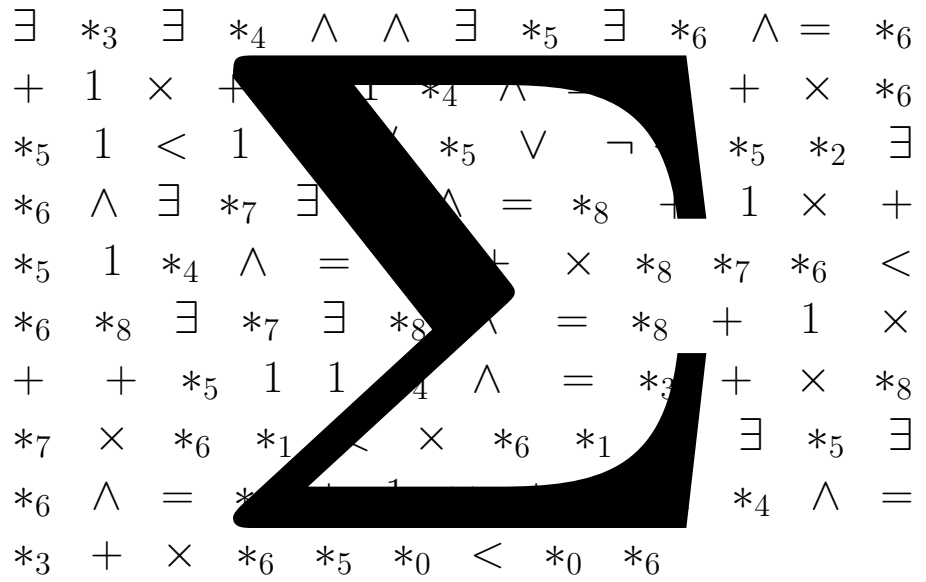
Define & Conquer



1 Aus der Kursbeschreibung

Denn eben wo Begriffe fehlen,
Da stellt *ein Wort* zur rechten Zeit sich ein.
Mit Worten läßt sich trefflich streiten,
Mit Worten ein System bereiten,
An Worte läßt sich trefflich glauben,
Von einem Wort läßt sich kein Jota rauben.
(Goethe, Faust)

Dass es oft gar nicht so einfach ist, das richtige Wort zu finden, ist das zentrale Thema des Kurses. Genauer: es ist meist sogar gar nicht möglich, überhaupt ein Wort zu finden. Gegenstand des Kurses sind die Mengen natürlicher Zahlen – und von denen gibt es mehr als Worte. Die meisten dieser Mengen werden also nicht definiert werden können. Aber unter denen, die definiert werden können, gibt es viel Interessantes. Es ist zu fragen, wie kompliziert die einfachst mögliche Definition einer solchen definierbaren Menge ist.

Diese Charakterisierung von Mengen nach der logischen Komplexität ist mehr als eine Spielerei: Diejenigen Mengen, die sich durch reine »es gibt«-Aussagen beschreiben lassen, sind genau die, für die es ein Programm gibt, das nach und nach alle Elemente ausgibt. Wenn zusätzlich noch eine Beschreibung durch eine »für alle«-Aussage hinzukommt, dann kann man sogar eindeutig sagen, ob eine Zahl dazu gehört, oder nicht.

Was aber ist mit Mengen, die sich nur durch kompliziertere Formeln beschreiben lassen? Da gibt es eben kein Verfahren, das sagt, ob eine Zahl dazu gehört. Im Kurs wird untersucht, in welchem Maße die Zugehörigkeit zur Menge durch ein Programm beantwortet werden kann. Dazu muss zunächst überlegt werden, was es bedeutet, dass ein Problem »noch unlösbarer« ist als ein anderes. Ein Problem soll dabei als »leichter« gelten, wenn es mit Kenntnis der Lösung des schwereren Problems gelöst werden kann. Es stellt sich heraus, dass es zu jeder noch so komplizierten Menge eine kompliziertere gibt, nämlich die Antwort auf die Frage, ob ein Programm, das Kenntnis der ersten Menge hat, bei gegebenen Eingaben anhält. Damit erhält man eine Hierarchie von immer komplizierteren Mengen, die aber noch durch Formeln beschrieben werden können. Die Stufen dieser Hierarchie spiegeln genau die logische Komplexität der zur Beschreibung nötigen Formeln wieder.

Es steckt also sehr viel nicht-triviale Theorie hinter der unscheinbaren Frage »Kannst du mir das Problem beschreiben?«. Der Kurs wird auf diese Weise die Grundlagen der Informatik lernen und viel Spaß dabei haben, verschiedene Probleme »richtig« zu definieren . . .

2 Einleitung

Der Kurstitel »Define & Conquer« (Definiere und Erobere) ist angelehnt an Cäsars Ausspruch »divide et impera« (Teile und Herrsche). Er steht für das Prinzip, die eigenen Gegner oder Untertanen gegeneinander auszuspielen und ihre Uneinigkeit für die eigene Machtausübung zu verwenden. Auf die theoretische Informatik übertragen bedeutet dies, größere Aufgaben in mehrere Teilprobleme zu zerlegen, diese einzeln zu lösen und die Ergebnisse zur Gesamtlösung zusammenzufassen.

Für die Teilnehmer des Kurses meint »Define & Conquer« die Notwendigkeit, Problemstellungen genau zu beschreiben (zu definieren), um diese berechnen zu können. Lässt sich ein Problem nämlich in einer bestimmten mathematischen Form darstellen, so existiert ein entsprechendes Maschinenmodell, mit dem sich dieses berechnen lässt.

3 Voraussetzungen

Zentrale Begriffe dieser Dokumentation sind »Worte«, »Sprachen« und »Algorithmen«. Diese, und weitere wesentlich Begriffe sollen hier eingeführt werden.

Die Menge der natürlichen Zahlen wird durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ dargestellt, wobei Null eine natürliche Zahl ist.

Ein Alphabet ist eine endliche, nicht-leere Menge. Die Elemente nennen wir auch »Zeichen« oder »Buchstaben«.

Ein Wort ist eine endliche Folge von Buchstaben. Dies schließt die leere (aus keinem Buchstaben bestehende) Folge mit ein. Wir bezeichnen sie mit ε. Wir identifizieren Wörter mit natürlichen Zahlen; dazu denken wir uns eine geeignete ein-eindeutige Abbildung fixiert.

Eine Sprache ist eine Menge von Wörtern.

Das Komplement einer Sprache ist die Menge der Wörter über dem gleichen Alphabet, die aber nicht zur Sprache gehören.

Ein Beispiel für eine Sprache ist die Menge aller Quadratzahlen ($\{1, 4, 9, \ldots, n^2, \ldots\}$). Hierbei ist jede der Quadratzahlen ein Wort der Sprache. Als Buchstaben werden die Ziffern von 0 bis 9 verwendet. Eines der Wörter dieser Sprache ist 49, welches aus den Buchstaben 4 und 9 besteht. Das Komplement dieser Sprache ist $\{\varepsilon, 0, 2, 3, 5, \ldots, 00, 01, 02, \ldots\}$.

Rekursion bedeutet wörtlich Selbstbezüglichkeit. Eine rekursive Formel bezieht sich auf sich selbst. Oft betrachten wir rekursive Prozeduren, die sich selbst für kleinere Teilprobleme aufrufen.

Unäre Darstellung ist die Darstellung einer Zahl durch eine Folge von gleichartigen Zeichen.

Binärsystem meint die Darstellung einer Zahl mittels des Stellenwertsystems zur Basis zwei.

Logarithmen sind für uns stets Logarithmen zur Basis 2 und werden durch $\log$ bezeichnet.

Die Potenzmenge von X, bezeichnet als $P(X)$, ist die Menge aller Teilmengen von X.

Inkrementieren meint, einen Wert um 1 zu erhöhen.

Dekrementieren meint, einen Wert um 1 zu erniedrigen.

4 Was ist ein Algorithmus?

(Martin Walczack)

Wir sind oft daran interessiert, Probleme nicht nur zu lösen, sondern ein systematisches Verfahren zu haben, dieses Problem zu lösen; anders ausgedrückt, sind wir an einem Algorithmus interessiert. Einen Algorithmus zu erstellen heißt im Einzelnen:

- Nur in einzelnen, kleineren und damit auch einfacheren Schritten vorgehen.
- Das Problem aufsplitten und so lösbare Teilprobleme bekommen, deren zusammengefügte Lösungen auch das ursprüngliche Problem lösen.

Es ist dabei zu beachten, dass ein Algorithmus eine Handlungsvorschrift ist, die immer über ein endliches Alphabet sowie ein endliches Regelwerk verfügt, deterministisch (d. h. im Vorfeld bestimmt) ist und in endlicher Zeit abgearbeitet werden kann.

5 Berechnungsmodelle

Ziel der folgenden Ausführungen ist die Beschäftigung mit der Frage nach der Mechanisierbarkeit von Berechnungen. Wir werden uns dabei mit vier Maschinenmodellen und deren Beziehungen zueinander auseinandersetzen.

Allen Modellen ist gemein, dass sie zur Ausführung der Berechnungen einen Speicher zur Verfügung haben, den sie entsprechend der Befehle eines Programms manipulieren können. Da die Programme dazu verwendet werden, Funktionen zu berechnen, muss man ihnen Argumente, die sogenannten Eingaben, übergeben können, die dazu meistens in einen festgelegten Speicherbereich abgelegt werden.

5.1 Loop Programme (Uwe Bau, Lennart Galinat, Christoph-Simon Senjak, Martin Walczack)

Im Folgenden betrachten wir die Funktionsweise der abstrakten Maschine zur Ausführung von Loop-Programmen. Wir wenden uns zuerst dem Aufbau dieser Maschine zu. Die Maschine arbeitet mit einem Speicher, der in so genannte Register unterteilt ist. Dabei handelt es sich um Speicherplätze, die jeweils eine natürliche Zahl beinhalten. Loop-Programme haben dabei nur endlich viele dieser Register zur Verfügung. Diese bezeichnen wir im Folgenden mit $R_0, \ldots, R_n$.

Durch Inkrementieren und Dekrementieren der Register und durch Schleifen, die verschachtelt werden können, kann man Berechnungen durchführen.

Es gibt zur Beschreibung dieser Vorgänge drei Befehle.

- R_k++ inkrementiert das Register R_k.
- R_k-- dekrementiert das Register R_k, sofern nicht bereits $R_k = 0$ gilt. Falls dies der Fall sein sollte, bleibt der Inhalt des Registers Null.

– Die Befehlssequenz
  LOOP R_k DO
    <Anweisungsblock>
  END
  wiederholt die Befehle, die im Anweisungsblock stehen R_k mal, wobei der Initialwert von R_k bei Erreichen der LOOP-Anweisung ausschlaggebend ist.

Die Ausgabe des Programms soll am Ende in R_0 stehen; der Initialwert von R_0 wird ist 0. Die Eingaben sollen in den Registern $R_1, \ldots, R_k$ stehen. Zusätzlich können Hilfsregister $R_{k+1}, \ldots, R_n$ genutzt werden, deren Initialwert 0 ist.

Es folgt als Beispiel ein Loop-Programm zur Berechnung von $R_1 \cdot R_2$.

```
LOOP R1 DO
   LOOP R2 DO
      R0++
   END
END
```

5.2 Turingmaschinen (Natascha Schiel, Alexander Mänz, Nils-Edvin Enkelmann)

Die Turingmaschine ist ein mathematisches Modell einer Maschine, welches vom englischen Mathematiker Alan Turing (1912–1954) entwickelt wurde.

Eine Turingmaschine besteht aus einer Kontrolleinheit und endlich vielen Bändern. Dabei kann die Kontrolleinheit mit Hilfe von Schreib- und Leseköpfen (pro Band ein Kopf) auf die Bänder zugreifen und sie bearbeiten.

Jedes Band besteht aus einzelnen Feldern, in denen Buchstaben eines Alphabets enthalten sind. Die Bänder sind in eine Richtung unbegrenzt, wobei das begrenzte Ende Anfangs durch ein Dollar-Zeichen markiert ist. Die Schreib- und Leseköpfe auf den Bändern können pro Befehl immer nur ein Zeichen lesen und schreiben.

Das Verhalten der Turingmaschine wird durch die Kontrolleinheit gesteuert, deren Verhalten wiederum von den gelesenen Zeichen, sowie dem momentanen Zustand der Kontrolleinheit abhängt. Die Kontrolleinheit kann nur endlich viele Zustände einnehmen, wobei der Zustand 0 die Turingmaschine beendet und 1 der Startzustand ist.

In den Turingtabellen ist festgelegt, wie die Kontrolleinheit in einem bestimmten Zustand bei einer bestimmten gelesenen Zeichenkombination reagiert. Dabei kann die Kontrolleinheit die gelesenen Zeichen überschreiben, die Schreib- und Leseköpfe um ein Feld nach links oder rechts bewegen und den Zustand der Maschine ändern.

Dies sei nun am Beispiel einer Turingmaschine mit drei Bändern zur Berechnung des Produktes zweier Zahlen näher erklärt. Dabei ist wichtig, dass die Darstellung der Zahlen unär erfolgt; beispielsweise entspricht die Zahl 3 auf dem Band der Zeichenfolge $\$1110000\ldots$.

Zustand I	11?	111	SRR	I
	10?	100	RLS	II
	01?	100	SSS	0
Zustand II	10?	100	SLS	II
	11?	110	SLS	II
	1$?	1$0	SRS	I

In der ersten Spalte der Tabelle stehen die von den Schreib- und Leseköpfen gelesen Zeichen (pro Band einmal »0« oder »1«; »?« ist hier kurz für »0 oder 1«), die zweite Spalte enthält die auf die Bänder geschriebenen Zeichen und in der dritten Spalte stehen die darauf folgenden Bewegungen der Schreib- und Leseköpfe (S: Stop, L: 1 nach links, R: 1 nach rechts). Schließlich enthält die vierte Spalte den neuen Zustand der Kontrolleinheit.

Diese Turingmaschine arbeitet nach folgendem Prinzip. Für jede 1 auf dem ersten Band geht sie das zweite Band durch und kopiert alle Einsen des zweiten Bands auf das dritte Band. Folglich erhalten wir bei der Eingabe x und y gerade $x \cdot y$ Einsen auf dem dritten Band als Ergebnis.

Schließlich bemerken wir noch, dass das weitere Verhalten einer Turingmaschine vollständig festgelegt ist durch die folgenden Daten.

– der Zustand der Kontrolleinheit
– der Inhalt der Bänder (gekürzt um die unendlich vielen Nullen am Ende)
– die Position der Schreib- und Leseköpfe

Diese Daten zusammen nennen wir auch die *Konfiguration* der im Lauf befindlichen Turingmaschine.

5.3 Registermaschinen (Nina Berges, Friederike Obergfell)

Die Komponenten aus denen sich eine Registermaschine zusammensetzt nennen sich *Register*. Sie lassen sich einteilen in die *Eingaberegister* $R_1, \ldots, R_k$, das *Ausgaberegister* R_0, in welchem stets das Ergebnis gespeichert wird, sowie endlich viele *Hilfsregister* $R_{k+1}, \ldots, R_n$, die genau wie das Ausgaberegister mit dem Wert 0 vorbelegt sind. Ebenfalls ist festgelegt, dass die Werte der Register aus dem Bereich der natürlichen Zahlen kommen.

Registermaschinen verfügen über die folgenden Befehle.

1) HALT. Das Programm stoppt.
2) R_x++ b. Das Register R_x wird um 1 erhöht ($R_x = R_x + 1$), anschließend wird mit dem Befehl (in Zeile) b fortgefahren.
3) R_x++ b. Das Register R_x wird um 1 erniedrigt ($R_x = R_x - 1$), falls nicht bereits den Wert 0 hat. Dann wird mit Befehl b fortgefahren.
4) IF R_x a b. Abhängig davon, ob das Register R_x den Wert 0 beinhaltet wird mit Befehl a (im Falle $R_x = 0$) oder mit Befehl b fortgefahren.

Die Buchstaben am Ende der Befehle – hier a und b – signalisieren dem Programm zu dem Befehl zu »springen«, der durch sie gekennzeichnet ist, und von dort aus weiterzuarbeiten.

Der Befehl HALT steht ausschließlich als 0-ter Befehl und der 0-te Befehl muss stets HALT sein. Die Ausführung des Programms beginnt mit dem 1-ten Befehl.

Die Funktionsweise der Registermaschine wird nun anhand von einigen Beispielen demonstriert.

Die Identität ist die Funktion, die den Eingabewert ins Ausgaberegister überträgt, also den Wert von R_1 nach R_0 verschiebt.

0	HALT
1	IF R_1 0 2
2	R_1-- 3
3	R_0++ 1

Erläuterung: Wir sehen hier eine Schleife, die so lange 1 von R_1 abzieht, bis $R_1 = 0$. Gleichzeitig wird die jeweils abgezogene 1 zu R_0 addiert. So bleibt die »Menge der Einsen«, genauer $R_1 + R_0$, erhalten und R_0 – das Ergebnis – trägt schließlich den Anfangswert von R_1, weil R_1 dann 0 ist.

