Theoretische Informatik - Zusammenfassung

Dino Wernli, Fabian Hahn

15. März 2009

1 Alphabete, Sprachen, Aufgaben

1.1 Alphabete, Wörter und Sprachen

Alphabete

Eine endliche, nichtleere Menge Σ heisst Alphabet. Die Elemente eines Alphabets werden Buchstaben, Zeichen oder Symbole genannt.

Wörter

Ein Wort über das Alphabet Σ ist eine endliche (evtl. leere) Folge von Buchstaben aus Σ.
Das leere Wort λ ist die leere Buchstabenfolge. Sie ist ein Wort über jedem Alphabet enthalten und es gilt $|\lambda| = 0$.
Die Länge $|w|$ eines Wortes w ist gegeben durch die totale Anzahl Buchstaben in w.
Die Menge aller möglichen Wörter, die man mit 0 oder mehr Buchstaben aus Σ bilden kann, heisst Σ^*. Analog definiert man die Menge der Wörter mit einem oder mehr Buchstaben aus Σ als Σ^+. Es gilt:

$$\Sigma^+ = \Sigma^* \setminus \{\lambda\}$$

Die Anzahl der Wörter der Länge $\leq i$, die man über Σ bilden kann, beträgt:

$$\# = \sum_{k=0}^{i} |\Sigma|^k = \frac{|\Sigma|^{i+1} - 1}{|\Sigma| - 1}$$

Konkatenation von Wörter

Die Konkatenation bezüglich ein Alphabet Σ ist eine Abbildung $K : \Sigma^* \times \Sigma^* \to \Sigma^*$. Es gilt für alle $x, y \in \Sigma^*$:

$$K(x, y) = x \cdot y = xy$$

Die Konkatenation ist assoziativ:

$$K(x, K(y, z)) = x \cdot (y \cdot z) = (x \cdot y) \cdot z = K(K(x, y), z)$$

Für alle $x, y \in \Sigma^*$ gilt:

$$|xy| = |x \cdot y| = |x| + |y|$$

Die i-te Iteration von $x \in \Sigma^*$ ist definiert als:

$$x^0 = \lambda, \quad x^1 = x, \quad x^i = x \cdot x^{i-1}$$

Für $x \in \Sigma^*,\ a \in \Sigma$ ist $|x|_a$ die Anzahl Vorkommen von a in x.

Teilwörter, Präfixe, Suffixe

- v Teilwort von $w \quad \leftrightarrow \quad \exists x, y \in \Sigma^* : w = xvy$
- v Präfix von $w \quad \leftrightarrow \quad \exists x \in \Sigma^* : w = vx$
- v Suffix von $w \quad \leftrightarrow \quad \exists x \in \Sigma^* : w = xv$

v heisst echtes Teilwort/Präfix/Suffix, falls zusätzlich gilt:

$$v \neq \lambda \wedge v \neq w$$

Bemerkung: λ und w sind immer Teilwort, Präfix und Suffix von w (gleichzeitig).

Kanonische Ordnung

Sei $\Sigma = s_1, s_2, \ldots, s_m,\ m \geq 1$ und sei $s_1 < s_2 < \cdots < s_m$ eine totale Ordnung auf Σ, so ist definiert man die kanonische Ordnung auf Σ^* für $u, v \in \Sigma^*$ als:

$$\begin{aligned} u < v \quad \leftrightarrow \quad & |u| < |v| \quad \vee \\ & |u| = |v| \ \wedge\ u = xs_iu' \ \wedge\ v = xs_jv', \quad i < j \end{aligned}$$

Sprachen

Eine Sprache L über Σ ist eine Teilmenge von Σ^*. Das Komplement L^c zu L bezüglich Σ ist die Sprache $\Sigma^* \setminus L$.
Spezielle Sprachen:

- $L_\emptyset = \emptyset$ ist die leere Sprache
- $L_\lambda = \{\lambda\}$ ist die Sprache mit nur dem leeren Wort

Die Konkatenation zweier Sprachen L_1 und L_2 ist definiert als:

$$L_1 \cdot L_2 = L_1L_2 = \{vw | v \in L_1 \ \wedge w \in L_2\}$$

Es gelten die Operationen:

$$\begin{aligned} L^0 = L_\lambda\ ,\ L^{i+1} = L^i \cdot L \\ L^* = \bigcup_{i \in \mathbb{N}} L^i\ ,\ L^+ = L \cdot L^* \end{aligned}$$

L^* nennt man den Kleene'schen Stern.
Es gelten die Beziehungen:

- $\Sigma^i = \{x \in \Sigma^* \mid \ |x| = i\}$
- $L_\emptyset \cdot L = L_\emptyset = \emptyset$
- $L_\lambda \cdot L = L$
- $L_1L_2 \cup L_1L_3 = L_1(L_2 \cup L_3)$
- $L_1(L_2 \cap L_3) \subseteq L_1L_2 \cap L_1L_3$

1.2 Algorithmische Probleme

Algorithmen

Ein Algorithmus ist eine Funktion $A : \Sigma_1^* \to \Sigma_2^*$, die für jede Eingabe eine eindeutige Ausgabe bestimmt.
Die Ausgabe des Algorithmus A mit der Eingabe x bezeichnen wir als $A(x)$. Zwei Algorithmen A und B heissen äquivalent, falls $\forall x : A(x) = B(x)$

Entscheidungsprobleme

Ein Entscheidungsproblem (Σ, L) ist, für jeden input $x \in \Sigma^*$ zu entscheiden, ob $x \in L$ oder nicht. Für Algorithmen, die das Entscheidungsproblem lösen, gilt:

$$A(x) = \begin{cases} 1, & x \in L \\ 0, & x \notin L \end{cases}$$

Sprachen, für die ein Algorithmus existiert, die sie erkennt, heissen rekursiv.

Transformationen

Ein Algorithmus A berechnet eine Transformation (Funktion) $f : \Sigma^* \to \Gamma^*$, falls:

$$\forall x \in \Sigma^* : \ A(x) = f(x)$$

Entscheidungsprobleme können als Spezialfälle von Funktionsberechnungen betrachtet werden, da sie die charakteristische Funktion einer Sprache berechnen.

Relationsprobleme

Ein Algorithmus A berechnet die Relation $R \subseteq \Sigma^* \times \Gamma^*$, falls:

$$\forall x \in \Sigma^* : \ (x, A(x)) \in R$$

Dabei reicht es, für jedes gegebene x eins von potentiell unendlich vielen y mit $(x, y) \in R$ zu berechnen.

Optimierungsprobleme

PLACEHOLDER

Aufzählungsprobleme

Ein Algorithmus A zählt eine Sprache L über Σ auf, falls er für die Eingabe n die ersten n Elemente der Sprache in kanonischer Reihenfolge ausgibt.

Binäre Kodierung von Zahlen

In vielen Beweisen zur Kolmogorov-Komplexität (nächstes Kapitel) kommt die Anzahl benötigter Zeichen zur binären Kodierung $Bin(n)$ von n. Diese Anzahl beträgt $\lceil \log_2(n+1) \rceil$. Der Grund für das $+1$ ist, dass das Wort $x = Bin(n)$ das $(n+1)$-te Wort aus Σ_{bool} ist (in kanonischer Ordnung).

Kolmogorov-Komplexität

Ein Algorithmus generiert $x \in \Sigma^*$, falls er für die Eingabe λ das Wort x ausgibt.
Die Kolmogorov-Komplexität $K(x)$ von $x \in \Sigma^*_{\text{bool}}$ ist die binäre Länge des kürzesten Programms, das x generiert. Sie ist unabhängig der gewählten Programmiersprache, da man jeden Übersetzer in der Konstanten unterbringen kann. Es gilt:

$$\forall x \in \Sigma^*_{\text{bool}} : \ K(x) \leq |x| + d$$

Die Konstante d bezeichnet dabei den nötigen Programmcode:

1. zur Spezifizierung der Längen von Präfix und Suffix
2. zur Ausgabe von x

Die Kolomogorov-Komplexität einer natürlich Zahl n ist gegeben durch:

$$K(n) = K(Bin(n))$$

Ein nicht komprimierbares Wort heisst zufällig:

- Bedingung für $x \in \Sigma_{\text{bool}} : \ K(x) \geq |x|$
- Bedingung für $n \in \mathbb{N} : \ K(n) \geq \lceil \log_2(n+1) \rceil \underbrace{-1}_{\text{führende 1}}$

Es existieren zufällige Zahlen jeder Länge:

$$\forall n \exists w_n \in (\Sigma_{\text{bool}})^n : \ K(w_n) \geq n$$

Beweis: Wir haben 2^n Wörter in $(\Sigma_{\text{bool}})^n$. Es ist klar, dass wir also 2^n verschiedene Programme $p_i \in \{0,1\}^*$ brauchen, um diese zu generieren. Die Anzahl aller nichtleeren Codestrings mit $|p| \leq n-1$ beträgt $\sum_{i=1}^{n-1} 2^i = 2^n - 2 < 2^n$. Somit haben mindestens 2 Programme mit $|p| > n-1$. ■

Sei L eine Sprache über Σ_{bool} und sei z_n das n-te Wort in L bezüglich kanonischer Ordnung. Löst ein Algorithmus das Entscheidungsproblem $(\Sigma_{\text{bool}}, L)$, so gilt:

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + c$$

Sei $Prim(n)$ die Anzahl der Primzahlen kleiner gleich n. Es gilt der Primzahlsatz:

$$\lim_{n \to \infty} Prim(n) = n / \ln n$$

Sei $n_1, n_2, \ldots$ eine unendliche Folge von Zahlen mit $K(n_i) \geq \lceil \log_2 n_i \rceil / 2$. Sei q_i die grösste Primzahl, die n_i teilt. Dann ist die Menge $Q = \{q_i | i \in \mathbb{N}\}$ unendlich.

Beweis: nehme an Q sei endlich und sei p_m die grösste Primzahl in Q. Dann kann man jede Zahl darstellen als:

$$n_i = p_1^{r_1} \cdot p_2^{r_2} \cdot \cdots \cdot p_m^{r_m}$$

Sei c die Länge des Programms A ohne die Darstellungen der r_i, dann gilt:

$$\begin{aligned} K(n_i) &\leq c + 2 \cdot \underbrace{(\lceil \log_2(r_1+1) \rceil + \cdots + \lceil \log_2(r_m+1) \rceil}_{m} \\ &= c + 2m \cdot \lceil \log_2(\log_2(n_i+1)) \rceil \end{aligned}$$

Die multiplikative Konstante 2 dient dazu, die r_i vom restlichen Code zu trennen. Da c und m Konstant sind, kann

$$\lceil \log_2 n_i \rceil / 2 \leq c + 2mc + 2m \cdot \lceil \log_2(\log_2(n_i+1)) \rceil$$

nur für endlich viele i gelten. Dies ist aber ein Widerspruch zu unserer Voraussetzung $\forall i : K(n_i) \geq \lceil \log_2 n_i \rceil / 2$ ■

2 Endliche Automaten

2.1 Definitionen

Formaler endlicher Automat

Endliche Automaten sind spezielle Programme, die ohne Speicher rechnen. Formal ist ein deterministischer endlicher Automat ein Quintupel $M = (Q, \Sigma, \delta, q_0, F)$. Dabei ist:

1. Q eine endliche Menge von Zuständen
2. Σ das Eingabealphabet
3. $q_0 \in Q$ der Anfangszustand
4. $F \subseteq Q$ die Endzustände, die den Input "akzeptieren"
5. $\delta : Q \times \Sigma \to Q$ die Übergangsfunktion

Konfiguration

Eine Konfiguration von M setzt sich aus dem aktuellen Zustand des Automaten und dem restlichen Teil des Inputwortes zusammen. Sie ist also ein Tupel:

$$(q, w) \in Q \times \Sigma^*$$

Die Konfiguration (q_0, x) heisst Startkonfiguration von M auf x. Endkonfigurationen sind Konfiguration aus $Q \times \{\lambda\}$.

