbb{N}$.

Sie ist nach dem Kompositionsschema primitiv rekursiv. Durch Komposition der Funktionen p_1 und p_2 kann man auch die Umkehrfunktionen $d_i^{(k)}$ zu dieser k-stelligen Codierfunktion erhalten:

$$\begin{aligned} d_1^{(k)}(n) &= p_1(n) \\ d_2^{(k)}(n) &= p_1(p_2(n)) \\ &\ldots \\ d_{k-1}^{(k)}(n) &= p_1(p_2(p_2(\ldots p_2(n)\ldots) \\ d_k^{(k)}(n) &= p_2(p_2(\ldots p_2(n)\ldots)) \end{aligned}$$

wobei p_2 bei $d_{k-1}^{(k)}(n)$ $(k-1)$-mal angewendet wird und bei $d_k^{(k)}(n)$ k-mal. Auch diese Umkehrfunktionen sind primitiv rekursiv.

Nun können wir den folgenden Satz beweisen:

Satz 4.4. *Eine Funktion ist primitiv rekursiv genau dann, wenn sie LOOP-berechenbar ist.*

Beweis :

Zuerst zeigen wir durch Induktion über den Aufbau der primitiv rekursiven Funktionen, dass jede primitiv rekursive Funktion LOOP-berechenbar ist. Die Grundfunktionen sind LOOP-berechenbar. Zum Beispiel wird die nullstellige Funktion Z mit konstantem Wert 0 durch das LOOP-Programm $x_0 := x_0 + 0$ berechnet. Die Nachfolgerfunktion S wird durch das LOOP-Programm $x_0 := x_1 + 1$ berechnet. Und die Projektionsfunktion pr_j^k wird durch das LOOP-Programm $x_0 := x_j + 0$ berechnet. Sei jetzt f nach dem Kompositionsschema definiert, d.h. $f = h(g_1, ..., g_m)$, und $g_1, ..., g_m$ seien k-stellige Funktionen. Nach Induktionssannahme gibt es LOOP-Programme, die die Funktionen h, g_1,..., g_m berechnen. Es ist klar, dass man sie zu einem LOOP-Programm für f zusammensetzen kann. Man muss dazu nur einige Variablen auswechseln.

Kommen wir zum Rekursionsschema:

$$\begin{aligned} f(x_1, ..., x_k, 0) &:= g(x_1, ..., x_k) \\ f(x_1, ..., x_k, y{+}1) &:= h(x_1, ..., x_k, y, f(x_1, ..., x_k, y)) \end{aligned}$$

Wir können wieder annehmen, dass LOOP-Programme zur Berechnung von g und h zur Verfügung stehen und können damit ein LOOP-Programm wie folgt konstruieren:

$x_0 := g(x_1, ..., x_k)$;
$t := 0$;
LOOP x_{k+1} DO $x_0 := h(x_1, ..., x_k, t, x_0); t := t + 1$ END;

wobei t eine Variable sein soll, die in den LOOP-Programmen für g und h nicht verwendet wird. Außerdem müssen die LOOP-Programme für g und h so abgewandelt werden, dass sie keine gemeinsamen Arbeitsvariablen benutzen. Die Variablen x_1,..., x_{k+1} sollten ferner nicht überschrieben werden, da sie die Eingabe enthalten und diese in den Schleifendurchläufen immer wieder gebraucht wird. Dieses LOOP-Programm berechnet offenbar $f(x_1, ..., x_k, x_{k+1})$.

Sei nun umgekehrt $f : \mathbb{N}^r \to \mathbb{N}$ LOOP-berechenbar. Dann gibt es ein LOOP-Programm P, das f berechnet. Die in P vorkommenden Variablen seien x_0,x_1,..., x_k mit $k \geq r$. Nun wird durch Induktion über die Länge von P gezeigt, dass es eine primitiv rekursive Funktion $g_P : \mathbb{N} \to \mathbb{N}$ gibt, die das Verhalten des LOOP-Programms P auf allen Variablen x_0, x_1,..., x_k simuliert. Dies ist in dem Sinne zu verstehen, dass zu gegebenen Anfangswerten a_0, a_1,..., a_k und Endwerten b_0, b_1,..., b_k der Variablen x_0, x_1,..., x_k gilt, dass

$$g_P(<a_0, a_1, ..., a_k>) = <b_0, b_1, ..., b_k>$$

Falls P die Form $x_i := x_j \pm c$ hat, so setzt man

$$g_P(z) := <d_1^{(k+1)}(z), ..., d_i^{(k+1)}(z), d_{j+1}^{(k+1)}(z) \pm c, d_{i+2}^{(k+1)}(z), ..., d_{k+1}^{(k+1)}(z)>$$

Hier benutzen wir, dass die oben definierte Bijektion $< \cdot >$ von $\mathbb{N}^{k+1}$ nach $\mathbb{N}$ und ihre Umkehrfunktionen $d_1^{(k+1)}$, $d_2^{(k+1)}$, ..., $d_{k+1}^{(k+1)}$ primitiv rekursiv sind. Man beachte die Indexverschiebung: Variable x_i entspricht der $i+1$-ten Komponenten des Zahlenvektors!

Hat P die Form $Q; R$, so sind die beiden Programme Q und R kürzer als P. Daher existieren nach Induktionsannahme primitiv rekursive Funktionen g_Q für Q und g_R für R. Wir definieren g_P einfach durch $g_P(z) = g_R(g_Q(z))$. Dann ist g_P auch primitiv rekursiv und hat die gewünschte Eigenschaft.

Kommen wir zu der Form `LOOP` x_i `DO` Q `END`, so definieren wir zunächst durch die primitive Rekursion mittels g_Q eine zweistellige Funktion h:

$$h(0, x) = x; \quad h(n+1, x) = g_Q(h(n, x))$$

und können so den Zustand der Arbeitsvariablen $x = <x_0, x_1, ..., x_k>$ nach n Anwendungen von Q beschreiben. Wir können setzen $g_P(x) := h(d_{i+1}^{(k+1)}(x), x)$.

Damit ist der Induktionsbeweis beendet. Wir haben gezeigt, dass es zu dem LOOP-Programm P eine primitiv rekursive Funktion g_P mit $g_P(<a_0, a_1, ..., a_k>) = <b_0, b_1, ..., b_k>$ gibt. Da $f(n_1, ..., n_r)$ der Wert der Programmvariablen x_0 nach Ausführung von P ist, gilt $f(n_1, ..., n_r) = d_1^{(k+1)}(g_P(<0, n_1, ..., n_r, 0, ..., 0>))$, wobei hier bei der Eingabe die letzten $(k - r)$ Stellen 0 sind. Also ist f primitiv rekursiv.

Eine echte Erweiterung der Klasse der primitiv rekursiven Funktionen wird durch die Hinzunahme eines neuen Erzeugungsschemas, des μ-Operators erreicht. Es sei f eine gegebene $(k+1)$-stellige Funktion. Dann definieren wir

$$\begin{aligned} g(x_1, ..., x_k) \quad = \quad & \min\{n \in \mathbb{N} \mid f(n, x_1, ..., x_k) = 0 \text{ und} \\ & \text{für alle } m < n \text{ ist } f(m, x_1, ..., x_k) \text{ definiert und} \\ & \text{für alle } m < n \text{ ist } f(m, x_1, ..., x_k) > 0\} \end{aligned}$$

oder kürzer

$$g(x_1, ..., x_k) = (\mu n)[f(n, x_1, ..., x_k) = 0]$$

Für g schreibt man auch μf. Die Funktion g ist k-stellig und partiell über $\mathbb{N}^k$ definiert, denn wenn das Minimum nicht existiert, sei g an dieser Stelle undefiniert. f braucht auch nur partiell definiert zu sein. Wenn f z.B. die $(k+1)$-stellige konstante Funktion mit Wert z.B. 42 ist und damit überall definiert ist, ist g die k-stellige nirgends definierte Funktion. Die Funktion g erhält man durch unbeschränkte Nullstellensuche für die Funktion f. Man beachte, dass man eine beschränkte Nullstellensuche noch durch das Rekursionsschema der primitiven Rekursion und das Kompositionsschema realisieren kann, vgl. Beispiel 4.4.3.

Definition 4.5. *Die Menge der μ-rekursiven (auch: partiell rekursiven) Funktionen besteht genau aus den Funktionen, die sich aus der Menge der Grundfunktionen durch endlich oft wiederholte Anwendung des Kompositionsschemas, des Rekursionsschemas und des μ-Operators bilden lassen.*

Satz 4.6. *Die Klasse der μ-rekursiven Funktionen stimmt mit der Klasse der WHILE- (TURING-) berechenbaren Funktionen überein.*

Beweis:

Wir können im Wesentlichen den Beweis des letzten Satzes übernehmen, müssen allerdings noch den Fall des μ-Operators im Zusammenhang mit der WHILE-Schleife betrachten.

Sei eine Funktion g durch Anwendung des μ-Operators auf eine μ-rekursive Funktion f definiert, also

$$g(x_1, ..., x_k) = (\mu n)[f(n, x_1, ..., x_k) = 0]$$

Nach Induktionsannahme gibt es ein WHILE-Programm zur Berechnung von f. Dieses wird in dem folgenden WHILE-Programm unter geeigneter Änderung -falls nötig- der Variablen verwendet. Das folgende WHILE- Programm berechnet die Funktion g:

$x_0 := 0;$
$y := f(0, x_1, ..., x_k);$
WHILE y DO
  $x_0 := S(x_0);$
  $y := f(x_0, x_1, ..., x_k);$
END;

Sei umgekehrt $f : \mathbb{N}^r \to \mathbb{N}$ WHILE-berechenbar. Dann gibt es ein WHILE-Programm P, das f berechnet. Die in P vorkommenden Variablen seien x_0, x_1,..., x_k mit $k \geq r$. Nun wird durch Induktion über die Länge von P gezeigt, dass es eine μ-rekursive Funktion $g_P : \mathbb{N} \to \mathbb{N}$ gibt derart, dass zu gegebenen Anfangswerten a_0, a_1,..., a_k und Endwerten b_0, b_1,..., b_k der Variablen x_0, x_1,..., x_k gilt, dass

$$g_P(<a_0, a_1, ..., a_k>) = <b_0, b_1, ..., b_k>$$

Nach den Überlegungen im vorigen Beweis müssen wir nur noch den Fall behandeln, dass das Programm P ein WHILE-Programm der Form WHILE x_i DO Q END ist. Wie im vorigen Beweis können wir wieder eine zweistellige Funktion $h(n, z)$ definieren, die den Zustand der Programmvariablen $z = <x_0, x_1, ..., x_k>$ nach n Anwendungen von Q beschreibt und die in diesem Fall μ-rekursiv ist. Dann setzen wir

$$g_P(z) = h((\mu n)[d_1^{(k+1)}(h(n, z)) = 0], z)$$

Das erste Argument liefert gerade die minimale Wiederholungszahl des Programmes Q.

Wir können noch den folgenden Satz ableiten:

Satz 4.7 (Kleene)**.** *Für jede k-stellige μ-rekursive Funktion f gibt es zwei $(k+1)$-stellige primitiv rekursive Funktionen p und q, so dass sich f darstellen lässt als*

$$f(x_1, ..., x_k) := p(x_1, ..., x_k, \mu q(x_0, x_1, ..., x_k))$$

Hierbei ist μq durch die Anwendung des μ-Operators auf q entstanden und steht abkürzend für

$$(\mu x_0)[q(x_0, x_1, ..., x_n) = 0]$$

Beweis:

Jede μ-rekursive Funktion kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden. Bei der Transformation eines derartigen WHILE-Programms in eine Darstellung als μ-rekursive Funktion, die mittels Grundfunktionen und Erzeugungsschemata erzeugt wird, erhält man die gewünschte Form, wie die Beweise von Satz 4.4.4 und Satz 4.4.6 zeigen.

5 Der λ-Kalkül

Noch leer

6 Eine totale WHILE-, aber nicht LOOP-berechenbare Funktion

Wir haben in den letzten Kapiteln gesehen, dass für Zahlenfunktionen die Begriffe der Turing-Berechenbarkeit, der WHILE-Berechenbarkeit, der GOTO-Berechenbarkeit und der μ-Rekursivität alle äquivalent sind. Außerdem sind auch die beiden Begriffe der LOOP-Berechenbarkeit und der primitiven Rekursivität äquivalent. Ferner ist jede LOOP-berechenbare Funktion WHILE-berechenbar. Der Begriff der WHILE-Berechenbarkeit ist natürlich allgemeiner als der der LOOP-Berechenbarkeit, da es WHILE-berechenbare Funktionen gibt, die nicht für alle Eingaben definiert sind, während alle LOOP-berechenbaren Funktionen total sind. In diesem Abschnitt wollen wir zeigen, dass auch für totale Funktionen der Begriff der WHILE-Berechenbarkeit ein umfassenderer Begriff als der der LOOP-Berechenbarkeit ist. Das heißt, wir wollen eine totale Zahlenfunktion konstruieren, die WHILE-berechenbar, aber nicht LOOP-berechenbar ist.

Eine berühmte, sehr einfach zu definierende totale Funktion, die WHILE-berechenbar, aber nicht LOOP-berechenbar ist, ist 1928 von Ackermann angegeben worden. Sie trägt heute den Namen Ackermannfunktion. Allerdings erfordert der Nachweis, dass sie nicht LOOP-berechenbar ist, eine Reihe von Induktionsbeweisen und Abschätzungen. Eine übersichtliche Darstellung findet man in dem Buch 'Theoretische Informatik- kurzgefasst' von Schöning.

Wir werden in diesem Kapitel eine andere Funktion konstruieren. Die Konstruktion basiert auf einem in der Berechenbarkeitstheorie und Komplexitätstheorie wichtigen Konstruktionsprinzip, der Diagonalisierung.

Zuerst möchten wir alle LOOP-Programme durch endliche Wörter über einem endlichen Alphabet codieren, zum Beispiel über dem folgenden Alphabet

$$\boldsymbol{A} = \{\ + - \ := ;\ \texttt{LOOP DO END}\ \boldsymbol{x}\ \mathbf{0}\ \mathbf{1}\ \}$$

mit 10 Symbolen (hier betrachten wir z.B. `LOOP` als ein Symbol). Dazu können wir Variablen $\boldsymbol{x_i}$ durch '$\boldsymbol{x\ bin(i)}$' codieren und Konstanten $\boldsymbol{c}$ durch $\boldsymbol{bin(c)}$. Alle anderen syntaktischen Komponenten von LOOP-Programmen können wir einfach übernehmen. Es ist klar, dass dadurch jedes LOOP-Programm durch ein endliches Wort über dem Alphabet $\boldsymbol{A}$ codiert wird. Fortan werden wir diese endlichen Wörter selbst als LOOP-Programme bezeichnen.

Nun können wir all diese LOOP-Programme (d.h. alle endlichen Wörter über dem Alphabet $\boldsymbol{A}$, die LOOP-Programme codieren) nach ihrer Länge sortieren und Wörter gleicher Länge alphabetisch sortieren (oder erst alle Wörter über dem Alphabet $\boldsymbol{A}$ nach der Länge sortieren und Wörter gleicher Länge alphabetisch sortieren und aus der entstehenden Liste dann alle Wörter herausstreichen, die kein LOOP-Programm sind). Dazu müssen wir nur vorab irgendeine Ordnung auf dem Alphabet $\boldsymbol{A}$ festlegen. Wir bekommen dadurch eine Liste aller LOOP-Programme:

$$\boldsymbol{P_0, P_1, P_2, P_3, \ldots}$$

Zum Beweis merken wir an, dass die obige Definition der Liste $\boldsymbol{P_0, P_1, P_2, P_3, \ldots}$ aller LOOP-Programme sicher in dem Sinne effektiv ist, dass man aus dem Index $\boldsymbol{i}$ das Wort $\boldsymbol{P_i}$ berechnen kann, z.B. mit einer Turingmaschine. Um $\boldsymbol{g(i,n)}$ zu berechnen, muss man also bei Eingabe von $\boldsymbol{i}$ und $\boldsymbol{n}$ erst das LOOP-Programm $\boldsymbol{P_i}$ berechnen, dann $\boldsymbol{P_i}$ mit der Eingabe $\boldsymbol{(0, n, 0, 0, \ldots)}$ in den Variablen $\boldsymbol{x_0, x_1, x_2, x_3, \ldots}$ simulieren und den Ausgabewert am Ende der Simulation aus der Variablen $\boldsymbol{x_0}$ herauslesen. Es ist intuitiv einleuchtend, dass man einen entsprechenden Algorithmus entwerfen kann. Dann wird man es auch mit einer Turingmaschine hinbekommen. Also ist die Funktion $\boldsymbol{g}$ Turing-berechenbar und daher auch WHILE-berechenbar.

Wir halten außerdem fest, dass $\boldsymbol{g}$ eine totale Funktion ist, d.h. dass $\boldsymbol{g(i,n)}$ für alle $\boldsymbol{i}$ und $\boldsymbol{n}$ definiert ist, da das LOOP-Programm $\boldsymbol{P_i}$ bei Eingabe von $\boldsymbol{n}$ nach endlich vielen Schritten anhält.

Wir behaupten nun, dass die Funktion $\boldsymbol{g}$ nicht LOOP-berechenbar ist.

Satz 6.2. *Die Funktion $\boldsymbol{g}$ aus Lemma 4.5.1 ist nicht LOOP-berechenbar.*

Außerdem gibt es eine totale Funktion $\boldsymbol{f : \mathbb{N} \to \mathbb{N}}$, die WHILE-berechenbar, aber nicht LOOP-berechenbar ist.

Eine gewünschte Funktion $\boldsymbol{f}$ erhalten wir aus der Funktion $\boldsymbol{g}$ durch eine Diagonalkonstruktion. Wir definieren

$$\boldsymbol{f(n) := g(n,n) + 1}$$

für alle natürlichen Zahlen n. Da g total ist, ist auch f total. Da g WHILE-berechenbar ist, ist auch f WHILE-berechenbar. Wäre g LOOP-berechenbar, so wäre auch f LOOP-berechenbar. Wir werden aber nun sehen, dass f nicht LOOP-berechenbar ist.

Wir führen einen Widerspruchsbeweis und nehmen an, f sei doch LOOP-berechenbar. Dann gibt es ein LOOP-Programm P_j, das die Funktion f berechnet. Also ist $f(n) = g(j, n)$ für alle natürlichen Zahlen n, insbesondere für $n = j$. Es gilt also $f(j) = g(j, j)$. Andererseits gilt nach Definition von f auch $f(j) = g(j, j) + 1$. Da die Zahlen $g(j, j)$ und $g(j, j) + 1$ niemals gleich sein können (man beachte, dass $g(j, j)$ definiert ist, da g total ist), haben wir einen Widerspruch hergeleitet. Also ist f tatsächlich nicht LOOP-berechenbar, und daher auch g nicht.

Diese Beweismethode nennt man Diagonalisierung. Der Begriff Diagonalisierung wird verständlich, wenn man sich die LOOP-Programme $P_0, P_1, P_2, P_3, \ldots$ untereinander schreibt und die Werte der von ihnen berechneten einstelligen Funktionen in den Zeilen daneben:

	0	1	2	3	$\cdots$
P_0	$g(0,0)$	$g(0,1)$	$g(0,2)$	$g(0,3)$	
P_1	$g(1,0)$	$g(1,1)$	$g(1,2)$	$g(1,3)$	
P_2	$g(2,0)$	$g(2,1)$	$g(2,2)$	$g(2,3)$	
P_3	$g(3,0)$	$g(3,1)$	$g(3,2)$	$g(3,3)$	
$\vdots$					

Die Funktion f wird gerade so definiert, dass sie sich in der Diagonale von all diesen Funktionen unterscheidet. Von der i-ten LOOP-berechenbaren einstelligen Funktion, nämlich von der von P_i berechneten einstelligen Funktion, unterscheidet sie sich gerade bei dem Eingabewert i. Das gilt für alle i.

7 Standardnotationen für berechenbare Funktionen

In den letzten Kapiteln haben wir gesehen, dass es einen mathematisch präzise definierten Berechenbarkeitsbegriff für Zahlenfunktionen gibt, der der Churchschen These zufolge gerade die intuitiv berechenbaren Zahlenfunktionen beschreibt. Ein Indiz für die Natürlichkeit dieses Berechenbarkeitsbegriffs ist die Tatsache, dass ganz verschiedene Ansätze alle zu dem gleichen Berechenbarkeitsbegriff geführt haben. In diesem Abschnitt halten wir zuerst fest, dass der über Turingmaschinen eingeführte Berechenbarkeitsbegriff für Wortfunktionen gleichermaßen natürlich ist und in gewisser Weise äquivalent zu dem für Zahlenfunktionen ist. Anschließend beschäftigen wir uns mit der Frage, welche Eigenschaften eine Programmiersprache für berechenbare Funktionen haben sollte.

