Turingmaschinen und Komplexität

Teil der Vorlesung
Automatisierung und Komplexität des logischen Schließens

Elmar Eder
Universität Salzburg

Wintersemester 2020/2021

Einleitung

In der Theoretischen Informatik sind drei Fragenkomplexe, die Algorithmen oder Computerprogramme betreffen, von besonderem Interesse.

Definition 1 Unter einem *Algorithmus* versteht man eine formalisierbare Rechenvorschrift.

Ein Algorithmus kann zu gegebenen Eingaben (englisch 'input') eine Ausgabe (englisch 'output') produzieren.

Beispiel 1
für einen Algorithmus. Der *Euklidische Algorithmus* ist ein Algorithmus, der zu zwei eingegebenen natürlichen Zahlen (Eingaben) den größten gemeinsamen Teiler (ggT) dieser beiden Zahlen als Ausgabe liefert.

Definition 2 Ein *Computerprogramm* ist ein in computerlesbarer Form formalisierter Algorithmus.

Fragenkomplexe, die dabei interessieren, sind unter anderen:

1. Was kann ein Algorithmus leisten?
   Anders ausgedrückt: Welche Probleme sind prinzipiell algorithmisch lösbar?
   Diese Fragen werden in der **Berechenbarkeitstheorie** behandelt. Wir beschäftigen uns hier mit Turingmaschinen.

2. Mit welchem Aufwand (Kosten) kann er das leisten?
   Hiermit ist der Verbrauch an Resourcen wie Rechenzeit oder Speicherplatz gemeint.
   Diese Fragen werden bilden das Thema der **Komplexitätstheorie**.

3. Wie beschreibt man das, was ein Computerprogramm leistet?
   Dies ist die Frage nach der Bedeutung eines Computerprogramms.
   Mit dieser Frage beschäftigt sich die **Formale Semantik**.

0.1 Einige Grundbegriffe

Unter den *natürlichen Zahlen* verstehen wir die nichtnegativen ganzen Zahlen, also die Zahlen $0, 1, 2, 3, \ldots$. Die Menge der natürlichen Zahlen bezeichnet man mit $\mathbb{N}$. Es ist also $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. Mit $\mathbb{N}^+$ bezeichnen wir die Menge der positiven natürlichen

Zahlen, also $\mathbb{N}^+ = \{1, 2, 3, \ldots, \}$. Wenn A eine Menge ist, so ist die *Potenzmenge* $\mathfrak{P}(A)$ definiert als die Menge aller Teilmengen von A, also

$$\mathfrak{P}(A) = \{X \mid X \subseteq A\}.$$

Wenn X und Y Mengen sind, dann bedeutet $f\colon X \to Y$, daß f eine Funktion von X nach Y ist. Der Definitionsbereich X von f wird mit $\mathrm{Def}(f)$ bezeichnet und das Ziel Y von f wird mit $\mathrm{Ziel}(f)$ bezeichnet. Es gilt also $f\colon \mathrm{Def}(f) \to \mathrm{Ziel}(f)$. Eine Funktion wird manchmal auch bezeichnet durch Angabe eines allgemeinen Argumentwertes und des zugehörigen Funktionswertes mit dem Symbol $\mapsto$ dazwischen. So kann man eine Funktion f auch schreiben als $x \mapsto f(x)$, genauer: $x \mapsto f(x)\colon \mathrm{Def}(f) \to \mathrm{Ziel}(f)$. Wenn $f\colon X \to Y$ eine Funktion ist und $A \subseteq X$ ist, dann ist die *Einschränkung* $f|_A$ der Funktion f auf die Menge A definiert durch $f|_A\colon A \to Y$ und $f|_A(x) = f(x)$ für $x \in A$.

0.1.1 Wörter und Sprachen

Definition 3

1. Unter einem *Anfangsabschnitt* von $\mathbb{N}^+$ versteht man eine endliche Teilmenge $\mathcal{A} \subseteq \mathbb{N}^+$ so, daß gilt
$$\bigwedge_{y \in \mathcal{A}} \bigwedge_{x \in \mathbb{N}^+} (x < y \implies x \in \mathcal{A}).$$
2. Für $n \in \mathbb{N}$ wird $\mathcal{A}_n$ definiert als
$$\mathcal{A}_n := \{x \in \mathbb{N}^+ \mid x \leq n\}$$

Beispiel 2

$$\begin{aligned}\mathcal{A}_0 &= \emptyset\\ \mathcal{A}_1 &= \{1\}\\ \mathcal{A}_2 &= \{1, 2\}\\ \mathcal{A}_3 &= \{1, 2, 3\}\end{aligned}$$

Alle diese Mengen sind Anfangsabschnitte von $\mathbb{N}^+$.

Es gilt: Die Anfangsabschnitte von $\mathbb{N}^+$ sind genau die Mengen $\mathcal{A}_n$ mit $n \in \mathbb{N}$.

Definition 4

1. Ein *Alphabet* ist eine endliche Menge Σ von Zeichen.
2. Ein *Wort* (Zeichenreihe, Zeichenkette, englisch string, word) über einem Alphabet Σ ist eine Funktion $w\colon \mathcal{A} \to \Sigma$, wobei $\mathcal{A}$ ein Anfangsabschnitt von $\mathbb{N}^+$ ist.

Beispiel 3
$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$. Das Wort $w\colon \mathcal{A}_5 \to \Sigma$ sei definiert durch

$$\begin{aligned} w(1) &= \mathtt{g} \\ w(2) &= \mathtt{e} \\ w(3) &= \mathtt{h} \\ w(4) &= \mathtt{e} \\ w(5) &= \mathtt{n} \end{aligned}$$

Definition 5 Sei Σ ein Alphabet. Dann bezeichnet man mit Σ^* die Menge aller Wörter über Σ:

$$\Sigma^* = \{w \mid w \text{ ist ein Wort über } \Sigma\}.$$

Notation 1 Wenn Σ ein Alphabet ist und $w\colon \mathcal{A}_n \to \Sigma$ ein Wort über Σ ist, so schreibt man w auch als $w(1)w(2)\ldots w(n)$. Im obigen Beispiel kann man das Wort w also auch schreiben als `gehen`.

Das *leere Wort* ε ist durch $\varepsilon\colon \emptyset \to \Sigma$ definiert.

Frage: Wozu betrachtet man Alphabete und Wörter (Zeichenreihen) über einem Alphabet?

1. Allgemeinheit, d.h. jede Information, die sich formalisieren und einem Computer mitteilen läßt, ist als Zeichenreihe darstellbar.
2. Mathematische Einfachheit.

Operationen für Wörter

1. *Einbettung* $\iota\colon \Sigma \to \Sigma^*$ mit $\iota(\sigma)\colon \mathcal{A}_1 \to \Sigma$ und $\iota(\sigma)(1) = \sigma$ für $\sigma \in \Sigma$.
2. *Konkatenation* $\cdot\colon \Sigma^* \times \Sigma^* \to \Sigma^*$. Sei $u\colon \mathcal{A}_m \to \Sigma$ und $v\colon \mathcal{A}_n \to \Sigma$. Dann ist $u \cdot v\colon \mathcal{A}_{m+n} \to \Sigma$ definiert durch

   $$(u \cdot v)(k) = \begin{cases} u(k) & \text{für } k \leq m \\ v(k-m) & \text{für } m < k \leq m+n. \end{cases}$$

   Zum Beispiel gilt $\mathtt{ge} \cdot \mathtt{hen} = \mathtt{gehen}$.
   Anstatt $u \cdot v$ schreibt man auch einfach uv.
3. Länge $|\cdot|\colon \Sigma^* \to \mathbb{N}$. Wenn $u\colon \mathcal{A}_m \to \Sigma$, dann ist $|u| = m$.
4. n-tes Zeichen $\pi(n, w)$ von w mit
   $\pi\colon \{(n, w) \mid n \in \mathbb{N}^+ \wedge w \in \Sigma^* \wedge n \leq |w|\} \to \Sigma$ und $\pi(n, w) = w(n)$.

5. n-te Potenz $\mathrm{pot}(n, w) = w^n$ von w mit $\mathrm{pot}\colon \mathbb{N} \times \Sigma^* \to \Sigma^*$. Definition durch vollständige Induktion nach n (siehe nächster Abschnitt):
   Wenn $w \in \Sigma^*$ ist, so ist

$$\begin{aligned} w^0 &= \varepsilon \\ w^{n+1} &= w^n w \qquad \text{für } n \in \mathbb{N}. \end{aligned}$$

6. Spiegelung $\cdot^R\colon \Sigma^* \to \Sigma^*$. Sei $w\colon \mathcal{A}_n \to \Sigma$. Dann ist $w^R\colon \mathcal{A}_n \to \Sigma$ definiert durch $w^R(k) = w(n-k+1)$. Zum Beispiel gilt
   $\texttt{gehen}^R(1) = \texttt{gehen}(5-1+1) = \texttt{gehen}(5) = \texttt{n}$
   $\texttt{gehen}^R(2) = \texttt{gehen}(4) = \texttt{e}$
   ...
   $\texttt{gehen}^R = \texttt{neheg}$

Definition 6 Unter einer *Sprache* (englisch *language*) L über einem Alphabet Σ versteht man eine beliebige Teilmenge von Σ^*.

Beispiel 4
$\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{Ä}, \texttt{Ö}, \texttt{Ü}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}, \texttt{ä}, \texttt{ö}, \texttt{ü}, \texttt{ß}\}$ mit
$L =$ Menge der Wörter der deutschen Sprache.
Weitere Beispiele für Sprachen sind

1. $\Sigma = \{\texttt{a}\}$
   $L = \{w \mid w \in \Sigma^* \wedge |w| \text{ ist gerade}\} = \{\varepsilon, \texttt{aa}, \texttt{aaaa}, \ldots\}$

2. $\Sigma = \{\texttt{a}, \texttt{b}\}$
   $L = \{w \mid w \in \Sigma^* \wedge w \text{ enthält gleich viele } \texttt{a} \text{ und } \texttt{b}\} =$
   $\{\varepsilon, \texttt{ab}, \texttt{ba}, \texttt{aabb}, \texttt{abab}, \ldots\}$

3. $\Sigma = \{\texttt{a}, \texttt{b}\}$
   $L = \{w \mid w \in \Sigma^* \wedge \neg \bigvee_{u,v \in \Sigma^*} w = u\texttt{aa}v\} =$
   $\{\varepsilon, \texttt{a}, \texttt{b}, \texttt{ab}, \texttt{ba}, \texttt{bb}, \ldots\}$

4. $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}\}$
   $L =$ Menge der arithmetischen Ausdrücke über `0` und `1`.
   Hier gilt zum Beispiel $\texttt{(1+0+(1+1)*1)*1} \in L$, aber
   $\texttt{)(1+1)} \notin L$ und $\texttt{10} \notin L$.

Definition 7 (Charakteristische Funktion)

- Sei n eine natürliche Zahl und $A \subseteq \mathbb{N}^n$. Dann ist die *charakteristische Funktion* $\chi_A\colon \mathbb{N}^n \to \mathbb{N}$ definiert durch

$$\chi_A(x_1, \ldots, x_n) := \begin{cases} 1, & \text{falls } (x_1, \ldots, x_n) \in A \text{ ist} \\ 0, & \text{falls } (x_1, \ldots, x_n) \in \mathbb{N}^n \setminus A \text{ ist.} \end{cases}$$

- Sei Δ ein Alphabet, n eine natürliche Zahl und $A \subseteq (\Delta^*)^n$. Dann ist die *charakteristische Funktion* $\chi_A \colon (\Delta^*)^n \to \mathbb{N}$ definiert durch

$$\chi_A(x_1, \ldots, x_n) := \begin{cases} 1, & \text{falls } (x_1, \ldots, x_n) \in A \text{ ist} \\ 0, & \text{falls } (x_1, \ldots, x_n) \in (\Delta^*)^n \setminus A \text{ ist.} \end{cases}$$

0.1.2 Induktion und Rekursion

Das Prinzip der vollständigen Induktion

Definition 8 Unter einem *Prädikat* auf einer Menge A versteht man eine Teilmenge $P \subseteq A$ von A. Wenn $x \in P$ ist, dann schreibt man auch $P(x)$ und man sagt, es *gelte* $P(x)$ oder, das Prädikat P *treffe auf* x *zu*.

Beispiel 5
Sei P die Menge der Primzahlen. Dann ist P ein Prädikat auf $\mathbb{N}$. Das Prädikat P trifft auf jede Primzahl zu und trifft auf alle natürlichen Zahlen, die nicht Primzahlen sind, nicht zu. Man sagt auch, P sei das Prädikat '... ist Primzahl'.