Löschen eines Registers R_x, also es auf 0 setzen, wird durch das folgende Programmfragment TOZERO(R_x, z) geleistet.

TOZERO(R_x, z)

a	IF R_x z b
b	R_x-- a

Erläuterung: An dieser Stelle muss erwähnt werden, dass wir Funktionen wie diese an verschiedenen Stellen eines Programms stehen haben möchten, ohne dass sie daraufhin abermals komplett erfunden werden müssen.

Dies erreichen wir, indem wir den Namen der Funktion (hier: TOZERO) schreiben und anschließend die dazugehörigen Parameter (hier: R_x, z) innerhalb der Klammer übergeben.

Dies ist als Abkürzung für das entsprechende Programm zu sehen, das mit diesen Registern arbeitet. Wir übertragen dazu die neu eingegebenen Parameter in das bereits existente Schema.

Im Fall TOZERO wird mit R_x das Register benannt, dessen Wert auf 0 gesetzt werden soll und mit z der Befehl, der nach Ausführung der Operation aufgerufen werden soll.

In dem Fall, dass der auf 0 zu setzende Registerwert bereits 0 ist, setzt das Porgramm die Berechnung an der Stelle z fort, andernfalls wird so lange 1 vom Wert R_x abgezogen, bis dieser 0 ist.

Addieren beziehungsweise Subtrahieren des Wertes in R_y zu dem Wert in R_x kann mit Hilfe der folgenden Funktion ADD(R_x, R_y, z) beziehungsweise MINUS(R_x, R_y, z) erreicht werden.

ADD(R_x, R_y, z)

a	IF R_y z b
b	R_x++ c
c	R_y-- a

Erläuterung: Im Grunde ist die Idee hinter diesem Programm, dass von dem Wert R_y, der addiert werden soll, so lange 1 abgezogen wird, bis $R_y = 0$ ist. Gleichzeitig werden entsprechend diese Einsen zu dem Wert, zu dem R_y addiert werden soll – nämlich R_x – dazu addiert. Alles was »hier« abgezogen wird, wird »da« wieder hinzugefügt.

Es ist zu beachten, dass der Wert R_y dabei verloren geht, also am Ende den Wert 0 trägt. Wie man solche Effekte verhindert werden kann, wird in dem nächsten Beispiel beschrieben.

Die Funktion MINUS funktioniert ähnlich. Verändert wird dabei nur Befehl b und zwar wird das ++ wird durch ein -- ersetzt. Es wird R_y von R_x abgezogen.

Kopieren des Registers R_x in das Register R_y geschieht mit Hilfe eines dritten Registers R_{Puffer}. Die Funktion wird im weiteren als COPY(R_x, R_y, R_{Puffer}, z) bezeichnet, wobei z wieder der Befehl ist, mit dem nach Ausführung fortgefahren werden soll.

Zur Erklärung dieses Programms wollen wir mit Befehl d anfangen. Prinzip des Programms ist ähnlich wie bei der Identität. R_x wird um 1 verringert und gleichzeitig wird R_y um 1 vergrößert, bis R_x Null ist.

So haben wir am Ende den ursprünglichen Wert von R_x in R_y, weil wir in Befehl a den Wert von R_y auf 0 gesetzt haben.

COPY(R_x, R_y, R_{Puffer}, z)

a	TOZERO(R_y, b)
b	TOZERO(R_{Puffer}, c)
c	IF R_x g d
d	R_x-- e
e	R_y++ f
f	R_{Puffer}++ c
g	ADD(R_x, R_{Puffer}, z)

Um zu vermeiden, dass R_x bei Terminierung der Funktion den Wert 0 hat, muss der ursprüngliche Wert gespeichert werden. Dafür haben wir ein drittes Register R_{Puffer}, in welchem der gleiche Wert wie in R_y gespeichert wird. Mit der Funktion ADD, die wir gerade gesehen haben, wird dieser Wert am Ende der Funktion COPY zu R_x addiert, also letztendlich $0 + R_y$, und R_x erhält wieder seinen alten Wert.

Vergleichen der Werte R_x und R_y und Sprung entsprechend des Ergebnisses des Vergleichs kann mit folgender Funktion COMPARE(R_x, R_y, R_{xPuffer}, R_{yPuffer}, $R_{\text{copyPuffer}}$, $\text{Befehl}_=$, $\text{Befehl}_<$, $\text{Befehl}_>$) erreicht werden.

Erläuterung: Zunächst werden in den ersten beiden Schritten die Werte aus den Registern R_x und R_y jeweils in Puffer kopiert mittels der bereits oben eingeführten Funktion COPY.

Um zu überprüfen, ob der Wert R_{xPuffer} größer, kleiner oder gleich dem Wert R_{yPuffer} ist, wird in den Befehlen c, d und e überprüft, ob einer der Werte 0 ist und dann entsprechend, ob der andere es auch ist, oder größer.

Für den Fall, dass beide Werte größer 0 sind, gibt es die Befehle f und g, in denen von beiden Werten immer wenn dieser Fall eintritt, 1 abgezogen wird, solange bis einer der Werte auf 0 angekommen ist; dann treten wieder die Befehle d, h, und l auf.

Hier ein Beispiel zur Verdeutlichung. Wir gehen von den Anfangswerten $R_y = 2$ und $R_y = 0$ aus:

COMPARE(R_x, R_y, R_{xPuffer}, R_{yPuffer}, $R_{\text{copyPuffer}}$, $\text{Befehl}_=$, $\text{Befehl}_<$, $\text{Befehl}_>$)

a	COPY(R_x, R_{xPuffer}, $R_{\text{copyPuffer}}$, b)
b	COPY(R_y, R_{yPuffer}, $R_{\text{copyPuffer}}$, c)
c	IF R_{xPuffer} d e
d	IF R_{yPuffer} h j
e	IF R_{yPuffer} l f
f	R_{xPuffer}-- g
g	R_{yPuffer}-- c
h	TOZERO(R_{xPuffer}, i)
i	TOZERO(R_{yPuffer}, $\text{Befehl}_=$)
j	TOZERO(R_{xPuffer}, k)
k	TOZERO(R_{yPuffer}, $\text{Befehl}_<$)
l	TOZERO(R_{xPuffer}, m)
m	TOZERO(R_{yPuffer}, $\text{Befehl}_>$)

- $R_{\text{xPuffer}} = 2$ und $R_{\text{yPuffer}} = 0$.
- R_{xPuffer} ungleich 0 also weiter zu e.
- $R_{\text{yPuffer}} = 0$ also weiter zu l.
- R_{xPuffer} wird wieder »geleert«, mit m wird fortgefahren.
- R_{yPuffer} wird geleert, anschließend wird zum Befehl $\text{Befehl}_>$ im Programm gesprungen da $x > y$ gilt; dementsprechend wird weitergearbeitet.

5.4 Random Access Machines (Robert Pfeiffer)

Die Random Access Machine (RAM) beschreibt ebenfalls eine Maschine, die ein abstraktes Modell von Berechenbarkeit darstellt. Das Programm wird zeilenweise nummeriert.

Die RAM besitzt einen Speicher, der unendlich viele Speicherregister $R_1, R_2, \ldots$ enthält. Auf diese Speicherregister wird über ihren Index, also ihre Adresse im Speicher, zugegriffen.

Außerdem stellt sie sechs Register zur Verfügung, nämlich die Arbeitsregister X, Y, Z, ferner das Adressregister A, das Befehlsregister B, und das Verzweigungsregister F, auf English »flag register«. Diese haben folgende Funktionen.

- A enthält die Adresse des aktuellen Speicherregisters.
- B enthält die Nummer der Programmzeile, die als nächstes ausgeführt wird.
- Das Register F enthält das Ergebnis des letzten Vergleichs. Es wird bei bedingten Sprüngen mit JUMP benötigt.
- X, Y und Z sind Operationsregister. In Y und Z werden die Ausgangswerte für die arithmetischen Operationen abgelegt. Das Ergebnis jeder arithmetischen Operationen wird in X geschrieben.

An Speicheroperationen stehen zur Verfügung:

- COPY(V, V'). Der Inhalt des Registers V' wird in das Register V geschrieben. V darf nicht B sein.
- LOAD(V, c). Der Wert c wird in das Register V geschrieben.
- READ(V). Der Inhalt des Speicherregisters, dessen Index im Adressregister steht, wird in das Register V abgelegt.
- WRITE(V). Der Inhalt des Registers V wird in das Speicherregister mit dem Index aus dem Adressregister abgelegt.

Der Programmablauf kann mit den folgenden Befehlen beinflusst werden.

- COMP $\circ$. Die Register X und Y werden mit der Relation $\circ$ verglichen. Das Ergebnis wird in F abgelegt. Dabei ist $\circ \in \{<, >, \leq, \geq, =, \neq\}$.
- JUMP b. Wenn F wahr enthält, wird b in das Befehlsregister geschrieben. Dadurch wird das Programm an der b-ten Zeile fortgesetzt.
- STOP. Die Ausführung des Programms wird beendet.

Arithmetische Operationen operieren auf den oben genannten Registern. Dabei steht ADD für Addition, SUB für Substraktion, MULT für Multiplikation, DIV für Division und SHIFT für Division durch 2.

Die Eingabe steht, wie üblich, in den Registern $R_1, \ldots, R_k$ und die Ausgabe soll im Register R_0 stehen, das, genau wie alle Hilfsregister $R_{k+1}, \ldots$ mit 0 vorbelegt ist. Die Ausführung des Programms beginnt mit dem 1-ten Befehl.

Als Beispiel ist hier eine naive Implementierung eines Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier positiver ganzer Zahlen angegeben.

Dazu erinnern wir uns an die folgenden Eigenschaften des ggT, die motiviert, stets die kleinere von der größeren Zahl abzuziehen bis die Zahlen schließlich gleich sind.

$$\text{ggT}(x, y) = \text{ggT}(x, x + y) = \text{ggT}(x + y, x)$$
$$\text{ggT}(x, x) = x$$

```
01  LOAD A, 1
02  READ X
03  LOAD A, 2
04  READ Y
05  COMP =
06  JUMP 21
07  COMP <
08  JUMP 15
09      COPY Z, Y
10      COPY Y, X
11      SUB
12      COPY Y, Z
13  LOAD F, 1
14  JUMP 05
15      COPY Z, X
16      SUB
17      COPY Y, X
18      COPY Z, X
19  LOAD F, 1
20  JUMP 05
21  LOAD A, 0
22  WRITE X
23  STOP
```

Es wird sukzessive in den eingerückten Zeilen die kleiner von der größeren Zahl abgezogen, bis die Zahlen gleich sind, was in Zeile 05 überprüft wird.

Zur Erläuterung: Das Programm lädt die Zahlen, deren größter gemeinsamer Teiler gebildet werden soll, als erstes aus den Speicherregistern R_1 und R_2 in X und Y. Danach wird der größte gemeinsame Teiler gebildet und in R_0 abgelegt.

6 Äquivalenz der Maschinentypen

Da wir nun verschiedene Berechnungsmodelle eingeführt haben stellt sich die Frage, wie sich diese zu einander verhalten. Einige der Modelle können wir ineinander einbetten.

6.1 Die Äquivalenz der Modelle Ein-Band Turing-Maschine und Mehr-Band Turingmaschine (Cornelia Strauß)

Eine k-Band-Turing Maschine mit $k \in \mathbb{N}$ hat k Bänder, k Schreib- und Leseköpfe auf den Bändern und gemeinsame Zustände für alle Bänder. Die Ein- und Ausgabe steht auf dem ersten Band.

Satz 1 *k-Band-Turingmaschinen sind genauso mächtig wie 1-Band-Turingmaschinen.*

Beweis *Da jede 1-Band-Turingmaschine auch eine k-Band-Turingmaschine ist, bleibt zu zeigen, dass sich jede k-Band-Turingmaschine durch eine 1-Band-Turingmaschine ersetzen lässt.*

Wir ersetzen jetzt eine beliebige k-Band-Turingmaschine durch eine 1-Band-Turingmaschine, deren Band in beide Richtungen unbegrenzt ist. Das Band der 1-Band-Turingmaschine wird in k Spuren unterteilt. Die k Schreib- und Leseköpfe werden durch einen einzigen Schreib- und Lesekopf ersetzt. Am Anfang stehen außer

der Eingabe nur Nullen oder ein anderes Markierungszeichen auf dem Band. Ein Problem ergibt dabei durch die gleichzeitige Verschiebung mehrerer Köpfe in verschiedene Richtungen. Die Verschiebung des Kopfes nach links oder rechts entspricht der Verschiebung des Bandes in die entgegengesetzte Richtung. Die Lösung besteht darum in der schrittweisen Verschiebung aller Spuren nacheinander, auf denen die Schreib- und Lesekopfposition verändert werden soll, in die entsprechende Richtung. Bei der schrittweisen Verschiebung einer Spur nach links z. B. bewegt sich der Kopf erst bis zum linken Ende der zu bearbeitenden Zeichen, anschließend bis zum rechten Ende und zurück zur Anfangsposition. Bei jeder Bewegung nach links wird das letzte Zeichen in einem Zustand gespeichert und auf die nächste Position geschrieben. Für jedes Zeichen wird dazu eindeutig ein neuer Zustand definiert. Für das Zurückfahren zum rechten Ende wird noch ein weiterer neuer Zustand definiert. Die Ausgangsposition wird mit einem neu definierten Zeichen markiert, um dort am Ende wieder stehen zu bleiben. Die neuen Zustände werden für jede einzelne Spur definiert.

Schließlich müssen wir noch zeigen, dass eine 1-Band Turingmaschine mit beidseitig unbegrenzten Band von einer 1-Band Turingmaschine mit nur einseitig unbegrenztem Band simuliert werden kann. Dazu verwenden wir einen ähnlichen Trick und teilen das Band in zwei Spuren, eine für die Zeichen links vom »Ursprung«, die andere für die Zeichen rechts vom Ursprung, sozusagen nach rechts umgeklappt. Auf welcher der beiden Spuren wir uns gerade befinden können wir uns im Zustand merken. Wenn wir das besondere Zeichen vor dem Rand des Bandes lesen dann gehen wir nach rechts und wechseln die Spur.

q.e.d.

Der letzte Abschnitt des obigen Beweis zeigt ein Ergebnis, das für sich genommen interessant ist, deshalb wiederholen wir es separat.

Satz 2 *1-Band Turingmaschinen mit einseitig begrenzten Band sind genau so mächtig, wie 1-Band Turingmaschinen mit zweiseitig unbegrenztem Band.*

6.2 Einbettung von Turingmaschinen in Registermaschinen

(Martin Müller, Christine Schmidt, Cornelia Strauß)

Im Folgenden ist zu zeigen, dass alle Funktionen, die durch eine Turingmaschine berechenbar sind, auch durch eine Registermaschine berechenbar sind. Wir gehen von einer 1-Band-Turingmaschine aus, da jede k-Band-Turingmaschine auf diese zurückführbar ist. Wir bauen alle möglichen Operationen auf einer Registermaschine nach. Diese sind Lesen, Schreiben und das Verschieben des Schreib- und Lesekopfs. Die verschiedenen Zustände findet man in der Befehlsfolge der Registermaschine wieder. Bei dieser Betrachtung verwenden wir eine Turingmaschine mit dem Eingabealphabet $\{0, 1, T\}$. Dies bedeutet keine Einschränkung, da jedes größere Alphabet auf dieses zurückführbar ist.

Wir gehen von folgender Konvention für die Eingabe der Turingmaschine aus, was offensichtlich keine Einschränkung darstellt. Das Band ist linksseitig durch ein Dollar-Zeichen begrenzt. Die zu beachtende Zahlenfolge wird durch das vereinbarte Zeichen »T« abgeschlossen. Danach folgen ausschließlich Nullen.

Wir interpretieren die Folge der Ziffern als Dualzahl. Damit sich jede der aufsummierten Zweierpotenzen aus der Positionsnummer ergibt, wird die Zeichenfolge umgedreht. Wir vereinbaren, dass das Programm bei einer Verschiebung über den festgelegten Rand hinaus stoppt.

Die von uns konstruierte Registermaschine ist wie folgt aufgebaut. In Register R_0 erfolgt die Ein- und Ausgabe natürlicher Zahlen. Register R_1 enthält die Teilzeichenfolge, die auf der umgedrehten Zeichenfolge vom T-Zeichen bis einschießlich des Lesekopfes gelesen wird, Register R_2 die Zeichenfolge, vom Dollar-Zeichen bis vor den Lesekopf. Für das Turing Band `$ 1 1 0 1 1 0 1 1 1 T 0 0 0 ...` erbit sich also nebenstehende Registerbelegung.

$R_1 = 11101_{(2)} = 29$
$R_2 = 1101_{(2)} = 13$
$R_3 = 4$

Wir können nun die einzelnen Operationen wie folgt umsetzen.

