TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Theoretische Informatik, Lehrstuhl Automatentheorie

Skript

Theoretische Informatik und Logik

Modul INF-D-330, INF-B-290

Teil 1 - Berechenbarkeit

Teil 2 - Komplexitätstheorie

Prof. Dr. Franz Baader

Sommersemester 2020

(letzte LaTeX-Bearbeitung 27. März 2020: Dr. Anton Claußnitzer)

Inhaltsverzeichnis

Organisation der Lehrveranstaltung

Die Lehrveranstaltung

„Theoretische Informatik und Logik“
(Modul INF-D-330, INF-B-290)

gliedert sich in

- **Teil 1:** Berechenbarkeit,
- **Teil 2:** Komplexitätstheorie, und
- **Teil 3:** Pradikatenlogik erster Stufe.

Verantwortlicher Hochschullehrer:

Prof. Dr.-Ing. Franz Baader

Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
01062 Dresden

`https://tu-dresden.de/ing/informatik/thi/lat`
`/die-professur/franz-baader`

Aktuelle Informationen zur Lehrveranstaltung:

`https://tu-dresden.de/ing/informatik/thi/lat`
`/studium/lehrveranstaltungen/sommersemester-2020`
`/theoretische-informatik-und-logik`

Hinweis

Dieses Skript ist als Hilfestellung für Studenten gedacht. Trotz großer Mühe beim Erstellen kann keine Garantie für Fehlerfreiheit übernommen werden. Es wird nochmals

darauf hingewiesen, dass der prüfungsrelevante Stoff durch die Vorlesung bestimmt wird und nicht mit dem Skriptinhalt vollständig übereinstimmen muss.

Das vorliegende Skript umfasst Teil 1 und Teil 2. Die Lehrunterlagen für den Teil 3 der Lehrveranstaltung werden gesondert veröffentlicht.

Einführung

Die Lehrveranstaltung „Theoretische Informatik und Logik“ gibt eine Einführung in drei wichtige Bereiche der Theoretischen Informatik:

Berechenbarkeit (Teil 1)

Welche Funktionen *Eingabe* $\rightarrow$ *Ausgabe* sind *überhaupt* berechenbar?

- *Formalisierung* des intuitiven Begriffs der Berechenbarkeit.
- Existenz *nicht-berechenbarer Funktionen.*
- *Entscheidbarkeit* und Existenz *unentscheidbarer Probleme.*

Komplexitätstheorie (Teil 2)

Wie schwer ist ein Berechnungs*problem „an sich“*?
Kein Verfahren kann besser sein als ...

- *Formalisierung* des Unterschieds zwischen *effizient* und *nicht-effizient* lösbaren Problemen.
- Die *„P=NP?“* Frage.
- *NP-vollständige Probleme*: kein effizientes Berechnungsverfahren bekannt.

Prädikatenlogik erster Stufe (Teil 3)

Logik: Formalismus, der es erlaubt, das *Wissen eines Anwendungsbereichs* (Mathematik, Medizin, Philosophie, ...) in einer Sprache mit *wohldefinierter Semantik* darzustellen.

- Semantik definiert eindeutig, welche *Konsequenzen* eine Aussage hat.
- *Kalküle*, die es erlauben, alle *Konsequenzen* einer Aussage *effektiv herzuleiten.*

Prädikatenlogik: Statt atomarer Aussagen verwende man *Prädikate* (z.B. $x < y$, $x + y = z$, $P(x_1, \ldots, x_n)$, $f(x) = y$), die in einer relationalen Struktur interpretiert werden.

Erste Stufe: *Quantifizierung* nur über Objekte (x, y, z) und *nicht* über *Prädikate* oder *Funktionen* (P, f).

Algorithmisierung

Algorithmische Lösungen für Probleme werden seit langem verwendet, u.a. von Euklid, Eratosthenes, Al-Khowarizim, Adam Riese, ...

Ein Algorithmus vereinfacht die Lösung des Problems von einer *intellektuellen Aufgabe* zu einem *rein mechanischen Berechnungsproblem.* Bei korrekter Anwendung steht dann die *Korrektheit der Lösung* außer Frage. Zunächst waren diese meist für *numerische Berechnungen* und zur Verwendung *durch Menschen* gedacht. Später folgte eine *Automatisierung* durch Rechenmaschinen, z.B. durch Leibniz.

> *Gottfried Wilhelm Leibniz*
> (geb. 1. Juli 1646 in Leipzig, gest. 14. November 1716 in Hannover)
> deutscher Philosoph und Wissenschaftler, Mathematiker, Diplomat, Physiker, Historiker, Politiker, Bibliothekar und Doktor des weltlichen und des Kirchenrechts
>
> „Es ist unwürdig, die Zeit von hervorragenden Leuten mit knechtischen Rechenarbeiten zu verschwenden, weil bei Einsatz einer Maschine auch der Einfältigste die Ergebnisse sicher hinschreiben kann."

Wie weit kann man die *Algorithmisierung / Automatisierung* treiben?

In der Hoffnung auf *universelle Algorithmisierbarkeit* suchen wir nach einer *universellen mathematischen Sprache* (lingua characteristica universalis, **Logik**), in der alles menschliche Wissen formalisiert werden kann, sowie *Rechenregeln* (calculus ratiocinator, **Kalkül**), die es erlauben, daraus automatisch alle *logischen Konsequenzen* zu berechnen:

> „Sollte es dann zwischen zwei Philosophen einen Meinungsstreit geben, so müssten sie darüber genauso wenig diskutieren, wie zwei Buchhalter über ein Rechenergebnis. Es würde genügen, Papier und Bleistift zu nehmen und zu sagen: *Lasst uns rechnen!* (lat. Calculemus!)"

Hilbert's Programm

In den Jahren 1922 bis 1930 formulierte *David Hilbert* sein Programm zur Realisierung des Leibniz'schen Traums für die Mathematik. Einerseits zielte er damit auf eine *Formalisierung der gesamten Mathematik* in einem **vollständigen Kalkül** ab, in dem *genau die wahren mathematischen Aussagen herleitbar* sind, und andererseits auf eine *Algorithmisierung der Mathematik*, d.h. auf ein mechanisches Verfahren, das es erlaubt zu entscheiden, ob eine mathematische Aussage wahr oder falsch ist.

Sein Vorhaben stieß auf reges Interesse, sodass sich alsbald dessen Unmöglichkeit herausstellte: 1930-1933 zeigte *Kurt Gödel* **Unvollständigkeit**, und wenig später in den Jahren 1936/1937 bewiesen *Alonzo Church* und *Alan Turing* die **Unentscheidbarkeit**.

David Hilbert
(geb. 23. Januar 1862 in Königsberg, gest. 14. Februar 1943 in Göttingen)
Mathematikprofessor in Königsberg und Göttingen
Einer der bedeutendsten Mathematiker der Neuzeit
Hilberts Probleme: 1900 stellt Hilbert die seiner Meinung nach 23 wichtigsten Probleme der Mathematik auf dem Mathematikerkongress in Paris vor

Kurt Gödel
(geb. 28. April 1906 in Brünn, gest. 14. Januar 1978 in Princeton)
Österreichisch-Amerikanischer Mathematiker
Einer der bedeutendsten Logiker des 20. Jahrhunderts
Dozent in Wien und Princeton
Unvollständigkeitssätze, Intuitionistische Logik, Kontinuumshypothese, Neumann-Bernays-Gödel-Mengenlehre, Ontologischer Gottesbeweis, Relativitätstheorie

Alonzo Church
(geb. 14. Juni 1903 in Washington, gest. 11. August 1995 in Hudson)
Amerikanischer Mathematiker und Logiker
Professor in Princeton 1929-1967 und an der University of California (LA) 1967-1990
λ-Kalkül, Church'sche These, Unentscheidbarkeit des Hilbert'schen Entscheidungsproblems
Doktorvater von Kleene, Rabin, Scott, Turing, ...

Alan Turing
(geb. 23. Juni 1912 in London, gest. 7. Juni 1954 in Wilmslow)
Englischer Mathematiker, Logiker und Informatiker
Turingmaschinen, Unentscheidbarkeit, Mitentwicklung des Manchester-Computer, Turing-Test
Entschlüsselung des Enigma-Codes (Bletchley Park)
Selbstmord wegen Verfolgung als Homosexueller

I. Berechenbarkeit

Einführung

Aus der Sicht der Theorie der *formalen Sprachen* geht es in diesem Teil darum, Automatenmodelle für die Typ-0- und die Typ-1-Sprachen zu finden. Allgemeiner geht es aber darum, die intuitiven Begriffe „*berechenbare* Funktion“ und „*entscheidbares* Problem“ zu formalisieren.

Um für eine (eventuell partielle) Funktion

$$\mathrm{f} : \mathbb{N}^k \to \mathbb{N} \quad (\text{bzw. } (\Sigma^*)^k \to \Sigma^*)$$

intuitiv klarzumachen, dass sie berechenbar ist, genügt es, ein Berechnungsverfahren (einen *Algorithmus*) für die Funktion anzugeben (z.B. in Form eines Modula-, Pascal- oder C-*Programmes* oder einer *abstrakten Beschreibung* der Vorgehensweise bei der Berechnung). *Eingaben* und *Ausgaben* für Berechnungsverfahren sind *endlich* und können daher als Tupel von *Wörtern* über einem geeigneten Alphabet dargestellt werden. *Wörter* kann man durch *natürliche Zahlen kodieren*, indem man eine (berechenbare) *Bijektion zwischen* Σ^* *und* $\mathbb{N}$ verwendet.

Entsprechend hatten wir in der Lehrveranstaltung „Formale Systeme“ die Entscheidbarkeit von Problemen (Wortproblem, Leerheitsproblem, Äquivalenzproblem, ...) dadurch begründet, dass wir *intuitiv* beschrieben haben, wie man die Probleme mit Hilfe eines Rechenverfahrens (d.h. *effektiv*) entscheiden kann. Aus dieser Beschreibung hätte man jeweils ein Modula-, Pascal-, etc. -Programm zur Entscheidung des Problems gewinnen können.

Beim Nachweis der Nichtentscheidbarkeit bzw. Nichtberechenbarkeit ist eine solche intuitive Vorgehensweise nicht mehr möglich, da man hier formal nachweisen muss, dass es *kein Berechnungsverfahren geben kann*. Damit ein solcher Beweis durchführbar ist, benötigt man eine formale Definition dessen, was man unter einem Berechnungsverfahren versteht. Man will also ein *Berechnungsmodell*, das

1) **einfach** ist, damit formale Beweise erleichtert werden (z.B. nicht Programmiersprache ADA),
2) **berechnungsuniversell** ist, d.h. alle intuitiv berechenbaren Funktionen damit berechnet werden können (z.B. nicht nur endliche Automaten).

Wir werden in diesem Teil ein derartiges Modell betrachten:

- Turingmaschinen

Es gibt noch eine Vielzahl anderer Modelle:

- μ-rekursive Funktionen
- WHILE-Programme
- Minskymaschinen
- GOTO-Programme
- URMs (unbeschränkte Registermaschinen)
- Pascal-Programme
- ...

Es hat sich herausgestellt, dass all diese Modelle *äquivalent* sind, d.h. die gleiche Klasse von Funktionen berechnen. Außerdem ist es bisher nicht gelungen, ein formales Berechnungsmodell zu finden, so dass

- die dadurch berechneten Funktionen noch intuitiv berechenbar erscheinen,
- dadurch Funktionen berechnet werden können, die nicht in den oben genannten Modellen ebenfalls berechenbar sind.

Aus diesen beiden Gründen geht man davon aus, dass die genannten Modelle genau den intuitiven Berechenbarkeitsbegriff formalisieren. Diese Überzeugung nennt man die:

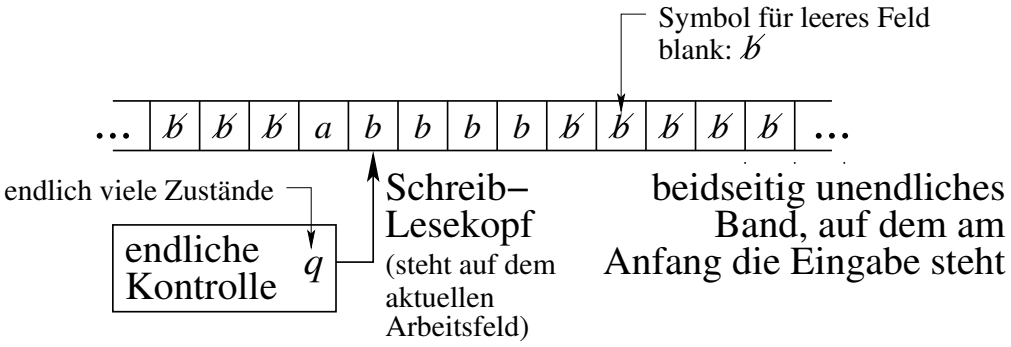
Churchsche These:
Die (intuitiv) berechenbaren Funktionen sind genau die mit Turingmaschinen (und damit mit GOTO-, WHILE-, Pascal-Programmen, URMs, ...) berechenbaren Funktionen.

Man spricht hier von einer *These* und nicht von einem *Satz*, da es nicht möglich ist, diese Aussage formal zu beweisen. Dies liegt daran, dass der intuitive Berechenbarkeitsbegriff ja nicht formal definierbar ist. Es gibt aber sehr viele Indizien, die für die Richtigkeit der These sprechen (Vielzahl äquivalenter Berechnungsmodelle).

Im Folgenden betrachten wir:

- Turingmaschinen
- Berechenbarkeit, Entscheidbarkeit, Aufzählbarkeit und Zusammenhänge
- Primitiv rekursive Funktionen und LOOP-Programme
- μ-rekursive Funktionen und WHILE-Programme
- Universelle Turingmaschinen und unentscheidbare Probleme
- Weitere unentscheidbare Probleme

1. Turingmaschinen



Zu jedem Zeitpunkt sind nur *endlich viele* Symbole auf dem Band verschieden von $\not b$.

Definition 1.1 (Turingmaschine)

Eine *Turingmaschine über dem Eingabealphabet* Σ hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, wobei

- Q endliche *Zustandsmenge* ist,
- Σ das *Eingabealphabet* ist,
- Γ das *Arbeitsalphabet* ist mit $\Sigma \subseteq \Gamma,\ \not b \in \Gamma \setminus \Sigma$,
- $q_0 \in Q$ der *Anfangszustand* ist,
- $F \subseteq Q$ die *Endzustandsmenge* ist und
- $\Delta \subseteq Q \times \Gamma \times \Gamma \times \{r, l, n\} \times Q$ die *Übergangsrelation* ist.

Dabei bedeutet $(q, a, a', \begin{matrix} r \\ l \\ n \end{matrix}, q')$:

- Im Zustand q
- mit a auf dem gerade gelesenen Feld (Arbeitsfeld)

kann die Turingmaschine $\mathcal{A}$

- das Symbol a durch a' ersetzen,
- in den Zustand q' gehen und
- den Schreib-Lesekopf entweder um ein Feld nach rechts (r), links (l) oder nicht (n) bewegen.

Die Maschine $\mathcal{A}$ heißt *deterministisch*, falls es für jedes Tupel $(q, a) \in Q \times \Gamma$ höchstens ein Tupel der Form $(q, a, \ldots, \ldots) \in \Delta$ gibt.

NTM steht im folgenden für (möglicherweise nichtdeterministische) Turingmaschinen und *DTM* für deterministische.

Einen Berechnungszustand (*Konfiguration*) einer Turingmaschine kann man beschreiben durch ein Wort $\alpha q \beta$ mit $\alpha, \beta \in \Gamma^+,\ q \in Q$:

- q ist der momentane Zustand
- α ist die Beschriftung des Bandes links vom Arbeitsfeld
- β ist die Beschriftung des Bandes beginnend beim Arbeitsfeld nach rechts

Dabei werden (um endliche Wörter $\alpha,\ \beta$ zu erhalten) unendlich viele Blanks weggelassen, d.h. α undβ umfassen mindestens den Bandabschnitt, auf dem Symbole $\neq \not b$ stehen.

Beispiel:
Der Zustand der Maschine zu Beginn des Abschnitts wird durch die Konfiguration *aqbbbb*, aber auch durch $\not b\not baqbbbb\not b$ beschrieben.

Die Übergangsrelation Δ ermöglicht die folgenden *Konfigurationsübergänge*:

Es seien $\alpha, \beta \in \Gamma^+,\ \beta' \in \Gamma^*,\ a, b, a' \in \Gamma,\ q, q' \in Q$.

- $\left.\begin{array}{rcl} \alpha qa\beta & \vdash_{\mathcal{A}} & \alpha a'q'\beta \\ \alpha qa & \vdash_{\mathcal{A}} & \alpha a'q'\not b \end{array}\right\}$ falls $(q, a, a', r, q') \in \Delta$
- $\left.\begin{array}{rcl} \alpha bqa\beta & \vdash_{\mathcal{A}} & \alpha q'ba'\beta \\ bqa\beta & \vdash_{\mathcal{A}} & \not b\, q'ba'\beta \end{array}\right\}$ falls $(q, a, a', l, q') \in \Delta$
- $\alpha qa\beta \ \vdash_{\mathcal{A}}\ \alpha q'a'\beta$ falls $(q, a, a', n, q') \in \Delta$

Weitere Bezeichnungen:

- Gilt $k \vdash_{\mathcal{A}} k'$, so heißt k' *Folgekonfiguration* von k.
- Die Konfiguration $\alpha q\beta$ heißt *akzeptierend*, falls $q \in F$.
- Die Konfiguration $\alpha q\beta$ heißt *Stoppkonfiguration*, falls sie keine Folgekonfiguration hat.
- Die von $\mathcal{A}$ *akzeptierte Sprache* ist
  $L(\mathcal{A}) = \{w \in \Sigma^* \mid \not b\, q_0 w \not b \vdash^*_{\mathcal{A}} k,$ wobei k *akzeptierende Stoppkonfiguration* ist$\}$.

Definition 1.2 (Turing-akzeptierbar, Turing-berechenbar)

1) Die Sprache $L \subseteq \Sigma^*$ heißt *Turing-akzeptierbar*, falls es eine NTM $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.
2) Die (partielle) Funktion $\mathrm{f} : (\Sigma^*)^n \to \Sigma^*$ heißt *Turing-berechenbar*, falls es eine DTM $\mathcal{A}$ gibt mit
   - $\forall x_1, \ldots, x_n \in \Sigma^* : \not b\, q_0 x_1 \not b \ldots \not b\, x_n \not b \vdash^*_{\mathcal{A}} k$ mit k Stoppkonfiguration gdw. $(x_1, \ldots, x_n) \in \mathrm{dom(f)}$ (Definitionsbereich von f).
   - Im Fall $(x_1, \ldots, x_n) \in \mathrm{dom(f)}$ muss die Beschriftung des Bandes in der Stoppkonfiguration k rechts vom Schreib-Lesekopf bis zum ersten Symbol $\notin \Sigma$ der Wert $y = \mathrm{f}(x_1, \ldots, x_n)$ der Funktion sein, d.h. k muss die Form $uqyv$ haben mit
     - $v \in (\Gamma \setminus \Sigma) \cdot \Gamma^* \cup \{\varepsilon\}$
     - $y = \mathrm{f}(x_1, \ldots, x_n)$

Beachte:

1) *Undefiniertheit* des Funktionswertes von f entspricht der Tatsache, dass die Maschine bei dieser Eingabe nicht terminiert.
2) Bei berechenbaren Funktionen betrachten wir nur *deterministische* Maschinen, da sonst der Funktionswert nicht eindeutig sein müsste.
3) Bei $|\Sigma| = 1$ kann man Funktionen von $(\Sigma^*)^n \to \Sigma^*$ als Funktionen von $\mathbb{N}^k \to \mathbb{N}$ auffassen (a^k entspricht k).

Beispiel 1.3

Die Funktion

$$\mathrm{f} : \mathbb{N} \to \mathbb{N}, \quad n \mapsto 2n$$

ist Turing-berechenbar. Wie kann eine Turingmaschine die Anzahl der a's auf dem Band verdoppeln?

Idee:

- Ersetze das erste a durch b,
- laufe nach rechts bis zum ersten blank, ersetze dieses durch c,
- laufe zurück bis zum zweiten a (unmittelbar rechts vom b), ersetze dieses durch b,
- laufe nach rechts bis zum ersten blank etc.
- Sind alle a's aufgebraucht, so ersetze noch die b's und c's wieder durch a's.

Dies wird durch die folgende *Übergangstafel* realisiert:

$(q_0,$	$\not b,$	$\not b,$	$n,$	$\text{stop}),$	$2 \cdot 0 = 0$
$(q_0,$	$a,$	$b,$	$r,$	$q_1),$	ersetze a durch b $\quad(\star)$
$(q_1,$	$a,$	$a,$	$r,$	$q_1),$	laufe nach rechts über a's
$(q_1,$	$c,$	$c,$	$r,$	$q_1),$	und bereits geschriebene c's
$(q_1,$	$\not b,$	$c,$	$n,$	$q_2),$	schreibe weiteres c
$(q_2,$	$c,$	$c,$	$l,$	$q_2),$	laufe zurück über c's und
$(q_2,$	$a,$	$a,$	$l,$	$q_2),$	a's
$(q_2,$	$b,$	$b,$	$r,$	$q_0),$	bei erstem b eins nach rechts und weiter wie $(\star)$ oder
$(q_0,$	$c,$	$c,$	$r,$	$q_3),$	alle a's bereits ersetzt
$(q_3,$	$c,$	$c,$	$r,$	$q_3),$	laufe nach rechts bis Ende der c's
$(q_3,$	$\not b,$	$\not b,$	$l,$	$q_4),$	letztes c erreicht
$(q_4,$	$c,$	$a,$	$l,$	$q_4),$	ersetze c's und b's
$(q_4,$	$b,$	$a,$	$l,$	$q_4),$	durch a's
$(q_4,$	$\not b,$	$\not b,$	$r,$	$\text{stop})$	bleibe am Anfang der erzeugten $2n$ a's stehen

Man sieht hier, dass das Programmieren von Turingmaschinen sehr umständlich ist. Wie bereits erwähnt, betrachtet man solche einfachen (und unpraktischen) Modelle, um das Führen von Beweisen zu erleichtern. Wir werden im folgenden häufig nur die Arbeitsweise einer Turingmaschine beschreiben, ohne die Übergangstafel voll anzugeben.

Beispiel:
Die Sprache $L = \{a^n b^n c^n \mid n \geq 0\}$ ist Turing-akzeptierbar. Die Turingmaschine, welche L akzeptiert, geht wie folgt vor:

- Sie ersetzt das erste a durch a', das erste b durch b' und das erste c durch c';
- läuft zurück zum zweiten a, ersetzt es durch a', das zweite b durch b' und das zweite c durch c' etc.
- Dies wird solange gemacht, bis nach erzeugtem c' ein $\not b$ steht.
- Danach wird von rechts nach links geprüft, ob – in dieser Reihenfolge – nur noch ein c'-Block, dann ein b'-Block und dann ein a'-Block (abgeschlossen durch $\not b$) vorhanden ist.
- Die Maschine blockiert, falls dies nicht so ist. Die kann auch bereits vorher geschehen, wenn erwartetes $a,\ b$ oder c nicht gefunden wird.