Ein Prädikat P wird dadurch eindeutig definiert, daß man für jede natürliche Zahl n angibt, ob $P(n)$ gilt oder nicht. Im Beispiel könnte man das Prädikat P auch folgendermaßen definieren.

$$P(n) : \Longleftrightarrow \; n \text{ ist Primzahl}$$

Prinzip 1 (Prinzip der vollständigen Induktion)
Sei P ein Prädikat auf $\mathbb{N}$ mit den folgenden Eigenschaften

INDUKTIONSANFANG $\quad P(0)$

INDUKTIONSSCHRITT $\quad \bigwedge_{n \in \mathbb{N}} \big(P(n) \implies P(n+1)\big).$

Dann gilt $\quad \bigwedge_{n \in \mathbb{N}} P(n).$

Im Induktionsschritt nennt man die Aussage $P(n)$ die *Induktionsvoraussetzung* und die Aussage $P(n+1)$ die *Induktionsbehauptung*.

Definition 9 Unter einem *Beweis* von $P(n)$ *durch vollständige Induktion* nach n versteht man einen Beweis von $P(n)$, der folgendermaßen abläuft.

Beweis von $P(n)$ durch Induktion nach n

INDUKTIONSANFANG: $P(0)$.

Beweis: *** Hier folgt der Beweis von $P(0)$ ***

INDUKTIONSVORAUSSETZUNG: $P(n)$

INDUKTIONSBEHAUPTUNG: $P(n+1)$

Beweis: *** Hier folgt der Beweis von $P(n+1)$ unter der Voraussetzung $P(n)$ ***

Nach dem Prinzip der vollständigen Induktion gilt $P(n)$ für alle $n \in \mathbb{N}$.

Beispiel 6
Eine natürliche Zahl n heißt *gerade*, wenn gilt $\bigvee_{x\in\mathbb{N}} n = 2x$. Eine natürliche Zahl n heißt *ungerade*, wenn gilt $\bigvee_{x\in\mathbb{N}} n = 2x+1$. Wir wollen beweisen, daß jede natürliche Zahl n gerade oder ungerade ist. Hierzu betrachten wir das folgende Prädikat P auf $\mathbb{N}$

$$P(n) :\Longleftrightarrow n \text{ ist gerade} \vee n \text{ ist ungerade}$$

und beweisen $P(n)$ durch vollständige Induktion nach n. Das schaut für dieses Prädikat wie folgt aus.

Beweis von

$$n \text{ ist gerade} \vee n \text{ ist ungerade}$$

durch vollständige Induktion nach n

INDUKTIONSANFANG: 0 ist gerade $\vee$ 0 ist ungerade.

Beweis: Wegen $0 = 2 \cdot 0$ ist 0 gerade. Also gilt die Behauptung.

INDUKTIONSVORAUSSETZUNG: n ist gerade $\vee$ n ist ungerade

INDUKTIONSBEHAUPTUNG: $n+1$ ist gerade $\vee$ $n+1$ ist ungerade

Beweis: Nach Induktionsvoraussetzung ist n gerade oder ungerade. Entsprechend unterscheiden wir zwei Fälle:

1. Fall n ist gerade:
Dann gibt es ein $x \in \mathbb{N}$ so, daß $n = 2x$. Dann ist $n+1 = 2x+1$ und damit ist $n+1$ ungerade und daher gilt die Induktionsbehauptung.

2. Fall n ist ungerade:
Dann gibt es ein $x \in \mathbb{N}$ so, daß $n = 2x+1$. Dann ist $n+1 = (2x+1)+1 = 2x+2 = 2(x+1)$. Damit ist $n+1$ gerade und es folgt die Induktionsbehauptung.

Nach dem Prinzip der vollständigen Induktion gilt

$$n \text{ ist gerade} \vee n \text{ ist ungerade}$$

für alle $n \in \mathbb{N}$.

Satz 1 (zur Definition durch vollständige Induktion) *Sei A eine Menge, $a \in A$ und $g\colon A \times \mathbb{N} \to A$. Dann gibt es genau eine Funktion $h\colon \mathbb{N} \to A$, die folgende Gleichungen erfüllt.*

$$\begin{aligned} h(0) &= a \\ h(x+1) &= g(h(x), x). \end{aligned}$$

Diese beiden Gleichungen werden *Rekursionsgleichungen* genannt. Man sagt, daß sie die Funktion h *durch vollständige Induktion definieren.* Eine solche Definition ist eine spezielle Art von *Rekursion*, d.h. einer Definition eines Begriffs durch sich selbst.

Beispiel 7
Seien $a_0, a_1, a_2, \cdots \in \mathbb{R}$. Dann wird durch die Rekursionsgleichungen

$$h(0) = a_0$$
$$h(x+1) = h(x) + a_{x+1}$$

eine Funktion $h\colon \mathbb{N} \to \mathbb{R}$ definiert. Dies ist ein Beispiel für eine Definition durch vollständige Induktion, wobei $A = \mathbb{R}$, $a = a_0$ und g die Funktion $(s, x) \mapsto s + a_{x+1}\colon \mathbb{R} \times \mathbb{N} \to \mathbb{R}$ ist. Für $h(x)$ schreibt man üblicherweise $\sum_{i=0}^{x} a_i$. Die Rekursionsgleichungen schauen dann so aus.

$$\sum_{i=0}^{0} a_i = a_0$$
$$\sum_{i=0}^{x+1} a_i = \sum_{i=0}^{x} a_i + a_{x+1}$$

Man sagt, daß diese Rekursionsgleichungen die Zahl $\sum_{i=0}^{x} a_i$ für jedes $x \in \mathbb{N}$ *durch vollständige Induktion nach x definieren.*

Beispiel 8
Nehmen wir an, wir wissen, was die Addition von natürlichen Zahlen ist. Dann kann man die Multiplikation folgendermaßen definieren. Für $x, y \in \mathbb{N}$ wird $x \cdot y$ durch vollständige Induktion nach y definiert durch die folgenden Rekursionsgleichungen.

$$x \cdot 0 = 0$$
$$x \cdot (y+1) = x \cdot y + x$$

Dies ist ebenfalls ein Beispiel für eine Definition durch vollständige Induktion. Hier ist $A = \mathbb{N}$, $a = 0$ und g ist die Funktion $(p, y) \mapsto p + x\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Wertverlaufsinduktion und -rekursion

Eine Variante des Prinzips der vollständigen Induktion ist das

Satz 2 (Prinzip der Wertverlaufsinduktion) *Sei P ein Prädikat auf $\mathbb{N}$ mit der folgenden Eigenschaft*

INDUKTIONSSCHRITT $\quad \bigwedge_{n\in\mathbb{N}} \big(\bigwedge_{x<n} P(x) \implies P(n)\big).$

Dann gilt $\quad \bigwedge_{n\in\mathbb{N}} P(n).$

Im Induktionsschritt nennt man die Aussage $\bigwedge_{x<n} P(x)$ die *Induktionsvoraussetzung* und die Aussage $P(n)$ die *Induktionsbehauptung.*

Im Gegensatz zum Prinzip der vollständigen Induktion ist bei der Wertverlaufsinduktion kein Induktionsanfang nötig, da der Induktionsschritt für $n = 0$ den Induktionsanfang bereits liefert.

Beispiel 9
Unter einer *Primzahl* versteht man eine natürliche Zahl > 1, die nur durch 1 und durch sich selbst teilbar ist. Wir beweisen durch Wertverlaufsinduktion nach n:

Jede natürliche Zahl $n > 1$ ist durch eine Primzahl teilbar.

Beweis durch Wertverlaufsinduktion nach n

INDUKTIONSVORAUSSETZUNG: Jede natürliche Zahl x mit $1 < x < n$ ist durch eine Primzahl teilbar.

INDUKTIONSBEHAUPTUNG: n ist durch eine Primzahl teilbar, falls $n > 1$ ist.

Beweis: Wenn n eine Primzahl ist, dann ist n durch sich selbst teilbar und damit gilt die Behauptung. Ist aber n keine Primzahl, so ist n nach Definition des Begriffs der Primzahl durch eine Zahl x mit $1 < x < n$ teilbar. Nach Induktionsvoraussetzung ist x durch eine Primzahl teilbar. Damit ist auch n durch dieselbe Primzahl teilbar und somit gilt auch in diesem Fall die Induktionsbehauptung.

Nach dem Prinzip der Wertverlaufsinduktion ist daher jede natürliche Zahl $n > 1$ durch eine Primzahl teilbar.

Das Prinzip der vollständigen Induktion ist ein Spezialfall der Wertverlaufsinduktion. Umgekehrt läßt sich das Prinzip der Wertverlaufsinduktion mit dem Prinzip der vollständigen Induktion beweisen.

Wie das Prinzip der vollständigen Induktion, so läßt sich auch das Prinzip der Wertverlaufsinduktion zur Definition verwenden. Man spricht dann von *Wertverlaufsrekursion.* Bei der Wertverlaufsrekursion einer Funktion $h : \mathbb{N} \to A$ wird $h(n)$ durch Bezugnahme auf die Werte $h(x)$ von h mit $x < n$, definiert. Die Definition von $h(n)$ nimmt also Bezug auf die Einschränkung $h|_{\{x|x\in\mathbb{N}\wedge x<n\}}$.

Satz 3 (Wertverlaufsrekursion) *Sei A eine Menge und sei F eine Funktion, die jeder Funktion $f\colon \{x \mid x \in \mathbb{N} \wedge x < n\} \to A$ ein $F(f) \in A$ zuordnet. Dann gibt es genau eine Funktion $h\colon \mathbb{N} \to A$ mit*

$$\bigwedge_{n\in\mathbb{N}} h(n) = F(h|_{\{x|x\in\mathbb{N}\wedge x<n\}}).$$

Beispiel 10
Die Fibonaccizahlen $\mathrm{fib}(n)$ für $n = 0, 1, 2, \ldots$ sind definiert durch

$$\mathrm{fib}(n) = \begin{cases} 1, & \text{wenn } n = 0 \\ 1, & \text{wenn } n = 1 \\ \mathrm{fib}(n-2) + \mathrm{fib}(n-1) & \text{sonst.} \end{cases}$$

Dies ist eine Wertverlaufsrekursion mit $A = \mathbb{N}$ und $F(f) := f(n-2) + f(n-1)$ für alle $f\colon \{x \mid x \in \mathbb{N} \wedge x < n\} \to \mathbb{N}$.

Der Begriff der induktiven Definition

Definition 10 Sei A eine Menge und sei $R \subseteq \mathfrak{P}(A) \times A$. Weiter sei $B \subseteq A$ derart, daß gilt

$$\bigwedge_{(X,y)\in R} (X \subseteq B \implies y \in B).$$

Dann sagt man, die Menge B sei *abgeschlossen* gegenüber der Relation R.

Beispiel 11
Sei Σ das Alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{+}, \mathtt{*}, \mathtt{(}, \mathtt{)}\}$ und sei $A = \Sigma^*$ die Menge der Wörter über dem Alphabet Σ. Sei B die Menge der voll geklammerten arithmetischen Ausdrücke über `0` und `1`. Die Relation $R_+ \subseteq \mathfrak{P}(\Sigma^*) \times \Sigma^*$ sei folgendermaßen definiert. Es gelte $(X, Y) \in R_+$ genau dann, wenn $u, v \in \Sigma^*$ existieren so, daß $X = \{u, v\}$ und $y = \mathtt{(}u\mathtt{+}v\mathtt{)}$. Dann ist die Menge B abgeschlossen gegenüber der Relation R_+. Man sagt auch, die Menge B sei *abgeschlossen* gegenüber der zweistelligen Operation $(u, v) \mapsto \mathtt{(}u\mathtt{+}v\mathtt{)} : \Sigma^* \times \Sigma^* \to \Sigma^*$ auf Σ^*.

Beispiel 12
Seien Σ, A und B wie im letzten Beispiel. Die Relation R sei folgendermaßen definiert. Es gelte $(X, y) \in R$ genau dann, wenn eine der folgenden Aussagen gilt.

1. $X = \emptyset$ und $y = \mathtt{0}$
2. $X = \emptyset$ und $y = \mathtt{1}$
3. $X = \{u, v\}$ mit $u, v \in \Sigma^*$ und $y = \mathtt{(}u\mathtt{+}v\mathtt{)}$.
4. $X = \{u, v\}$ mit $u, v \in \Sigma^*$ und $y = \mathtt{(}u\mathtt{*}v\mathtt{)}$.