Lesen nach R_0

a	IF »$R_1 \bmod 2$« b d
b	IF R_0 fertig c
c	R_0-- b
d	IF R_0 f e
e	R_0-- d
f	R_0++ fertig

Eine 0 schreiben.

a	IF »$R_1 \bmod 2$« fertig b
b	R_1-- fertig

Eine 1 schreiben.

a	IF »$R_1 \bmod 2$« b fertig
b	R_1++ fertig

Verschieben nach links

a IF R_3 stop b
b R_3-- c
c IF »$R_2 \bmod 2$« d e
d »$R_1 = 2 \cdot R_1$« f
e »$R_1 = 2 \cdot R_1 + 1$« f
f »$R_2 = R_2 \operatorname{div} 2$« fertig

Verschieben nach rechts

a R_3++
b IF »$R_1 \bmod 2$« c d
c »$R_2 = 2 \cdot R_2$« e
d »$R_2 = 2 \cdot R_2 + 1$« e
e »$R_1 = R_1 \operatorname{div} 2$« fertig

Wenn wir im oben erwähnten Beispiel nach rechts verschieben, ergeben sich also schließlich die Werte $1110_{(2)}$ für R_1 und $11011_{(2)}$ für R_2. Ferner enthält R_3 dann den Wert 5.

6.3 Einbettung von Registermaschinen in Random Access Machines (Nils-Edvin Enkelmann)

Hier wird bewiesen, dass jede Registermaschine auch in einer RAM-Maschine simulierbar ist. Dazu geben wir für die Eingabe der Registermaschine in die RAM folgende Eingabekonventionen vor:

- In R_1 steht die Anzahl der Befehle n der simulierten Registermaschine
- In R_2 bis R_{4n+1} stehen die einzelnen Befehle, wobei für jeden Befehl vier Register verbraucht werden
- Jeder Befehlsblock ist wie folgt aufgebaut: im ersten Register steht der Befehlstyp, im zweiten das Register, auf welches zugegriffen wird und im dritten (bei der if-Anweisung auch im vierten) steht die Befehlsnummer, mit welcher weitergemacht wird.

Zusätzlich vereinbaren wir für unsere interne Speicherorganisation

- ab dem Register R_{4n+2} folgen die simulierten Register der Registermaschine, beginnend mit dem simulierten R_0 in R_{4n+2}.

Das Programm arbeitet dann wie folgt: Zuerst wird zum aktuellen Befehl gesprungen (und die Befehlsnummer des aktuellen Befehls in R_0 gespeichert), der Befehlstyp gelesen und das je nach Befehlstyp entsprechende Makro aufgerufen. Wird der HALT-Befehl aufgerufen, wird der Wert aus dem simulierten R_0 in das echte R_0 kopiert und die Maschine beendet.

Da eine Random Access Mashine alle Befehle einer Registermaschine zur Verfügung hat, geht diese Simulation glatt durch; lediglich darf man nicht vergessen, die Register umzurechnen, da das simulierte R_k ja real an Stelle R_{4n+2+k} steht.

6.4 Einbettung von Random Access Machines in Turingmaschinen (Lennart Galinat)

Im folgenden Text soll beschrieben werden, wie man eine beliebige RAM in eine Turingmaschine eingebettet werden kann.

Hierzu verwenden wir je ein Band für die Register X,Y und Z, ein Band für das Register A, ein Band für das Register B, ein Band für den Programmtext, ein Band für den Zusatzspeicher und ein zusätzliches Hilfsband. Die einzelnen Register bis zu dem Register mit dem maximal verwendeten Index werden mit einem Trennzeichen (im folgenden »T« genannt) voneinander getrennt.

Zum Lesen eines Registers aus dem Speicher suchen wir auf dem Speicherband das entsprechende Register, indem wir die Anzahl der Trennzeichen auf dem Speicherband gegen eine Kopie von A abstreichen. Dann wird der Inhalt des Bandes bis zum nächsten Trennzeichen zur weiteren Verwendung auf das Hilfsband kopiert. Sollten nicht genügend Trennzeichen auf dem Hilfsband vorhanden sein, so springt die Turingmaschine in einen Sonderzustand und gibt den Wert Null zurück, da jedes neu initialisierte Register einer RAM den Wert Null trägt.

Beim Schreiben auf dem Speicherband können zwei potentielle Probleme auftreten.

- Auch wenn auf das Register R_k noch nicht zugegriffen wurde, kann es vorkommen, dass R_{k+1} beschrieben werden soll. In diesem Fall wird einfach die entsprechende Anzahl an Trennzeichen auf das Band geschrieben, um so das neue Register auf dem Speicherband anzulegen (und korrekter Weise mit 0 vorzubelegen).

- Wenn R_k überschrieben werden soll und die Darstellung des neuen Wertes länger oder kürzer ist, als der zu überschreibende Wert, könnte bei naiver Realisierung Datenverlust oder Datenkorruption auftreten. Um dieses Problem zu lösen, wird zuerst alles, was hinter R_k steht, auf das Hilfsband H kopiert. Nun

wird R_k überschrieben, der Inhalt des Hilfsbandes dahinter eingefügt und der Inhalt des Hilfsbandes gelöscht.

Wie nun in groben Zügen gezeigt wurde, ist es möglich eine beliebige RAM auf einer Turingmaschine abzubilden. Die oben beschriebene Methode geht allerdings noch einen Schritt weiter und erlaubt das Ausführen eines beliebigen RAM Programmes, wenn dieses codiert auf dem Eingabeband steht, auf einer Turingmaschine. Dies bedeutet, dass diese Turingmaschine eine beliebige RAM simulieren kann.

Maschinen, die alle anderen Maschinen ihrer Art simulieren können, beziehungsweise als eine Art Interpret für diese dienen, nennt man universelle Maschinen. Solche universellen Maschinen werden in einem eigenen Kapitel betrachten.

6.5 Zusammenfassung (Lennart Galinat, Nils-Edvin Enkelmann)

Da wir gezeigt haben, dass sich die drei Maschinentypen RAM, TM und RM direkt oder indirekt (z. B. Random Access Machines in Registermaschinen via Turingmaschinen) von den jeweils anderen Maschinentyp simulieren lassen, ist damit die Äquivalenz der drei Maschinentypen bewiesen. Daher verwenden wir im Folgenden RAM-Berechenbarkeit, Turing-Berechenbarkeit und Register-Berechenbarkeit synonym und sprechen nur kurz von *berechenbar*.

7 Das optimale Loop-Programm

(Nils-Edvin Enkelmann)

Loop-Programme haben im Gegensatz zu den anderen von uns betrachteten Modellen den Nachteil, dass sie auf jeden Fall terminieren; es sind also keine Endlosschleifen möglich. Nun stellt sich die Frage, wie »mächtig« ein Loop-Programm einer bestimmten Länge ℓ denn höchstens sein kann. In der Tat werden wir in diesem Abschnitt eine berechenbare, aber nicht Loop-berechenbare Funktion finden, und damit zeigen, dass Loop-Programme echt weniger mächtig sind, als die anderen Berechnungsmodelle.

Um dies herauszufinden, suchen wir eine Turing-berechenbare Funktion, welche uns den maximalen Ausgabewert eines Loop-Programms der Länge ℓ und der Eingabe n errechnet. Wenn wir zu einer solchen Funktion noch eins dazuaddieren, ist sie nicht mehr Loop-berechenbar. Da sie aber noch Turing-berechenbar ist, kann ein Loop-Programm nicht jede Turingmaschine simulieren. Bei dem gesuchten Programm setzen wir entgegen der bisherigen Konvention voraus, dass R_0 Eingabe- und Ausgaberegister zugleich ist und $n > 1$ gilt.

Dazu beweisen wir, dass die Struktur des optimalen Loop-Programms wie folgt aussieht.

```
LOOP R_0
    LOOP R_0
        ...
        LOOP R_0
            R_0++
        END
        ...
    END
END
```

Beweis *Gegeben sei ein Loop-Programm P. Als erstes entfernen wir alle R_x-- innerhalb des Programms, da diese nicht zur Erhöhung irgendeines Registers beitragen können. Dann ersetzen wir alle R_x durch R_0. Auch dadurch können wir den Ausgabewert nur erhöhen, denn immer wenn wir den Wert eines anderen Registers erhöhen ist es "effizienter" stattdessen R_0 zu erhöhen. Schlussendlich entfernen wir noch alle Schleifen, welche kein R_0++ enthalten, da sie keinen Einfluss auf die Ausgabe haben. Ein so verändertes Loop-Programm nennen wir normalisiert.*

Nehmen wir an, unser normalisiertes Programm P enthalte mehr als nur ein R_0++. Dann sieht man leicht, dass ein Unterprogramm Q in P existiert, so dass Q die Struktur $Q_1; Q_2$ hat und sowohl Q_1 als auch Q_2 jeweils genau ein R_0++ enthalten. Sei nun Q_m das schneller wachsende der beiden Programm Q_1 und Q_2; da Q_1 und Q_2 nur die obere Programmstruktur haben können, ist eins von beiden unabhängig vom Wert in R_0 immer schneller

wachsend als das andere. Folglich ist $Q_m;Q_m$ effektiver als $Q_1;Q_2$ und, da $R_0 \geq n > 1$ gilt, ist LOOP R_0 Q_m END effektiver als $Q_m;Q_m$, weswegen wir dies anstelle von $Q_1;Q_2$ in P einsetzen. Bei dieser Umformung von P hat sich die Länge von P nicht erhöht. Diesen Schritt können wir nun so lange wiederholen, bis P exakt ein R_0++ enthält. Damit kann das optimierte P nur noch die obige Programmstruktur besitzen, was zu beweisen war.

q.e.d.

Die maximale Ausgabe eines Loop-Programms der Länge ℓ und der Eingabe n (mit $n > 1$) erhalten wir jetzt durch die Ausführung des optimierten Loop-Programms mit der Länge ℓ und der Eingabe n.

8 Universelle Maschinen

(Uwe Bau, Benedikt Ewald)

Wie wir schon bei der Einbettung von Registermaschinen in Random Access Machines gesehen haben, gibt es Maschinen, die in der Lage sind, das Verhalten beliebiger Maschinen zu simulieren. Solche Maschinen nennt man *universell*.

Dafür wird eine Beschreibung der zu simulierenden Maschine als Eingabe an die universelle Maschine übergeben. Somit ist eine universelle Maschine in der Lage, jede Berechnung durchzuführen. Dadurch muss man nicht für jede Berechnung eine Maschine bauen, da es ausreicht, eine universelle Maschine zu bauen, die in der Lage ist jegliche Berechnung zu simulieren. Dieses Prinzip realisiert ein jeder PC, da man diesen ebenfalls als universelle Maschine verstehen kann, denn er ist in der Lage, jegliches Programm auszuführen. Insbesondere müssen wir uns nicht für jede neue Anwendung einen neuen Rechner kaufen.

Wir werden im Folgenden eine universelle RAM konstruieren. Als erstes betrachtet man die Eingabekonvention.

- R_0 ist die Ausgabe der »virtuellen« RAM (und damit auch der universellen, »realen« RAM)
- R_1 enthält die Anzahl n der Befehle der simulierten RAM.
- In $R_2, \ldots, R_{3n+1}$ steht der Programmcode der simulierten RAM, erstellt aus den formalisierten Eingaben. Pro Befehl werden hierfür jeweils drei Register benötigt. Das erste Register gibt des Befehlstyp an, die beiden folgenden Register die Parameter. Wenn ein Befehl weniger als zwei Parameter hat, dann werden die betreffenden Register mit 0 belegt. Die Befehlstypen codieren wir wie folgt: 0 HALT – 1 LOAD – 2 READ – 3 WRITE – 4 COPY – 5 COMP – 6 JUMP – 7 ADD – 8 SUB – 9 SHIFT – 10 MULT – 11 DIV.
- In R_{3n+2} wird die Anzahl p der Eingabeparameter der zu simulierenden RAM, eingetragen. Darauf folgen in $R_{3n+3}, \ldots, R_{3n+p+2}$ die Eingaben.

Für Bearbeitung und Ausgabe haben wir folgende Konventionen.

- $R_{3n+p+3}, \ldots, R_{3n+p+8}$ sind die sechs Arbeitsregister X, Y, Z, A, B und F.
- Die R_{3n+p+9} und $R_{3n+p+10}$ sind Hilfsregister, die stets die beiden Argumente des aktuell zu simulierenden Befehls enthalten.
- Bis hierhin wurden die Register komplett dazu genutzt, eine RAM zu simulieren. Erst ab $R_{3n+p+11}$ werden dann die Register für ihren eigentlichen Zweck von der zu simulierenden RAM benutzt.

Das konkrete Programm der universellen RAM, das die Befehle für jede RAM nachbildet ist natürlich extrem umfangreich. Auf eine Darstellung wird verzichtet.

9 Codes von Maschinen

Wie wir im vorangegangen Kapitel gesehen haben, können wir eine beliebige (in diesem Fall Random Access) Maschine eindeutig durch eine Folge von Zahlen und damit letztendlich durch ein Wort codieren.

Wir wählen ein für alle mal, für jeden Maschinentyp eine systematische Art um eine Maschine T durch ein Wort $\ulcorner T \urcorner$ eindeutig zu codieren. Aus Bequemlichkeit vereinbaren wir weiterhin, dass jede Zeichenkette, die sich bisher noch nicht als $\ulcorner T \urcorner$ für ein T ergibt, als Code für diejenige Maschine verstanden wird, die nichts tut und sofort anhält ohne ihre Eingabe zu betrachten.

10 Funktionen versus Prädikate

Bisher haben wir uns vor allem für Funktionen interessiert. Im Folgenden werden aber auch *Prädikate* immer wichtiger, also die Frage, ob eine Zeichenkette zu einer bestimmten Menge von Zeichenketten gehört.

Hierzu ist eine andere Konvention üblich, mit der uns die Maschine ihre Antwort mitteilt. Wir vereinbaren, dass es als *Akzeptanz* gilt, wenn die Maschine jemals den Haltezustand erreicht. Wir können ohne Einschränkung annehmen, dass die Maschinen, bei Wörtern, die sie verwerfen will unendlich lange rechnet.

Auf Grund dieser verschiedenen Konventionen muss man deutlich zwischen einem Prädikat und der dazugehörigen charakteristischen Funktion unterscheiden; dabei ist die *charakteristische Funktion* eines Prädikats diejenige Funktion, die genau auf den Wörtern, die zur Sprache gehören, 1 ist und 0 sonst.

Weiter sagen wir, dass eine Maschine eine Sprache *entscheidet* genau dann, wenn sie ihre charakteristische Funktion berechnet.

Also können wir eine Sprache akzeptieren, wenn sie entscheidbar ist. Die Umkehrung ist im Allgemeinen nicht wahr. Ein Beispiel werden im Kapitel über das Halteproblem sehen.

Schließlich bemerken wir noch den folgenden Zusammenhang.

Satz 3 *Eine Sprache ist genau dann entscheidbar, wenn sie und ihr Komplement akzeptiert werden können.*

Beweis *Offensichtlich können wir aus einer Entscheidung auch immer eine Akzeptanz machen, in dem wir bei der entsprechenden Antwort in eine Endlosschleife springen.*

Für die umgekehrte Richtung simulieren wir die beiden Maschinen, die Sprache beziehungsweise Komplement akzeptieren – allerdings gleichzeitig. Genauer führen wir abwechselnd Simulationsschritte der einen und der anderen Maschine aus. Da jedes Wort entweder zur Sprache oder zu ihrem Komplement gehört, wird in jedem Fall (genau) eine der beiden simulierten Maschinen den Haltezustand erreichen. Wenn dies geschieht, können wir die entsprechende Antwort ausgeben und anschließend terminieren.

q.e.d.

11 Unentscheidbare Probleme

Nicht alle Probleme können mittels Algorithmen gelöst werden. Wir werden uns einige spezielle unentscheidbare Probleme ansehen.

11.1 NonSelfAccepting (Benedikt Ewald)

Die Sprache NonSelfAccepting (NSA) ist genau die Menge von Zahlen $\ulcorner T \urcorner$ für eine Turingmaschine T, die angesetzt auf $\ulcorner T \urcorner$ nicht terminiert.

Satz 4 *Es gibt keine Maschine M die NSA akzeptiert.*

Beweis *Angenommen es gäbe doch so eine Maschine M. Wir fragen uns, was M auf der Eingabe $\ulcorner M \urcorner$ tut. Falls M akzeptiert, müsste $\ulcorner M \urcorner$ zur Sprache NSA gehören, was aber heißen würde, dass M die Eingabe $\ulcorner M \urcorner$ nicht akzeptieren dürfte. Ein Widerspruch.*

Falls andererseits M die Eingabe $\ulcorner M \urcorner$ nicht akzeptiert, so dürfte $\ulcorner M \urcorner$ nicht zur Sprache NSA gehören, was heißen würde, dass M bei der Eingabe $\ulcorner M \urcorner$ halten müsste. Ebenfalls ein Widerspruch.

q.e.d.

Schließlich bemerken wir noch, dass das Komplement von NSA offentsichtlich akzeptiert werden kann. Damit haben wir ein akzeptierbares, aber nicht entscheidbares Prädikat gefunden, nämlich $\mathrm{co}-NSA$, die Sprache derjenigen Turingmaschinen, die ihren eigenen Code akzeptieren.

11.2 Busy Beaver (Friederike Obergfell, Martin Walczack)

Eine weitere unlösbare Berechnungsaufgabe für eine Turingmaschine stellt die Funktion *Busy Beaver* dar.