Eine entsprechende Turingmaschine $\mathcal{A}$ kann z.B. wie folgt definiert werden:
$\mathcal{A} = (\{q_0, q_{akz}, finde\_b, finde\_c, zu\_Ende\_?, zu\_Ende\_!, zurück\}$,
$\{a, b, c\}$,
$\{a, a', b, b', c, c', \not b\}$,
$q_0, \Delta, \{q_{akz}\})$ mit $\Delta =$

$\{(q_0,$	$\not b,$	$\not b,$	$N,$	$q_{akz}),$
$(q_0,$	$a,$	$a',$	$R,$	$finde\_b),$
$(finde\_b,$	$a,$	$a,$	$R,$	$finde\_b),$
$(finde\_b,$	$b',$	$b',$	$R,$	$finde\_b),$
$(finde\_b,$	$b,$	$b',$	$R,$	$finde\_c),$
$(finde\_c,$	$b,$	$b,$	$R,$	$finde\_c),$
$(finde\_c,$	$c',$	$c',$	$R,$	$finde\_c),$
$(finde\_c,$	$c,$	$c',$	$R,$	$zu\_Ende\_?),$
$(zu\_Ende\_?,$	$c,$	$c,$	$L,$	$zurück),$
$(zurück,$	$c',$	$c',$	$L,$	$zurück),$
$(zurück,$	$b,$	$b,$	$L,$	$zurück),$
$(zurück,$	$b',$	$b',$	$L,$	$zurück),$
$(zurück,$	$a,$	$a,$	$L,$	$zurück),$
$(zurück,$	$a',$	$a',$	$R,$	$q_0),$
$(zu\_Ende\_?,$	$\not b,$	$\not b,$	L	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$c',$	$c',$	$L,$	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$b',$	$b',$	$L,$	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$a',$	$a',$	$L,$	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$\not b,$	$\not b,$	$N,$	$q_{akz})\}$

Varianten von Turingmaschinen:

In der Literatur werden verschiedene Versionen der Definition der Turingmaschine angegeben, die aber alle äquivalent zueinander sind, d.h. dieselben Sprachen akzeptieren und dieselben Funktionen berechnen. Hier zwei Beispiele:

- Turingmaschinen mit nach links begrenztem und nur nach rechts unendlichem Arbeitsband
- Turingmaschinen mit mehreren Bändern und Schreib-Leseköpfen

Wir betrachten das zweite Beispiel genauer und zeigen Äquivalenz zur in Definition 1.1 eingeführten 1-Band-TM.

Definition (k-Band-TM)

Eine *k-Band-NTM* hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ mit

- $Q, \Sigma, \Gamma, q_0, F$ wie in Definition 1.1 und
- $\Delta \subseteq Q \times \Gamma^k \times \Gamma^k \times \{r, l, n\}^k \times Q$.

Dabei bedeutet $(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$:

- Vom Zustand q aus
- mit $a_1, \ldots, a_k$ auf den Arbeitsfeldern der k Bänder

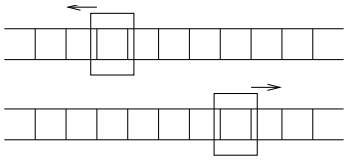
kann $\mathcal{A}$

- das Symbol a_i auf dem i-ten Band durch b_i ersetzen,
- in den Zustand q' gehen und
- die Schreib-Leseköpfe der Bänder entsprechend d_i bewegen.

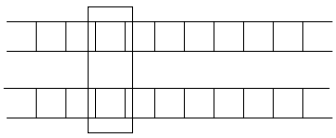
Das erste Band wird (o.B.d.A.) als Ein- und Ausgabeband verwendet.

Beachte:

Wichtig ist hier, dass sich die Köpfe der verschiedenen Bänder auch verschieden bewegen können:



Wären die Köpfe gekoppelt, so hätte man im Prinzip nicht mehrere Bänder, sondern ein Band mit mehreren Spuren:



k Spuren erhält man einfach, indem man eine normale NTM (nach Definition 1.1) verwendet, die als Bandalphabet Γ^k statt Γ hat.

Offenbar kann man jede 1-Band-NTM (nach Definition 1.1) durch eine k-Band-NTM ($k > 1$) simulieren, indem man nur das erste Band wirklich verwendet. Der nächste Satz zeigt, dass auch die Umkehrung gilt:

Satz
Wird die Sprache L durch eine k-Band-NTM akzeptiert, so auch durch eine 1-Band-NTM.

Beweis. Es sei $\mathcal{A}$ eine k-Band-NTM. Gesucht ist

- eine 1-Band-NTM $\mathcal{A}'$ und
- eine Kodierung $k \mapsto k'$ der Konfigurationen k von $\mathcal{A}$ durch Konfigurationen k' von $\mathcal{A}'$,

so dass gilt:

$$k_1 \vdash_{\mathcal{A}} k_2 \quad \underline{\text{gdw}}. \quad k_1' \vdash_{\mathcal{A}'}^* k_2'$$

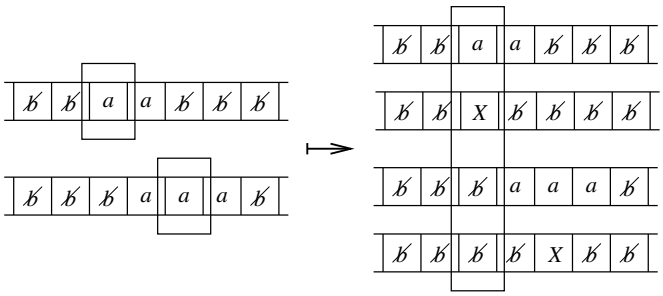
(Zur Simulation eines Schrittes von $\mathcal{A}$ sind mehr Schritte nötig).

Arbeitsalphabet von $\mathcal{A}'$: $\Gamma^{2k} \cup \Sigma \cup \{\not b\}$;

- $\Sigma \cup \{\not b\}$ wird für Eingabe benötigt;
- Γ^{2k} sorgt dafür, dass sich $\mathcal{A}'$ wie eine 1-Band-NTM mit $2k$ Spuren verhält.

Kodierung: Jeweils 2 Spuren kodieren ein Band von $\mathcal{A}$.

- Die erste Spur enthält die Bandbeschriftung.
- Die zweite Spur enthält eine Markierung X (und sonst Blanks), die zeigt, wo das Arbeitsfeld des Bandes ist, z.B.



Hierbei wird angenommen, dass das Arbeitsfeld von $\mathcal{A}'$ stets bei dem am weitesten links stehenden X liegt. *Initialisierung:* Zunächst wird die Anfangskonfiguration

$$\not b q_0 a_1 \ldots a_m \not b$$

von $\mathcal{A}'$ in die *Kodierung* der entsprechenden Anfangskonfiguration von $\mathcal{A}$ umgewandelt (durch entsprechende Übergänge):

ß	ß	a_1	a_2	...	a_m	ß	Spur 1 und 2 kodieren
ß	ß	X	ß	...	ß	ß	Band 1
ß	ß	ß	ß	...	ß	ß	Spur 3 und 4 kodieren
ß	ß	X	ß	...	ß	ß	Band 2
		$\vdots$					

Simulation der Übergänge: Betrachte $(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$.

- Von links nach rechts suche die mit X markierten Felder. Dabei merke man sich (in dem Zustand der TM) die Symbole, welche jeweils über dem X stehen. Außerdem zählt man (durch Zustand der TM) mit, wie viele X man schon gelesen hat, um festzustellen, wann das k-te erreicht ist. Man bestimmt so, ob das aktuelle Tupel tatsächlich $(a_1, \ldots, a_k)$ ist.
- Von rechts nach links gehend überdrucke man die a_i bei den X-Marken jeweils durch das entsprechende b_i und verschiebt die X-Marken gemäß d_i.
- Bleibe bei der am weitesten links stehenden X-Markierung, lasse den Kopf dort stehen und gehe in Zustand q'. □

Bemerkung.

1) War $\mathcal{A}$ deterministisch, so liefert obige Konstruktion auch eine deterministische 1-Band-Turingmaschine.
2) Diese Konstruktion kann auch verwendet werden, wenn man sich für die berechnete Funktion interessiert. Dazu muss man am Schluss (wenn $\mathcal{A}$ in Stoppkonfiguration ist) in der Maschine $\mathcal{A}'$ noch die Ausgabe geeignet aufbereiten.

Bei der Definition von Turing-berechenbar haben wir uns von vornherein auf *deterministische* Turingmaschinen beschränkt. Der folgende Satz zeigt, dass man dies auch bei Turing-akzeptierbaren Sprachen machen kann, ohne an Ausdrucksstärke zu verlieren.

Satz

Zu jeder NTM gibt es eine DTM, die dieselbe Sprache akzeptiert.

Beweis. Wegen Satz 1 und Bemerkung 1 genügt es, eine deterministische 3-Band-Turingmaschine zu konstruieren. Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ eine NTM.

Die Maschine $\mathcal{A}'$ soll für wachsendes n auf dem dritten Band jeweils alle Konfigurationsfolgen

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} k_2 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_n$$

beginnend mit der Startkonfiguration $k_0 = ß\, q_0 w ß$ erzeugen.

Die Kontrolle, dass tatsächlich alle solchen Folgen erzeugt werden, wird auf dem zweiten Band vorgenommen. Das erste Band speichert das Eingabewort w (damit man stets weiß, was k_0 sein muss).

Genauer: Es sei

$$r = \text{ maximale Anzahl von Transitionen in } \Delta \text{ pro festem Paar } (q, a) \in Q \times \Gamma$$

(entspricht dem maximalen Verzweigungsgrad der nichtdeterministischen Berechnung).

Eine Indexfolge $i_1, \ldots, i_n$ mit $i_j \in \{1, \ldots, r\}$ bestimmt dann von k_0 aus für n Schritte die Auswahl der jeweiligen Transition, und somit von k_0 aus eine feste Konfigurationsfolge

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_n$$

Zählt man daher alle endlichen Wörter über $\{1, \ldots, r\}$ auf und zu jedem Wort $i_1 \ldots i_n$ die zugehörige Konfigurationsfolge, so erhält man eine Aufzählung aller endlichen Konfigurationsfolgen.

$\mathcal{A}'$ realisiert dies auf den drei Bändern wie folgt:

- Auf Band 1 bleibt die Eingabe gespeichert.
- Auf dem zweiten Band werden sukzessive alle Wörter $i_1 \ldots i_n \in \{1, \ldots, r\}^*$ erzeugt.
- Für jedes dieser Wörter wird auf dem dritten Band die zugehörige Konfigurationsfolge realisiert. Erreicht man hierbei eine akzeptierende Konfiguration von $\mathcal{A}$, so geht auch $\mathcal{A}'$ in eine akzeptierende Konfiguration. □

2. Berechenbarkeit, Entscheidbarkeit, Aufzählbarkeit

Wir werden hier – gemäß der These von Church – die Begriffe *Turing-berechenbar* und *(intuitiv) berechenbar* als *synonym* verwenden.

Wegen der anfangs erwähnten Äquivalenz von Berechnungsmodellen treffen alle Definitionen und Resultate auch auf die anderen Modelle zu.

Wir werden meist intuitiv die Existenz eines Berechnungsverfahrens (einer DTM) begründen. Diese intuitiven Argumente können aber leicht (wenn auch im Detail technisch und zeitaufwendig) in TM-Konstruktionen übersetzt werden.

Definition 2.1 (Berechenbarkeit)
Wir betrachten partielle oder totale Funktionen

$$\mathrm{f} : (\Sigma^*)^n \to \Sigma^*.$$

- f heißt *berechenbar (partiell rekursiv)*, falls sie Turing-berechenbar ist.
- Ist f total (d.h. $\mathrm{dom}(\mathrm{f}) = (\Sigma^*)^n$), so heißt f *rekursiv*.

Bei partiellen Funktionen entspricht Undefiniertheit des Funktionswertes der Tatsache, dass das Berechnungsverfahren (die DTM) bei dieser Eingabe nicht terminiert.

Definition 2.2 (entscheidbar, partiell entscheidbar, rekursiv aufzählbar)
Wir betrachten n-stellige Relationen $R \subseteq (\Sigma^*)^n$.

1) R heißt *entscheidbar (rekursiv)*, falls ihre *charakteristische Funktion*

$$\chi_R : (\Sigma^*)^n \to \Sigma^* \text{ mit } (x_1, \ldots, x_n) \mapsto \begin{cases} a & \text{falls } (x_1, \ldots, x_n) \in R \\ \varepsilon & \text{sonst} \end{cases}$$

   berechenbar ist. Dabei ist a ein Element von Σ (beliebig aber fest gewählt).

2) R heißt *partiell entscheidbar (partiell rekursiv)*, falls R Definitionsbereich einer berechenbaren Funktion f ist.

   D.h. es gibt eine DTM, die bei Eingabe $(x_1, \ldots, x_n) \in (\Sigma^*)^n$
   - terminiert, falls $(x_1, \ldots, x_n) \in R$ ist und
   - nicht terminiert sonst.

3) R heißt *rekursiv aufzählbar*, falls R von einer Aufzähl-Turingmaschine aufgezählt wird. Diese ist wie folgt definiert:

   Eine *Aufzähl-Turingmaschine* $\mathcal{A}$ ist eine DTM, die einen speziellen Ausgabezustand q_{Ausgabe} hat.

   Eine *Ausgabekonfiguration* für eine n-stellige Relation ist von der Form

$$u q_{\text{Ausgabe}}\, x_1 \not{b}\, x_2 \not{b} \ldots x_n \not{b} v \text{ mit } u, v \in \Gamma^* \text{ und } x_1, \ldots, x_n \in \Sigma^*.$$

Diese Konfiguration hat $(x_1, \ldots, x_n)$ als *Ausgabe*.

Die durch $\mathcal{A}$ *aufgezählte Relation* ist

$$R = \{(x_1, \ldots, x_n) \in (\Sigma^*)^n \mid (x_1, \ldots, x_n) \text{ ist Ausgabe einer Ausgabekonfiguration, die von } \mathcal{A} \text{ ausgehend von } \not b q_0 \not b \text{ erreicht wird}\}.$$

Bemerkung 2.3.

1) Probleme wie z.B. das *Wortproblem* für kontextfreie Sprachen können als *Relationen* aufgefasst werden:

   Es sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w \in \Sigma^*$.

   Die Grammatik G kann als Wort $\text{code}(G) \in \Gamma^*$ über einem erweiterten Alphabet Γ aufgefasst werden. Das Wortproblem für G entspricht der Relation

   $$R = \{(\text{code}(G), w) \mid w \in L(G)\} \subseteq (\Gamma^*)^2.$$

2) Für $n = 1$ stimmen die Begriffe „partiell entscheidbar“ und „Turing-akzeptierbar“ überein, denn:

   Bei Turing-akzeptierbar war neben dem Anhalten der (o.B.d.A. deterministischen) TM zusätzlich gefordert, dass man einen akzeptierenden Zustand erreicht hat. Man kann aber einfach von allen nichtakzeptierenden Stoppkonfigurationen aus in eine Endlosschleife gehen.

3) Im Fall $n = 1$ sind folgende Charakterisierungen äquivalent:
   (a) R ist rekursiv aufzählbar
   (b) $R = \emptyset$ oder R ist *Wertebereich* einer rekursiven Funktion.
   (c) R ist *Wertebereich* einer partiell rekursiven Funktion.

   Beweis.

   „$a \Rightarrow b$“ Es sei R rekursiv aufzählbar und $R \neq \emptyset$. Weiterhin sei $\mathcal{A}$ eine Aufzähl-TM für R.

   Wähle $w_0 \in R$ beliebig.

   Wir definieren die Funktion $\text{f} : \Sigma^* \times \Sigma^* \to \Sigma^*$ wie folgt:

   $$\text{f(u, v)} := \begin{cases} u & \text{falls } \mathcal{A} \text{ nach} \leq |v| \text{ Schritte } u \text{ als Ausgabe erzeugt} \\ w_0 & \text{sonst} \end{cases}$$

   Es ist leicht zu sehen, dass der Wertebereich ran(f) von f die Menge R ist.

   Außerdem kann man aus $\mathcal{A}$ leicht eine TM erhalten, die f berechnet.

   „$b \Rightarrow c$“ Für $R \neq \emptyset$ ist dies trivial da jede rekursive Funktion auch partiell rekursive ist.

   Für $R = \emptyset$ betrachten wir die überall undefinierte Funktion. Diese ist offenbar partiell berechenbar und hat als Wertebereich die leere Menge.

„$c \Rightarrow a$“ Es sei $R = \text{ran(f)}$ und $\mathcal{A}$ eine DTM, die f berechnet.

Die Aufzähl-TM $\mathcal{B}$ für R arbeitet wie folgt:

Es sei $e_1, e_2, \ldots$ eine effektive Aufzählung aller Eingaben für $\mathcal{A}$.
Dann lässt $\mathcal{B}$

1) die Berechnung von $\mathcal{A}$ bei Eingabe e_1 *einen Schritt* laufen
2) die Berechnung von $\mathcal{A}$ bei Eingabe e_1 *zwei Schritte* und bei Eingabe e_2 ebenfalls *2 Schritte* laufen
   ⋮
n) die Berechnung von $\mathcal{A}$ mit den Eingaben $e_1, e_2, \ldots, e_n$ jeweils n *Schritte* laufen
   ⋮

Wird dabei ein Funktionswert von f berechnet, so wird dieser ausgegeben. □

Satz 2.4

Es sei $R \subseteq (\Sigma^*)^n$ eine Relation.

1) R ist rekursiv aufzählbar gdw. R ist partiell entscheidbar.

2) Ist R entscheidbar, so auch partiell entscheidbar.

3) Ist R entscheidbar, so auch $\overline{R} = (\Sigma^*)^n \setminus R$.

4) R ist entscheidbar gdw. R und $\overline{R}$ partiell entscheidbar sind.

Beweis.

1) „⇒“: Es sei R rekursiv aufzählbar und $\mathcal{A}$ eine Aufzähl-DTM für R. Die Maschine $\mathcal{A}'$ arbeitet wie folgt:

   - Sie speichert die Eingabekonfiguration $\not b q_0 \not b x_1 \not b x_2 \ldots \not b x_n \not b$ ab (z.B. auf zusätzlichem Band).
   - Sie beginnt (auf anderem Band) mit der Aufzählung von R.
   - Bei jeder Ausgabekonfiguration überprüft sie, ob die entsprechende Ausgabe mit der gespeicherten Eingabe übereinstimmt. Wenn ja, so terminiert $\mathcal{A}'$. Sonst sucht sie die nächste Ausgabekonfiguration von $\mathcal{A}$.
   - Terminiert $\mathcal{A}$, ohne dass $(x_1, \ldots, x_n)$ ausgegeben wurde, so gehe in Endlosschleife.

   $\mathcal{A}'$ terminiert daher genau dann nicht, wenn $(x_1, \ldots, x_n)$ nicht in der Aufzählung vorkommt.

   „⇐“: Es sei $\mathcal{A}$ ein partielles Entscheidungsverfahren für R, d.h. $\mathcal{A}$ berechnet eine Funktion f mit $\text{dom(f)} = R$.

   Es sei $\vec{x}^{(1)}, \vec{x}^{(2)}, \vec{x}^{(3)}, \ldots$ eine Auflistung von $(\Sigma^*)^n$.
   Die Maschine $\mathcal{A}'$ arbeitet wie folgt:

1) Simuliere die Berechnung von $\mathcal{A}$ mit der Eingabe $\vec{x}^{(1)}$ *einen Schritt.*

2) Simuliere die Berechnung von $\mathcal{A}$ mit der Eingabe $\vec{x}^{(1)}$ *zwei Schritte* und mit der Eingabe $\vec{x}^{(2)}$ *zwei Schritte.*

   $\vdots$

n) Simuliere die Berechnung von $\mathcal{A}$ mit den Eingaben $\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}$ jeweils *n Schritte.*

   $\vdots$

Terminiert $\mathcal{A}$ für eine dieser Eingaben, so gebe diese Eingabe aus und mache weiter.

Beachte:

Man kann nicht $\mathcal{A}$ zunächst ganz mit Eingabe $\vec{x}^{(1)}$ laufen lassen (ohne Schrittbeschränkung), da $\mathcal{A}$ darauf nicht terminieren muss. Das gewählte Vorgehen nennt man *dove-tailing* (Ineinander-Verzahnen mehrerer Berechnungen).

2) Eine DTM $\mathcal{A}$, die χ_R berechnet, wird wie folgt modifiziert:

- Ist die Ausgabe ε (d.h. in der Stoppkonfiguration von $\mathcal{A}$ steht der Kopf auf $\not b$), so gehe in Endlosschleife.
- Sonst gehe in Stoppkonfiguration.

3) Eine DTM $\mathcal{A}$, die χ_R berechnet, wird wie folgt zu einer DTM für $\chi_{\overline{R}}$ modifiziert:

- Ist die Ausgabe von $\mathcal{A}$ das leere Wort ε, so erzeuge Ausgabe a.
- Sonst erzeuge Ausgabe ε.

4) „$\Rightarrow$“: Ergibt sich aus 2) und 3).

„$\Leftarrow$“: Sind R und $\overline{R}$ partiell entscheidbar, so mit 1) auch rekursiv aufzählbar.

Für Eingabe $\vec{x}$ lässt man die Aufzähl-DTMs $\mathcal{A}$ und $\mathcal{B}$ für R und $\overline{R}$ parallel laufen (d.h. jeweils abwechselnd ein Schritt von $\mathcal{A}$ auf einem Band gefolgt von einem Schritt von $\mathcal{B}$ auf dem anderen).

Die Eingabe $\vec{x}$ kommt in einer der beiden Aufzählungen vor:

$$\vec{x} \in (\Sigma^*)^n = R \cup \overline{R}$$

Kommt $\vec{x}$ bei $\mathcal{A}$ vor, so erzeuge Ausgabe a, sonst Ausgabe ε. □

Für $\mathrm{f} : (\Sigma^*)^n \to \Sigma^*$ definieren wir:

$$\mathrm{Graph(f)} := \{(x_1, \ldots, x_n, x_{n+1}) \mid (x_1, \ldots, x_n) \in \mathrm{dom(f)} \text{ und } \mathrm{f}(x_1, \ldots, x_n) = x_{n+1}\}$$

Satz 2.5

f *ist berechenbar gdw.* Graph(f) *ist rekursiv aufzählbar.*

Beweis.

„⇐“: Sei Graph(f) rekursiv aufzählbar. Um den Funktionswert $f(x_1, \ldots, x_n)$ zu berechnen, startet man das Aufzählungsverfahren und wartet, bis ein Tupel der Form $(x_1, \ldots, x_n, y)$ auftaucht.

- Wenn ja, so ist y der Funktionswert.
- Sonst ist $(x_1, \ldots, x_n) \notin \text{dom}(f)$, und die Maschine terminiert nicht, d.h. sie zählt unendlich auf oder geht in Endlosschleife, wenn die Aufzählung abgebrochen wird, ohne dass Tupel der Form $(x_1, \ldots, x_n, \cdot)$ gefunden wurde).

„⇒“: Mit Satz 2.4 genügt es zu zeigen, dass Graph(f) partiell entscheidbar ist.

Das partielle Entscheidungsverfahren für Graph(f) erhält man wie folgt:

Berechne bei Eingabe $(x_1, \ldots, x_n, x_{n+1})$ den Funktionswert von f an der Stelle $(x_1, \ldots, x_n)$.

1. Fall: $f(x_1, \ldots, x_n)$ ist definiert und $= x_{n+1}$.
Dann terminiere.

2. Fall: $f(x_1, \ldots, x_n)$ ist definiert und $\neq x_{n+1}$.
Gehe in Endlosschleife.

3. Fall: $f(x_1, \ldots, x_n)$ ist undefiniert.
Dann terminiert die Berechnung des Wertes bereits nicht. □

3. Primitiv rekursive Funktionen und Loop-Programme

In den nächsten beiden Abschnitten betrachten wir Berechnungsmodelle, die eng verwandt sind mit modernen imperativen und funktionalen Programmiersprachen. Die in diesem Abschnitt eingeführten Modelle sind allerdings *nicht* stark genug, um alle (Turing-)berechenbaren Funktionen zu erfassen. Wir werden im Abschnitt 4 zeigen, wie man diese Modelle erweitern muss, um berechnungsuniverselle Modelle zu erhalten.

Wir betrachten hier nur Funktionen von

$$\mathbb{N}^n \to \mathbb{N}.$$

Dies entspricht dem Spezialfall $|\Sigma| = 1$ bei Wortfunktionen, ist aber keine echte Einschränkung, da es berechenbare Kodierungsfunktionen gibt, d.h.

$$\pi : \Sigma^* \to \mathbb{N} \quad \text{bijektiv}$$

mit π und π^{-1} berechenbar.