Dann ist B abgeschlossen gegenüber der Relation R. Äquivalent zu dieser Aussage ist die folgende Aussage:

Die Menge B der voll geklammerten arithmetischen Ausdrücke über `0` und `1` enthält die Wörter `0` und `1` und ist abgeschlossen gegenüber den zweistelligen Operationen $(u, v) \mapsto \mathtt{(}u\mathtt{+}v\mathtt{)}$ und $(u, v) \mapsto \mathtt{(}u\mathtt{+}v\mathtt{)}$ auf Σ^*.

Satz 4 (zur induktiven Definition) *Sei A eine Menge und sei $R \subseteq \mathfrak{P}(A) \times A$. Dann gibt es eine kleinste Menge $B \subseteq A$, die abgeschlossen ist gegenüber der Relation R.*

Man sagt, diese Menge sei die durch die Relation R *induktiv definierte Menge.*

Beweis des Satzes: Offenbar gibt es eine Menge B, die abgeschlossen ist gegenüber R, nämlich die Menge A. Daher existiert der Durschnitt B_0 aller Mengen B, die abgeschlossen sind gegenüber R. Die Menge B_0 ist ebenfalls abgeschlossen gegenüber R, denn sei $(X, y) \in R$ mit $X \subseteq B_0$. Dann gilt für alle Mengen B, die aggeschlossen gegenüber R sind, nach Definition von B_0 die Aussage $B_0 \subseteq B$ und daher $X \subseteq B$. Da die Menge B die Aussage abgeschlossen gegenüber R ist, gilt $y \in B$. Da dies für alle Mengen B, die abgeschlossen gegenüber R sind, gilt, folgt aus der Definition von B_0, daß $y \in B_0$ ist. Also ist auch B_0 abgeschlossen gegenüber R. Aus der Definition von B_0 folgt die Behauptung. q.e.d.

Beispiel 13
Sei R wie im letzten Beispiel definiert. Die durch die Relation R induktiv definierte Menge ist die Menge der voll geklammerten arithmetischen Ausdrücke über `0` und `1`. Die induktive Definition läßt sich auch so schreiben.

1. `0` ist ein voll geklammerter arithmetischer Ausdruck.
2. `1` ist ein voll geklammerter arithmetischer Ausdruck.
3. Wenn u und v voll geklammerte arithmetische Ausdrücke sind, dann ist auch `(`u`+`v`)` ein voll geklammerter arithmetischer Ausdruck.
4. Wenn u und v voll geklammerte arithmetische Ausdrücke sind, dann ist auch `(`u`*`v`)` ein voll geklammerter arithmetischer Ausdruck.

Man sagt, daß diese vier Regeln eine *induktive Definition des Begriffs* eines voll geklammerten arithmetischen Ausdrucks bilden. Ein Wort ist genau dann ein voll geklammerter arithmetischer Ausdruck, wenn es sich durch endlichmalige Anwendung der Regeln als ein solcher nachweisen läßt.

Satz 5 (Beweis durch Induktion über eine induktive Definition) *Sei A eine Menge und sei $B \subseteq A$ induktiv definiert durch eine Relation $R \subseteq \mathfrak{P}(A) \times A$. Weiter sei P ein Prädikat auf A mit der folgenden Eigenschaft*

$$\textsc{Induktionsschritt} \qquad \bigwedge_{(X,y)\in R}\big(\bigwedge_{x\in X} P(x) \implies P(y)\big).$$

Dann gilt $\bigwedge_{b\in B} P(b)$.

Im Induktionsschritt nennt man die Aussage $\bigwedge_{x\in X} P(x)$ die *Induktionsvoraussetzung* und die Aussage $P(y)$ die *Induktionsbehauptung*.

Beweis des Satzes: Die Menge P ist eine Teilmenge von A. Nach Definition von B ist B die kleinste Menge, die abgeschlossen gegenüber R ist. Aber nach dem Induktionsschritt ist auch die Menge P abgeschlossen gegenüber R. Daher ist $B \subseteq P$. Somit gilt $\bigwedge_{b\in B} P(b)$. q.e.d.

Beispiel 14
Jeder voll geklammerte arithmetische Ausdruck über `0` und `1` hat gleichviele öffnende wie schließende Klammern.

Beweis durch Induktion über die Definition des Begriffs eines voll geklammerten arithmetischen Ausdrucks über `0` und `1`: (Der Beweis benutzt Satz 5 mit dem Prädikat P auf Σ^*, das definiert ist durch

$$P(u) : \Longleftrightarrow u \text{ hat gleich viele öffnende wie schließende Klammern}$$

für alle $u \in \Sigma^*$.)

1. `0` hat gleichviele öffende wie schließende Klammern, nämlich 0.

2. `1` hat gleichviele öffende wie schließende Klammern, nämlich 0.

3. Als Induktionsvoraussetzung nehmen wir an, daß u gleichviele öffnende wie schließende Klammern hat (sagen wir m) und v ebenfalls (sagen wir n). Dann hat `(`u`+`v`)` ebenfalls gleichviele öffnende wie schließende Klammern, nämlich $m + n + 1$.

4. Als Induktionsvoraussetzung nehmen wir an, daß u gleichviele öffnende wie schließende Klammern hat (sagen wir m) und v ebenfalls (sagen wir n). Dann hat `(`u`*`v`)` ebenfalls gleichviele öffnende wie schließende Klammern, nämlich $m + n + 1$.

q.e.d.

Beispiel 15
Die *Nachfolgerfunktion* $N\colon \mathbb{N} \to \mathbb{N}$ ist gegeben durch $N(n) = n + 1$. Mit Hilfe der Nachfolgerfunktion läßt sich der Begriff der natürlichen Zahlen induktiv definieren.

Eine induktive Definition der natürlichen Zahlen:

1. 0 ist eine natürliche Zahl.

2. Wenn n eine natürliche Zahl ist, dann auch $N(n)$.

Ein Beweis durch Induktion über diese induktive Definition ist dann nichts anderes als ein Beweis durch vollständige Induktion.

1 Berechenbarkeitstheorie

Algorithmus: 'mechanisches' Verfahren, das zu einer Eingabe $x \in A$ eine Ausgabe $y \in B$ zu berechnen versucht. Die Menge A der möglichen Eingaben und die Menge B der möglichen Ausgaben kann sein

- Eine Zahlenmenge (z.B. $\mathbb{N}$ oder $\mathbb{Z}$)
- Die Menge der Wörter (oder Zeichenketten, englisch 'strings')
- Bilder, Filme, Audioaufnahmen, etc.
- Zusammengesetzte Datenstrukturen (z.B. $\mathbb{N} \times \mathbb{N}$).

Jede mögliche Ein- oder Ausgabe läßt sich im Prinzip als Wort über einem geeigneten Zeichensatz darstellen. Ebenso läßt sie sich, wie wir später (bei den Gödel-Nummerierungen) sehen werden, durch eine natürliche Zahl darstellen. Der Berechenbarkeitsbegriff läßt sich daher, wie wir sehen werden, unabhängig vom Datentyp formulieren und dann auf die anderen Datentypen übertragen.

Definition 11 Eine Funktion $f : A \to B$ heißt *berechenbar*, wenn es einen Algorithmus gibt, der bei Eingabe von $x \in A$ stets die Ausgabe $f(x)$ liefert. Man sagt dann, die Funktion f werde durch diesen Algorithmus *berechnet*.

Die Theorie der Berechenbarkeit befaßt sich mit der Frage, welche Funktionen berechenbar sind.

Systeme zur Beschreibung des Berechenbarkeitsbegriffes

Um den Begriff der berechenbaren Funktion formal zu fassen, wurden, vor allem in der ersten Hälfte des 20. Jahrhunderts, verschiedene Formalismen entwickelt, darunter die folgenden.

- μ-partiellrekursive Funktionen
- Chomski-Grammatiken
- Turing-Maschinen
- Kalküle
- λ-Kalül
- Kombinatoren

⋮

- Verschiedene Programmiersprachen

Jeder dieser Formalismen definiert eine Klasse von berechenbaren Funktionen und gibt an, wie diese Funktionen zu berechnen sind. Wir werden uns hier mit dem Formalismus der Turingmaschinen befassen.

Es hat sich herausgestellt, daß all diese Formalismen die gleiche Menge von berechenbaren Funktionen beschreiben. Da man trotz aller Anstrengungen bisher keine Prinzipien zur Formulierung von Algorithmen gefunden hat, die sich nicht in jedem dieser Formalismen darstellen ließen, liegt die folgende These nahe.

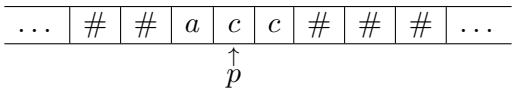
Churchsche These Jede berechenbare Funktion ist in jedem dieser Formalismen berechnenbar.

1.1 Turingmaschinen

1.1.1 Grundbegriffe

Eine Turingmaschine ist ein Automat, der sich zu jedem Zeitpunkt in einem Zustand aus einer vorgegebenen endlichen Zustandsmenge Q befindet und dem zusätzlich ein unbegrenzter Speicher in Form eines nach rechts und links unbegrenzten Bandes zur Verfügung steht. Dieses Band ist aufgeteilt in nebeneinander gereihte Felder.

Auf jedem der Felder steht ein Zeichen aus einem Alphabet (*Bandalphabet*) Σ. Der Automat besitzt einen Schreib/Lesekopf, der sich zu jedem Zeitpunkt auf genau einem der Felder befindet. Dieses Feld nennt man das (momentane) *Arbeitsfeld*. Die Situation einer Turingmaschine zu einem gegebenen Zeitpunkt nennen wir *Konfiguration*. Wir stellen sie mit einem Bild so dar, daß wir den Schreib/Lesekopf durch einen Pfeil symbolisieren, der auf das Arbeitsfeld zeigt. Darunter schreiben wir den Namen des momentanen Zustandes. In der folgenden Abbildung steht der Schreib/Lesekopf auf dem linken c und die Maschine befindet sich im Zustand p



bei einem Bandalphabet $\{a, b, c, \#\}$ und bei einem Band, das von links nach rechts zunächst unendlich oft das Zeichen #, dann a, c und c enthält und schließlich wieder unendlich oft das Zeichen #.

Eine Turingmaschine führt nacheinander jeweils einen Berechnungsschritt durch. Ein Berechnungsschritt besteht aus der Ausführung eines Elementarbefehls und dem Übergang der Turingmaschine in einen neuen Zustand. Es gibt drei Arten von Elementarbefehlen:

Elementarbefehle

1. Das Ersetzen des Zeichens auf dem Arbeitsfeld durch ein Zeichen $a \in \Sigma$. Diesen Elementarbefehl bezeichnen wir mit a.

2. Das Verschieben des Schreib/Lesekopfes um ein Feld nach links. Das neue Arbeitsfeld ist dann der linke Nachbar des alten Arbeitsfeldes. Diesen Elementarbefehl bezeichnen wir mit dem Zeichen L.

3. Das Verschieben des Schreib/Lesekopfes um ein Feld nach rechts. Das neue Arbeitsfeld ist dann der rechte Nachbar des alten Arbeitsfeldes. Diesen Elementarbefehl bezeichnen wir mit dem Zeichen R.

Die Elementarbefehle werden also mit Zeichen aus dem Alphabet $B := \Sigma \cup \{L, R\}$ bezeichnet, wobei L und R Zeichen sind, die nicht in Σ vorkommen.

Eines der Zeichen aus Σ wird ausgezeichnet und als *Leerzeichen* $\#$ bezeichnet. Am Anfang einer Berechnung enthält das Band die Eingabe in Form einer Beschriftung von endlich vielen Feldern mit Zeichen aus dem Bandalphabet Σ. Alle übrigen Felder sind dabei mit dem Leerzeichen $\#$ beschriftet. Der Inhalt des nicht leeren Teiles des Bandes nach der Berechnung dient als Ausgabe.

Definition 12 Eine *Turingmaschine* ist ein Quintupel $(Q, \Sigma, S, H, \delta)$ mit

1. Menge der *Zustände* Q ist eine endliche Menge.

2. *Bandalphabet* Σ ist ein Alphabet, *Leerzeichen* $\# \in \Sigma$ und $L, R \notin \Sigma$.

3. *Anfangszustand* oder *Startzustand* $S \in Q$.

4. *Endzustand* oder *Haltezustand* $H \in Q$ mit $H \neq S$.

5. *Übergangsfunktion* $\delta\colon (Q \setminus \{H\}) \times \Sigma \to B \times (Q \setminus \{S\})$ mit $B := \Sigma \cup \{L, R\}$.

Am Anfang einer Berechnung befindet sich die Turingmaschine im Anfangszustand S, am Ende einer Berechnung im Endzustand H. Befindet sie sich zu einem Zeitpunkt im Zustand q und ist das Zeichen auf dem Arbeitsfeld a und gilt $\delta(q, a) = (b, p)$, so führt die Turingmaschine den Elementarbefehl b aus und geht in den Zustand p über.