Schritt

Ein Schritt beschreibt den Übergang von einer Konfiguration in die nächste. Sie ist eine Relation $R_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$:

$$(q, w)\ R_M\ (p, x) \leftrightarrow (w = ax,\ a \in \Sigma) \wedge (\delta(q, a) = p)$$

Berechnung

Eine Berechnung $C = C_0, C_1, \ldots, C_n$ ist eine Folge von Konfigurationen, für die gilt:

$$\text{für } 0 \leq i \leq n-1 :\ C_i\ R_M\ C_{i+1}$$

C heisst Berechnung von M auf der Eingabe x, falls gilt:

$$C_0 = (q_0, x),\ C_n \in Q \times \{\lambda\}$$

Akzeptierte Sprachen

Falls $C_n \in F$ für ein Wort x, wird x akzeptiert, sonst wird es verworfen. Die (eindeutige) von M akzeptierte Sprache $L(M)$ ist definiert als:

$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$$

Die Menge $\mathcal{L}(EA) = \{L(M) \mid M \text{ ist ein EA }\} \subseteq 2^{\Sigma^*}$ ist die Klasse von Sprachen, die alle von endlichen Automaten akzeptiert werden. Solche Sprache heissen regulär.

Transitive Hülle

Die reflexive und transitive Hülle R_M^* der Schrittrelation R_M eines endliche Automaten M ist definiert als:

$$(q, w) R_M^* (p, u) \ \leftrightarrow\ (q = p \wedge w = u) \vee \exists k \in \mathbb{N} :$$

1. $w = a_1 a_2 \ldots a_k u, \quad a_i \in \Sigma$
2. $\exists r_1, \ldots, r_{k-1} \in Q :$
   $(q, w)\ R_M\ (r_1, a_2, \ldots, a_k u)\ R_M\ \ldots\ R_M\ (r_{k-1}, a_k u)\ R_M\ (p, u)$

Zustandsklassen

Zunächst definieren wie die Funktion $\hat{\delta} : Q \times \Sigma^* \to Q$, die jedem Wort sein Endzustand zuordnet:

- $\hat{\delta}(q, \lambda) = q, \quad q \in Q$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a), \quad a \in \Sigma, w \in \Sigma^*, q \in Q$

Ein endlicher Automat M partitioniert die Menge Σ^* in $|Q|$ disjunkte Teilmengen:

$$Kl[p] = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) = p\} = \{w \in \Sigma^* \mid (q_0, w)\ R_M^*\ (p, \lambda)\}$$

Da Partitionen durch Äquivalenzrelationen entstehen, gilt:

- $Kl[p] \cap Kl[q] = \emptyset, \quad p \neq q$
- die Relation besagt: $x \sim y \leftrightarrow \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$
- $L(M) = \bigcup_{p \in F} Kl[p]$

Korrektheitsbeweise

Korrektheitsbeweise für endliche Automaten bestehen darin zu zeigen, dass $L(M) = L$ für ein gegebenes L. Sie werden grundsätzlich nach folgendem Schema per Induktion über die Länge des Inputs durchgeführt:

1. Beweise die Annahme für λ und für Wörter $x \in \Sigma^1$
2. Annahme gilt $x \in \Sigma^i$, beweise sie auch für $y \in \Sigma^{i+1}$

Für den Beweis des Induktionsschritts erweist sich meist folgende Umformung als besonders nützlich:

$$\hat{\delta}(q_0, y) = \hat{\delta}(q_0, za) = \delta(\hat{\delta}(q_0, z), a)$$

Nun kann man die Induktionsvoraussetzung auf $\hat{\delta}(q_0, z)$ anwenden und eine Fallunterscheidung für a durchführen.

2.2 Simulationen

Seien $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1), M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ zwei endliche Automaten. Für jede Mengenoperation $\diamond = \{\cup, \cap, \setminus\}$ existiert ein endlicher Automat M, sodass:

$$L(M) = L(M_1) \diamond L(M_2)$$

Die Konstruktion von $M = (Q, \Sigma, \delta, q_0, F_\diamond)$ lässt sich wie folgt darstellen:

1. $Q = Q_1 \times Q_2$
2. $q_0 = (q_{01}, q_{02})$
3. $\forall q \in Q_1, p \in Q_2, a \in \Sigma :\ \delta((q, p), a) = (\delta_1(q, a), \delta_2(p, a))$
4. $F_\diamond = \begin{cases} F_1 \times Q_2 \cup F_2 \times Q_1, & \diamond = \cup \\ F_1 \times F_2, & \diamond = \cap \\ F_1 \times (Q_2 \setminus F_2), & \diamond = \setminus \end{cases}$

Beweis: Es reicht, die folgende Gleichheit zu zeigen:

$$\hat{\delta}((q_{01}, q_{02}), x) = (\hat{\delta_1}(q_{01}, x), \hat{\delta_2}(q_{02}, x))$$

Wir beweisen per Induktion über $|x|$.
Induktionsanfang: Offensichtlich erfüllt, falls $x = \lambda$.
Induktionsschritt: Betrachte das Wort $w \in \Sigma^{i+1}$ mit $w = za,\ z \in \Sigma^i, a \in \Sigma$ und nehme an, die Behauptung gilt für $|x| \leq i$, also auch für z.

$$\begin{aligned}
\hat{\delta}((q_{01}, q_{02}), w) &= \hat{\delta}((q_{01}, q_{02}), za) \\
&= \delta(\hat{\delta}((q_{01}, q_{02}), z), a) \\
\text{Ind. } \rightarrow &= \delta((\hat{\delta_1}(q_{01}, z), \hat{\delta_2}(q_{02}, z)), a) \\
\text{Def. } \delta \rightarrow &= (\delta_1(\hat{\delta_1}(q_{01}, z), a), \delta_2(\hat{\delta_2}(q_{02}, z), a)) \\
&= (\hat{\delta_1}(q_{01}, za), \hat{\delta_2}(q_{02}, za)) \\
&= (\hat{\delta_1}(q_{01}, w), \hat{\delta_2}(q_{02}, w)) \quad \blacksquare
\end{aligned}$$

2.3 Beweise der Nichtregularität

Um zu zeigen, dass eine Sprache nicht regulär ist, reicht es zu zeigen, dass es keinen endichen Automaten gibt, der die Sprache akzeptiert: $L \notin \mathcal{L}(EA)$. Für solche beweise gibt es 3 Varianten der Argumentation, die sich auf 3 Lemmas stützen.

Variante 1 - Lemma 3.3

Lemma 3.3: Sei $A = (Q, \Sigma, \delta_A, q_0, F)$ ein endlicher Automat und seien $x, y \in \Sigma^*,\ x \neq y$ mit $x, y \in Kl[p]$. Dann gilt:

$$\forall z \in \Sigma^*,\ \exists r \in Q :\ xz, yz \in Kl[r]$$

Als dirkte Folge des Lemmas gilt:

$$xz \in L(A) \ \leftrightarrow\ yz \in L(A)$$

Beispielbeweis: Indirekt für $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. Sei $A = (Q, \Sigma, \delta_A, q_0, F)$ ein endlicher Automat mit $L(A) = L$. Betrachte die Wörter $0, 0^2, 0^3, \ldots, 0^{|Q|+1}$. Weil es mehr Wörter als Zustände gibt, muss folgendes gelten:

$$\exists i < j \in \{1, 2, \ldots, |Q| + 1\} :\ \hat{\delta}_A(q_0, 0^i) = \hat{\delta}_A(q_0, 0^j)$$

Nach dem Lemma muss für alle $z \in \Sigma^*_{\text{bool}}$ gelten:

$$0^i z \in L \ \leftrightarrow\ 0^j z \in L$$

Nun kann man aber $z = 1^i$ wählen. Es gilt: $0^i 1^i \in L,\ 0^j 1^i \notin L$. Also haben wir einen Widerspruch erhalten. $\blacksquare$

Variante 2 - Pumping Lemma

Pumping Lemma: Sei L regulär. Dann exisitert ein $n \in \mathbb{N}$, sodass sich jedes Wort w mit $|w| \geq n$ in $w = yxz$ zerlegen lässt, wobei

1. $|yx| \leq n_0$
2. $|x| \geq 1$
3. $\{yx^k z \mid k \in \mathbb{N}\} \subseteq L\ \vee\ \{yx^k z \mid k \in \mathbb{N}\} \cap L = 0$

Beispielbeweis: Indirekt für $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. Sei $A = (Q, \Sigma, \delta_A, q_0, F)$ ein endlicher Automat mit $L(A) = L$. Betrachte nun das Wort $w = 0^{|Q|}1^{|Q|}$. Laut Lemma muss eine Zerlegung $w = yxz$ existieren, mit allen 3 Eigenschaften. Weil $|yx| < |Q|$, muss $x = 0^i$ für ein bestimmtes i gelten. Nun muss aber $w' = yx^k z \in L$ gelten. Dies gilt aber offensichtlich nicht, da sie Bedingung der Sprache verletzt wäre. Also haben wir einen Widerspruch. $\blacksquare$

Variante 3 - Satz 3.1

Sei $L \subseteq (\Sigma_{\text{bool}})^*$ regulär. Man definiert für jedes $x \in \Sigma^*$ die Menge $L_x = \{y \in \Sigma^* \mid xy \in L\}$. Sei nun w das n-te Wort aus L_x, dann existiert die Konstante c, sodass:

$$K(w) \leq \lceil \log_2(n+1) \rceil + c$$

Beispielbeweis: Indirekt für $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. Unter der Annahme das L regulär ist, bilden wir:

$$L_{0^m} = \{y \mid 0^m y \in L\} = \{1^m, 01^{m+1}, \ldots\} = \{0^j 1^{m+j} \mid j \in \mathbb{N}\}$$

Nun ist 1^m das erste Wort, also gilt:

$$K(1^m) \leq \log_2(1+1) + c = 1 + c$$

Da man zu jedem m die jeweilige Gruppe L_{0^m} bilden kann, hat man gezeigt, dass unendlich viele Wörter die Kolmogorov-Komplexität $1+c$ besitzen. Dies ist aber nicht möglich, da die Menge aller Programme mit Länge $1+c$ endlich ist. Also haben wir den Widespruch. $\blacksquare$

2.4 Nichtdeterminismus

Definitionen

Nichtdeterministische endliche Automaten (NEA) unterstützen die Möglichkeit, aus manchen Konfigurationen eine Auswahl aus mehreren Aktionen zu treffen. Formal sind sie definiert als $M = (Q, \Sigma, \delta, q_0, F)$. Dabei ist:

1. Q eine endliche Menge von Zuständen
2. Σ das Eingabealphabet
3. $q_0 \in Q$ der Anfangszustand
4. $F \subseteq Q$ die Endzustände, die den der Input "akzeptieren"
5. $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ die Übergangsfunktion

Ein Schritt $R_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ ist definiert durch:

$$(q, w)\ R_M\ (p, x) \leftrightarrow w = ax, a \in \Sigma\ \wedge\ p \in \delta(q, a)$$

Insbesondere bedeutet dies, dass mehrere verschiedene Schritte aus einer Konfiguration heraus gültig sind, da man im Allgemeinen mehrere p finden kann.
Eine Berechnung von M auf $w \in \Sigma^*$ ist eine Folge von Konfgurationen $C_0 C_1 \ldots C_n$ von M mit $C_0 = (q_0, w)$ und $\forall i < n : C_i\ R_M\ C_{i+1}$, die eine der folgenden Bedingungen erfüllt:

- $C_n \in Q \times \{\lambda\}$
- $C_n = (q, ax)$ mit $\delta(q, a) = \emptyset$

Eine akzeptierende Berechnung von M auf w ist eine Berechnung, für die gilt: $C_n = (p, \lambda), p \in F$. Eine Sprache L wird von einem NEA akzeptiert, falls für alle $w \in L$ eine akzeptierende Berechnung von M auf w existiert.
Analog zum deterministischen Fall definiert man R^*_M als die reflexive und transitive Hülle von R_M. Auch analog zu EA, ist die Funktion $\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ definiert als:

- $\forall q \in Q : \hat{\delta}(q, \lambda) = \{q\}$
- $\hat{\delta}(q, wa) = \{p \mid \exists r \in \hat{\delta}(q, w) : p \in \delta(r, a)\}$
  $= \bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a)$

Anschaulich ist $\hat{\delta}(q_0, w)$ die Menge aller Zustände, die aus q_0 durch vollständiges Lesen von w erreichbar sind.
Die von einem NEA M akzeptierte Sprache L ist definiert als:

$$\begin{aligned} L(M) &= \{w \in \Sigma^* \mid \exists p \in F : \ (q_0, w) R_M^* (p, \lambda)\} \\ &= \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\} \end{aligned}$$

Um nachzuprüfen, ob ein NEA ein Wort x akzeptiert, muss man alle Berechnungen betrachten. Dies macht man am besten durch den sogenannten Berechnungsbaum $\mathcal{B}_M(x)$.