Beispiel 7.1. *Die berechenbaren Wortfunktionen haben wir über Turingmaschinen definiert. Da wir die berechenbaren Zahlenfunktionen über Turingmaschinen definiert haben, ist es naheliegend, eine These ähnlich der Churchschen These nun auch für die berechenbaren Wortfunktionen aufzustellen: die durch Turingmaschinen berechenbaren Wortfunktionen sind genau die intuitiv berechenbaren Wortfunktionen. Wir wollen dies auch noch dadurch belegen, dass wir eine natürliche Bijektion zwischen berechenbaren Wortfunktionen und berechenbaren Zahlenfunktionen herstellen.*

Sei $\boldsymbol{E} = \{e_1, e_2, \ldots, e_n\}$ ein endliches Alphabet. Wir definieren eine Bijektion $\boldsymbol{\nu} : \mathbb{N} \to \boldsymbol{E}^*$, indem wir die Wörter in $\boldsymbol{E}^*$ der Länge nach und Wörter gleicher Länge alphabetisch sortieren:

i	0	1	2	$\ldots$	n	$n{+}1$	$n{+}2$	$\ldots$	$n^2{+}n$	$n^2{+}n{+}1$	$\ldots$
$\nu(i)$	ε	e_1	e_2	$\ldots$	e_n	e_1e_1	e_1e_2	$\ldots$	e_ne_n	$e_1e_1e_1$	$\ldots$

Es ist klar, dass diese Abbildung $\boldsymbol{\nu} : \mathbb{N} \to \boldsymbol{E}^*$ eine Bijektion ist. Daher ist auch ihre Umkehrabbildung $\boldsymbol{\nu^{-1}} : \boldsymbol{E}^* \to \mathbb{N}$ definiert. Ist nun $\boldsymbol{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ eine einstellige Zahlenfunktion, so wird durch $\boldsymbol{\nu \circ f \circ \nu^{-1}}$ eine Wortfunktion definiert, die man erhält, indem man auf das Eingabewort $\boldsymbol{w}$ erst die Abbildung $\boldsymbol{\nu^{-1}}$ anwendet, dann auf die erhaltene Zahl $\boldsymbol{\nu^{-1}(w)}$ die Zahlenfunktion $\boldsymbol{f}$, und dann auf die erhaltene Zahl $\boldsymbol{f\nu^{-1}(w)}$ wiederum die Bijektion $\boldsymbol{\nu}$, die die Zahl wieder in das korrespondierende Wort umwandelt. Ist andererseits $\boldsymbol{g} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^*$ eine Wortfunktion, so wird durch $\boldsymbol{\nu^{-1} \circ g \circ \nu}$ eine einstellige Zahlenfunktion definiert. Es ist klar, dass diese beiden Zuordnungen genau invers zueinander sind. Also definieren sie eine Bijektion zwischen der Menge aller (möglicherweise partiellen) Zahlenfunktionen $\boldsymbol{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ und der Menge aller (möglicherweise partiellen) Wortfunktionen $\boldsymbol{g} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^*$.

Satz 7.2.

1. *Ist $\boldsymbol{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ eine (Turing-)berechenbare Zahlenfunktion, so ist $\boldsymbol{\nu \circ f \circ \nu^{-1}}$ eine (Turing-)berechenbare Wortfunktion.*
2. *Ist $\boldsymbol{g} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^*$ eine (Turing-)berechenbare Wortfunktion, so ist $\boldsymbol{\nu^{-1} \circ g \circ \nu}$ eine (Turing-)berechenbare Zahlenfunktion.*

Wir beweisen (1). Der Beweis von (2) geht analog. Sei $\boldsymbol{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ eine berechenbare Zahlenfunktion. Dann gibt es eine Turingmaschine $\boldsymbol{TM}$, die aus der Binärdarstellung $\boldsymbol{bin(n)}$ einer Zahl $\boldsymbol{n}$ im Definitionsbereich von $\boldsymbol{f}$ die Binärdarstellung $\boldsymbol{bin(f(n))}$ von $\boldsymbol{f(n)}$ berechnet. Wir müssen eine Turingmaschine konstruieren, die aus einem Wort $\boldsymbol{w}$ im Definitionsbereich von $\boldsymbol{\nu \circ f \circ \nu^{-1}}$ das Wort $\boldsymbol{\nu \circ f \circ \nu^{-1}(w)}$ berechnet. Eine solche Turingmaschine kann folgendermaßen vorgehen: zuerst bestimmt sie aus dem Wort $\boldsymbol{w}$ die Binärdarstellung $\boldsymbol{bin(\nu^{-1}(w))}$ der Zahl $\boldsymbol{\nu^{-1}(w)}$. Darauf wendet sie die Turingmaschine $\boldsymbol{TM}$ an. Das Ergebnis, das natürlich die Binärdarstellung der Zahl $\boldsymbol{f(\nu^{-1}(w))}$ ist, wird schließlich wieder codiert in das Wort $\boldsymbol{\nu(f(\nu^{-1}(w)))}$. Das Ganze geht mit Turingmaschinen, da die Funktion, die Wörter $\boldsymbol{w}$ auf $\boldsymbol{bin(\nu^{-1}(w))}$ abbildet, sicher berechenbar ist, ebenso ihre Umkehrung.

Der Satz zeigt, dass die Berechenbarkeitsbegriffe für Zahlen und für Funktionen im Wesentlichen gleichwertig sind. Wir werden in der Zukunft manchmal einfach von berechenbaren Funktionen sprechen. Zum Teil werden wir auch als Eingabe Wörter und als Ausgabe Zahlen zulassen oder umgekehrt. Diesen Fall kann man mit Hilfe der Bijektion $\boldsymbol{\nu}$ auf den Fall der berechenbaren Wortfunktionen zurückführen.

Im zweiten Teil dieses Abschnitts möchten wir uns mit der Frage beschäftigen, ob und wie man eine beliebige berechenbare Funktion durch ein Wort beschreiben kann. Wir wollen nur Funktionen über einem festen Alphabet $\boldsymbol{E}$

betrachten, das mindestens die Symbole $\mathbf{0}$, $\mathbf{1}$ und $\#$ enthalten möge. Es liegt nahe, dies über Turingmaschinen zu tun. Wir sind dabei nur an (deterministischen) Turingmaschinen interessiert, die Wortfunktionen berechnen. Wie immer sei $\delta(s, a)$ undefiniert für alle Endzustände s und alle Symbole a aus dem Arbeitsalphabet, d.h. die Maschine kann bei Erreichen eines Endzustands nicht weiterrechnen.

Zur Erinnerung: Bei einer Turingmaschine M ist die berechnete Wortfunktion f_M wie folgt festgelegt: $f_M(w) =$ undefiniert, falls M bei Eingabe von w nie anhält, und $f_M(w) = y$, falls M bei Eingabe von w nach endlich vielen Schritten in einer Konfiguration us_fv mit einem Endzustand s_f und Wörtern u, v aus dem Arbeitsalphabet anhält; dabei sei y das längste Präfix von v über dem Eingabealphabet.

Lemma 7.3. *Die Menge aller von Turingmaschinen berechneten Funktionen über einem gegebenen Alphabet E ist abzählbar, es gibt zu jedem E eine totale Funktion h mit Definitionsbereich $\{0, 1\}^*$, deren Bild alle(!) berechenbaren Funktionen umfasst.*

Wir werden nun zuerst derartige Turingmaschinen $TM = (S, E, A, \delta, s_0, \square, F)$ durch Wörter beschreiben. Zuerst nummerieren wir die Zustände einer Turingmaschine durch:

$$S = \{s_0, s_1, \ldots, s_n\}$$

Dabei sei s_0 der Startzustand. Wir können außerdem die Nummerierung so wählen, dass die Endzustände am Ende stehen, d.h. es gibt eine Zahl e in $\{1, \ldots, n\}$ derart, dass

$$F = \{s_{n-e+1}, \ldots, s_n\}$$

die Menge der Endzustände ist. Ebenso nummerieren wir die Symbole im Arbeitsalphabet A einer Turingmaschine durch, d.h. wir nehmen an, dass

$$A = \{a_0, a_1, \ldots, a_k\}$$

Dabei sollen die Symbole des Eingabealphabets E, das ja in A enthalten ist, am Anfang stehen, d.h.

$$E = \{a_0, a_1, \ldots, a_l\} \text{ für ein } l \leq k$$

Einen Übergang der Form

$$\delta(s_i, a_j) \text{ enthält } (s_t, a_m, B)$$

kann man nun durch ein Wort der Form

$$\#\#bin(i)\#bin(j)\#bin(t)\#bin(m)\#bin(b)$$

beschreiben. Dabei sei $b = \mathbf{0}$, wenn $B = L$, $b = \mathbf{1}$, wenn $B = N$ und $b = \mathbf{2}$, wenn $B = R$.

Die gesamte Turingmaschine kann man beschreiben, indem man alle derartigen Wörter, die Übergänge der Turingmaschine beschreiben, hintereinander schreibt, und davor noch das Wort

$$bin(n)\#bin(e)\#bin(k)\#bin(l)\#bin(d)\#$$

schreibt, das die Zahl der Zustände, die Zahl der Endzustände, die Zahl der Symbole des Arbeitsalphabets, die Zahl der Symbole des Eingabealphabets und den Index d des Blanksymbols beschreibt.

Auf diese Weise können wir eine beliebige Turingmaschine durch ein endliches Wort über dem Alphabet $\{\mathbf{0}, \mathbf{1}, \#\}$ beschreiben. Indem wir schließlich $\mathbf{0}$ durch $\mathbf{00}$, $\mathbf{1}$ durch $\mathbf{01}$ und $\#$ durch $\mathbf{11}$ codieren, erhalten wir sogar eine Codierung aller Turingmaschinen durch Wörter über dem Alphabet $\{\mathbf{0}, \mathbf{1}\}$.

Nicht alle Wörter über $\{\mathbf{0}, \mathbf{1}\}$ sind Beschreibungen von deterministischen Turingmaschinen, die die oben genannte Zusatzbedingungen erfüllen. Sei Maschine M^{ud}, die nur Übergänge $\delta(s_0, a) = (s_0, a, N)$ für $a \in A$ enthält und einen von s_0 verschiedenen Endzustand s_f hat, also bei keiner Eingabe je anhält.

Zu jedem Binärwort w definieren wir eine deterministische Turingmaschine M_w, die eine Wortfunktion berechnet, indem wir setzen: $M_w := M$, falls M eine deterministische Turingmaschine ist, die die oben genannten Zusatzbedingungen und falls w die Turingmaschine M in der oben erläuterten Weise beschreibt, und $M_w = M^{ud}$, andernfalls. Die von M_w berechnete Wortfunktion sei mit h_w bezeichnet.

Wir können jetzt w als Programm für die Funktion $h_w : E^* \dashrightarrow E^*$ auffassen. Die Abbildung, die einem Wort w über dem Alphabet $\{0,1\}$ die Funktion $h_w : E^* \dashrightarrow E^*$ zuordnet, kann als Programmiersprache aufgefasst werden. Diese Idee wollen wir allgemeiner formulieren. Wir wollen der Einfachheit halber nur Programmiersprachen betrachten, deren Programme Binärwörter sind.

Definition 7.4. *Sei E ein Alphabet. Eine Notation für die berechenbaren Funktionen $g : E^* \dashrightarrow E^*$ ist eine surjektive Funktion*

$$h' : \{0,1\}^* \to \{ \text{ berechenbare Wortfunktionen über } E^* \}$$

- *Jedem $w \in \{0,1\}^*$ wird eine berechenbare Wortfunktion zuordnet.*
- *Zu jeder berechenbaren Wortfunktion g gibt es ein Wort $w \in \{0,1\}^*$ derart, dass $h'(w)$ gerade die Funktion g ist.*

Wir schreiben oft auch h'_w für die Funktion $h'(w)$.

Wir möchten zwei Eigenschaften, die Notationen h' haben können, festhalten:

utm-Eigenschaft

Die Wortfunktion, die Wörter $w\#x$ für $w \in \{0,1\}^*$ und $x \in E^*$ auf $h'_w(x)$ abbildet, ist berechenbar.

smn-Eigenschaft

Ist $g : E^* \dashrightarrow E^*$ eine berechenbare Wortfunktion, so gibt es eine totale berechenbare Wortfunktion r derart, dass für alle $x \in \{0,1\}^*$ und alle $y \in E^*$ gilt:

$$g(x\#y) = h'_{r(x)}(y)$$

Satz 7.5. *Die von uns oben definierte Funktion h, die jedes Wort w aus $\{0,1\}^*$ auf eine berechenbare Funktion $h_w : E^* \dashrightarrow E^*$ abbildet, ist eine Notation der berechenbaren Wortfunktionen.*

h hat zudem die utm-Eigenschaft und die smn-Eigenschaft.

Beweis:

Nach Konstruktion ist h Notation der berechenbaren Wortfunktionen:

- h_w ist berechenbare Funktion für jedes w.
- Jede (normierte) Turingmaschine wird durch ein w kodiert.

Zur utm-Eigenschaft: Aus einem Wort w kann man das Verhalten der Turingmaschine M_w ablesen. Daher kann man eine Turingmaschine bauen, die bei Eingabe eines Wortes $w\#x$ für w aus $\{0,1\}^*$ die Turingmaschine M_w mit Eingabe x simuliert. Eine derartige Turingmaschine berechnet gerade die gewünschte Funktion.

Zur smn-Eigenschaft: Sei $g : E^* \dashrightarrow E^*$ eine berechenbare Wortfunktion. Dann gibt es eine Turingmaschine M_g, die g berechnet. Wenn ein Wort $w \in \{0,1\}^*$ gegeben ist, können wir daraus sicher eine Beschreibung $r(w)$ für eine Turingmaschine $M_{r(w)}$ berechnen, die bei Eingabe x die Maschine M_g angesetzt auf $w\#x$ simuliert.

Anwendung der smn-Eigenschaft in JAVA:

Betrachte Funktion zu Addition zweier Zahlen in JAVA aus Kapitel 1.2, wende smn-Eigenschaft für ersten Parameter m an:

```
class SMN {
public static void main(String args[]) {
System.out.println(¨import java.math.BigInteger;¨);
System.out.println(¨class Addition {¨);
System.out.println(¨public static void main(String args[]){¨);
System.out.println(¨BigInteger m=new BigInteger(\"¨
                +  args[0] + ¨\");¨ );
System.out.println(¨BigInteger n=new BigInteger(args[0]);¨);
System.out.println(¨while(n.compareTo(BigInteger.ZERO)>0){¨);
System.out.println(¨n = n.subtract(BigInteger.ONE);¨);
System.out.println(¨m = m.add(BigInteger.ONE);¨);
System.out.println(¨}¨);
System.out.println(¨System.out.println(m.toString());¨);
System.out.println(¨}¨);
System.out.println(¨}¨);
} }
```

Das Kürzel utm steht übrigens für *Universelle Turing-Maschine*. Eine universelle Turingmaschine arbeitet wie ein ganz normaler Computer in der Praxis, der als Eingabe einerseits ein Programm (hier das Wort w) erhält, und andererseits den jeweiligen Eingabewert (hier das Wort x) für das Programm. Es scheint sehr natürlich zu sein, von einer Notation oder Programmiersprache zu erwarten, dass man aus einem Programm w, geschrieben in der Programmiersprache, und einem Eingabewert x auch den Ausgabewert des Programms bei dieser Eingabe berechnen kann. Das heißt, man sollte von Programmiersprachen sicherlich verlangen, dass sie die utm-Eigenschaft haben.

Auch die smn-Eigenschaft ist natürlich. Sie besagt, dass man zu einer berechenbaren Funktion g und einem Wort x effektiv eine Programm $r(x)$ schreiben können sollte, das bei Eingabe eines weiteren Wortes y gerade das Wort $g(x\#y)$ ausrechnet. Typische Anwendungen sehen oft so aus, dass man aus x ein Programm für eine Maschine berechnen möchte, die bei Eingabe y die Maschine M_x in irgendeiner Form simulieren soll.

Das Kürzel smn hat nur historische Gründe. Es diente als mnemonische Abkürzung für eine mehrstellige Funktion, die in der ursprünglichen Fassung der smn-Eigenschaft auftrat.

Wir wollen nun Notationen vergleichen. Wir sagen, dass man eine Notation h'' in eine andere Notation h' *übersetzen* kann, wenn es eine berechenbare Wortfunktion $u : \{0,1\}^* \to \{0,1\}^*$ gibt, die jedes Programm w in $\{0,1\}^*$ bezüglich der ersten Notation auf ein Programm $u(w)$ in $\{0,1\}^*$ bezüglich der zweiten Notation abbildet, das die gleiche Wortfunktion beschreibt, d.h. es soll gelten $h''_w(x) = h'_{u(w)}(x)$ für alle Programme w aus $\{0,1\}^*$ und alle Wörter x aus E^* . Dann nennen wir u eine *Übersetzungsfunktion*. Wir nennen zwei Notationen *äquivalent*, wenn man jede in die andere übersetzen kann.

Satz 7.6 (Äquivalenzsatz von Rogers)**.** *Eine Notation für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation h äquivalent, wenn sie die utm-Eigenschaft und die smn-Eigenschaft hat.*

Beweis:

Sei h die von uns oben definierte Notation und h' irgendeine weitere Notation für die berechenbaren Wortfunktionen.

Wir nehmen zuerst an, dass die beiden Notationen äquivalent sind. Sei u eine Übersetzungsfunktion von h nach h' und u' eine Übersetzungsfunktion von h' nach h. Da unsere Notation h die utm-Eigenschaft hat und die Komposition von berechenbaren Funktionen wieder berechenbar ist, ist auch die Funktion berechenbar, die $w\#x$ mit w aus $\{0,1\}*$ auf $h_{u'(w)}(x) = h'_w(x)$ abbildet. Das ist gerade die utm-Eigenschaft für h'.

Sei $g : E^* \dashrightarrow E^*$ irgendeine berechenbare Wortfunktion. Da unsere Notation h die smn-Eigenschaft hat, gibt es eine berechenbare totale Funktion r derart, dass für alle x aus $\{0,1\}^*$ und alle y aus E^* gilt: $g(x\#y) = h_{r(x)}(y)$. Wegen $h_{r(x)}(y) = h'(u(r(x))(y)$ und weil die Funktion ur total und als Komposition berechenbarer Funktionen selbst wieder berechenbar ist, hat auch die Notation h' die smn-Eigenschaft.

Nun nehmen wir an, dass auch die Notation h' die utm-Eigenschaft und die smn-Eigenschaft hat. Wir wollen zeigen, dass h' zu h äquivalent ist: Wendet man die smn-Eigenschaft für h' auf die laut utm-Eigenschaft für h berechenbare Wortfunktion $w\#x \to h_w(x)$ an, so erhält man eine Übersetzungsfunktion r von h nach h'**.** Eine Übersetzungsfunktion in der umgekehrten Richtung erhält man, indem man die smn-Eigenschaft für h auf die laut utm-Eigenschaft

für h' berechenbare Funktion $w\#x \dashrightarrow h'_w(x)$ anwendet.

Äquivalente Notationen können als gleichwertig betrachtet werden, da man sie effektiv ineinander übersetzen kann. Während die utm-Eigenschaft sehr anschaulich ist und sicher eine Minimalforderung an jede Notation für berechenbare Funktionen, mag die smn-Eigenschaft auf den ersten Blick etwas unanschaulich sein. Aber man kann zeigen, dass man sie in dem Äquivalenzsatz von Rogers durch eine sehr anschauliche Eigenschaft ersetzen kann.

Satz 7.7. *Eine Notation h' für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation h äquivalent, wenn sie die utm-Eigenschaft und die folgende Eigenschaft hat:*

> *(effektive Komposition) Es gibt eine totale berechenbare Wortfunktion $r : \{0,1\}^*\#\{0,1\}^* \to \{0,1\}^*$ derart, dass für alle v, w aus $\{0,1\}^*$ und alle x aus E^* gilt:*
>
> $$h'_v(h'_w(x)) = h'_{r(v\#w)}(x)$$

r bestimmt also aus zwei 'Programmen' v, w ein 'Programm' $r(v\#w)$ für die Komposition der Funktionen h'_v und h'_w.

Wir lassen den Beweis weg.

Die effektive Kompositionseigenschaft ist sicherlich auch eine Minimalforderung an eine Notation für berechenbare Funktionen. Satz 7.7 zeigt, dass jede Notation, die diese beiden Minimalforderungen erfüllt, bereits zu der von uns oben explizit definierten Notation äquivalent ist. Die letzten drei Sätze zeigen also zweierlei. Einerseits gibt es eine Notation für die berechenbaren Wortfunktionen, die die utm-Eigenschaft, die smn-Eigenschaft und die effektive Kompositionseigenschaft erfüllt. Andererseits ist jede Notation, die die utm-Eigenschaft und die smn-Eigenschaft (bzw. die effektive Kompositionseigenschaft) erfüllt, bereits zu dieser Notation äquivalent. Es gibt also bis auf Äquivalenz nur eine „vernünftige "Notation für die berechenbaren Wortfunktionen. Man bezeichnet diejenigen Notationen der berechenbaren Wortfunktionen, die zu der von uns definierten Notation h äquivalent sind, auch als *Standardnotationen*. Man könnte Notationen auch als Programmiersprachen bezeichnen.