Definition 1 *Die* Busy Beaver *Funktion an der Stelle n ist die größte von einer Turingmaschine mit n Zuständen, die auf der Eingabe 1^n terminiert, ausgegebene Anzahl von aufeinander folgenden Einsen.*

Satz 5 *Die Busy Beaver Funktion ist nicht durch eine Turingmaschine berechenbar.*

Beweis *Angenommen, die Busy Beaver Funktion $f(n)$ ist durch eine Turingmaschine T_f berechenbar. Dann können wir eine Turingmaschine T konstruieren, welche die Funktion $f(n)+1$ berechnet.*

Die Turingmaschine T habe m Zuständen und gibt damit bei Eingabe m den Funktionswert $f(m)+1$ aus, was einen Widerspruch zur Definition der Busy Beaver Funktion steht.

q.e.d.

12 Die Prädikatenlogik erster Stufe

12.1 Syntax (Benedikt Ewald, Luen To)

Zum Beschreiben der Sprache der Mathematik ist zunächst ein Vokabular von Nöten, das sich aus Relations- und Funktionssymbolen mit verschiedener Stelligkeit zusammensetzt. Die Stelligkeit gibt an, wie viele Argumente erwartet werden. Wenn die Stelligkeit aus dem Zusammenhang klar ist, werden wir darauf verzichten, sie anzugeben. Beispiele von Relationssymbolen sind $=$, $<$, $\leq$ und »Ist prim«. Funktionsymbole sind etwa $+$, $-$ und $\times$.

Terme sind Folgen von Zeichenketten. Als Zeichen stehen dabei Funktionszeichen und Variablen zur Verfügung. Variablen sind $*_0, *_1, *_2, \ldots$ und werden durch $x, y, z, \ldots$ mitgeteilt. Ein Term besteht aus Variablen alleine oder hat die Form $ft_1 \ldots t_n$, wobei f ein n-stelligen Funktionssymbol ist und t_1, …, t_n Terme sind.

Wir verwenden zur Darstellung der Terme in der Logik die *polnische* Notation, die vom polnischen Logiker Lukasiewicz (1878–1956) eingeführt wurde. Dabei stehen die vom Funktionssymbol f^n verlangten n Argumente $a_1, \ldots, a_n$ *hinter* diesem.

Terme über der Signatur $\langle R_1, R_2; +^2, -^1, f^3\rangle$ sind beispielsweise $-^1 +^2 xy$, was in infix-Schreibweise $-(x+y)$ geschrieben wird. Weitere Beispiele für Terme sind $+^2 y +^2 zx$ und $f^3 xyz$.

Atome sind die nächstgrößere Einheit und setzen sich aus Termen und Relationssymbolen zusammen. Sind t_1,…,t_k Terme und ist R ein k-stelliges Relationssymbol, so ist $R^k t_1 ... t_k$ ein Atom.

Formeln werden aus Atomen gebildet. Formeln sind die kleinste Menge, die Atome enthalten und abgeschlossen sind gegenüber der

- Konjunktion $\wedge FG$,
- Disjunktion $\vee FG$,
- Negation $\neg F$,
- universellen Quantifikation $\forall xF$ (sprich »für alle x gilt F«) und der
- existentiellen Quantifikation $\exists xF$ (sprich »es gibt ein x für das F gilt«),

wobei F und G bereits Formeln sind und x für eine Variable steht.

Wir verwenden außerdem die Abkürzung $\forall x < t.A$ für $\forall x \vee \neg < xtA$, um auszudrücken, dass A für alle x kleiner t gilt.

12.2 Semantik (Christoph-Simon Senjak)

Die Tarski-Semantik ist nun eine Möglichkeit, die Bedeutung der syntaktischen Elemente aus dem vorherigen Abschnitt festzulegen.

Dazu muss man zunächst wissen, was die syntaktischen Elemente bedeuten. Man muss sich hier zunächst einmal über den Unterschied zwischen einem syntaktischen Element und seiner Bedeutung klar werden. Beispielsweise ist Pluszeichen »$+$« nur ein syntaktisches Element, ein Zeichen, das ohne genauere Spezifikation keine Bedeutung hat. Erst wenn man festlegt, dass eben dieses »$+$« das Verknüpfungssymbol (also ein zweistelliges Funktionszeichen) für die Addition ist, also für eine zweistellige Funktion, dann bekommt es eine wirkliche Bedeutung. Trotzdem bleibt das »$+$« ein Zeichen, es ist nicht selbst eine zweistellige Funktion.

Es sei A eine Menge von Relationszeichen, Funktionszeichen, logischen Symbolen und Variablennamen, A_r seien ihre von »=« verschiedenen Relationszeichen und A_f ihre Funktionssymbole. Eine Funktion I_f, die den Funktionssymbolen aus A_f Funktionen über einer bestimmten nicht leeren Menge T zuordnet, nennt man Interpretation für Funktionen. Analog gibt es eine Interpretation I_r für Relationen. Die Menge T heißt Träger. Das Tripel $\mathcal{M} = (T; I_r; I_f)$ nennt man nun ein A-*Struktur*, oder kurz *Struktur*. Damit ist allen Relations- und Funktionssymbolen aus A eine Bedeutung zugewiesen. Wir schreiben oft $R^{\mathcal{M}}$ für $I_r(R)$ und $f^{\mathcal{M}}$ für $I_f(f)$.

A enthält aber auch Variablensymbole $*_0, *_1, *_2, \ldots$; die Menge der Variablensymbole sei mit A_v bezeichnet. Man muss nun den Variablen einen konkreten Wert zuweisen. Dies geschieht mit einer *Belegungsfunktion* $\eta\colon A_v \to T$, oder kurz *Belegung*. Diese weist den Variablensymbolen Elemente aus dem Träger zu.

Dass der Träger mindestens ein Element enthalten muss, ist insofern klar, weil man ansonsten keine Belegungsfunktion angeben kann. Wir werden uns weiter unten aber auch mit einer anderen Begründung dafür auseinander setzen.

Zunächst geben wir Termen eine Bedeutung. Sie sollen stets Elemente des Trägers bezeichnen.

Definition 2 *Durch Induktion über den Aufbau des Terms t definieren wir $t^{\mathcal{M}}[\eta]$ wie folgt:*

- $x^{\mathcal{M}}[\eta] = \eta(x)$
- $(ft_1 \ldots t_n)^{\mathcal{M}}[\eta] = f^{\mathcal{M}}(t_1^{\mathcal{M}}[\eta], \ldots, t_n^{\mathcal{M}}[\eta])$

Anhand der Definitionen kann man nun induktiv einen semantischen Wahrheitsbegriff definieren. Dabei soll $\mathcal{M} \models \phi[\eta]$ für »die Aussage, die dem Ausdruck ϕ unter Verwendung der Struktur $\mathcal{M}$ und der Belegung η zugeordnet wird, ist wahr« stehen, man sagt auch »$\mathcal{M}$ ist *Modell* von ϕ unter der Belegung η«.

Definition 3 *Durch Induktion über die Formel ϕ definieren wir $\mathcal{M} \models \phi[\eta]$ wie folgt.*

- $\mathcal{M} \models (= t_1 t_2)[\eta]$ *genau dann, wenn* $t_1^{\mathcal{M}}[\eta]$ *gleich* $t_2^{\mathcal{M}}[\eta]$.
- $\mathcal{M} \models (Rt_1 \ldots t_n)[\eta]$ *genau dann, wenn* $R^{\mathcal{M}}(t_1^{\mathcal{M}}[\eta], \ldots, t_n^{\mathcal{M}}[\eta])$ *gilt.*
- $\mathcal{M} \models (\wedge AB)[\eta]$ *genau dann, wenn* $\mathcal{M} \models A[\eta]$ *und* $\mathcal{M} \models B[\eta]$.
- $\mathcal{M} \models (\vee AB)[\eta]$ *genau dann, wenn* $\mathcal{M} \models A[\eta]$ *oder* $\mathcal{M} \models B[\eta]$.
- $\mathcal{M} \models (\neg A)[\eta]$ *genau dann, wenn nicht* $\mathcal{M} \models A[\eta]$.
- $\mathcal{M} \models (\exists *_k A)[\eta]$ *genau dann, wenn es ein* $q \in T$ *und*

$$\eta'(*_x) = \begin{cases} q \text{ für } x = k \\ \eta(*_x) \text{ sonst} \end{cases}$$

  gibt, so dass gilt $\mathcal{M} \models A[\eta']$.
- $\mathcal{M} \models (\forall *_k A)[\eta]$ *genau dann, wenn für alle* $q \in T$ *und*

$$\eta'(*_x) = \begin{cases} q \text{ für } x = k \\ \eta(*_x) \text{ sonst} \end{cases}$$

  gilt $\mathcal{M} \models A[\eta']$.

Damit ist der semantische Wahrheitsbegriff induktiv definiert. Für viele Formeln gibt es sowohl Strukturen und Belegungen, in denen sie gelten, also auch solche, in denen sie nicht gelten. Ein Beispiel wäre $\forall x \exists y = x \times 2y$. Über dem Träger $\mathbb{N}$ auf $I_f(\times)$ als die übliche Multiplikation ist die Aussage offensichtlich falsch – es ist nicht jede natürliche Zahl durch zwei teilbar. Über dem Träger $\mathbb{Q}$ mit gängiger Muiltiplikation ist der Satz allerdings wahr.

Definition 4 *Ein Ausdruck heißt allgemeingültig, wenn er unter allen Strukturen und Belegungen gilt.*

Ein Beispiel für eine atomare allgemeingültige Aussage wäre $= xx$, diese gilt offensichtlich unter allen Modellen und Belegungen. Gleiches gilt für $\exists x = xx$. Hier wird auch schnell klar, warum man auf die Forderung an den Träger T, nicht leer zu sein, nicht verzichten sollte: $= xx$ sollte eigentlich unabhängig von der Belegung gelten, und selbst wenn man nun auf irgendeine möglichst sinnvolle Weise definieren könnte, wie man x für $T = \{\}$ zu belegen hätte, der Ausdruck $\exists x = xx$ wäre auf jeden Fall falsch. Andererseits wäre $\forall x \forall y = xy$ wiederum wahr, denn für alle $x, y \in \{\}$ gilt ja offensichtlich $x = y$ (da es ja keine solchen Elemente gibt). Derartiges rechtfertigt die Forderung nach mindestens einem Element.

13 Codierung von Folgen

(Lennart Galinat)

Da einige der folgenden Beiträge codierte Listen bzw. Folgen verwenden werden, wird in diesem Kapitel kurz eine Methode zur Codierung von Listen und ihr Beweis dargestellt. Die genaue Form der erhaltenen Formel wird später eine wichtige Rolle spielen; genauer handelt es sich um ein Σ-Formel im Sinnes des nächsten Kapitels.

Satz 6 *Für jede Zahlenfolge $\langle n_0, \ldots, n_k \rangle$ gibt es ein a und b, so dass für $i = 0, 1, \ldots, k$ gilt $n_i = a \bmod (1 + (1 + i) \cdot b)$.*

Dies bedeutet, dass sich jede Folge $\langle n_0 \ldots n_k \rangle$ eindeutig mit drei natürlichen Zahlen k, a, und b codieren lässt.

Beweis *Wir bezeichnen das Maximum aller Listeneinträge und der Länge der Liste als s. Als nächstes setzen wir $b = s!$. Zuerst werden wir zeigen, dass die Zahlen $b_i = 1 + (1 + i) \cdot b$ für $0 \leq i \leq k$ paarweise teilerfremd sind, also dass es keine Primzahl p gibt gibt, die sowohl b_i, als auch b_j teilt, falls $i \neq j$. Hierzu nehmen wir das Gegenteil an: p teile sowohl b_i als auch b_j. Ohne Beschränkung der Allgemeinheit gelte $i < j$. Aus der Annahme folgt, dass p auch $b_j - b_i = (1 + (1 + j) \cdot b) - (1 + (1 + i) \cdot b) = (j - i) \cdot b$ teilt. Da sowohl i als auch j kleiner oder gleich dem Wert von k sind, muss auch die Differenz der beiden kleiner oder gleich k sein. k wiederum ist laut Definition kleiner oder gleich s und somit teilt $(j - i)$ die Zahl $b = s! = 1 \cdot 2 \cdot \ldots \cdot (j - i) \cdot \ldots \cdot s$. Somit geht jeder Primteiler von $(j - i) \cdot b$ also auch in b auf. Die Primzahl p teilt somit sowohl b_i als auch b. Dies führt zu einem Widerspruch, da p somit 1 teilen muss. Somit ist gezeigt, dass zwei beliebige b_i teilerfremd sind. Nun gilt es noch zu zeigen, dass für zwei verschiedene Zahlen a_1 und a_2 mit $0 \leq a_1 < a_2 < b_0 \cdot b_1 \cdot \ldots \cdot b_k$ auch die Lösungen der Restsysteme*

$$\begin{aligned} n_{i1} &= a_1 \bmod b_i \qquad (i = 0, \ldots, k) \\ n_{i2} &= a_2 \bmod b_i \qquad (i = 0, \ldots, k) \end{aligned}$$

unterschiedlich sind. Hierzu nehmen wir an, dass $n_{i1} = n_{i2}$ für alle $i = 0, \ldots, k$. Daraus kann man schließen, dass jedes b_i die Zahl $a_2 - a_1$ teilt, da

$$\begin{aligned} a_2 &= q_{i2} \cdot b_i + n_{i2} \\ a_1 &= q_{i1} \cdot b_i + n_{i1} \end{aligned}$$

und also $a_2 - a_1 = (q_{i2} - q_{i1}) \cdot b_i + \underbrace{n_{i2} - n_{i1}}_{0}$.

Da die b_i paarweise teilerfremd sind, teilt somit auch das Produkt $b_0 \cdot \ldots \cdot b_k$ die Differenz $a_2 - a_1$. Da $a_2 - a_1 < b_0 \cdot \ldots \cdot b_k$ ist dies ein Widerspruch und somit sind die Restsysteme für verschiedene a tatsächlich unterschiedlich.

Die Zahlen a für $0 \leq a < b_0 \cdot \ldots \cdot b_k$ liefern somit $b_0 \cdot \ldots \cdot b_k$ verschiedene Restsysteme; dies bedeutet $b_0 \cdot \ldots \cdot b_k$ viele verschiedene Folgen $\langle n_0, \ldots, n_k \rangle$ mit der Eigenschaft $n_i < b_i$. Damit ist gezeigt, dass jede dieser Zahlenfolgen genau einmal als Lösung eines Restsystems auftritt. Zu einer beliebigen Zahlenfolge wählt man nun b wie oben beschrieben und a so, dass die Folge als Lösung des zu a gehörenden Restsystems auftritt.

q.e.d.

Hiermit haben wir nicht nur gezeigt, dass eine Liste auf die oben beschriebene Art und Weise eindeutig codiert werden kann, sondern auch, dass jedes Gleichungssystem $a_i = a \bmod b_i$ für alle teilerfremden $b_0, \ldots, b_k$ eindeutig für $a < b_0 \cdot \ldots \cdot b_k$ gelöst werden kann, da alle oben beschriebenen Zahlenfolgen $\langle n_0, \ldots, n_k \rangle$ genau ein Restsystem lösen und es genauso viele Restsysteme wie Zahlenfolgenmöglichkeiten für ein festes k gibt. Diese Erkenntnis wird auch als »Chinesischer Restsatz« bezeichnet.

Hierzu noch ein kleines Beispiel, welches die Folge $\langle 1, 2 \rangle$ codiert. Das Maximum von $\{2, 1, 2\}$ ist 2 und somit ist $b = 2! = 2$. Durch eine einfache Rechnung erhält man $b_0 = 3$ und $b_1 = 5$. Die rechts angegebene Liste enthält nun alle Zahlenfolgen für $a = 0$ bis $(b_0 \cdot b_1) - 1$.

Die Kodierung für $\langle 1, 2 \rangle$ ist also $a = 7$, $b = 2$ und $k = 1$.

a	$a \bmod 3$	$a \bmod 5$
0	0	0
1	1	1
2	2	2
⋮	⋮	⋮
6	0	1
7	1	2
8	2	3
⋮	⋮	⋮
13	1	3
14	2	4

Der Zugriff auf das i-te Element n_i einer durch a und b kodierte Liste n in einer Formel φ, kann nun durch die Umformung

$$\varphi(n_i) \equiv \exists z \wedge = z n_i \varphi(z)$$

erreicht werden, wobei der Ausdruck $z = n_i$ durch eine entsprechende Formel für $z = a \bmod (1 + (1 + i) \cdot b)$ zu ersetzten ist.

14 Sigma-Formeln und Berechenbarkeit

(Lennart Galinat, Martin Müller, Robert Pfeiffer, Christoph-Simon Senjak)

In den folgenden Überlegungen sei »$+$« stets die Addition, »$\times$« die Multiplikation und »$<$« die Kleinerrelation über dem fest gewählten Träger $\mathbb{N}$. Dann soll »0« das Zeichen für Null und »1« das Zeichen für Eins sein. Man spricht hier von der Standardstruktur N der Sprache der Peano-Arithmetik.