PLACEHOLDER FOR DA KRASSE BERECHNUGSBAUM!!1

Zurückführung auf deterministische EA

Es gilt der folgende Satz (M: NEA, A: EA)

$$\forall M \ \exists A : \ L(M) = L(A)$$

Die formale Konstruktion des EA $A = (Q_A, \Sigma_A, \delta_A, q_{0A}, F_A)$ zum NEA $M = (Q, \Sigma, \delta_M, q_0, F)$ geschieht wie folgt:

- $Q_A = \{\langle P \rangle \mid P \subseteq Q\}$
  Dabei bedeutet $\langle P \rangle$, dass man alle Zustände aus P zusammengefasst als einen Zustand betrachtet.
- $\Sigma_A = \Sigma$
- $q_0 A = \{\langle q_o \rangle\}$
- $F_A = \{\langle P \rangle \mid P \subseteq Q \ \wedge \ P \cap F \neq \emptyset\}$
- $\delta_A : Q_A \times \Sigma_A \to Q_A$ ist für jedes $\langle P \rangle \in Q_A$ so definiert:

$$\begin{aligned} \delta(\langle P \rangle, a) &= \left\langle \bigcup_{p \in P} \delta_M(p, a) \right\rangle \\ &= \langle \{q \in Q \mid \exists p \in P : q \in \delta_M(p, a)\} \rangle \end{aligned}$$

Korrektheitsbeweis: zu zeigen ist die folgende Äquivalenz:

$$\forall x \in \Sigma^* : \hat{\delta}_M(q_0, x) = P \ \leftrightarrow \ \hat{\delta}_A(q_{0A}, x) = \langle P \rangle$$

Wir zeigen dies per Induktion über $|x|$.
Verankerung: für $|x| = 0$ gilt die Behauptung offensichtlich, da $\hat{\delta}_M(q_0, \lambda) = \{q_o\}$ und $q_{0A} = \langle \{q_0\} \rangle$ beide gelten.
Schritt: sei die Behauptung gültig für alle $z \in \Sigma^*$ mit $|z| \leq m$. Wir zeigen, dass sie auch für alle $xa = y \in \Sigma^{m+1}$ gilt.

$$\begin{aligned} \hat{\delta}_A(q_{0A}, xa) &= \delta_A(\hat{\delta}_A(q_{0A}, x), a) \\ &= \delta_A(\left\langle \hat{\delta}_M(q_0, x) \right\rangle, a) \\ &= \left\langle \bigcup_{p \in \hat{\delta}_M(q_0, x)} \delta_M(p, a) \right\rangle \\ &= \left\langle \hat{\delta}_M(q_0, xa) \right\rangle \qquad \blacksquare \end{aligned}$$

rsectionGrammatiken Eine Grammatik ist ein Mechanismus zu generierung von Wörtern. Die Menge aller Sprachen, die durch Grammatiken erzeugbar sind, enspricht genau der Menge aller rekursiven Sprachen.

3 Grammatiken

3.1 Formale Definition

Eine Grammatik ist ein Quadrupel $G = (\Sigma_N, \Sigma_T, P, S)$, wobei:

- Σ_N ist das Nichtterminalalphabet
- Σ_T ist das Terminalalphabet. Es gilt: $\Sigma_N \cap \Sigma_T = \emptyset$
- $S \in \Sigma_T$ ist das Startnichtterminal
- Für $\Sigma = \Sigma_N \cup \Sigma_T$ ist P eine endliche Teilmenge von $\Sigma^* \Sigma_N \Sigma^* \times \Sigma^*$. Sie stellt die Menge der Ableitungsregeln dar

Man verwendet die Notation:

$$(\alpha, \beta) \in P \ \Leftrightarrow \ \alpha \to_G \beta$$

Konventionen der Notation

Die Nutzung der vorkommenden Symbole wird folgendermassen festgelegt:

- Für Terminalsymbole nutzt man a, b, c, d, e
- Grossbuchstaben A, B, C, X, Y, Z werden für Nichtterminale gebraucht
- Die Kleinbuchstaben w, x, y, z bezeichnen Wörter aus Σ_T
- Die Griechischen Buchstaben α, β werden für beliebige Wörter aus $\Sigma = \Sigma_T \cup \Sigma_N$ verwendet

Die Notation $X \to aXb$ bedeutet, dass man jedes auftreten des Nichtterminals X durch aXb ersetzen darf.

3.2 Ableitungen, erzeugte Sprachen

Seien γ, δ Wörter aus Σ^*. $\gamma \Rightarrow_G \delta$ bedeutet, dass δ aus γ in einem Schritt ableitbar ist:

$$\exists \omega_1, \omega_2 \in \Sigma \ \exists (\alpha, \beta) \in P : \ \gamma = \omega_1 \alpha \omega_2, \ \delta = \omega_1 \beta \omega_2$$

Wir sagen δ ist aus γ ableitbar, falls entweder:

- $\gamma = \delta$
- eine Sequenz von Wörter $\omega_1, \omega_2, \ldots, \omega_n$ existiert, wobei:

$$\gamma = \omega_0 \ \wedge \ \delta = \omega_n \ \wedge \ \forall i : \omega_i \Rightarrow_G \omega_{i+1}$$

Man schreibt $\gamma \Rightarrow_G^* \delta$. Falls $S \Rightarrow_G^* x$ für $x \in \Sigma_T^*$, dann wird x von G erzeugte. Die erzeugt Sprache von G ist:

$$L(G) = \{x \in \Sigma_T^* \mid S \Rightarrow_G^* x\}$$

Eine Ableitung, die genau i Schritte benötigt, bezeichnet man mit $\alpha \Rightarrow_G^i \beta$.

Beispiel mit Beweis

Man möchte beweisen, dass die Grammatik G die folgende Sprache erzeugt:

$$L(G) = L = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$$

Dabei ist $G = (\{S\}, \{a, b\}, P, S)$ mit:

$$P = \{S \to \lambda, \ S \to SS, \ S \to aSb, \ S \to bSa\}$$

Beweis von "$\subseteq$": Da jede mögliche Ableitung gleich viele a und b erzeugt, ist offensichtlich, dass jedes von G genereierte Wort in L ist.

Beweis von "$\supseteq$": Per Induktion über die Wortlänge n. Induktionsanfang: $n = 0$. Das einzige Wort aus L mit $|L| = 0$ ist λ. Nach den Regeln in P gilt:

$$S \Rightarrow_G^* \lambda$$

Induktionsannahme: alle Wörter aus L der Länge $2(n-1)$ können durch die Grammatik G aus S hergeleitet werden (Wörter ungerader Länge kann G nicht produzieren).
Induktionsschritt: wir wollen zeigen, dass aus der Annahme folgt, dass alle Wörter aus L der Länge $2n$ von G generiert werden können. Für ein beliebiges Wort $x \in L$ der Länge $2n$ tritt einer der folgenden Fälle ein:

1. $x = ayb$: da $x \in L$, muss $|y|_a = |y|_b$ gelten. Da $|y| = 2(n-1)$, gilt nach Induktionsannahme: $S \Rightarrow_G^* y$. Wir können also folgern: $S \Rightarrow_G aSb \Rightarrow_G^* ayb$.

2. $x = bya$: Begründung identisch zu Punkt 1

3. $x = aya$: Weil $|x|_a = |x|_b$, gilt: $|y|_b = |y|_a + 2$. Es muss also eine Zerlegung $y = uv$ geben, so dass $|u|_b = |u|_a + 1$ und $|v|_b = |v|_a + 1$.
   Nun gilt also $x = auva$, wobei $au \in L \wedge va \in L$. Nach der Induktionsannahme können au und va durch G generiert werden: $S \Rightarrow_G^* au$, $S \Rightarrow_G^* va$.
   Auch erlaubt ist: $S \Rightarrow SS \Rightarrow_G^* auva$

4. $x = byb$: Begründung identisch zu Punkt 3

3.3 Chomsky-Hierarchie

Die Chomsky-Hierarchie besteht aus 4 Klassen von Grammatiken:

- Typ-0: allgemeine, uneingeschränkte Grammatiken
- Typ-1: kontextsensitive Grammatiken
  $\forall(\alpha, \beta) \in P: \ |\alpha| \leq |\beta|$
- Typ-2: kontextfreie Grammatiken
  $\forall(\alpha, \beta) \in P: \ |\alpha| = 1 \ \wedge \ \beta \in (\Sigma_N \cup \Sigma_T)^*$
- Typ-3: reguläre Grammatiken
  $\forall(\alpha, \beta) \in P: \ |a| = 1 \ \wedge \ \beta \in (\Sigma_T^* \cdot \Sigma_N) \cup \Sigma_T^*$

Eine Sprache L ist vom Typ $i \in \{0, 1, 2, 3\}$, falls eine Grammatik G existiert mit $L(G) = L$. Die Familie der Sprachen des Typs i werden mit $\mathcal{L}_i$ bezeichnet. Es gilt:

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0$$

3.4 Reguläre Grammatiken

Die Menge $\mathcal{L}_3$ enthält offensichtlich alle endlichen Sprachen, da man immer eine Grammatik bauen kann, die Ableitungen von S zu jedem Wort der Sprache erlaubt.

$\mathcal{L}_3$ ist abgeschlossen bezüglich Vereinigung.

Beweis: Seien $L, L' \in \mathcal{L}_3$. Es existieren also Grammatiken G, G' mit $L(G) = L, L(G') = L'$. Da wir Nichtterminale umbenennen dürfen, können wir oBdA annehmen, dass $\Sigma'_N \cap \Sigma_N = \emptyset$. Nun konstruieren wir die Typ-3 Grammatik $G'' = (\Sigma''_N, \Sigma''_T, P'', S'')$, für die $L(G'') = L(G) \cup L(G')$ gilt:

- $\Sigma''_N = \{S''\} \cup \Sigma_N \cup \Sigma'_N$
- $\Sigma''_T = \Sigma_T \cup \Sigma'_T$
- $P'' = P \cup P' \cup \{S'' \to S, S'' \to S'\}$

Inklusion $L(G') \cup L(G) \subseteq L(G'')$: Sei $x \in L(G) \cup L(G')$. Wir können oBdA annehmen, $x \in L(G)$. Dann existiert die Ableitung:

$$S \Rightarrow_G^* x$$

in G. Daraus folgt, dass die Ableitung in G'' existiert

$$S'' \Rightarrow_{G''} S \Rightarrow_{G''}^* x$$

da alle Operationen von G auch in G'' erlaubt sind.

Inklusion $L(G'') \subseteq L(G') \cup L(G)$: Sei $x \in L(G'')$, dann existiert die Ableitung in G:

$$S'' \Rightarrow_{G''} \alpha_1 \Rightarrow_{G''} \alpha_2 \Rightarrow_{G''} \cdots \Rightarrow_{G''} \alpha_n \Rightarrow_{G''} x$$

Wir wissen, dass S'' nur nach S oder S' abgeleitet werden kann. Wir unterscheiden also 2 Fälle:

1. $\alpha_1 = S$: dann ist $S \Rightarrow_{G''} \cdots \Rightarrow_{G''} x$ auch eine Ableitung in G, da $\Sigma_N \cap \Sigma'_N = \emptyset$. Also gilt: $x \in L(G)$.
2. $\alpha_2 = S'$: analog zu Fall 1, $x \in L(G')$. ■

Es gilt: $\mathcal{L}_3$ ist abgeschlossen bzgl. Konkatenation und Kleene'schen Stern. Ich beschreibe dafür die nötige Beweis-Grammatik $G'' = (\Sigma''_N, \Sigma''_T, P'', S'')$, nicht den ganzen Beweis:

- $\Sigma''_N = \Sigma_N \cup \Sigma'_N$
- $\Sigma''_T = \Sigma_T \cup \Sigma'_T$
- $S'' = S$
- $P'' = P' \cup (P \cap \Sigma_N \times \Sigma_T^* \Sigma_N) \cup \{A \to wS' \mid (A \to w) \in P\}$

Normierte reguläre Grammatik

Eine reguläre Grammatik heisst normiert, falls alle Regeln eine der folgenden Formen haben:

- $S \to \lambda$, S Startsymbol
- $A \to a, \ A \in \Sigma_N, a \in \Sigma_T$
- $B \to bC, \ B, C \in \Sigma_N, b \in \Sigma_T$

Zu jeder regulären Grammatik G existiert eine äquivalente, normierte Grammatik G'. Um diese zu erstellen, muss man:

1. Kettenregeln entfernen: $X \to Y, Y \to aa$ wird zu $X \to aa$
2. Regeln der Form $A \to \lambda$ entfernen, wobei A nicht das Startsymbol ist. Aus $X \to aY, Y \to \lambda$ wird neu $X \to a$
3. Regeln der Form $X \to abcY$ entfernen, indem man $X \to aY_1, Y_1 \to bY_2, Y_2 \to c$.