Definition 3.1 (Grundfunktionen)
Die folgenden Funktionen sind *primitiv rekursive Grundfunktionen*:

1) $\mathrm{s} : \mathbb{N} \to \mathbb{N}$ mit $x \mapsto x + 1$ *(Nachfolgerfunktion)*
2) Für alle $n \geq 0$ und i, $1 \leq i \leq n$:
   - $\pi_i^{(n)} : \mathbb{N}^n \to \mathbb{N}$ mit $(x_1, \ldots, x_n) \mapsto x_i$ *(Projektion)*
   - $\mathrm{null}^{(n)} : \mathbb{N}^n \to \mathbb{N}$ mit $(x_1, \ldots, x_n) \mapsto 0$ *(Nullfunktion)*

Aus diesen einfachen Funktionen kann man mit Hilfe von Operatoren komplexere Funktionen aufbauen.

Definition 3.2 (Komposition)
Die Funktion $\mathrm{f} : \mathbb{N}^n \to \mathbb{N}$ entsteht aus

$$\mathrm{g} : \mathbb{N}^m \to \mathbb{N} \text{ und}$$
$$\mathrm{h}_1, \ldots, \mathrm{h}_m : \mathbb{N}^n \to \mathbb{N}$$

durch *Komposition*, falls für alle $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gilt:

$$\mathrm{f}(x_1, \ldots, x_n) = \mathrm{g}(\mathrm{h}_1(x_1, \ldots, x_n), \ldots, \mathrm{h}_m(x_1, \ldots, x_n)).$$

Wendet man Komposition auf echt partielle Funktionen an, so gilt:
$\mathrm{f}(x_1, \ldots, x_n)$ ist *undefiniert* gdw.

- eines der $\mathrm{h}_i(x_1, \ldots, x_n)$ ist *undefiniert* oder

- alle h_i-Werte sind *definiert*, aber g von diesen Werten ist *undefiniert*.

Beispiel
$g(x, y) = x$, $h_1(0) = 0$, $h_2(0)$ undefiniert.
Dann ist $g(h_1(0), h_2(0))$ undefiniert (entspricht call-by-value-Auswertung).

Definition 3.3 (primitive Rekursion)
Es sei $n \geq 0$. Die Funktion $f : \mathbb{N}^{n+1} \to \mathbb{N}$ entsteht aus

$$g : \mathbb{N}^n \to \mathbb{N} \text{ und}$$
$$h : \mathbb{N}^{n+2} \to \mathbb{N}$$

durch *primitive Rekursion*, falls gilt:

- $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$
- $f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, f(x_1, \ldots, x_n, y), y)$

Auch hier setzen sich wie bei der Komposition wieder undefinierte Werte fort.

Durch Induktion über die letzte Komponente zeigt man leicht, dass dieses Schema die Funktion f eindeutig definiert.

Beispiel 3.4 (Addition)
Die Addition natürlicher Zahlen kann durch primitive Rekursion wie folgt definiert werden:

$$\begin{aligned} \mathrm{add}(x, 0) &= x && = g(x) \\ \mathrm{add}(x, y + 1) &= \mathrm{add}(x, y) + 1 && = h(x, \mathrm{add}(x, y), y) \end{aligned}$$

Das heißt also: add entsteht durch primitive Rekursion aus den Funktionen

- $g : \mathbb{N} \to \mathbb{N}$ mit $x \mapsto x$,
  d.h. $g = \pi_1^{(1)}$ ist Grundfunktion,
- $h : \mathbb{N}^3 \to \mathbb{N}$ mit $(x, z, y) \mapsto z + 1$,
  d.h. $h(x, z, y) = s(\pi_2^{(3)}(x, z, y))$.
  Also entsteht h durch Komposition aus Grundfunktionen.

Definition 3.5 (Klasse der primitiv rekursiven Funktionen)
Die *Klasse der primitiv rekursiven Funktionen* besteht aus allen Funktionen, die man aus den *Grundfunktionen* durch endlich oftes Anwenden von

- *Komposition* und
- *primitiver Rekursion*

erhält.

Offenbar sind die Grundfunktionen *total* und die Operationen Komposition und primitive Rekursion erzeugen aus totalen Funktionen wieder totale.

Satz 3.6

Die Klasse der primitiv rekursiven Funktionen enthält nur totale Funktionen.

Trotzdem sind aber die Operationen primitive Rekursion und Komposition auch für partielle Funktionen definiert, und wir werden sie später auch auf partielle Funktionen anwenden.

Beispiel 3.4 zeigt, dass add zur Klasse der primitiv rekursiven Funktionen gehört. Wir betrachten nun weitere Beispiele für primitiv rekursive Funktionen.

Multiplikation erhält man durch primitive Rekursion aus der Addition:

$$\begin{aligned} \text{mult}(x,0) &= 0 && = \text{null}^{(1)}(x) \\ \text{mult}(x,y+1) &= \text{add}(x,\text{mult}(x,y)) && = \text{add}(\pi_1^{(3)},\pi_2^{(3)})(x,\text{mult}(x,y),y) \end{aligned}$$

Exponentiation erhält man durch primitive Rekursion aus der Multiplikation:

$$\begin{aligned} \exp(x,0) &= 1 && = \text{s}(\text{null}^{(1)}(x)) \\ \exp(x,y+1) &= \text{mult}(x,\exp(x,y)) \end{aligned}$$

Die Funktion

$$\text{min1} : \mathbb{N} \to \mathbb{N} \text{ mit } x \mapsto x \dot{-} 1 := \begin{cases} x-1 & x > 0 \\ 0 & x = 0 \end{cases}$$

ist ebenfalls primitiv rekursiv:

$$\begin{aligned} \text{min1}(0) &= 0 && = \text{null}^{(0)}(), \\ \text{min1}(y+1) &= y && = \pi_2^{(2)}(\text{min1}(y),y) \end{aligned}$$

Übung:

Zeige, dass

$$\text{min} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x,y) \mapsto x \dot{-} y := \begin{cases} x-y & x \geq y \\ 0 & \text{sonst} \end{cases}$$

primitiv rekursiv ist.

Beispiel 3.7

Aus den bisher betrachteten Funktionen erhält man damit die Funktion

$$\text{c} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x,y) \mapsto 2^x \cdot (2y+1) \dot{-} 1$$

durch Komposition, d.h. c ist primitiv rekursiv. Diese Funktion ist interessant, da sie eine Bijektion von $\mathbb{N}^2 \to \mathbb{N}$ ist, d.h. man kann mit ihr Tupel natürlicher Zahlen in eine natürliche Zahl kodieren.

Lemma 3.8

Die Funktion c *aus Beispiel 3.7 ist eine Bijektion.*

Beweis.

Surjektivität: Es sei $z \in \mathbb{N}$. Dann betrachten wir die größte Zweierpotenz 2^x, die $z+1$ teilt. Offenbar ist dann $\frac{z+1}{2^x}$ eine ungerade Zahl, d.h. es gibt ein y mit

$$\frac{z+1}{2^x} = 2y+1.$$

Damit ist $z = 2^x \cdot (2y+1)\dot{-}1 = \mathrm{c}(x,y)$.

Injektivität: Offenbar ist die größte Zweierpotenz, die $z+1$ teilt, eindeutig, d.h. x ist eindeutig durch z bestimmt. Damit ist aber auch y eindeutig bestimmt. □

Da c eine Bijektion ist, gibt es die Umkehrfunktionen c_0 und c_1 mit der Eigenschaft:

- $\mathrm{c}_0(\mathrm{c}(x,y)) = x$ und $\mathrm{c}_1(\mathrm{c}(x,y)) = y$
- $\mathrm{c}(\mathrm{c}_0(z), \mathrm{c}_1(z)) = z$

Diese ergeben sich im Prinzip aus dem Beweis von Lemma 3.8, d.h.

$$\mathrm{c}_0(z) = \max\{x \in \mathbb{N} \mid 2^x | (z+1)\}$$
$$\mathrm{c}_1(z) = ((\frac{z+1}{2^{\mathrm{c}_0(z)}})\dot{-}1)/2$$

Um zu zeigen, dass c_0, c_1 ebenfalls primitiv rekursiv sind, benötigen wir noch etwas Vorarbeit.

Lemma 3.9
Die Funktionen $\mathrm{sign} : \mathbb{N} \to \mathbb{N}$ *und* $\overline{\mathrm{sign}} : \mathbb{N} \to \mathbb{N}$ *mit*

$$\mathrm{sign}(x) = \begin{cases} 0 & \textit{falls } x = 0, \textit{ und} \\ 1 & \textit{falls } x > 0, \end{cases} \qquad \textit{sowie} \qquad \overline{\mathrm{sign}}(x) = \begin{cases} 1 & \textit{falls } x = 0, \textit{ und} \\ 0 & \textit{falls } x > 0 \end{cases}$$

sind primitiv rekursiv.

Beweis.

$$\mathrm{sign}(0) = 0 \qquad \mathrm{sign}(y+1) = 1$$
$$\overline{\mathrm{sign}}(0) = 1 \qquad \overline{\mathrm{sign}}(y+1) = 0$$

sind primitiv rekursive Definitionsschemata dafür. □

Definition 3.10 (Fallunterscheidung)
Es seien $\mathrm{g}_1, \mathrm{g}_2, \mathrm{h} : \mathbb{N}^n \to \mathbb{N}$ gegeben.

Die Funktion $\mathrm{f} : \mathbb{N}^n \to \mathbb{N}$ entsteht daraus durch *Fallunterscheidung*, falls für alle $\underline{x} \in \mathbb{N}^n$ gilt:

$$\mathrm{f}(\underline{x}) = \begin{cases} \mathrm{g}_1(\underline{x}) & \text{falls} \quad \mathrm{h}(\underline{x}) = 0 \\ \mathrm{g}_2(\underline{x}) & \text{falls} \quad \mathrm{h}(\underline{x}) > 0 \end{cases}$$

Lemma 3.11
Sind g_1, g_2, h *primitiv rekursiv, so auch* f.

Beweis.
$$f(\underline{x}) = g_1(\underline{x}) \cdot \overline{\text{sign}}(h(\underline{x})) + g_2(\underline{x}) \cdot \text{sign}(h(\underline{x})).$$ □

Definition 3.12 (beschränkte Minimalisierung)
Es sei $n \geq 0$. Die Funktion $f : \mathbb{N}^{n+1} \to \mathbb{N}$ entsteht aus $g : \mathbb{N}^{n+1} \to \mathbb{N}$ durch *beschränkte Minimalisierung*, falls gilt:

$$f(\underline{x}, y) = \begin{cases} j & \text{falls} \quad j = \min\{i \leq y \mid g(\underline{x}, i) = 0\} \text{ existiert} \\ y+1 & \text{sonst} \end{cases}$$

Wir schreiben dann $f = \overline{\mu}g$.

Lemma 3.13
Ist g *primitiv rekursiv, so auch* $\overline{\mu}g$.

Beweis. Wir definieren $\overline{\mu}g$ mittels primitiver Rekursion wie folgt:

1) $\overline{\mu}g(\underline{x}, 0) = \begin{cases} 0 & \text{falls} \quad g(\underline{x}, 0) = 0 \\ 1 & \text{sonst} \end{cases}$

   D.h. $\overline{\mu}g(\underline{x}, 0) = \text{sign}(g(\underline{x}, 0))$

2) $\overline{\mu}g(\underline{x}, y+1) = \begin{cases} \overline{\mu}g(\underline{x}, y) & \text{falls} \quad \overline{\mu}g(\underline{x}, y) \leq y \text{ oder } g(\underline{x}, y+1) = 0 \\ y+2 & \text{sonst} \end{cases}$

Es handelt sich hier also um eine Fallunterscheidung. Es bleibt zu zeigen, dass die Funktion
$$h(\underline{x}, z, y) = \begin{cases} 0 & \text{falls} \quad z \leq y \text{ oder } g(\underline{x}, y+1) = 0 \\ 1 & \text{sonst} \end{cases}$$
primitiv rekursiv ist.
Offenbar ist aber
$$h(\underline{x}, z, y) = \text{sign}((z \dot{-} y) \cdot g(\underline{x}, y+1))$$
und damit primitiv rekursiv. □

Wir kommen nun zurück zu den Umkehrfunktionen der Bijektion aus Beispiel 3.7. Betrachten wir zunächst die Teilbarkeitsrelation in der Definition von c_0.

Lemma 3.14
Die Funktion
$$\text{teilt} : \mathbb{N}^2 \to \mathbb{N} \textit{ mit } (x, y) \mapsto \begin{cases} 0 & x|y \\ 1 & \textit{sonst} \end{cases}$$
ist primitiv rekursiv.

Beweis. Die Funktion

$$\mathrm{h}(x,y,z) = \begin{cases} 0 & x \cdot z = y \\ 1 & \text{sonst} \end{cases}$$

ist primitiv rekursiv, da

$$\mathrm{h}(x,y,z) = \mathrm{sign}((x \cdot z \dot{-} y) + (y \dot{-} x \cdot z)).$$

Damit ist auch $\overline{\mu}\mathrm{h}$ primitiv rekursiv, und es gilt:

$$\overline{\mu}\mathrm{h}(x,y,y) = \begin{cases} j & \text{falls} \quad j = \min\{i \leq y \mid x \cdot i = y\} \\ y+1 & \text{sonst} \end{cases}$$

Damit ist aber $\mathrm{teilt}(x,y) = \overline{\mu}\mathrm{h}(x,y,y) \dot{-} y$. □

Lemma 3.15

Die Umkehrfunktion c_0 *der Funktion* c *aus Beispiel 3.7 ist primitiv rekursiv.*

Beweis. $\mathrm{c}_0(z) = \max\{x \in \mathbb{N} \mid 2^x | (z+1)\}$.

Es genügt dafür, nach dem kleinsten $i \leq z+1$ zu suchen mit

$$2^{(z+1)\dot{-}i} \mid (z+1)$$

d.h. nach dem kleinsten $i \leq z+1$ mit

$$\mathrm{teilt}(2^{(z+1)\dot{-}i}, z+1) = 0.$$

Dies gelingt durch beschränkte Minimalisierung. □

Um zu zeigen, dass auch c_1 primitiv rekursiv ist, betrachten wir die ganzzahlige Division

$$\mathrm{div} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x,y) \mapsto \begin{cases} \lfloor x/y \rfloor & \text{falls} \quad y > 0 \\ x & \text{sonst} \end{cases}$$

wobei $\lfloor x/y \rfloor$ die größte natürliche Zahl unterhalb von x/y ist.

Beachte:

Ist $y > 0$ und x durch y teilbar, so ist $\mathrm{div}(x,y) = x/y$. In der Definition von c_1 sind diese Bedingungen erfüllt. Es ist daher

$$\mathrm{c}_1(z) = \mathrm{div}(\mathrm{div}(z+1, 2^{\mathrm{c}_0(z)}) \dot{-} 1, 2).$$

Lemma 3.16

Die Umkehrfunktion c_1 *der Funktion* c *aus Beispiel 3.7 ist primitiv rekursiv.*

Beweis. Es genügt zu zeigen, dass div primitiv rekursiv ist. Wir betrachten dazu die Funktion

$$\mathrm{f}(x,y,z) = \begin{cases} 0 & \text{falls} \quad z \cdot y > x \\ 1 & \text{falls} \quad z \cdot y \leq x \end{cases}$$

Diese ist primitiv rekursiv, da

$$\mathrm{f}(x,y,z) = \overline{\mathrm{sign}}(z \cdot y \dot{-} x).$$

Damit ist auch $\overline{\mu}\mathrm{f}$ primitiv rekursiv und somit (unter Verwendung geeigneter Projektionen) auch

$$\mathrm{g}(x,y) := \overline{\mu}\mathrm{f}(x,y,x).$$

Es gilt nun aber

$$\mathrm{g}(x,y) = \begin{cases} \text{das kleinste } i \leq x \text{ mit } i \cdot y > x & \text{falls existent} \\ x+1 & \text{sonst} \end{cases}$$

Ist $y > 1$, so existiert so ein i stets und es ist $\mathrm{div}(x,y) = i - 1$.

Ist $y = 1$ oder $y = 0$, so existiert so ein i nicht, d.h. $x + 1$ wird ausgegeben. In diesen Fällen ist aber auch $\mathrm{div}(x,y) = x$. Also gilt:

$$\mathrm{div}(x,y) = \mathrm{g}(x,y) \dot{-} 1. \qquad \square$$

Wir betrachten nun eine einfache imperative Programmiersprache, die genau die primitiv rekursiven Funktionen berechnen kann.

LOOP-Programme sind aus den folgenden Komponenten aufgebaut:

- Variablen: x_0, x_1, x_2, ...
- Konstanten: 0, 1, 2, ... (also die Elemente von $\mathbb{N}$)
- Trennsymbole: `;` und `:=`
- Operationssymbole: $+$ und $\dot{-}$
- Schlüsselwörter: `LOOP`, `DO`, `END`

Definition 3.17 (Syntax LOOP)
Die *Syntax von LOOP-Programmen* ist induktiv definiert:

1) Jede *Wertzuweisung*

$$x_i := x_j + c \text{ und}$$
$$x_i := x_j \dot{-} c$$

für $i, j \geq 0$ und $c \in \mathbb{N}$ ist ein LOOP-Programm.

2) Falls P_1 und P_2 LOOP-Programme sind, so ist auch

$$P_1 \,;\, P_2 \quad \textit{(Hintereinanderausführung)}$$

ein LOOP-Programm.

3) Falls P ein LOOP-Programm ist und $i \geq 0$, so ist auch

$$\texttt{LOOP } x_i \texttt{ DO } P \texttt{ END}$$

ein LOOP-Programm.

Die *Semantik* dieser einfachen Sprache ist wie folgt definiert:

Bei einem LOOP-Programm, das eine Funktion $\mathrm{f} : \mathbb{N}^k \to \mathbb{N}$ berechnen soll:

- werden die *Variablen* $x_1, \ldots, x_k$ mit den *Eingabewerten* $n_1, \ldots, n_k$ vorbesetzt.
- Alle anderen Variablen erhalten den Wert 0.
- *Ausgabe* ist der Wert der Variablen x_0 nach Ausführung des Programms.

Die einzelnen Programmkonstrukte haben die folgende Bedeutung:

1) x_i := $x_j + c$
Der neue Wert der Variablen x_i ist die Summe des alten Wertes von x_j und c.

   x_i := $x_j \dot{-} c$
Der neue Wert der Variablen x_i ist der Wert von x_j minus c, falls dieser Wert ≥ 0 ist und 0 sonst.

2) P_1;P_2
Hier wird zunächst P_1 und dann P_2 ausgeführt.

3) LOOP x_i DO P END
Das Programm P wird sooft ausgeführt, wie der Wert von x_i zu Beginn angibt. Änderungen des Wertes von x_i während der Ausführung von P haben keinen Einfluss auf die Anzahl der Schleifendurchläufe.

Definition 3.18 (LOOP-berechenbar)
Die Funktion

$$\mathrm{f} : \mathbb{N}^k \to \mathbb{N}$$

heißt *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das f in dem folgenden Sinne berechnet:

- *Gestartet* mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen)
- *stoppt* P mit dem Wert $\mathrm{f}(n_1, \ldots, n_k)$ in der Variablen x_0.

Bemerkung:
Offenbar terminieren LOOP-Programme stets, da für jede Schleife eine feste Anzahl von Durchläufen durch den anfänglichen Wert der Schleifenvariablen festgelegt wird. Daher sind alle durch LOOP-Programme berechneten Funktionen total.

Beispiel 3.19

Die Additionsfunktion ist LOOP-berechenbar:

x_0 := $x_1 + 0$;
LOOP x_2 DO x_0 := $x_0 + 1$ END

Mit den Programmkonstrukten von Definition 3.17 kann man auch andere in Programmiersprachen vorhandene Konstrukte simulieren:

IF $x = 0$ THEN P END

kann z.B. simuliert werden durch

y := 1;
LOOP x DO y := 0 END;
LOOP y DO P END

wobei y eine neue Variable ist, die nicht in P vorkommt und $\neq x_0$ ist.

Satz 3.20

Die Klasse der primitiv rekursiven Funktionen stimmt mit der der LOOP-berechenbaren Funktionen überein.

Beweis.

(I) Alle primitiv rekursiven Funktionen sind LOOP-berechenbar:

- Für die *Grundfunktionen* ist klar, dass sie LOOP-berechenbar sind.
- Komposition:

$$\mathrm{f}(x_1, \ldots, x_n) = \mathrm{g}(\mathrm{h}_1(x_1, \ldots, x_k), \ldots, \mathrm{h}_m(x_1, \ldots, x_k))$$

  Es seien $P_1, \ldots, P_m, P$ LOOP-Programme für $\mathrm{h}_1, \ldots, \mathrm{h}_m, \mathrm{g}$.

  Durch Speichern der Eingabewerte und der Zwischenergebnisse in unbenutzten Variablen kann man zunächst die Werte von $\mathrm{h}_1, \ldots, \mathrm{h}_m$ mittels $P_1, \ldots, P_m$ berechnen und dann auf diese Werte P anwenden.

- primitive Rekursion:

$$\begin{aligned} \mathrm{f}(x_1, \ldots, x_n, 0) &= \mathrm{g}(x_1, \ldots, x_n) \\ \mathrm{f}(x_1, \ldots, x_n, x_{n+1} + 1) &= \mathrm{h}(x_1, \ldots, x_n, \mathrm{f}(x_1, \ldots, x_n, x_{n+1}), x_{n+1}) \end{aligned}$$

  Die Funktion f kann durch das LOOP-Programm

  z_1 := $\mathrm{g}(x_1, \ldots, x_n)$; $(\star)$
  z_2 := 0;
  z_3 := x_{n+1};
  LOOP z_3 DO
    z_1 := $\mathrm{h}(x_1, \ldots, x_n, z_1, z_2)$; $(\star)$
    z_2 := $z_2 + 1$

`END`

berechnet werden.

Dabei sind die mit ($\star$) gekennzeichneten Anweisungen Abkürzungen für Programme, welche Ein- und Ausgaben geeignet kopieren und die Programme für g und h anwenden.

Die Variablen z_1, z_2, z_3 sind neue Variablen, die in den Programmen für g und h *nicht* vorkommen.

(II) Alle LOOP-berechenbaren Funktionen sind primitiv rekursiv.

Dazu beschaffen wir uns zunächst eine primitiv rekursive Bijektion

$$\mathrm{c}^{(n)} : \mathbb{N}^n \to \mathbb{N} \quad (n \geq 2)$$

sowie die zugehörigen Umkehrfunktionen

$$\mathrm{c}_0^{(n)}, \ldots, \mathrm{c}_{n-1}^{(n)}.$$

Wir definieren dazu

$$\begin{aligned}
\mathrm{c}^{(n)}(x_1, \ldots, x_n) &:= \mathrm{c}(x_1, \mathrm{c}(x_2, \ldots, \mathrm{c}(x_{n-1}, x_n)\ldots)) \\
\mathrm{c}_0^{(n)}(z) &:= \mathrm{c}_0(z) \\
\mathrm{c}_1^{(n)}(z) &:= \mathrm{c}_1(\mathrm{c}_2(z)) \\
&\vdots \\
\mathrm{c}_{n-2}^{(n)}(z) &:= \mathrm{c}_1(\mathrm{c}_2^{(n-2)}(z)) \\
\mathrm{c}_{n-1}^{(n)}(z) &:= \mathrm{c}_2^{n-1}(z)
\end{aligned}$$

Da c und c_0, c_1 primitiv rekursiv sind, sind auch $\mathrm{c}^{(n)}$ und $\mathrm{c}_0^{(n)}, \ldots, \mathrm{c}_{n-1}^{(n)}$ primitiv rekursiv.

Es sei nun P ein LOOP-Programm, das die Funktion $\mathrm{f} : \mathbb{N}^r \to \mathbb{N}$ berechnet. Es sei l der maximale Index der in P vorkommenden Variablen und $k := \max\{r, l\}$.

Wir zeigen durch Induktion über den Aufbau von LOOP-Programmen, dass die Funktion $\mathrm{g}_P : \mathbb{N} \to \mathbb{N}$ primitiv rekursiv ist, wobei

$$\mathrm{g}_P(z) = \mathrm{c}^{(k+1)}(b_0, \ldots, b_k)$$

wenn

- $b_0, \ldots, b_k$ die Werte von $x_0, \ldots, x_k$ nach Ausführung des Programms P
- bei Startwerten $a_0 = \mathrm{c}_0^{(k+1)}(z), \ldots, a_k = \mathrm{c}_k^{(k+1)}(z)$ sind.