Notation 2 Als abkürzende Schreibweise für eine Turingmaschine verwenden wir die folgende: Für jedes $(q, a) \in (Q \setminus \{H\}) \times \Sigma$ schreiben wir eine Zeile $q\ a\ b\ p$, wobei $\delta(q, a) = (b, p)$. Für eine Turingmaschine $(Q, \Sigma, S, H, \delta)$ schreiben wir also

$$\begin{array}{cccc} q_1 & a_1 & b_1 & p_1 \\ & & \vdots & \\ q_n & a_n & b_n & p_n \end{array}$$

wobei $\{(q_i, a_i) \mid i = 1, \ldots, n\} = (Q \setminus \{H\}) \times \Sigma$ und $\delta(q_i, a_i) = (b_i, p_i)$ für $i = 1, \ldots, n$.

Um nicht jedesmal den Anfangs- und Endzustand explizit angeben zu müssen, vereinbaren wir außerdem, daß eine der Zeilen, die mit S beginnen, an den Anfang geschrieben wird und eine der Zeilen, die mit H enden, ans Ende: $q_1 = S$ und $p_n = H$. Außerdem vereinbaren wir, daß wir Zeilen der Form $q\ a\ a\ q$ weglassen dürfen.

Beispiel 16
Sei $\Sigma = \{a, b, c, \#\}$ ein Alphabet und $Q = \{S, q, H\}$. Sei $L_\#$ die Turingmaschine $(Q, \Sigma, S, H, \delta)$, wobei

$$\begin{aligned}
\delta(S, a) &:= (L, q) \\
\delta(S, b) &:= (L, q) \\
\delta(S, c) &:= (L, q) \\
\delta(S, \#) &:= (L, q) \\
\delta(q, a) &:= (L, q) \\
\delta(q, b) &:= (L, q) \\
\delta(q, c) &:= (L, q) \\
\delta(q, \#) &:= (\#, H)
\end{aligned}$$

Dann können wir für die Turingmaschine $L_\#$ kurz schreiben

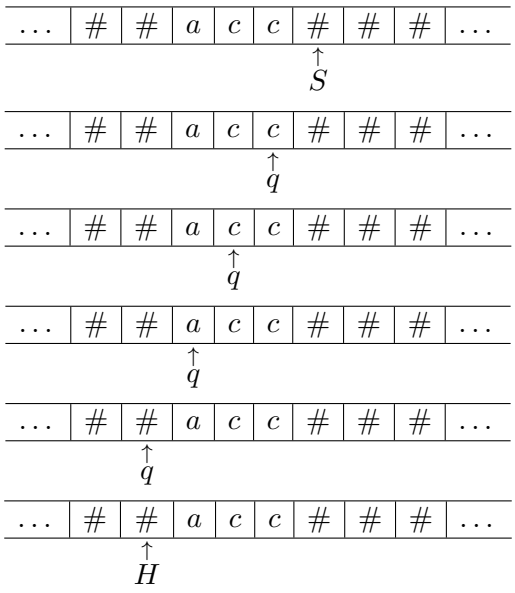
S	a	L	q
S	b	L	q
S	c	L	q
S	$\#$	L	q
q	a	L	q
q	b	L	q
q	c	L	q
q	$\#$	$\#$	H

Läßt man diese Turingmaschine etwa starten mit der Konfiguration

$\ldots$	$\#$	$\#$	a	c	c	$\#$	$\#$	$\#$	$\ldots$
						$\uparrow$			
						S			

,

so läuft die Berechnung folgendermaßen ab.



Hier ist die Maschine im Haltezustand und hält an. Diese Maschine macht also nichts anderes als vom anfänglichen Arbeitsfeld aus solange nach links zu gehen, bis sie ein Feld mit dem Leerzeichen # findet.

Wir werden Konfigurationen oft auch kurz so schreiben, daß wir die Abbildung des Turingbandes und den Pfeil sowie führende Leerzeichen links vom Arbeitsfeld und nachfolgende Leerzeichen rechts vom Arbeitsfeld weglassen. Für die Startkonfiguration im Beispiel schaut das dann so aus:

$$\underset{S}{acc\#}$$

Definition 13 Wenn T eine Turingmaschine ist, dann definieren wir die zweistellige Relation $\Rightarrow_T$ auf der Menge der Konfigurationen von T folgendermaßen. Für zwei Konfigurationen K_1 und K_2 gilt $K_1 \Rightarrow_T K_2$ genau dann, wenn die Turingmaschine T die Konfiguration K_1 in einem Berechnungsschritt in die Konfiguration K_2 überführt.

Im Beispiel gilt

$$\underset{S}{acc\#} \Rightarrow_{L\#} \underset{q}{acc} \Rightarrow_{L\#} \underset{q}{acc} \Rightarrow_{L\#} \underset{q}{acc} \Rightarrow_{L\#} \underset{q}{\#acc} \Rightarrow_{L\#} \underset{H}{\#acc}$$

Das Beispiel läßt sich verallgemeinern auf ein beliebiges Bandalphabet:

Beispiel 17
Sei $\Sigma = \{a_1, \ldots, a_k, \#\}$ ein Alphabet und $Q = \{S, q, H\}$. Sei $L_\#$ die Turingmaschine

$(Q, \Sigma, S, H, \delta)$, wobei

$$\begin{array}{ll} \delta(S, a) := (L, q) & \text{für } a \in \Sigma \\ \delta(q, a_i) := (L, q) & \text{für } i = 1, \ldots, k \\ \delta(q, \#) := (\#, H) & \end{array}$$

Dann können wir für die Turingmaschine $L_\#$ kurz schreiben

$$\begin{array}{cccc} S & a & L & q \qquad (a \in \Sigma) \\ q & a_i & L & q \qquad (i = 1, \ldots, k) \\ q & \# & \# & H \end{array}$$

Definition 14 Wenn eine Turingmaschine T eine Konfiguration $u\underset{S}{a}v$ im Startzustand in eine Konfiguration $w\underset{H}{b}x$ im Haltezustand überführt, also $u\underset{S}{a}v \Rightarrow_T^* w\underset{H}{b}x$, so schreiben wir $u\underline{a}v \Rrightarrow_T w\underline{b}x$.

Zum Beispiel gilt im vorletzten Beispiel $acc\underline{\#} \Rrightarrow_{L_\#} \underline{\#}acc$. Allgemein:

Lemma 1 *Für alle $u, w \in \Sigma^*$, $a \in \Sigma$ und $v \in (\Sigma \setminus \{\#\})^*$ gilt*

$$u\#v\underline{a}w \Rrightarrow_{L_\#} u\underline{\#}vaw.$$

1.1.2 Zusammensetzung von Turingmaschinen

Seien $T, T', T_1, T_2, \ldots$ Turingmaschinen.

Definition 15 (von TT') Wenn

$$T = \begin{array}{cccc} q_1 & a_1 & b_1 & p_1 \\ & \vdots & & \\ q_m & a_m & b_m & p_m \end{array}, \qquad T' = \begin{array}{cccc} q'_1 & a'_1 & b'_1 & p'_1 \\ & \vdots & & \\ q'_n & a'_n & b'_n & p'_n \end{array}$$

und $p_m = q'_1$ und T und T' sonst keine gemeinsamen Zustände haben, dann ist

$$TT' = \begin{array}{cccc} q_1 & a_1 & b_1 & p_1 \\ & \vdots & & \\ q_m & a_m & b_m & p_m \\ q'_1 & a'_1 & b'_1 & p'_1 \\ & \vdots & & \\ q'_n & a'_n & b'_n & p'_n \end{array}.$$

Andernfalls benenne man die Zustände von T' vorher so um, daß die Bedingung für die Zustände erfüllt ist.

Beispiel 18
Die Turingmaschine $L_\#$ für das Bandalphabet $\{a, \#\}$ ist

$$\begin{array}{cccc} S & a & L & q \\ S & \# & L & q \\ q & a & L & q \\ q & \# & \# & H \end{array}$$

Wir wollen die Maschine $L_\# L_\#$ (wir schreiben auch $L_\#^2$) bestimmen. Hierzu müssen wir zwei Kopien der Zeilen von $L_\#$ untereinander schreiben. Bei der zweiten Kopie müssen wir die Zustände von $L_\#$ umbenennen, und zwar den Anfangszustand S in den Endzustand H der ersten Kopie von $L_\#$ und alle übrigen Zustände in neue Zustände, also etwa q in p und H in Z. Bezeichnen wir das Ergebnis der Umbenennung der Zustände in der Turingmaschine $L_\#$ mit $\tilde{L}_\#$, so schauen die Zeilen von $\tilde{L}_\#$ so aus.

$$\begin{array}{cccc} H & a & L & p \\ H & \# & L & p \\ p & a & L & p \\ p & \# & \# & Z \end{array}$$

Die Zeilen der Maschine $L_\# L_\#$ sind nun die Zeilen von $L_\#$ gefolgt von den Zeilen von $\tilde{L}_\#$, also

$$\begin{array}{cccc} S & a & L & q \\ S & \# & L & q \\ q & a & L & q \\ q & \# & \# & H \\ H & a & L & p \\ H & \# & L & p \\ p & a & L & p \\ p & \# & \# & Z \end{array}$$

S ist der Startzustand von $L_\# L_\#$ und Z ist der Haltezustand von $L_\# L_\#$.

Lemma 2 *Seien $u, v, y, z \in \Sigma^*$ und $a, c \in \Sigma$. Dann gilt $u\underline{a}v \Rrightarrow_{TT'} y\underline{c}z$ genau dann, wenn es $w, x \in \Sigma^*$ und $b \in \Sigma$ gibt mit $u\underline{a}v \Rrightarrow_T w\underline{b}x$ und $w\underline{b}x \Rrightarrow_{T'} y\underline{c}z$.*

Eine Berechnung durch die Turingmaschine TT' bewirkt also das gleiche wie eine Berechnung durch T gefolgt von einer Berechnung durch T'.

Definition 16 Sind $q_1, p_1, \ldots, q_n, p_n$ beliebige Zustände und $a_1, \ldots, a_n \in \Sigma$ so, daß $(q_1, a_1), \ldots, (q_n, a_n)$ paarweise verschieden sind und ist q_i der Anfangszustand von T_i und p_i der Endzustand von T_i und sind alle übrigen Zustände von T_i verschieden von den Zuständen $q_1, p_1, \ldots, q_n, p_n$ und von den Zuständen aller anderen T_j, dann seien die Zeilen der Turingmaschine

$$\begin{array}{cccc} q_1 & a_1 & T_1 & p_1 \\ & & \vdots & \\ q_n & a_n & T_n & p_n \end{array}$$

alle Zeilen von T_1 außer denen, die mit $q_1 a$ beginnen mit $a \neq a_1$ und ... und alle Zeilen von T_n außer denen, die mit $q_n a$ beginnen mit $a \neq a_n$.

Andernfalls benenne man die Zustände von $T_1, \ldots, T_n$ vorher so um, daß die Bedingung für die Zustände erfüllt ist.

Beispiel 19
Die Turingmaschine $L_\#$ für das Bandalphabet $\{a, \#\}$ ist

$$\begin{array}{cccc} S & a & L & q \\ S & \# & L & q \\ q & a & L & q \\ q & \# & \# & H \end{array}$$

Daher ist

$$\begin{array}{cccc} S & a & L_\# & H \end{array}$$

die Turingmaschine

$$\begin{array}{cccc} S & a & L & q \\ q & a & L & q \\ q & \# & \# & H \end{array}$$

Sie ergibt sich aus $L_\#$ durch Streichung der zweiten Zeile, die mit S und einem von a verschiedenen Zeichen (nämlich # beginnt).

Lemma 3 *Es gilt* $u\underline{a}v \Rrightarrow_{\substack{q_1 a_1 T_1 p_1 \\ \ldots \\ q_n a_n T_n p_n}} u'\underline{a}'v'$ *genau dann, wenn es* $i_1, \ldots, i_k$ *und* $u_1, v_1, \ldots, u_k, v_k \in \Sigma^*$ *gibt so, daß*

1. $q_{i_1} = q_1 \wedge \bigwedge_{j<k} p_{i_j} = q_{i_{j+1}} \wedge p_{i_k} = p_n$

2. $u\underline{a}v = u_1\underline{a_{i_1}}v_1 \Rrightarrow_{T_{i_1}} \ldots \Rrightarrow_{T_{i_{k-1}}} u_k\underline{a_{i_k}}v_k \Rrightarrow_{T_{i_k}} u'\underline{a}'v'$

Die Turingmaschine

$$\begin{array}{cccc} q_1 & a_1 & T_1 & p_1 \\ & & \vdots & \\ q_n & a_n & T_n & p_n \end{array}$$

hat also die gleiche Wirkung, die wir erzielen würden, wenn wir den Begriff der Turingmaschine so erweitern würden, daß wir anstatt Elementarbefehlen auch ganze Turingmaschinen zulassen würden und dann diese Zeilen als Bezeichnung für eine so verallgemeinerte Turingmaschine auffassen würden. Die Schreibweise erlaubt uns also Turingmaschinen $T_1, \ldots, T_n$ als Unterprogramme in einer anderen Turingmaschine zu verwenden.