Bezug zum DEA

Es gilt der folgende Satz:

$$\mathcal{L}_3 = \mathcal{L}(EA)$$

Beweisansatz der Inklusion "$\supseteq$": zu zeigen ist, dass man zu jedem endlichen Automaten eine Typ-3 Grammatik konstruieren kann, die dieselbe Sprache akzeptiert:

- $\Sigma_N = Q_A$
- $\Sigma_T = \Sigma_A$
- $S = q_0$
- $P = \{p \to aq \mid a \in \Sigma_A,\ p, q \in Q \text{ mit } \delta(p, a) = q\} \cup \{f \to \lambda \mid f \in F_A\}$

Beweisansatz der Inklusion "$\subseteq$": zu zeigen ist, dass man zu jeder Typ-3 Grammatik einen endlichen Automaten konstruieren kann, der dieselbe Sprache akzeptiert:
oBdA. darf man davon ausgehen, dass eine normierte Grammatik vorliegt. Weiter reicht es auch, einen äquivalenten NEA $M = (\Sigma_N \cup \{q_F\}, \Sigma_T, \delta, S, F)$ zu konstruieren:

$$F = \begin{cases} \{q_F\}, & S \to \lambda \notin P \\ \{q_F, S\}, & S \to \lambda \in P \end{cases}$$

$\delta(q_F, a) = \emptyset, \quad \forall a \in \Sigma_T$

$Y \in \delta(X, a) \quad$ falls: $X \to aY \in P,\ X, Y \in \Sigma_N,\ a \in \Sigma_T$

$q_F \in \delta(X, a) \quad$ falls: $X \to a \in P,\ X \in \Sigma_N,\ a \in \Sigma_T$

Die Menge $\mathcal{L}_3$ ist auch abgeschlossen bezüglich Komplement. Das heisst:

$$L \subseteq \Sigma^*, L \in \mathcal{L}_3 \ \Rightarrow\ (\Sigma^* \setminus L) \in \mathcal{L}_3$$

Die Konstruktion einer regulären Grammatik, die das Komplement der Sprache einer anderen gegebenen regulären Grammatik G akzeptiert, geschieht wie folgt:

1. reguläre Grammatik G
2. normierte Grammatik G' aus G machen
3. zu G' äquivalenten NEA konstruieren
4. aus dem NEA ein EA konstruieren (sehr aufwändig, da Potenzmengenkonstruktion)
5. aus EA einen EA' bauen, der das Komplement akzeptiert
6. wieder eine reguläre Typ-3 Grammatik aus EA' bauen

3.5 Kontextfreie Grammatiken

Die Sprachmenge $\mathcal{L}_2$ oder $\mathcal{L}_{CF}$ heisst kontextfrei. Es gilt:

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2$$

Ableitungsbaum

Sei $G = (\Sigma_N, \Sigma_T, P, S)$ eine kontextfreie Grammatik und $x \in L(G)$. Sei $\mathcal{A}$ eine Ableitung von x in G. Dann definiert man den Ableitungsbaum $T_\mathcal{A}$:

1. $T_\mathcal{A}$ hat die Wurzel S
2. Knoten von $T_\mathcal{A}$ sind Symbole aus $\Sigma_N \cup \Sigma_T \cup \lambda$
3. Innere Knoten sind Symbole aus Σ_N
4. Blätter sind Symbole aus $\Sigma_T \cup \lambda$. Von links in-order gelesen ergeben sie x
5. Für jede Anwendung der Ableitung $A \to \alpha_1\alpha_2 \ldots \alpha_k$ ergibt sich die Menge $\{v_1, \ldots, v_n\}$ von Kinderknoten

Normierte kontextfreie Grammatik

Eine kontextfreie Grammatik G heisst normiert, falls G:

- keine nutzlosen Symbole enthält (ein Nichtterminal heisst nutzlos, falls es nie durch Ableitungen von S erreicht werden kann)
- keine Regeln der Form $X \to \lambda$ enthält
- keine Kettenregeln enthält

Links- und Rechtsableitung

Eine Linksableitung ist eine Ableitung eines ganzen Wortes, bei der immer nur das linkseste Nichtterminal ersetzt wird. Die Rechtsableitung ist analog definiert.

Für jede kontextfreie Grammatik existieren äquivalente Grammatiken in Chomsky-Normalform und Greibach-Normalform.

Chomsky-Normalform

Eine kontextfreie Grammatik ist in Chomsky-Normalform, falls alle Regeln eine der folgenden Formen haben:

- $A \to BC, \quad A, B, C \in \Sigma_N$
- $A \to a, \quad A \in \Sigma_N,\ a \in \Sigma_T$

Greibach-Normalform

Eine Grammatik ist in Greibach-Normalform, falls alle Regeln die folgende Form haben:

- $A \to a\alpha, \quad A \in \Sigma_N,\ a \in \Sigma_T,\ \alpha \in \Sigma_N^*$

Pumping-Lemma für kontextfreie Sprachen

Für jede kontextfreie Sprache L existiert eine Konstante n_L, so dass $\forall z \in L$ mit $|z| \geq n_L$ eine Zerlegung $z = uvwxy$ existiert, mit:

- $|vx| \geq 1$
- $|vwx| \leq n_L$
- $\{uv^iwx^iy \mid i \in \mathbb{N}\} \subseteq L$

3.6 Nichtdeterministische Kellerautomaten

Der Kellerautomat ist ein Modell einer Maschine zur Erkennung von kontextfreien Sprachen. Er besteht aus:

- einem Ein-Weg-Lesekopf zum Lesen des Wortes
- einer endliche Kontrolle, die in jedem Schritt entschdeidet, ob das nächste Eingabesymbol oder λ gelesen werden soll
- einem unendlich grossen Keller (Stack)

Formale Definition

Formal sind die Tupelelemente eines Kellerautomaten $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ definiert als:

- Q ist die endliche Menge der Zustände
- Σ ist das Eingabealphabet
- Γ ist das Kelleralphabet
- q_o ist der Anfangszustand
- Z_0 ist das Initialisierungssymbol des Kellers
- $\delta : Q \times (\Sigma \cup \lambda) \times \Gamma \to Q \times \Gamma^*$ ist die Übergangsfunktion

Konfigurationen

Eine Konfiguration von M ist ein Tripel:

$$(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$$

wobei:

- $q \in Q$ der aktuelle Zustand ist
- $w \in \Sigma^*$ der bisher nicht gelesene Teil der Eingabe ist
- $\alpha \in \Gamma^*$ der aktuelle Kellerinhalt ist

Für jedes Wort $x \in \Sigma^*$ ist (q_0, x, Z_0) die Anfangskonfiguration von M auf x.

Schritte

Ein Schritt R_M von M ist eine Relation auf Konfigurationen, definiert als:

- $(q, aw, \alpha X)\ R_M\ (p, w, \alpha\beta)$
  falls $(p, \beta) \in \delta(q, a, X)$ für $p, q \in Q, a \in \Sigma, X \in \Gamma$ und $\alpha, \beta \in \Gamma^*$
- $(q, w, \alpha X)\ R_M\ (p, w, \alpha\beta)$
  falls $(p, \beta) \in \delta(q, \lambda, X)$

Die reflexive und transitive Hülle R_M^* von R_M ist analog zum EA definiert.

Berechnungen

Kellerautomaten können unendliche Berechnungen ausführen, jedoch interessieren wir uns nur für die endlichen Berechnungen, da nur sie ein Wort akzeptieren können.
Eine endliche Berechnung von M ist eine Folge von Konfigurationen $C_1, C_2, \ldots, C_m$, so dass $C_i\ R_M\ C_{i+1},\ \forall i \in \{1, 2, \ldots, m-1\}$.
Eine (endliche) Berechnung von M auf $x \in \Sigma^*$ heisst akzeptierend, falls:

$$C_0 = (q_0, x, Z_0) \quad \text{und} \quad C_m = (p, \lambda, \lambda),\ p \in Q$$

Das Wort x wird von M akzeptiert, falls eine akzeptierende Berechnung von M auf x existiert. Die akzeptierende Sprache $L(M)$ von M ist definiert als:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0)\ R_M^*\ (p, \lambda, \lambda) \text{ für } p \in Q\}$$

Bezug zu kontextfreien Grammatiken

Lemma: Sei L eine kontextfreie Sprache, die λ nicht enthält. Dann existiert ein Kellerautomat M mit $L = L(M)$.
Die Spezifizierung mit λ kommt daher, dass sich jeder Automat am Anfang entscheiden kann (falls eine entsprechende Regel existiert), nicht zu lesen (λ) und das Initialsymbol aus dem Keller zu löschen. Diese Regel muss existieren, damit der Automat weiss, wann er terminieren soll.

Beweis: Um den obigen Satz zu beweisen, stellen wir zunächst fest, dass wir oBdA. annehmen können, dass die kontextfreie Grammatik $G = (\Sigma_N, \Sigma_T, P, S)$ mit $L(G) = L$ in Greibach-Normalform ist. Wir konstruieren nun einen Kellerautomaten M mit nur einem Zustand, sodass $L = L(M)$:

$$M = (\{q_0\}, \Sigma_T, \Sigma_N, \delta, q_0, S)$$

Für δ definieren wir:

$$\delta(q_0, a, A) = \left\{(q_0, \beta^R \mid A \to a\beta\right\}$$

Da wir von Linksableitungen ausgehen dürfen (kontextfrei), wird in jedem Schritt ein Nichtterminal abgeleitet und durch ein Terminal und ≥ 0 Nichtterminale ersetzt. Falls das Wort in der Sprache ist, rät der Automat, welche Regel angewandt wurde, um das gelesene Nichtterminal zu produzieren.
Er wählt also irgendeine (die richtige) Regel, die a aus A produziert, und pusht die entsprechenden Nichtterminale der Regel auf den Keller (in umgekehrter Reihenfolge). Falls keine solche Ableitung existiert (also das Wort nicht in der Sprache ist), bleibt der Automat einfach stecken, da $\delta(q_0, c, \gamma) = \emptyset$.

Lemma: Sei M ein Kellerautomat mit nur einem Zustand. Dann ist $L(M)$ kontextfrei.

Lemma: Für jeden Kellerautomaten $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ existiert ein äquivalenter Kellerautomat M_1 mit nur einem Zustand.