8 Entscheidbarkeit und rekursive Aufzählbarkeit

Wir werden in diesem Abschnitt über den Berechenbarkeitsbegriff für Wort- und Zahlenfunktionen Berechenbarkeitsbegriffe für Mengen einführen. Dann werden wir sehen, dass es auch für die Praxis wichtige Entscheidungsprobleme gibt, die unlösbar sind.

Zu einer Teilmenge A von E^* definieren wir zwei Funktionen:

- Die charakteristische Funktion $ch_A : E^* \to \{0,1\}$ ist total und nimmt auf allen Wörtern in A den Wert 1 an und auf allen Wörtern aus dem Komplement von A, also aus $E^* \setminus A$, den Wert 0:

$$ch_A(w) := \begin{cases} 1, & w \in A \\ 0, & w \notin A \end{cases}$$

- Die eingeschränkte charakteristische Funktion $ch'_A : E^* \dashrightarrow \{0,1\}$ ist partiell und nur auf den Wörtern in A definiert. Auf diesen Wörtern nimmt sie den Wert 1 an.

$$ch'_A(w) := \begin{cases} 1, & w \in A \\ \text{undefiniert}, & w \notin A \end{cases}$$

Definition 8.1. *1. Eine Teilmenge $A \subseteq E^*$ heißt **entscheidbar**, wenn ihre charakteristische Funktion $ch_A : E^* \to \{0,1\}$ **berechenbar** ist.*

*2. Eine Teilmenge $A \subseteq E^*$ heißt **semi-entscheidbar**, wenn ihre eingeschränkte charakteristische Funktion $ch'_A : E^* \dashrightarrow \{0,1\}$ **berechenbar** ist.*

Für Mengen $A \subseteq \mathbb{N}^k$ werden die Begriffe analog definiert. Der Begriff 'A ist unentscheidbar' bedeutet im folgenden das Gleiche wie 'A ist nicht entscheidbar'.

Eine Menge A von Wörtern ist also entscheidbar, wenn es einen Algorithmus (um genau zu sein: eine deterministische Turingmaschine) gibt, der bei Eingabe eines Wortes w nach endlich vielen Schritten anhält und sagt, ob w zur Menge gehört oder nicht.

Bei einer semi-entscheidbaren Menge wird weniger verlangt. Da muss es nur einen Algorithmus geben, der sich bei Eingabe eines Wortes w aus A wie folgt verhält: falls w aus A ist, muss er nach endlich vielen Schritten anhalten und verkünden, dass w aus A ist. Falls w nicht aus A ist, liefert er nie eine Antwort. Das Problem dabei ist, dass man von einem Wort, bei dem der Algorithmus zu einem bestimmten Zeitpunkt noch nicht angehalten hat, eben nicht weiß, ob er nie anhalten wird (dann liegt das Wort nicht in der Menge A) oder ob er vielleicht etwas später doch noch anhalten wird (dann liegt das Wort in der Menge A).

Der Zusammenhang zwischen den beiden Begriffen wird durch den folgenden Satz hergestellt.

Satz 8.2. *Eine Sprache A ist genau dann entscheidbar, wenn sowohl A als auch ihr Komplement $E^* \setminus A$ semi-entscheidbar sind.*

Beweis:

Wenn A entscheidbar ist, ist die charakteristische Funktion ch_A berechenbar. Wenn man eine Maschine, die sie berechnet, so umbaut, dass sie statt 0 auszugeben, in eine Endlosschleife läuft, so erhält man eine Maschine, die die eingeschränkte charakteristische Funktion ch'_A berechnet. Also ist A dann semi-entscheidbar. Ebenso folgt, dass das Komplement von A semi-entscheidbar ist: dazu muss man eine Maschine, die die charakteristische Funktion ch_A berechnet, so umbauen, dass sie statt 1 auszugeben in eine Endlosschleife läuft und anstelle von 0 jeweils 1 ausgibt.

Die Umkehrung folgt durch Kombination von Semi-Entscheidungsalgorithmen für A und Komplement $E^* \setminus A$: man lässt bei Eingabe eines Wortes w eine Maschine, die die eingeschränkte charakteristische Funktion von A berechnet, und eine Maschine, die die eingeschränkte charakteristische Funktion von Komplement $E^* \setminus A$ berechnet, parallel

laufen. Genau eine der beiden wird nach endlich vielen Schritten anhalten. Dann weiß man, ob w in A liegt oder im Komplement von A und kann entsprechend 1 oder 0 ausgeben.

Die Umkehrung folgt durch Kombination von Semi-Entscheidungsalgorithmen für A und Komplement $E^* \setminus A$: man lässt bei Eingabe eines Wortes w eine Maschine, die die eingeschränkte charakteristische Funktion von A berechnet, und eine Maschine, die die eingeschränkte charakteristische Funktion von Komplement $E^* \setminus A$ berechnet, parallel laufen. Genau eine der beiden wird nach endlich vielen Schritten anhalten. Dann weiß man, ob w in A liegt oder im Komplement von A und kann entsprechend 1 oder 0 ausgeben.

Die semi-entscheidbaren Mengen können auch auf andere Weise charakterisiert werden.

Definition 8.3. *Eine Sprache $A \subseteq E^*$ heißt rekursiv aufzählbar, falls A leer ist oder Bildbereich einer totalen berechenbaren Funktion $f : \mathbb{N} \to E^*$ ist:*

$$A := \{f(0), f(1), ...\}$$

Für Teilmengen $A \subseteq \mathbb{N}^k$ wird der Begriff analog definiert.

Satz 8.4. *Eine Sprache A aus E^* ist genau dann semi-entscheidbar, wenn sie rekursiv aufzählbar ist.*

Beweis:

Sei A rekursiv aufzählbar. Wenn A die leere Menge ist, ist A natürlich auch entscheidbar, denn die charakteristische Funktion der leeren Menge ist die konstante Funktion mit Wert 0, und die ist berechenbar. Sei A Wertebereich einer totalen berechenbaren Funktion $f : \mathbb{N} \to E^*$.

Der folgende Algorithmus hält bei Eingabe eines Wortes w genau dann an (und gibt dann 1 aus), wenn w aus A ist. Also ist A semi-entscheidbar.

```
INPUT(w);
FOR n = 0, 1, 2, 3, ... DO
    IF f(n) = w THEN OUTPUT(1) END;
END.
```

Sei nun A semi-entscheidbar und nicht leer; M sei eine Turingmaschine, die ch'_A berechnet. Wir müssen zeigen, dass es eine totale berechenbare Funktion $f : \mathbb{N} \to E^*$ mit $A := \{f(0), f(1), ...\}$ gibt. Dazu wählen wir zuerst irgendein Element $a \in A$. Der folgende Algorithmus berechnet eine derartige Funktion f.

```
INPUT(n);
k := p1(n); l := p2(n); w := ν(k);
IF  Angesetzt auf w stoppt M
          nach höchstens l Schritten mit Ausgabe von 1
THEN OUTPUT(w) ELSE OUTPUT(a)
END.
```

In dem Algorithmus verwenden wir die beiden LOOP-berechenbaren Funktionen p_1 und p_2, also die Umkehrfunktionen zu der Cantorschen Bijektion c zwischen $\mathbb{N}^2$ und $\mathbb{N}$ aus Abschnitt 51 Das heißt, wir interpretieren n als Codierung zweier Zahlen k und l.

Außerdem interpretieren wir die Zahl k mittels der Bijektion ν zwischen $\mathbb{N}$ und E^* aus Abschnitt 7.1 als Wort $w \in E^*$.

Wir testen also mit dem Algorithmus für alle möglichen Wörter $w \in E^*$ und alle möglichen Laufzeiten l, ob die Maschine M bei Eingabe von w nach l Schritten anhält. Falls ja, geben wir w aus, sonst das Wort a, das ja sicher in A liegt.

Der Algorithmus hält also immer an und gibt nur Wörter aus A aus (möglicherweise mit Wiederholungen). Da es aber andererseits zu jedem w aus A eine Schrittzahl l gibt, nach der M mit Ausgabe 1 anhält, gibt der Algorithmus bei Eingabe von $n = c(\nu^{-1}(w), l)$ dann w aus. Also ist A der Wertebereich von f.

Man nennt dieses Verfahren oft *dove-tailing*.

Die bisherigen Sätze gelten für Teilmengen von $\mathbb{N}$ oder von $\mathbb{N}^k$ natürlich ebenso wie für Sprachen. Damit sind die folgenden Aussagen äquivalent:

Für $A \subseteq E^*$ sind folgende Aussagen äquivalent:

- A ist Sprache vom Typ 0
- A wird von einer nichtdeterministischen Turingmaschine erkannt
- A wird von einer deterministischen Turingmaschine erkannt
- Es gibt eine deterministische Turingmaschine, die bei Eingabe eines Wortes $w \in E^*$ genau dann nach endlich vielen Schritten anhält, wenn $w \in A$ ist
- A ist semi-entscheidbar, d.h. die eingeschränkte charakteristische Funktion ch'_A ist berechenbar
- A ist Definitionsbereich einer berechenbaren Funktion
- A ist rekursiv aufzählbar
- A ist leer oder Wertebereich einer totalen berechenbaren Funktion

Äquivalent dazu ist auch

- A ist Wertebereich einer (partiellen) berechenbaren Funktion

(Beweis der noch nicht bewiesenen Äquivalenzen als Übung...)

Wir werden nun zeigen, dass es rekursiv aufzählbare Mengen gibt, die nicht entscheidbar sind. Dafür kommen wir auf die Notation von Turingmaschinen und berechenbaren Wortfunktionen aus 7.3 zurück. Wir hatten Turingmaschinen und damit auch berechenbare Wortfunktionen durch Wörter über dem Binäralphabet $\{0,1\}$ codiert. Ist w ein Binärwort, so sei M_w die wie in 7.3 durch w bezeichnete Turingmaschine und h_w die von M_w berechnete Wortfunktion.

Definition 8.5. *Das spezielle Halteproblem oder Selbstanwendbarkeitsproblem ist die Menge*

$$\begin{aligned} K &:= \{w \in \{0,1\}^* \mid M_w \textit{ angesetzt auf } w \\ &\qquad \textit{hält nach endlich vielen Schritten an}\} \\ &= \{w \in \{0,1\}^* \mid h_w(w) \textit{ ist definiert }\} \end{aligned}$$

Satz 8.6. *Das spezielle Halteproblem K ist rekursiv aufzählbar, aber nicht entscheidbar.*

Dass die Menge K rekursiv aufzählbar ist, liegt an der utm-Eigenschaft der Notation h für die berechenbaren Wortfunktionen. Wir geben noch einmal die Begründung: Ist w gegeben, so kann man daraus effektiv das Aussehen von M_w ablesen. Daher gibt es eine Turingmaschine, die bei Eingabe von w die Maschine M_w, angesetzt auf w, simuliert und genau dann anhält, wenn M_w, angesetzt auf w, anhält.

Wäre die Menge K entscheidbar, so wäre nach Satz 8.2 auch ihr Komplement, also die Menge $E^* \setminus K$, semi-entscheidbar. Dann gäbe es also eine Turingmaschine TM, die, angesetzt auf ein beliebiges Binärwort w genau dann nach endlich vielen Schritten anhält, wenn w nicht aus K ist. Sei x ein Codewort für so eine Maschine, d.h. M_x sei so eine Maschine. Wir setzen diese Maschine auf ihr eigenes Codewort x an und beobachten:

- Wenn M_x angesetzt auf x nach endlich vielen Schritten anhält, dann gilt nach Definition von M_x, dass $x \notin K$ ist. Aber nach Definition von K darf dann M_x angesetzt auf x niemals anhalten. Widerspruch!

- Wenn M_x angesetzt auf x nicht nach endlich vielen Schritten anhält, dann gilt nach Definition von M_x, dass $x \in K$ ist. Aber nach Definition von K muss dann M_x angesetzt auf x doch nach endlich vielen Schritten anhalten. Widerspruch!

In beiden Fällen erhalten wir einen Widerspruch. Also war die Annahme falsch, und K ist nicht entscheidbar.

Bemerkung 8.7. *1. Wenn wir uns den Beweis für die Nichtentscheidbarkeit von K ansehen, sehen wir, dass dies wieder eine Diagonalisierung ist, vgl.* **??**

2. *Im Beweis haben wir lediglich die utm-Eigenschaft der Programmiersprache h benutzt, und die auch nur zum Nachweis der rekursiven Aufzählbarkeit. Man kann den Satz daher direkt auf andere Notationen für die berechenbaren Wortfunktionen übertragen. Das heißt, ist h' irgendeine Notation für die berechenbaren Wortfunktionen, die die utm-Eigenschaft hat, so ist die Menge*

$$K = \{w \in \{0,1\}^* \mid h'_w(w) \text{ ist definiert}\}$$

rekursiv aufzählbar, aber nicht entscheidbar.

Das heißt, es gibt keine Notation für die berechenbaren Wortfunktionen, die die utm-Eigenschaft hat und die Eigenschaft, dass man bei Eingabe eines Programms entscheiden kann, ob es auf sich selbst angesetzt je anhält.

Die Unlösbarkeit des speziellen Halteproblems mag nicht besonders wichtig erscheinen. Aber ausgehend von diesem Problem kann man auch von anderen Problemen zeigen, dass sie nicht entscheidbar sind. Eine ganze Klasse von nicht entscheidbaren Sprachen oder Mengen kann man über das folgende Reduktionslemma erschließen:

Definition 8.8. *Seien $A, B \subseteq E^*$ Sprachen.*

Dann heißt A auf B reduzierbar ('$A \leq B$'), wenn es eine totale berechenbare Funktion $f : E^* \to E^*$ gibt, so dass für alle $x \in E^*$ gilt

$$x \in A \iff f(x) \in B$$

Die Grundidee dabei ist, dass wir bei $A \leq B$ das Problem A durch das Problem B lösen können (daher: 'A auf B reduziert'). Allerdings ist B in der Regel ein schwereres Problem als A, aber eventuell ist die 'Lösung' für B ja schon bekannt....

Es folgt sofort:

Satz 8.9. *Sei die Sprache A auf B mittels der Funktion f reduzierbar. Dann gilt:*

- *Ist B entscheidbar, so ist auch A entscheidbar.*
- *Ist A nicht entscheidbar, so ist auch B nicht entscheidbar.*
- *Ist B rekursiv-aufzählbar, so ist auch A rekursiv-aufzählbar.*
- *Ist A nicht rek.-aufzählbar, so ist auch B nicht rek.-aufzählbar.*

Beweis: Wir wollen annehmen, dass die charakteristische Funktion ch_B von B berechenbar ist. Dann ist auch die Kompositionsfunktion $ch_B \circ f$ berechenbar.

Es gilt aber:

$$ch_A(x) = 1 \iff x \in A \iff f(x) \in B \iff ch_B(f(x)) = 1$$

Analog gilt

$$ch_A(x) = 0 \iff x \notin A \iff f(x) \notin B \iff ch_B(f(x)) = 0$$

Im Fall der Semi-Entscheidbarkeit ersetzt man ch durch ch' und beachtet, dass ch' im Fall der Nichtzugehörigkeit undefiniert ist.

Definition 8.10. *Das allgemeine Halteproblem ist die Sprache*

$$H = \{\ w\$x \in \{0,1\}^*\{\$\}E^* \ \mid M_w \text{ angesetzt auf } x \text{ hält an}\}$$

Dabei sei $\$$ *irgendein Symbol, das in* E *nicht vorkommt.*

Satz 8.11. *Das allgemeine Halteproblem ist rekursiv aufzählbar, aber nicht entscheidbar.*

Beweis:

Dass H semi-entscheidbar und daher auch rekursiv aufzählbar ist, folgt wieder aus der utm-Eigenschaft. Zum Beweis der Nicht-Entscheidbarkeit reduzieren wir K auf H. Dies geschieht mit der totalen und sicherlich berechenbaren Abbildung f definiert durch $f(w) = w\$w$.

Der folgende Satz von Rice wird auch über die Reduktionsmethode bewiesen. Er liefert eine große Klasse von nichtentscheidbaren Problemen.

Satz 8.12 (Rice)**.** *Sei* R *die Klasse aller Turing-berechenbaren Wortfunktionen über dem Alphabet* E*.*

Sei S *eine echte, nichttriviale Teilmenge von* R*, d.h.* $\emptyset \neq S \neq R$

Dann ist die folgende Sprache unentscheidbar:

$$C(S) = \{w \in \{0,1\}^* \mid \text{die von } M_w \text{ berechnete Funktion } h_w \text{ liegt in } S\}$$

Beweis: Sei S mit $\emptyset \neq S \neq R$ gegeben, sei ud die überall undefinierte (berechenbare!) Funktion.

Also $ud \in S$ oder $ud \notin S$. Wir behandeln diese beiden Fälle getrennt.

(a) Nehmen wir an, dass ud aus S ist. Da nach Voraussetzung $S \neq R$, gibt es eine Funktion q aus dem Komplement von S und eine sie berechnende Turingmaschine Q.

Wir betrachten die Turingmaschine TM, die sich wie folgt verhalte:

> Bei Eingabe von $w\#x$ simuliert TM erst die Maschine M_w angesetzt auf w. Kommt diese Rechnung zu einem Ende, so soll TM anschließend Q angesetzt auf x simulieren.

g sei die von TM berechnete Funktion.

Mit der smn-Eigenschaft von h gibt es ein totales berechenbares $r : \{0,1\}^* \to \{0,1\}^*$ mit $g(w\#x) = h_{r(w)}(x)$

Damit gilt:

- Hält M_w auf w, so ist $h_{r(w)} = q$.
- Hält M_w nicht auf w, so ist $h_{r(w)} = ud$

Dann gilt:

$w \in K \Rightarrow$ Angesetzt auf w stoppt M_w
$\Rightarrow$ $h_{r(w)}$ ist die Funktion q
$\Rightarrow$ $h_{r(w)}$ liegt nicht in S
$\Rightarrow$ $r(w) \notin C(S)$.

sowie

$w \notin K \Rightarrow$ Angesetzt auf w stoppt M_w nicht
$\Rightarrow$ $h_{r(w)}$ ist die Funktion ud
$\Rightarrow$ $h_{r(w)}$ liegt in S
$\Rightarrow$ $r(w) \in C(S)$.

Damit vermittelt r eine Reduktion vom Komplement von K auf $C(S)$, was für unseren Zweck ausreicht, da auch das Komplement von K unentscheidbar ist.

(b) Ist ud nicht aus S, so zeigt man analog: Es existiert eine berechenbare Funktion, die eine Reduktion von K nach $C(S)$ vermittelt.

Bemerkung 8.13.
(1) Der Beweis des Satzes von Rice nutzt nur die utm-Eigenschaft und die smn-Eigenschaft von h. Eine andere Formulierung ist daher:

> *Ist h' Notation der berechenbaren Wortfunktionen mit utm-Eigenschaft und smn-Eigenschaft, dann gilt:*
> *Für jede Teilmenge S von R mit $\emptyset \neq S \neq R$ ist die Menge*
>
> $$C(S) = \{w \in \{0,1\}^* | h'_w \text{ liegt in } S\}$$
>
> *nicht entscheidbar.*

(2) Anwendungsbeispiele:

- $S_1 = \{f$ *berechenbar und total* $\}$
  Man kann bei (realen) Programmen i.d.R. nicht entscheiden, ob sie immer halten.
- $S_2 = \{g\}$ *für ein gegebenes* g
  Man kann bei (realen) Programmen i.d.R. nicht entscheiden, ob sie eine exakt vorgegebene Funktion berechnen
- $S_3 = \{g \mid g(\varepsilon) = \varepsilon\}$
  Man kann bei (realen) Programmen i.d.R. nicht entscheiden, ob sie eine vorgegebene Spezifikation erfüllen (hier: $g(\varepsilon) = \varepsilon$)

Die Beispiele zeigen, dass zum Beispiel Verifikation von Software schwierig ist, sobald die Programmiersprache 'vernünftig' ist (d.h. alle berechenbaren Funktionen umfasst und utm/smn-Eigenschaften hat).

Es gibt Probleme, die 'noch unlösbarer' als das spezielle oder allgemeine Halteproblem sind.

Dazu gehört das Äquivalenzproblem für Turingmaschinen:

$$A = \{v\$w \mid \text{die Turingmaschinen } M_v \text{ und } M_w \text{ berechnen dieselbe Funktion}\}$$

Man kann zwar K auf A reduzieren, aber A nicht auf K. Darauf aufbauend kann man ganze Hierarchien im Grad der Unlösbarkeit schwieriger werdender Probleme aufbauen.