1.1.3 Weitere Beispiele für Turingmaschinen

Sei Σ ein fest vorgegebenes Alphabet mit $\# \in \Sigma$. Sei $\Delta := \Sigma \setminus \{\#\}$.

Suchen des nächsten Leerzeichens rechts vom Arbeitsfeld Sei $R_\#$ die Turingmaschine

$$\begin{array}{cccc} S & a & R & q \qquad (a \in \Sigma) \\ q & a & R & q \qquad (a \in \Delta) \\ q & \# & \# & H. \end{array}$$

Lemma 4 *Für alle $u, w \in \Sigma^*$, $a \in \Sigma$ und $v \in \Delta^*$ gilt $u\underline{a}v\#w \Rrightarrow_{R_\#} uav\underline{\#}w$.*

Elementare Turingmaschinen Für $a \in \Sigma$ bezeichnen wir mit a die Turingmaschine

$$S \quad c \quad a \quad H \qquad (c \in \Sigma).$$

Lemma 5 *Für $u, v \in \Sigma^*$ und $a, c \in \Sigma$ gilt $u\underline{c}v \Rrightarrow_a u\underline{a}v$.*

Die Maschine L

$$S \quad a \quad L \quad H \qquad (a \in \Sigma)$$

Lemma 6 *Für $u, v \in \Sigma^*$ und $a, b \in \Sigma$ gilt $ua\underline{b}v \Rrightarrow_L u\underline{a}bv$.*

Die Maschine R

$$S \quad a \quad R \quad H \qquad (a \in \Sigma)$$

Lemma 7 *Für $u, v \in \Sigma^*$ und $a, b \in \Sigma$ gilt $u\underline{a}bv \Rrightarrow_R ua\underline{b}v$.*

Kopierung K_n

$$\begin{array}{cccc} S & \# & (L_\#^n R) & q \\ q & a & T_a & q \qquad (a \in \Delta) \\ q & \# & R_\#^n & H \end{array}$$

wobei $T_a = \#R_\#^{n+1}aL_\#^{n+1}aR$.

Lemma 8 *Für alle $u \in \Sigma^*$ und $w_1, \ldots, w_n \in \Delta^*$ gilt*

$$u\#w_1\#\ldots\#w_n\underline{\#} \Rrightarrow_{K_n} u\#w_1\#\ldots\#w_n\#w_1\underline{\#}.$$

Linksverschiebung V

$$\begin{array}{cccc} S & \# & R & q \\ q & a & (LaRR) & q \qquad (a \in \Delta) \\ q & \# & (L\#R) & H \end{array}$$

Lemma 9 *Für $u, w \in \Sigma^*$ und $v \in \Delta^*$ gilt $u\underline{\#}v\#w \Rrightarrow_V uv\#\underline{\#}w$.*

Streichung S_n

$$\begin{array}{llll} S & \# & (L^n_{\#}L) & q \\ q & a & (\#RV^nLL^n_{\#}L) & q \qquad (a \in \Delta) \\ q & \# & (RV^nL) & H \end{array}$$

Lemma 10 *Für* $u \in \Sigma^*$ *und* $w_0, \ldots, w_n \in \Delta^*$ *gilt* $u\#w_0\#w_1\#\ldots\#w_n\underline{\#} \Rrightarrow_{S_n} u\#w_1\#\ldots\#w_n\underline{\#}$.

1.1.4 Turing-Berechenbarkeit

Definition 17 Sei $f\colon (\Delta^*)^r \xrightarrow{\text{part}} \Delta^*$. Wir sagen, f werde partiell berechnet durch eine Turingmaschine $T = (Q, \Sigma, S, H, \delta)$, wenn für alle $u_1, \ldots, u_r \in \Delta^*$ gilt

1. Wenn $(u_1, \ldots, u_r) \in \mathrm{Def}(f)$, dann $u_1\#\ldots\#u_r\underline{\#} \Rrightarrow_T f(u_1, \ldots, u_r)\underline{\#}$.
2. Andernfalls gibt es keine Berechnung, die mit $u_1\#\ldots\#u_r\underset{S}{\#}$ beginnt und hält, d.h. im Zustand H endet.

Man sagt dann, f sei partiell turingberechenbar. Ist außerdem f total, so sagt man, T berechne f, und f sei turingberechenbar.

Eine partielle zahlentheoretische Funktion $f\colon \mathbb{N}^r \xrightarrow{\text{part}} \mathbb{N}$ wird (partiell) durch eine Turingmaschine T berechnet und heißt dann (partiell) turingberechenbar, wenn das für die Funktion $(|^{a_1}, \ldots, |^{a_r}) \mapsto |^{f(a_1,\ldots,a_r)}\colon (\{|\}^*)^r \xrightarrow{\text{part}} \{|\}^*$ gilt.

Wir wollen zeigen, wie man durch Zusammensetzung von Turingmaschinen für einige partielle Funktionen auf Wörtern und auf Zahlen zeigen kann, dass sie partiell turingberechenbar sind. Hierzu benötigen wir als technisches Hilfsmittel eine etwas abgewandelte Variante des eben eingeführten Begriffs:

Definition 18 Eine *Turingmaschine für* eine partielle Funktion $f\colon (\Delta^*)^r \xrightarrow{\text{part}} (\Delta^*)^s$ ist eine Turingmaschine $T = (Q, \Sigma, S, H, \delta)$ so, daß für alle $x \in \Sigma^*$ und $u_1, \ldots, u_r \in \Delta^*$ gilt

1. Wenn $(u_1, \ldots, u_r) \in \mathrm{Def}(f)$, dann

   $$x\#u_1\#\ldots\#u_r\underline{\#} \Rrightarrow_T x\#v_1\#\ldots\#v_s\underline{\#},$$

   wobei $(v_1, \ldots, v_s) := f(u_1, \ldots, u_r)$ sei.
2. Andernfalls gibt es keine Berechnung, die mit $x\#u_1\#\ldots\#u_r\underset{S}{\#}$ beginnt und hält.

Die Idee dabei ist, dass die Turingmaschine ihren Schreib-Lesekopf nie weiter nach links bewegt als bis zum Leerzeichen nach dem x. Deshalb bleiben alle Felder links von diesem Leerzeichen von der Turingmaschine unberührt und ungelesen. Man kann solche Bereiche auf dem Band bei der

Analog wie oben sprechen wir auch von einer *Turingmaschine für* eine partielle zahlentheoretische Funktion.

Beispiel 20
K_n ist eine Turingmaschine für $(w_1, \ldots, w_n) \mapsto (w_1, \ldots, w_n, w_1)$. S_n ist eine Turingmaschine für $(w_0, w_1, \ldots, w_n) \mapsto (w_1, \ldots, w_n)$.

Lemma 11 *Wenn T eine Turingmaschine für eine r-stellige partielle Funktion f ist und $k \in \mathbb{N}$, dann ist T auch eine Turingmaschine für die $k+r$-stellige Funktion $x \cdot u \mapsto x \cdot f(u)$, wobei $\cdot$ die Operation des Aneinanderhängens von Tupeln sei.*

Im Folgenden werden einige Funktionen angegeben, für die es Turingmaschinen gibt, sowie Operationen, die, angewendet auf partielle Funktionen, für die Turingmaschinen existieren, wieder eine partielle Funktion liefern, für die eine Turingmaschine existiert.

Komposition Wenn T eine Turingmaschine für $f\colon (\Delta^*)^r \xrightarrow{\text{part}} (\Delta^*)^s$ ist und T' eine Turingmaschine für $g\colon (\Delta^*)^s \xrightarrow{\text{part}} (\Delta^*)^t$ ist, dann ist TT' eine Turingmaschine für $g \circ f$.

Permutation, Streichung, Vervielfältigung von Argumenten Seien $r, s \in \mathbb{N}$ und $i_1, \ldots, i_s \in \{1, \ldots, r\}$. Dann ist $K_{r+1-i_1} \ldots K_{r+s-i_s} S_s^r$ eine Turingmaschine für die Funktion

$$(w_1, \ldots, w_r) \mapsto (w_{i_1}, \ldots, w_{i_s})\colon (\Delta^*)^r \to (\Delta^*)^s.$$

Gibt es eine Turingmaschine für $f\colon (\Delta^*)^r \xrightarrow{\text{part}} \Delta^*$, so gibt es auch eine Turingmaschine für die Funktion

$$(w_1, \ldots, w_r) \mapsto (f(w_1, \ldots, w_r), w_1, \ldots, w_r)\colon (\Delta^*)^r \xrightarrow{\text{part}} (\Delta^*)^{r+1}.$$

Denn durch Komposition erhalten wir $(w_1, \ldots, w_r) \mapsto (w_1, \ldots, w_r, w_1, \ldots, w_r) \mapsto (w_1, \ldots, w_r, f(w_1, \ldots, w_r)) \mapsto (f(w_1, \ldots, w_r), w_1, \ldots, w_r)$.

Zusammensetzung von einwertigen Funktionen zu einer mehrwertigen Funktion Für

$$f_i\colon (\Delta^*)^r \xrightarrow{\text{part}} \Delta^* \qquad (i = 1, \ldots, s)$$

sei $[f_1, \ldots, f_s]\colon (\Delta^*)^r \xrightarrow{\text{part}} (\Delta^*)^s$ definiert durch

$$[f_1, \ldots, f_s](\mathfrak{w}) := (f_1(\mathfrak{w}), \ldots, f_s(\mathfrak{w})) \qquad \text{für} \quad \mathfrak{w} \in (\Delta^*)^r.$$

Wenn es Turingmaschinen für $f_1, \ldots, f_s$ gibt, dann gibt es auch eine Turingmaschine für $[f_1, \ldots, f_s]$. Denn durch Komposition erhalten wir $\mathfrak{w} \mapsto (f_1(\mathfrak{w})) \cdot \mathfrak{w} \mapsto \cdots \mapsto (f_1(\mathfrak{w}), \ldots, f_s(\mathfrak{w})) \cdot \mathfrak{w} \mapsto (f_1(\mathfrak{w}), \ldots, f_s(\mathfrak{w}))$.

Schleife mit Abbruch beim leeren Wort Für $1 \leq k \leq n$ sei die Projektion $P_k^n\colon (\Delta^*)^n \to \Delta^*$ definiert durch $P_k^n(w_1, \ldots, w_n) := w_k$. Für $n \in \mathbb{N}$ und $f\colon (\Delta^*)^{n+1} \xrightarrow{\text{part}} (\Delta^*)^{n+1}$ sei die partielle Funktion $\text{Schl}_n(f)\colon (\Delta^*)^{n+1} \xrightarrow{\text{part}} (\Delta^*)^{n+1}$ folgendermaßen definiert. Für alle $\mathfrak{w} \in (\Delta^*)^{n+1}$ gelte

- Wenn ein $k \in \mathbb{N}$ exisitiert mit $P_{n+1}^{n+1}(f^k(\mathfrak{w})) = \varepsilon$, so ist $\mathrm{Schl}_n(f)(\mathfrak{w}) := f^r(\mathfrak{w})$, wobei r das kleinste solche k ist.
- Andernfalls ist $\mathrm{Schl}_n(f)(\mathfrak{w})$ undefiniert.