Wir werden nun den Begriff »berechenbar« als Definierbarkeit in einer gewissen Formelklasse charakterisieren.

Definition 5 *Die Menge der Σ-Formeln ist wie folgt induktiv definiert:*

- *Jeder atomare und jeder negierte atomare Ausdruck, also $= RS$, $\neg = RS$, $< RS$ oder $\neg < RS$ für die Terme r und s, ist eine Σ-Formel.*
- *Sind A und B beide Σ-Formeln, ist auch $\wedge AB$ eine Σ-Formel.*
- *Sind A und B beide Σ-Formeln, ist auch $\vee AB$ eine Σ-Formel.*
- *Ist A eine Σ-Formel, ist auch $\exists x A$ eine Σ-Formel.*
- *Ist A eine Σ-Formel, und kommt x nicht in t vor, so ist auch $\forall x < t.A$ eine Σ-Formel.*

Wir bemerken, dass die Aussage $x = a \bmod b$ sich durch eine Σ-Formal ausdrücken läßt. Dazu sagen wir, dass die ganzzahlige Division existieren soll; genauer ist $x = a \bmod b$ äquivalent zu

$$\exists y \wedge = a + \times y b x < x b$$

Insbesondere kann die erwähnte Codierung von Listen als Σ-Formel ausgedrückt werden.

Im Folgenden wird die Frage nach der Überprüfbarkeit von solchen Formeln behandelt. Man wird zum Ergebnis kommen, dass genau die Aussagen, die der folgenden Definition entsprechen, berechenbar sind.

Definition 6 *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt Σ-definierbar, falls es eine Σ-Formel φ gibt, so dass für alle $n_1, \ldots, n_k$ und m gilt*

$$f(n_1, \ldots, n_k) = m \text{ genau dann wenn } N \models \varphi[\eta^{m\, n_1}_{*_0 *_1} \cdots {}^{n_k}_{*_k}]$$

für alle Belegung η.

Dabei ist

$$\eta^a_x(y) = \begin{cases} a & x = y \\ \eta(y) & \text{sonst} \end{cases}$$

und η^{ab}_{xy} kurz für $(\eta^a_x)^b_y$.

Die rechte Seite der »genau dann wenn«-Aussage kann man auch so verstehen. Ersetzt man die genannten Variablen in φ durch ihre Werte, so ist daraus entstandene Formel in der Standardstruktur wahr.

Im Folgenden zeigen wir, dass Σ-Definierbarkeit und Turingberechenbarkeit gleichwertig sind.

14.1 Sigma-Formeln sind berechenbar

(Lennart Galinat, Martin Müller, Robert Pfeiffer, Christoph-Simon Senjak)

Auch wenn in der Definition der Σ-Berechenbarkeit auf alle Belegungen Bezug genommen wird, ist nur ein endlicher Teil der Belegung ausschlaggebend, da in einer Formel nur endliche viele Variabeln auftreten können.

Angenommen, wir können entscheiden, ob $N \models \varphi[\eta]$ für eine beliebige Belegung η zutrifft oder nicht, so kann man einfach den Funktionswert der durch φ repräsentierte Funktion bestimmen. Man setzt solange alle Zahlen als Funktionswert einsetzt, bis φ unter der entsprechende Belegung wahr wird.

Die Entscheidung, ob $N \models \varphi[\eta]$ gilt, wird mehr oder minder durch die Semantik diktiert. Naiv könnte man für jede Teilformel ein Unterprogramm auf einer Turingmaschine realisieren, die, gegeben den relevanten Teil einer Belegung, entscheidet, ob die Teilformel unter dieser Belegung wahr wird. Für Relationen, Funktionen und Junktoren ist diesen offensichtlich. Auch die Auswertung des beschränkten Allquantors stellt kein Problem dar, da nur endlich viele Zahlen durchlaufen werden müssen. Dies ist beim Existenzquantor nicht unbedingt garantiert. Es könnte beispielsweise folgende Σ-Formel auftreten.

$$\vee\wedge = x_0 0 \exists x_2 = \times 2 x_2 x_1 \wedge = x_0 1 \exists x_3 = + \times 2 x_3 1 x_1$$

Sie besagt, dass $x_0 = x_1 \bmod 2$, also ist die berechnende Funktion $f(n)$ gleich $n \bmod 2$.

In diesem Fall würden wir für die Teilformeln $\exists x_2 = \times 2 x_2 x_1$ und $\exists x_3 = + \times 2 x_3 1 x_1$ versuchen, nacheinander herauszufinden, ob sie unter gegebener Belegung wahr sind. Das Programm könnte aber in eine Endlosschleife geraten, wenn x_1 ungerade ist, da das Programm dann alle möglichen Belegungen von x_2 testet für den Fall, dass x_1 gerade ist.

Um solchen Situationen vorzubeugen, müssen wir alle möglichen Belegungen gleichzeitig durchsuchen. Dazu beschränken wir bei allen Existenzquantoren und bei der Suche nach dem Funktionswert den Suchraum global und erhöhen diesen Schrittweise. Bei Existenzquantoren werden so alle Belegungskombinationen berücksichtigt.

14.2 Darstellung von Registermaschinen durch Sigma-Formeln (Robert Pfeiffer)

Da Registermaschinen, RAMs und Turingmaschinen äquivalent sind, ist deren Darstellung durch Σ-Formeln ebenfalls möglich, der Einfachheit halber wird hier die Registermaschine betrachtet.

Der Zustand der Registermaschine nach jeder Anweisung wird in einer Matrix mit den Ausmaßen $z+1$ und $m+1$ gespeichert, dabei nimmt jeder Zustand eine Zeile ein. Die Matrix kann lediglich endlich viele Spalten haben, da im Programm nur endlich viele Register benötigt werden. Desgleichen kann sie nur endlich viele Zeilen haben, da sonst das Programm nicht terminiert. Abläufe von Registermaschinen, die nach endlicher Zeit ein Ergebnis liefern sind deshalb Σ-definierbar.

$$\begin{pmatrix} n_{00} & \dots & n_{0m} \\ \vdots & \ddots & \vdots \\ n_{z0} & \dots & n_{zm} \end{pmatrix}$$

Das jeweils erste Element n_{s0} jeder Zeile s ist die Zeilennummer der jeweiligen Anweisung. Die folgenden Elemente $n_{s1} \dots n_{sm}$ enthalten die Wertbelegungen der Register; diese seien ohne Einschränkung $R_0 \dots R_{m-1}$. Jede Zeile enthält den Zustand der Registermaschine zu einem Zeitpunkt. Die erste Zeile enthält den Anfangszustand der Maschine mit der Anfangsbelegung der Register. Jede Zeile, mit Ausnahme der ersten, entsteht durch einen Zustadsübergang aus der vorherigen Zeile. Die letzte Zeile enthält die Haltekonfiguraton der Registermaschine und den Ausgabewert. Die ganze Matrix wird mithilfe der bereits vorgestellten Codierung von Listen als natürliche Zahlen dargestellt.

Die entsprechende Σ-Formel beschreibt also, dass es drei natürliche Zahlen gibt, die eine Liste codieren, die in Matrixschreibweise dem Programmablauf entsprechen. Sie ist dann wahr, wenn in der ersten Zeile der Eingabewert steht, in der letzten Zeile der Ausgabewert steht, und jede Zeile durch einen gültigen Zustandsübergang der Registermaschine aus der vorherigen hervorgeht. Alle drei Eigenschaften sind durch Σ-Formeln ausdrückbar. Dabei benötigen wir Allquantoren lediglich, um auszudrücken, dass etwas für alle Zeilen oder alle Register gelten soll; diese Quantoren sind also durch die Abmessungen der Matrix beschränkt.

Damit ist es möglich, alles Akzeptierbare durch eine Σ-Formel darzustellen. Da wir uns bereits überlegt haben, dass sich gültige Σ-Formeln mit Turingmaschinen akzeptieren lassen, sind Σ-Definierbarkeit und Akzeptierbarkeit äquivalent.

15 Alternierungstiefe von Formeln

(Lennart Galinat, Martin Müller, Robert Pfeiffer)

Wir haben gesehen, dass Σ-Formeln gerade den berechenbaren Funktionen entsprechen. Wir betrachten nun kompliziertere Formelklassen.

Definition 7 *Σ_0-Formeln sind Atome, d. h. von der Form $< st$ bzw. $= st$, wobei s und t Terme sind.*

Σ_{n+1}-Formeln sind die kleinste Menge von Formeln, so dass

- *A und $\neg A$ sind Σ_{n+1}-Formeln, wenn A eine Σ_n-Formel ist.*
- *$\wedge AB$ oder $\vee AB$ sind Σ_{n+1}-Formeln, wenn A und B jeweils Σ_{n+1}-Formeln sind.*
- *$\forall x < t.A$ ist Σ_{n+1}-Formel, falls x in dem Term t nicht vorkommt und A eine Σ_{n+1}-Formel ist.*
- *$\exists x A$ ist Σ_{n+1}-Formel, falls A eine Σ_{n+1}-Formel ist.*

Wir bemerken, dass die bereits eingeführten Σ-Formeln genau die Σ_1-Formeln sind.

15.1 Orakel-Turingmaschinen (Lennart Galinat, Martin Müller, Robert Pfeiffer)

Eine Orakel-Turingmaschine T^S unterscheidet sich von einer herkömmlichen Turingmaschine T durch ein weiteres so genanntes Orakel-Band und drei zusätzlichen Zuständen, den Fragezustand und den beiden Antwortzustände »Ja« und »Nein«. Welchselt die Turingmaschine in den Fragezustand, so springt die unmittelbar in einer der beiden Antwortzustände, je nachdem, ob das Wort auf dem Frageband in der Sprache S des Orakels liegt oder nicht. In jedem Fall wird dabei das Frageband wieder gelöscht.

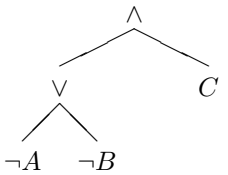
Im Folgenden werden wir Sprachen betrachten, deren Elemente eine erfüllende Belegung für eine fest gewählte Formel darstellen. Also für $S = \{m\$n_1\$ \dots \$n_k \mid N \models \varphi[\eta^{m\,n_1}_{*_0\,*_1} \cdots {}^{n_k}_{*_k}]\}$ für eine Formel φ. Eine solche Turingmaschine schreibt also Belegung der freien Variablen auf das Band und erhält Auskunft darüber, ob für diese Belegung die Formel wahr wird. Da diese Formel ausschließlich Σ_k-Formeln sein werden, verwenden wir die Notation T^{Σ_k}. Die eigentliche Formel ist dabei der Turingmaschine assoziiert wie auch schon ihre Turingtabelle. Wir wollen nun zeigen, dass Turingberechenbarkeit mit einem Σ_k-Orakel gleichbedeutend ist mit Σ_{k+1}-Berechenbarkeit.

15.2 Sigma-Formeln sind berechenbar mit einem Orakel nächstkleinerer Stufe (Lennart Galinat, Martin Müller, Robert Pfeiffer)

Wir zeigen, dass eine Turingmaschine mit einem Σ_n-Orakel eine Σ_{n+1}-Formel akzeptieren kann. Als eine Konsequenz der Definition von Σ_n-Formeln, kann jede Σ_{n+1}-Formel als eine Σ_1-Formel mit Löcher verstanden werden. Durch Einsetzen geeigneter Σ_n-Formeln für diese Löcher erhält man dann die ursprüngliche Formel. Bei der Bestimmung des Wahrheitswerts einer gegeben Σ_{n+1}-Formel gehen wir damit analog vor wie bei den Σ-Formeln zuvor. Nur mit der Ausnahme, dass wir für die Löcher, also für die Σ_n Formel, das Orakel befragen.

Da wir jedoch nur ein einziges Orakel zur Verfügung haben und damit auch nur für eine einzige Σ_n-Formel Fragen stellen dürfen, behelfen wir uns mit einem Trick. Als Orakel verwenden wird die Disjunktion aller Formeln für diese Löcher versehen mit einem Schalter, so dass wir die gewünschte Formel mit jeder Orakelanfrage neu auswählen können.

Beispiel. Sei $\wedge\vee\neg A\neg BC$ eine Σ_{n+1}-Formel und A, B, C jeweils Σ_n-Formeln. Als Orakel definieren wir die Σ_n-Formel $\vee\vee\wedge = z0A\wedge = z1B\wedge = z2C$, wobei der »Schalter« z dem Orakel nur mitteilt, ob A, B oder C überprüft werden soll.



Wir gehen die Σ_{n+1}-Formel argumentweise von außen nach innen durch und überprüfen dann die einzelnen Ausdrücke von innen nach außen. In Baumform sieht dies wie rechts angegeben aus.

Nun nehmen wir eine Turingmaschine mit je einem zusätzlichen Band für A, B und C auf denen das Orakel ausgeben soll, ob die überprüfte Σ_n-Formel wahr oder falsch ist, und einem weiterem Band F, auf welchem der Wert von z gespeichert ist, der dem Orakel übergeben wird. Jetzt muss man nur noch folgende Schritte abarbeiten, wobei die genauere Art und Weise schon aus vorherigen Texten, vor allem aus dem Text über Orakel-Turingmaschinen schon bekannt sein sollten:

1) Null auf Band F schreiben, um vom Orakel den Wahrheitswert von A zu bekommen. Diesen merken wir uns auf Band A.

2) Eins auf Band F schreiben, um vom Orakel den Wahrheitswert von B zu bekommen; wir merken ihn uns auf Band C.
3) Zwei auf Band F schreiben, um vom Orakel den Wahrheitswert von C zu bekommen und auf Band C zu schreiben.
4) Die Negationen von A und B bilden und auf das jeweilige Band schreiben.
5) Die Disjunktion von A und B bilden und auf Band B schreiben.
6) Die Konjunktion der Werte von Band B und C ausgeben.

Auf die jeweilige Σ_{n+1}-Formel angepasst, kann man so alle Σ_{n+1}-Formeln als Turingmaschinen mit Σ_n-Orakel konstruieren.

15.3 Orakel-Turingmaschinen sind durch eine Sigma-Formel nächsthöherer Stufe darstellbar

Analog zu früher stellen wir den Berechnungsfluss durch eine Matrix dar, deren Zeilen je einer Konfiguration entsprechen. Neben dem Zustand und dem Inhalt des Arbeitsbands, beinhaltet die Konfiguration nun auch den Inhalt des Fragebands. Es sind noch die folgenden Informationen in Formeln zu gießen, wobei wir in diesem Abschnitt informell den Formelaufbau infix geschrieben darstellen.

Da die »Eingabe« x_0 der Σ-Formel eine Zahl ist, muss ihre (unäre) Darstellung definiert werden.

$$
\begin{array}{l}
(\text{ »Zelle } 0 \text{ zur Zeit } 0 \text{ enthält das Trennzeichen \$«}) \wedge \\
(\forall 0 < i \leq x_0 \text{ »Zelle } i \text{ zur Zeit } 0 \text{ enthält eine Eins«}) \wedge \\
(\forall x_0 < i \leq s \text{ »Zelle } i \text{ zur Zeit } 0 \text{ enthält eine Null«})
\end{array}
$$

Hierbei ist s der maximal Index der verwendete Zelle. Der Text in Anführungszeichen steht für eine entsprechende quantorenfreie Formel.

Da wir nur an Akzeptanz interessiert sind, reicht es zu sagen, dass die letzte Konfiguration im Haltezustand ist.

Die Konsistenzaussage, dass der Übergang von einer Zeile in die nächste konform mit der Turingtabelle ist, kann offenbar als eine Konjunktion von Wenn-Dann-Aussagen ausgedrückt werden. Beim Erreichen des Fragezustandes steht die Dartstellung(!) der Parameter auf dem Frageband. Die Antwort des Orakels, von der ja der nächste Zustand abhängt, kann durch die zugehörige Σ_n-Formel ausgedrückt werden. Wir drücken die Forderungen, dass bestimmte Antworten bestimmte Folgezustände erzwingen als geeignete Disjunktionen aus. In diesen kommt die Orakelformel einmal negiert vor, aber sonst keine weiteren Negationen oder unbeschränkte universelle Quantifikationen.

Betrachten wir ein Beispiel für eine Orakel für die Σ_n-Formel $\varphi(m, n_1)$ einer einstelligen Funktion.

$$
\begin{array}{l}
\exists m, n_1. \\
\qquad (\forall i < m . F_{i,t} = 1) \\
\qquad (F_{m,t} = 0) \wedge \\
\qquad (\forall m < i \leq m + n_1 . F_{i,t} = 1) \wedge \\
\qquad (\forall m + n_1 < i \leq s . F_{i,t} = 0) \wedge \\
\qquad\quad (\neg\varphi(m, n_1) \vee \text{ »nachfolgender Zustand ist JA und Frageband leer«}) \wedge \\
\qquad\quad (\varphi(m, n_1) \vee \text{ »nachfolgender Zustand ist NEIN und Frageband leer«})
\end{array}
$$

Hierbei steht $F_{i,t}$ für den Inhalt der Zelle i zur Zeit t des Fragebands.

Somit ist die gesamte Formel, die den Lauf einer Turingmaschiene mit einem Σ_n-Orakel codiert, durch eine Σ_{n+1}-Formel darstellbar.