9 Das Postsche Korrespondenzproblem

Das folgende von Post formulierte Problem (PCP = Post's Correspondence Problem) formuliert die folgende algorithmische Aufgabenstellung:

- Eingabe: eine endliche Folge von Wortpaaren

  $$(x_1, y_1),\ (x_2, y_2), ...,\ (x_k, y_k)$$

  wobei $k \geq 1$ und $x_i, y_i \in E^+$ seien, (also $x_i, y_i \neq \varepsilon$).

- Frage: gibt es eine Folge $\mathcal{I}$ von Indizes i_1,..., i_n aus $\{1, 2, ..., k\}$ mit $n \geq 1$ und mit

  $$x_{i_1} ... x_{i_n} = y_{i_1} ... y_{i_n}$$

Eine derartige Folge $\mathcal{I}$ nennt man *Lösung* des Korrespondenzproblems (x_1, y_1), (x_2, y_2),..., (x_k, y_k).

Beispiel 9.1.

i	x_i	y_i
1	bab	a
2	ab	abb
3	a	ba

Bei der Indexfolge $\mathcal{I} = (2, 1, 3)$ *ergibt sich* $ab\ bab\ a = abb\ a\ ba$.

Dass das PCP *ein hohes Maß an Komplexität besitzt, zeigt das folgende harmlos aussehende Beispiel aus dem Buch von Schöning:*

i	x_i	y_i
1	001	0
2	01	011
3	01	101
4	10	001

Dieses Problem besitzt eine Lösung, aber die kürzeste Lösung besteht aus 66 Indizes:

$$2, 4, 3, 4, 4, 2, 1, 2, 4, 3, 4, 3, 4, 4, 3, 4, 4, 2, 1, 4, 4, 2, 1, 3, 4, 1, 1, 3,$$
$$4, 4, 4, 2, 1, 2, 1, 1, 1, 3, 4, 3, 4, 1, 2, 1, 4, 4, 2, 1, 4, 1, 1, 3, 4, 1, 1, 3,$$
$$1, 1, 3, 1, 2, 1, 4, 1, 1, 3.$$

Bemerkung 9.2. *Die naive Vorgehensweise, die bei gegebener Eingabe* (x_1, y_1), (x_2, y_2),..., (x_k, y_k) *systematisch durch Probieren immer längere Indexfolgen* $i_1, i_2, ...$ *daraufhin untersucht, ob sie eine Lösung darstellen und im positiven Fall stoppt, zeigt, dass das* PCP *semi-entscheidbar (oder rekursiv aufzählbar) ist: Es gibt ein Verfahren, das bei Eingaben, die eine Lösung besitzen, diese nach endlich vielen Schritten findet und stoppt. Bei Eingaben, die keine Lösung besitzen, stoppt das Verfahren jedoch nicht.*

Wir zeigen im Folgenden, dass das PCP unentscheidbar ist, d.h. dass es keinen Algorithmus gibt, der bei Eingabe einer nichtleeren Folge von Paaren nichtleerer Wörter nach endlich vielen Rechenschritten entscheidet, ob das durch die Folge gegebene Korrespondenzproblem eine Lösung hat oder nicht. Wir werden also zeigen, dass die Menge

$$\begin{aligned} PCP \ = \ \{&((x_1, y_1), (x_2, y_2), ..., (x_k, y_k)) \mid \\ &k \in \mathbb{N}, x_i, y_i \in E^+ \text{ für } i = 1, \ldots, k, \text{ und} \\ &((x_1, y_1), (x_2, y_2), ..., (x_k, y_k)) \text{ hat eine Lösung}\} \end{aligned}$$

eine unentscheidbare Menge ist. Wir machen dazu Gebrauch von der Idee der Reduktion und unserem Wissen, dass das Selbstanwendbarkeitsproblem K unentscheidbar ist. Wir reduzieren zunächst K auf ein anderes Problem, genannt $MPCP$ ('modifiziertes PCP'), und dann $MPCP$ auf PCP.

Das Problem $MPCP$ ist definiert wie folgt:

- Eingabe: wie beim Problem PCP.
- Frage: gibt es eine Lösung i_1, i_2,...,i_n des PCP mit $i_1 = 1$?

Lemma 9.3. *Das Problem $MPCP$ ist auf PCP reduzierbar.*

Beweis:

Wir führen zunächst einige Notationen ein: Seien \$ und # neue Symbole, die im Alphabet E des $MPCP$s nicht vorkommen. Für ein Wort $w = a_1a_2...a_m$ aus E^+ sei

$$\begin{aligned} w^a &:= \#a_l\#a_2\#...\#a_m\# \\ w^b &:= \ \ a_1\#a_2\#...\#a_m\# \\ w^c &:= \#a_1\#a_2\#...\#a_m \end{aligned}$$

Jeder Eingabe von k Paaren $P = ((x_1, y_1), (x_2, y_2),..., (x_k, y_k))$ für das $MPCP$ wird nun die folgende Eingabe von $k+2$ Paaren zugeordnet:

$$f(P) = ((x_1^a, y_1^c), (x_1^b, y_1^c), (x_2^b, y_2^c), ..., (x_k^b, y_k^c), (\$, \#\$))$$

Diese Abbildung f ist offensichtlich berechenbar, da sie im Prinzip nur die Eingabe leicht abändert. Wir zeigen nun, dass diese Funktion f eine Reduktion von $MPCP$ nach PCP ist. Dazu müssen wir zeigen:

$$\begin{aligned} & P \text{ besitzt eine Lösung mit } i_1 = 1 \\ \iff \quad & f(P) \text{ besitzt (irgend)eine Lösung} \end{aligned}$$

Beweis von '$\Longrightarrow$': P besitze eine Lösung i_1, i_2,...,i_n mit $i_1 = 1$ ($MPCP$!).

Dann ist $(1, i_2{+}1, ..., i_n{+}1, k{+}2)$ eine Lösung für $f(P)$.

Beweis von '$\Longleftarrow$': $f(P)$ besitze eine Lösung i_1, i_2,...,i_n aus $\{1, ..., k{+}2\}$.

Dann kann wegen der Bauart der Wortpaare nur $i_1 = 1$ und $i_n = k{+}2$ sein. Wenn keine echte Anfangsteilfolge der Lösung ebenfalls Lösung ist (in dem Fall betrachten wir die kürzeste Anfangsteilfolge, die Lösung ist), muss außerdem i_j aus $\{2, ..., k{+}l\}$ für $2 \leq j \leq n-1$ sein. In diesem Fall ist $(1, i_2{-}1, ..., i_{n-1}{-}1)$ eine Lösung für P.

Lemma 9.4. *K ist reduzierbar auf $MPCP$.*

Beweis:

Wir müssen eine totale berechenbare Funktion angeben, die jedes Wort w (für die Codierung einer Turingmaschine M_w und gleichzeitig als Eingabewort $w \in E^*$ für M_w) überführt in eine Folge (x_1, y_1), (x_2, y_2),..., (x_k, y_k) von Wortpaaren (die die Voraussetzungen des PCP erfüllt), so dass gilt:

(*) M_w angesetzt auf w hält genau dann nach endlich vielen Schritten an, wenn $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$ eine Lösung mit $i_1 = 1$ besitzt.

Sei w gegeben und $M_w = (S, E, A, \delta, s_0, \square, F)$.

Das Alphabet für das zu konstruierende $MPCP$ wird $A \cup S \cup \{\#\}$ sein; das erste Wortpaar soll $(\#, \#\Box s_0 w \Box \#)$ sein. Die uns interessierenden Lösungswörter müssen also mit diesem Paar beginnen.

Die weiteren Paare werden wie folgt konstruiert:

1. *Kopierregeln*:

(a, a) für alle $a \in A \cup \{\#\}$

2. *Überführungsregeln*:

Für alle Übergänge der Turingmaschine M_w:

$(sa, s'c)$	falls $\delta(s, a) = (s', c, N)$
(sa, cs')	falls $\delta(s, a) = (s', c, R)$
$(bsa, s'bc)$	falls $\delta(s, a) = (s', c, L)$, für alle b aus A
$(\#sa, \#s'\Box c)$	falls $\delta(s, a) = (s', c, L)$
$(s\#, s'c\#)$	falls $\delta(s, \Box) = (s', c, N)$
$(s\#, cs'\#)$	falls $\delta(s, \Box) = (s', c, R)$
$(bs\#, s'bc\#)$	falls $\delta(s, \Box) = (s', c, L)$, für alle b aus A

3. *Löschregeln*:

(as_f, s_f) und $(s_f a, s_f)$ für alle $a \in A$ und $s_f \in F$.

4. *Abschlussregeln*:

$(s_f \#\#, \#)$ für alle $s_f \in F$.

Wir halten zuerst fest, dass die Funktion, die jedem w wie oben angegeben, eine derartige Folge von Wortpaaren zuordnet, sicher berechenbar ist. Jetzt müssen wir noch zeigen, dass sie auch die Bedingung (*) erfüllt.

Falls die Turingmaschine M_w bei Eingabe w stoppt, so gibt es eine Folge von Konfigurationen $(k_0, k_1, ..., k_t)$, so dass gilt: $k_0 = \Box s_0 w \Box$, k_t ist eine Endkonfiguration (also $k_t = u s_f v$ mit $u, v \in A^*$ und $s_f \in F$), und $k_i \vdash k_{i+1}$ für $i = 0, ..., t-1$.

Die oben angegebene Eingabe für das $MPCP$ besitzt dann eine Lösung mit einem Lösungswort der Form

$$\#k_0\#k_1\#...\#k_t\#k_t'\#k_t''\#...\#s_f\#\#$$

Hierbei entstehen $k_t', k_t'', ...$ aus $k_t = u s_f v$ durch Löschen von Nachbarsymbolen von s_f.

Der erste Teil der Lösung baut sich so auf, dass die Folge der x_{ij} eine Konfiguration hinter der Folge der y_{ij} 'hinterherhinkt'. Der 'Überhang' hat also immer die Länge einer Konfiguration, skizzierbar wie folgt:

$$\begin{array}{lccccccccc} x: & \# & k_0\# & k_1\# & k_2\# & ... & k_t\# & k_t'\# & k_t''\# & ... & s_f\#\# \\ y: & \#k_0\# & k_1\# & k_2\# & ... & k_t\# & k_t'\# & k_t''\# & ... & s_f\# & \# \end{array}$$

Falls umgekehrt die obige Eingabe für das $MPCP$ eine Lösung (mit $i_1 = 1$) besitzt, so lässt sich aus dieser Lösung in ähnlicher Weise eine stoppende Rechnung der Turingmaschine M_w bei Eingabe w ablesen.

Damit ist gezeigt, daß die Abbildung, die w, wie oben angegeben, eine Eingabe für das $MPCP$ zuordnet, eine Reduktion von K nach $MPCP$ vermittelt.

Unter Ausnützen der Tatsache, dass K unentscheidbar ist (Satz 8.6) und wegen Satz 8.9 über die Reduktion erhält man wegen Lemma 9.4, dass $MPCP$ unentscheidbar ist, und dann wegen Lemma 9.3, dass auch PCP unentscheidbar ist.

Satz 9.5. *Das Postsche Korrespondenzproblem PCP ist unentscheidbar.*

Sogar der folgende spezielle Fall ist unentscheidbar:

Satz 9.6. *Das PCP ist bereits dann nicht entscheidbar, wenn man sich auf das Alphabet $\{0,1\}$ beschränkt. Diese Problemvariante heißt $01-PCP$.*

Zum Beweis zeige, dass PCP auf $01-PCP$ reduzierbar ist:

Sei $E = \{a_1, ..., a_m\}$ das Alphabet des gegebenen PCPs. Jedem Symbol $a_j \in E$ ordne das Wort $a'_j = 01^j \in \{0,1\}^*$ zu; verallgemeinere dies auf beliebige Wörter

$$w = a_1...a_n \in E^+ \quad \mapsto \quad w' = a'_1...a'_n \in \{0,1\}^*$$

Dann gilt offensichtlich:

$$\begin{array}{ll} & (x_1, y_1), ..., (x_k, y_k) \text{ hat eine Lösung} \\ \iff & (x'_1, y'_1), ..., (x'_k, y'_k) \text{ hat eine Lösung} \end{array}$$

10 Unentscheidbare Grammatikprobleme

Wir hatten im Kapitel über die regulären Sprachen gesehen, dass alle Fragen, die wir dort über reguläre Sprachen gestellt hatten, entscheidbar waren. Später hatten wir aber angemerkt, dass es zu kontextfreien Sprachen Fragestellungen gibt, die nicht entscheidbar sind. Damals hatten wir den Nachweis aber in Ermangelung eines präzise definierten Entscheidbarkeitsbegriffs noch nicht führen können. Mittlerweile haben wir einen präzise definierten Entscheidbarkeitsbegriff kennengelernt. In diesem Kapitel werden wir nun mit Hilfe der Unentscheidbarkeit des Postschen Korrespondenzproblems nachweisen, dass einige Fragestellungen aus dem Bereich der kontextfreien und kontextsensitiven Grammatiken nicht entscheidbar sind. Sie betreffen das Schnitt-, das Endlichkeits- und das Äquivalenzproblem.

Satz 10.1. *Die folgenden Fragestellungen sind unentscheidbar:*

Gegeben seien zwei kontextfreie Grammatiken G_1 und G_2.

1. *Ist $L(G_1) \cap L(G_2) = \emptyset$?*
2. *Ist $|L(G_1) \cap L(G_2)| = \infty$?*
3. *Ist $L(G_1) \cap L(G_2)$ kontextfrei ?*
4. *Ist $L(G_1) \subseteq L(G_2)$?*
5. *Ist $L(G_1) = L(G_2)$?*

Beweis:

(1) Jedem Postschen Korrespondenzproblem $P = \{(x_1, y_1), ..., (x_k, y_k)\}$ über dem Alphabet $\{0, 1\}$ können effektiv zwei kontextfreie Grammatiken $G_1 = (N_1, T, P_1, S)$ und $G_2 = (N_2, T, P_2, S)$ wie folgt zugeordnet werden:

$$N_1 := \{S, A, B\}, \quad N_2 := \{S, R\}, \quad T = \{0, 1, \$, a_1, ..., a_k\}$$

Die Grammatik G_1 besitzt die Ableitungsregeln P_1

$$\begin{array}{lcl} S & \rightarrow & A\$B \\ A & \rightarrow & a_1Ax_1 \mid ... \mid a_kAx_k \mid a_1x_1 \mid ... \mid a_kx_k \\ B & \rightarrow & y_1^rBa_1 \mid ... \mid y_k^rBa_k \mid y_1^ra_1 \mid ... \mid y_k^ra_k \end{array}$$

wobei mit w^r das gespiegelte Wort zu w bezeichnet wird.

Diese Grammatik G_1 erzeugt die Sprache

$$L_1 = \{a_{i_1}...a_{i_n}x_{i_n}...x_{i_1}\$y_{j_1}^r...y_{j_m}^ra_{j_m}...a_{j_1} \mid n, m \geq 1, i_\nu, j_\mu \in \{1, .., k\}\}$$

Betrachte eine zweite Grammatik G_2 mit folgenden Regeln P_2:

$$\begin{array}{lcl} S & \rightarrow & a_1Sa_1 \mid ... \mid a_kSa_k \mid R \\ R & \rightarrow & 0R0 \mid 1R1 \mid \$ \end{array}$$

Sie erzeugt die Sprache

$$L_2 = \{uv\$v^ru^r \mid u \in \{a_1, ..., a_k\}^*, v \in \{0, 1\}^*\}$$

Beide Sprachen sind übrigens sogar deterministisch kontextfrei.

Das Korrespondenzproblem P besitzt eine Lösung $i_n, ..., i_1$, also $x_{i_n}...x_{i_1} = y_{i_n}...y_{i_1}$, genau dann, wenn $L_1 \cap L_2$ nicht leer ist.

Im Schnitt liegt dann das Wort:

$$w = a_{i_1}...a_{i_n}x_{i_n}...x_{i_1}\$y^r_{i_1}...y^r_{i_n}a_{i_n}...a_{i_1}$$

Man kann sagen, dass das Enthaltensein eines Wortes wie in L_1 in L_2 erzwingt, dass erstens die verwendeten Indizes bei den den x_i und den y_j die Gleichen sind und dass zweitens das aus den x_i gebildete Wort gleich dem aus den y_j gebildeten Wort ist.

Das bedeutet, dass $P \rightarrow (G_1, G_2)$ eine Reduktion von PCP auf das Komplement des Schnittproblems ist. Da PCP unentscheidbar ist, ist also auch das Komplement des Schnittproblems unentscheidbar, und damit auch das Schnittproblem selbst unentscheidbar.

(2) Wenn P mindestens eine Lösung besitzt, so besitzt P auch unendlich viele Lösungen, indem man die Lösungsfolge beliebig oft wiederholt. Damit ist das Problem, ob $|L(G_1) \cap L(G_2)| = \infty$ ist, ebenso unentscheidbar, mit der gleichen Reduktion.

(3) In dem Falle, dass $L(G_1) \cap L(G_2)$ nicht leer und damit unendlich ist, ist weiterhin $L_1 \cap L_2$ keine kontextfreie Sprache. Wir skizzieren eine Begründung, die auf dem Pumping-Lemma für kontextfreie Sprachen beruht. Nehmen wir einmal an, $L_1 \cap L_2$ wäre kontextfrei. Alle Wörter in $L_1 \cap L_2$ haben die oben angegebene Form $uv\$v^ru^r$, wobei die Indexfolge der a_i in u gleich der gespiegelten Indexfolge der x_i in v ist und die Indexfolge der a_i in u^r gleich der gespiegelten Indexfolge der y_i in v^r ist. Ein derartiges Wort besteht also aus vier Wortteilen, die alle voneinander abhängen.

Wir wählen nun ein Wort der Form $uv\$v^ru^r$ aus $L_1 \cap L_2$ derart, dass schon u und v alleine länger als die Pumpingzahl n aus dem Pumping-Lemma sind. Laut dem Pumping-Lemma kann dieses Wort dann in 5 Teilwörter zerlegt werden, von denen die drei inneren zusammen nicht länger als n sind, und Teilwort 2 und 4 nicht beide leer sein dürfen. Beim Aufpumpen der Teilwörter 2 und 4 werden dann höchstens zwei der vier Wortteile von $uv\$v^ru^r$ verändert. Damit geht aber die spezielle Struktur des Wortes $uv\$v^ru^r$ verloren, da man zum Erhalten dieser Struktur alle vier Wortteile ändern müsste. Daher ist $L_1 \cap L_2$ nicht kontextfrei und das Problem, die Kontextfreiheit festzustellen, unentscheidbar, wieder mit der gleichen Reduktion.

Annahme: L sei kontextfrei (und unendlich), d.h. Pumping-Lemma für kontextfreie Sprachen sei anwendbar, n sei die Pumpingzahl.

Wähle $ab\$c^rd^r \in L$ mit $n < |a| = |d| \leq |b| = |c|$

Betrachte beliebige Zerlegung $ab\$c^rd^r = uvwxy$ mit $uwy = uv^0wx^0y \in L$, $|vx| \neq 0$ und $|vwx| \leq n$

Fall (1): $|u| \geq |a|$: damit a Präfix von uwy, aber b, c, d durch a festgelegt $\Rightarrow$ Widerspruch zu $|vx| > 0$ und $uwy \in L$.

Fall (2): $|u| < |a|$: damit $|uvwx| < |ab|$ und d Suffix von uwy, aber a, b, c durch d festgelegt $\Rightarrow$ wieder Widerspruch zu $|vx| > 0$ und $uwy \in L$.

Damit also: $|L| = \infty \Rightarrow L$ nicht kontextfrei

Zusammen:

$$\begin{array}{lllll} P \in PCP & \Rightarrow & |L| = \infty & \Rightarrow & L \text{ nicht kontextfrei} \\ P \notin PCP & \Rightarrow & L = \emptyset & \Rightarrow & L \text{ kontextfrei} \end{array}$$

Die weiteren Reduktionen sollen nur skizziert werden.

(4) und (5): Man kann ausnutzen, dass die Sprachen L_1, L_2 tatsächlich sogar deterministisch kontextfrei sind und dass diese Klasse unter Komplementbildung abgeschlossen ist. Sie ist sogar effektiv unter Komplementbildung abgeschlossen, d.h. zu G_1 und G_2 kann man effektiv Grammatiken G'_1, G'_2 konstruieren mit $L(G'_1)$ =Komplement(L_1) und $L(G'_2)$ =Komplement(L_2). Dann folgt

$$\begin{array}{lll} L_1 \cap L_2 = \emptyset & \Leftrightarrow & L_1 \subseteq L(G'_2) \\ & \Leftrightarrow & L_1 \cup L(G'_2) = L(G'_2) \\ & \Leftrightarrow & L(G_3) = L(G'_2) \end{array}$$

Dabei ist G_3 eine aus G_1 und G_2 konstruierte, kontextfreie Grammatik, die die (kontextfreie, aber nicht unbedingt deterministisch kontextfreie) Vereinigung der Sprachen $L_1 \cup L(G_2')$ erzeugt.