Wenn T eine Turingmaschine für f ist, dann ist

$$\begin{array}{cccc} S & \# & L & q \\ q & a & (RTL) & q \\ q & \# & R & H \end{array} \qquad (a \in \Delta)$$

eine Turingmaschine für $\mathrm{Schl}_n(f)$. Die Wirkung dieser Turingmaschine entspricht der eines Pseudocodes

while $w_{n+1} \neq \varepsilon$
do $(w_1, \ldots, w_{n+1}) := f(w_1, \ldots, w_{n+1})$

Als Beispiel für $\mathrm{Schl}_n(f)$ betrachten wir etwa $\Delta = \{|\}$ und die folgendermaßen definierte Funktion $f\colon (\Delta^*)^2 \to (\Delta^*)^2$

$$f(u, \varepsilon) := (u, \varepsilon)$$
$$f(u, x|) := (uu, x)$$

für $u, x \in \Delta^*$. Dann ist

$$f(|^m, |^n) = (|^{2m}, |^{n-1}) \qquad \text{für } m, n \in \mathbb{N},\ n > 0.$$

Für $k \leq n$ ist $f^k(|^m, |^n) = (|^{m \cdot 2^k}, |^{n-k})$ und daher $P_2^2(f^k(|^m, |^n)) = |^{n-k}$. Das kleinste k mit $P_2^2(f^k(|^m, |^n)) = \varepsilon$ ist daher n. Also ist

$$\mathrm{Schl}_1(f)(|^m, |^n) = f^n(|^m, |^n) = (|^{m \cdot 2^n}, \varepsilon).$$

Ein Pseudocode, der die Funktion $\mathrm{Schl}_1(f)$ berechnet, ist etwa der folgende.

```
u := |^m
x := |^n
while x ≠ ε do
    ⌜ Entferne einen Strich aus x
      u := uu ⌟
return (u,x)
```

Vorgängerfunktion

$$\begin{array}{cccc} S & \# & L & q \\ q & a & \# & H \\ q & \# & R & H \end{array} \qquad (a \in \Delta)$$

ist eine Turingmaschine für die Funktion $V\colon \Delta^* \to \Delta^*$ mit $V(wa) := w$ für $w \in \Delta^*$, $a \in \Delta$ und $V(\varepsilon) := \varepsilon$. Im Fall $\Delta = \{|\}$ entspricht die Funktion V gerade der Vorgängerfunktion auf der Menge der natürlichen Zahlen.

1.2 Churchsche These

Beim Versuch den Begriff der (partiellen) Berechenbarkeit von (partiellen) Funktionen zu formalisieren hat man verschiedene formale Berechnungssysteme aufgestellt, darunter die folgenden Systeme.

- μ-partiellrekursive Funktionale
- Turingmaschinen
- Grammatiken
- λ-Kalkül
- Kombinatorenkalkül
- Markov-Algorithmen
- Termersetzungssysteme (rewrite systems)
- Post-Systeme
- Universelle Programmiersprachen (C, Lisp, Prolog, Smalltalk, ...)

Einige dieser Systeme verfolgen voneinander grundverschiedene Ansätze. Trotzdem hat sich gezeigt, daß alle diese Systeme den gleichen (partiellen) Berechenbarkeitsbegriff beschreiben, also zueinander äquivalent sind. Eine in einem dieser Systeme (und damit in allen dieser Systeme) beschreibbare (partielle) Funktion heißt *(partiell)rekursiv*.

Church und Turing haben nun die unter dem Namen 'Churchsche These' bekannte These aufgestellt, daß diese Kalküle bereits alle partiell berechenbaren Funktionen erfassen.

Churchsche These Eine partielle Funktion ist genau dann partiell berechenbar, wenn sie partiellrekursiv ist.

Die Churchsche These ist keine mathematisch formulierbare Aussage und daher nicht mathematisch beweisbar. Sie sagt aus, daß die aufgestellten Beschreibungssysteme für partiell berechenbare Funktionen adäquate mathematische Beschreibungen für den intuitiven Begriff der paritellen Berechenbarkeit darstellen.

Für totale Funktionen besagt die Churchsche These, daß eine totale Funktion genau dann berechenbar ist, wenn sie rekursiv ist.

1.3 Rekursiv aufzählbare und rekursive Mengen

Definition 19 Eine Menge $A \subseteq \mathbb{N}^n$ oder $A \subseteq (\Delta^*)^n$ heißt *rekursiv aufzählbar*, wenn sie der Definitionsbereich einer partiellrekursiven Funktion ist. Sie heißt *rekursiv*, wenn ihre charakteristische Funktion χ_A rekursiv ist.

Definition 20 Eine Menge $A \subseteq \mathbb{N}^n$ oder $A \subseteq (\Delta^*)^n$ heißt *semientscheidbar*, wenn es einen Algorithmus gibt, der bei Eingabe von $\mathfrak{a} \in \mathbb{N}^n$ bzw. $\mathfrak{a} \in (\Delta^*)^n$ genau dann hält, wenn $\mathfrak{a} \in A$ ist. Sie heißt *entscheidbar*, wenn ihre charakteristische Funktion χ_A berechenbar ist.

Nach der Churchschen These ist eine Menge genau dann semientscheidbar, wenn sie rekursiv aufzählbar ist, und genau dann entscheidbar, wenn sie rekursiv ist.

Jede rekursive Menge ist rekursiv aufzhlbar, aber nicht umgekehrt. Es gilt der Satz (hier ohne Beweis):

Satz 6 *Eine Menge $A \subseteq \mathbb{N}$ ist genau dann rekursiv, wenn A und $\mathbb{N} \setminus A$ rekursiv aufzählbar sind.*

und entsprechend fr $A \subseteq (\Delta^*)$.

Definition 21 Sei T eine Turingmaschine mit Bandalphabet Σ und $\Delta \subseteq \Sigma \setminus \{\#\}$. Man sagt, T *akzeptiere* ein Wort $w \in \Delta^*$, wenn T bei Start mit der Konfiguration $w\underset{S}{\#}$ anhält, d.h. nach endlich vielen Schritten in den Haltezustand übergeht. Die Berechnung, die T dabei ausführt heißt *akzeptierende Berechnung* durch T für w. Die von T *akzeptierte Sprache* ist die Menge der von T akzeptierten Wörter.

Satz 7 *Eine Sprache wird genau dann von einer Turingmaschine akzeptiert, wenn sie rekursiv aufzählbar ist.*

Beweis: $\Leftarrow$: Sei L rekursiv aufzählbar. Dann ist L Definitionsbereich einer partiellrekursiven Funktion f. Daher wird f partiell durch eine Turingmaschine T berechnet. Die Sprache L wird dann von T akzeptiert.

$\Rightarrow$: Sei T eine Turingmaschine, die die Sprache L akzeptiert. Wenn T eine partielle Funktion f partiell berechnet, dann ist f partiellrekursiv und L der Definitionsbereich von f. Also ist L rekursiv aufzählbar.

Andernfalls konstruiert man aus T eine neue Turingmaschine T', die die gleiche Sprache L akzeptiert und eine partielle Funktion partiell berechnet. Hierzu fügt man ein neues Zeichen $\#'$ zum Bandalphabet hinzu und ersetzt jede Zeile $q\#bp$ durch die beiden Zeilen $q\#(\#'b)p$ und $q\#'bp$. Sei T'' die so entstandene Turingmaschine und $T' := T''\#'R_{\#}$. q.e.d.

Definition 22 Eine Sprache $L \subseteq \Delta^*$ heißt *Ausgabesprache* einer Turingmaschine T, wenn

$$L = \{v \in \Delta^* \mid \bigvee_{u \in \Delta^*} u\underline{\#} \Rrightarrow_T v\underline{\#}\}.$$

Satz 8 *Eine Sprache ist genau dann Ausgabesprache einer Turingmaschine, wenn sie rekursiv aufzählbar ist.*

Zum Beweis bemerkt man, daß jede rekursiv aufzählbare Sprache L der Wertebereich einer partiellrekursiven Funktion ist. Diese Funktion wird von einer Turingmaschine partiell berechnet und die Ausgabesprache dieser Turingmaschine ist dann die Sprache L.

Wenn umgekehrt L die Ausgabesprache einer Turingmaschine T ist, dann kann man T so zu einer Turingmaschine T' abändern, daß T' dasselbe Ergebnis wie T liefert, wenn T ein Wort aus Δ^* liefert, und andernfalls nicht hält. Die Turingmaschine T' berechnet dann partiell eine partielle Funktion mit Wertebereich L. Daher ist L dann rekursiv aufzählbar.

Definition 23 Man sagt, eine Sprache $L \subseteq \Delta^*$ werde von einer Turingmaschine T *aufgezählt*, wenn es einen Zustand q von T gibt so, daß gilt

$$L = \{w \in \Delta^* \mid \bigvee_{u \in \Delta^*} \underset{S}{\#} \Rightarrow_T^* \underset{q}{u\#w\#}\}.$$

Satz 9 *Eine Sprache wird genau dann von einer Turingmaschine aufgezählt, wenn sie rekursiv aufzählbar ist.*

Allgemein ist eine von irgendeinem formalen System akzeptierte oder erzeugte Sprache immer rekursiv aufzählbar (nach der Churchschen These).

1.4 Varianten von Turingmaschinen

Im Beweis der partiellen Turingberechenbarkeit der partiellrekursiven Funktionen werden bei allen Berechnungen durch eine Turingmaschine die Felder links von den Eingabewörtern nicht berührt. Wir können uns also auf ein einseitig (links) begrenztes Band beschränken. Andererseits kann man den Begriff der Turingmaschine erweitern. In der Literatur wird oft der Begriff der Turingmaschine so verwendet, dass in jedem Schritt zunchst auf das Arbeitsfeld geschrieben wird und dann der Schreib-Lese-Kopf nach rechts oder nach links bewegt wird. Diese Variante des Begriffs der Turingmaschine erlaubt die gleichen partiellen Funktionen zu berechnen und die gleichen Sprachen zu akzeptieren wie die oben eingefhrte Definition. Es gibt aber noch viele andere Varianten:

Die folgenden Maschinen können einander gegenseitig simulieren

- Turingmaschinen (mit beidseitig unbegrenztem Band)
- Turingmaschinen mit einseitig begrenztem Band
- Turingmaschinen mit mehreren Bändern
- Turingmaschinen mit mehreren Köpfen
- Turingmaschinen mit mehrdimensionalen Bändern

sowie alle Kombinationen dieser Erweiterungen.

1.4.1 Nichtdeterministische Turingmaschinen

Die bisher definierten Turingmaschinen werden auch *deterministische Turingmaschinen* genannt, da für jede Konfiguration der nächste Berechnungsschritt eindeutig festgelegt ist und damit zu jeder Anfangskonfiguration die gesamte Berechnung eindeutig bestimmt (determiniert) ist. Bei einer *nichtdeterministischen Turingmaschine* hat die Maschine bei jedem Berechnungsschritt die Wahl zwischen endlich vielen Wahlmöglichkeiten.

Definition 24 Eine *nichtdeterministische Turingmaschine* ist ein Quintupel $(Q, \Sigma, S, H, \Delta)$ mit

1.–4. wie für (deterministische) Turingmaschinen

5′. *Übergangsrelation* $\Delta \subseteq (Q \setminus \{H\}) \times \Sigma \times B \times (Q \setminus \{S\})$.

Eine Zeile einer solchen nichtdeterministischen Turingmaschine ist dann $q\ a\ b\ p$ mit $(q, a, b, p) \in \Delta$. Hier können auf eine Konfiguration null, ein oder mehr Zeilen anwendbar sein. Entsprechend kann es zu einer Anfangskonfiguration mehrere verschiedene mögliche Berechnungen geben.

Definition 25 Man sagt, eine nichtdeterministische Turingmaschine T *akzeptiere* ein Wort w, wenn es eine akzeptierende Berechnung durch T für w gibt. Die von T *akzeptierte Sprache* ist die Menge der von T akzeptierten Wörter.

Man kann jede nichtdeterministische Turingmaschine durch eine deterministische Turingmaschine in dem Sinne „simulieren“, daß man etwa mit breadth first search alle möglichen Berechnungen der nichtdeterministischen Turingmaschine nachvollzieht — ähnlich, wie wir dies für Grammatiken gemacht haben. Insbesondere gilt daher

Satz 10 *Jede von einer nichtdeterministischen Turingmaschine akzeptierte Sprache ist rekursiv aufzählbar.*

Die Simulation kann aber sehr viel länger werden als die ursprüngliche Berechnung. Wenn zum Beispiel die nichtdeterministische Turingmaschine bei jeder Konfiguration 2 Wahlmöglichkeiten hat und die Berechnung die Länge n hat, so muß man schlimmstenfalls 2^n Berechnungen durchsuchen.

1.4.2 Terminierung von Computerprogrammen

Definition 26 Eine Programmiersprache heißt *universell*, wenn sich darin jede partiellrekursive Funktion durch ein Programm implementieren läßt.

Zu den universellen Programmiersprachen gehören alle gängigen allgemeinen Programmiersprachen wie C, Lisp, Prolog, Smalltalk, usw. Nicht dazu gehören einige spezialisierte Sprachen, z.B. einige Datenbanksprachen und die Macro-Sprachen einiger Texteditoren.