15.4 Universelle Formeln

Die beiden vorangegangen Abschnitte erlauben es uns, aus unserer Konstruktion der universellen Turingmaschine universelle Σ_n-Formeln zu konstruieren.

Wir können die universelle Turingmaschine durch eine Σ_1-Formel beschreiben. Diese Formel beschreibt dann aber auch das Verhalten jeder anderen Turingmaschine, da die universelle dies bei entsprechender Eingabe annimmt. Da jede Σ_1-Formel als Akzeptanz einer geeigneten Turingmaschine ausgedrückt werden kann, können wir auf diesem Weg jede beliebige Σ_1-Formel durch geeignete Parametersetzung aus der Σ_1-Formel gewinnen, welche die universelle Turingmaschine beschreibt. Diese Σ_1-Formel ist also universell.

Da diese eine Σ_1-Formel genau so gut ist wie alle anderen zusammen, können wir unter der Klasse der Turingmaschinen mit Σ_1-Orakel eine universelle finden. Die Konstruktion funktioniert genauso, wie die der universellen Turingmaschine; hinzu kommt lediglich die Anfrage an das Orakel, die wir durch eine Anfrage an das universelle Orakel beantworten können.

Mit dem gleichen Argument wie oben erhalten wir also eine universelle Σ_2-Formel und damit dann eine universelle Turingmaschine mit Σ_2-Orakel, damit wiederum eine universelle Σ_3-Formel und so weiter.

15.5 Halteprobleme für Orakel-Turingmaschinen

(Robert Pfeiffer, Natascha Schiel und Christoph-Simon Senjak)

Definition 8 *Ist M^{Σ_n} eine Turingmaschine mit einem Orakel für eine Σ_n Formel φ, so sei – wie zuvor – $\ulcorner M^{\Sigma_n} \urcorner$ eine Kodierung von M, jedoch zusammen mit der Kodierung der Formel φ. Jede Zeichenkette, die sich bisher noch nicht als $\ulcorner M^{\Sigma_n} \urcorner$ darstellbar ist, sei als Code für diejenige Maschine verstanden, die nichts tut und sofort anhält ohne ihre Eingabe zu betrachten.*

Definition 9 *Das Halteproblem H^{Σ_n} ist die Sprache*

$$\{(\ulcorner M^{\Sigma_n} \urcorner, x) \mid M^{\Sigma_n} \text{ terminiert bei Eingabe } x\}$$

Da sich eine Turingmaschine durch eine Σ_1-Formel darstellen lässt, ist es möglich das Halteproblem für eine beliebige Turingmaschine mithilfe einer Turingmaschine mit Σ_1-Orakel zu beantworten. Umgekehrt lässt sich das Halteproblem für Turingmaschinen mit Σ_n-Orakel durch Turingmaschinen mit Σ_{n+1}-Orakel lösen. Das Halteproblem für Turingmaschinen gleicher Stufe ist jedoch nicht lösbar, analog zum NonSelfAccepting-Problem bei normalen Turingmaschinen.

Satz 7 *Es gibt keine Orakel-Turingmaschine T mit einem Orakel für Σ_n, die für eine Eingabe $(\ulcorner M^{\Sigma_n} \urcorner, x)$ entscheidet, ob die Turingmaschine M^{Σ_n} auf der Eingabe x terminiert. Die Sprache H^{Σ_n} kann also mit keiner Turingmaschine mit einem Orakel für Σ_n entschieden werden.*

Beweis *Angenommen, es gäbe eine solche Orakel-Turingmaschine T^{Σ_n}. Dann terminiert sie insbesondere für jede Eingabe.*

D^{Σ_n} sei eine Turingmaschine bei Eingabe x, zunächst T^{Σ_n} auf der Eingabe x, x ausführt und anschließend genau dann terminiert falls $T^{\Sigma_n}(x, x)$ »nein« geliefert hat.

Nun stellt sich folgender Widerspruch ein.

$$D^{\Sigma_n}(\ulcorner D^{\Sigma_n} \urcorner) \text{ terminiert}$$

genau dann wenn (nach Konstruktion von D)

$$T^{\Sigma_n}(\ulcorner D^{\Sigma_n} \urcorner, \ulcorner D^{\Sigma_n} \urcorner) \text{ »nein« liefert}$$

genau dann wenn (nach Annahme über T)

$$D^{\Sigma_n}(\ulcorner D^{\Sigma_n} \urcorner) \text{ nicht terminiert}$$

q.e.d.

Abschließend bemerken wir noch, dass die Sprache H^{Σ_n} von einer Turingmaschine mit Σ_n-Orakel akzeptiert werden kann. Dies kann leicht mit der universellen Turingmaschine mit Σ_n-Orakel erreicht werden.

16 Komplexitätstheorie

Bisher haben wir uns mit dem Begriff der Berechenbarkeit befaßt. Dabei haben wir keine Einschränkungen an die benötigten Ressourcen wie Zeit und Platz gemacht. Für praktische Anwendungen ist es aber von entscheidender Bedeutung wie lange eine Berechnung dauert. Deshalb wollen wir uns nun theoretisch mit diesem Problem auseinander setzen.

16.1 Linear Speedup Theorem (Nina Berges, Friederike Obergfell, Martin Walczack)

Satz 8 *Für jede Turingmaschine T und jedes $\varepsilon > 0$ lässt sich eine andere Turingmaschine S mit folgenden Eigenschaften angeben:*

- *T and S berechnen die gleiche Funktion bzw. das gleiche Prädikat, also terminiert T insbesondere genau dann, wenn auch S terminiert.*
- *S benötigt höchstens $\varepsilon t + (1+\varepsilon)n + c$ viele Schritte für eine Eingabe der Länge n für die T gerade t viele Schritte benötigt. Dabei ist c eine Konstante, insbesondere unabhängig von n, t und ε.*

Beweis *Die Idee ist, mehrere, etwa m viele Schritte in einem Schritt zu simulieren. Dabei tritt jedoch das Problem auf, dass wir pro Schritt den Kopf nur um ein Zeichen bewegen dürfen. Deswegen fassen wir m Zeichen zu einem Block zusammen.*

Konkret betrachten wir für die zu konstruierenden Maschine S das Alphabet $\Gamma \cup (\Gamma \cup \tilde{\Gamma})^m$, wobei Γ das Alphabet der Maschine T ist und $\tilde{\Gamma} = \{\tilde{x} \mid x \in \Gamma\}$ eine »markierte« Version von Γ ist. Die Markierung soll die Position des Lesekopfes auf dem zu simulierenden Band zeigen. Δ^m ist dabei kurz für das m-fache kartesische Produkt von Δ. Für $\Delta = \{0,1\}$ ist Δ^2 gleich $\{00, 01, 10, 11\}$. Eine Elemente sind als ein einziges Zeichen auf dem Band zu verstehen – genauso wie das »=« auch aus zwei Strichen besteht.

Zuerst liest der Turingmaschine S die Eingabe und schreibt sie komprimiert auf das Arbeitsband. Dies benötigt ungefähr $(1+\varepsilon)n$ Schritte.

Zur Simulation von T auf dem komprimierten Arbeitsband liest S den Block, in dem sich der Lesekopf von T befindet, sowie deren beiden Nachbarn ein und simuliert m Schritte von T, oder bricht früher ab, falls T einen Endzustand erreicht hat. Dabei können die Blöcke nicht verlassen werden. Alsdann werden die veränderten Blöcke samt der neuen Markierung zurückgeschrieben und der Kopf von S entsprechend über dem Block positioniert, den die Markierung für den Kopf von T enthält.

Insgesamt benötigen wir eine feste Anzahl ℓ von Schritten für Lesen, Schreiben und Repositionieren, was m Schritten von T entspricht. Wenn wir also $m \geq \frac{\ell}{\varepsilon}$ wählen, so erhalten wir den gewünschten Speedup.

q.e.d.

16.2 Laufzeitklassen (Natascha Schiel, Christoph-Simon Senjak)

Wir werden die folgenden Konventionen verwenden:

- Die Betrachtung beschränkt sich von nun an auf Turingmaschinen, ohne dass im Weiteren darauf hingewiesen wird.
- Es gibt genau ein beidseitig unbegrenztes Speicherband. Das Alphabet des Speicherbandes enthält mindestens die Zeichen »0«, »1«, »\$« und »$\star$«.
- Die Eingabe steht bei Programmstart hinter dem Schreib- und Lesekopf.
- Eine Eingabe besteht nur aus zusammenhängenden Nullen und Einsen, ansonsten stehen nur $\star$ im Speicherband.
- Die Folge von Nullen und Einsen $a_1 a_2 \dots a_n$ soll die Zahl $\overline{1a_1a_2\dots a_n} - 1$ darstellen, wobei

$$\overline{q_n \cdots q_1 q_0} := \sum_{0 \leq i \leq n} q_i 2^i = q_0 + 2q_1 + \cdots + 2^n q_n.$$

- Wenn wir vom Logarithmus sprechen, meinen wir den binären.

Definition 10 *Die* Laufzeit *ist die Anzahl der Schritte eines terminierenden Programms mit einer bestimmten Eingabe.*

Unter diesem Gesichtspunkt kann man Probleme in verschiedene Klassen einteilen.

Definition 11 *Für jede natürliche Zahl n, ist ihre Länge $|n|$ die Länge ihrer Binärdarstellung, die logarithmisch mit n wächst.*

Da das Linear Speedup Theorems erlaubt, jede Turingmaschine, die mindestens lineare Laufzeit hat, um einen belieben Faktor zu beschleunigen, spielen lineare Terme keine wesentliche Rolle. Daher werden wir $DTime$ entsprechend definieren.

Definition 12 *Die Klasse DTime$_k(f) \subseteq P(\mathbb{N})$ bezeichnet die Mengen derjenigen Sprachen, für die es jeweils Konstanten N und $c > 0$ gibt, so dass diese Sprache von einer k-Band Turingmaschine, die als Eingabe die Binärdarstellung von x erhält, entschieden wird und die für Eingaben aus mindestens N Zeichen in höchstens $c \cdot f(|x|)$ Schritten terminiert. Wir schreiben DTime(f) für DTime$_1(f)$.*

Unmittelbar aus der Definition ergibt sich folgende Eigenschaft.

Satz 9 *Wenn $\left|\frac{f(x)}{g(x)}\right|$ eine obere Schranke besitzt, dann gilt DTime$(f(x)) \subseteq$ DTime$(g(x))$.*

Betrachtet man das Laufzeitverhalten einer oben besprochenen universellen Turingmaschine $UTM(\ulcorner M \urcorner, x)$, sieht man, dass die Laufzeitfunktion von $UTM(\ulcorner M \urcorner, x)$ durch ein Polynom in der Laufzeitfunktion von $M(x)$ beschränkt ist.

Ähnlich verhält es sich mit der Laufzeit bei der Übersetzung der verschieden Berechnungsmodelle ineinander. Dass diese Übersetzung nicht effizienter geschehen kann, zeigen wir später anhand der Palindrome.

Um eine größtmögliche Unabhängigkeit von Berechnungsmodellen und Simulationen zu erreichen, betrachten wir Klassen, deren Laufzeit gegen Polynome abgeschlossen ist. Das natürlichste Beispiel ist polynomielle Zeit.

Definition 13 *Die Laufzeitklasse $\mathcal{P}$ der in polynomieller Zeit entscheidbaren Sprachen ist definiert durch*

$$\mathcal{P} := \bigcup_{p \text{ Polynom}} \mathit{DTime}(p)$$

16.3 Platzklassen (Christoph-Simon Senjak, Nils-Edvin Enkelmann)

Definition 14 *Der* Platz *ist die Anzahl der Zellen die ein bestimmtes Programms mit einer bestimmten Eingabe besucht, bis es terminiert. Unter diesem Gesichtspunkt kann man Probleme ähnlich der Laufzeitklassen einteilen.*

Man kann sich für Platz ein Analogon zum Linear Speedup Theorem überlegen. Dies motiviert den folgenden Begriff.

Definition 15 *Die Klasse DSpace$_k(f) \subseteq P(\mathbb{N})$ bezeichnet die Mengen derjenigen Sprachen, für die es jeweils Konstanten N und $c > 0$ gibt, so dass diese Sprache von einer k-Band Turingmaschine, die als Eingabe die Binärdarstellung von x erhält, entschieden wird und die für Eingaben aus mindestens N Zeichen mit weniger als $c \cdot f(|x|)$ Zellen auskommt. Wir schreiben DSpace(f) für DSpace$_1(f)$.*

Satz 10 *Wenn $\left|\frac{f(x)}{g(x)}\right|$ eine obere Schranke besitzt, dann ist DSpace$(f(x)) \subseteq$ DSpace$(g(x))$.*

Da wir uns später mit sublinearen Platzklassen beschäftigen wollen, macht die bisherige Ein- und Ausgabekonvention wenige Sinn, da z. B. das Eingabeband als zusätzlicher Speicher verwendet werden können und somit eine sublinearen Platzklassen absurd wäre. Also trennen wir die drei Komponenten wie folgt:

- Das Eingabeband ist nicht beschreibbar (aber mehrfach lesbar)
- Das Ausgabeband ist nicht lesbar, und sein Schreibkopf kann nur nach rechts verschoben werden.
- Die Maschine besitzt ein Arbeitsband, das (zunächst) beidseitig unbegrenzt ist, von dem gelesen und geschrieben werden kann. Auf diesem Band wird nun der Platzverbrauch betrachtet, und zu bestimmten Betrachtungen nimmt man dieses Band auch als beidseitig begrenzt an. Dabei bezeichnet die Anzahl an besuchten Zellen dieses Arbeitsbandes den Platzverbrauch.

16.4 Die Platzklasse $\mathcal{L}$

Eine große Bedeutung für spätere Betrachtungen erhält die Laufzeitklasse LogSpace $\mathcal{L}$.

Definition 16 $\mathcal{L} := \mathit{DSpace}(\log)$

Satz 11 *Zu jedem LogSpace-Programm P lässt sich ein Polynom p finden, das seine Laufzeit nach oben begrenzt. In anderen Worten, $\mathcal{L} \subseteq \mathcal{P}$. Insbesondere ist die Länge der Ausgabe ebenfalls durch eine Polynom in der Eingabelänge begrenzt.*

Beweis *Das Programm benutzt nur logarithmisch viele Zellen des Arbeitsbandes in Abhängigkeit der Eingabe x, etwa $k \cdot \log|x|$ Zellen. Auf höchstens $k \cdot \log|x|$ Zellen, die jeweils nur Nullen und Einsen speichern können, kann man höchstens $2^{k \cdot \log|x|} = |x|^k$ verschiedene Wörter speichern. Insbesondere gibt es also auch nur polynomiell viele Konfigurationen. Jede Konfiguration kann in einer terminierenden Berechnung aber nur einmal betreten werden, da sich sonst eine Endlosschleife ergibt.*

q.e.d.

17 Hierarchiesätze

Mehr von einer Ressource zu erlauben, erlaubt ein mehr an durchführbaren Berechnungen. Dies formal zu zeigen ist Inhalt dieses Kapitels.

17.1 Mehr Zeit ist besser (Martin Müller, Christoph-Simon Senjak)

Ausgangspunkt ist eine revidierte Version des Halteproblem, nun aber mit einer Einschränkung an die Zeit.

Definition 17 $H_f := \{\ulcorner X \urcorner \$ w \mid X(w)$ *akzeptiert nach höchstens* $f(|w|)$ *Schritten*$\}$

Satz 12 *Es gibt keine Turingmaschine, die H_f in maximal $f(\lfloor \frac{n}{2} \rfloor)$ Schritten entscheidet.*

Beweis *Angenommen, es gäbe ein Turingprogramm M, das für alle Eingaben $\ulcorner X \urcorner \$ w$ mit einer Schrittzahl von höchstens $f(\left\lfloor \frac{|\ulcorner X \urcorner \$ w|}{2} \right\rfloor)$ entscheidet, ob $X(w)$ in $f(|w|)$ Schritten terminiert. Daraus konstruieren wir das Programm*

$$D_f(x) = \begin{cases} \text{«ja«} & \text{falls } M(x\$x) = \text{nein} \\ \text{«nein«} & \text{falls } M(x\$x) = \text{ja} \end{cases}$$

welches in höchstens $f(\left\lfloor \frac{|x|+1+|x|}{2} \right\rfloor) = f(|x|)$ Schritten terminiert. Gilt nun $D_f(\ulcorner D_f \urcorner)$?

$$D_f(\ulcorner D_f \urcorner) \text{ akzeptiert in höchstens } f\left(\left|\ulcorner D_f \urcorner\right|\right) \text{ Schritten}$$

genau dann, wenn (nach Konstruktion von D_f)

$$M(\ulcorner D_f \urcorner \$ \ulcorner D_f \urcorner) \text{ «nein« liefert}$$

genau dann, wenn (nach Annahme über M)

$$D_f(\ulcorner D_f \urcorner) \text{ nicht in höchstens } f(\left|\ulcorner D_f \urcorner\right|) \text{ Schritten akzeptiert .}$$

Dies ist aber ein Widerspruch.

q.e.d.