Insgesamt haben wir zwei Reduktionen $P \to (G_1, G_2')$ bzw. $P \to (G_3, G_2')$ vom PCP auf das Inklusions- bzw. Äquivalenzproblem bei kontextfreien Sprachen. Beide Probleme sind damit unentscheidbar.

(Das Äquivalenzproblem für deterministisch kontextfreie Sprachen ist bekanntlich entscheidbar).

Mit der Nichtentscheidbarkeit des Äquivalenzproblems für nichtdeterministische kontextfreie Grammatiken gilt das Gleiche für nichtdeterministische Kellerautomaten, kontextsensitive Grammatiken, LBA's, Turingmaschinen, LOOP-, WHILE-Programme etc.

Da die Sprachen L_1 und L_2 sogar deterministisch kontextfrei sind, können wir den vorigen Satz noch verschärfen.

Satz 10.2. *Deterministisch kontextfreie Sprachen L_1 und L_2 seien durch ihre deterministische Kellerautomaten gegeben.*

Dann sind die folgenden Fragestellungen unentscheidbar:

1. *Ist $L_1 \cap L_2 = \emptyset$?*
2. *Ist $|L_1 \cap L_2| = \infty$?*
3. *Ist $L_1 \cap L_2$ kontextfrei ?*
4. *Ist $L_1 \subseteq L_2$?*

Wir erwähnen noch einige weitere Unentscheidbarkeitsergebnisse zu formalen Sprachen. Beweise findet man z.B. im Buch von Schöning, Kapitel 2.8.

Satz 10.3. *Betrachte kontextfreie Grammatiken G.*

Dann sind die folgenden Fragen unentscheidbar:

1. *Ist G mehrdeutig?*
2. *Ist das Komplement von $L(G)$ kontextfrei?*
3. *Ist $L(G)$ regulär?*
4. *Ist $L(G)$ deterministisch kontextfrei?*

Satz 10.4. *Betrachte kontextfreie Sprachen L_1 und reguläre Sprachen L_2.*
Es ist nicht entscheidbar, ob $L_1 = L_2$ gilt.

Das letzte Ergebnis betrifft kontextsensitive Sprachen und lautet:

Satz 10.5. *Das Leerheits- und das Endlichkeitsproblem für Chomsky Typ-1 Sprachen sind nicht entscheidbar.*

11 Der Gödelsche Satz

In einem Ausblick wollen wir über die Auswirkungen der Nichtberechenbarkeit auf die Axiomatik und Beweisbarkeit in der Zahlentheorie und Geometrie sprechen. Dies soll aber in einer eher informellen Form geschehen. Die Menge der arithmetischen Formeln können wir als (kontextfreie) Sprache $\mathcal{F}$ über folgendem Alphabet darstellen:

$$\{0, 1, x, (,), +, *, =, \neg, \wedge, \vee, \exists, \forall\}$$

Zunächst definierern wir die Menge $\mathcal{T}$ aller Terme durch

- Jede natürliche Zahl $i \in \mathbb{N}$ ist ein Term
- Jede Variable x_i ist ein Term (mit $i \in \mathbb{N}$), setze $V := \{x_i \mid i \in \mathbb{N}\}$
- Sind t_1, t_2 Terme, dann auch $(t_1 + t_2)$ und $(t_1 * t_2)$

Die Sprache $\mathcal{F}$ der Formeln wird dann definiert durch:

- Sind t_1, t_2 Terme, so ist $(t_1 = t_2)$ eine Formel
- Sind F, G Formeln, so auch $\neg F$, $(F \wedge G)$ und $(F \vee G)$
- Ist x Variable und F Formel, so sind $\exists x\ F$ und $\forall x\ F$ Formeln.

Dabei werden Zahlen i und die Indizes bei x_i binär notiert.
Terme können ausgewertet werden, wenn den Variablen Werte zugewiesen werden:

- Eine Belegung ϕ ist eine Abbildung $\phi : V \to \mathbb{N}$
- ϕ kann auf Terme erweitert werden durch

$$\begin{aligned} \phi(n) &= n \\ \phi(\ (t_1 + t_2)\ ) &= \phi(t_1) + \phi(t_2) \\ \phi(\ (t_1 * t_2)\ ) &= \phi(t_1) \cdot \phi(t_2) \end{aligned}$$

Beispiel: Mit $\phi(x_1) = 5$ und $\phi(x_2) = 6$ ist $\phi(\ (x_1 * (2 + x_2))\ ) = 40$
ϕ kann auf Formeln erweitert werden; $\phi(F)$ ist dabei einer der Wahrheitswerte '$wahr$' oder '$falsch$'

- $\phi(\ (t_1 = t_2)\ ) = wahr$ genau dann, wenn $\phi(t_1) = \phi(t_2)$
- $\phi(\neg F)$ ist $wahr$, falls $\phi(F)$ nicht $wahr$ ist,
- $\phi(\ (F \wedge G)\ )$ ist $wahr$, falls $\phi(F)$ und $\phi(G)$ $wahr$ sind,
- $\phi(\ (F \vee G)\ )$ ist $wahr$, falls $\phi(F)$ oder $\phi(G)$ $wahr$ ist,
- $\phi(\ \exists y\ F)$ ist $wahr$, falls $\phi'(F) = wahr$ bei einer beliebigen Belegung ϕ' mit $\phi'(x) = \phi(x)$ für $x \neq y$,
- $\phi(\ \forall y\ F\ )$ ist $wahr$, falls $\phi'(F) = wahr$ bei allen Belegungen ϕ' mit $\phi'(x) = \phi(x)$ für $x \neq y$.

Beispiele:

- $\phi(\ (x_1 + 1) = x_2\ ) = wahr$ für jedes ϕ mit $\phi(x_1) + 1 = \phi(x_2)$
- $\phi(\ \exists x_1\ ((x_1 + 1) = x_2)\ ) = wahr$ für jedes ϕ mit $\phi(x_2) > 0$

- $\phi(\ \exists x_1\ ((x_1 + 1) = x_2)\ ) = \boldsymbol{falsch}$ für jedes ϕ mit $\phi(x_2) = 0$
- $\phi(\ \exists x_2\ ((x_1 + 1) = x_2)\ ) = \boldsymbol{wahr}$ für jedes beliebige ϕ

Eine Funktion $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ heißt arithmetisch repräsentierbar, wenn es eine arithmetische Formel F gibt, so dass für alle $n_0, n_1, ..., n_k \in \mathbb{N}$ gilt:

$$f(n_1, ..., n_k) = n_0 \text{ genau dann,}$$
$$\text{wenn } \phi(F) = \boldsymbol{wahr} \text{ für alle } \phi \text{ mit } \phi(x_i) = n_i, 0 \leq i \leq k$$

Beispiele:

- Die Addition wird repräsentiert durch $x_0 = x_1 + x_2$
- Die Subtraktion wird repräsentiert durch $(x_2 + x_0 = x_1) \vee (x_0 = 0 \wedge \exists x_3(x_1 + x_3 + 1 = x_2))$

Es gilt nun der folgende Satz:

Lemma 11.1. *Jede WHILE-berechenbare Funktion ist arithmetisch repräsentierbar.*

Beweis: durch Induktion über den Aufbau von WHILE-Programmen. Der Beweis ist allerdings technisch etwas aufwändig...

Eine arithmetische Formel F heißt wahr (oder gültig), wenn für alle ϕ stets $\phi(F) = \boldsymbol{wahr}$ ist, z.B

$$x_1 + x_2 = x_2 + x_1$$
$$\neg\exists x_4\ ((x_1 + 1)^{x_4+3} + (x_2 + 1)^{x_4+3} = x_3^{x_4+3})$$

Mit Reduktion eines unserer Halteprobleme auf die Menge der wahren arithmetischen Formeln zeigt man:

Lemma 11.2. *Die Menge der wahren arithmetischen Formeln ist nicht rekursiv aufzählbar.*

Ein Beweissystem (B, Ψ) für eine Formelmenge A besteht aus einer Menge B von Beweisen und einer Interpretationsfunktion $\Psi : B \to A$.

Ein Beweis $b \in B$ ist eine Zeichenkette über einem Alphabet E , die nach gewissen syntaktischen Regeln und Schlussschemata aufgebaut sein muss. Eine Minimalforderung ist sicherlich, dass die Menge der Beweise B entscheidbar ist: Man sollte einer Zeichenkette ansehen können, ob sie ein Beweis ist oder nicht. Außerdem sollte man aus einem Beweis auch die dadurch bewiesene Aussage herauslesen können. Dies leistet die Interpretationsfunktion Ψ mit Definitionsbereich B, die angibt, welche Formel $y \in A$ durch den Beweis $b \in B$ bewiesen wurde. Ψ sollte sinnvollerweise eine berechenbare Funktion sein.

Mit

$$Bew(B, \Psi) = \{y \in E^* \mid \text{ es gibt } b \in B \text{ mit } \Psi(b) = y\}$$

wird dann die Gesamtheit der durch (B, Ψ) beweisbaren Formeln bezeichnet.

Der folgende Satz ist eine Version des 'Gödelschen Unvollständigkeitssatzes':

Satz 11.3 (Gödelscher Unvollständigkeitssatz)**.** *Für jedes Beweissystem (B, Ψ) für wahre arithmetische Formeln gilt:*

Die Menge der wahren arithmetischen Formeln ist eine echte Obermenge der Menge $Bew(B, \Psi)$.

Beweis:

Da die Menge B der Beweise eines Beweissystems entscheidbar ist und die Funktion Ψ, die jedem Beweis die von ihm bewiesene Formel zuordnet, berechenbar ist, ist die Menge aller in diesem Beweissystem beweisbaren Formeln rekursiv aufzählbar. Wir hatten oben aber angemerkt, dass die Menge der wahren arithmetischen Formeln nicht rekursiv aufzählbar ist. Da wir annehmen, dass das Beweissystem korrekt ist, d.h. nur wahre arithmetische Formeln beweist, bleiben demnach wahre arithmetische Formeln übrig, die nicht beweisbar sind.

12 Komplexitätsklassen und das P-NP-Problem

Im vierten Kapitel haben wir über den Begriff der Berechenbarkeit gesprochen. Dabei ging es um die prinzipielle Lösbarkeit von Berechnungsproblemen. Jetzt wollen wir unser Augenmerk auf den Berechnungsaufwand von Algorithmen richten. Für das praktische Lösen von Berechnungsproblemen ist es nämlich nicht genug, einen Algorithmus zu haben, der das Problem löst. Um von praktischem Interesse zu sein, darf dieser Algorithmus nicht zu viele Ressourcen benötigen. Die beiden wichtigsten Ressourcen, in denen man die Berechnungskomplexität misst, sind Rechenzeit und Speicherplatz. Für die praktische Lösung eines Berechnungsproblems wird man im Allgemeinen nach einem möglichst effizienten Algorithmus suchen. Hat man für ein Berechnungsproblem einen Algorithmus gefunden und seinen Bedarf an Rechenzeit und Speicherplatz abgeschätzt, so hat man zumindest schon mal eine obere Schranke für die Komplexität des Berechnungsproblems. Falls es einen noch effizienteren Algorithmus gibt, würde man den natürlich auch gerne haben. Für eine vollständige Analyse der Berechnungskomplexität eines Berechnungsproblems ist es daher auch von Interesse, eine untere Schranke für seine Berechnungskomplexität zu bestimmen. Das Bestimmen einer unteren Schranke ist oft schwierig, da diese untere Schranke ja für alle Algorithmen gelten muss, die das Problem lösen.

Wir werden uns in der folgenden kurzen Einführung in die Komplexitätstheorie auf den Zeitaufwand von Algorithmen und auf die Zeitkomplexität von Berechnungsproblemen konzentrieren. Dies sollte aber nicht darüber hinwegtäuschen, dass auch der Speicherplatzbedarf („Bandkomplexität “) ein wichtiges Kriterium für die Effizienz von Algorithmen und die Komplexität von Problemen ist. Oft gibt es bei Problemen auch einen trade-off zwischen Zeitbedarf und Speicherplatzbedarf von Algorithmen, die das Problem lösen, d.h. es gibt einerseits Algorithmen, die das Problem schnell, aber mit hohem Speicherplatzbedarf lösen und andererseits Algorithmen, die viele Rechenschritte benötigen, aber nur wenig Speicherplatz.

Wie misst man den Zeitbedarf eines Algorithmus? Für die Praxis kommt es natürlich auf den tatsächlichen Zeitbedarf, z.B. in Sekunden, an. Da dieser Zeitbedarf aber stark von der verwendeten Hardware abhängt und die Hardware sich ständig weiterentwickelt, macht eine derartige Angabe bei einer theoretischen Untersuchung wenig Sinn. Stattdessen sollte man ein theoretisches Computermodell betrachten, dass in der Praxis existierende Computer hinreichend gut modelliert, so dass Komplexitätsaussagen in diesem Modell sich auch auf in der Praxis existierende Computer übertragen lassen. Das zu diesem Zweck meistbenutzte Modell ist das Mehrband-Turingmaschinenmodell. Wir wollen uns auf Turingmaschinen beschränken, die bei jeder Eingabe nach endlich vielen Schritten aufhören zu rechnen. Berechnungsprobleme kann man meistens durch eine zu berechnende Wortfunktion beschreiben oder als Wortproblem für eine geeignete Sprache auffassen.

Definition 12.1. *Für eine deterministische Mehrband-Turingmaschine M sei*

$$time_M(w) = \text{ Schrittzahl von } M \text{ bei Eingabe von } w$$

(Schrittzahl: Zahl Übergänge von Startkonf. bis Berechnungsende)

Sei $f : \mathbb{N} \to \mathbb{N}$ eine totale Zahlenfunktion.

- *Die Klasse $FTIME(f)$ besteht aus allen totalen Wortfunktionen $g : E^* \to E^*$ (über irgendeinem Alphabet E), für die es eine deterministische Mehrband-Turingmaschine M mit Eingabealphabet E gibt, die die Funktion g berechnet und bei Eingabe eines Wortes w nach höchstens $f(|w|)$ Schritten anhält, d.h. $time_M(w) \leq f(|w|)$ für alle Eingaben w erfüllt.*
- *Die Klasse $TIME(f)$ besteht aus allen Sprachen L (über irgendeinem Alphabet E), für die es eine deterministische Mehrband-Turingmaschine M mit Eingabealphabet E gibt, die $L(M) = L$ und $time_M(w) \leq f(|w|)$ für alle Eingaben w erfüllt.*

Bemerkung 12.2. *(1) Wir messen den Zeitbedarf hier nur in Abhängigkeit von der Länge $|w|$ des jeweiligen Eingabewortes w. Man könnte den Zeitbedarf noch genauer in Abhängigkeit direkt von dem jeweiligen Eingabewort w messen. Für die meisten Untersuchungen reicht unsere Definition aber aus.*

(2) Wir verwenden Mehrbandmaschinen, da sie etwas realistischere Zeitkomplexitäten liefern als Einbandmaschinen. Bei Einbandmaschinen kann man z.B. Zwischenergebnisse nur auf einem noch nicht benutzten Teil des Bandes ablegen und muss für das Hin-und-Herlaufen des Lese-Schreibkopfes viel Zeit verwenden, während dies in der Praxis nicht der Fall ist. Es sei daran erinnert, dass man Mehrbandmaschinen durch Einbandmaschinen simulieren kann (Satz **??***). Eine Analyse des Beweises zeigt, dass jede Mehrbandmaschine, die in Zeit $O(f(n))$ arbeitet, durch eine Einbandmaschine simuliert werden kann, die in Zeit $O(f^2(n))$ arbeitet.*

(3) Die verwendeten Maschinen M müssen in allen Fällen anhalten. Bei $w \notin M$ darf es also keine Endlosschleifen geben, d.h. M muß hier in einem Nichtendzustand steckenbleiben.

(4) Oft werden Zahlenfunktionen $h : \mathbb{N} \to \mathbb{N}$ berechnet. Man betrachtet dann im Allgemeinen die korrespondierende Wortfunktion $g : E^* \to E^*$, die für jede Zahl n das Wort $bin(n)$ auf $bin(h(n))$ abbildet (und Binärwörter, die keine natürliche Zahl darstellen, z.B. auf das leere Wort abbildet), und untersucht deren Komplexität.

(5) Dieses Komplexitätsmaß heißt Bitkomplexität. Die Bezeichnung rührt daher, dass jeder einzelne Konfigurationsübergang gezählt wird. Ein Konfigurationsübergang beinhaltet nämlich eine Änderung die sich durch konstant viele Bits beschreiben lässt. Ein anderer Name dafür ist logarithmisches Kostenmaß. Der Grund für diese Bezeichnung wird beim Vergleich mit dem nachfolgend eingeführten uniformen Kostenmaß klar.

(6) Neben dieser Bitkomplexität findet man in der Literatur aber auch die so genannte uniforme Komplexität. Dabei zählt man nur die Zahl der 'elementaren' Rechenoperationen, die nötig sind, um aus n den Wert $f(n)$ zu berechnen. Die uniforme Komplexität ist aber nicht immer angemessen. Zum Beispiel kann man die Zahl 2^{2^n} berechnen, indem mit $x := 2$ startet und dann n-mal die Operation $x := x^2$ durchführt. Es reichen also n Multiplikationen. Aber allein zum Hinschreiben des Ergebnisses, der Zahl 2^{2^n}, in Binärschreibweise, braucht man 2^n Schritte. Die uniforme Komplexität ist hier unangemessen, da die Elementaroperationen hier auf sehr große Zahlen angewendet werden. In der Praxis kann man aber wie im Turingmaschinenmodell letzten Endes pro Zeiteinheit nur eine Bitoperation durchführen. Das führt dazu, dass bei der Umrechnung vom uniformen Kostenmaß in die Bitkomplexität ein Faktor proportional zur Operandengröße auftritt (Bsp.: Um zwei N-Bit-Zahlen zu addieren, benötigt man eine Anzahl von Bitoperationen, die proportional zu N ist.)

Es stellt sich die Frage, welche Komplexitätsklassen für die Praxis von Interesse sind.

Einerseits sind Algorithmen, die exponentiell viel Zeit verbrauchen, d.h. bei einer Eingabe der Länge n z.B. 2^n Zeiteinheiten benötigen, sicher uninteressant.

Zu verlangen, dass ein Algorithmus bei Eingabe der Länge n nur $O(n)$ Schritte benötigt, scheint wiederum etwas zu streng zu sein. Realistisch, zumindest bei nicht zu großen Eingaben, sind auch Algorithmen, die in Polynomzeit arbeiten, mit einem Polynom nicht zu hohen Grades. Da die Klasse aller Polynome sehr gute theoretische Eigenschaften hat, werden wir uns im Folgenden auf Probleme konzentrieren, die in Polynomzeit gelöst werden können.

Definition 12.3.

- *FP sei die Menge aller in Polynomzeit berechenbaren Wortfunktionen:*

$$FP = \bigcup_{p \text{Polynom}} FTIME(p)$$

- *P sei die Menge aller in Polynomzeit lösbaren Probleme:*

$$P = \bigcup_{p \text{Polynom}} TIME(p)$$

In Analogie kann man die Klasse der in exponentieller Zeit lösbaren Probleme definieren, indem man als Zeitschranken Funktionen der Form $2^{c \cdot n}$ zulässt.

Wir wollen den Zeitaufwand auch für nichtdeterministische Turingmaschinen definieren.

Definition 12.4. *Für eine nichtdeterministische Turingmaschine M sei*

$$ntime_M(w) = \max\{ \quad \text{Schrittzahl irgendeiner Rechnung von } M \text{ bei Eingabe von } w\}$$

- *Sei* $f : \mathbb{N} \to \mathbb{N}$ *eine totale Zahlenfunktion.*

  Die Klasse $NTIME(f)$ *besteht aus allen Sprachen* L *über irgendeinem Alphabet* E *derart, dass es eine nichtdeterministische Mehrband-Turingmaschine* M *gibt mit* $L(M) = L$ *und* $ntime_M(w) \leq f(|w|)$ *für alle Eingaben* w.

- $NP = \bigcup_{p \text{Polynom}} NTIME(p)$

Für eine nichtdeterministische Turingmaschine M gehören all diejenigen Wörter zu $L(M)$, für die es mindestens eine akzeptierende Berechnung der Maschine M gibt. Wir verlangen bei der Definition der Zeitklasse $NTIME(f)$, dass die M bei Eingabe eines Wortes w der Länge n also bei jeder möglichen Berechnung nach höchstens $f(n)$ Schritten aufhört zu rechnen, d.h. sie muss anhalten.

Das kann entweder in einem Endzustand geschehen — dann wird das Eingabewort akzeptiert.

Oder sie hält in einem Nicht-Endzustand — dann wird das Eingabewort nicht akzeptiert.