Das Cantorsche Diagonalverfahren läßt sich auch auf jede universelle Programmiersprache anwenden. So kann ein Programm P, das einen String einliest, auch sich selbst einlesen. Wir definieren

$$P \text{ ist selbstterminierend} : \Longleftrightarrow P \text{ terminiert bei Eingabe } P$$

Für eine universelle Programmiersprache gilt

Satz 11 *Die Terminierung von Programmen ist unentscheidbar.*

Denn andernfalls wäre nach der Churchschen These die Menge der terminierenden Paare (Programm, Eingabe) rekursiv. Da die Programmiersprache universell ist, gäbe es dann ein Programm, das feststellen kann, ob ein beliebig gegebenes Programm bei einem beliebig gegebenen Eingabewort terminiert. Dann könnte man aber auch ein Programm P_0 schreiben, das das Folgende tut.

> Lies P.
> Wenn P selbstterminierend ist, dann gehe in eine Endlosschleife.
> Andernfalls brich ab.

Dann gälte

$$\begin{aligned} P_0 \text{ ist selbstterminierend} &\Longleftrightarrow P_0 \text{ terminiert bei der Eingabe } P_0 \\ &\Longleftrightarrow P_0 \text{ ist nicht selbstterminierend} \end{aligned}$$

Dies wäre ein Widerspruch.

1.4.3 Weitere unentscheidbare Probleme

Die folgenden Probleme sind unentscheidbar

- Allgemeingültigkeit prädikatenlogischer Formeln
- Terminierung von Computerprogrammen
- Korrektheit von Computerprogrammen
- Frage, ob ein gegebenes Wort im Sprachschatz einer gegebenen Grammatik liegt

Es gibt

- Turingmaschinen mit unentscheidbarer akzeptierter Sprache
- Turingmaschinen mit unentscheidbarer erzeugter Sprache
- Grammatiken mit unentscheidbarem Sprachschatz
- Computerprogramme, von denen nicht entscheidbar ist, für welche Eingaben sie terminieren

2 Komplexitätstheorie

2.1 Einleitung

Die *Komplexitätstheorie* befaßt sich mit dem Resourcenverbrauch von Algorithmen. Die wichtigsten Resourcen sind *Zeit* (engl. *time*) und *Speicherplatz* (engl. *space*). Entsprechend unterscheidet man zwischen *Zeit-* und *Speicherplatzkomplexität*.

Man interessiert sich insbesondere dafür, wie schnell der Resourcenverbrauch mit der Größe des Problems ansteigt. Hierzu verwendet man die *Landausymbole* O und o. Wenn $f\colon \mathbb{N} \to \mathbb{N}$, dann ist

$$O(f) := \{g\colon \mathbb{N} \to \mathbb{N} \mid \bigvee_{c\in\mathbb{R}^+} \bigvee_{n_0\in\mathbb{N}} \bigwedge_{n\geq n_0} g(n) \leq c \cdot f(n)\}$$
$$o(f) := \{g\colon \mathbb{N} \to \mathbb{N} \mid \bigwedge_{c\in\mathbb{R}^+} \bigvee_{n_0\in\mathbb{N}} \bigwedge_{n\geq n_0} g(n) \leq c \cdot f(n)\}$$

Statt $g \in O(f)$ schreibt man meistens $g(n) = O(f(n))$ und statt $g \in o(f)$ meistens $g(n) = o(f(n))$.

Beispiel 21

$$\sum_{k=1}^{n} k = O(n^2)$$
$$\sum_{k=1}^{n} \frac{1}{k} = O(\log n)$$

Bei $O(\log n)$ braucht man die Basis des Logarithmus nicht anzugeben, da es bei den Landausymbolen auf einen konstanten Faktor nicht ankommt und da

$$\log_a n = \log_a b \cdot \log_b n$$

ist.

Man unterscheidet weiter zwischen dem Anwachsen der Resourcen im *schlimmsten Fall* (engl. *worst case*), im *Mittel* (engl. *average case*) und im *besten Fall* (engl. *best case*).

Beispiel 22
Algorithmus zum Suchen eines Elementes x in einer sortierten Liste von n Elementen.

1. Vergleiche x mit dem Element y in der Mitte der Liste.
2. Wenn $x < y$, suche in der Teilliste vor y.
3. Wenn $x = y$, Element gefunden.
4. Wenn $x > y$, suche in der Teilliste hinter y.

Dieser rekursiv definierte Algorithmus hat Zeitkomplexität

- $O(\log n)$ im schlimmsten Fall
- $O(\log n)$ im Mittel
- $O(1)$ im besten Fall.

und Speicherplatzkomplexität $O(n)$.

Beispiel 23
Der Algorithmus Quicksort zum Sortieren einer Liste von n Elementen hat Zeitkomplexität

- $O(n^2)$ im schlimmsten Fall
- $O(n \cdot \log n)$ im Mittel

Offenbar hängen Zeitkomplexität und Speicherplatzkomplexität eines Algorithmus vom *Maschinenmodell* ab. Es gibt Probleme, die sich mit einer (Einband-)Turingmaschine nur mit Zeitkomplexität $O(n^2)$, aber mit einer Mehrband-Turingmaschine mit Zeitkomplexität $O(n)$ lösen lassen.

Man sagt, ein Algorithmus habe *polynomielle Zeitkomplexität*, wenn es ein Polynom p gibt so, daß die Rechenzeit bei Eingabe eines Wortes einer Länge n höchstens $p(n)$ beträgt.

Wenn ein Algorithmus in einem der bekannten Maschinenmodelle polynomielle Zeitkomplexität hat, dann hat er auch auf einer geeigneten Turingmaschine polynomielle Zeitkomplexität.

Denn die Realisierung eines Algorithmus in einem Maschinenmodell läßt sich durch eine Turingmaschine polynomiell simulieren, d.h. es gibt ein Polynom q so, daß, wenn der Algorithmus im Maschinenmodell eine Zeit t benötigt, die Turingmaschine eine Zeit $\leq q(t)$ benötigt. Wenn der Algorithmus im Maschinenmodell eine Zeit $\leq p(n)$ benötigt, dann benötigt die Turingmaschine eine Zeit $\leq q(p(n))$. Und für zwei Polynome p und q ist auch $q \circ p$ ein Polynom.

2.2 P und NP

Definition 27 Eine deterministische oder nichtdeterministische Turingmaschine T *akzeptiert* eine Sprache $L \subseteq \Delta^*$ *in polynomieller Zeit*, wenn es ein Polynom p gibt so, daß gilt

1. T akzeptiert jedes Wort $w \in L$ in höchstens $p(|w|)$ Berechnungsschritten.
2. T akzeptiert kein Wort aus $\Delta^* \setminus L$.

Definition 28

- P ist die Klasse aller Sprachen, die in polynomieller Zeit von einer deterministischen Turingmaschine akzeptiert werden.
- NP ist die Klasse aller Sprachen, die in polynomieller Zeit von einer nichtdeterministischen Turingmaschine akzeptiert werden.

Definition 29 Unter einem *nichtdeterministischen Algorithmus* versteht man ein formalisierbares Rechenverfahren, bei dem — im Gegensatz zum (deterministischen) Algorithmus — bei jedem Berechnungsschritt noch eine Wahl zwischen mehreren Möglichkeiten offengelassen werden darf.

Definition 30 Sei A ein nichtdeterministischer Algorithmus.

- A *akzeptiert* ein Wort w, wenn es für A eine haltende Berechnung bei Eingabe w gibt.
- Die von A *akzeptierte Sprache* ist die Menge der von A akzeptierten Wörter.
- A *akzeptiert eine Sprache* $L \subseteq \Delta^*$ *in polynomieller Zeit*, wenn es ein Polynom p gibt so, daß gilt
  1. A akzeptiert jedes Wort $w \in L$ in höchstens $p(|w|)$ Berechnungsschritten.
  2. A akzeptiert kein Wort aus $\Delta^* \setminus L$.

Eine Sprache L ist genau dann in P, wenn es einen (deterministischen) Algorithmus gibt, der von einem Wort in polynomieller Zeit entscheiden kann, ob es in L ist.

Eine Sprache L ist genau dann in NP, wenn sie von einem nichtdeterministischen Algorithmus in polynomieller Zeit akzeptiert wird.

Beispiel 24

- $\{a^n b^n \mid n \in \mathbb{N}\} \in P$.
- $\{d_n \# d_j \# d_k \mid n, j, k \in \mathbb{N} \wedge \bigvee_{t \in \mathbb{N}} (j \leq t \leq k \wedge t | n)\} \in NP$, wobei d_n die Dezimaldarstellung von n sei für $n \in \mathbb{N}$.

  Z.B. ist das Wort 38#17#20 in dieser Menge, da 19|38 und $17 \leq 19 \leq 20$ ist.

Eine Sprache kann man auch als Problem formulieren. Im letzten Beispiel:

Gegeben $n, j, k \in \mathbb{N}$. Frage: Gibt es ein $t \in \mathbb{N}$ mit $j \leq t \leq k$ und $t | n$.

Man sagt auch, dieses Problem sei ein *NP-Problem*. Falls es auch ein P-Problem ist, dann ist Primfaktorzerlegung in polynomieller Zeit möglich. Bis heute weiß man nicht, ob Primfaktorzerlegung in polynomieller Zeit möglich ist.

Es gilt $P \subseteq NP$.

2.3 Das $P = NP$-Problem

Das wohl bekannteste bisher ungelöste Problem der Informatik ist das

$P = NP$-**Problem:** Gilt $P = NP$?

Man weiß bis heute nicht, ob die Antwort auf diese Frage „ja“ oder „nein“ ist. Eine Antwort „ja“ würde bedeuten, daß eine ganze Reihe von schwierigen Problemen der Informatik, darunter das Erfüllbarkeitsproblem der Aussagenlogik und das Problem des Handelsreisenden, in polynomieller Zeit lösbar wären, daß es dafür also (einigermaßen) effiziente Algorithmen gäbe. Bisher hat man aber für keines dieser Probleme einen in polynomieller Zeit terminierenden Algorithmus gefunden. Daher vermuten die meisten Informatiker, daß es einen solchen Algorithmus nicht gibt, daß also $P \neq NP$ ist. Andererseits konnte man trotz intensivster Bemühungen bisher keinen Beweis dafür finden, daß $P \neq NP$ ist.

2.4 Das Erfüllbarkeitsproblem der Aussagenlogik

Gegeben seien abzählbar unendlich viele Zeichen $U_1, U_2, \ldots$, genannt *Aussagenvariable*.

Definition 31 (Induktive Definition der Formeln)

1. Jede Aussagenvariable U ist eine Formel.
2. Ist F eine Formel, dann ist auch $\neg F$ eine Formel.
3. Sind F und G Formeln, dann sind auch $(F \wedge G)$ und $(F \vee G)$ Formeln.

Definition 32 Eine *Belegung* einer Formel F ist eine Abbildung i von der Menge der in F auftretenden Aussagenvariablen in die Menge der Wahrheitswerte $\{w, f\}$.

Definition 33 Der *Wahrheitswert* iF einer Formel F bei einer Belegung i ist definiert durch

1. $iU := i(U)$
2. $i\neg F := \begin{cases} w, & \text{wenn } iF = f \\ f & \text{sonst} \end{cases}$
3. $i(F \wedge G) := \begin{cases} w, & \text{wenn } iF = w \text{ und } iG = w \\ f & \text{sonst} \end{cases}$
4. $i(F \vee G) := \begin{cases} w, & \text{wenn } iF = w \text{ oder } iG = w \\ f & \text{sonst} \end{cases}$

Definition 34 Ein *Modell* einer Formel F ist eine Belegung i mit $iF = w$. Eine Formel heißt *erfüllbar*, wenn sie ein Modell hat.

Jede Formel kann als Wort über $\Delta := \{U, 0, \ldots, 9, (,), \neg, \wedge, \vee\}$ geschrieben werden.

Satz 12 *Es gibt eine Turingmaschine, die zu einem Wort $F \in \Delta^*$ der Länge n in $O(2^n)$ Schritten entscheiden kann, ob F eine erfüllbare Formel ist.*

Dies geht mit dem folgenden Algorithmus:

1. Wenn F nicht Formel, dann brich ab mit 'nein'.
2. Suche Modell von F.
3. Gib aus, ob Modell gefunden.

Schritt 1 geht in polynomieller Zeit.
F hat höchstens $\frac{n+1}{2}$ Aussagenvariable und daher höchstens $2^{\frac{n+1}{2}}$ Belegungen. Überprüfung, ob Belegung Modell ist, geht in polynomiell vielen Schritten und daher in Zeit $O(2^{\frac{n-1}{2}})$. Also geht Schritt 2 in Zeit $O(2^n)$.
Schritt 3 geht in Zeit $O(1)$.
Insgesamt braucht der Algorithmus also eine Zeit $O(2^n)$.
Es ist nicht schwierig (aber recht mühsam) diesen Algorithmus in Form einer (deterministischen) Turingmaschine zu formulieren, die zu einem Wort $F \in \Delta^*$ der Länge n in $O(2^n)$ Schritten entscheiden kann, ob F eine erfüllbare Formel ist.