1) Hat P die Form x_i `:=` $x_j + c$, so ist

$$\mathrm{g}_P(z) = \mathrm{c}^{(k+1)}(\mathrm{c}_0^{(k+1)}(z), \ldots, \mathrm{c}_{i-1}^{(k+1)}(z), \mathrm{c}_j^{(k+1)}(z) + c, \mathrm{c}_{i+1}^{(k+1)}(z), \ldots).$$

Primitiv rekursiv.

Entsprechend kann die andere Zuweisung behandelt werden.

2) $P = Q;R$
Dann ist
$$\mathrm{g}_P(z) = \mathrm{g}_R(\mathrm{g}_Q(z)).$$
Da
- $\mathrm{g}_Q, \mathrm{g}_R$ nach Induktionsvoraussetzung primitiv rekursiv sind und
- g_P durch Komposition daraus entsteht

ist auch g_P primitiv rekursiv.

3) $P =$ `LOOP` x_i `DO` Q `END`
Wir definieren zunächst die zweistellige Funktion h durch primitive Rekursion aus g_Q:
$$\mathrm{h}(x, 0) = x,$$
$$\mathrm{h}(x, y+1) = \mathrm{g}_Q(\mathrm{h}(x, y))$$
Offenbar liefert $\mathrm{h}(z, n)$
- die Kodierung der Werte der Variablen $x_0, \ldots, x_k$,
- nachdem man, beginnend mit $x_0 = \mathrm{c}_0^{(k+1)}(z), \ldots, x_k = \mathrm{c}_k^{(k+1)}(z)$,
- das Programm Q n-mal ausgeführt hat.

Daher gilt:
$$\mathrm{g}_P(z) = \mathrm{h}(z, \mathrm{c}_i^{(k+1)}(z)).$$
Dies schließt den Induktionsbeweis ab, dass die Funktion g_P für jedes LOOP-Programm P primitiv rekursiv ist.
Ist
$$\mathrm{f} : \mathbb{N}^r \to \mathbb{N}$$
die von P berechnete Funktion, so gilt:
$$\mathrm{f}(z_1, \ldots, z_r) = \mathrm{c}_0^{(k+1)}(\mathrm{g}_P(\mathrm{c}^{(k+1)}(0, z_1, \ldots, z_r, \underbrace{0, \ldots, 0}_{k-r}))). \qquad \square$$

Da wir durch LOOP-berechenbare/primitiv rekursive Funktionen nur totale Funktionen erhalten, ist *nicht* jede (intuitiv) berechenbare Funktion in dieser Klasse enthalten. Aber was ist mit den totalen berechenbaren Funktionen?

Satz 3.21
Es gibt totale berechenbare Funktionen, die nicht LOOP-berechenbar sind.

Beweis. Wir definieren die Länge von LOOP-Programmen induktiv über deren Aufbau:

1) $|x_i$ `:=` $x_j + c| := i + j + c + 1$
$|x_i$ `:=` $x_j \dot{-} c| := i + j + c + 1$

2) $|P;Q| := |P| + |Q| + 1$

3) $|\texttt{LOOP } x_i \texttt{ DO } P \texttt{ END}| := |P| + i + 1$

Für eine gegebene Länge n gibt es nur endlich viele LOOP-Programme dieser Länge. Deshalb macht die folgende Definition Sinn:

$$\mathrm{f}(x,y) := 1 + \max\{\mathrm{g}(y) \mid \mathrm{g} : \mathbb{N} \to \mathbb{N} \text{ wird von einem LOOP-Programm der Länge } x \text{ berechnet}\}$$

Behauptung 1:
Die Funktion

$$\mathrm{d} : \mathbb{N} \to \mathbb{N} \text{ mit } z \mapsto \mathrm{f}(z,z)$$

ist nicht LOOP-berechenbar, denn:

Sei P ein LOOP-Programm, das d berechnet und sei $n = |P|$.

Wir betrachten $\mathrm{d}(n) = \mathrm{f}(n,n)$. Nach Definition ist $\mathrm{f}(n,n)$ größer als der maximale Funktionswert, den ein LOOP-Programm der Länge n bei Eingabe n berechnen kann. Dies widerspricht der Tatsache, dass d von einem LOOP-Programm der Länge n berechnet wird.

Behauptung 2:
Die Funktion d ist (intuitiv) berechenbar, denn:

Bei Eingabe z zählt man die *endlich vielen* LOOP-Programme der Länge z auf und wendet sie jeweils auf die Eingabe z an. Da alle diese Aufrufe terminieren, kann man in endlicher Zeit den maximalen so erhaltenen Funktionswert berechnen.

Mit der Churchschen These ist diese Funktion auch Turing-berechenbar. □

Eine prominente *nicht* primitiv rekursive, aber berechenbare Funktion ist die sogenannte *Ackermannfunktion* A, die wie folgt definiert ist:

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

Durch (geschachtelte) Induktion zeigt man leicht, dass A dadurch eindeutig definiert ist. Man kann zeigen, dass A zwar Turing-berechenbar ist, aber nicht LOOP-berechenbar, da sie schneller wächst als jede LOOP-berechenbare Funktion (siehe [Schö_01]).

4. μ-rekursive Funktionen und While-Programme

Um alle berechenbaren Funktionen zu erhalten, muss man die primitiv rekursiven Funktionen um eine weitere Operation, die *unbeschränkte Minimalisierung*, erweitern. Die LOOP-Programme muss man um eine *While-Schleife* (d.h. eine Schleife ohne vorher festgelegte Anzahl der Durchläufe) erweitern.

Definition 4.1 (unbeschränkte Minimalisierung)
Es sei $n \geq 0$. Die Funktion $f : \mathbb{N}^n \to \mathbb{N}$ entsteht aus $g : \mathbb{N}^{n+1} \to \mathbb{N}$ durch *unbeschränkte Minimalisierung* (Anwendung des *μ-Operators*), falls gilt:

$$f(x_1, \ldots, x_n) = \begin{cases} y & \text{falls } g(x_1, \ldots, x_n, y) = 0 \text{ und} \\ & g(x_1, \ldots, x_n, z) \text{ ist definiert und } \neq 0 \\ & \text{für alle } 0 \leq z < y \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Wir schreiben dann auch $f = \mu g$.

Beachte:
Der μ-Operator sucht nach dem kleinsten y, so dass $g(\underline{x}, y) = 0$ ist. Dabei müssen aber alle vorherigen Werte $g(\underline{x}, z)$ für $z < y$ definiert sein.

Daher ist $\mu g(\underline{x})$ undefiniert, falls

- kein y existiert mit $g(\underline{x}, y) = 0$ oder
- ein solches kleinstes y existiert, aber $g(\underline{x}, z)$ undefiniert ist für ein $z < y$.

Beispiel 4.2
Anwendungen des μ-Operators:

1) $g_1(x, y) = \begin{cases} x - y & \text{falls } x \geq y \\ \text{undefiniert} & \text{falls } x < y \end{cases}$

   $\mu g_1(x) = x$,

   d.h. man erhält durch Anwenden des μ-Operators auf die partielle Funktion g_1 die totale Identitätsfunktion $f_1 : \mathbb{N} \to \mathbb{N}$ mit $x \mapsto x$.

2) $g_2(x, y) = \begin{cases} y - x & \text{falls } x \leq y \\ \text{undefiniert} & \text{falls } x > y \end{cases}$

   $\mu g_2(x) = \begin{cases} 0 & \text{falls } x = 0 \\ \text{undefiniert} & \text{falls } x > 0 \quad \text{(vor Wert 0 hat man undefiniert)} \end{cases}$

3) $g_3(x, y) = x + y$

   $\mu g_3(x) = \begin{cases} 0 & \text{falls } x = 0 \\ \text{undefiniert} & \text{falls } x > 0 \quad \text{(kein Wert 0)} \end{cases}$

   Durch Anwenden von μ kann man also auch partielle Funktionen aus totalen erhalten.

Definition 4.3 (Klasse der μ-rekursiven Funktionen)
Die Klasse der *μ-rekursiven Funktionen* besteht aus den Funktionen, welche man aus den *Grundfunktionen* (Definition 3.1) durch endlich oftes Anwenden von

- Komposition,
- primitiver Rekursion und
- unbeschränkter Minimalisierung

erhält.

Definition 4.4 (Syntax von WHILE-Programmen)
Die Syntax von WHILE-Programmen enthält alle Konstrukte in der Syntax von LOOP-Programmen und zusätzlich

4) Falls P ein WHILE-Programm ist und $i \geq 0$, so ist auch
   `WHILE` $x_i \neq 0$ `DO` P `END`
   ein WHILE-Programm.

Die *Semantik* dieses Konstrukts ist wie folgt definiert:

- Das Programm P wird solange iteriert, bis x_i den Wert 0 erhält.
- Geschieht das nicht, so terminiert diese Schleife nicht.

Beachte:
Man könnte bei der Definition der WHILE-Programme auf das LOOP-Konstrukt verzichten, da es durch WHILE simulierbar ist:

`LOOP` x `DO` P `END`
kann simuliert werden durch:

y `:=` $x + 0$`;`
`WHILE` $y \neq 0$ `DO` y `:=` $y \dot{-} 1$`;`P `END`
wobei y eine neue Variable ist.

Definition 4.5 (WHILE-berechenbar)
Die (partielle) Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das f in dem folgenden Sinne berechnet:

- *Gestartet* mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen)
- *stoppt* P mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen x_0, falls dieser Wert definiert ist.
- Sonst stoppt P nicht.

Beispiel 4.6
Die Funktion

$$f : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x, y) \mapsto \begin{cases} x - y & \text{falls} \quad x \geq y \\ \text{undefiniert} & \text{sonst} \end{cases}$$

ist WHILE-berechenbar durch das folgende Programm:

```
WHILE x_2 ≠ 0 DO
  y := x_1;
  WHILE y ≠ 0 DO
    x_1 := x_1 ∸ 1;
    x_2 := x_2 ∸ 1;
    y := 0
  END
END;
x_0 := x_1
```

Satz 4.7

Die Klasse der μ-rekursiven Funktionen stimmt genau mit der Klasse der WHILE-berechenbaren Funktionen überein.

Beweis. Wir müssen hierzu den Beweis von Satz 3.20 um die Behandlung des zusätzlichen Operators/Konstrukts ergänzen.

1) Alle μ-rekursiven Funktionen sind WHILE-berechenbar:

   Es sei $\mathrm{f} = \mu\mathrm{g}$ und P (nach Induktionsvoraussetzung) ein WHILE-Programm für g. Dann berechnet das folgende Programm die Funktion f:

   ```
   x_0 := 0;
   y := g(x_1, ..., x_n, x_0);   (Realisierbar mittels P)
   WHILE y ≠ 0 DO
     x_0 := x_0 + 1;
     y := g(x_1, ..., x_n, x_0)
   END
   ```

   Beachte:

   Dieses Programm ist nur deshalb korrekt, weil wir in der Definition von μ gefordert haben, dass alle vorherigen Werte nicht nur $\neq 0$, sondern auch definiert sein müssen.

2) Alle WHILE-berechenbaren Funktionen sind μ-rekursiv:

   Betrachte das Programm `WHILE` $x_i \neq 0$ `DO` Q `END`.
   Wie im Beweis von Satz 3.20 bei der Behandlung von LOOP beschaffen wir uns eine μ-rekursive Funktion

   $$\mathrm{h} : \mathbb{N}^2 \to \mathbb{N},$$

   für die

   - $\mathrm{h}(z, n)$ die Kodierung der Werte der Variablen $x_0, \ldots, x_k$ ist, nachdem man,
   - beginnend mit $x_0 = \mathrm{c}_0^{(k+1)}(z), \ldots, x_k = \mathrm{c}_k^{(k+1)}(z)$
   - das Programm Q n-mal ausgeführt hat.

Der μ-Operator, angewandt auf die Komposition $c_i^{(k+1)}(h)$, sucht nach der kleinsten Iterationszahl, so dass die Variable $x_i = 0$ wird. Daher ist

$$g_P(z) = h(z, \mu(c_i^{(k+1)}h)(z)).$$ □

Es bleibt noch zu zeigen, dass die Klasse der μ-rekursiven/WHILE-berechenbaren Funktionen mit der Klasse der Turing-berechenbaren Funktionen übereinstimmt.

Satz 4.8

Jede μ-rekursive Funktion ist Turing-berechenbar.

Beweis. Es ist leicht zu zeigen, dass die Grundfunktionen Turing-berechenbar sind. (Übung)

Komposition: Seien $\mathcal{A}_g, \mathcal{A}_{h_1}, \ldots, \mathcal{A}_{h_m}$ DTM, die $g, h_1, \ldots, h_m$ berechnen.

Verwende $m + 1$ Bänder:

- Kopiere die Eingabe $\underline{x}$ auf Band2, ..., Band$(m + 1)$
- $\mathcal{A}_{h_i}$ berechnet $h_i(\underline{x})$ auf Band $(i + 1)$
- Überschreibe Band1 mit $\not{b}\, a^{h_1(\underline{x})}\, \not{b} \ldots \not{b}\, a^{h_m(\underline{x})}\, \not{b}$
- Berechne g-Wert davon mit $\mathcal{A}_g$

Primitive Rekursion: Seien $\mathcal{A}_g, \mathcal{A}_h$ DTM für g, h.

Verwende 4 Bänder:

- Band1: Speichert $\underline{xy}$
- Band2: Zählt von 0 ab hoch bis y (aktueller Wert: z)
- Band3: Enthält $f(\underline{x}, z)$ (für $z = 0$ mittels $\mathcal{A}_g$ berechnet)
- Band4: Berechnung von $\mathcal{A}_h$

μ-Operator: Sei $\mathcal{A}_g$ eine DTM für g.

Verwende 3 Bänder:

- Band1: Speichert Eingabe $\underline{x}$
- Band2: Zählt von 0 ab hoch (aktueller Wert: z)
- Band3: Berechne $g(\underline{x}, z)$ mittels $\mathcal{A}_g$ und teste, ob Wert $= 0$ ist □

Satz 4.9

Jede Turing-berechenbare Funktion ist WHILE-berechenbar.

Beweisskizze. Um diesen Satz zu beweisen, müssen wir Konfigurationen von Turingmaschinen der Form

$$\alpha q \beta \text{ für } \alpha, \beta \in \Gamma^+ \text{ und } q \in Q$$

in drei natürlichen Zahlen kodieren.

Diese werden dann in den drei Programmvariablen x_1, x_2, x_3 des WHILE-Programms gespeichert:

- x_1 repräsentiert α,
- x_2 repräsentiert q,
- x_3 repräsentiert β.

Es sei o.B.d.A. $\Gamma = \{a_1, \ldots, a_n\}$ und $Q = \{q_1, \ldots, q_k\}$.

Die Konfiguration

$$a_{i_1} \ldots a_{i_l} q_m a_{j_1} \ldots a_{j_r}$$

wird dargestellt durch

$$\begin{aligned} x_1 &= (i_1, \ldots, i_l)_b && := \sum_{\nu=1}^{l} i_\nu \cdot b^{l-\nu} \\ x_2 &= m \\ x_3 &= (j_r, \ldots, j_1)_b && := \sum_{\rho=1}^{r} j_\rho \cdot b^{\rho-1}, \end{aligned}$$

wobei $b > |\Gamma|$ ist, d.h.

- $a_{i_1} \ldots a_{i_l}$ repräsentiert x_1 in b-närer Zahlendarstellung und
- $a_{j_r} \ldots a_{j_1}$ (Reihenfolge!) repräsentiert x_3 in b-närer Zahlendarstellung.

Herauslesen des aktuellen Symbols a_{j_1}
Ist $x_3 = (j_r, \ldots, j_1)_b$, so ist $j_1 = x_3 \bmod b$.

Ändern dieses Symbols zu a_j
Der neue Wert von x_3 ist $(j_r, \ldots, j_2, j)_b = \mathrm{div}((j_r, \ldots, j_2, j_1)_b, b) \cdot b + j$

Verschieben des Schreib-Lesekopfes
kann durch ähnliche arithmetische Operationen realisiert werden.

All diese Operationen sind offensichtlich WHILE-berechenbar (sogar LOOP!).

Das WHILE-Programm, welches die gegebene DTM simuliert, arbeitet wie folgt:

1) Aus der Eingabe wird die Kodierung der Startkonfiguration der DTM in den Variablen x_1, x_2, x_3 erzeugt.
2) In einer WHILE-Schleife wird bei jedem Durchlauf ein Schritt der TM-Berechnung simuliert (wie oben angedeutet)
   - In Abhängigkeit vom aktuellen Zustand (Wert von x_2) und
   - dem gelesenen Symbol, d.h. von x_3 mod b
   - wird das aktuelle Symbol verändert und
   - der Schreib-Lesekopf bewegt.

Die WHILE-Schleife terminiert, wenn der aktuelle Zustand zusammen mit dem gelesenen Symbol keinen Nachfolgezustand hat.

All dies ist durch einfache (WHILE-berechenbare) arithmetische Operationen möglich.

3) Aus dem Wert von x_3 nach Termination der WHILE-Schleife wird der Ausgabewert herausgelesen und in die Variable x_0 geschrieben. □

Insgesamt haben wir also gezeigt:

Theorem 4.10
Die folgenden Klassen von Funktionen stimmen überein:

1) Turing-berechenbare Funktionen

2) WHILE-berechenbare Funktionen

3) μ-rekursive Funktionen

Diese Äquivalenz ist ein wichtiges Argument für die Korrektheit der Churchschen These.

5. Universelle Maschinen und unentscheidbare Probleme

Wir werden hier zeigen, dass es Relationen gibt, die nicht Turing-entscheidbar sind, d.h. Relationen, deren charakteristische Funktion nicht Turing-berechenbar ist. Mit der Churchschen These sind diese Relationen dann nicht entscheidbar im intuitiven Sinn.

Dazu konstruieren wir eine *universelle Turingmaschine*, die alle Turingmaschinen simulieren kann. Die universelle Maschine erhält als zusätzliche Eingabe eine Kodierung der zu simulierenden Turingmaschine.

Konventionen:

- *Arbeitsalphabete* der betrachteten Turingmaschinen sind Teilmengen von $\{a_1, a_2, a_3, \ldots\}$, wobei $a_1 = a,\ a_2 = b,\ a_3 = \not{b}$.
- *Zustandsmengen* sind Teilmengen von $\{q_1, q_2, q_3, \ldots\}$, wobei q_1 stets der *Anfangszustand* ist.

Definition 5.1 (Kodierung einer Turingmaschine)
Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_1, \Delta, F)$ eine Turingmaschine, die o.B.d.A. die obigen Konventionen erfüllt.

1) Eine *Transition*
$$t = (q_i, a_j, a_{j'}, \begin{matrix} l \\ r \\ n \end{matrix}, q_{i'})$$
wird *kodiert* durch
$$\mathrm{code}(t) = a^i b a^j b a^{j'} b \begin{matrix} a \\ aa \\ aaa \end{matrix} b a^{i'} bb.$$

2) Besteht Δ aus den Transitionen $t_1, \ldots, t_k$ und ist $F = \{q_{i_1}, \ldots, q_{i_r}\}$, so wird $\mathcal{A}$ *kodiert* durch
$$\mathrm{code}(\mathcal{A}) = \mathrm{code}(t_1) \ldots \mathrm{code}(t_k) b a^{i_1} b \ldots b a^{i_r} bbb.$$

Bemerkung 5.2.

1) Für $x = \mathrm{code}(\mathcal{A})w$ mit $w \in \Sigma^*$ folgt w genau auf den zweiten Block bbb, kann also aus x eindeutig wieder herausgelesen werden.

2) Es gibt eine DTM $\mathcal{A}_{CODE}$, welche die Relation
$$CODE = \{\mathrm{code}(\mathcal{A}) \mid \mathcal{A} \text{ ist DTM über } \Sigma\}$$
entscheidet, denn:

- Überprüfe bei Eingabe w zunächst, ob w eine Turingmaschine kodiert (d.h. eine Folge von Transitionskodierungen gefolgt von einer Endzustandsmengenkodierung ist).
- Überprüfe dann, ob die kodierte Turingmaschine deterministisch ist (bei jeder Transition wird nachgeschaut, ob es eine andere mit demselben Anfangsteil gibt).

Satz 5.3 (Turing)
Es gibt eine universelle DTM $\mathcal{U}$ über Σ, d.h. eine DTM mit der folgenden Eigenschaft: Für alle DTM $\mathcal{A}$ und alle $w \in \Sigma^*$ gilt:

$$\mathcal{U} \text{ akzeptiert } \mathrm{code}(\mathcal{A})w \quad \underline{\text{gdw}}. \quad \mathcal{A} \text{ akzeptiert } w.$$

Es ist also $\mathcal{U}$ ein „Turing-Interpreter für Turingmaschinen“.

Beweis. $\mathcal{U}$ führt bei Eingabe $\mathrm{code}(\mathcal{A})w$ die $\mathcal{A}$-Berechnung

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \dots$$

in kodierter Form aus, d.h. $\mathcal{U}$ erzeugt sukzessive Bandbeschriftungen

$$\mathrm{code}(\mathcal{A})\mathrm{code}(k_0),\ \mathrm{code}(\mathcal{A})\mathrm{code}(k_1),\ \mathrm{code}(\mathcal{A})\mathrm{code}(k_2),\ \dots$$

Konfigurationskodierung:

$$\mathrm{code}(a_{i_1} \dots a_{i_l} q_j a_{i_{l+1}} \dots a_{i_r}) = a^{i_1} b \dots a^{i_l} b a^j b b a^{i_{l+1}} b \dots a^{i_r} b$$

Arbeitsweise von $\mathcal{U}$: (bei Eingabe $\mathrm{code}(\mathcal{A})w$)

- Erzeuge aus $\mathrm{code}(\mathcal{A})w$ die Anfangsbeschriftung

$$\mathrm{code}(\mathcal{A})\mathrm{code}(\underbrace{\not b\, q_1 w\, \not b}_{k_0}).$$

- Simuliere die $\mathcal{A}$-Schritte, ausgehend vom jeweiligen Konfigurationskode

$$\dots b a^j b b a^i b \dots \text{ (Zustand } q_j\text{, gelesenes Symbol } a_i\text{)}.$$

  - Aufsuchen einer Transitionskodierung $\mathrm{code}(t)$, die mit $a^j b a^i$ beginnt.
  - Falls es so eine gibt, Änderungen der Konfigurationskodierung entsprechend t.
  - Sonst geht $\mathcal{U}$ in Stoppzustand. Dieser ist akzeptierend gdw. a^j in der Kodierung der Endzustandsmenge vorkommt. □

Satz 5.4
Die Relation

$$UNIV = \{\text{code}(\mathcal{A})w \in \Sigma^* \mid \mathcal{A} \textit{ ist DTM über } \Sigma, \textit{ die } w \textit{ akzeptiert}\}$$

ist partiell entscheidbar, aber nicht entscheidbar.

Beweis.

1) Wir zeigen, dass $UNIV$ Turing-akzeptierbar (und damit partiell entscheidbar) ist:

   Bei Eingabe x geht die TM, welche $UNIV$ akzeptiert, wie folgt vor:

   - Teste, ob x von der Form $x = x_1 w$ ist mit

     $$x_1 \in CODE \text{ und } w \in \Sigma^*.$$

   - Wenn ja, so wende $\mathcal{U}$ auf x an.

2) Angenommen, $UNIV$ ist rekursiv. Dann ist auch $\overline{UNIV} = \Sigma^* \setminus UNIV$ rekursiv und es gibt eine DTM $\mathcal{A}_0$, die $\overline{UNIV}$ entscheidet.

   Wir betrachten nun die Sprache

   $$D = \{\text{code}(\mathcal{A}) \mid \text{code}(\mathcal{A})\text{code}(\mathcal{A}) \notin UNIV\}$$

   (d.h. die Maschine $\mathcal{A}$ akzeptiert ihre eigene Kodierung nicht).