Aus der Definition der Turingmaschinen ist sofort klar, dass P in NP enthalten ist. Ob jedoch $P = NP$ oder $P \neq NP$ ist, ist ein berühmtes offenes Problem, wohl das berühmteste Problem der theoretischen Informatik. Es wird als das 'P-NP-Problem' bezeichnet. Auf die Lösung dieses Problem ist seit dem Jahr 2000 ein Preis von 1 Million US Dollar ausgesetzt, siehe `http://www.claymath.org/millennium`.

Es gibt viele Probleme, die für die Praxis wichtig sind und in NP liegen und von denen man gerne wüsste, ob sie auch in P liegen. Wir werden in dem folgenden Kapitel sehen, dass es sogar viele Probleme gibt, nämlich die NP-vollständigen, die alle in NP liegen und in dem folgenden Sinn typisch für das P-NP-Problem sind:

Könnte man schon für ein einziges von ihnen nachweisen, dass es auch in P liegt oder dass es nicht in P liegt, so hätte man damit schon das P-NP-Problem gelöst.

Da niemand trotz jahrzehntelanger Suche einen Polynomzeitalgorithmus für ein NP-vollständiges Problem gefunden hat, nehmen die meisten Wissenschaftler in der Komplexitätstheorie an, dass $P \neq NP$ ist.

Wir werden uns in den folgenden Kapiteln auf das P-NP-Problem konzentrieren.

Vorher möchten wir aber noch zeigen, dass die Klasse NP in einer deterministischen Zeitkomplexitätsklasse enthalten ist.

Satz 12.5. *Die Klasse* NP *ist enthalten in* $\bigcup_{p \text{Polynom}} TIME(2^{p(n)})$.

Beweisskizze: Sei L eine Sprache in NP. Dann gibt es ein Polynom q und eine nichtdeterministische Turingmaschine M, die L erkennt und in Zeit $q(n)$ arbeitet. M führt also bei jeder möglichen Rechnung zu einer Eingabe w der Länge n höchstens $q(n)$ Schritte durch. Alle möglichen Rechnungen, die M bei einer festen Eingabe durchführen kann, kann man sich als Berechnungsbaum aufschreiben. Die Zahl der Verzweigungen an einem Knoten ist gerade die Zahl der verschiedenen Möglichkeiten, wie die Maschine von der Konfiguration aus weiterrechnen kann. Diese Zahl kann aber durch eine Konstante c, die nur von der Überführungsfunktion von M abhängt, nach oben beschränkt werden. Also hat der Berechnungsbaum höchstens $cq(n)$ Berechnungspfade. Eine geeignete deterministische Turingmaschine kann systematisch alle möglichen Rechnungen von M mit Eingabe w durchprobieren, also alle Pfade in diesem Berechnungsbaum daraufhin überprüfen, ob es einen akzeptierenden Pfad gibt. Das geht in Zeit, die nicht wesentlich größer ist als $cq(n)$, also für eine geeignetes Polynom p in Zeit $2^{p(n)}$. Also gibt es ein Polynom p, so dass L in $TIME(2^{p(n)})$ enthalten ist.

Schließlich merken wir an, dass alle Sprachen in dieser Komplexitätsklasse und sogar in weit größeren Komplexitätsklassen nicht nur entscheidbar sind, sondern sogar LOOP-berechenbar. Dabei nennen wir eine Sprache L LOOP-berechenbar, wenn die charakteristische Funktion der korrespondierenden Teilmenge $\nu^{-1}(L)$ der natürlichen Zahlen LOOP-berechenbar ist (hier ist ν die Bijektion zwischen natürlichen Zahlen und Wörtern aus 7.1).

Satz 12.6. *Ist* $f : \mathbb{N} \to \mathbb{N}$ *eine LOOP-berechenbare Funktion, so ist jede Sprache* L *aus* $TIME(f)$ *LOOP-berechenbar.*

Beweisskizze: Sei L aus $TIME(f)$ und M eine Turingmaschine, die L erkennt und in Zeit f arbeitet. Da die Transformation zwischen einer Eingabe(-zahl) n und $\nu(n)$ nicht schwierig ist, kann man daraus eine Turingmaschine konstruieren, die die charakteristische Funktion der Menge $\nu^{-1}(L)$ in Zeit g berechnet, wobei g ebenfalls eine LOOP-berechenbare Funktion ist.

Wir hatten in Satz 3.5 gesehen, dass man eine Turingmaschine, die eine Zahlenfunktion berechnet, durch ein WHILE-Programm simulieren kann. Das WHILE-Programm konnte dabei so konstruiert werden, dass es nur eine einzige WHILE-Schleife enthielt. Außerdem zeigt eine Analyse des Beweises, dass die Zahl der Durchläufe dieser WHILE-Schleife im Wesentlichen durch die Zahl der Rechenschritte der Turingmaschine beschränkt werden kann. Daher kann man in dem WHILE-Programm die WHILE-Schleife

`WHILE` $x_z \neq 0$ `DO`

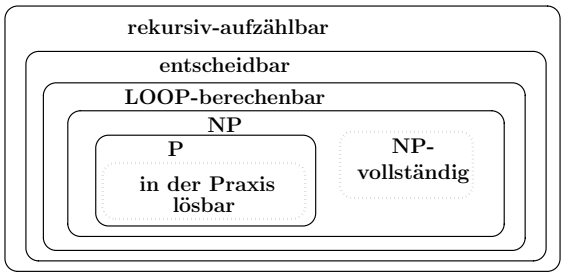
ersetzen durch

$y := g(n)$; `LOOP` y `DO`

Da die Funktion g auch LOOP-berechenbar ist, kann man aus dem WHILE-Programm daher ein äquivalentes LOOP-Programm erhalten.

Zum Beispiel sind alle Polynome p und auch alle Funktionen der Form $2^{p(n)}$ (p Polynom) LOOP-berechenbar.

Das folgende Diagramm gibt die Situation nach dem heutigen Stand der Kenntnisse und Vermutungen wieder.



13 NP-Vollständigkeit

Wir wollen nun die NP-vollständigen Sprachen einführen und untersuchen. Dazu brauchen wir eine Polynomzeit-Version des Reduzierbarkeitsbegriffs, siehe Definition 8.8

Definition 13.1. *Seien A und B Sprachen über einem Alphabet E.*

Dann heißt A auf B polynomial reduzierbar, wenn es eine in Polynomzeit berechenbare Funktion $f : E^* \to E^*$ gibt, so dass für alle $w \in E^*$ gilt:

$$w \in A \iff f(w) \in B$$

Schreibweise: $A \leq_p B$

Man beachte, dass in Polynomzeit berechenbare Funktionen automatisch total sind.

Bemerkung: Die Definition kann leicht auf den Fall verschiedener Alphabete für A und B übertragen werden.

Lemma 13.2.

- *Falls $A \leq_p B$ und $B \in P$, so ist auch $A \in P$.*
- *Falls $A \leq_p B$ und $B \in NP$, so ist auch $A \in NP$.*

Beweis: Wir nehmen an, dass B aus P sei und dass M_B eine Turingmaschine sei, die B erkennt und in Zeit q arbeitet. Dabei sei q ein Polynom. Ferner werde die Reduktion von A nach B durch eine Funktion f und eine die Funktion f in Zeit p berechnende Turingmaschine M_f vollzogen. Dabei sei p ein Polynom.

Wir schalten nun die Maschinen M_f und M_B hintereinander, d.h. auf ein Eingabewort w wird erst M_f angewendet und auf das Ergebnis $f(w)$ dann M_B. Diese kombinierte Maschine erkennt aber gerade A.

Ihr Zeitbedarf ist nach oben beschränkt durch $p(|w|) + q(|f(w)|)$. Wir können das Polynom q eventuell etwas vergrößern und monoton machen, indem wir alle negativen Koffezienten in q durch ihren jeweiligen Betrag ersetzen. Das resultierende Polynom, es sei q' genannt, hat die Eigenschaften $q(m) \leq q'(m)$ (es ist eine obere Schranke für q) und $q'(k) \leq q'(l)$ für $k \leq l$ (es ist monoton). Also ist der Zeitbedarf der kombinierten Maschine wegen $|f(w)| \leq p(|w|) + |w|$ nach oben beschränkt durch $p(|w|) + q(|f(w)|) \leq p(|w|) + q'(p(|w|) + |w|)$. Der Zeitbedarf ist also durch ein Polynom beschränkt. Der additive Term $+|w|$ wird nur benötigt, um auch den Fall einzuschließen, dass die Eingabe w gar nicht ganz gelesen wird, sondern z.B. nur nach links verlängert wird.

Der Fall B aus NP wird analog behandelt.

Lemma 13.3. *Die Relation $\leq_p$ ist transitiv, d.h. aus $A \leq_p B \leq_p C$ folgt $A \leq_p C$.*

Dies beweist man ebenso wie Lemma 13.2 durch Hintereinanderschaltung zweier Maschinen, die in Polynomzeit Reduktionsfunktionen berechnen.

Die Relation $\leq_p$ ist übrigens auch reflexiv, aber nicht symmetrisch.

Definition 13.4.

- *Eine Sprache L heißt NP-hart, wenn* alle *Sprachen aus NP auf sie polynomial reduzierbar sind.*
- *Eine Sprache L heißt NP-vollständig, wenn sie aus NP ist und NP-hart ist.*

Nach dieser Definition sind die NP-vollständigen Probleme die schwierigsten unter den Problemen in NP. Der folgende Satz besagt, dass man nur für ein einziges $NP-$vollständiges Problem einen Polynomzeitalgorithmus finden müsste, um nachzuweisen, dass P und NP gleich sind.

Satz 13.5. *L sei NP-vollständig. Dann folgt:*

$$L \in P \iff P = NP$$

Beweis: Die Richtung '$\Leftarrow$' ist trivial. Wir zeigen '$\Rightarrow$'. Wir nehmen an, L sei aus P und NP-vollständig. Da klar ist, dass P in NP enthalten ist, ist nur zu zeigen, dass NP in P enthalten ist. Sei L' aus NP beliebig. Da L NP-hart ist, lässt sich L' auf L polynomial reduzieren. Daher ist nach Lemma 13.2 auch L' aus P. Da L' beliebig war, folgt die Behauptung.

Damit der Begriff der NP-Vollständigkeit überhaupt interessant ist, muss man nachweisen, dass es tatsächlich NP-vollständige Probleme gibt.

Ein solches Problem ist das 'Erfüllbarkeitsproblem der Aussagenlogik'.

Dazu müssen wir beliebige aussagenlogische Formeln durch endliche Wörter über einem endlichen Alphabet codieren.

Die Menge der aussagenlogischen Formeln ist ähnlich definiert wie die arithmetischen Formeln aus Kapitel 146, allerdings betrachten wir nur den logischen Teil und diesen auch nur ohne Quantoren. Wir beginnen daher mit Variablen, die Wahrheitswerte (hier $\mathbf{0}$ für 'falsch' und $\mathbf{1}$ für 'wahr') annehmen können: Wir definieren entsprechend eine (kontextfreie) Sprache $\mathcal{F}$ wie folgt:

- Jede Variable x_i ist eine Formel (mit $i \in \mathbb{N}$), setze $V := \{x_i \mid i \in \mathbb{N}\}$
- Sind F, G Formeln, so auch $\neg F$, $(F \wedge G)$ und $(F \vee G)$

Formeln können ausgewertet werden, wenn den Variablen Werte zugewiesen werden:

- Eine Belegung ϕ ist eine Abbildung $\phi : V \to \{0,1\}$
- ϕ kann auf Formeln erweitert werden durch

$$\begin{aligned} \phi(\neg F) &= 1 - \phi(F) \\ \phi(\,(F \wedge G)\,) &= \min\{\phi(F), \phi(G)\} \\ \phi(\,(F \vee G)\,) &= \max\{\phi(F), \phi(G)\} \end{aligned}$$

Eine aussagenlogische Formel F mit Variablen x_i kann man durch ein Wort $code(F)$ über einem endlichen Alphabet $E = \{0,1,x,(,),\neg,\wedge,\vee\}$ codieren, indem man jede Variable x_i durch das Wort $x\ bin(i)$ codiert. Hat eine Formel die Länge m und enthält n Variablen, so hat das Wort $code(F)$ die Länge $m \log n$.

Im Folgenden werden wir in den Formeln F den Variablen auch andere (aussagekräftigere) Namen geben. Wenn wir dann von $code(F)$ sprechen, ist folgendes gemeint: Bestimme zunächst die in F vorkommenden Variablen v, ordne sie nach einer beliebigen Reihenfolge ($\pi(v)$ gebe dabei an, dass v in dieser Reihenfolge an $\pi(v)$-ter Stelle kommt), benenne jede Variable v zu $x_{\pi(v)}$ um und schreibe die enstandene Formel mit binär notierten Indizes auf. Zum Beispiel würde $\neg$(wichtige Variable $\vee$ (andere Variable $\wedge$ wichtige Variable)) notiert als $\neg(x1 \vee (x10 \wedge x1))$.

Wir werden auch auf Klammern verzichten, wo dies ohne Mißverständnisse möglich ist, und Abkürzungen verwenden, etwa $\bigvee_{1 \le k \le 4} x_k$ für $x_1 \vee x_2 \vee x_3 \vee x_4$ mit Codierung $((x1 \vee x10) \vee (x11 \vee x100))$, bzw. $x \to y$ als Umschreibung für $\neg x \vee y$.

Lemma 13.6. *Für jedes m gibt es eine Formel G der Länge $O(m^2)$ mit den Variablen $x_1, ..., x_m$, so dass G genau dann den Wahrheitswert 1 erhält, wenn* genau eine *der Variablen mit 1 belegt wird.*

Beweis: Man wähle

$$G(x_1, ..., x_m) = (x_1 \vee ... \vee x_m) \wedge \Big(\bigwedge_{1 \le j \le m-1} \ \bigwedge_{j+1 \le l \le m} \neg(x_j \wedge x_l) \Big)$$

Die erste Teilformel wird genau dann wahr, wenn *mindestens eine* Variable wahr ist. Die zweite Teilformel wird genau dann wahr, wenn *höchstens eine* Variable wahr ist. D.h., die Formel G leistet das Gewünschte. Die Länge der Formel ist $O(m^2)$.

Definition 13.7 (Erfüllbarkeitsproblem der Aussagenlogik)**.** $\boldsymbol{SAT := \{code(F)}$ *aus* $\boldsymbol{E^*}$ | $\boldsymbol{F}$ *erfüllbare Formel der Aussagenlogik*$\boldsymbol{\}}$

Die übliche (etwas weniger formale) Formulierung dieses Problems ist:

- gegeben: eine aussagenlogische Formel $\boldsymbol{F}$
- gefragt: Ist $\boldsymbol{F}$ erfüllbar, d.h. gibt es eine Belegung der Variablen in $\boldsymbol{F}$ mit $\mathbf{0}$ und $\mathbf{1}$, so dass die Formel $\boldsymbol{F}$ den Wert $\mathbf{1}$ erhält?

Satz 13.8 (Cook)**.** *Das Erfüllbarkeitsproblem* $\boldsymbol{SAT}$ *der Aussagenlogik ist* $\boldsymbol{NP}$*-vollständig.*

Beweisskizze: a) Wir zeigen zuerst, dass $\boldsymbol{SAT}$ aus $\boldsymbol{NP}$ ist. Wir müssen eine nichtdeterministische Turingmaschine beschreiben, die die Sprache $\boldsymbol{SAT}$ erkennt und in polynomialer Zeit arbeitet. Sei eine aussagenlogische Formel $\boldsymbol{F}$ in der Form $\boldsymbol{code(F)}$ gegeben. Unsere Maschine soll in zwei Phasen arbeiten. In der ersten, nichtdeterministischen Phase rät die Maschine für jede in der Formel $\boldsymbol{F}$ vorkommende Variable $\boldsymbol{x_i}$ einen Wert $\mathbf{0}$ oder $\mathbf{1}$. Das geht sicher in Polynomzeit (d.h. in Zeit $\boldsymbol{O(p(n))}$, wobei $\boldsymbol{p}$ ein Polynom und $\boldsymbol{n}$ die Länge des Wortes $\boldsymbol{code(F)}$ ist).

In der zweiten Phase, die tatsächlich sogar deterministisch ist, wertet die Maschine die Formel mit der geratenen Belegung der Variablen aus, bestimmt den Wert ($\mathbf{0}$ oder $\mathbf{1}$) der Formel und hält in einem Endzustand/in einem Nicht-Endzustand an, je nachdem, ob der Wert $\mathbf{1}$ oder $\mathbf{0}$ ist. Auch das geht sicher in Polynomzeit. Also arbeitet unsere Maschine insgesamt in Polynomzeit. Sie erkennt die Sprache $\boldsymbol{SAT}$. Denn wenn die vorgegebene Formel $\boldsymbol{F}$ erfüllbar ist, gibt es auch eine Rechnung der Maschine bei Eingabe von $\boldsymbol{code(F)}$, bei der die Maschine den Wert $\mathbf{1}$ errechnet, also das Wort akzeptiert. Wenn die vorgegebene Formel $\boldsymbol{F}$ hingegen nicht erfüllbar ist, kann die Maschine auch keine erfüllende Belegung raten, und es gibt keine akzeptierende Berechnung.

Bei Problemen aus $\boldsymbol{NP}$ kann man oft eine nichtdeterministische Maschine angeben, die das Problem in zwei Phasen löst, einer nichtdeterministischen Ratephase und einer deterministischen Prüfphase (*guess and check*).

Beweisskizze zu $\boldsymbol{SAT \in NP}$:

Eine aussagenlogische Formel $\boldsymbol{F}$ sei gegeben durch $\boldsymbol{code(F)}$

Betrachte nichtdeterministische Turingmaschine $\boldsymbol{M}$ wie folgt:

- Phase 1: $\boldsymbol{M}$ rät einen Wert $\mathbf{0}$ oder $\mathbf{1}$ für jede Variable $\boldsymbol{x_i}$ von $\boldsymbol{F}$. Aufwand: polynomial (d.h. $\leq \boldsymbol{p}(|\boldsymbol{code(F)}|)$ für ein Polynom $\boldsymbol{p}$)
- Phase 2: $\boldsymbol{M}$ bestimmt Wert $\boldsymbol{z}$ von $\boldsymbol{F}$ bei diesen Werten für die $\boldsymbol{x_i}$ Falls $\boldsymbol{z = 1}$, akzeptiert $\boldsymbol{M}$ die Eingabe $\boldsymbol{code(F)}$ Falls $\boldsymbol{z = 0}$, verwirft $\boldsymbol{M}$ die Eingabe $\boldsymbol{code(F)}$ [1ex] Aufwand: wieder polynomial in $|\boldsymbol{code(F)}|$

Offensichtlich: $\boldsymbol{M}$ arbeitet stets in Polynomzeit und

$$\boldsymbol{code(F) \in SAT \iff M} \text{ kann akzeptieren}$$

Übliche Nachweismethode für $\boldsymbol{L \in NP}$: *guess and check*

— rate nichtdeterministisch Lösungversuch (Phase 1, guess)

— überprüfe, ob Lösungsversuch korrekt (Phase 2, check)

b) Nun ist noch zu zeigen, dass $\boldsymbol{SAT}$ $\boldsymbol{NP}$-hart ist. Sei dazu $\boldsymbol{L}$ ein beliebiges Problem aus $\boldsymbol{NP}$ und $\boldsymbol{M}$ eine nichtdeterministische Turingmaschine, die die Sprache $\boldsymbol{L}$ erkennt und in Polynomzeit arbeitet. Wir hatten angemerkt, dass man Mehrbandmaschinen durch Einbandmaschinen simulieren kann und dass sich der Zeitverbrauch dabei höchstens quadriert. Da es uns sowieso nur auf Polynomzeit ankommt, können wir daher annehmen, dass $\boldsymbol{M}$ eine Einbandmaschine ist. Sei $\boldsymbol{p}$ ein Polynom und eine obere Schranke für den Zeitverbrauch der Maschine. Außerdem können wir benutzen, dass die Maschine einen einmal erreichten Endzustand nie mehr verlässt.

Wir wollen nun zeigen, dass $\boldsymbol{L}$ in Polynomzeit auf $\boldsymbol{SAT}$ reduziert werden kann.

Wir müssen also zu jeder Eingabe $y = y_1, ..., y_n$ in polynomieller Zeit eine Boolesche Formel F_y konstruieren, so dass die Eingabe y genau dann aus L ist, wenn F_y erfüllbar ist.

Die Formel wird so konstruiert, dass es genau dann eine erfüllende Belegung der Variablen mit 0 oder 1 gibt, wenn es bei der Eingabe $y_1, ..., y_n$ eine akzeptierende Berechnung der Maschine M gibt.

Dazu beschreiben wir im Wesentlichen die Funktionen der Maschine durch Klauselformen in konjunktiver Form.

Wir bezeichnen mit $A = \{a_1, ..., a_l\}$ das Arbeitsalphabet, mit $S = \{s_0, ..., s_k\}$ die Zustandsmenge und mit S_f die Endzustandsmenge.