Beweis: Um dies zu beweisen, konstruieren wir den Automaten $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0^1, Z_0^1)$. Der Trick besteht darin, die Menge der Kellersymbole zu erweitern, sodass in M_1 zusätzlich zum Kellerinhalt von M auch Zustände von M gespeichert werden können:

$$Q_1 = \left\{q_0^1\right\}$$
$$\Gamma_1 = Q \times \Gamma \times Q$$

Dabei bedeutet $A' = (q, A, p) \in \Gamma_1$, dass M das Symbol A nur beim Übergang von q zu p löschen kann. Nun bleibt noch, δ anzugeben.
Zuerst betrachten wir die Abbauphase. Das erste Symbol im obersten Tupel bezeichnet immer den Zustand, in dem sich M aktuell befindet. Sei nun das Tupel (r, A, h) ganz oben. Falls M in nächsten Schritt nichts auf den Keller schreibt, tut dies M_1 auch nicht, sondern liest nur das oberste Tupel.
M_1 liest also, dass M in diesem Schritt A vom Keller nimmt und von r nach h übergehen sollte. Falls M das wirklich macht, geht die Berechnung weiter und beide Automaten entfernen ihr oberstes Element. Falls nicht, hört die Berechnung auf und das Wort ist nicht in L. Dies ergibt für δ_1:

$$\delta_1(q_0^1, a, (r, A, h)) = \left\{(q_0^1, \lambda) \mid (h, \lambda) \in \delta(r, a, A)\right\}$$

Falls M aber den Keller aufbaut, muss M_1 seinen Keller auch aufbauen können. Betrachten wir nun die Regel von δ:

$$(s, (B_1 B_2 \dots B_m)^R) \in \delta(r, aA)$$

also kann M von r nach s übergehen und $(B_1 \dots B_m)^R$ auf den Keller schreiben. Wir erlauben M_1 beim Aufbau die folgedenden Übergänge:

$$\begin{aligned} &\delta_1(q_0^1, a, (r, A, h)) = \\ \big\{(q_0^1, [(s, B_1, s_2)(s_2, B_2, s_3) \dots (s_m, B_m, h)]^R) \mid \\ &\forall s_2, s_3, \dots, s_m \in Q \\ &\forall (s, (B_1 B_2 \dots B_m)^R) \in \delta(r, a, A)\big\} \end{aligned}$$

Wichtig: Falls M beim Abbauen des Kellers von s in s_2 übergeht, muss zu jedem Zeitpunkt der aktuelle Zustand von M richtig sein. Dadurch, dass wir jede Kombinationen von Zuständen für $s_2, \dots, s_m$ aus Q erlauben, muss M_1 (nichtdeterministisch korrekt) raten, in welchen Zuständen M die Symbole aus dem Keller abbauen wird.
Jetzt bleibt nur noch ein Problem: M_1 wird mit Z_0^1 initialisiert. Damit wir das erste Tupel bekommen, führen wir die Regel für δ ein:

$$\delta_1(q_0^1, \lambda, Z_0^1) = \big\{(q_0^1, (q_0, Z_0, q)) \mid q \in Q\big\}$$

Somit entscheidet M_1 schon im allerersten Schritt, wo der Keller geleert wird, also wo eine akzeptierende Berechung von M enden wird. Da dies in jedem Zustand passieren könnte (Keller muss nur leer sein), müssen alle Zustände der Regel von δ_1 hinzugefügt werden und der Automat wird den richtigen von ihnen nichtdeterministisch korrekt raten.

Aus den obigen Lemmas folgt: die nichtdeterministischen Kellerautomaten erkennen genau die Klasse der kontextfreien Sprachen.

3.7 Cocke-Younger-Kasami - Algorithmus

Sei $G = (\Sigma_N, \Sigma_T, P, S)$ kontextfrei. Der CYK-Algorithmus überprüft in $O(n^3)$ für ein gegebenes Wort $w = a_1 a_2 \dots a_n,\ w \in \Sigma_T^*$, ob $w \in L(G)$ gilt. Zunächst führt man die folgende Menge $V_{i,j}$ ein:

$$V_{i,j} = \{A \in \Sigma_N \mid A \Rightarrow_G^* a_i a_{i+1} \dots a_j\}, \quad 1 \le i \le j \le n$$

Aus der Definition von V folgt:

$$\begin{aligned} w \in L(G) &\Leftrightarrow S \in V_{1,n} \\ V_{i,i} &= \{A \mid A \to a_i \in P\} \end{aligned}$$

Aus $V_{i,k}$ und $V_{k+1,j}$ lässt sich $V_{i,j}$ wie folgt rekursiv herleiten:

$$\begin{aligned} &A \in V_{i,j} \Leftrightarrow \\ \exists k, i \le k < j :\ &A \to BC \in P \wedge B \in V_{i,k} \wedge C \in V_{k+1,j} \end{aligned}$$

Dieser rekursive Algorithmus profitiert sehr von der Technik der dynmischen Programmierung, da die selben V immer wieder benutzt werden. Wenn man diese im Sinne der dynamischen Programmierung zwischenspeichert, kann man die Laufzeit auf $O(n^3 \cdot |P|)$ beschränken.

4 Turingmaschinen

Eine Turingmaschine besteht aus:

- einem unendlichen Band
- einer endlichen Kontrolle (dem Programm)
- einem Lese- und Schreibkopf, der sich in beiden Richtungen auf dem Band bewegen kann

4.1 Formale Defintion

Formal sind die Tupelelemente einer Turingmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ definiert als:

- Q ist die endliche Menge der Zustände
- Σ ist das Eingabealphabet, wobei $\Sigma \subseteq \Gamma$ und ¢ , $\sqcup \notin \Sigma$
- Γ ist das Arbeitsalphabet, wobei $Q \cap \Gamma = \emptyset$ und ¢ , $\sqcup \in \Gamma$
- q_o ist der Anfangszustand
- q_{acc} ist der akzeptierende Zustand
- q_{rej} ist der verwerfende Zustand
- $\delta : (Q \setminus \{q_{acc}, q_{rej}\} \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$ ist die Übergangsfunktion

Besonderheiten

Eine Aktion der Turingmaschine mit den Argumenten

- aktueller Zustand
- Symbol auf dem Band an der Position des Kopfes

besteht aus:

- einer Zustandsänderung
- dem Schreiben eines Symbols auf das Band an der Stelle, wo das Symbol gelesen wurde
- dem Bewegen des Kopfs nach Links (L), nach Rechts (R) oder gar nicht (N)

Das Band fängt ganz links mit dem Zeichen ¢ an. Dieses darf nie durch ein anderes Symbol ersetzt werden.
Das Band ist nach rechts unendlich lang, wobei nicht beschriftete Felder das Symbol $\sqcup$ enthalten.
Vor einer Berechnung befindet sich der gesamte Input der Berechnung direkt nach dem ¢ auf dem Band.

Konfigurationen

Eine Konfiguration von M ist ein Element aus:

$$(\{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^+) \cup (Q \cdot \{¢\} \cdot \Gamma^*)$$

Die beiden Möglichkeiten kommen daher, dass ein Element der zweiten Menge eine Konfiguration darstellt, wo der Kopf auf dem Symbol ¢ im Band steht. Die zweite Menge stellt alle anderen Konfigurationen dar. Für jedes Wort $x \in \Sigma^*$ ist $(q_0 ¢ x)$ die Anfangskonfiguration von M auf x.

Schritte

Ein Schritt R_M von M ist eine Relation auf Konfigurationen, definiert als:

- $x_1 \dots x_{i-1} q x_i x_{i+1} \dots x_n \; R_M \; x_1 \dots x_{i-1} p y x_{i+1} \dots x_n$
  falls $\delta(q, x_i) = (p, y, N)$
- $x_1 \dots x_{i-1} q x_i x_{i+1} \dots x_n \; R_M \; x_1 \dots x_{i-2} p x_{i-1} y x_{i+1} \dots x_n$
  falls $\delta(q, x_i) = (p, y, L)$
- $x_1 \dots x_{i-1} q x_i x_{i+1} \dots x_n \; R_M \; x_1 \dots x_{i-1} y p x_{i+1} \dots x_n$
  falls $\delta(q, x_i) = (p, y, R)$ für $i < n$
  $x_1 \dots x_{n-1} q x_n \; R_M \; x_1 \dots x_{n-1} y p \sqcup$
  falls $\delta(q, x_n) = (p, y, R)$

Die reflexive und transitive Hülle R_M^* von R_M ist analog zum EA definiert.

Berechnungen

Eine Berechnung von M ist eine Folge von Konfigurationen $C_1, C_2, \dots, C_m$, so dass $C_i \; R_M \; C_{i+1}, \; \forall i \in \{1, 2, \dots, m-1\}$. Eine Berechnung von M auf $x \in \Sigma^*$ beginnt mit der Konfiguration $q_0 ¢ x$ und ist entweder akzeptierend, nicht akzeptierend oder unendlich. Eine Berechung kann nicht auf einem anderen Zustand terminieren als $\{q_{acc}, q_{rej}\}$, da man immer noch weitere $\sqcup$ vom Band lesen kann.
Eine Berechung heisst akzeptierend, falls sie in $w_1 q_{acc} w_2$ endet. Eine Berechung heisst verwerfend, falls sie in $w_1 q_{rej} w_2$ endet. Eine nicht-akzeptierende Berechung auf x ist entweder eine unendliche oder eine verwerfende Berechung auf x.

Die von M akzeptierte Sprache $L(M)$ ist definiert als:

$$L(M) = \{w \in \Sigma^* \mid q_0 ¢ W \; R_M^* \; y q_{acc} z, \; y, z \in \Gamma^*\}$$

4.2 Sprachklassen

Die Klasse der rekursiv aufzählbaren Sprachen ist definiert als:

$$\mathcal{L}_{RE} = \{L(M) \mid M \text{ ist TM}\}$$

Eine Sprache $L \subseteq \Sigma^*$ heisst rekursiv oder entscheidbar, falls $L = L(M)$ für eine TM M, wobei für alle $x \in \Sigma^*$:

$$q_0 ¢ x \; R_M^* \; y q_{acc} z, \quad y, z \in \Gamma^*, \text{ falls } x \in L$$
$$q_0 ¢ x \; R_M^* \; u q_{rej} v, \quad u, v \in \Gamma^*, \text{ falls } x \notin L$$

Dies bedeutet, dass die Berechnung für alle x terminiert. Die Menge der entscheidbaren oder algorithmisch erkennbaren Sprachen ist definiert als:

$$\mathcal{L}_R = \{L(M) \mid M \text{ ist eine TM, die immer hält}\}$$

4.3 Funktionen

M berechnet die Funktion $F : \Sigma^* \to \Gamma^*$, falls

$$\forall x \in \Sigma^* : \; q_0 ¢ x \; R_M^* \; q_{acc} ¢ F(x)$$

Eine Funktion $F : \Sigma_1^* \to \Sigma_2^*$ heisst total berechenbar, falls eine TM existiert, die F berechnet. Man beachte, dass dies bedeutet, dass M immer hält, da es sinnlos wäre, einen berechneten Funktionswert zu verwerfen.

4.4 Mehrband-Turingmaschinen

Für jede positive, ganze Zahl k hat eine k-Band-Turingmaschine die Komponenten:

- endliche Kontrolle
- endliches Eingabeband mit Lesekopf
- k Arbeitsbänder, je mit eigenem Lese- und Schreibkopf

Eine Mehrband-Turingmaschine (MTM) mit $k = 1$ heisst Einband-Turingmaschine.
Am Anfang jeder Berechnung auf w enthält das Leseband das Wort ¢w\$, wobei ¢ der linke und \$ der rechte Rand ist. Die Maschine startet ausserdem in q_0, auf allen Arbeitsbändern steht ¢$\sqcup \sqcup \dots$ und die Lese- und Schreibköpfe stehen jeweils auf ¢.
Ein Schritt verläuft völlig analog zur Turingmaschine, ausser dass der Lesekopf weder über den linken noch über den rechten Rand hinausdarf, und man auf jedes Arbeitsband nach $\{L, R, N\}$ darf.
Eine Konfiguration der k-Band-Turingmaschine M wird also zu

$$(q, w, i, u_1, i_1, u_2, i_2, \dots, u_k, i_k)$$

und ist ein Element aus der Menge

$$Q \times \Sigma^* \times \mathbb{N} \times (\Gamma \times \mathbb{N})^k$$

mit der Bedeutung:

- M ist im Zustand q
- auf dem Eingabeband steht ¢w\$ und der Lesekopf befindet sich an i-ter Stelle (beginnend mit 0)
- für $j \in \{1, 2, \dots, k\}$ ist der Inhalt des j-ten Bandes u_j und der Lese- und Schreibkopf befindet sich jeweils an der Stelle i_j (beginnend mit 0)

Die Transitionsfunktion wird also zu:

$$\delta : \; Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma^k \to Q \times \{L, R, N\} \times (\Gamma \times \{L, R, N\})^k$$

Eine Berechung wird akzeptiert, falls man irgendwann im Verlauf einer Berechung auf q_{acc} landet und verworfen, falls man irgendwann auf q_{rej} landet. M akzeptiert w, falls M in q_{acc} landet, andernfalls verwirft sie w (auch bei unendlichen Berechnungen).