   Diese Menge kann leicht mit Hilfe von $\mathcal{A}_0$ entschieden werden:

   - Bei Eingabe x dupliziert man x und
   - startet dann $\mathcal{A}_0$ mit Eingabe xx.

   Es sei $\mathcal{A}_D$ die DTM, die D entscheidet. Es gilt nun (für $\mathcal{A} = \mathcal{A}_D$):

   $$\begin{aligned} \mathcal{A}_D \text{ akzeptiert code}(\mathcal{A}_D) \quad & \text{gdw.} \quad \text{code}(\mathcal{A}_D) \in D \\ & \text{gdw.} \quad \text{code}(\mathcal{A}_D)\text{code}(\mathcal{A}_D) \notin UNIV \\ & \text{gdw.} \quad \mathcal{A}_D \text{ akzeptiert code}(\mathcal{A}_D) \text{ nicht.} \end{aligned}$$

   Widerspruch. □

Die in Teil 2 des Beweises verwendete Vorgehensweise nennt man *Diagonalisierung*. Wir hatten hier ja zwei Dimensionen

- Kodierung der TM
- Eingabe der TM,

und die Menge D identifiziert die beiden (betrachtet TM, die als Eingabe ihre eigene Kodierung haben), d.h. D bewegt sich in der Diagonalen.

Ein anderes Beispiel für ein Diagonalisierungsargument ist Cantors Beweis für die Überabzählbarkeit von $\mathbb{R}$.

Wir wenden im folgenden Einschub Diagonalisierung an, um zu zeigen, dass es nichtkontextsensitive Typ-0-Sprachen gibt.

Satz 5.5
$\mathcal{L}_1 \subset \mathcal{L}_0$

Beweis. Es sei

- $G_0, G_1, \ldots$ eine effektive (d.h. mit TM machbare) Aufzählung aller kontextsensitiven Grammatiken mit Terminalalphabet $\Sigma = \{a, b\}$
- und $w_0, w_1, \ldots$ eine effektive Aufzählung aller Wörter über Σ.

Wir definieren nun $L \subseteq \{a, b\}^*$ durch

$$w_i \in L :\Leftrightarrow w_i \notin L(G_i) \text{ (Diagonalisierung!).}$$

- Es ist L Turing-akzeptierbar, da man für eine kontextsensitive Grammatik G_i und ein Wort w_i entscheiden kann, ob $w_i \in L(G_i)$ ist.

  Beachte:

  nichtkürzende Produktionen $\Rightarrow$ Aufzählen aller ableitbaren Wörter bis zur Länge $|w_i|$ genügt.

- L ist nicht kontextsensitiv. Anderenfalls gäbe es einen Index k mit $L = L(G_k)$. Nun ist aber

  $$w_k \in L(G_k) \;\; \underline{\text{gdw}}. \;\; w_k \in L \;\; \underline{\text{gdw}}. \;\; w_k \notin L(G_k).$$

  Widerspruch. □

Aus der Unentscheidbarkeit von $UNIV$ kann man weitere wichtige Unentscheidbarkeitsresultate herleiten.

Satz 5.6
Das Wortproblem für DTM (und damit auch für $\mathcal{L}_0$) ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das zu jeder gegebenen DTM $\mathcal{A}$ und jedem Eingabewort w entscheidet, ob $\mathcal{A}$ das Wort w akzeptiert.

Beweis. Wenn das Wortproblem für DTM entscheidbar wäre, so wäre auch $UNIV$ entscheidbar (ist es aber nicht).

Um zu entscheiden, ob $\text{code}(\mathcal{A})w \in UNIV$ ist, müsste man ja nur das Entscheidungsverfahren für das Wortproblem feststellen lassen, ob $\mathcal{A}$ das Wort w akzeptiert. □

Satz 5.7

Das Halteproblem für DTM ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das zu jeder gegebenen DTM $\hat{\mathcal{A}}$ entscheidet, ob $\hat{\mathcal{A}}$ beginnend mit leerem Eingabeband terminiert.

Beweis. Wäre das Halteproblem entscheidbar, so auch das Wortproblem.

Um zu gegebener TM $\mathcal{A}$ und gegebenem Wort w zu entscheiden, ob $\mathcal{A}$ das Wort w akzeptiert, könnte man dann nämlich wie folgt vorgehen:

Modifiziere $\mathcal{A}$ zu einer TM $\hat{\mathcal{A}}$:

- Bei leerem Band schreibt $\hat{\mathcal{A}}$ zunächst w auf das Band.
- Danach verhält sich $\hat{\mathcal{A}}$ wie $\mathcal{A}$.
- Stoppt $\mathcal{A}$ mit *akzeptierender* Stoppkonfiguration, so stoppt auch $\hat{\mathcal{A}}$.
  Hält $\mathcal{A}$ mit *nichtakzeptierender* Stoppkonfiguration, so geht $\hat{\mathcal{A}}$ in eine Endlosschleife.

Damit gilt:

$$\hat{\mathcal{A}} \text{ hält mit leerem Eingabeband} \quad \underline{\text{gdw}}. \quad \mathcal{A} \text{ akzeptiert } w.$$

Mit dem Entscheidungsverfahren für das Halteproblem (angewandt auf $\hat{\mathcal{A}}$) könnte man also das Wortproblem („Ist w in $L(\mathcal{A})$?“) entscheiden. □

Satz 5.8

Das Leerheitsproblem für DTM (und damit auch für $\mathcal{L}_0$) ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das bei gegebener DTM $\hat{\mathcal{A}}$ entscheidet, ob es eine Eingabe w gibt, auf der $\hat{\mathcal{A}}$ terminiert.

Beweis. Wäre das Leerheitsproblem entscheidbar, so auch das Halteproblem.

Um bei gegebener DTM $\mathcal{A}$ zu entscheiden, ob $\mathcal{A}$ auf leerer Eingabe hält, konstruiert man die DTM $\hat{\mathcal{A}}$ wie folgt:

- $\hat{\mathcal{A}}$ löscht seine Eingabe.
- Danach verhält sich $\hat{\mathcal{A}}$ wie $\mathcal{A}$.
- Stoppt die Berechnung, so geht $\hat{\mathcal{A}}$ in eine akzeptierende Stoppkonfiguration.

Offenbar gibt es eine Eingabe, für die $\hat{\mathcal{A}}$ hält $\underline{\text{gdw}}$. $\mathcal{A}$ auf leerem Eingabeband hält. □

Satz 5.9

Das Äquivalenzproblem für DTM (und damit auch für $\mathcal{L}_0$) ist unentscheidbar.

Beweis. Offenbar kann man leicht eine DTM $\hat{\mathcal{A}}$ konstruieren mit $L(\hat{\mathcal{A}}) = \emptyset$.

Wäre das Äquivalenzproblem

$$L(\mathcal{A}_1) = L(\mathcal{A}_2)?$$

entscheidbar, so könnte man durch den Test

$$L(\mathcal{A}) = L(\hat{\mathcal{A}})?$$

das Leerheitsproblem für $\mathcal{A}$ entscheiden. □

Satz 5.10
$\mathcal{L}_0$ ist nicht unter Komplement abgeschlossen.

Beweis. Wir wissen von der in Satz 5.4 eingeführten Sprache $UNIV$:

- $UNIV$ ist partiell entscheidbar, d.h. gehört zu $\mathcal{L}_0$.
- $UNIV$ ist *nicht* entscheidbar.

Wäre $\overline{UNIV} \in \mathcal{L}_0$, d.h. partiell entscheidbar, so würde aber mit Satz 2.4 (Teil 4) folgen, dass $UNIV$ *entscheidbar* ist. □

Wie kann man *Unentscheidbarkeit* eines Problems (formal: einer Relation) *zeigen*?

1. Durch ein *Diagonalisierungsargument* wie im Beweis von Satz 5.4.
2. Das in den Beweisen der Sätze 5.6 bis 5.9 gewählte Vorgehen nennt man *Reduktion*:
   - Ein Problem P_1 (z.B. Halteproblem) wird *auf* ein Problem P_2 (z.B. Äquivalenzproblem) *reduziert*.
   - Wäre daher P_2 *entscheidbar*, so *auch* P_1.
   - Weiss man bereits, dass P_1 *unentscheidbar* ist, so folgt daher, dass *auch* P_2 unentscheidbar ist.

Formaler betrachten wir (o.B.d.A.) einstellige Relationen $R \subseteq \Sigma^*$.

Definition 5.11 (Reduktion)

1) Eine *Reduktion von* $R_1 \subseteq \Sigma^*$ *auf* $R_2 \subseteq \Sigma^*$ ist eine berechenbare Funktion

$$\mathrm{f} : \Sigma^* \to \Sigma^*,$$

für die gilt:

$$w \in R_1 \ \ \underline{\text{gdw}}. \ \ \mathrm{f}(w) \in R_2.$$

2) Wir schreiben

$$R_1 \leq_m R_2 \quad (R_1 \text{ wird auf } R_2 \textit{ reduziert}),$$

falls es eine Reduktion von R_1 nach R_2 gibt.

Lemma 5.12

1) $R_1 \leq_m R_2$ und R_2 entscheidbar $\Rightarrow$ R_1 entscheidbar.

2) $R_1 \leq_m R_2$ und R_1 unentscheidbar $\Rightarrow$ R_2 unentscheidbar.

Beweis.

1) Um „$w \in R_1$“ zu entscheiden,
   - berechnet man $\mathrm{f}(w)$ und
   - entscheidet „$\mathrm{f}(w) \in R_2$“.
2) Folgt unmittelbar aus 1). □

Mit Hilfe einer Reduktion des Halteproblems kann man zeigen:

Jede *nichttriviale semantische Eigenschaft* von Programmen (DTM) ist *unentscheidbar.*

- *Semantisch* heißt hier: Die Eigenschaft hängt nicht von der syntaktischen Form des Programms, sondern nur von der berechneten Funktion ab.
- *Nichttrivial:* Es gibt berechenbare Funktionen, die sie erfüllen, aber nicht alle berechenbaren Funktionen erfüllen sie.

Beispiele für solche Eigenschaften:

- die berechnete Funktion ist total, d.h. die DTM hält für jede Eingabe.
- die berechnete Funktion ist bei Eingabe ε definiert (Halteproblem).
- die berechnete Funktion ist die Nullfunktion, d.h. der Funktionswert ist für jede Eingabe gleich 0.
- die berechnete Funktion liefert bei Eingabe ε den Wert 0.
- ...

Satz 5.13 (Satz von Rice)
Es sei E eine Eigenschaft partiell berechenbarer Funktionen $\mathrm{f} : \Sigma^* \to \Sigma^*$, so dass gilt:

$$\emptyset \subset \{\mathrm{f} : \Sigma^* \to \Sigma^* \mid \mathrm{f} \textit{ erfüllt } E\} \subset \{\mathrm{f} : \Sigma^* \to \Sigma^* \mid \mathrm{f} \textit{ ist partiell berechenbar}\}$$

Dann ist

$$L(E) := \{\mathrm{code}(\mathcal{A}) \mid \textit{die von } \mathcal{A} \textit{ berechnete Funktion } \mathrm{f} : \Sigma^* \to \Sigma^* \textit{ erfüllt } E\}$$

unentscheidbar.

Henry Gordon Rice
(geb. 1. Januar 1920)
Amerikanischer Mathematiker und Logiker
Professor an der University of New Hampshire
Beweis des Theorems im Jahr 1951 in seiner Doktorarbeit an der Syracuse University

Beweis. Angenommen, $L(E)$ ist entscheidbar.

Wir zeigen, dass man dann auch ein Entscheidungsverfahren für das Halteproblem erhält (Widerspruch zu Satz 5.7).

Mit f_u bezeichnen wir die überall undefinierte Funktion, welche offenbar partiell berechenbar ist. O.B.d.A. erfülle f_u die Eigenschaft E.
Sonst könnte man statt E die Eigenschaft $\overline{E}$: „f erfüllt E nicht“ betrachten. Mit $L(E)$ ist auch $L(\overline{E}) = \overline{L(E)}$ entscheidbar.

Da nicht alle partiell berechenbaren Funktionen E erfüllen, gibt es eine partiell berechenbare Funktion g, welche E *nicht* erfüllt.

Es sei nun $\mathcal{A}_\mathrm{g}$ eine DTM für g und $\mathcal{A}_u$ eine für f_u. Wir konstruieren nun eine DTM $\mathcal{A}'$, welche eine Funktion $\mathrm{f} : \Sigma^* \to \Sigma^*$ berechnet mit

$$\begin{array}{l} w = \mathrm{code}(\mathcal{A}) \text{ für eine DTM} \mathcal{A}, \\ \text{die auf leerer Eingabe terminiert} \end{array} \qquad \underline{\text{gdw}}.\ \ \mathrm{f}(w) \notin L(E) \quad (\star).$$

Wäre daher $L(E)$ entscheidbar, so auch das Halteproblem.

Konstruktion von $\mathcal{A}'$:
Die DTM $\mathcal{A}'$ testet bei Eingabe w zunächst, ob w Kodierung einer DTM ist.

- Falls nein, so gibt $\mathcal{A}'$ $\mathrm{code}(\mathcal{A}_u) \in L(E)$ aus.
- Falls ja, so ist $w = \mathrm{code}(\mathcal{A})$ für eine DTM $\mathcal{A}$. Die DTM $\mathcal{A}'$ gibt dann $\mathrm{code}(\mathcal{A}'')$ aus, wobei $\mathcal{A}''$ noch geeignet zu definieren ist:

$$\mathrm{code}(\mathcal{A}'') \notin L(E) \ \ \underline{\text{gdw}}.\ \ \mathcal{A} \text{ hält auf leerer Eingabe}$$

Definition von $\mathcal{A}''$:

- $\mathcal{A}''$ ignoriert zunächst die Eingabe x und simuliert das Verhalten von $\mathcal{A}$ auf dem leeren Eingabeband.
- Im Anschluss (d.h. falls $\mathcal{A}$ auf leerem Eingabeband terminiert) verhält sich $\mathcal{A}''$ wie $\mathcal{A}_\mathrm{g}$ bei Eingabe x.

Damit gilt für $\mathcal{A}''$ offenbar:

- Terminiert $\mathcal{A}$ auf leerer Eingabe *nicht*, so berechnet $\mathcal{A}''$ die Funktion f_u, d.h. $\mathrm{code}(\mathcal{A}'') \in L(E)$, da f_u E erfüllt.
- Terminiert $\mathcal{A}$ auf leerer Eingabe, so berechnet $\mathcal{A}''$ die Funktion g, d.h. $\mathrm{code}(\mathcal{A}'') \notin L(E)$, da g E nicht erfüllt.

Insgesamt haben wir also gezeigt, dass die von $\mathcal{A}'$ berechnete Funktion f die Bedingung $(\star)$ erfüllt, das Halteproblem also auf das Problem, $L(E)$ zu entscheiden, reduziert.

Da das Halteproblem unentscheidbar ist, folgt die Unentscheidbarkeit von $L(E)$. □

6. Weitere unentscheidbare Probleme

Die bisher als unentscheidbar nachgewiesenen Probleme betreffen alle nur (semantische) Eigenschaften von Programmen. Aber auch Probleme, die (direkt) nichts mit Programmen zu tun haben, können unentscheidbar sein.

Das folgende von Emil Post definierte Problem ist sehr nützlich, um mittels Reduktion Unentscheidbarkeit zu zeigen. Wir werden es verwenden um Unentscheidbarkeit von Problemen für kontextfreie/kontextsensitive Grammatiken und (später) für die Prädikatenlogik erster Stufe zu zeigen.

Emil Leon Post
(geb. 11. Februar 1897 in Augustow, gest. 21. April 1954 in New York)
Amerikanischer Mathematiker und Logiker
Doktorarbeit an der Columbia University, PostDoc in Princeton
Danach Mathematiklehrer in New York

Definition 6.1 (Postsches Korrespondenzproblem)
Eine Instanz des *Postschen Korrespondenzproblems (PKP)* ist gegeben durch eine endliche Folge

$$P = (x_1, y_1), \ldots, (x_k, y_k)$$

von Wortpaaren mit $x_i, y_i \in \Sigma^+$ für ein endliches Alphabet Σ.

Eine *Lösung* des Problems ist eine *Indexfolge* $i_1, \ldots, i_m$ mit

- $m > 0$ und
- $i_j \in \{1, \ldots, k\}$,

so dass gilt: $x_{i_1} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$.

Beispiel 6.2
Zwei Beispielinstanzen des PKP:

1) $P_1 = (a, aaa), (abaa, ab), (aab, b)$
   hat z.B. die Folgen 2, 1 und 1, 3 als Lösungen

   $abaa|a$ $\quad$ $a|aab$
   $ab|aaa$ $\quad$ $aaa|b$

   und damit auch 2, 1, 1, 3 sowie 2, 1, 2, 1, 2, 1, ...

2) $P_2 = (ab, aba), (baa, aa), (aba, baa)$
   hat keine Lösung:

   $ab|aba|aba|\ldots$
   $aba|baa|baa|\ldots$

Um die Unentscheidbarkeit des PKP zu zeigen, führen wir zunächst ein Zwischenproblem ein, das *modifizierte PKP (MPKP)*:

Hier muss für die Lösung zusätzlich $i_1 = 1$ gelten, d.h. das Wortpaar, mit dem man beginnen muss, ist festgelegt.

Lemma 6.3
Das MPKP kann auf das PKP reduziert werden.

Beweis. Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$ eine Instanz des MPKP über dem Alphabet Σ. Es seien $\#, \$$ Buchstaben, die nicht in Σ vorkommen.

Wir definieren die Instanz $\mathrm{f}(P)$ des PKP über $\hat{\Sigma} = \Sigma \cup \{\#, \$\}$ wie folgt:

$$\mathrm{f}(P) := (x'_0, y'_0), (x'_1, y'_1), \ldots, (x'_k, y'_k), (x'_{k+1}, y'_{k+1}),$$

wobei gilt:

- Für $1 \leq i \leq k$ entsteht x'_i aus x_i, indem man *hinter jedem* Buchstaben ein $\#$ einfügt.
  Ist z.B. $x_i = abb$, so ist $x'_i = a\#b\#b\#$.
- Für $1 \leq i \leq k$ entsteht y'_i aus y_i, indem man *vor jedem* Buchstaben ein $\#$ einfügt.
- $x'_0 := \#x'_1$ und $y'_0 := y'_1$
- $x'_{k+1} := \$$ und $y'_{k+1} := \#\$$

Offenbar ist f berechenbar, und man kann leicht zeigen, dass gilt:

„Das MPKP P hat eine Lösung.“ gdw. „Das PKP $\mathrm{f}(P)$ hat eine Lösung.“ □

Beispiel:
$P = (a, aaa), (aab, b)$ ist als MPKP lösbar mit Lösung 1, 2.

$$\begin{array}{rcccc} & (x'_0, y'_0) & (x'_1, y'_1) & (x'_2, y'_2) & (x'_3, y'_3) \\ \mathrm{f}(P) = & (\#a\#, \#a\#a\#a), & (a\#, \#a\#a\#a), & (a\#a\#b\#, \#b), & (\$, \#\$) \end{array}$$

Die Lösung 1, 2 von P liefert die Lösung 0, 2, 3 von $\mathrm{f}(P)$:

$\#a\#|\, a\#a\#b\#|\$$
$\#a\#\, a\#a|\#b|\#\$$

Offenbar muss jede Lösung von $\mathrm{f}(P)$ mit 0 beginnen.

Wäre daher das PKP entscheidbar, so auch das MPKP. Um die Unentscheidbarkeit des PKP zu zeigen, genügt es also zu zeigen, dass das MPKP unentscheidbar ist.

Lemma 6.4
Das Halteproblem kann auf das MPKP reduziert werden.

Beweis. Gegeben sei eine DTM $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ und ein Eingabewort $w \in \Sigma^*$.

Wir müssen zeigen, wie man $\mathcal{A}, w$ effektiv in eine Instanz $\mathrm{f}(\mathcal{A}, w)$ des MPKP überführen kann, so dass gilt:

$$\mathcal{A} \text{ hält auf Eingabe } w \quad \underline{\text{gdw}}. \quad \mathrm{f}(\mathcal{A}, w) \text{ hat eine Lösung.}$$

Wir verwenden für das MPKP $\mathrm{f}(\mathcal{A}, w)$ das Alphabet $\Gamma \cup Q \cup \{\#\}$ mit $\# \notin \Gamma \cup Q$. Das MPKP $\mathrm{f}(\mathcal{A}, w)$ besteht aus den folgenden Wortpaaren:

1) Anfangsregel:
$$(\#,\ \#\not b\, q_0 w\, \not b\#)$$

2) Kopierregeln:
$$(a,\ a) \text{ für alle } a \in \Gamma \cup \{\#\}$$

3) Übergangsregeln:
$$\begin{aligned}
(qa,\ q'a') &\quad \text{falls } (q, a, a', n, q') \in \Delta \\
(qab,\ a'q'b) &\quad \text{falls } (q, a, a', r, q') \in \Delta,\ b \in \Gamma \\
(cbqa,\ cq'ba') &\quad \text{falls } (q, a, a', l, q') \in \Delta \text{ und } b, c \in \Gamma \\
(\#bqa,\ \#\not b\, q'ba') &\quad \text{falls } (q, a, a', l, q') \in \Delta,\ b \in \Gamma \\
(qa\#,\ a'q'\not b\#) &\quad \text{falls } (q, \not b, a', r, q') \in \Delta
\end{aligned}$$

4) Löschregeln:
(aq, q) und (qa, q) für alle $a \in \Gamma$ und $q \in Q$ Stoppzustand
(O.B.d.A. hänge in $\mathcal{A}$ das Stoppen nur vom erreichten Zustand ab.)

5) Abschlussregel:
$$(q\#\#\#, \#) \text{ für alle } q \in Q \text{ mit } q \text{ Stoppzustand}$$

Falls $\mathcal{A}$ bei Eingabe w hält, so gibt es eine Folge von Konfigurationsübergängen

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_t$$

mit $k_0 = \not b\, q_0 w\, \not b$ und $k_t = u\hat{q}v$ mit $\hat{q}$ Stoppzustand.

Daraus kann man nun eine Lösung des MPKP bauen. Zunächst erzeugt man

$$\begin{aligned}
&\#\#k_0\#\#k_1\#\#k_2\#\#\ldots\# \\
&\#\#k_0\#\#k_1\#\#k_2\#\#\ldots\#\#k_t\#
\end{aligned}$$

- Dabei beginnt man mit $(\#, \#\#k_0\#)$.
  - Durch Kopierregeln erzeugt man die Teile von k_0 und k_1, die sich nicht unterscheiden.

  - Der Teil, der sich unterscheidet, wird durch die entsprechende Übergangsregel realisiert.

  Man erhält so:
  $\#\#k_0\#$
  $\#\#k_0\#\#k_1\#$

- Nun macht man dies so weiter, bis die Stoppkonfiguration k_t mit Stoppzustand $\hat{q}$ erreicht ist:

$$\#\#k_0\#\#k_1\#\#k_2\#\#\ldots\#$$
$$\#\#k_0\#\#k_1\#\#k_2\#\#\ldots\#\#k_t\#$$

- Durch Verwenden von Löschregeln und Kopierregeln löscht man nacheinander die dem Stoppzustand benachbarten Symbole von k_t.
- Danach wendet man die Abschlussregel an und erhält eine Lösung des MPKP.

Umgekehrt zeigt man leicht, dass jede Lösung des MPKP einer haltenden Folge von Konfigurationsübergängen entspricht, welche mit k_0 beginnt:

- Man muss mit k_0 beginnen, da wir das MPKP betrachten.
- Durch Kopier- und Übergangsregeln kann man die erzeugten Wörter nicht gleich lang machen.
- Daher muss ein Stoppzustand erreicht werden, damit Lösch- und Abschlussregeln eingesetzt werden können. □

Da das Halteproblem unentscheidbar ist, folgt die Unentscheidbarkeit des MPKP und damit (wegen Lemma 6.3) die Unentscheidbarkeit des PKP.

Satz 6.5
Das PKP ist unentscheidbar.