Wir geben nun die Variablen der Formel F_y und ihre Bedeutung an:

Variable	Indizes	Bedeutung
$zust_{t,s}$	$t = 0, ..., p(n)$ $s \in S$	$zust_{t,s} = 1 \Leftrightarrow$ nach t Schritten befindet sich M im Zustand s
$pos_{t,i}$	$t = 0, ..., p(n)$ $i = -p(n), ..., p(n)$	$pos_{t,i} = 1 \Leftrightarrow$ Schreib-/Lesekopf von M befindet sich nach t Schritten auf Position i
$band_{t,i,a}$	$t = 0, ..., p(n)$ $i = -p(n), ..., p(n)$ $a \in A$	$band_{t,i,a} = 1 \Leftrightarrow$ nach t Schritten befindet sich auf Position i das Zeichen a

Hier ist zu beachten, dass die Variable i bei $pos_{t,i}$ und $band_{t,i,a}$ von $-p(n)$ bis $p(n)$ läuft. Das reicht, um den Bandinhalt der Maschine in einer innerhalb von $p(n)$ Rechenschritten erreichbaren Konfiguration zu beschreiben, da man mit dem Lese-Schreibkopf pro Rechenschritt höchstens eine Stelle nach links oder rechts wandern kann.

Nunmehr kommen wir auf die Struktur der Formel F_y zu sprechen. F_y setzt sich aus fünf durch $\wedge$ verbundene Teilformeln zusammen:

R beschreibt gewisse Randbedingungen, B_a die Anfangsbedingung, U_1, U_2 die Übergangsbedingung und B_e die Endbedingung.

Wir beschreiben jetzt die Teilformeln im Einzelnen:

$$F_y = R \wedge B_a \wedge U_1 \wedge U_2 \wedge B_e$$

In R wird ausgedrückt (mit Formel G aus Lemma 13.6), dass:

- Zu jedem Zeitpunkt t gilt $zust_{t,s} = 1$ für genau ein s.
- Zu jedem Zeitpunkt t gibt es genau eine Bandposition i, über der der Kopf steht: $pos_{t,i} = 1$.
- Zu jedem Zeitpunkt t und jeder Position i gibt es genau ein a mit $band_{t,i,a} = 1$.

Also:

$$\begin{aligned} R := \bigwedge_t \; [\; & G(zust_{t,s_0}, ..., zust_{t,s_k}) \\ & \wedge \; G(pos_{t,-p(n)}, ..., pos_{t,p(n)}) \\ & \wedge \; \bigwedge_i G(band_{t,i,a_1}, ..., band_{t,i,a_l}) \;] \end{aligned}$$

B_a beschreibt den Status der Variablen zum Startzeitpunkt $t = 0$:

$$B_a = zust_{0,s_0} \wedge pos_{0,1} \wedge \bigwedge_{1 \le j \le n} band_{0,j,y_j} \wedge \bigwedge_{-p(n) \le j \le 0} band_{0,j,\square} \wedge \bigwedge_{n+1 \le j \le p(n)} band_{0,j,\square}$$

B_e prüft nach, ob im Zeitpunkt $p(n)$ ein Endzustand erreicht wird:

$$B_e = \bigvee_{s \text{ Endzustand}} zust_{p(n),s}$$

U_2 betrifft die Felder des Bandes, über denen der Kopf nicht steht und die sich nicht ändern dürfen.

$$U_2 = \bigwedge_{t,i,a} ((\neg pos_{t,i} \wedge band_{t,i,a}) \rightarrow band_{t+1,i,a})$$

U_1 betrifft die Bandpositionen, an denen sich der Kopf befindet und beschreibt den nichtdeterministischen Übergang vom Zeitpunkt t zum Zeitpunkt $t+1$. Dabei steht b für -1 (L), 0 (N) oder 1 (R).

$$U_1 = \bigwedge_{t,s,i,a} [(zust_{t,s} \wedge pos_{t,i} \wedge band_{t,i,a}) \rightarrow \bigvee_{s',a',b \text{ mit } (s',a',b) \in \delta(s,a)} (zust_{t+1,s'} \wedge pos_{t+1,i+b} \wedge band_{t+1,i,a'})]$$

Ist dabei $\delta(s, a)$ leer (für Endzustände $s \in S_f$ oder wenn M in s steckenbleibt), so setzen wir in der obigen Formel $\delta(s, a) := (s, a, N)$, d.h. die Konfiguration bleibt dann beim Zeitpunkt $t + 1$ exakt wie beim Zeitpunkt t!

Nehmen wir an, dass $y = y_1 ... y_n$ in L ist. Dann gibt es eine nichtdeterministische Rechnung der Länge $\le p(n)$, die in einen Endzustand führt. Wenn alle Variablen der Formel F_y, die wir oben eingeführt haben, ihrer Bedeutung nach mit Wahrheitswerten belegt sind, so haben alle Teilformeln den Wert 1. Damit ist F_y erfüllbar.

Ist umgekehrt F_y durch eine gewisse Belegung ihrer Variablen erfüllt, so haben alle Teilformeln den Wert 1. Insbesondere ist R erfüllt. Daher können für jedes t alle Variablenwerte von $zust_{t,s}$, $pos_{t,i}$ und $band_{t,i,a}$ sinnvoll als Konfiguration von M interpretiert werden. Ferner erfüllt die Belegung B_a und so haben wir für $t = 0$ eine aus den Variablen abzulesende gültige Startkonfiguration des Automaten M. Aus den erfüllten Belegungen von U_1 und U_2 können wir immer gültige Übergänge von$t$ nach $t+1$ ablesen, die eine nichtdeterministische Rechnung definieren. Schließlich hat auch B_e den Wert 1 mit der Konsequenz, dass die Rechnung in einen Endzustand gelangt. Damit liegt die Eingabe in der von der M erkannten Sprache.

Nunmehr wollen wir noch die Komplexität der Berechnung der einzelnen Teilformeln notieren: Wir wollen die Anzahl der Variablenpositionen in den Teilformeln zählen. Die korrekt codierten Teilformeln sind dann nur unwesentlich länger (das hängt dann noch von der verwendeten Codierung ab). Die Zahl der Variablen in den Formeln R, U_1 und U_2 ist in $O(p^2(n))$, die Variablenzahl in B_a ist in $O(p(n))$ und die Variablenzahl in B_e ist in $O(1)$. Daher ist die Länge von R ebenso wie die Länge der Gesamtformel F_y in $O(p^3(n))$. Die Länge ihrer Codierung $code(F_y)$ ist in $O(p^3(n) \log p(n))$. Die Codierung $code(F_y)$ der Formel F_y wird nun bei der Eingabe von y hingeschrieben. Dies geht in Zeit linear in der Länge von $code(F_y)$, also in Polynomzeit bezüglich der Länge n des Eingabewortes $y = y_1, ..., y_n$.

Wir haben damit gezeigt, dass das Problem L polynomial auf SAT reduzierbar ist.

In dem folgenden Kapitel werden wir weitere NP-vollständige Probleme kennenlernen.

14 Weitere NP-vollständige Probleme

Es gibt eine Fülle von interessanten $\boldsymbol{NP}$-vollständigen Problemen. In diesem Abschnitt sollen einige von ihnen vorgestellt werden. Am Schluss betrachten wir zwei $\boldsymbol{NP}$-harte Probleme aus der Theorie der formalen Sprachen.

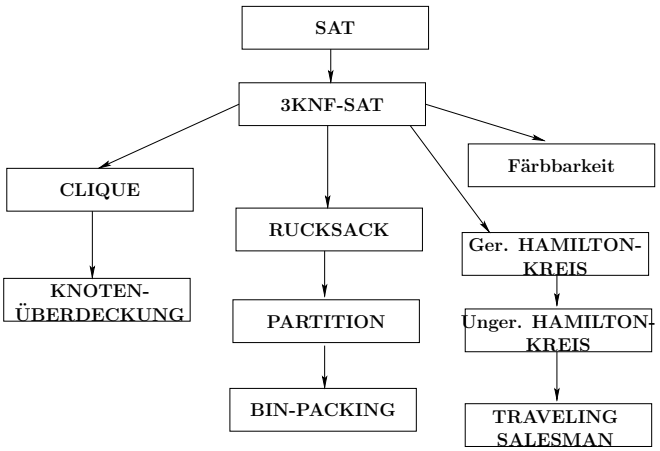
Wie kann man von einem gegebenen Problem $\boldsymbol{L}$ nachweisen, dass es tatsächlich $\boldsymbol{NP}$-vollständig ist? Man muss dazu zwei Dinge zeigen:

1. dass es in $\boldsymbol{NP}$ liegt. Dazu genügt es, eine nichtdeterministische Turingmaschine anzugeben, die das Problem in Polynomzeit löst, d.h. die die entsprechende Sprache $\boldsymbol{L}$ erkennt und in Polynomzeit arbeitet. Für die meisten Probleme geht das mit der 'Guess and Check'-Methode , d.h. mit einer Maschine, die ähnlich arbeitet wie die Maschine für das $\boldsymbol{SAT}$-Problem: in einer ersten, nichtdeterministischen Phase wird zu einer Eingabe $\boldsymbol{w}$ etwas (ein Lösungsvorschlag, ein Beweisversuch) geraten, und in einer zweiten, deterministischen Phase wird der geratene Lösungsvorschlag überprüft. Die Maschine kann natürlich nur dann eine 'Lösung' raten, wenn eine existiert, d.h. wenn $\boldsymbol{w}$ zu $\boldsymbol{L}$ gehört.

2. dass es $\boldsymbol{NP}$-hart ist. Dazu genügt es, von einem als $\boldsymbol{NP}$-hart bekannten Problem $\boldsymbol{L'}$ nachzuweisen, dass es polynomial reduzierbar auf das Problem $\boldsymbol{L}$ ist, d.h. dass $\boldsymbol{L' \leq_p L}$ gilt.

   Es ist nämlich jedes Problem $\boldsymbol{L''}$ aus $\boldsymbol{NP}$ auf $\boldsymbol{L'}$ polynomial reduzierbar, d.h. es gilt $\boldsymbol{L'' \leq L'}$. Da die polynomiale Reduzierbarkeitsrelation transitiv ist (Lemma 13.3), folgt $\boldsymbol{L'' \leq L}$, also dass $\boldsymbol{L''}$ auf $\boldsymbol{L}$ polynomial reduzierbar ist. Da $\boldsymbol{L''}$ beliebig aus $\boldsymbol{NP}$ gewählt war, ist damit auch $\boldsymbol{L}$ als $\boldsymbol{NP}$-hart erkannt.

Wir werden nun die in die in dem folgenden Diagramm aufgeführten $\boldsymbol{NP}$-vollständigen Probleme vorstellen und von einigen zeigen, dass sie tatsächlich $\boldsymbol{NP}$-vollständig sind. Die Pfeile in dem Diagramm geben mögliche Reduktionen an, mit denen man zeigen kann, dass die Probleme $\boldsymbol{NP}$-hart sind, da $\boldsymbol{SAT}$ schon als $\boldsymbol{NP}$-hart bekannt ist (Satz 13.8). Einige der Reduktionen werden wir durchführen. Die restlichen findet man in (Schöning: Theoretische Informatik — kurzgefasst).



Im Einzelnen handelt es sich um die folgenden Probleme (nach Schöning, S. 164-165):

Satz 14.1 (NP-vollständige Probleme)**.** $\boldsymbol{NP}$*-vollständig sind:*

3KNF-Sat

- *gegeben: Eine Boolesche Formel* $\boldsymbol{F}$ *in konjunktiver Normalform (Klauselform) mit höchstens* $\boldsymbol{3}$ *Literalen pro Klausel.*
- gefragt: *Ist* $\boldsymbol{F}$ *erfüllbar?*

CLIQUE

- gegeben: *Ein ungerichteter Graph* $G = (V, E)$ *und eine Zahl* $k \in \mathbb{N}$
- gefragt: *Besitzt* G *eine 'Clique' der Größe mindestens* k*?*

(Eine 'Clique' der Größe k *ist eine Menge* $V' \subseteq V$ *mit* $|V'| \geq k$ *und* $\{u, v\} \in E$ *für alle* $u, v \in V'$ *mit* $u \neq v$*.)*

KNOTENÜBERDECKUNG

- gegeben: *Ein ungerichteter Graph* $G = (V, E)$ *und eine Zahl* $k \in \mathbb{N}$
- gefragt: *Besitzt* G *eine 'überdeckende Knotenmenge' der Größe höchstens* k*?*

(Eine 'Knotenüberdeckung' der Größe k *ist eine Menge* $V' \subseteq V$ *mit* $|V'| \leq k$*, so dass für alle* $\{u, v\} \in E$ *gilt:* $u \in V'$ *oder* $v \in V'$*.)*

RUCKSACK *(oder* ***SUBSET SUM****)*

- gegeben: *Natürliche Zahlen* $a_1, a_2, ..., a_k, b \in \mathbb{N}$
- gefragt: *Gibt es eine Teilmenge* $J \subseteq \{1, 2, ..., k\}$ *mit*

$$\sum_{i \in J} a_i = b$$

PARTITION

- gegeben: *Natürliche Zahlen* $a_1, a_2, ..., a_k \in \mathbb{N}$
- gefragt: *Gibt es eine Teilmenge* $J \subseteq \{1, 2, ..., k\}$ *mit*

$$\sum_{i \in J} a_i = \sum_{i \notin J} a_i$$

BIN PACKING

- gegeben: *Eine 'Behältergröße'* $b \in \mathbb{N}$*, die Anzahl* $k \in \mathbb{N}$ *der Behälter, Objekte* $a_1, a_2, ..., a_n \in \mathbb{N}$*.*
- gefragt: *Können die Objekte so auf die* k *Behälter verteilt werden, dass kein Behälter überläuft?*

(Gibt es eine Abbildung $f : \{1, ..., n\} \to \{1, ..., k\}$*, so dass für alle* $j \in \{1, ..., k\}$ *gilt* $\sum_{f(i)=j} a_i \leq b$*?)*

GERICHTETER HAMILTON-KREIS

- gegeben: *Ein gerichteter Graph* $G = (V, E)$*.*
- gefragt: *Besitzt* G *einen Hamilton-Kreis?*

(D.h. gibt es Permutation π *der Knotenindizes* $(v_{\pi(1)}, v_{\pi(2)}, ..., v_{\pi(n)})$*, so dass stets* $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ *und* $(v_{\pi(n)}, v_{\pi(1)}) \in E$*)*

UNGERICHTETER HAMILTON-KREIS

- gegeben: *Ein ungerichteter Graph* $G = (V, E)$*.*

- gefragt: *Besitzt G einen Hamilton-Kreis?*

(D.h. gibt es Permutation π der Knotenindizes $(v_{\pi(1)}, v_{\pi(2)}, ..., v_{\pi(n)})$, so dass stets $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ und $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$)

TRAVELING SALESMAN

- gegeben: *Eine $n \times n$-Matrix $(M_{i,j})$ von 'Entfernungen' zwischen n 'Städten' und eine Zahl k.*
- gefragt: *Gibt es Permutation π (eine 'Rundreise'), so dass*

$$\sum_{i=1}^{n-1} M_{v_{\pi(i)}, v_{\pi(i+1)}} + M_{v_{\pi(n)}, v_{\pi(1)}} \leq k$$

FÄRBBARKEIT

- gegeben: *Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$*
- gefragt: *Gibt es eine Färbung der Knoten in V mit k verschiedenen Farben, so dass keine zwei benachbarten Knoten in G dieselbe Farbe haben.*

Für jedes dieser Probleme ist der Nachweis, dass es in NP liegt, leicht mit der Guess and Check-Methode zu führen. Dazu zeigt man, dass man bei jedem dieser Probleme auf eine Eingabe hin einen Vorschlag für eine Lösung in Polynomzeit raten und diesen Vorschlag in Polynomzeit verifizieren kann. In allen Fällen handelt es sich um arithmetische oder Graphenprobleme mit Summenbildungen. Daher ist dies für jedes Problem unmittelbar einsichtig. Wir beschränken uns daher jeweils auf die entsprechende Reduktion in dem Diagramm oben, um nachzuweisen, dass das jeweilige Problem NP-hart ist.

Satz 14.2. ***3KNF-SAT*** *ist NP-vollständig.*

Zur Erinnerung: ein *Literal* ist eine Variable oder eine negierte Variable. Eine *Klausel* ist eine Disjunktion von Literalen. Eine aussagenlogische Formel ist in *konjunktiver Normalform* (**KNF**), wenn sie eine Konjunktion von Klauseln ist. Eine Formel ist in **3KNF**, wenn sie eine Konjunktion von Klauseln ist, von denen jede höchstens drei Literale enthält.

Beweis: Wir müssen **SAT** auf **3KNF-SAT** polynomial reduzieren. Ausgehend von einer beliebigen Booleschen Formel F müssen wir dazu eine konjunktive Form F' mit höchstens 3 Literalen pro Klausel angeben, so dass gilt

$$F \text{ erfüllbar } \Leftrightarrow F' \text{ erfüllbar}$$

Das bedeutet nicht Äquivalenz im strengen Sinne, sondern Erfüllbarkeitsäquivalenz. Wir haben es hier also nicht mit einer allgemeinen Umrechnung in eine konjunktive Form zu tun, sondern nur mit einer Umformung, die lediglich die Erfüllbarkeit erhalten muss.

Beispiel:

$$\begin{aligned}
F = F_0 &= \neg(\neg(x_1 \vee \neg x_3) \vee x_2) \\
\text{Schritt 1: } F_1 &= ((x_1 \vee \neg x_3) \wedge \neg x_2 \\
\text{Schritt 2: } F_2 &= (y_1 \wedge \neg x_2) \\
&\quad \wedge (y_1 \Leftrightarrow (x_1 \vee \neg x_3)) \\
F_3 &= y_2 \\
&\quad \wedge [y_2 \Leftrightarrow (y_1 \wedge \neg x_2)] \\
&\quad \wedge [y_1 \Leftrightarrow (x_1 \vee \neg x_3)] \\
\text{Schritt 3: } F_4 &= y_2 \\
&\quad \wedge (\neg y_2 \vee y_1) \wedge (\neg y_2 \vee \neg x_2) \wedge (y_2 \vee \neg y_1 \vee x_2) \\
&\quad \wedge (y_1 \vee \neg x_1) \wedge (y_1 \vee x_3) \wedge (\neg y_1 \vee x_1 \vee \neg x_3)
\end{aligned}$$

Stets: F_i erfüllbar $\Leftrightarrow$ F_{i+1} erfüllbar, damit: F erfüllbar $\Leftrightarrow$ F_4 erfüllbar

Aufwand der Umformungen: Polynomial in der Länge von F

Anmerkung: Das entsprechende Problem **2KNF-SAT** mit je maximal zwei Literalen liegt bereits in P...

Satz 14.3. *Das **CLIQUE**-Problem ist NP-vollständig.*

Beweis: Zeige **3KNF-SAT** $\leq_p$ **CLIQUE**

Sei F eine Formel in **3KNF**. Da man eine Klausel mit nur einem Literal z durch die Klausel $(z \vee z \vee z)$ und eine Klausel $(y \vee z)$ mit nur zwei Literalen durch die Klausel $(y \vee z \vee z)$ ersetzen kann, können wir annehmen, dass F genau drei Literale pro Klausel hat:

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge ... \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$$

Dabei sind die z_{jk} aus $\{x_1, ..., x_n\} \cup \{\neg x_1, ..., \neg x_n\}$.

Der Formel F wird nun ein Graph auf die folgende Weise zugeordnet:

$G = (V, E)$, $V = \{(1,1), (1,2), (1,3), ..., (m,1), (m,2), (m,3)\}$, $E = \{\ \{(i,p), (j,q)\} \mid i \neq j \wedge z_{ip} \neq \neg z_{jq}\}$

Anmerkung: Die Vervielfachung der Literale ist nur zur leichteren Formulierung notwendig...)

Als Beispiele Graphen zu:

- $$(x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y)$$
- $$x \wedge y \wedge (\neg x \vee \neg y)$$

Außerdem ordnen wir der Formel F die Zahl $k := m$ zu.

Dann ist F durch eine Belegung erfüllbar

- genau dann, wenn es in jeder Klausel ein Literal gibt, das den Wert 1 erhält, z.B. $z_{1p_1}, ..., z_{mp_m}$,
- genau dann, wenn es Literale $z_{1p_1}, ..., z_{mp_m}$ gibt mit $z_{ip_i} \neq \neg z_{jp_j}$ für $i \neq j$
- genau dann, wenn es Knoten $(1, p_1), (2, p_2), ..., (m, p_m)$ in G gibt, die paarweise verbunden sind,
- genau dann, wenn es in G eine Clique der Größe $k = m$ gibt.

Satz 14.4. ***KNOTENÜBERDECKUNG** ist NP-vollständig.*

Beweis: Man kann **CLIQUE** leicht nach **KNOTENÜBERDECKUNG** reduzieren: Der Graph $G = (V, E)$ und die Zahl k werden abgebildet auf den Komplementgraphen $G' = (V, V^2 \setminus E)$ und die Zahl $|V| - k$.