4.5 Äquivalenz der TM-Modelle

Wir definieren, dass zwei Maschinen A und B, die auf dem gleichen Eingabealphabet Σ arbeiten, äquivalent heissen, falls für jedes $x \in \Sigma^*$:

1. A akzeptiert $x \Leftrightarrow B$ akzeptiert x
2. A verwirft $x \Leftrightarrow B$ verwirft x
3. A arbeitet unendlich lange auf $x \Leftrightarrow B$ arbeitet unendlich lange auf x

Wir halten an dieser Stelle fest, dass die Tatsache $L(A) = L(B)$ keine Garantie dafür ist, dass A und B äquivalent sind, weil der dritte Punkt eventuell verletzt sein könnte.
Wir definieren, dass 2 Maschinenmodelle $\mathcal{A}$ und $\mathcal{B}$ äquivalent heissen, falls:

- für jedes $A \in \mathcal{A}$ existiert ein äquivalentes $B \in \mathcal{B}$
- für jedes $C \in \mathcal{B}$ existiert ein äquivalentes $D \in \mathcal{A}$

Im Buch wurde folgendes bewiesen:

- zu jeder TM A existiert eine äquivalente 1-Band-TM B
- für jede MTM A existiert eine äquivalente TM B

Daraus folgt nun, dass die Maschinenmodelle von Turingmaschinen und Mehrbandturingmaschinen äquivalent sind. Eine weitere wichtige Eigenschaft der Klasse von Turingmaschinen ist, dass die Äquivalenz zwischen Turingmaschinen und der Sprache Assembler bewiesen wurde. Da man jedes Programm, das in einer High-Level-Programmiersprache verfasst ist, zuerst in Assembler übersetzt, bevor man es ausführt, folgt direkt, dass Turingmschinen und alle Programmiersprachen gleich mächtig sind.

4.6 Church'sche These

Die These besagt, dass die Turingmaschinen die Formalisierung des Begriffes Algorithmus sind. Das heisst, dass die Klasse der rekursiven Sprachen mit der Klasse der algorithmisch erkennbaren Sprachen übereinstimmt.
Die Church'sche These ist nicht beweisbar, sie ist das einzige Axiom der Informatik. Falls sie widerlegt werden sollte, müsste die gesamte Grundlage der theoretischen Informatik revidiert werden.

4.7 Nichtdeterministische Turingmaschinen

Wie beim endlichen Automaten führen wir auch bei Turingmaschinen den Nichtdeterminismus ein. Für jedes Argument besteht jetzt die Möglichkeit zur Auswahl aus endlich vielen Aktionen. Formal sind die nichtdeterministischen Turingmaschinen (NTM) als 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ definiert, wobei:

- $Q, \Sigma, \Gamma, q_0, q_{acc}, q_{rej}$ die gleiche Bedeutung wie bei den TM haben
- $\delta : (Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$ die Übergangsfunktion von M ist mit der Eigenschaft:

$$\delta(p, ¢) \subseteq \{(q, ¢, X) \mid q \in Q, X \in \{R, N\}\}$$

Eine Konfiguration von M ist ein Element aus der Menge:

$$(\{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^*) \cup (Q \cdot \{¢\} \cdot \Gamma^*)$$

mit identischer Bedeutung wie bei der TM. Eine Konfiguration ist akzeptierend, falls sie q_{acc} enthält und ist verwerfend, falls sie q_{rej} enthält.
Der Schritt und die Berechnung bei der NTM sind gleich definiert wie bei den TM, ausser dass die δ-Funktion nicht mehr eindeutig ist, sondern ≥ 0 Möglichkeiten zur Auswahl stehen. Ein Wort w wird von einer NTM akzeptiert, falls eine akzeptierende Berechnung auf w existiert. Die von der NTM M akzeptierte Sprache ist:

$$L(M) = \{w \in \Sigma^* \mid q_0 ¢ w \ R_M^* \ y q_{acc} z, \text{ für } y, z \in \Gamma^*\}$$

Ein Berechnungsbaum $T_{M,x}$, der einer Berechnung der NTM M auf einem Wort $x \in \Sigma^*$ entspricht, ist ein potentiell unendlicher gerichteter Baum, der wie folgt definiert ist:

- Jeder Knoten von $T_{M,x}$ stellt eine Konfiguration dar
- Die Wurzel ist die Ausgangskonfiguration $q_0 ¢ x$
- Ein Knoten, der eine Konfiguration C darstellt, besitzt genauso viele Kinderknoten, wie C Nachfolgekonfigurationen besitzt. Diese Kinder stellen die Nachfolgekonfigurationen dar

Sei M eine NTM. Dann existiert eine TM A, so dass $L(M) = L(A)$. Falls M keine unendlichen Berechnungen durchführt, wird A für jeden Input halten.

4.8 Kodierung von Turingmaschinen

Zunächst entwickeln wir eine Methode zur Abbildung von Turingmaschinen auf das Alphabet $\{0, 1, \#\}$. Sei die zu abbildende Turingmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$, wobei $Q = \{q_0, q_1, \ldots, q_m, q_{acc}, q_{rej}\}$ und $\Gamma = \{A_1, A_2, \ldots, A_r\}$. Die einzelnen Symbole werden wie folgt kodiert:

$$\begin{aligned} C(q_i) &= 10^{i+1}1 \\ C(q_{acc}) &= 10^{m+2}1 \\ C(q_{rej}) &= 10^{m+3}1 \\ C(A_j) &= 110^j11, \quad j = 1, 2, \ldots, r \\ C(N) &= 1110111 \\ C(R) &= 1110^2111 \\ C(L) &= 1110^3111 \end{aligned}$$

Eine Transition (für $\alpha \in \{L, R, N\}$) wird dargestellt als:

$$C(\delta(p, A_l) = (q, A_s, \alpha)) = \#C(p)C(A_l)C(q)C(A_s)C(\alpha)\#$$

Nun können wir eine beliebige Turingmaschine eindeutig kodieren, indem wir die Anzahl Zustände, dann die Anzahl Symbole aus dem Arbeitsalphabet und zum Schluss die Transitionen angeben:

$$C(M) = \#0^{m+3}\#0^r\#\#C(Trans_1)\#C(Trans_2)\#\ldots$$

Um diese Kodierung in eine Kodierung auf Σ_{bool} zu erhalten, wenden wir folgenden Homomorphismus an $h : \{0, 1, \#\}^* \to (\Sigma_{bool})^*$ an:

$$h(\#) = 01, \quad h(0) = 00, \quad h(1) = 11$$

Für jede TM M ist die Kodierung von M definiert als:

$$Kod(M) = h(C(M))$$

Die Menge der Kodierungen von Turingmaschinen ist:

$$\text{KodTM} = \{Kod(M) \mid M \text{ ist TM }\}$$

Im folgenden ist A_{ver} in Programm, was für jeden Bitstring entscheidet, ob es eine Turingmaschine darstellt oder nicht.
Sei $x \in \Sigma_{bool}^*$. Für jedes $i \in \mathbb{N}_+$ ist x die Kodierung der i-ten TM, falls:

- $x = Kod(M)$ für eine TM M
- die Menge $\{y \in (\Sigma_{bool})^* \mid y < x \text{ in kanonischer Reihenfolge }\}$ enthält genau $i - 1$ Kodierungen von TM

Die i-te TM heisst M_i und hat die Ordnung i. Es ist ausserdem nicht schwer, mit A_{ver} ein Programm zu schreiben, welches die Kodierung der TM der Ordnung i generiert.

5 Berechenbarkeit

5.1 Mengenlehre und Unendlichkeit

Seien A und B zwei Mengen. Wir sagen $|A| \leq |B|$, falls eine injektive Funktion $f : A \to B$ existiert. Es gilt:

$$|A| = |B| \quad \Leftrightarrow \quad (|A| \leq |B|) \wedge (|B| \leq |A|)$$

Falls $|A| \leq |B|$, aber keine injektive Abbildung von B nach A existiert, dann sagen wir $|A| < |B|$.

Eine Menge A heisst abzählbar, falls A endlich ist oder $|A| = |\mathbb{N}|$. Daraus folgt zum Beispiel, dass für ein beliebiges Alphabet Σ gilt: Σ^* ist abzählbar.

Für folgende Mengen wurde im Buch bewiesen, dass sie abzählbar sind:

- KotTM
- $\mathbb{Q}^+$
- $(\mathbb{N} \setminus 0) \times (\mathbb{N} \setminus 0)$

Die Menge $[0, 1]$ ist nicht abzählbar. Dies kann man mit der Diagonalisierungsmethode beweisen. Die Idee besteht darin, die unendliche Darstellung aller Zahlen der Menge $[0, 1]$ zu betrachten und eine Zahl zu konstruieren, die sich zu jeder anderen Zahl an jeweils mindestens eine Stelle unterscheidet. Dies ist aber nur möglich, da jede Zahl, die so konstruiert wird, wieder in $[0, 1]$ enthalten sein muss.

Die Menge $\mathcal{P}((\Sigma_{bool})^*)$ ist nicht abzählbar. Dies beweisen wir, indem wir eine injektive Abbildung von $[0, 1]$ nach $\mathcal{P}((\Sigma_{bool})^*)$ konstruieren, womit wir zeigen, dass $|\mathcal{P}((\Sigma_{bool})^*)| \geq [0, 1]$. Wie im Buch beschrieben wird, lässt sich eine solche Abbildung relativ einfach herstellen, woraus sich dann auch die nicht rekursive Sprache $\mathcal{L}_{diag}$ ergibt:

$$\mathcal{L}_{diag} \notin \mathcal{L}_{RE}$$

Aus dem obigen Beweis folgt dann direkt, dass nichtrekursive Sprachen existieren:

$$|\text{KodTM}| < \mathcal{P}((\Sigma_{bool})^*)$$

5.2 Die Methode der Reduktion

Eine Sprache $L_1 \subseteq \Sigma_1^*$ ist auf die Sprache $L_2 \subseteq \Sigma_2^*$ rekursiv reduzierbar, $L_1 \leq_R L_2$, falls:

$$L_2 \in \mathcal{L}_R \quad \Rightarrow \quad L_1 \in \mathcal{L}_R$$

Eine Sprache $L_1 \subseteq \Sigma_1^*$ ist auf die Sprache $L_2 \subseteq \Sigma_2^*$ EE-reduzierbar, $L_1 \leq_{EE} L_2$, falls eine TM M existiert, die die Abbildung $f_M : \Sigma_1^* \to \Sigma_2^*$ für alle $x \in \Sigma_1^*$ berechnet:

$$x \in L_1 \quad \Leftrightarrow \quad f_M(x) \in L_2$$

Konkret bedeutet dies, dass man in der Maschine für L_1 zuerst den Input mit M bearbeitet, dann die Maschine für L_2 verwendet, den Output aber nicht mehr bearbeiten darf, sondern direkt als Output der Maschine für L_1 benutzen muss.
Es gilt:

$$L_1 \leq_{EE} L_2 \quad \Rightarrow \quad L_1 \leq_R L_2$$

Für jede Sprache L gilt:

$$L \leq_R L^C, \quad L^C \leq_R L$$

Dabei reicht es, die Richtung $L^C \leq_R L$ zu zeigen, da die andere Richtung direkt aus $(L^C)^C = L$ folgt. Den Beweis liefert der Algorithmus für L, wobei q_{acc} und q_{rej} vertauscht werden. Daraus folgt direkt $(\mathcal{L}_{diag})^C \notin \mathcal{L}_R$.

Es gilt $(\mathcal{L}_{diag})^C \in \mathcal{L}_{RE}$. Dies wird gezeigt, indem eine TM für $(\mathcal{L}_{diag})^C$ gebaut wird. Zu beachten, ist dass diese nur ein Programm ist, kein Algorithmus.
Daraus folgt, dass $(\mathcal{L}_{diag})^C \in (\mathcal{L}_{RE} \setminus \mathcal{L}_R)$, also auch:

$$\mathcal{L}_R \subsetneq \mathcal{L}_{RE}$$

Die universelle Sprache ist definiert als:

$$L_U = \{Kod(M)\#w \mid w \in (\Sigma_{bool})^*, M \text{ akzeptiert } w\}$$

Für diese Sprache existiert ein Turingmaschine U mit $L(U) = L_U$. Dies bedeutet, dass $L_U \in \mathcal{L}_{RE}$. Die Turingmaschine U simuliert im Wesentlichen die Arbeit der Turingmaschine M auf w. Entsprechend akzeptiert sie, verwirft sie, oder rechnet unendlich lange.