Sei s der Grad des Polynoms, das den Laufzeitzuwachs bei der Simulation einer Turingmaschine durch eine universelle Turingmaschine festlegt, so erhalten wir aus obigem Satz für $1 \leq k \in \mathbb{N}$

$$\mathsf{DTime}(n^k) \subsetneq \mathsf{DTime}(n^{ks})$$

Die Inklusion ist nach Definition klar. Die Echtheit der Inklusion wird durch H_{n^k} bezeugt. Offenbar ist dieses Problem in Zeit linear in n^{ks} entscheidbar. Angenommen es gäbe eine Turingmaschine T die das Problem in maximal cn^k vielen Schritten entscheidet. So kann nach dem Linear Speedup Theorem eine äquivalente Maschine T' angeben werden, die in Zeit $\frac{n^k}{2^k}$ arbeitet für hinreichend große n. Dies steht aber im Widerspruch zu obigem Satz.

17.2 Mehr Platz ist besser (Christine Schmidt)

Im Folgenden wollen wir zeigen, dass es Sprachen gibt, welche nur von Turingmaschinen mit einer gewissen Mindestzahl an Zellen akzeptiert werden können. Dazu benötigen wir zunächst einen weiteren Begriff.

Definition 18 *Eine Funktion f heißt* platzkonstruierbar*, falls es eine deterministische Turingmaschine gibt, die auf eine unäre(!) Eingabe n genau $f(n)$ Felder auf dem Arbeitsband markiert, dann hält und bei der Berechnung nur den markierten Platz verwendet.*

Satz 13 *Falls $s(n)$ platzkonstruierbar ist, $s'(n) \geq \log n$ und $\lim_{n\to\infty} s'(n)/s(n) = 0$ gilt, dann gilt*

$$\mathit{DSpace}(s'(n)) \subsetneq \mathit{DSpace}(s(n))$$

Beweis *Ohne Beschränkung der Allgemeinheit betrachten wir Turingmaschinen über dem Alphabet $\Gamma \supseteq \{0,1\}$. Wir verwenden eine Diagonalisierungsmaschine D die auf der Eingabe w der Länge n wie folgt arbeitet:*

- *Überprüfung, ob die Eingabe die Form $0\ldots01^{\ulcorner}T^{\urcorner}$ für eine Turingmaschine T hat. Falls nicht, verwerfe.*
- *Berechnung von $s(n)$ und Begrenzung des Bandes auf den Bereich von $-s(n)$ bis $+s(n)$, durch die Setzung von eindeutigen Abgrenzungszeichen.*
- *Wir simulieren T auf $^{\ulcorner}T^{\urcorner}$ für höchstens $|^{\ulcorner}T^{\urcorner}| \cdot |\Gamma|^{s(n)}$ Schritte.*
- *Falls dabei eines der Begrenzungszeichen erreicht wird oder nicht in der vorgegebenen Zeit angehalten wird, so akzeptieren wir die Eingabe; ebenso akzeptieren wir die Eingabe, wenn T das Wort $^{\ulcorner}T^{\urcorner}$ verwerfen würde. Andernfalls verwerfen wir die Eingabe.*

Wir bezeichnen die Sprache der Diagonalisierungsmaschine D mit L. Es gilt $L \in \mathit{DSpace}(s(n))$, da D maximal den Platz

$$2 \cdot s(n) + \log(|^{\ulcorner}T^{\urcorner}| \cdot |\Gamma|^{s(n)}) = 2 \cdot s(n) + \log(|^{\ulcorner}T^{\urcorner}|) + s(n) \cdot \log(|\Gamma|) \leq Cs(n)$$

für große n und $C = 3 + \log(|\Gamma|)$, da $\log(|^{\ulcorner}T^{\urcorner}|) \leq \log n \leq s'(n) \leq s(n)$ schließlich.

Nun zeigen wir indirekt, dass $L \notin \mathit{DSpace}(s'(n))$ gilt. Wir nehmen daher an, dass $L \in \mathit{DSpace}(s'(n))$ gilt. Das würde also bedeuten, dass L die Sprache einer $s'(n)$-platzbeschränkten Turingmaschine T ist.

Sei $n_0 > |^{\ulcorner}T^{\urcorner}|+2$ so groß, dass $s'(n_0) < s(n_0)$. Wir betrachten nun das Wort $w_T = 0...01^{\ulcorner}T^{\urcorner}$ mit $|w_T| = n_0$. Nun setzen wir die Turingmaschine T auf die Eingabe w_T an. Da T und D nach Definition die gleichen Sprachen besitzen, akzeptiert T das Wort w_T genau dann, wenn D es tut. Nach Konstruktion von D ist dies genau dann der Fall, wenn T das Wort w_T verwerfen würde. In der Tat, da T nur den Platz $s'(n_0) < s(n_0)$ benötigt, tritt ein Überschreiten der Platzbeschränkung nie auf. Die Wahl der zu simulierenden Schritte ist(!) so groß gewählt, dass ein Überschreiten der Zeitbeschränkung nur bei nicht terminierenden Programmen auftreten kann.

q.e.d.

17.3 Palindrome (Robert Pfeiffer, Christoph-Simon Senjak)

Definition 19 *Ein Palindrom ist eine Zeichenkette, die vorwärts und rückwärts gelesen gleich ist.*

Beispiele für Palindrome sind »TOT«, »RELIEFPFEILER« und »EINNEGERMITGAZELLEZAGTIMREGENNIE«.

Wir stellen uns nun die Frage danach, zu welcher Laufzeitklasse die Sprache der Palindrome gehört.

Satz 14 *Die Sprache der Palindrome liegt in $DTime_1(n^2)$.*

Beweis *Eine Turingmaschine die Palindrome erkennt, könnte wie folgt beschrieben werden:*

- *Lese Zeichen und überschreibe es mit einem Begrenzungszeichen,*
- *gehe zum Zeichen vor dem rechten Begrenzungszeichen,*
- *sind beide verschieden, verwerfe Eingabe, ansonsten mache weiter,*
- *überschreibe soeben gelesenen Zeichen durch eine Begrenzungszeichen,*
- *gehe wieder nach links zum ersten Zeichen nach dem linken Begrenzungszeichen*
- *wiederholde dies, bis nur noch maximal ein Zeichen zwischen den Begrenzungszeichen steht. In diesem Fall akzeptiere.*

Für eine Eingabe der Länge n benötigt diese Maschine also maximal ein Vielfaches von

$$\sum_{1 \leq k \leq \lceil n/2 \rceil} k$$

Schritte. Diese arithmetische Reihe ist quadratisch in n.

q.e.d.

Wir wollen nun folgen Satz beweisen.

Satz 15 *Jede 1-Band Turingmaschine benötigt mindestens quadratisch viele Schritte.*

Doch dazu brauchen wir erst noch ein paar Hilfsmittel.

Definition 20 *Sei M eine 1-Band Turingmaschine und $i \in \mathbb{N}$. Die* Crossing Sequence *$CS(X, i)$ ist die Folge von Zuständen (in chronologischer Reihenfolge) mit denen M bei Eingabe X zwischen den Zellen i und $i+1$ wechselt.*

Satz 16 *Seien X_1X_2 und Y_1Y_2 Eingaben für eine Turingmaschine M mit $CS(X_1X_2, |X_1|) = CS(Y_1Y_2, |Y_1|)$, so akzeptiert M das Wort X_1X_2 genau dann wenn sie auch X_1Y_2 akzeptiert.*

Beweis *Das Verhalten von M beim Übergang von i nach $i+1$ bzw. anders herum abhängt nur vom Zustand ab. Da die Crossing Sequences gleich sind, sind die Teilläufe auf der linken Seite bei der Eingabe X_1X_2 konform mit den Teilläufen auf der rechten Seite für die Eingabe Y_1Y_2. Also können diese Teilläufe zu einem Ablauf für die Eingabe X_1Y_2 zusammengebaut werden.*

q.e.d.

Satz 17 *Jede 1-Band Turingmaschine benötigt mindestens quadratisch viele Schritte.*

Beweis *Sei M eine Turingmaschine mit $Q-1$ vielen Zuständen, welche die Sprache der Palindrome entscheidet.*

Um die untere Schranke zu zeigen, reicht es offenbar diese für eine unendliche Teilmenge der Palindromsprache zu zeigen. Wir beschränken uns daher auf Palindrome der Form $u1^{2n}\tilde{u}$ für ein u der Länge $n \geq 5$, dabei ist $\tilde{u}$ die rückwärts gelesenen Version von u.

Stellen echt zwischen dem n-ten und $2n$-ten Zeichen nennen wir im Mittelfeld*. Wegen dem vorigen Lemma müssen $CS(u1^{2n}\tilde{u}, i)$ und $CS(u'1^{2n}\tilde{u'}, i)$ verschieden sein, für alle i im Mittelfeld und alle $u \neq u'$ der Länge n. Ansonsten würde M auch $u1^{2n}\tilde{u'}$ versehentlich als Palindrom erkennen.*

Eine Crossing-Sequence einer Länge maximal

$$k := \frac{n}{2 \log Q}$$

nennen wir kurz*, andernfalls nennen wir sie* lang*.*

Sei i im Mittelfeld. Die Anzahl der Wörter u, so dass $u1^{2n}\tilde{u}$ bei i eine kurze Crossing-Sequence aufweist, kann nach oben abgeschätzt werden durch

$$Q^k = 2^{k \log Q} = \sqrt{2^n}\,.$$

Hierbei haben wir Crossings-Sequences der Länge echt kleiner k als Crossings-Sequences der Länge genau k aufgefasst, indem wir einen Dummy-Zustand hinzugefügt haben – für die Überquerungen, die nicht mehr stattgefunden haben.

Die Anzahl der Wörter, die irgendwo im Mittelfeld eine kurze Crossing-Sequence haben, ist somit kleiner als

$$n\sqrt{2^n} < 2^n\,,$$

da $n \geq 5$.

Somit muss es ein Wort u geben, das an jeder Stelle im Mittelfeld eine lange Crossing-Sequence hat. Die Akzeptanz des Palindroms $u1^{2n}\tilde{u}$ benötigt also mindestens

$$nk = \frac{n^2}{2\log Q}$$

viele Schritte.

q.e.d.

Jedoch gilt diese Aussage nicht mehr für 2-Band Turingmaschinen.

Satz 18 *Die Sprache der Palindrome liegt in $DTime_2(n)$.*

Beweis *Kopiere die Eingabe in umgekehrter Reihenfolge auf das Arbeitsband, vergleiche nun simultan die Eingabe mit dem Arbeitsband und akzeptiere genau dann wenn beide gleich sind.*

q.e.d.

18 Nichtdeterministische Turingmaschinen

(Natascha Schiel, Christoph-Simon Senjak)

In den vergangenen Kapiteln wurden Berechnungsmodelle betrachtet, die auf eine bestimmte Eingabe stets auf die gleiche Weise reagieren, und selbst deterministische Lösungsverfahren darstellen. Im Folgenden werden wir diese Vorstellung erweitern.

Hierzu definiert man die so genannten nicht-deterministischen Turingmaschinen. Sie entstehen aus den Turingmaschinen durch Hinzunahme eines zusätzlichen Speicherbandes, dem sogenannten nicht-deterministischen Band. Auf dieses kann die Maschine nur lesend zugreifen kann und den Lesekopf nur in eine Richtung (im Folgenden rechts) Bewegen. Auf dem nicht-deterministischen Band steht ein »potentieller Beweis« dafür das die Eingabe zur Sprache der Maschine gehört. Die Überprüfung dieses Beweises erfolgt in der gewohnten deterministischen Weise. Man reduziert also die Lösung eines Problems auf die Verifikation eines Beweises. Genauer gehört ein Wort x zur Sprache einer nicht-deterministischen Turingmaschine T, wenn es ein Wort w gibt, so dass $T(x, w)$ akzeptiert.

Betrachten wir beispielsweise die Verifikation von nicht-Primzahlen. Im deterministischen Berechnungsmodell muss man (naiv) solange nach Teilern suchen (abgesehen von eventuell schnelleren Verfahren – aber es geht hier nur um das Prinzip) bis man einen Teiler gefunden hat – dies impliziert meist, dass man mehrere Teiler durchprobieren und verifizieren muss. Im nichtdeterministischen Modell hingegen reicht es wenn auf dem nicht-deterministischen Band ein Teiler dieser Zahl ist. Man muss dann also nur einen Teiler verifizieren.

Es sollen im Folgenden sowohl die deterministischen Turingmaschinen, als auch die nicht-deterministischen Turingmaschinen betrachtet werden.

Für die nicht-deterministischen Turingmaschinen gelten die folgenden Vereinbarungen:

- Es gibt genau ein beidseitig unbegrenztes, les- und beschreibbares Arbeitsband. Das Alphabet dieses Speicherbandes besteht aus den Zeichen »0«, »1« und »⋆«, ihm wird die Eingabe übergeben.
- Es gibt genau ein nicht-deterministisches Speicherband, auf das das Programm nur Lesezugriff hat, dessen Alphabet aber beliebig gewählt werden kann.

Analog kann man die Klasse der Sprachen, die in polynomialer Zeit mit einer nicht-deterministischen Turingmaschine entschieden werden können, definieren.

Definition 21 *Mit $\mathcal{NP}$ bezeichnet man die Menge aller derjenigen Sprachen L, so dass es eine deterministische Turingmaschine T mit zwei Eingabebändern gibt, deren Laufzeit polynomial durch die Länge des ersten Eingabebandes beschränkt ist und für die gilt*

$$L = \{x \mid \text{ es gibt ein } w \text{ und } T(x, w) \text{ akzeptiert}\}.$$

Offenbar gilt damit $\mathcal{P} \subseteq \mathcal{NP}$.

Ähnlich zur deterministische Klasse $\mathcal{L}$ sind, müssen für eine sinnvolle Definition bei $\mathcal{NL}$ einige Einschränkungen hinnehmen.

Definition 22 *Die Klasse $\mathcal{NL}$ besteht aus den Sprachen von nicht-deterministischen Turingmaschinen mit folgenden Einschränkungen.*

- *Eingabe erfolgt über ein separates Band, von dem nur gelesen werden kann.*
- *Das nicht-deterministische Band kann nur sequentiell gelesen werden. Ein Zurückgehen oder Schreiben ist nicht möglich.*
- *Ausgabe erfolgt auf einem separaten Band, auf das nur fortlaufend geschrieben werden werden kann.*
- *Auf dem Arbeitsband werden nur logarithmisch viele Zelle genutzt.*

Eine ähnliche Einschränkung für $\mathcal{NP}$ liefert die gleiche Menge von Sprachen. Diese Einschränkungen für $\mathcal{NL}$ verhindern, dass durch die Position des Lesekopfes auf dem nicht-deterministischen Band zusätzliche Information codiert wird und, wichtiger, dass das nicht-deterministische Band durch Aufzeigen sukzessiver Konfigurationen eines erfolgreichen Laufs den zur Verfügung stehenden Platz de facto erweitert.

19 Vollständigkeit bezüglich einer Klasse von Sprachen

(Nils-Edvin Enkelmann, Martin Müller, Christoph-Simon Senjak)

Wir wollen im Folgenden einen Begriff erarbeiten, der unseren intuitiven Vorstellungen über die Komplexität von Berechnungen möglichst nahe kommt. Wir stellen uns zum Beispiel die Frage nach einer sinnvollen Definition einer Vergleichsrelation bezüglich der Komplexität eines Programms.

Definition 23 *Seien L und M Sprachen. Dann bedeute $L \leq_l^m M$ (»L ist log-space many-one reducible auf M«), dass es eine Turingmaschine R mit logarithmischen Platzverbrauch gibt, so dass*

$$w \in L \Leftrightarrow R(w) \in M$$

Wir sagen also, dass eine Sprache sich auf eine andere zurückführen lässt, wenn man von der einen zur anderen durch ein log-space Programm gelangt.

Satz 19 *Die Relation $\leq_l^m$ ist transitiv.*

Beweis *Gelte $A \leq_l^m B$ und $B \leq_l^m C$. Wir müssen zeigen, dass dann auch $A \leq_l^m C$ gilt. Seien nun $P(x)$ und $Q(x)$ LogSpace-Programme, für welche $B(P(x)) \leftrightarrow A(x)$ bzw. $C(Q(x)) \leftrightarrow B(x)$ gilt. Es genügt also zu beweisen, dass es ein log-space Programm gibt, das $P(Q(x))$ berechnet. Da P auf die Ausgabe nur schreibend zugreifen und den Schreibkopf nur nach rechts bewegen kann, lässt sich daraus ein Programm P' konstruieren, das den Wert der n-ten ausgegebenen Zelle berechnet, indem man alle Schreibzugriffe in einem dualen Zähler zählt. Der duale Zähler braucht nur logarithmisch viel Platz, denn ein log-space Programm kann höchstens eine polynominelle Ausgabe haben und der Logarithmus eines Polynoms ist wiederum logarithmisch, da $\log n^a = a \log n$. Zur Abarbeitung von Q muss man nun lediglich dafür sorgen, dass, wenn Q die Ausgabe von P an einer bestimmten Stelle liest, P' entsprechend lange ausgeführt wird. Fügt man nun P', Q und den Zähler in geeigneter Weise zusammen, so erhält man ein Programm, das für die folgenden Prozesse Platz benötigt:*

- *Eine Zelle für die Speicherung des aktuellen Rückgabewertes von P',*
- *jeweils ein logarithmisch von der Eingabe abhängendes Teilband zur Berechnung von P' bzw. Q, und*
- *ein logarithmisch von der Eingabe abhängendes Zählwerk.*

Damit erhält man ein log-space Programm, das $P(Q(x))$ berechnet.

q.e.d.