Satz 14.5. ***RUCKSACK** ist NP-vollständig.*

Zum Beweis reduzieren wir **3KNF-SAT** auf **RUCKSACK**. Sei F eine Formel in **3KNF**. Wie oben können wir annehmen, dass jede Klausel in F genau drei Literale enthält:

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge ... \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$$

Dabei sind die z_{jk} aus $\{x_1, ..., x_n\} \cup \{\neg x_1, ..., \neg x_n\}$.

Wir müssen nun Zahlen $a_1, ..., a_k$ und b angeben, so dass die Summe der a_i gerade b ergibt. Die Zahl b ist gegeben durch $4...41...1$ (m-mal die Zahl 4, n-mal die 1 im Dezimalsystem).

Die Menge $\{a_1, ..., a_k\}$ der Zahlen a_i setzt sich aus vier verschiedenen Klassen von Zahlen zusammen: Zahlen $v_1, ..., v_n$, Zahlen $v'_1, ..., v'_n$, Zahlen $c_1, ..., c_m$ und Zahlen $d_1, ..., d_m$.

Die Zahlen v_k werden folgendermaßen definiert. Geschrieben als Dezimalzahl wie die Zahl b steht an der i-ten Position (von links) der Zahl v_k die Anzahl des Vorkommens der Variablen x_k in positiver Form in der Klausel i. Im hinteren Ziffernblock wird der Index k der Variablen $(1...n)$ in der Stellencodierung angezeigt: die erste Stelle wird 1 für x_1, die zweite für x_2 etc., die anderen bleiben 0. Analog werden dann Zahlen v'_k für das negative Vorkommen der Variablen x_k in der Klausel i definiert.

Die Zahlen c_j und d_j für $j = 1, .., m$ sind analog aufgebaut und haben im ersten Ziffernblock an der $j-$ten Stelle einfach eine 1 bzw. eine 2.

Beispiel:

$$F = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_4 \vee x_5) \wedge (\neg x_2 \vee \neg x_2 \vee \neg x_5)$$

mit $m = 3$, $n = 5$ und

$v_1 = 100\,10000$	$v'_1 = 010\,10000$
$v_2 = 000\,01000$	$v'_2 = 002\,01000$
$v_3 = 000\,00100$	$v'_3 = 100\,00100$
$v_4 = 010\,00010$	$v'_4 = 000\,00010$
$v_5 = 110\,00001$	$v'_5 = 001\,00001$
$c_1 = 100\,00000$	$d_1 = 200\,00000$
$c_2 = 010\,00000$	$d_2 = 020\,00000$
$c_3 = 001\,00000$	$d_3 = 002\,00000$

Nun zeigen wir folgendes: Wenn F eine erfüllende Belegung besitzt, so lässt sich eine Auswahl der Zahlen treffen, die sich zu b aufsummiert. Dabei nehme man v_k bzw. v'_k in die Auswahl auf, falls $x_k = 1$ bzw. $x_k = 0$ in der Belegung gilt. Dazu beachte man, dass zu jedem Index nur v_k oder v'_k, aber nicht beide auftreten können. Daher ergibt die Summe der letzten n Ziffern immer genau $1..1$. Schauen wir nun die ersten m Ziffern an. Sie spiegeln das Vorkommen von x_k bzw. $\neg x_k$ in den m Klauseln wider. In den einzelnen Ziffern summieren sie sich zu 1, 2 oder 3 auf. Ergänzt man dies in geeigneter Weise mit c_j und/oder d_j, so ergibt sich in jeder Ziffer gerade 4.

Sei nun umgekehrt eine Auswahl von Zahlen getroffen derart, dass die Summe gerade b ergibt. Dann muss diese Auswahl für jeden Index k aus $\{1, \dots n\}$ entweder v_k oder v'_k enthalten. Da man durch Summation von c_j und d_j allein auf höchstens den Wert 3 an der Stelle j (von links) kommt, muss auch für jeden Index j aus $\{1, \dots, m\}$ (also jeden Index einer Klausel) mindestens eine Zahl v_k oder v'_k ausgewählt sein, so dass die Summe von c_j und d_j in der Ziffer j zu 4 ergänzt wird. Das zugehörige Literal muss dann in der Klausel auftreten. Also erhält man aus der Auswahl v_k oder v'_k eine erfüllende Belegung der Formel F.

Sei andererseits

$$\sum_{a \in A} a = b = \underbrace{4...4}_{m}\underbrace{1...1}_{n}$$

Mit $c_1 + ... + c_m + d_1 + ... + d_m = \underbrace{3...3}_{m}\underbrace{0...0}_{n}$ gilt

- entweder v_k in A oder v'_k in A für $1 \leq k \leq n$
- v_k, v'_k nicht beide gleichzeitig in A!
- Summe der v_k, v'_k in A: $z_1...z_m \underbrace{1...1}_{n}$ mit $z_i \geq 0$

Also: F erfüllt durch Belegung mit

$$x_k = 1 \Leftrightarrow v_k \in A$$

Satz 14.6. ***PARTITION*** *ist* $\mathbf{NP}$*-vollständig.*

Es gibt eine elementare Reduktion von **RUCKSACK** auf **PARTITION**. Sei $(a_1, ..., a_k, b)$ ein Rucksackproblem und $M = a_1 + ... + a_k$, dann bilden wir ab:

$$(a_1, ..., a_k, b) \mapsto (a_1, ..., a_k, M - b + 1, b + 1)$$

Ist die Indexmenge I aus $\{1, ..., k\}$ eine Lösung des Problems **RUCKSACK**, so ist $I \cup \{k + 1\}$ eine Lösung von **PARTITION**, denn es gilt

$$\sum_{I} a_i + M - b + 1 = b + M - b + 1 = M - b + b + 1 = \sum_{CI} a_i + b + 1$$

Dabei bezeichnet CI das Komplement von I.

Gibt es andererseits eine Lösung J von **PARTITION**, so können die beiden neuen Indizes $k + 1$ und $k + 2$ nicht beide in J oder beide im Komplement von J liegen, da die Summe der Zahlen mit Indizes in dieser Menge dann automatisch mindestens $M - b + 1 + b + 1 = M + 2$ wäre, also größer wäre als die Summe der Zahlen mit Indizes in der anderen Menge. Diejenige unter den beiden Indexmengen J und 'Komplement von J', die den Index $k + 1$ enthält, ist ohne diesen Index eine Lösung von **RUCKSACK**.

Satz 14.7. ***BIN-PACKING*** *ist* $\mathbf{NP}$*-vollständig.*

Es gibt eine sehr einfache Reduktion von **PARTITION** auf **BIN-PACKING**: $(a_1, ..., a_k)$ sei dabei die Eingabe für **PARTITION**.

Falls $\sum_{1 \leq i \leq k} a_i$ ungerade ist, kann es keine Lösung geben, dann wählen wir eine eine Eingabe für **BIN-PACKING**, die keine Lösung hat.

Ansonsten setzen wir die Behältergröße mit $b = \sum_{1 \leq i \leq k} a_i / 2$ an und wählen 2 Behälter und die Zahlen $(a_1, ..., a_k)$.

Für die verbleibenden Reduktionen in dem Diagramm oben, mit denen man die folgenden vier Sätze beweist, sei auf Schöning verwiesen.

Satz 14.8. *Das* ***GERICHTETE HAMILTON-KREIS*** *Problem* $\mathbf{GHK}$ *ist* $\mathbf{NP}$*-vollständig.*

Satz 14.9. *Das* ***UNGERICHTETE HAMILTON-KREIS*** *Problem* $\mathbf{UHK}$ *ist* $\mathbf{NP}$*-vollständig.*

Satz 14.10. *Das* ***TRAVELING SALESMAN****-Problem* $\mathbf{TSP}$ *ist* $\mathbf{NP}$*-vollständig.*

Satz 14.11. *Das* ***Färbbarkeits****problem ist* $\mathbf{NP}$*-vollständig.*

Zum Abschluss wollen wir noch einige Probleme untersuchen, die mit formalen Sprachen zu tun haben.

Satz 14.12. *Das Wortproblem für monotone Grammatiken ist* $\mathbf{NP}$*-hart.*

Es sei angemerkt, dass nich bekannt ist, ob das Wortproblem für monotone Grammatiken in **NP** liegt. Tatsächlich ist das Problem sogar **PSPACE**-vollständig. Das heißt, erstens ist es mit polynomialem Speicherplatz lösbar, und zweitens können alle mit polynomialem Speicherplatz lösbaren Problemen auf dieses Problem polynomial reduziert werden. Man kann leicht sehen, dass **NP** in **PSPACE** enthalten ist. Ob die Umkehrung gilt, ist unbekannt. Es wird vermutet, dass **NP** eine echte Teilmenge von **PSPACE** ist.

Satz 14.13. *Das Problem* ***Regulär-Inäquivalenz****, für zwei reguläre Ausdrücke festzustellen, ob sie inäquivalent sind, d.h. ob die zugehörigen Sprachen nicht identisch sind, ist* $\mathbf{NP}$*-hart.*

Wir zeigen, dass **3KNF-SAT** auf dieses Problem polynomial reduzierbar ist. Sei $F = K_1 \wedge ... \wedge K_m$ eine Formel in konjunktiver Form. In ihr mögen die Variablen $x_1, x_2, ..., x_n$ vorkommen. Wir konstruieren zwei reguläre Ausdrücke α und β über dem Alphabet $\{0, 1\}$. Sei $\alpha = (\alpha_1|\alpha_2|...|\alpha_m)$, wobei $\alpha_i = \gamma_{i,1}...\gamma_{i,n}$ ist mit

$$\gamma_{i,j} = \begin{cases} 0 & \text{falls } x_j \text{ in } K_i \text{ vorkommt} \\ 1 & \text{falls } \neg x_j \text{ in } K_i \text{ vorkommt} \\ (0|1) & \text{sonst} \end{cases}$$

Nun bemerkt man leicht Folgendes:

- Eine Variablenbelegung $a_1, ..., a_n$ erfüllt die Klausel K_i genau dann nicht, wenn $a_1...a_n$ aus $L(\alpha_i)$.
- Eine Belegung $a_1, ..., a_n$ erfüllt die Formel F genau dann nicht, wenn sie eine der Klauseln nicht erfüllt, also genau dann, wenn $a_1...a_n$ aus $L(\alpha)$.

Also gibt es eine Belegung, die F erfüllt genau dann, wenn $L(\alpha)$ nicht gleich $\{0, 1\}^n$ ist.

Daher setzen wir $\beta = (0|1)(0|1)...(0|1)$ (n-mal), also $L(\beta) = \{0, 1\}^n$.

Wir erhalten: F ist erfüllbar genau dann, wenn $L(\alpha) \neq L(\beta)$. Die Abbildung $F \to (\alpha, \beta)$ ist also eine passende Reduktionsabbildung von **3KNF-SAT** auf das Inäquivalenzproblem.

Wir hatten gesehen, das man zu einem regulären Ausdruck effektiv einen NEA konstruieren kann. Das geht sogar in polynomialer Zeit. Die Umformung in DEA's ist aber hart, so dass die Lösung des Äquivalenzproblems für DEA's in Polynomialzeit nichts hilft.

$G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ seien ungerichtete Graphen.

Eine Bijektion p von V_1 nach V_2 heißt Isomorphismus zwischen G_1 und G_2, falls gilt: $(v, w) \in E_1 \Leftrightarrow (p(v), p(w)) \in E_2$.

Satz 14.14 (Isomorphie von Graphen)**.** *Betrachte folgende graphentheoretische Probleme:*

GRAPH-ISOMORPHIE

- gegeben: *Ungerichtete Graphen* $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$
- gefragt: *Sind* G_1 *und* G_2 *isomorph?*

SUBGRAPH-ISOMORPHIE

- gegeben: *Ungerichtete Graphen* $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$
- gefragt: *Ist* G_2 *zu einem Subgraphen von* G_1 *isomorph, d.h. gibt es* $V_1' \subseteq V_1$*, so dass* G_2 *und* $(V_1', E_1 \cap V_1' \times V_1')$ *isomorph sind?*

Dabei gilt:

- ***SUBGRAPH-ISOMORPHIE*** *ist NP-vollständig.*
- ***GRAPH-ISOMORPHIE*** $\in$ *NP, aber Vollständigkeit unbekannt!*
- Sowohl **SUBGRAPH-ISOMORPHIE** als auch **GRAPH-ISOMORPHIE** liegen offensichtlich in NP.
- **CLIQUE**$\leq_p$ **SUBGRAPH-ISOMORPHIE** über die Reduktion

  $$(G, k) \mapsto (G, C_k)$$

  wobei C_k der vollständige Graph mit k Knoten ist (d.h. die Clique der Größe k).

- Vermutung:

  GRAPH-ISOMORPHIE weder NP-vollständig noch in P.

Daher immerhin:

- Ist **GRAPH-ISOMORPHIE** reduzierbar auf ein Problem A, so ist A wahrscheinlich nicht in P...

Wichtige Teilklasse von NP: Constraint Satisfaction Probleme (CSP).
Aufbau der Probleme:

- n Variablen $x_1, ..., x_n$ für Werte aus endlichem Grundbereich D; Lösungsraum ist die Menge D^n.
- m Constraints $C_1, ..., C_m$, d.h. 0-1-wertige Funktionen auf D^n.
  C_j ist "erfüllt" für ein $(a_1, ..., a_n) \in D^n$, wenn $C_j(a_1, ..., a_n) = 1$
- Wenn jedes C_j nur von maximal k Variablen abhängt, hat das Problem die Ordnung k.

Angabe der Constraints geeignet codiert (Formeln, Graphen, o.ä..)
dabei: Überprüfung eines einzelnen Constraints sehr schnell möglich
Aufgabe: Gibt es eine Wertebelegung $(a_1, ..., a_n) \in D^n$ für die Variablen, so dass alle Constraints erfüllt sind?
Beispiele:

- **3-KNF-SAT** ist ein CSP:
  - mit $D = \{0, 1\}$
  - mit Klauseln als Constraints
  - der Ordnung 3
- allgemein: k-KNF-SAT bei maximal k Variablen pro Klausel als CSP mit $|D| = 2$ und Ordnung k
- **k-Färbbarkeit** ist CSP:
  - $D = \{1, ..., k\}$ Menge der Farben
  - Constraints = Kanten, $C_{(u,v)}$ ist erfüllt, wenn u und v verschieden gefärbt sind (also Ordnung $= 2$)

Generell: CSPs sind NP-vollständig, wenn

- $|D| \geq 2$ und Ordnung ≥ 3 oder
- $|D| \geq 3$ und Ordnung ≥ 2

Sonderfall:$|D| = 2$ und Ordnung $= 2$, z.B. **2-KNF-SAT**

Satz 14.15 (2-KNF-SAT)**.** *Das folgende Problem liegt in P:*

2KNF-Sat

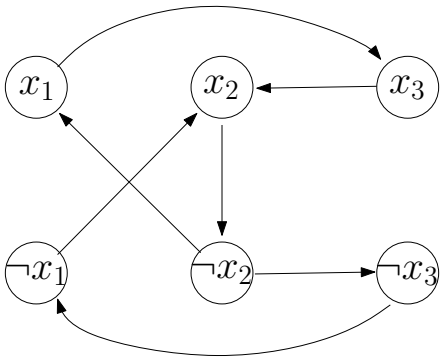
- *gegeben: Eine Boolesche Formel F in konjunktiver Normalform mit höchstens 2 Literalen pro Klausel.*
- gefragt: *Ist F erfüllbar?*

Beweisidee:

- Gegeben 2-KNF-Formel F mit Variablen $x_1, x_2, ..., x_n$.
- O.B.d.A: zwei Literale pro Klausel (statt (a) verwende $(a \vee a)$)
- Klausel $(a \vee b)$ entspricht Implikation $(\neg a \to b)$ und $(\neg b \to a)$
- Betrachte gerichteten Graphen $G_F = (V, E)$ mit Knotenmenge $V = \{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$
- Zu E: Für Klausel $(a \vee b)$ verwende Kanten $(\neg a, b)$ und $(\neg b, a)$
- Damit: Klausel mit einem Literal (a) ergibt Einzelkante $(\neg a, a)$

Beispiel: Graph G_F zur folgenden Formel F:

$$F = \underbrace{(x_1 \vee x_2)}_{1} \wedge \underbrace{(\neg x_1 \vee x_3)}_{2} \wedge \underbrace{(x_2 \vee \neg x_3)}_{3} \wedge \underbrace{(\neg x_2)}_{4}$$



F ist nicht erfüllbar:

in Formel F: x_2 falsch $\overset{1}{\rightsquigarrow}$ x_1 wahr $\overset{2}{\rightsquigarrow}$ x_3 wahr $\overset{3}{\rightsquigarrow}$ x_2 wahr $\overset{4}{\rightsquigarrow}$ x_2 falsch
im Graphen: $\neg x_2 \longrightarrow x_1 \longrightarrow x_3 \longrightarrow x_2 \longrightarrow \neg x_2$

Also: Nutze Erreichbarkeit im Graphen G_F!

Betrachte transitive Hülle $G_F^* = (V, E^*)$ des Graphen G_F, d.h. $(a, b) \in E^* \Leftrightarrow$ es gibt in E Pfad von a nach b.

E^* kann in polynomialer Zeit berechnet werden (z.B. mit Warshall-Algorithmus, kubische Komplexität in n)

Lemma 14.16. *Eine 2-KNF-formel F ist genau dann erfüllbar, wenn für kein i in G_F ein Kreis der Form $x_i \to \cdots \to \neg x_i \to \cdots \to x_i$ existiert, d.h. wenn in G_F^* nie beide Kanten $(x_i, \neg x_i)$ und $(\neg x_i, x_i)$ existieren.*

Beweis von '$\Longrightarrow$': $(a \vee b)$ entspricht Kanten $(\neg a, b), (\neg b, a) \in E$

Bei einer erfüllenden Belegung von F gilt damit

1. Ist Literal a wahr, sind alle Literale b wahr mit $(a, b) \in E^*$.
2. Ist Literal b falsch, sind alle Literale a falsch mit $(a, b) \in E^*$.
3. Ist $(\neg a, a) \in E^*$ für Literal a, so muss a wahr sein.

Damit direkt "F erfüllbar $\Rightarrow \neg\exists$ Kreis". '$\Longleftarrow$': Für die Rückrichtung definiere 'Belegung' (mit $\{0, 1\}$) wie folgt:

(1) Für alle Literale mit $(\neg a, a) \in E^*$ setze $a \mapsto 1$ (damit $\neg a \mapsto 0$).

(2) dann ergänze für diese a:

- Setze alle b mit $(a, b) \in E^*$ auf $\mathbf{1}$.
- Setze alle b mit $(b, \neg a) \in E^*$ auf $\mathbf{0}$.

(3) Solange noch nicht alle Literale einen Wahrheitswert haben:

- Wähle beliebiges(!) noch nicht gesetztes Literal a, setze es auf $\mathbf{1}$.
- Setze wieder alle b mit $(a, b) \in E^*$ auf $\mathbf{1}$.
- Setze wieder alle b mit $(b, \neg a) \in E^*$ auf $\mathbf{0}$.

Gibt es keine Kanten $(x_i, \neg x_i)$ und $(\neg x_i, x_i)$ in E^*, so ist die Belegung wohldefiniert und erfüllt F: Nie folgt auf $\mathbf{1}$ im Graphen G eine $\mathbf{0}$, d.h. alle Klauseln sind erfüllt.

Anmerkungen zu den NP-vollständigen Problemen: Viele von ihnen sind in der Praxis von großer Bedeutung. Obwohl sie NP-vollständig sind und ein effizienter Algorithmus zu ihrer Lösung daher nach dem heutigen Stand der Dinge nicht bekannt ist, muss man diese Probleme doch in irgendeiner Form lösen. Wie kann man vorgehen? Einerseits gibt es manchmal deterministische Algorithmen, die im Mittel polynomiale Laufzeit haben, z.B. für das Hamiltonkreisproblem. Allerdings bezieht sich dieser Mittelwert dann auf eine angenommene Verteilung der möglichen Eingaben, bei der man dann sorgfältig überprüfen muss, ob sie realistisch ist. Andererseits geht es bei einigen Problemen, z.B. dem Traveling Salesman Problem, in der Praxis gar nicht darum, zu prüfen, ob es eine Reiseroute gibt, deren Kosten eine gewisse Grenze nicht überschreiten, sondern darum, eine möglichst günstige Reiseroute zu finden. Wenn man sich mit einer Reiseroute zufrieden gibt, die nicht optimal sein muss, sondern z.B. bis zu 50% teurer als eine optimale Reiseroute sein darf, so gibt es unter gewissen weiteren Voraussetzungen einen deterministischen polynomialen Algorithmus, der so eine Route liefert. Allerdings gibt es auch Probleme, bei denen selbst eine derartige scheinbar leichtere Version noch NP-vollständig ist.