Wir verwenden dieses Resultat, um Unentscheidbarkeit von Problemen für kontextfreie und kontextsensitive Sprachen nachzuweisen. Wir zeigen zunächst:

Lemma 6.6
Es ist nicht entscheidbar, ob für kontextfreie Grammatiken G_1, G_2 gilt:

$$L(G_1) \cap L(G_2) \neq \emptyset.$$

Beweis. Wir reduzieren das PKP auf dieses Problem, d.h. wir zeigen:

Zu jeder Instanz P des PKP kann man effektiv kontextfreie Grammatiken $G_P^{(l)}, G_P^{(r)}$ konstruieren, so dass gilt:

$$P \text{ hat Lösung} \quad \underline{\text{gdw}}. \quad L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset.$$

Da das PKP unentscheidbar ist, folgt dann die Aussage des Lemmas.

Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$. Wir definieren $G_P^{(l)} = (N_l, \Sigma_l, P_l, S_l)$ mit

- $N_l = \{S_l\}$,
- $\Sigma_l = \Sigma \cup \{1, \ldots, k\}$ und
- $P_l = \{S_l \longrightarrow w_i S_l i, S_l \longrightarrow w_i i \mid 1 \leq i \leq k\}$.

$G_P^{(r)}$ wird entsprechend definiert. Es gilt:

$$L(G_P^{(l)}) = \{x_{i_1} \ldots x_{i_m} i_m \ldots i_1 \mid m \geq 1, i_j \in \{1, \ldots, k\}\}$$

$$L(G_P^{(r)}) = \{y_{i_1} \ldots y_{i_m} i_m \ldots i_1 \mid m \geq 1, i_j \in \{1, \ldots, k\}\}$$

Daraus folgt nun unmittelbar:

$$\begin{aligned} & L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset \\ \underline{\text{gdw}}. \quad & \exists m \geq 1 \; \exists i_1, \ldots, i_m \in \{1, \ldots, k\} : x_{i_1} \ldots x_{i_m} i_m \ldots i_1 = y_{i_1} \ldots y_{i_m} i_m \ldots i_1 \\ \underline{\text{gdw}}. \quad & P \text{ hat Lösung.} \end{aligned}$$

□

Da jede kontextfreie Sprache auch kontextsensitiv ist und da die kontextsensitiven Sprachen unter Durchschnitt abgeschlossen sind, folgt daraus unmittelbar:

Satz 6.7

Für $\mathcal{L}_1$ sind das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.

Beachte:

Das Leerheitsproblem ist ein Spezialfall des Äquivalenzproblems, da

$$L(G) = \emptyset \;\; \underline{\text{gdw}}. \;\; L(G) = L(G_\emptyset) \quad (G_\emptyset : \text{ kontextsensitive Grammatik für } \emptyset).$$

Für kontextfreie Sprachen wissen wir bereits, dass das Leerheitsproblem entscheidbar ist. Dies ist kein Widerspruch zu Lemma 6.6, da $\mathcal{L}_2$ nicht unter Durchschnitt abgeschlossen ist.

Satz 6.8

Für $\mathcal{L}_2$ ist das Äquivalenzproblem unentscheidbar.

Beweis.

1. Man kann sich leicht überlegen, dass die Sprachen $L(G_P^{(l)})$ und $L(G_P^{(r)})$ aus dem Beweis von Lemma 6.6 durch deterministische Kellerautomaten akzeptiert werden können.
2. Die von deterministischen Kellerautomaten akzeptierten kontextfreien Sprachen sind nun aber unter Komplement abgeschlossen.

   D.h. es gibt auch einen (effektiv berechenbaren) deterministischen Kellerautomaten und damit eine kontextfreie Grammatik für

   $$\overline{L(G_P^{(l)})}$$

   (siehe z.B. Satz 10.16 im Skript „Formale Systeme“, oder in [Wege_99]).

3. Es sei $\overline{G}$ die kontextfreie Grammatik mit $L(\overline{G}) = \overline{L(G_P^{(l)})}$. Nun gilt:

$$\begin{aligned} L(G_P^{(l)}) \cap L(G_P^{(r)}) = \emptyset \quad & \underline{\text{gdw}}. \quad L(G_P^{(r)}) \subseteq L(\overline{G}) \\ & \underline{\text{gdw}}. \quad L(G_P^{(r)}) \cup L(\overline{G}) = L(\overline{G}) \\ & \underline{\text{gdw}}. \quad L(G_\cup) = L(\overline{G}), \end{aligned}$$

   wobei $G_\cup$ die effektiv konstruierbare kontextfreie Grammatik für $L(G_P^{(r)}) \cup L(\overline{G})$ ist.

4. Wäre also das Äquivalenzproblem für kontextfreie Grammatiken entscheidbar, so auch
$$L(G_P^{(l)}) \cap L(G_P^{(r)}) = \emptyset$$
   und damit das PKP. □

Als nächstes verwenden wir das PKP, um zu zeigen, dass die Gültigkeit von Formeln der Prädikatenlogik erster Stufe (PL1) unentscheidbar ist.

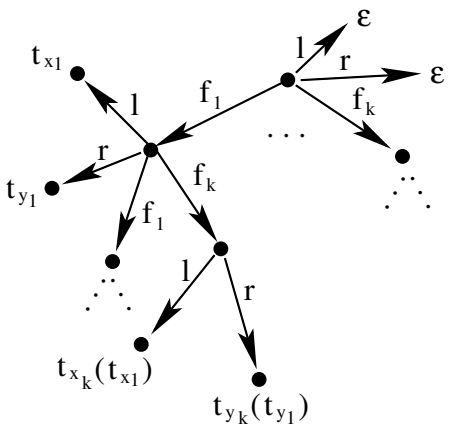
Satz 6.9
Das PKP kann auf die Gültigkeit von PL1-Formeln reduziert werden.

Beweis. Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$ ein PKP über dem Alphabet $\Sigma = \{a_1, \ldots, a_n\}$.
Wir verwenden

- einstellige Funktionssymbole $a_1, \ldots, a_n$,
- einstellige Funktionssymbole $f_1, \ldots, f_k$,
- einstellige Funktionssymbole ℓ, r,
- das Gleichheitssymbol $=$ (welches als Gleichheit interpretiert werden muss),
- ein Konstantensymbol ε.

Ein Wort
$$w = a_{i_1} \ldots a_{i_m} \in \Sigma^*$$
wird dargestellt als Term
$$t_w = a_{i_m}(a_{i_{m-1}}(\ldots(a_{i_1}(\varepsilon))\ldots)).$$
Die Konkatenation entspricht daher dem Einsetzen von Termen, d.h.
$$t_{uv} = t_v(t_u).$$

Idee:
Der Suchbaum für eine Lösung des PKP kann wie folgt dargestellt werden:



Dies wird durch die folgende Formel beschrieben:

$$\begin{aligned} B_P : \exists x.\ & (\ell(x) = \varepsilon \quad \wedge \quad r(x) = \varepsilon) \quad \wedge \\ & \forall x.\ (\ell(f_1(x)) = t_{x_1}(\ell(x)) \neq \varepsilon \quad \wedge \quad r(f_1(x)) = t_{y_1}(r(x)) \neq \varepsilon \quad \wedge \\ & \vdots \\ & \ell(f_k(x)) = t_{x_k}(\ell(x)) \neq \varepsilon \quad \wedge \quad r(f_k(x)) = t_{y_k}(r(x)) \neq \varepsilon) \end{aligned}$$

Die Lösbarkeit des PKP wird dann ausgedrückt durch die Formel:

$$F_P :\ B_P \ \Rightarrow\ \exists z.\ (\ell(z) = r(z) \neq \varepsilon)$$

Behauptung:

F_P ist gültig gdw. P eine Lösung hat.

denn: Jede Interpretation, die B_P erfüllt, „enthält“ den Suchbaum.

Beweis der Behauptung.

„⇒“: Angenommen, F_P ist gültig. Wir benutzen den Suchbaum, um eine Interpretation $\mathcal{I}$ zu definieren, die B_P erfüllt:

- $dom(\mathcal{I}) := \Sigma^* \cup \{1, \ldots, k\}^*$
- $\varepsilon^{\mathcal{I}} := \varepsilon$
- $a_i : dom(\mathcal{I}) \to dom(\mathcal{I})$

  $$a_i^{\mathcal{I}}(u) := \begin{cases} ua_i & \text{falls } u \in \Sigma^* \\ \varepsilon & \text{sonst} \end{cases}$$
- $f_j : dom(\mathcal{I}) \to dom(\mathcal{I})$

  $$f_j^{\mathcal{I}}(u) := \begin{cases} uj & \text{falls } u \in \{1, \ldots, k\}^* \\ j & \text{sonst} \end{cases}$$
- $\ell : dom(\mathcal{I}) \to dom(\mathcal{I})$

  $$\ell^{\mathcal{I}}(u) := \begin{cases} x_{i_1} \ldots x_{i_m} & \text{falls } u = i_1 \ldots i_m \in \{1, \ldots, k\}^* \\ \varepsilon & \text{sonst} \end{cases}$$

- $r : dom(\mathcal{I}) \to dom(\mathcal{I})$

$$r^{\mathcal{I}}(u) := \begin{cases} y_{i_1} \dots y_{i_m} & \text{falls } u = i_1 \dots i_m \in \{1, \dots, k\}^* \\ \varepsilon & \text{sonst} \end{cases}$$

Man sieht leicht, dass $\mathcal{I}$ die Formel B_P wahr macht:

- $\exists x.\ \ell(x) = \varepsilon \wedge r(x) = \varepsilon$ ist erfüllt, da

$$\ell^{\mathcal{I}}(\varepsilon^{\mathcal{I}}) = \varepsilon = \varepsilon^{\mathcal{I}} \text{ und } r^{\mathcal{I}}(\varepsilon^{\mathcal{I}}) = \varepsilon = \varepsilon^{\mathcal{I}}.$$

- Sei $u \in dom(\mathcal{I})$:
  - $u = i_1 \dots i_m \in \{1, \dots, k\}^*$:

$$\begin{array}{ccccc} \ell^{\mathcal{I}}(f_j^{\mathcal{I}}(u)) & = & \ell^{\mathcal{I}}(i_1 \dots i_m j) & = & x_{i_1} \dots x_{i_m} x_j \\ & & & & \shortparallel \\ t_{x_j}^{\mathcal{I}}(\ell^{\mathcal{I}}(u)) & = & t_{x_j}^{\mathcal{I}}(x_{i_1} \dots x_{i_m}) & = & x_{i_1} \dots x_{i_m} x_j \end{array}$$

  - $u \notin \{1, \dots, k\}^*$:

$$\begin{array}{ccccc} \ell^{\mathcal{I}}(f_j^{\mathcal{I}}(u)) & = & \ell^{\mathcal{I}}(j) & = & x_j \\ & & & & \shortparallel \\ t_{x_j}^{\mathcal{I}}(\ell^{\mathcal{I}}(u)) & = & t_{x_j}^{\mathcal{I}}(\varepsilon) & = & x_j \end{array}$$

- Für r geht das entsprechend.

Also erfüllt $\mathcal{I}$ die Formel B_P. Damit muss $\mathcal{I}$ auch

$$\exists z.\ (\ell(z) = r(z) \neq \varepsilon)$$

erfüllen, d.h. es gibt ein $u \in dom(\mathcal{I})$ mit

$$\ell^{\mathcal{I}}(u) = r^{\mathcal{I}}(u) \neq \varepsilon^{\mathcal{I}} = \varepsilon.$$

1. Fall: $u = i_1 \dots i_m \in \{1, \dots, k\}^+$. Dann ist

$$\begin{array}{ccc} \ell^{\mathcal{I}}(u) & = & x_{i_1} \dots x_{i_m} \\ \shortparallel & & \\ r^{\mathcal{I}}(u) & = & y_{i_1} \dots y_{i_m} \end{array}$$

d.h. $i_1 \dots i_m$ ist eine Lösung des PKP.

2. Fall: $u \in \Sigma^*$.

- $u = \varepsilon$: $\ell^{\mathcal{I}}(u) = \varepsilon = r^{\mathcal{I}}(u)$ kann nicht sein
- $u \in \Sigma^+$: Dann ist ebenfalls $\ell^{\mathcal{I}}(u) = \varepsilon = r^{\mathcal{I}}(u)$, d.h. auch dieser Fall kann nicht eintreten.

„⇐“: Angenommen, P hat eine Lösung. Dann sei $\mathcal{I}$ eine Interpretation, die B_P wahr macht.

Zu zeigen: $\mathcal{I}$ erfüllt auch

$$\exists z.\ (\ell(z) = r(z) \neq \varepsilon).$$

Es sei $i_1 \dots i_m$ für $m > 0$ eine Lösung von P, d.h.

$$x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}.$$

Da $\mathcal{I}$ die Formel B_P erfüllt, gilt:

$$\begin{aligned} \ell^{\mathcal{I}}(f^{\mathcal{I}}_{i_m}(\cdots f^{\mathcal{I}}_{i_1}(\varepsilon^{\mathcal{I}}))) &= t^{\mathcal{I}}_{x_{i_1}\dots x_{i_m}} \neq \varepsilon^{\mathcal{I}} \\ r^{\mathcal{I}}(f^{\mathcal{I}}_{i_m}(\cdots f^{\mathcal{I}}_{i_1}(\varepsilon^{\mathcal{I}}))) &= t^{\mathcal{I}}_{y_{i_1}\dots y_{i_m}} \neq \varepsilon^{\mathcal{I}} \end{aligned}$$

Aus $x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}$ folgt $t^{\mathcal{I}}_{x_{i_1}\dots x_{i_m}} = t^{\mathcal{I}}_{y_{i_1}\dots y_{i_m}}$,

d.h. $f^{\mathcal{I}}_{i_m}(\cdots f^{\mathcal{I}}_{i_1}(\varepsilon^{\mathcal{I}}))$ ist das gesuchte z. □

II. Komplexität

Einführung

In der Praxis genügt es nicht zu wissen, dass eine Funktion berechenbar ist. Man interessiert sich auch dafür, wie groß der *Aufwand* zur Berechnung ist.

Dabei betrachtet man verschiedene Fragestellungen in Bezug auf *Aufwand/Komplexität:*

Rechenzeit
(bei TM: Anzahl der Übergänge bis zum Halten)

Speicherplatz
(bei TM: Anzahl der benutzten Felder)

Beides soll abgeschätzt werden als *Funktion in der Größe der Eingabe.*

Man kann die Komplexität eines speziellen Algorithmus/Programms betrachten (wie in der Vorlesung „Algorithmen und Datenstrukturen“) oder die

Komplexität eines Entscheidungsproblems: Wieviel Aufwand benötigt der „beste“ Algorithmus im „schlimmsten“ Fall?

Wir betrachten hier den zweiten Fall.

7. Komplexitätsklassen

Wir führen im folgenden den Begriff der *Komplexitätsklasse* allgemein ein und definieren dann einige fundamentale *Zeit-* und *Platzkomplexitätsklassen*. Zunächst führen wir formal ein, was es heißt, dass der Aufwand durch eine Funktion der Größe der Eingabe *beschränkt* ist.

Definition 7.1 (f(n)-zeitbeschränkt, f(n)-platzbeschränkt)
Es sei $\mathrm{f} : \mathbb{N} \to \mathbb{N}$ eine Funktion und $\mathcal{A}$ eine DTM über Σ.

1) $\mathcal{A}$ heißt f(n)-*zeitbeschränkt*, falls für alle $w \in \Sigma^*$ die Maschine $\mathcal{A}$ bei Eingabe w nach $\leq \mathrm{f}(|w|)$ Schritten anhält.
2) $\mathcal{A}$ heißt f(n)-*platzbeschränkt*, falls für alle $w \in \Sigma^*$ die Maschine $\mathcal{A}$ bei Eingabe w $\leq \mathrm{f}(|w|)$ viele Felder des Bandes benutzt.

Auf *NTM* kann man die Definition dadurch übertragen, dass die Aufwandsbeschränkung für *alle* bei der gegebenen Eingabe *möglichen Berechnungen* zutreffen muss.

Wir betrachten im folgenden nicht (wie in „Algorithmen und Datenstrukturen“) die Komplexität einzelner Algorithmen (DTMs), sondern die *Komplexität von Entscheidungsproblemen*, wobei wir uns o.B.d.A. auf einstellige Relationen (d.h. $L \subseteq \Sigma^*$) beschränken.

Definition 7.2 (Komplexitätsklassen)

$$\begin{aligned}
\mathit{DTIME}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine f}(n)\text{-}\boldsymbol{zeit}\text{beschränkte } \boldsymbol{D}\text{TM, die } L \textbf{ entscheidet}\}\\
\mathit{NTIME}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine f}(n)\text{-}\boldsymbol{zeit}\text{beschränkte } \boldsymbol{N}\text{TM, die } L \textbf{ akzeptiert}\}\\
\mathit{DSPACE}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine f}(n)\text{-}\boldsymbol{platz}\text{beschränkte } \boldsymbol{D}\text{TM, die } L \textbf{ entscheidet}\}\\
\mathit{NSPACE}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine f}(n)\text{-}\boldsymbol{platz}\text{beschränkte } \boldsymbol{N}\text{TM, die } L \textbf{ akzeptiert}\}
\end{aligned}$$

Man kann einige elementare Zusammenhänge zwischen den Komplexitätsklassen feststellen:

Satz 7.3

1) $\mathit{DTIME}(\mathrm{f}(n)) \subseteq \mathit{DSPACE}(\mathrm{f}(n)) \subseteq \mathit{NSPACE}(\mathrm{f}(n))$
2) $\mathit{DTIME}(\mathrm{f}(n)) \subseteq \mathit{NTIME}(\mathrm{f}(n))$

Beweis. Offenbar kann man in k Schritten höchstens k Felder benutzen. Außerdem ist eine DTM ja auch eine NTM. □

Wir betrachten die folgenden *fundamentalen* Komplexitätsklassen:

Definition 7.4 ($P, NP, PSPACE$)

$$P := \bigcup_{\text{p Polynom in } n} DTIME(\mathrm{p}(n))$$

$$NP := \bigcup_{\text{p Polynom in } n} NTIME(\mathrm{p}(n))$$

$$PSPACE := \bigcup_{\text{p Polynom in } n} NSPACE(\mathrm{p}(n))$$

Bei der Definition von *PSPACE* hätte man genauso gut *DSPACE* statt *NSPACE* verwenden können, d.h.

$$PSPACE = \bigcup_{\text{p Polynom in } n} DSPACE(\mathrm{p}(n)),$$

denn:

Wir hatten gesehen, dass man jede NTM durch eine DTM simulieren kann. Nach einem Resultat von ***Savitch*** kann diese Simulation so gestaltet werden, dass der Platzbedarf der simulierenden DTM quadratisch im Platzbedarf der NTM ist (ohne Beweis).

Bemerkung 7.5.

Die Bedeutung der in Definition 7.4 eingeführten Komplexitätsklassen ergibt sich u.a. aus den folgenden Überlegungen:

1) Alle üblichen berechnungsuniversellen Maschinenmodelle lassen sich auf TM in Polynomialzeit simulieren, d.h.:

   Benötigt das Maschinenmodell zur Berechnung n Schritte, so benötigt die simulierende TM $\mathrm{q}(n)$ Schritte, wobei q ein Polynom ist.

   Es folgt, dass die in Definition 7.4 eingeführten Komplexitätsklassen unabhängig vom speziellen Maschinenmodell sind.

2) Probleme, die in P sind, werden i.a. als *effizient lösbar* („tractable“, d.h. „machbar“) bezeichnet. Dies liegt daran, dass Polynome im Vergleich zu Exponentialfunktionen ($n \mapsto 2^n$) ein recht moderates Wachstum haben.

   Aber:

   In Anwendungen, bei denen man für sehr große Eingaben sehr schnell (in Echtzeit) Antworten haben will, ist ein Polynom mit hohem Grad häufig auch nicht mehr tolerierbar.

3) *NP* enthält viele Probleme, für die derzeit nur deterministische Entscheidungsverfahren mit exponentiellem Zeitaufwand bekannt sind. Man bezeichnet diese Probleme daher häufig als *nicht effizient lösbar* („intractable“).

Aber:

Die eingeführten Komplexitätsklassen (wie NP) betrachten stets den schwierigsten Fall („*worst-case*“-Komplexität). Es kann durchaus sein, dass es wegen gewisser pathologischer Situationen keine polynomiale Schranke für die Rechenzeit gibt, aber für typische Fälle oder im Mittel doch polynomiales Verhalten erzielt werden kann.

Wie ist das Verhältnis zwischen den Komplexitätsklassen? Aus Satz 7.3 ergibt sich:

$$P \subseteq NP \subseteq PSPACE$$

Ob diese Inklusionen *echt* sind, ist bisher nicht bekannt. Die Frage, ob P gleich NP ist, d.h. das

$$\boldsymbol{P = NP}\textbf{ - Problem,}$$

ist eines der fundamentalen offenen Probleme der Informatik, dessen Beantwortung weitreichende Konsequenzen hätte.

8. NP-vollständige Probleme

Wegen $P \subseteq NP$ enthält die Klasse NP auch einfach lösbare Probleme, d.h. nicht alle Probleme in NP sind schwer.

Wir suchen nun nach den „schwersten“ Problemen in der Klasse NP: Es geht hier um Probleme in NP, die „mindestens so schwer“ wie jedes Problem in NP sind. Ein solches Problem ist SAT:

Definition 8.1 (SAT)
SAT ist die Menge der erfüllbaren („satisfiable“) Booleschen Ausdrücke

- mit Variablen x_i ($i \geq 1$, kodiert als Dualzahl bei Eingabe für TM)
- und Operatoren $\neg, \wedge, \vee$.

Beispiel:

- Der Ausdruck $(x_1 \wedge x_2) \vee \neg x_1$ ist erfüllbar, da er von der Belegung $x_1 \mapsto 1$, $x_2 \mapsto 1$ wahr gemacht wird.
- $x_1 \wedge \neg x_1$ ist *nicht* erfüllbar.

Lemma 8.2
$SAT \in NP$

Beweis. Eine NTM akzeptiert die Elemente von SAT wie folgt:

1) Teste, ob die Eingabe ein Boolescher Ausdruck ist (entspricht dem Wortproblem für kontextfreie Grammatiken, polynomial entscheidbar).
2) Durchlaufe den Ausdruck und ersetze *nichtdeterministisch* jede Variable x_i durch 0 oder 1 (polynomialer Aufwand).
3) Werte den Ausdruck aus und akzeptiere, falls das Ergebnis 1 ist (geht polynomial).

□

Inwiefern ist nun SAT „das schwierigste“ NP-Problem?

Definition 8.3 (polynomialzeitberechenbar,-reduzierbar, NP-vollständig)

1) Die Funktion $\mathrm{f} : \Sigma^* \to \Sigma^*$ heißt *polynomialzeitberechenbar*, falls es
   - ein Polynom p und
   - eine $\mathrm{p}(n)$-zeitbeschränkte DTM gibt,
   - die f berechnet.
2) $L \subseteq \Sigma^*$ ist *polynomialzeitreduzierbar* auf $L' \subseteq \Sigma^*$ (geschrieben als $L \leq_p L'$), falls es eine *polynomialzeitberechenbare* Funktion f gibt mit

   $$w \in L \quad \text{gdw.} \quad \mathrm{f}(w) \in L'$$

   für alle $w \in \Sigma^*$.

3) L_0 heißt *NP-vollständig*, falls gilt:
   - $L_0 \in NP$ (L_0 ist in NP)
   - Für alle $L \in NP$ gilt: $L \leq_p L_0$
     L_0 ist *NP-hart*, d.h. mindestens so hart zu lösen wie jedes andere NP-Problem.

Satz 8.4 (Cook)

1) Ist L_0 NP-vollständig, so gilt:

$$L_0 \in P \Rightarrow P = NP$$

(Es ist bisher kein NP-vollständiges L_0 bekannt, das in P ist!)

2) SAT ist NP-vollständig.

Stephen A. Cook
(geb. 14. Dezember 1939 in Buffalo, New York)
Amerikanischer Mathematiker und Informatiker
Promotion an der Harvard University 1966
Professor in Berkeley 1966-1970
Seitdem Professor an der University of Toronto
Turing-Award 1982

Beweis.