Definition 24 *Sei $\mathcal{C}$ eine Menge von Sprachen und sei $L \in \mathcal{C}$. Dann heißt L bezüglich $\mathcal{C}$ vollständig, wenn für alle $M \in \mathcal{C}$ gilt $M \leq_l^m L$.*

20 Das Erfüllbarkeitsproblem ist NP-vollständig

(Natascha Schiel, Cornelia Strauß)

Die Menge der aussagenlogischen Formeln ist die kleinste Menge, die nur nullstellige Relationen (in unserem Fall Variablen für wahr oder falsch) enthält und abgeschlossen ist gegen Konjunktion, Disjunktion und Negation. Eine Formel heißt erfüllbar, wenn es ein Modell für sie gibt.

SAT (satisfyability) ist die Sprache der erfüllbaren Formeln.

Beispiele: Seien F und G Aussagenvariablen. Dann sind F, $\vee F \neg F$ und $\vee F \wedge \neg F G$ erfüllbare Formeln. $\wedge F \neg F$ ist hingegen nicht erfüllbar.

Satz 20 $SAT \in \mathcal{NP}$.

Der Satz besagt, dass es eine deterministische polynomailzeit Turingmaschine R gibt, so dass $\exists w.R(x,w)$ genau dann zutrifft wenn $x \in SAT$. Dabei kann ein geeignetes w als Zeuge oder Beweis verstanden werden, dass x erfüllbar ist.

Beweis *Sei n die Länge der Eingabe. Im ersten Schritt werde überprüft, ob die Eingabe wohlgeformt ist, also ob es sich überhaupt um eine Formel handelt. Ist dies nicht der Fall wird die Eingabe verworfen.*

Der Inhalt des nicht-deterministische Band wird sodann als eine Abbildung der in der Eingabe vorkommenden Variablen nach »falsch« oder »wahr« verstanden. Wir müssen nun kanonisch für jede Variable den gegeben Wahrheitswert einsetzen und anschließend überprüfen, ob es sich um eine wahre Aussage handelt. Die Auswertung kann mit einer mehrbandigen Turingmaschine leicht in polynomieller Zeit erfolgen. Daher erfordert die gesamte Erkennung von SAT durch eine mehrbandige nicht-deterministische Turingmaschine mit polynominellem Zeitaufwand.

q.e.d.

Satz 21 $SAT \in DTime(n^k 2^n)$ *für ein* $k \in \mathbb{N}$.

Beweis *Eine deterministische Turingmaschine kann bei einer Eingabe der Länge n jeder der maximal 2^n möglichen Belegungen durchgehen und für jeder dieser in polynomineller Zeit überprüfen, ob diese die Formel erfüllt.*

q.e.d.

Satz 22 SAT *ist $\mathcal{NP}$-vollständig.*

Beweis *Nach dem obigen Satz reicht es zu zeigen, dass es für jede Sprache L aus $\mathcal{NP}$ eine log-space Reduktion auf SAT gibt.*

Wir können annehmen, dass eine nicht-deterministische Turingmaschine T für L und ein polynominales $p(n)$ existieren, so dass bei einer Eingabe der Länge n nie mehr als $p(n)$ Schritte benötigt werden; insbesondere ist die Anzahl der Zellen des nicht-deterministischen Bandes, die eingesehen werden durch $p(n)$ beschränkt. Wir stellen den Algorithmus beim Erkennen einer Formel in SAT als eine Formel dar. Die Formel ist die Konjunktion von Anfangskonfiguration (Anfangszustand und Eingabewort als Bandinhalt), den Übergängen zwischen den Konfigurationen und dem Halten in einem akzeptierenden Endzustand.

Dann lässt sich der Programmablauf als ein Rechteckschema mit den Achsen »Zeit« und »Platz« darstellen. Für jeden Zeitpunkt verwenden wir Variablen für den Ort des Lesekopfes, das Symbol in einer bestimmten Position und für den Zustand. Das Lesen, Schreiben, Verändern des Zustandes und Verschieben des Lesekopfes lassen sich nun als Formeln ausdrücken. In der letzten Konfiguration muß der Endzustand erreicht werden.

q.e.d.

21 Abgeschlossenheit von $\mathcal{NL}$ unter dem Komplement

(Nils-Edvin Enkelmann)

Wir wollen beweisen, dass alle Sprachen, welche in $\mathcal{NL}$ (nichtdeterministisch und logarithmischer Platzverbrauch) liegen, auch ihr Komplement in $\mathcal{NL}$ liegt. Dazu reicht es, wenn wir für ein $\mathcal{NL}$-vollständiges Problem zeigen, dass ihre Negation in $\mathcal{NL}$ liegt. Ein solches Problem ist z. B. die Fragestellung der »Directed Graph Accessability« (kurz DGA = Erreichbarkeit in gerichteten Graphen).

Definition 25 Directed Graph Accessibility *ist das Problem, für einen gegebenen gerichteten Graph zu entscheiden, ob es einen Weg von einem gegebenen Knoten A zu einem gegebenen Knoten B gibt.*

Satz 23 *DGA ist $\mathcal{NL}$-vollständig.*

Beweis *Zuerst zeigen wir, dass es eine nicht-deterministische Maschine mit logarithmischen Platzverbrauch gibt, welche die Erreichbarkeit überprüfen kann. Dazu lassen wir uns vom nicht-deterministischen Band einen Weg von A nach B angeben. Die Überprüfung erfolgt, indem wir immer zwei aufeinanderfolgende Punkte des Weges auf das Arbeitsband kopieren und im Eingabeband, welches den kompletten Graphen enthält, nach dem ersten Punkt suchen und dann schauen, ob dieser wirklich eine Verbindung zum zweiten Punkt besitzt. Nebenher zählt man auf dem Arbeitsband noch die Länge des Weges mit, denn spätestens wenn die Länge des Weges gleich der Anzahl der Punkte im Graphen entspricht, weiß man, dass der angegebene Weg nicht richtig ist.*

Jetzt wird noch gezeigt, dass dieses Problem mindestens genauso schwer ist wie jedes andere Problem in $\mathcal{NL}$ ist. Dies zeigt man, indem man sich für jedes Programm einen Graphen denkt, wobei jeder Punkt des Graphen eine potentielle Konfiguration der Maschine darstellt. Da eine Konfiguration durch logarithmisch viele Zeichen beschrieben werden kann, wissen wir, dass der Graph polynomielle Größe hat und in $\mathcal{L}$ bestimmt werden kann. Außerdem setzen wir ohne Einschränkung voraus, dass die Maschine nach der Lösungsfindung ihr Arbeitsband löscht, da es dadurch nur einen zu betrachteten Endzustand gibt.

q.e.d.

Satz 24 *Das Komplement der DGA, also die Nichterreichbarkeit in gerichteten Graphen, liegt auch in $\mathcal{NL}$.*

Beweis *Die Idee geht wie folgt: Wir lassen uns vom nicht-deterministischen Band nacheinander beweisen, wie viele und welche Punkte in höchstens k Schritten erreichbar sind. Sobald k den gleichen Wert hat wie die Anzahl der Punkte im Graph beträgt, wissen wir, dass es keinen Weg von A nach B gibt, wenn in den in k Schritten erreichbaren Punkten B nicht enthalten ist.*

Die Maschine handelt folgendermaßen: Zuerst lässt sie sich vom nicht-deterministischen Band die Anzahl der Punkte, die in k Schritten erreichbar sind, angeben. Danach geht sie jeden einzelnen Punkt des Graphen durch und fragt, ob der entsprechende Punkt in k Schritten erreichbar ist. Wenn ja, so lässt sich die Maschine dies durch die Angabe eines gültigen Weges zu dem Punkt beweisen, wenn nein, so lässt sich die Maschine noch einmal alle Punkte, die in $k-1$ Schritten erreichbar sind (mit Weg als Beweis) angeben und überprüft selbst, ob diese eine Verbindung zum entsprechenden Punkt haben. Damit das nicht-deterministische Band beim Aufzählen von Punkten keine Punkte doppelt nennen kann, müssen alle Punkte z. B. lexikographisch aufgezählt werden. Sind alle Punkte des Graphen überprüft worden und stimmt die vom nicht-deterministischen Band angegebene Anzahl an erreichbaren Punkten mit der mitgezählten überein, so wird k um 1 erhöht und das Ganze beginnt von vorne. Für diese Vorgehensweise muss die Maschine eine feste Anzahl an Zahlen und Punkten zwischenspeichern, wobei jede Zahl und jeder Punkt nur logarithmischen Platzbedarf hat, weswegen die Maschine auch nur logarithmisch viel Platz verbraucht.

q.e.d.

22 Tales of Define and Conquer

22.1 Was passiert, wenn Informatiker ohne Computer. . . (Luen To, Martin Walczack)

Es war einmal vor langer, langer Zeit in einem Dörfchen nahe Düsseldorf (definiere »Dörfchen« als »Hilden«). . . als sich eine wagemutige Gemeinschaft aus 15 Irr. . . äh, Schergen fand, die sich in die trockenen Gefilde der theoretischen Informatik und Mathematik wagte. Doch nicht nur diese Inhalte, nein, auch neue revolutionäre Lernmethoden fanden hier Verwendung. So etwa der allmorgendliche »Dual-Frontalunterricht«

($\forall\ Raeume : vorne \neq hinten \wedge \neg\exists\ hinten$, sprich $vorne = ueberall$), der sie mehr oder minder mit den nötigen Waffen (Kreide in mindestens 20 Farben, Fotoapparat, zig Tafeln Schokolade, Schwamm und Wischer) ausrüstete. Dermaßen präpariert stürzten sie sich in kleinen Trupps auf die unterschiedlichsten Problemstellungen. Nachdem diese niedergerungen wurden, ging es hochmotiviert zum Mittagessen. Gestärkt und ausgeruht nach einem kurzen Verdauungsschläfchen (wichtige Lernmethode, nicht verzichtbar!) wurden des Abends die morgendlichen Heldentaten im Plenum den anderen Mitstreitern verkündet und erläutert. Unterbrochen wurde diese Routine nur von der zeitweiligen Aufstockung des mathematischen Waffenarsenals in Form des bereits bekannten »Dual-Frontalunterrichts«. So verbrachten diese tapferen Recken ihre zweieinhalb Wochen in der Akademie und handelten stets nach dem Motto, für das sie standen:

»Define & Conquer«

22.2 Tafelphilosophen (Alexander Mänz, Natascha Schiel)

Die Tafel, das Schulsymbol schlechthin, findet hin und wieder auch Verwendung in Kursen der DSA. Doch ihre Funktion kann hier völlig neue Dimensionen annehmen. Ja, sie dient auch zur Erläuterung und Konkretisierung von dem zu vermittelnden Lehrstoff, doch sobald diese ursprünglichen Aufgaben der Tafel ausgeschöpft sind, erlebt man hier eine neuartige Mischung von Kunst, Pädagogik, Logik und Logistik. Tatsachen vorweg: Eine normale Schultafel ist schlicht und einfach zu wenig. Um dem Künstler (im folgenden »Gestalter« genannt) genügend Freiraum für seinen geistigen und künstlerischen Exkurs zu verschaffen, benötigt man mindestens 10 m^2 beschreibbaren Untergrund. Fenster und Möbel inklusive. Obendrein sollten möglichst viele dieser Beschriftungsmöglichkeiten mobil sein, wenn möglich gebäude- und geländeübergreifend, damit der Tafelkünstler sich beim Erstellen seines Kunstwerkes von seiner Umwelt inspirieren lassen kann. Sollten einige der Schreibflächen zeitweise für das Auge eines Normalsterblichen verschlossen bleiben (zum Beispiel drehbare oder aufklappbare Tafeln), so bietet das weiteren Freiraum für die Kreativität und eröffnet dem Gestalter neue Pforten zur Pointenbildung, Portionierung und damit zur Fokussierung seiner künstlerischen Potenz. Dass Tafelbilder eine gewisse Philosophie beinhalten, zeigt unter anderem die einzig mögliche Entstehungsform von Tafeluniversen. Zentralismus ist nicht nur bei Baguetteliebhabern die bevorzugte Anordnung. Das Auge erfasst die Mitte zuerst, es benötigt nicht viel Fachkenntnis, um diese Tatsache einzusehen. Folglich entsteht die Verwirklichung des Gestalters konzentrisch, wodurch die Eckbereiche einer rechteckigen Schreibfläche weitere Facetten bieten. Gerade durch natürliche Nicht-Beachtung dieser kleinen Kosmen gewinnen sie wiederum an Attraktivität seitens der Betrachter. Die nutzbare Schreibfläche, die dem Gestalter mindestens zur Verfügung steht, lässt sich durch die Gleichung

$$V = -(A - \pi r^2) \cdot (F - 1) + A \cdot F$$

beschreiben, wobei V den Verwirklichungsraum, A die gesamte beschreibbare Fläche, r den spezifische Schreibradius des Gestalters (Abhängig von kreativer Potenz und Armlänge) und F die Anzahl der vorhandenen Farben bezeichnet. Da nämlich der aufmerksame Beobachter durchaus in der Lage ist, gleichzeitig bis zu zehn Farbschichten und damit bis zu 20 unabhängige Tafelbilder pro 10 m^2 zu erfassen, lässt sich mit geschickter Farbwahl und strukturierter Anordnung ein Vielfaches an Information und künstlerischem Wert auf gleichbleibender Fläche unterbringen. Allerdings müssen diese Farbschichten nicht zwingend unabhängig sein. Der geschickte Freigeist kann mit einer gelungenen Kombination aus Farben, Formen, Metaphern und Anordnung der Elemente seinen Selbstverwirklichungsdrang optimal ausleben und den Wert des künstlerischen Aspekts potenzieren. Weiterhin werden somit störende Pausen durch körperlich unzumutbares Reinigen der Schreibfläche dezimiert. Die oben erwähnten Randbezirke der Tafel dürfen natürlich keinesfalls mit mehrfarbiger Beschriftung in Berührung kommen, da sonst das gesamte Tafeluniversum aus dem Gleichgewicht geraten würde und ein durch Menschenhand geschaffenes Tafelbild die resultierenden Konsequenzen selten verkraftet.

Doch aufgeschoben ist nicht aufgehoben. Irgendwann steht im Leben eines jeden Gestalters das Tafelsäubern an, wobei sorgfältig zwischen Tafelputzen und Tafelwischen unterschieden werden muss. Man wischt die Tafel um Platz zu schaffen. Im Gegenzug dazu vernichtet man unwiederruflich einige künstlerische Aspekte, meist um diese durch Neuinspirationen zu substituieren. Eine gewisse Schmiererei ist zwar das unvermeidliche Überbleibsel ehemaliger Tafelwunderwerke, sie wird aber oft nicht als störend, sondern als zentrales Element der darauf entstehenden Gestaltung genutzt. Im starken Gegensatz dazu steht das Tafelputzen, welches aus mehreren Schritten besteht. Zuerst wird das Werk in möglichst vielen, chaotischen Zügen mit einem triefendnassen Schwamm mit einer Wasserschicht überzogen, um selbige danach mit einem Abzieher auf den sich darunter befindlichen Fußboden zu verteilen, inklusive der gesamten Schöpfung. Somit wird das einst in unharmonischer Ordnung befindliche Tafelwerk in ein harmonisches Chaos gestürzt, was den Geist des Gestalters in vollster Befriedigung zurücklässt.

Literatur

[1] Blum, Norbert: *Theoretische Informatik – Eine anwendungsorientierte Einführung.* Oldenbourg Verlag. 2001[2].

[2] `http://www.mathematik.uni-muenchen.de/~buchholz/articles/rekursion.ps.gz`
Wilfried Buchholz: »Rekursionstheorie«, Vorlesungsskript für das Sommersemester 2001 an der Ludwig-Maximilians-Universität in München

[3] Goethe, Johann Wolfgang von: *Faust. Der Tragödie erster Teil.*

[4] Immerman, Neil: *Descriptive Complexity.* Springer Verlag. 1999.

[5] `http://www.tcs.informatik.uni-muenchen.de/lehre/WS99-00/KomplTheorie/`
Reinhold Letz: »Komplexitätstheorie«, Vorlesungsskript für das Wintersemester 1999/2000 an der Ludwig-Maximilians-Universität in München

[6] Papadimitriou, Christo H.: *Computational Complexity.* Addison-Wesley. 1995.

[7] Reischuk, Karl-Rüdiger: *Einführung in die Komplexitätstheorie.* Teubner. 1990.

[8] Schöning, Uwe: *Theoretische Informatik – kurzgefasst.* Spektrum Akademischer Verlag. 2001[4].

[9] `http://de.wikipedia.org/wiki/Teile_und_herrsche`
Wikipedia: Teile und herrsche.