Satz 13 *Die Menge der erfüllbaren Formeln liegt in NP.*

Denn der nichtdeterministische Algorithmus

1. Wenn F nicht Formel, halte nicht.
2. Rate eine Belegung i von F.
3. Wenn i Modell von F, dann halte, sonst halte nicht.

akzeptiert die Menge der erfüllbaren Formeln in polynomieller Zeit. Dieser nichtdeterministische Algorithmus läßt sich auch als nichtdeterministische Turingmaschine formulieren.

2.5 NP-Vollständigkeit

Definition 35 Eine totale Funktion $f\colon \Delta^* \to \Delta^*$ heißt *polynomialzeitberechenbar*, wenn es eine (deterministische) Turingmaschine T und ein Polynom p gibt so, daß gilt

1. T berechnet f.
2. Wenn $w \in \Delta^*$ und $|w| = n$, dann hat die Berechnung durch T bei Eingabe w höchstens $p(n)$ Schritte.

Definition 36 Eine Sprache $L' \subseteq \Delta^*$ ist auf eine Sprache $L \subseteq \Delta^*$ *polynomialzeitreduzierbar* (in Zeichen $L' \leq_p L$), wenn es eine polynomialzeitberechenbare Funktion f gibt so, daß für alle $w \in \Delta^*$ gilt

$$w \in L' \iff f(w) \in L.$$

Satz 14 $L_1 \leq_p L_2 \wedge L_2 \leq_p L_3 \implies L_1 \leq_p L_3$.

Satz 15 $L \in P \wedge L' \leq_p L \implies L' \in P$.

Dasselbe gilt für NP.

Definition 37 Eine Sprache L ist *NP-hart*, wenn jede Sprache aus NP auf L polynomialzeitreduzierbar ist. L ist *NP-vollständig*, wenn $L \in NP$ ist und L NP-hart ist.

Satz 16 *Wenn eine NP-vollständige Sprache in P liegt, dann ist $P = NP$.*

Beweis: Sei $L \in P$ NP-vollständig und $L' \in NP$. Dann ist $L' \leq_p L$ und daher nach dem letzten Satz $L' \in P$. q.e.d.

2.6 NP-Vollständigkeit von SAT

Definition 38

- Ein *Literal* ist eine Aussagenvariable U oder die Negation $\neg U$ einer Aussagenvariablen.
- Eine *Klausel* ist eine Disjunktion $L_1 \vee \cdots \vee L_k$ von Literalen.
- Eine *konjunktive Normalform* (*KNF*) ist eine Konjunktion von Klauseln:

$$(L_{11} \vee \cdots \vee L_{1k_1}) \wedge \cdots \wedge (L_{j1} \vee \cdots \vee L_{jk_j})$$

Definition 39 SAT ist die Menge der erfüllbaren KNFen.

Lemma 12 *$SAT \in NP$.*

Lemma 13 *SAT ist NP-hart.*

Beweis: (Nur Idee): Sei $L \in NP$. Zu zeigen: $L \leq_p$ SAT.
Es gibt eine nichtdeterministische Turingmaschine T, die L in polynomieller Zeit akzeptiert. Jedes $w \in L$ wird in höchstens $p(n)$ Schritten akzeptiert ($n := |w|$).
Wir repräsentieren eine Berechnung B durch T als Belegung i.
Wir ordnen jeder möglichen Eingabe w eine aussagenlogische Formel $F = f(w)$ zu so, daß gilt

i ist Modell von $f(w)$
$\iff$ B ist Berechnung durch T für Eingabe w in höchstens $p(n)$ Schritten

(wobei $n := |w|$). Hierzu betrachten wir alle Zeitpunkte t, alle Felder f (Feld 0 ist das Feld, das am Anfang der Berechnung das Arbeitsfeld war, links davon die Felder mit negativer Nummer f und rechts die Felder mit positiver Nummer f), alle Zustände und alle Zeichen, die in einer Berechnung mit Eingabe w eine Rolle spielen können. Es sind dies

- t Zeit $(0 \leq t \leq p(n))$
- f Feld $(-p(n) \leq f \leq p(n))$
- q Zustand (alle Zustände der Turinmaschine)
- z Zeichen (alle Zeichen des Bandalphabets)

Für jedes dieser t, f, q und z führen wir die folgenden Aussagenvariablen ein, denen wir uns, falls eine Berechnung vorliegt, jeweils die hinter dem Doppelpunkt stehende Aussage als Bedeutung zugeordnet denken.

- Q_{tq}: Zur Zeit t ist der Zustand q.
- A_{tf}: Zur Zeit t ist Feld f das Arbeitsfeld.
- Z_{tfz}: Zur Zeit t steht auf Feld f das Zeichen z.

Man kann dann die Aussage

Es liegt eine Berechnung vor, die w in höchstens $p(n)$ Schritten akzeptiert

als KNF F formulieren, die genau dann erfüllbar ist, wenn eine solche Berechnung existiert.

$$w \in L \iff F \in \mathrm{SAT}.$$

Wenn zum Beispiel $p\ a\ b\ q$ die einzige Zeile ist, die mit $p\ a$ beginnt, dann könnte man diese Zeile durch die folgenden Implikationen ausdrücken.

$$\begin{aligned} Q_{tp} \wedge A_{tf} \wedge Z_{tfa} &\to Q_{t+1,q} \\ Q_{tp} \wedge A_{tf} \wedge Z_{tfa} &\to A_{t+1,f} \\ Q_{tp} \wedge A_{tf} \wedge Z_{tfa} &\to Z_{t+1,f,b} \\ Q_{tp} \wedge A_{tf} \wedge Z_{tfa} \wedge Z_{tf'x} &\to Z_{t+1,f',x} \qquad (f' \neq f). \end{aligned}$$

Für jedes zulässige t, p, f, f' und x hat man eine solche Implikation. Jede dieser Implikationen läßt sich wieder als Klausel schreiben, zum Beispiel die erste Implikation als

$$\neg Q_{tp} \vee \neg A_{tf} \vee \neg Z_{tfa} \vee Q_{t+1,q}.$$

Entsprechend bekommt man auch Klauseln für alle anderen Zeilen der nichtdeterministischen Turingmaschine T. Zusätzlich zu den Klauseln, die den Zeilen der Turingmaschine entsprechen und die die Abarbeitung durch die Turingmaschine beschreiben, muß man noch Klauseln angeben, die ausdrücken, daß die Turingmaschine mit dem Wort w im

Anfangszustand startet und daß sie sich nach höchstens $p(n)$ Schritten im Haltezustand befindet. Man muß auch explizit angeben, daß eine Turingmaschine nicht gleichzeitig in zwei Zuständen sein kann und daß nicht auf einem Feld gleichzeitig zwei Zeichen stehen dürfen.

Die sich schließlich ergebende Formel F ist zwar recht kompliziert hinzuschreiben, aber ihre Länge ist höchstens ein Polynom von der Länge n der Eingabe w der Turingmaschine T. Darüberhinaus läßt sie sich aus w in polynomieller Zeit berechnen (auch mit einer deterministischen Turingmaschine, obwohl es sehr aufwendig wäre diese Turingmaschine hinzuschreiben).

Es gilt also: $f : w \mapsto F$ ist polynomialzeitberechenbar. Also ist $L \leq_p$ SAT. Da dies für jede Sprache $L \in NP$ gilt, ist SAT NP-hart. q.e.d.

Satz 17 *SAT ist NP-vollständig.*

2.7 Nachweis der NP-Vollständigkeit einer Sprache durch Polynomialzeitreduktion

Wir haben gesehen, daß es recht aufwendig ist von einer Sprache nachzuweisen, daß sie NP-vollständig ist. Wenn man aber bereits von einer Sprache L weiß, daß sie NP-vollständig ist, dann kann man dieses Wissen dazu verwenden auch die NP-Vollständigkeit anderer Sprachen nachzuweisen. Hierzu verwendet man den folgenden Satz.

Satz 18 *Sei L NP-vollständig, $L' \in NP$ und $L \leq_p L'$. Dann ist L' NP-vollständig.*

Beweis: Da $L' \in NP$ vorausgesetzt wurde, ist noch zu zeigen, daß L' NP-hart ist, d.h. daß jede Sprache L'' aus NP auf L' polynomialzeitreduzierbar ist. Hierzu nehmen wir an $L'' \in NP$. Da L NP-vollständig ist, gilt $L'' \leq_p L$. Nach Voraussetzung ist $L \leq_p L'$. Wegen der Transitivität von $\leq_p$ ist daher $L'' \leq_p L'$. q.e.d.

Hieraus ergibt sich die folgende

Methode zum Nachweis der NP-Vollständigkeit einer Sprache: Um zu zeigen, daß eine Sprache L' NP-vollständig ist, zeigt man, daß $L' \in NP$ ist und daß irgendeine bekanntermaßen NP-vollständige Sprache L auf L' polynomialzeitreduzierbar ist.

Beispiel 25 (Das Cliquenproblem)
Gegeben ein ungerichteter Graph G und $n \in \mathbb{N}$. Frage: Gibt es einen vollständigen Teilgraphen ('Clique') von G mit n Ecken?

Man kann zu jeder KNF F sehr leicht einen Graphen G und eine natürliche Zahl n angeben so, daß F genau dann erfüllbar ist, wenn es in G eine Clique aus n Ecken gibt (hier ohne Beweis). Die Berechnung von G und n aus F geht in Polynomialzeit.

SAT ist also polynomialzeitreduzierbar auf das Cliquenproblem. Daher ist auch das Cliquenproblem NP-vollständig.

Weitere NP-vollständige Probleme:

- Problem des Handelsreisenden
- Rucksackproblem
- ...

Wenn eines dieser Probleme in P ist, dann sind alle NP-vollständigen Probleme (und überhaupt alle NP-Probleme) in P.

3 Formale Semantik

Die Semantik ist die Lehre von der Bedeutung. Bei der Semantik von Programmiersprachen unterscheidet man zwischen *deklarativer* oder *denotationeller Semantik* und *prozeduraler* oder *operationaler Semantik*. Erstere befaßt sich mit der Wirkung eines Programms als Ganzes, letztere mit der Art und Weise, wie ein Programm in einzelnen Schritten abgearbeitet wird.

3.1 Semantik in der logischen Programmierung

Deklarative Semantik wird in der logischen Programmierung durch den Begriff der *korrekten Antwortsubstitution* und den Begriff des *kleinsten Herbrandmodells* gegeben.

Bei definiten Klauseln (=Hornklauseln) ist θ eine *korrekte Antwortsubstitution* für ein Programm P und eine Zielklausel $\leftarrow A_1, \ldots, A_k$ genau dann, wenn

$$P \models \forall[(A_1 \wedge \cdots \wedge A_k)\theta].$$

Eine *Herbrandinterpretation* ist eine Interpretation, deren Individuenbereich die Menge der Grundterme ist und die jeden Grundterm durch sich selbst interpretiert. Eine Herbrandinterpretation ist eindeutig bestimmt durch Angabe der Menge der in dieser Interpretation geltenden Grundatomformeln. Es wird daher oft mit dieser Menge identifiziert. Insbesondere sagt man, eine Herbrandinterpretation I sei *kleiner* als eine andere J, wenn jede in I geltende Grundatomformel auch in J gilt. Ein *Herbrandmodell* einer Formel ist eine Herbrandinterpretation, die Modell dieser Formel ist.

Kleinstes Herbrandmodell M_P von P ist die Menge der Grundatomformeln, die semantisch aus P folgen.

Prozedurale Semantik: Für eine Herbrandinterpretation I sei $T_P(I)$ die Menge der Grundatomformeln, die sich mit P in einem Schritt aus Formeln aus I ableiten lassen.

Äquivalenz von deklarativer und prozeduraler Semantik: M_P ist der kleinste Fixpunkt von T_P.

Bei nichtdefiniten Klauseln

$$\mathrm{comp}(P) \models \forall[(A_1 \wedge \cdots \wedge A_k)\theta].$$

$\mathrm{comp}(P)$ ist die *Clarksche Vervollständigung* von P.

Z.B. Programm P:

$$\begin{aligned} p(y) &\leftarrow q(y), \neg r(a,y) \\ p(f(z)) &\leftarrow \neg q(z) \\ p(b) &\leftarrow \end{aligned}$$

Dann ist comp(P) die Formel

$$\begin{aligned} \forall x \big(p(x) \leftrightarrow & \exists y (x = y \wedge q(y) \wedge \neg r(a,y)) \\ & \vee \exists z (x = f(z) \wedge \neg q(z)) \\ & \vee x = b \big) \\ \wedge \quad & \text{Gleichheitstheorie.} \end{aligned}$$