Für die universelle Sprache gilt $L_U \notin \mathcal{L}_R$. Dies bedeutet im Wesentlichen, dass man ohne Simulation nicht wissen kann, ob eine Turingmaschine ein Wort akzeptieren wird oder nicht. Dies wird dadurch gezeigt, dass mittels Reduktion $(\mathcal{L}_{diag}) \leq_R L_U$ bewiesen wird. Die Idee dahinter, ist dass man mit L_U entscheiden kann, ob ein Wort in $(\mathcal{L}_{diag})^C$ ist oder nicht. Wir wissen aber, dass $(\mathcal{L}_{diag})^C \notin \mathcal{L}_R$.

Eine weitere Erkenntnis, ist dass für das Komplement einer beliebigen Sprache $L \in \mathcal{L}_{RE}$ gelten muss:

$$L^C \notin \mathcal{L}_{RE}$$

Falls dies nämlich nicht der Fall wäre, könnte ein Programm A für L und ein Programm B für L^C existieren. Jedes Wort in $\Sigma_A = \Sigma_B$ muss von entweder A oder B akzeptiert werden, also könnte man beide Maschinen parallel simulieren und hätte nach endlich vielen Schritten eine Antwort, da mindestens eine Maschine terminieren muss.

Das Halteproblem ist das Entscheidungsproblem $(\{0, 1, \#\}^*, L_H)$, wobei:

$$L_H = \left\{Kod(M)\#x \mid x \in \{0,1\}^*, M \text{ hält auf } x\right\}$$

Trivialerweise gilt $L_H \in \mathcal{L}_{RE}$, weiter gilt aber auch $L_H \notin \mathcal{L}_R$. Dies zeigt man dadurch, dass man $L_U \leq_R L_H$ beweist, wobei wir wissen, dass $L_U \notin \mathcal{L}_R$.

Wir betrachten die Sprache(n):

$$L_{empty} = \{Kod(M) \mid L(M) = \emptyset\}$$
$$(L_{empty})^C = \left\{x \in (\Sigma_{bool})^* \mid x \neq Kod(\overline{M}) \vee x = Kod(M), L(M) \neq \emptyset\right\}$$

Es gilt $(L_{empty})^C \in \mathcal{L}_{RE}$. Dies kann man mit einer nichtdeterministischen Turingmaschine M zeigen, die genau $(L_{empty})^C$ akzeptiert. Sie rät ein Wort y, und falls dieses von M akzeptiert wird, akzeptiert sie, sonst verwirft sie.
Durch $L_U \leq_{EE} (L_{empty})^C$ kann man zeigen, dass $(L_{empty})^C \notin \mathcal{L}_R$ ist. Somit folgt:

- $(L_{empty})^C \in \mathcal{L}_{RE} \setminus \mathcal{L}_R$
- $L_{empty} \notin \mathcal{L}_{RE}$

5.3 Satz von Rice

Eine Sprache $L \subseteq \{Kod(M) \mid M \text{ ist eine } TM\}$ heisst semantisch nichttriviales Entscheidungsproblem über Turingmaschinen, falls:

- Es gibt eine TM M_1, sodass $Kod(M_1) \in L$
- Es gibt eine TM M_2, sodass $Kod(M_2) \notin L$
- Für zwei TM A und B impliziert $L(A) = L(B)$ die Äquivalenz:

$$Kod(A) \in L \quad \Leftrightarrow \quad Kod(B) \in L$$

Wir definieren auch noch die folgende Sprache als spezifisches Halteproblem:

$$L_{H,\lambda} = \{Kod(M) \mid M \text{ hält auf } \lambda\}$$

Um zu zeigen, dass $L_{H,\lambda}$ nicht entscheidbar ist, wird $L_H \leq_{EE} L_{H,\lambda}$ dadurch gezeigt, dass die Maschine für L_H gebaut wird, die im Inneren eine Maschine für $L_{H,\lambda}$ enthält. Im Inneren von L_H generiert eine Maschine A für eine Eingabe x, die nicht die Form $Kod(M)\#w$ hat, eine Maschine H_x die auf jeder Eingabe unendlich arbeitet. Falls x die gewünschte Form besitzt, wird die von A generierte Maschine H_x für jeden Input einfach die Berechnung von M auf x simulieren. Die Maschine A übergibt dann $Kod(H_x)$ dem $L_{H,\lambda}$-Baustein.

Der Satz von Rice besagt, dass jedes semantische nichttriviale Entscheidungsproblem über Turingmaschinen unentscheidbar ist. Der Beweis ist so aufgebaut, dass für jedes Entscheidungsproblem L entweder $L_{H,\lambda} \leq_{EE} L$ oder $L_{H,\lambda} \leq_{EE} L^C$ gilt.

Aus dem Satz von Rice folgt, dass die Sprache $L_{EQ} = \{Kod(M)\#Kod(N) \mid L(M) = L(N)\}$ nicht entscheidbar ist, also $L_{EQ} \notin \mathcal{L}_R$.

5.4 Methode der Kolmogorov-Komplexität

Das Problem, für ein $x \in \Sigma^*_{bool}$ die Kolmogorov-Komplexität $K(x)$ zu bestimmten, ist nicht algorithmisch entscheidbar. Dies wird im Buch indirekt dadurch bewiesen, dass ein Programm angegeben wird, das für ein n das erste Wort x_n generiert, für welches $K(x_n) \geq n$ gilt.
Da dieses Programm aber nur den Input n benötigt und x_n generiert, ist $K(x_n) \leq \lceil \log_2 n \rceil + c$. Nun kann $\lceil \log_2 n \rceil + c \geq K(x_n) \geq n$ aber nur für endlich viele Zahlen n gelten. Dies ist ein Widerspruch zur Annahme, dass ein Algorithmus zur Bestimmung der Kolmogorov-Komplexität existiert.

Mit dieser Methode kann man zum Beispiel zeigen: falls $L_H \in \mathcal{L}_R$, dann existiert ein Algorithmus zur Berechnung von $K(x)$ für alle $x \in \Sigma^*_{bool}$.

6 Komplexitätstheorie

6.1 Komplexitätsmasse

Bei der Betrachtung der folgenden beiden Komplexitätsmasse ist anzumerken, dass bei beiden die Grösse des Arbeitsalphabets keine Rolle spielt. Folglich kann man die Speicher- und Zeitkomplexität von Turingmaschinen reduzieren, indem man das Alphabet vergrössert. In der Praxis geht dies natürlich nicht, da echte physische Rechner fixe Arbeitsalphabete haben.

Zeitkomplexität

Sei M eine TM oder MTM, die immer hält und sei Σ ihr Eingabealphabet. Sei $x \in \Sigma^*$ und sei $D = C_1, C_2, \ldots, C_k$ die Berechnung von M auf x. Dann gelten folgende Definitionen:

- Die Zeitkomplexität der Berechnung von M auf x:
  $\text{Time}_M(x) = k - 1$
- Die Zeitkomplexität von M:
  $\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid x \in \Sigma^n\}$

Sei k eine positive ganze Zahl. Für jede k-Band MTM A, die immer hält, existiert eine k-Band-MTM B, so dass $L(A) = L(B)$ und

$$\text{Time}_A(n) \leq \frac{\text{Time}_M}{2} + 2n$$

Speicherkomplexität

Sei M eine k-Band MTM, die immer hält und sei Σ ihr Eingabealphabet. Sei $x \in \Sigma^*$ und sei $D = C_1, C_2, \ldots, C_l$ die Berechnung von M auf x. Sei $C = (q, x, i, \alpha_1, i_1, \alpha_2, i_2, \ldots, \alpha_k, i_k)$ eine Konfiguration. Dann gelten folgende Definitionen:

- Die Speicherkomplexität der Konfiguration C:
  $\text{Space}_M(C) = \max\{|\alpha_i| \mid i = 1, \ldots, k\}$
- Die Speicherkomplexität von M auf x:
  $\text{Space}_M(x) = \max\{\text{Space}_M(C_i) \mid i = 1, \ldots, l\}$
- Die Speicherkomplexität von M:
  $\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid x \in \Sigma^n\}$

Sei k eine positive ganze Zahl. Für jede k-Band MTM A, die immer hält, existiert eine 1-Band-TM B, so dass

$$\text{Space}_B(n) \leq \text{Space}_A(n)$$

Sei k eine positive ganze Zahl. Für jede k-Band MTM A, die immer hält, existiert eine k-Band-MTM B, so dass $L(A) = L(B)$ und

$$\text{Space}_B(n) \leq \frac{\text{Space}_A(n)}{2} + 2$$

Grundlegende Asymptotik

Da die Komplexitätsmasse relativ grob sind, interessiert uns eher die asymptotische Analyse als die genauen Funktionswerte. Im Folgenden sind $f(n), g(n), r(n)$ Funktionen von $\mathbb{N} \to \mathbb{R}^+$:

- Funktionen, die nicht schneller als f wachsen:
  $O(f(n)) = \{r(n) \mid \exists n_0, c \in \mathbb{N},\ \forall n \geq n_0 :\ r(n) \leq c \cdot f(n)\ \}$
- Funktionen, die mindestens so schnell wachsen, wie f:
  $\Omega(f(n)) = \left\{r(n) \mid \exists n_0, c \in \mathbb{N},\ \forall n \geq n_0 :\ r(n) \geq \frac{1}{c} \cdot f(n)\ \right\}$
- Funktionen, die gleich schnell wachsen, wie f:
  $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- Funktionen, die langsamer als f wachsen:

$$o\left(f(n)\right) = \left\{ g(n) \,\middle|\, \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \right\}$$

Komplexitätsschranken für Probleme

Um die Komplexität eines Problems zu definieren, würde es sich anbieten, die Komplexität des optimalen Algorithmus für das Problem zu betrachten. Der folgende Satz macht solche Ansätze zunichte:

Es gibt ein Entscheidungsproblem Z, so dass für jede MTM A, die Z entscheidet, eine MTM B existiert, die Z auch entscheidet, wobei für unendlich viele n gilt:

$$\text{Time}_B(n) \leq \log_2(\text{Time}_A(n))$$

Da es also Algorithmen gibt, die man immer wesentlich verbessern kann, spricht man bei der Komplexität von Problemen von oberen und unteren Schranken:

- $O(f(n))$ ist eine obere Schranke für die Zeitkomplexität einer Sprache L, falls eine MTM A existiert, die L in $O(f(n))$ entscheidet
- $\Omega(f(n))$ ist eine untere Schranke für die Zeitkomplexität einer Sprache L, falls für jede MTM A, die L entscheidet, gilt: $\text{Time}_A \in \Omega(f(n))$

Eine MTM A heisst optimal für L, falls $\Omega(f(n))$ eine untere Schranke für die Zeitkomplexität von L ist und L von A in $O(f(n))$ entschieden wird.

Uniformes und logarithmisches Kostenmass

Beim Uniformen Kostenmass ist der Preis jeder Operation bezüglich Zeitkomplexität 1 und die Speicherkomplexität ist gleich der Anzahl benutzter Variablen. Dieses Mass ist besonders grob und macht in Fällen Sinn, wo die Operanden die Grösse eines Rechnerwortes nicht überschreiten können.

Das logarithmische Kostenmass reflektiert viel besser die genauen Kosten einer Berechnung. Dabei misst man die zeitlichen Kosten einer Operation als die Summe der Längen der vorkommenden Operanden. Die Zeitkomplexität einer Berechnung ist dann die Summe der Kosten aller Operationen.
Die Speicherkomplexität ist die Summe der Darstellungslängen der Inhalte der benutzten Variablen.