1) Es sei L_0 NP-vollständig und $L_0 \in P$, d.h. L_0 wird von einer $\mathrm{p}(n)$-zeitbeschränkten DTM (für ein Polynom p) entschieden.

   Wir müssen zeigen, dass daraus $NP \subseteq P$ folgt.

   Sei also $L \in NP$. Es ist daher $L \leq_p L_0$, d.h. es gibt ein f, das mit Zeitaufwand $\leq \mathrm{q}(n)$ berechenbar ist (für Polynom q), so dass

$$w \in L \quad \underline{\text{gdw}}. \quad \mathrm{f}(w) \in L_0.$$

   Die DTM, welche L entscheidet, geht wie folgt vor:

   - Bei Eingabe w berechnet sie $\mathrm{f}(w)$. Sie benötigt $\leq \mathrm{q}(|w|)$ viel Zeit. Daher ist auch die erzeugte Ausgabe $\leq |w| + \mathrm{q}(|w|)$.
   - Wende Entscheidungsverfahren für L_0 auf $\mathrm{f}(w)$ an.

   Insgesamt benötigt man somit

$$\leq \mathrm{q}(|w|) + \mathrm{p}(|w| + \mathrm{q}(|w|))$$

   viele Schritte, was ein Polynom in $|w|$ ist.

2) Wir beschreiben nur kurz die Idee:

Zu zeigen ist:

$$\forall L : L \in NP \Rightarrow L \leq_p SAT.$$

Sei $\mathcal{A}$ eine polynomialzeitbeschränkte NTM, welche $L \subseteq \Sigma^*$ akzeptiert. Die gesuchte Reduktionsfunktion f weist jedem Wort $w \in \Sigma^*$ einen Booleschen Ausdruck β_w zu, so dass

- Der Übergang von w zu β_w ist in Polynomialzeit berechenbar.
- $\mathcal{A}$ akzeptiert w gdw. β_w ist erfüllbar.

Die genaue Definition von β_w ist sehr aufwendig. Es muss ja die Arbeitsweise von $\mathcal{A}$ auf Eingabe w durch einen Booleschen Ausdruck beschrieben werden.

Idee:

- Aussagenvariablen $Band_{t,i,a}$ für

  „Zum Zeitpunkt t steht auf Feld i das Symbol a“.

  Da $\mathcal{A}$ polynomialzeitbeschränkt ist, muss man nur polynomial viele Bandzellen und Zeitpunkte betrachten!

- Aussagenvariablen $Kopf_{t,i}$

  „Kopf zum Zeitpunkt t auf Feld i“,

- $Zustand_{t,q}$

  „Zustand zum Zeitpunkt t ist q“.

- Übergänge: (p, a, a', r, q) liefert z.B.:

  $$Band_{t,i,a} \wedge Kopf_{t,i} \wedge Zustand_{t,p} \longrightarrow Band_{t+1,i,a'} \wedge Kopf_{t+1,i+r} \wedge Zustand_{t+1,q}.$$

- Kodierung der Eingabe w zum Zeitpunkt 0.
- Kodierungsbedingungen, z.B.

  $$\neg(Band_{t,i,a} \wedge Band_{t,i,b}) \text{ für } a \neq b$$

- etc. □

Weitere NP-vollständige Probleme erhält man durch polynomielle Reduktion.

Satz 8.5

1) Ist $L_2 \in NP$ und gilt $L_1 \leq_p L_2$, so ist auch L_1 in NP.

2) Ist L_1 NP-hart und gilt $L_1 \leq_p L_2$, so ist auch L_2 NP-hart.

Beweis.

1) Wegen $L_2 \in NP$ gibt es eine polynomialzeitbeschränkte NTM $\mathcal{A}$, welche L_2 akzeptiert.
Wegen $L_1 \leq_p L_2$ gibt es eine polynomialzeitberechenbare Funktion f mit

$$w \in L_1 \ \ \underline{\text{gdw}}. \ \ \text{f}(w) \in L_2.$$

Die polynomialzeitbeschränkte NTM für L_2 arbeitet wie folgt:

- Bei Eingabe w berechnet sie zunächst f(w).
- Dann wendet sie $\mathcal{A}$ auf f(w) an.

2) Wir müssen zeigen, dass für alle $L \in NP$ gilt:

$$L \leq_p L_2.$$

Die polynomialzeitberechenbare Reduktionsfunktion f mit

$$w \in L \ \ \underline{\text{gdw}}. \ \ \text{f}(w) \in L_2$$

erhält man wie folgt:

- Da L_1 NP-hart ist, gibt es eine polynomialzeitberechenbare Funktion g mit

$$w \in L \ \ \underline{\text{gdw}}. \ \ \text{g}(w) \in L_1.$$

- Wegen $L_1 \leq_p L_2$ gibt es eine polynomialzeitberechenbare Funktion h mit

$$u \in L_1 \ \ \underline{\text{gdw}}. \ \ \text{h}(u) \in L_2.$$

Wir definieren

$$\text{f}(w) := \text{h}(\text{g}(w)).$$

Dann gilt:

$$w \in L \ \ \underline{\text{gdw}}. \ \ \text{g}(w) \in L_1 \ \ \underline{\text{gdw}}. \ \ \text{h}(\text{g}(w)) \in L_2. \qquad \square$$

Zunächst zeigen wir, dass auch ein Teilproblem von SAT, bei dem man nur Boolesche Ausdrücke einer ganz speziellen Form zulässt, bereits NP-hart ist.

Definition 8.6 ($3SAT$)

- Eine *3-Klausel* ist von der Form

$$l_1 \vee l_2 \vee l_3,$$

wobei l_i eine Variable oder eine negierte Variable ist.

- Ein $3SAT$*-Problem* ist eine endliche Konjunktion von 3-Klauseln.
- $3SAT$ ist die Menge der erfüllbaren $3SAT$-Probleme.

Satz 8.7
$3SAT$ ist NP-vollständig.

Beweis.

1) $3SAT \in NP$ folgt unmittelbar aus $SAT \in NP$, da jedes $3SAT$-Problem ein Boolescher Ausdruck ist.

2) Es sei F ein beliebiger Boolescher Ausdruck. Wir geben ein polynomielles Verfahren an, das F in ein $3SAT$-Problem F' umwandelt, so dass gilt:

$$F \text{ erfüllbar} \quad \underline{\text{gdw}}. \quad F' \text{ erfüllbar.}$$

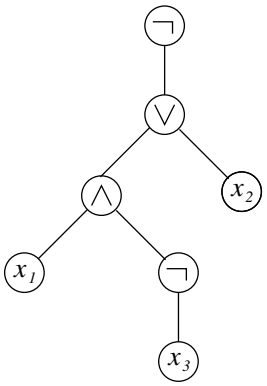
Beachte:
Es ist nicht nötig, dass F und F' (als Boolesche Ausdrücke) äquivalent sind.

Die Umformung erfolgt in mehreren Schritten, die wir am Beispiel des Ausdrucks

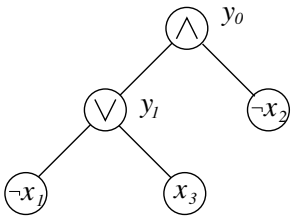
$$\neg((x_1 \wedge \neg x_3) \vee x_2)$$

veranschaulichen.

Wir stellen diesen Ausdruck als Baum dar:



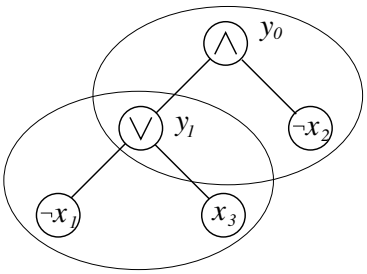
1. Schritt: Wende *de Morgan* an, um die Negationszeichen zu den Blättern zu schieben.



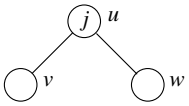
Dies erfordert nur einen Durchlauf durch den Baum, ist also in *linearer* Zeit realisierbar.

2. Schritt: Ordne jedem inneren Knoten eine neue Variable aus $\{y_0, y_1, \ldots\}$ zu, wobei die Wurzel y_0 erhält.

3. Schritt: Fasse jede Verzweigung (gedanklich) zu einer Dreiergruppe zusammen:



Jeder Verzweigung der Form



mit $j \in \{\wedge, \vee\}$ ordnen wir eine Formel der folgenden Form zu:

$$(u \Leftrightarrow (v\ j\ w))$$

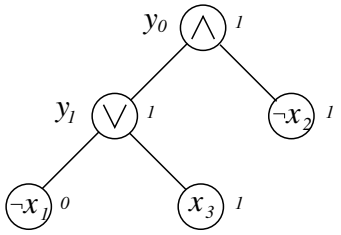
Diese Formeln werden nun konjunktiv mit y_0 verknüpft, was die Formel F_1 liefert.

Im Beispiel ist F_1 :

$$y_0 \wedge (y_0 \Leftrightarrow (y_1 \wedge \neg x_2)) \wedge (y_1 \Leftrightarrow (\neg x_1 \vee x_3))$$

Die Ausdrücke F und F_1 sind *erfüllbarkeitsäquivalent*, denn:

Eine erfüllende Belegung für F kann zu einer für F_1 erweitert werden, indem man die Werte für die Variablen y_i durch die Auswertung der entsprechenden Teilformel bestimmt, z.B.:



$$\begin{aligned} x_1 &= 1 \\ x_2 &= 0 \\ x_3 &= 1 \\ y_1 &= 1 \\ y_0 &= 1 \end{aligned}$$

Umgekehrt ist jede erfüllende Belegung für F_1 auch eine für F.

4. Schritt: Jede Konjunktion in F_1 wird separat in eine konjunktive Normalform umgeformt:

$$\begin{aligned} a \Leftrightarrow (b \vee c) &\rightsquigarrow (\neg a \vee (b \vee c)) \wedge (\neg(b \vee c) \vee a) \\ &\rightsquigarrow (\neg a \vee b \vee c) \wedge (\neg b \vee a) \wedge (\neg c \vee a) \\ &\rightsquigarrow (\neg a \vee b \vee c) \wedge (\neg b \vee a \vee a) \wedge (\neg c \vee a \vee a) \end{aligned}$$

$$\begin{aligned} a \Leftrightarrow (b \wedge c) &\rightsquigarrow (\neg a \vee (b \wedge c)) \wedge (\neg(b \wedge c) \vee a) \\ &\rightsquigarrow (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee a) \\ &\rightsquigarrow (\neg a \vee b \vee b) \wedge (\neg a \vee c \vee c) \wedge (\neg b \vee \neg c \vee a) \end{aligned}$$

Dadurch erhält man Formeln der gewünschten $3SAT$-Form.

Beachte:
Pro Teilformel ist der Aufwand hierfür konstant. □

Ein weiteres interessantes NP-vollständiges Problem ist das *Mengenüberdeckungsproblem.*

Definition 8.8 (Mengenüberdeckung)

Gegeben: Ein *Mengensystem* über einer endlichen Grundmenge M, d.h.

$$T_1, \ldots, T_k \subseteq M$$

sowie eine Zahl $n \leq k$.

Frage: Gibt es eine Auswahl von n Mengen, die ganz M überdecken, d.h.

$$\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\} \text{ mit } T_{i_1} \cup \ldots \cup T_{i_k} = M.$$

Beispiele für Anwendungen von Mengenüberdeckungen:

- Begutachtung von Projekten zu Themenbereichen aus $M = \{t_1, \ldots, t_m\}$.
- Gutachter $G_1, \ldots, G_k$ die jeweils Teilmengen $T_i \subseteq M$ der Themenbereiche abdecken.
- Ist es möglich, n Gutachter auszuwählen, die zusammen alle Themenbereiche abdecken?

Satz 8.9
Mengenüberdeckung ist NP-vollständig.

Beweis.

1) Mengenüberdeckung ist in NP:
   - Wähle nichtdeterministisch Indices $i_1, \ldots, i_n$ und
   - überprüfe, ob $T_{i_1} \cup \ldots \cup T_{i_k} = M$ gilt.
2) Um NP-Härte zu zeigen, reduzieren wir $3SAT$ auf Mengenüberdeckung.

   Sei also $F = K_1 \wedge \ldots \wedge K_m$ eine Instanz von $3SAT$, welche die Variablen $x_1, \ldots, x_n$ enthält.

   Wir definieren $M := \{1, \ldots, m, m+1, \ldots, m+n\}$.

Für jedes $i \in \{1, \ldots, n\}$ sei

$$T_i := \{j \mid x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{m + i\}$$
$$T_i' := \{j \mid \neg x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{m + i\}$$

Wir betrachten das Mengensystem:

$$T_1, \ldots, T_n, T_1', \ldots, T_n',$$

und fragen, ob es eine Überdeckung von M mit n Mengen gibt.

a) Für eine gegebene Belegung der Variablen, welche F erfüllt, wählen wir

- T_i, falls $x_i = 1$
- T_i', falls $x_i = 0$

Dies liefert n Mengen, in denen jedes Element von M vorkommt:

- j mit $1 \leq j \leq m$, da jedes K_j zu 1 evaluiert wird.
- $m + i$ mit $1 \leq i \leq n$, da für jedes i entweder T_i oder T_i' gewählt wird.

b) Sei umgekehrt

$$\{U_1, \ldots, U_n\} \subseteq \{T_1, \ldots, T_n, T_1', \ldots, T_n'\} \text{ mit } U_1 \cup \ldots \cup U_n = M$$

gegeben.

Da für $1 \leq i \leq n$ das Element $m + i$ in $U_1 \cup \ldots \cup U_n$ vorkommt, kommt T_i oder T_i' in $\{U_1, \ldots, U_n\}$ vor. Da wir nur n verschiedene Mengen haben, kommt jeweils genau eines von beiden vor.

O.B.d.A. sei $U_i \in \{T_i, T_i'\}$ für $i = 1, \ldots, n$.

Wir definieren nun die Belegung:

$$x_i = \begin{cases} 1 & \text{falls } U_i = T_i \\ 0 & \text{falls } U_i \neq T_i \end{cases}$$

Diese erfüllt jedes K_j, da j in $U_1 \cup \ldots \cup U_n$ vorkommt. □

Viele Probleme für Graphen sind NP-vollständig, z.B. auch das folgende:

Definition 8.10 (Clique)

Gegeben: Ein gerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$

Frage: Besitzt G eine k-*Clique*, d.h. eine Teilmenge $U \subseteq V$ mit

- $|U| = k$ und
- für alle $u \neq v$ in U gilt: $(u, v) \in E$.

Mögliche Anwendung für Clique:

- Gutachter $V = \{G_1, \ldots, G_n\}$ sowie Information darüber, welche Gutachter welchen anderen Gutachter kennt (E).
- Ist es möglich, k Gutachter so auszuwählen, dass jeder ausgewählte Gutachter jeden anderen ausgewählten Gutachter kennt?

Satz 8.11

Clique ist NP-vollständig.

Beweis.

1) Clique ist in NP:
   - Rate eine Teilmenge der Größe k und
   - teste, ob sie eine k-Clique ist.
2) Clique ist NP-hart:

   Wir reduzieren $3SAT$ auf Clique.

   Sei also
   $$F = \overbrace{(l_{11} \vee l_{12} \vee l_{13})}^{K_1} \wedge \ldots \wedge \overbrace{(l_{m1} \vee l_{m2} \vee l_{m3})}^{K_m}$$
   mit $l_{ij} \in \{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$.

   Der Graph G wird nun wie folgt definiert:
   - $V := \{(i,j) \mid 1 \leq i \leq m \text{ und } 1 \leq j \leq 3\}$
   - $E := \{((i,j),(p,q)) \mid i \neq p \text{ und } l_{ij} \neq \bar{l}_{pq}\}$, wobei
     $$\bar{l} = \begin{cases} \neg x & \text{falls } \; l = x \\ x & \text{falls } \; l = \neg x \end{cases} .$$
   - $k = m$

   Es gilt nun:

   F ist erfüllbar mit einer Belegung B

   gdw. es gibt in jeder Klausel K_i ein Literal l_{ij_i} mit Wert 1

   gdw. es gibt Literale $l_{1j_1}, \ldots, l_{mj_m}$, die paarweise nicht komplementär sind

   gdw. es gibt Knoten $(1, j_1), \ldots, (m, j_m)$, die paarweise miteinander verbunden sind

   gdw. es gibt eine k-Clique in G

□

Färbung von Graphen

Für einen ungerichteten Graphen und eine Menge von k Farben stellt sich die folgende Frage:

> „Können die Knoten des Graphen so mit diesen Farben eingefärbt werden, dass benachbarte Knoten stets verschiedene Farben haben?“

Beispiel

Wir betrachten Prüfungsplanung graphentheoretisch wie folgt:

- Knoten: Klausuren
- Kanten: besagen, dass sich Studenten für beide Klausuren angemeldet haben
- Farben: Tage des Prüfungszeitraums, an denen Prüfungen stattfinden können

Das Ziel besteht nun darin, Tage für die Klausuren zu finden, so dass kein Student zwei Klausuren am selben Tag schreiben muss.

Definition 8.12

Der ungerichtete Graph $G = (V, E)$ heißt *k-färbbar*, falls es eine Funktion $f\colon V \to \{1, \ldots, k\}$ gibt mit der Eigenschaft:

$$\{u, v\} \in E \Longrightarrow f(u) \neq f(v).$$

Die *chromatische Zahl* eines ungerichteten Graphen G ist die minimale Anzahl k, für die G k-färbbar ist.

Offenbar ist die chromatische Zahl von $G = (V, E)$ stets $\leq |V|$. Im Beispiel der Prüfungsplanung ist die chromatische Zahl die minimale Anzahl der Prüfungstage.

Der vollständige Graph K_5 hat chromatische Zahl 5, da jeder Knoten jeden anderen als Nachbarn hat. Der vollständige bipartite Graph $K_{3,3}$ hat chromatische Zahl 2.

Graphen $G = (V, E)$ mit chromatischer Zahl 2 heißen auch *bipartite Graphen*. Bei bipartiten Graphen kann die Knotenmenge V in zwei disjunkte Teile $V = V_1 \cup V_2$ zerlegt werden, so dass Knoten aus V_1 nur Knoten aus V_2 als Nachbarn haben (und umgekehrt).

Der Kreisgraph C_6 hat chromatische Zahl 2, und der Kreisgraph C_5 hat chromatische Zahl 3. Allgemein gilt: Kreise gerader Länge haben chromatische Zahl 2 und Kreise ungerader Länge haben chromatische Zahl 3.

2-Färbbarkeit von Graphen

Satz 8.13

Ein zusammenhängender ungerichteter Graph ist genau dann bipartit, wenn er keinen Kreis ungerader Länge als Teilgraphen enthält.

Beweisidee. Betrachte einen beliebigen Knoten v_0 in G und bestimme für jeden Knoten v den Abstand zwischen v_0 und v (d.h. die Länge eines kürzesten Wegs von v_0 nach v).

Knoten mit geradem Abstand werden mit 1 gefärbt und Knoten mit ungeradem Abstand mit 2. Die so erhaltene Funktion $f\colon V \to \{1,2\}$ erfüllt die Bedingung

$$\{u,v\} \in E \Longrightarrow f(u) \neq f(v),$$

falls es keine Kreise ungerader Länge in G gibt. □

Da kürzeste Wege in ungerichteten Graphen in polynomieller Zeit berechnet werden können, liefert der Beweis des Satzes einen polynomiellen Algorithmus zum Testen von 2-Färbbarkeit:

Wende das im Beweis beschriebene Färbungsverfahren auf G an.
Wenn es erfolgreich ist, so ist der Graph 2-färbbar und ansonsten nicht.

Satz 8.14

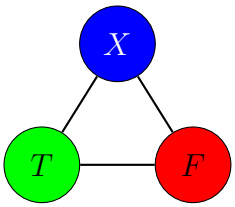
Zweifärbbarkeit von ungerichteten Graphen ist in P.

Für $k \geq 3$ ist kein effizienter Algorithmus für den Test auf k-Färbbarkeit bekannt.

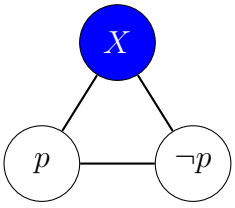
Satz 8.15

3-Färbbarkeit von ungerichteten Graphen ist NP-vollständig.

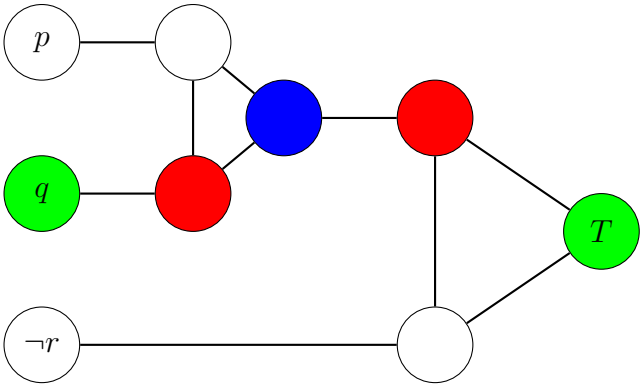
Beweisskizze. Für NP-Härte konstruiere eine Reduktion von 3SAT auf 3-Färbbarkeit. Gegeben sei ein 3SAT-Problem ϕ. Wir übersetzen dieses in einen Graphen.



Dieser Teilgraph stellt sicher, dass der Knoten T eine andere Farbe hat als der Knoten F.
Die Farbe von T (hier grün) steht dann für „wahr“ (true)
und die von F (hier rot) für „falsch“.



Dieser Teilgraph stellt sicher, dass p und $\neg p$ verschiedene Farben erhalten,
wobei eine die von T („wahr“) und die andere die von F („falsch“) ist.



Kodierung der 3-Klausel $p \vee q \vee \neg r$
Der Graph stellt sicher, dass eines der Literale wahr wird.

Setzt man diese Teile zu einem Graphen zusammen, wobei

- die Knoten mit Namen (T, F, X, p, $\neg p$, ...) nur einmal auftreten,
- die Knoten ohne Namen alle verschieden sind,

so erhält man einen Graphen, der genau dann 3-färbbar ist, wenn das 3SAT-Problem ϕ erfüllbar ist. □

Das Vier-Farben-Problem

Ist es stets möglich, die Länder auf einer Landkarte so mit maximal vier verschiedenen Farben einzufärben, dass aneinandergrenzende Länder stets verschieden gefärbt sind?

Im Jahr 1852 stellte Francis Guthrie seinem Bruder Frederick in einem Brief diese Frage. Frederick gab diese Frage an einen seiner Professoren, den berühmten Mathematiker Augustus de Morgan weiter.

Fasst man die Länder als Knoten auf und beschreibt die Nachbarschaft durch Kanten, so liefert eine Landkarte einen planaren[1] Graphen.

Graphen-theoretisch formuliert fragt also das Vier-Farben-Problem, ob jeder planare Graph eine chromatische Zahl ≤ 4 hat.

4-Färbung von planaren Graphen

Satz (Appel, Haken, 1976)
Jeder planare Graph ist 4-färbbar.

Der Beweis ist ungewöhnlich da Appel und Haken 1476 Fälle mit Hilfe eines Computerprogramms untersuchten, was ca. 1200 Stunden Rechenzeit benötigte. Viele Mathematiker fanden den Beweis daher unelegant und zweifelten auch dessen Korrektheit an, da die Vielzahl von Fällen nicht manuell überprüft werden konnte und die Korrektheit des Programms nicht formal verifiziert war.

Im Jahr 2004 wurde schließlich ein vereinfachter Beweis (von Robertson, Sanders, Seymour, und Thomas; 1995) inklusive der verwendeten Programme mit einem interaktiven Theorembeweiser formal verifiziert.

Für planare Graphen ist damit k-Färbbarkeit für $k \leq 2$ und $k \geq 4$ effizient entscheidbar. Für $k = 3$ ist auch für planare Graphen kein effizienter Algorithmus für den Test auf k-Färbbarkeit bekannt.

[1] In Ebene überkreuzungsfrei darstellbar.