6.2 Komplexitätsklassen und die Klasse P

Definition der Klassen

Für alle Funktionen f, g von $\mathbb{N} \to \mathbb{R}^+$ definiert man:

$$\begin{aligned}
\text{TIME}(f) &= \{L(B) \mid B \text{ ist MTM mit } \text{Time}_B(n) \in O(f(n))\} \\
\text{SPACE}(f) &= \{L(B) \mid B \text{ ist MTM mit } \text{Space}_B(n) \in O(f(n))\} \\
\text{DLOG} &= \text{SPACE}(\log_2 n) \\
\text{P} &= \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c) \\
\text{PSPACE} &= \bigcup_{c \in \mathbb{N}} \text{SPACE}(n^c) \\
\text{EXPTIME} &= \bigcup_{d \in \mathbb{N}} \text{TIME}\left(2^{n^d}\right)
\end{aligned}$$

Konsturierbarkeit von Funktionen

Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ heisst platzkonstruierbar, falls eine 1-Band-MTM existiert, so dass:

- $\forall n \in \mathbb{N} : \text{Space}_M(n) \leq f(n)$
- für die Eingabe $0^n, n \in \mathbb{N}$ generiert M das Wort $0^{(f(n))}$ auf dem Arbeitsband

Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ heisst zeitkonstruierbar, falls eine k-Band-MTM existiert, so dass:

- $\forall n \in \mathbb{N} : \text{Time}_M(n) \in O(f(n))$
- für die Eingabe $0^n, n \in \mathbb{N}$ generiert M das Wort $0^{(f(n))}$ auf dem ersten Arbeitsband

Sei die Funktion f platzkonstruierbar. Sei M eine MTM mit $\text{Space}_M(x) \leq f(|x|)$ für alle $x \in L(M)$. Dann existiert eine MTM A mit $L(A) = L(M)$ und:

$$\forall n : \ \text{Space}_A(n) \leq f(n)$$

Sei die Funktion f zeitkonstruierbar. Sei M eine MTM mit $\text{Time}_M(x) \leq f(|x|)$ für alle $x \in L(M)$. Dann existiert eine MTM A mit $L(A) = L(M)$ und:

$$\forall n : \ \text{Time}_A(n) \in O(f(n))$$

Die beiden obigen Lemmata werden sehr ähnlich bewiesen. Die Idee besteht darin, in A den Funktionswert von f zuerst zu berechnen, und dann die Arbeit von M zu simulieren, wobei abgebrochen und verworfen wird, falls mehr Zeit (Platz) als der Funktionswert benötigt wird.

Beziehungen zwischen den Klassen

Da eine MTM M in der Zeit $\text{Time}_M(n)$ nicht mehr als $\text{Time}_M(n)$ Felder beschriften kann, muss für jede Funktion $t : \mathbb{N} \to \mathbb{R}^+$ die Beziehung gelten:

$$\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$$

Für jede platzkonstruierbare Funktion f mit $f(n) \geq log_2(n)$ gilt:

$$\text{SPACE}(f(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{f(n)})$$

Dabei beachte man, dass in der Definition von SPACE verlangt wird, dass die Turingmaschine immer hält. Dieser Satz gilt aber auch für MTM, die unendliche Berechnungen besitzen.

Aus den obigen beiden Sätzen folgt die Hierarchie:

$$\text{DLOG} \subseteq \text{P} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

Seien f und g Funktionen mit den folgenden Eigenschaften:

- $g(n) \geq \log_2(n)$
- $g(n)$ ist platzkonstruierbar
- $f(n) = o(g(n))$

dann gilt die Beziehung:

$$\text{SPACE}(f) \subsetneq \text{SPACE}(g)$$

Seien f und g Funktionen mit den folgenden Eigenschaften:

- g ist zeitkonstruierbar
- $f(n) \cdot \log_2(f(n)) = o(g(n))$

dann gilt die Beziehung:

$$\text{TIME}(f) \subsetneq \text{TIME}(g)$$

Praktische Lösbarkeit

Ein Problem heisst praktisch lösbar genau dann, wenn ein polynomieller Algorithmus zu seiner Lösung existiert. Die Klasse P ist die Klasse der praktisch entscheidbaren Probleme. Zu dieser Entscheidung führten im Wesentlichen 2 Gründe:

- Obwohl Probleme in $O(n^c)$ für ein grosses c praktisch unlösbar sind, hat die Erfahrung gezeigt, dass man in fast allen Fällen einen äquivalenten Algorithmus in $O(n^6)$ finden kann.
- Die Klasse der praktisch lösbaren Probleme muss insofern robust sein, dass die unabhängig des gewählten Berechungsmodells ist. Da die bestehenden Modelle alle polynomiell auf einander reduzierbar sind, muss die Klasse der praktisch lösbaren Probleme als "mindestens" polynomiell gewählt werden.

6.3 Nichtdeterministische Komplexitätsmasse

Zeitkomplexität

Sei M eine nichtdeterministische MTM und sei Σ ihr Eingabealphabet. Sei $x \in L(M) \subseteq \Sigma^*$. Dann gelten folgende Definitionen:

- Die Zeitkomplexität $\text{Time}_M(x)$ ist die Zeitkomplexität der kürzesten akzeptierenden Berechnung von M auf x
- Die Zeitkomplexität von M:
  $\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid x \in L(M) \cap \Sigma^n\}$

Speicherplatzkomplexität

Sei M eine nichtdeterministische MTM und sei Σ ihr Eingabealphabet. Sei $x \in L(M) \subseteq \Sigma^*$ und sei $D = C_1, C_2, \ldots, C_l$ eine akzeptierende Berechnung von M auf x. Sei $\text{Space}_M(C_i)$ die Spricherplatzkomplexität der Konfiguration C_i. Dann gelten folgende Definitionen:

- Die Speicherkomplexität der Berechnung D:
  $\text{Space}_M(D) = \max\{\text{Space}_M(C_i) \mid i = 1, \ldots, l\}$
- Die Speicherkomplexität von M auf x:
  $\text{Space}_M(x) = \min\{\text{Space}_M(D) \mid D$ ist akzeptierende Berechnung von M auf $x\}$
- Die Speicherkomplexität von M:
  $\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid x \in L(M) \cap \Sigma^n\}$

Definition der Klassen

Für alle Funktionen f, g von $\mathbb{N} \to \mathbb{R}^+$ definiert man:

$$\text{NTIME}(f) = \{L(B) \mid B \text{ ist N-MTM mit } \text{Time}_B(n) \in O(f)\}$$
$$\text{NSPACE}(f) = \{L(B) \mid B \text{ ist N-MTM mit } \text{Space}_B(n) \in O(f)\}$$
$$\text{NLOG} = \text{NSPACE}(\log_2 n)$$
$$\text{NP} = \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c)$$
$$\text{NPSPACE} = \bigcup_{c \in \mathbb{N}} \text{NSPACE}(n^c)$$

Beziehungen zwischen den Klassen

Für alle Funktionen $s(n)$ und $t(n)$ mit $s(n) \geq \log_2(n)$ gilt:

$$\text{NTIME}(t) \subseteq \text{NSPACE}$$
$$\text{NSPACE}(s) \subseteq \bigcup_{c \in \mathbb{N}} \text{NTIME}(c^{s(n)})$$

Für jede zeitkonstruierbare Funktion t und jede platzkonstruierbare Funktion s mit $s(n) \geq \log_2(n)$ gilt:

$$\text{TIME}(t) \subseteq \text{NTIME}(t)$$
$$\text{SPACE}(t) \subseteq \text{NSPACE}(t)$$
$$\text{NTIME}(s(n)) \subseteq \text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$$

Also folgt direkt:

$$\text{NP} \subseteq \text{PSPACE}$$

Für jede platzkonstruierbare Funktion $s(n)$ mit $s(n) \geq \log_2(n)$ gilt:

$$\text{NSPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$$

Also folgt direkt:

$$\text{NLOG} \subseteq \text{P} \quad \text{und} \quad \text{NSPACE} \subseteq \text{EXPTIME}$$

Der Satz von Savitch besagt, dass für jede platzkonstruierbare Funktion s mit $s(n) \geq \log_2(n)$ gilt:

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$$

Es folgt direkt:

$$\text{PSPACE} = \text{NPSPACE}$$

Zusammenfassend gilt nun die Hierarchie:

$$\text{DLOG} \subseteq \text{NLOG} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

6.4 NP und Beweisverifikation

Die Zeitkomplexität von deterministischen Berechnungen entpricht der nötigen Zeitkomplexität, einen Beweis herzustellen, da eine akzeptierende deterministische Berechnung offensichtlich einen Beweis der Aussage liefert.
In diesem Kapitel wird gezeigt, dass die nichtdeterministischen Berechnungen der zur Beweisverifikation nötigen Zeitklasse entsprechen.

Wir definieren zunächst die Sprache:

$\text{SAT} = \{x \in (\Sigma_{logic})^* \mid x$ kodiert eine erfüllbare Formel in KNF $\}$

Eine nichtdeterministische MTM, die SAT löst, rät zuerst eine Belegung für jede Variable und verifiziert diese anschliessend deterministisch. Da das Raten nicht länger geht, als die Anzahl Belegungen, reduziert sich die Zeit der nichtdeterministischen Berechnung auf die deterministische Verifikation.
Ausschalggebend ist, dass man zeigen kann, dass jede Berechnung einer nichtdeterministischen TM in einen ersten "Rateteil" und einen deterministischen zweiten Verifikationsteil aufgeteilt werden kann.

Sei $L \subseteq \Sigma^*$ und sei $p : \mathbb{N} \to \mathbb{N}$ eine Funktion. Ein Algorithmus A ist ein p-Verifizierer von L, $V(A) = L$, falls A wie folgt auf allen Eingaben von $\Sigma^* \times (\Sigma_{bool})^*$ arbeitet:

- Für jede Eingabe (w, x) gilt: $\text{Time}_A(w, x) \leq p(|w|)$
- Falls $w \in L$ existiert ein $x \in (\Sigma_{bool})^*$ mit $|x| \leq p(|w|)$ und $(w, x) \in L(A)$. Das Wort x ist dann der Beweis für die Behauptung $w \in L$
- Falls $w \notin L$ gilt für alle $z \in (\Sigma_{bool})^* : (y, z) \notin L(A)$

Daraus folgt:

$$V(A) = \{w \in \Sigma^* \mid \exists x \text{ mit } |x| \leq p(|w|),\ (w, x) \in L(A)\}$$

Falls $p(n) \in O(n^k)$, so heisst A ein Polynomialzeitverfizierer. Wir definieren die Klasse aller Sprachen, die in Polynomialzeit verfizierbar sind:

$$\text{VP} = \{V(A) \mid A \text{ ist ein Polynomialzeit-Verifizierer}\}$$

Nacht Satz 6.8 im Buch gilt: VP = NP

Daraus folgt nun, dass die Frage nach P = NP äquivalent zur Frage ist, ob es einfacher ist, Beweise zu verfizieren als zu erstellen.

6.5 NP-Vollständigkeit

Zunächst einige Argumente für die Glaubwürdigkeit der Aussage P $\subsetneq$ NP:

- Inutitive Annahme, dass Beweisverifikation einfacher ist, als Beweisherstellung
- Langjährige Erfahrung, dass keine polynomiellen Algorithmen für NP-Probleme bekannt sind
- Ungläubigkeit, dass der Mangel eines Beweises für P = NP eine Folge der Unfähigkeit der Mathematiker ist

Seien L_1 und L_2 zwei Sprachen. Wir sagen, dass $L_1 \subseteq \Sigma_1^*$ polynomiell auf $L_2 \subseteq \Sigma_2^*$ reduzierbar ist, $L_1 \leq_p L_2$, falls ein polynomieller Algorithmus A existiert, der für alle $x \in \Sigma_1^*$ ein Wort $A(x) \in \Sigma_2^*$ berechnet, so dass:

$$x \in L_1 \ \Leftrightarrow\ A(x) \in L_2$$

A heisst polynomielle Reduktion von L_1 auf L_2 genannt.

Eine Sprache L heisst NP-schwer, falls für alle Sprachen $L' \in$ NP gilt: $L' \leq_p L$. Falls die Sprache L zusätzlich selbst in NP ist, heisst sie NP-vollständig.

Es folgt direkt, dass:

$$(L \in P\ \wedge\ L \text{ ist NP-schwer})\ \rightarrow\ P = NP$$