9. PSPACE-vollständige Probleme

Zur Erinnerung:

$$\begin{aligned} PSPACE &:= \bigcup_{p \text{ Polynom in } n} NSPACE(p(n)) \\ &= \bigcup_{p \text{ Polynom in } n} DSPACE(p(n)) \qquad \text{nach Savitch} \end{aligned}$$

Offenbar kann man aus einer polynomialplatzbeschränkten *DTM, die L entscheidet*, sehr leicht eine polynomialplatzbeschränkte *DTM, die* $\overline{L}$ *entscheidet*, konstruieren.

Dies zeigt, dass $PSPACE$ unter Komplement abgeschlossen ist, d.h.

$$L \in PSPACE \quad \text{gdw} \quad \overline{L} \in PSPACE.$$

Entsprechend zur Definition von NP-vollständig kann man auch *PSPACE-vollständig* definieren.

Definition 9.1 (PSPACE-vollständig)
Die Sprache L_0 heißt PSPACE-*vollständig*, falls gilt:

- $L_0 \in$ PSPACE (L_0 ist in PSPACE)
- Für alle $L \in$ PSPACE gilt: $L \leq_p L_0$
  (L_0 ist PSPACE-hart, d.h. mindestens so hart zu lösen wie jedes andere PSPACE-Problem.)

Gibt es überhaupt PSPACE-vollständige Probleme?

Wir führen dazu eine Erweiterung des aussagenlogischen Erfüllbarkeitsproblems ein.

Definition 9.2 (QBF)
Eine *quantifizierte Boole'sche Formel (quantified Boolean formula, QBF)* ist von der Form

$$\psi = Q_1 p_1 . \ldots Q_n p_n . \phi$$

wobei $Q_i \in \{\exists, \forall\}$ Quantoren sind, $p_1, \ldots, p_n \in \mathcal{P}$ Aussagenvariablen und ϕ eine aussagenlogische Formel. Diese QBF heißt *geschlossen*, falls $Var(\phi) \subseteq \{p_1, \ldots, p_n\}$ gilt.

Beispiel

$$\forall p_1 . \exists p_2 . \exists p_3 . (p_1 \rightarrow (p_2 \wedge p_3))$$

Idee: die Quantoren quantifizieren über die Wahrheitswerte.

Im Beispiel muss man daher beide möglichen Werte für p_1 (0 und 1) betrachten, abhängig vom jeweils gewählten Wert für p_1 aber nur einen für p_2 (0 oder 1) finden, etc.

Für eine Wertzuweisung $w : \mathcal{P} \to \{0, 1\}$, eine Variable $p \in \mathcal{P}$ und einen Wahrheitswert $b \in \{0, 1\}$ bezeichne $w[p/b]$ die Wertzuweisung, die

- auf allen Variablen in $\mathcal{P} \setminus \{p\}$ mit w übereinstimmt
- und für die $w(p) = b$ gilt.

Definition 9.3 (Semantik von QBF)
Es sei $w : \mathcal{P} \to \{0, 1\}$ eine Wertzuweisung und

$$\psi = Q_1 p_1 . \ldots Q_n p_n . \phi$$

eine QBF. Der *Wahrheitswert* $\hat{w}(\psi)$ der QBF ψ unter der Wertzuweisung w ist induktiv definiert:

- Ist $n = 0$, so ist $\psi = \phi$ eine aussagenlogische Formel und wir definieren $\hat{w}(\psi) := w(\phi)$.
- Ist $n > 0$, so ist $\psi = Q_1 p_1 . \psi'$ für eine QBF ψ' mit $n - 1$ Quantoren.
  - Ist $Q_1 = \exists$, so definieren wir $\hat{w}(\psi) := \widehat{w[p_1/0]}(\psi') \vee_S \widehat{w[p_1/1]}(\psi')$
  - Ist $Q_1 = \forall$, so definieren wir $\hat{w}(\psi) := \widehat{w[p_1/0]}(\psi') \wedge_S \widehat{w[p_1/1]}(\psi')$

Definition 9.4 (Erfüllbarkeit, Allgemeingültigkeit)
Es sei ψ eine QBF.

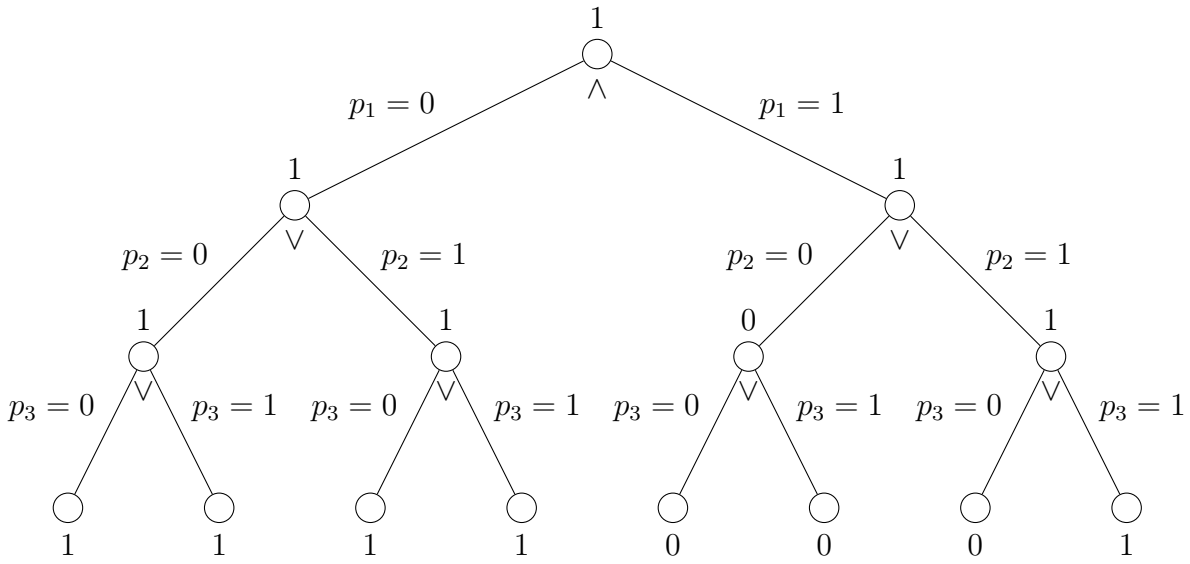
- Die QBF ψ heißt *erfüllbar* gdw. eine Wertzuweisung w existiert mit $\hat{w}(\psi) = 1$.
- Die QBF ψ heißt *allgemeingültig* gdw. für jede Wertzuweisung w gilt: $\hat{w}(\psi) = 1$.

Bemerkung 9.5.

- Ist $\psi = Q_1 p_1 . \ldots Q_n p_n . \phi$ und v, w Wertzuweisungen, die auf den Variablen $p_1, \ldots, p_n$ übereinstimmen, so gilt $\hat{v}(\psi) = \hat{w}(\psi)$.
- Ist $\psi = Q_1 p_1 . \ldots Q_n p_n . \phi$ geschlossen, so gilt: ψ erfüllbar gdw. ψ allgemeingültig.
- Ist $\psi = Q_1 p_1 . \ldots Q_n p_n . \phi$ und gilt $Var(\phi) \setminus \{p_1, \ldots, p_n\} = \{q_1, \ldots, q_k\}$, so ist ψ erfüllbar gdw. $\exists q_1 . \ldots \exists q_k . \psi$ erfüllbar ist.

Beispiel 9.6
$\forall p_1 . \exists p_2 . \exists p_3 . (p_1 \to (p_2 \wedge p_3))$ ist erfüllbar und allgemeingültig

Betrachte *Auswertungsbaum*:



Definition 9.7 (QBFSAT)
QBFSAT ist die Menge der *erfüllbaren quantifizierten Boole'schen Formeln.*

Lemma 9.8
QBFSAT $\in$ PSPACE

Beweisskizze. Der Auswertungsbaum ist zwar exponentiell groß, es genügt aber, zu jedem Zeitpunkt nur einen Ast des Baums mit den bisher erzielten Auswertungen im Speicher zu halten. (Platzbedarf linear in der Anzahl der Variablen.) □

Lemma 9.9
QBFSAT *ist* PSPACE-*hart.*

Beweis. Wir müssen zeigen:

$$\forall L : L \in \text{PSPACE} \Rightarrow L \leq_p \text{QBFSAT}$$

Sei $\mathcal{A}$ eine polynomialplatzbeschränkte NTM, welche $L \subseteq \Sigma^*$ akzeptiert.

Die gesuchte Reduktionsfunktion f weist jedem Wort $w \in \Sigma^*$ eine QBF ψ_w zu, so dass

- Der Übergang von w zu ψ_w ist in Polynomialzeit berechenbar.
- $\mathcal{A}$ akzeptiert w gdw ψ_w ist erfüllbar.

Da $\mathcal{A}$ $p(n)$-platzbeschränkt ist für ein Polynom p, muss man nur ein Band dieser (polynomiellen) Länge repräsentieren.

Im Folgenden bezeichne n die Länge von w.

Aussagenvariablen:

- X_q für alle Zustände $q \in Q$

  „$\mathcal{A}$ befindet sich im Zustand q“

- $X'_{a,i}$ für alle Bandsymbole $a \in \Gamma$ und alle $i \leq p(n)$

  „Im Feld i steht das Symbol a“

- X''_j für alle $j \leq p(n)$

  „Das Feld j ist das Arbeitsfeld“

$\overline{X}$ bezeichne das Tupel dieser Variablen.

Beachte:

- Im Unterschied zu der Reduktion auf SAT im Abschnitt §8 werden die Zeitpunkte nicht durch die Aussagenvariablen kodiert.
- Dies würde zu einer exponentiellen Reduktionsfunktion führen, da eine polynomialplatzbeschränkte NTM exponentiell lange Berechnungen durchführen kann.

Formeln:

- $\mathrm{Conf}(\overline{X})$: die Wertzuweisung für $\overline{X}$ kodiert eine Konfiguration von $\mathcal{A}$

  $$\bigvee_{q \in Q} \left(X_q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg X_{q'} \right) \wedge$$ genau ein Zustand

  $$\bigwedge_{i \leq p(n)} \left(\bigvee_{a \in \Gamma} \left(X'_{a,i} \wedge \bigwedge_{b \in \Gamma \setminus \{a\}} \neg X'_{b,i} \right) \right) \wedge$$ genau ein Symbol pro Feld

  $$\bigvee_{j \leq p(n)} \left(X''_j \wedge \bigwedge_{\substack{i \neq j \\ i \leq p(n)}} \neg X''_i \right)$$ Kopf auf genau einem Feld

- $\mathrm{Input}_w(\overline{X})$: die Wertzuweisung für $\overline{X}$ kodiert die Startkonfiguration bei Eingabe $w = a_1 \dots a_n$

  $$\mathrm{Conf}(\overline{X}) \wedge X_{q_0} \wedge X''_1 \wedge \bigwedge_{1 \leq i \leq n} X'_{a_i,i} \wedge \bigwedge_{n+1 \leq j \leq p(n)} X'_{\not b,j}$$

- $\mathrm{Acc}(\overline{X})$: die Wertzuweisung für $\overline{X}$ kodiert eine akzeptierende Stoppkonfiguration

  $$\mathrm{Conf}(\overline{X}) \wedge \bigvee_{\substack{1 \leq j \leq p(n) \\ (q,a) \in Stopp}} X''_j \wedge X'_{a,j} \wedge X_q$$

  wobei $Stopp := \{(q,a) \mid \text{es gibt keine Transition } (q,a,\dots)\}$

- Next($\overline{X}, \overline{Y}$): die Wertzuweisung für $\overline{Y}$ kodiert eine Folgekonfiguration der durch die Wertzuweisung für $\overline{X}$ kodierten Konfiguration

  Dabei wird angenommen, dass $\overline{Y}$ ein Tupel von Variablen ist, welches das Tupel $\overline{X}$ umbenennt (d.h. $Y_q, Y'_{a,i}, Y''_j$).

  Man kann dann z.B. auch Conf($\overline{Y}$) schreiben für die Formel, bei der alle X-Variablen durch die entsprechenden Y-Variablen ersetzt sind.

  Next($\overline{X}, \overline{Y}$):

$$\begin{aligned}&\mathrm{Conf}(\overline{X}) \wedge \mathrm{Conf}(\overline{Y}) \wedge \\ &\bigwedge_{1 \le j \le p(n)} \left(X''_j \rightarrow \left((\bigwedge_{i \neq j} Y'_{a,i} \leftrightarrow X'_{a,i}) \wedge \right. \right. \\ &\qquad \left. \left. \bigwedge_{\substack{q \in Q \\ a \in \Gamma}} (X_q \wedge X'_{a,j}) \rightarrow \bigvee_{(q,a,b,m,q') \in \Delta)} (Y_{q'} \wedge Y'_{b,j} \wedge Y''_{f(m,j)}) \right) \right)\end{aligned}$$

  wobei $f(m, j)$ das neue Arbeitsfeld bestimmt.

- Eq($\overline{X}, \overline{Y}$): Die Wertzuweisung für $\overline{Y}$ kodiert dieselbe Konfiguration wie die Wertzuweisung für $\overline{X}$

$$\begin{aligned}&\mathrm{Conf}(\overline{X}) \wedge \mathrm{Conf}(\overline{Y}) \wedge \\ &\bigwedge_{q \in Q} (X_q \leftrightarrow Y_q) \wedge \bigwedge_{\substack{1 \le i \le p(n) \\ a \in \Gamma}} (X'_{a,i} \leftrightarrow Y'_{a,i}) \wedge \bigwedge_{1 \le j \le p(n)} (X''_j \leftrightarrow Y''_j)\end{aligned}$$

Alle diese Formeln können (bei fest vorgegebener NTM $\mathcal{A}$) in polynomieller Zeit aus w konstruiert werden.

Wir wollen nun eine Formel $Reach_m(\overline{X}, \overline{Y})$ definieren, die ausdrückt: die durch die Wertzuweisung für $\overline{X}$ kodierte Konfiguration hat die durch die Wertzuweisung für $\overline{Y}$ kodierte Konfiguration als Folgekonfiguration in $\le 2^m$ Schritten.

$Reach_0(\overline{X}, \overline{Y})$:

$$Eq(\overline{X}, \overline{Y}) \vee Next(\overline{X}, \overline{Y})$$

$Reach_m(\overline{X}, \overline{Y})$ für $m > 0$:

$$\exists \overline{Z}.(Reach_{m-1}(\overline{X}, \overline{Z}) \wedge Reach_{m-1}(\overline{Z}, \overline{Y}))$$

Dies ist zwar eine korrekte Definition für $Reach_m(\overline{X}, \overline{Y})$, würde aber leider zu einer exponentiellen Reduktionsfunktion führen, da *Reach* dupliziert wird:

$$\left|Reach_m(\overline{X},\overline{Y})\right| = 2 \cdot \left|Reach_{m-1}(\overline{X},\overline{Y})\right| \qquad \longrightarrow \qquad \left|Reach_m(\overline{X},\overline{Y})\right| \geq 2^m$$

$Reach_m(\overline{X},\overline{Y})$ für $m > 0$: durch die Verwendung universeller Quantoren kann man das exponentielle Wachstum vermeiden.

$$\begin{aligned}\exists\overline{Z}.\exists\overline{U}.\exists\overline{V}.((Eq(\overline{X},\overline{U}) \wedge Eq(\overline{Z},\overline{V}))\vee\\ (Eq(\overline{Z},\overline{U} \wedge Eq(\overline{Y},\overline{V})))\\ \rightarrow Reach_{m-1}(\overline{U},\overline{V})\end{aligned}$$

$(Eq(\overline{X},\overline{U}) \wedge Eq(\overline{Z},\overline{V}))$ liefert $Reach_{m-1}(\overline{X},\overline{Z})$
und $(Eq(\overline{Z},\overline{U}) \wedge Eq(\overline{Y},\overline{V}))$ liefert $Reach_{m-1}(\overline{Z},\overline{Y})$

$$\begin{aligned}&|Reach_0(\overline{X},\overline{Y})| = O(p(n))\\ &|Reach_m(\overline{X},\overline{Y})| = |Reach_{m-1}(\overline{X},\overline{Y})| + O(p(n))\end{aligned} \qquad \longrightarrow \qquad |Reach_m(\overline{X},\overline{Y})| = O(m \cdot p(n))$$

Da $\mathcal{A}$ $p(n)$-platzbeschränkt und n die Länge der Eingabe w ist, benötigt $\mathcal{A}$ zur Akzeptanz der Eingabe w nur $2^{O(p(n))}$ viel Zeit, d.h. Zeit $2^{c \cdot p(n)}$ für eine Konstante c.

Wir setzen daher $m := c \cdot p(n)$ und definieren

$$\psi_w := \exists\overline{X}.\exists\overline{Y}.(Input_w(\overline{X}) \wedge Acc(\overline{Y}) \wedge Reach_m(\overline{X},\overline{Y}))$$

Man sieht nun leicht:

- Der Übergang von w zu ψ_w ist in Polynomialzeit berechenbar.
- $\mathcal{A}$ akzeptiert w gdw. ψ_w ist erfüllbar.

Beachte:
Streng genommen ist ψ_w keine QBF wie in Definition 9.2 eingeführt, da die Quantoren nicht alle als Präfix am Anfang stehen. Man kann die Quantoren aber nach außen ziehen ohne die Semantik zu verändern. (siehe später bei Prädikatenlogik) □

Satz 9.10
QBFSAT *ist* PSPACE-*vollständig.*

Auch im Bereich *Formale Sprachen* gibt es PSPACE-vollständige Probleme.

Definition (Das Universalitätsproblem)

Gegeben: Ein nicht-deterministischer endlicher Automat $\mathcal{A}$ über dem Alphabet Σ.

Frage: Gilt $L(\mathcal{A}) = \Sigma^*$?

Satz 9.11
Das Universalitätsproblem für nicht-deterministische endliche Automaten ist PSPACE-*vollständig.*

Beweis. Anstelle eines formalen Beweises beschreiben wir nur die wesentlichen *Beweisideen*.

In PSPACE

Das Universalitätsproblem kann auf das Leerheitsproblem reduziert werden:

$$L(\mathcal{A}) = \Sigma^* \quad \text{gdw} \quad \overline{L(\mathcal{A})} = \emptyset$$

Einen (deterministischen) endlichen Automaten für $\overline{L(\mathcal{A})}$ erhält man durch Potenzmengenkonstruktion.

Der DEA $\mathcal{A}' = (2^Q, \Sigma, \{q_0\}, \delta, F')$ mit $L(\mathcal{A}) = L(\mathcal{A}')$ ist definiert durch:

- $\delta(P, a) = \bigcup_{p \in P} \{p' \mid (p, a, p') \in \Delta\}$ für alle $P \in 2^Q$ und $a \in \Sigma$
- $F' = \{P \in 2^Q \mid P \cap F \neq \emptyset\}$

Daraus erhält man den DEA $\overline{\mathcal{A}} = (2^Q, \Sigma, \{q_0\}, \delta, F'')$ mit $L(\overline{\mathcal{A}}) = \overline{L(\mathcal{A})}$ durch Vertauschen von Endzuständen mit Nichtendzuständen:

- $\delta(P, a) = \bigcup_{p \in P} \{p' \mid (p, a, p') \in \Delta\}$ für alle $P \in 2^Q$ und $a \in \Sigma$
- $F'' = \{P \in 2^Q \mid P \cap F = \emptyset\}$

Es gilt nun:

$$L(\mathcal{A}) = \Sigma^* \quad \text{gdw} \quad \overline{L(\mathcal{A})} = \emptyset \quad \text{gdw} \quad L(\overline{\mathcal{A}}) = \emptyset$$

Da PSPACE unter Komplement abgeschlossen ist, genügt es, eine polynomialplatzbeschränkte NTM zu konstruieren, die das Komplementproblem $L(\overline{\mathcal{A}}) \neq \emptyset$ akzeptiert:

1. Beginne mit $P := \{q_0\}$.
2. Rate ein Symbol $a \in \Sigma$ und berechne $P' := \delta(P, a)$.
3. Teste, ob $P' \cap F = \emptyset$.
   - Wenn ja, dann akzeptiere.
   - Wenn nein, dann setze $P := P'$ und mache weiter bei 2.

Im Prinzip muss man also stets nur den aktuellen Zustand von $\overline{\mathcal{A}}$ im Speicher behalten, was zeigt, dass man mit polynomiell viel Platz auskommt.

PSPACE-Härte

Wir reduzieren $L \in$ PSPACE auf das Universalitätsproblem, indem wir für jedes Wort $w \in \Sigma^*$ einen NEA $\mathcal{A}_w$ konstruieren, so dass gilt:

- Der Übergang von w zu $\mathcal{A}_w$ ist in Polynomialzeit berechenbar.
- $w \in L$ gdw $L(\mathcal{A}_w) = \Sigma^*$.

Wir betrachten dazu eine polynomialbeschränkte NTM $\mathcal{N}$, die $\overline{L}$ akzeptiert und konstruieren die NEAs $\mathcal{A}_w$ so, dass gilt:

- $\mathcal{N}$ akzeptiert w nicht gdw $L(\mathcal{A}_w) = \Sigma^*$.

Idee: Akzeptierende $\mathcal{N}$-Berechnungen für die Eingabe w

$$k_0 \vdash_{\mathcal{N}} k_1 \vdash_{\mathcal{N}} k_2 \vdash_{\mathcal{N}} \ldots k_n$$

können als Wörter

$$\#k_0\#k_1\#k_2\#\ldots\#k_n\#$$

über einem geeigneten Alphabet Σ dargestellt werden.

Wir nennen diese Wörter akzeptierende Konfigurationswörter für w.

Offenbar gilt dann:

- $\mathcal{N}$ akzeptiert w nicht gdw es gibt kein akzeptierendes Konfigurationswort für w.

Wir konstruieren daher $\mathcal{A}_w$ so, dass gilt:

- $\mathcal{A}_w$ akzeptiert genau die Wörter in Σ^*, die nicht akzeptierende Konfigurationswörter für w sind.

Wir beschreiben dazu durch "kleine" Automaten alle Fälle, die verhindern, dass ein Wort über Σ ein akzeptierendes Konfigurationswort für w ist:

- Zwischen zwei #-Symbolen steht nicht genau ein Zustand.
- Der erste Zustand ist nicht der Anfangszustand.
- Symbole in aufeinanderfolgenden Konfigurationen, die nicht durch eine Transition verändert wurden, sind trotzdem verschieden.
- usw. *Haben nur polynomiellen Abstand voneinander!*

$\mathcal{A}_w$ erhält man als Vereinigung dieser kleinen Automaten.

□

Abkürzungsverzeichnis

bzw. ..... beziehungsweise
DEA ..... deterministischer endlicher Automat
d.h. ..... das heißt
DTM ..... deterministische Turingmaschine
EBNF ..... erweiterte Backus-Naur-Form
etc. ..... et cetera
gdw. ..... genau dann wenn
geg. ..... gegeben
i.a. ..... im allgemeinen
LBA ..... linear beschränkter Automat
MPKP ..... modifiziertes Postsches Korrespondenzproblem
NEA ..... nichtdeterministischer endlicher Automat
NP ..... nichtdeterministisch polynomiell
NTM ..... nichtdeterministische Turingmaschine
o.B.d.A. ..... ohne Beschränkung der Allgemeingültigkeit
PDA ..... pushdown automaton (Kellerautomat)
PKP ..... Postsches Korrespondenzproblem
PL1 ..... Prädikatenlogik erster Stufe
SAT ..... satisfiability problem (Erfüllbarkeitstest der Aussagenlogik)
TM ..... Turingmaschine (allgemein)
u.a. ..... unter anderem
URM ..... unbeschränkte Registermaschine
vgl. ..... vergleiche
z.B. ..... zum Beispiel
□ ..... was zu beweisen war (q.e.d)

Literaturverzeichnis

[Skript] Skript zur Vorlesung und Folien (siehe Web-page).

[SkriptFS] Skript zur Vorlesung „Formale Systeme“

[Schö_01] Uwe Schöning, *Theoretische Informatik – kurzgefaßt.* Spektrum Akademischer Verlag, 2001.

[Wege_99] Ingo Wegener, *Theoretische Informatik – eine algorithmenorientierte Einführung*, Teubner, 1999.

[Schö_00] Uwe Schöning, *Logik für Informatiker*, Spektrum akademischer Verlag, 2000.