Prof. Dr. Hans-Jörg Kreowski
Studiengang Informatik
Universität Bremen
Linzer Str. 9a, OAS 3001
Tel.: 2956, 3697 (Sekr.), Fax: 4322
E-Mail: kreo@informatik.uni-bremen.de
www.informatik.uni-bremen.de/theorie

April 2004

Theoretische Informatik 2

Sommersemester 2004

1 Szenarien – theoretisch

Mit diesem einführenden Kapitel sollen Gegenstände, Ideen und Grundsätze der theoretischen Informatik dargestellt und motiviert werden. Insbesondere wird auf Sachverhalte und Überlegungen eingegangen, die in der Berechenbarkeits- und Komplexitätstheorie eine wichtige Rolle spielen. Den Ausgangspunkt bilden einige Szenarien, in denen fiktiv Anwendungen von Informationstechnik und Situationen aus dem Leben von Informatikerinnen und Informatikern beschrieben werden. Entstehende Ähnlichkeiten mit tatsächlichen Problemen sind beabsichtigt. Die Szenarien sind so zugespitzt, dass die korrekte Beantwortung der gestellten Fragen Kenntnisse und Einsichten der theoretischen Informatik verlangt. Wer die Antworten weiß oder sich richtig überlegen kann, braucht diese Lehrveranstaltung nicht mitzumachen, sondern kann sich den Leistungsnachweis nach einer kurzen Rücksprache abholen.

Anknüpfend an die Szenarien werden einige Wesenszüge der theoretischen Informatik erläutert und einige zentrale Gesichtspunkte von Berechenbarkeits- und Komplexitätstheorie angerissen.

1.1 Terminationstest endlich lieferbar

In der Zeitschrift ITCT, die monatlich von etwa 40.000 Computerfachleuten gelesen wird, fand sich neulich folgende Anzeige:

> Müssen Sie bei Ihren Datenbankanfragen oft lange auf Antwort warten, oder kommt häufig gar keine vernünftige Antwort? Sind Ihre Programme sehr zeitintensiv? Erhalten Sie bei der Ausführung Ihrer Programme oft Fehlermeldungen statt eines Ergebnisses? Brechen Sie häufig Berechnungen ab, weil Sie nicht

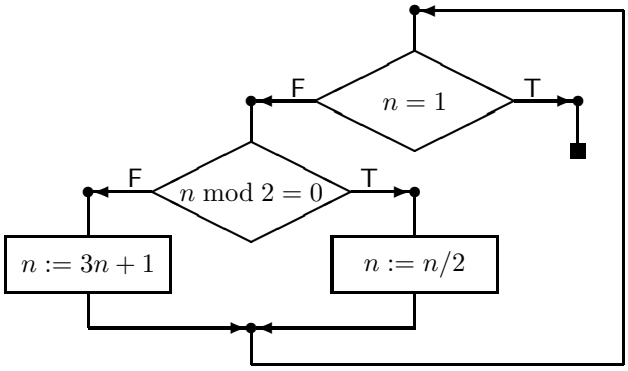

Abbildung 1: Ein Flussdiagramm für die Ulam-Funktion

> mehr warten wollen oder können? Wenn Sie eine dieser Fragen mit JA beantworten, dann haben wir etwas für Sie: *HaltCheck*. Die Weltneuheit *HaltCheck* läuft auf allen gängigen Rechnern und ist für die Datenbankanfragesprache SQL und für die Programmiersprachen C++ und ML erhältlich. Für jede Datenbankanfrage bzw. für jedes Programm und jede Eingabe testet es in Sekundenschnelle die Termination und gibt im positiven Fall eine obere Zeitschranke an, bis zu der man höchstens auf das Berechnungsergebnis warten muss. *HaltCheck* spart Ihnen Zeit und Nerven. Diese sensationelle Entwicklung ist im Fachhandel für nur 1.199 € erhältlich. Kaufen Sie sofort.

Ist das Angebot interessant und günstig? Sollte man mit dem Kauf warten, bis der Preis fällt? Oder was ist von einem derartigen Hilfsprogramm zu halten?

Wenn ein Programm wie *HaltCheck* existierte, wäre es ausgesprochen nützlich. Wer das bezweifelt, möge das Flussdiagramm in Abbildung 1 betrachten, das die Berechnungen der sogenannten Ulam-Funktion beschreibt. Eingabe ist eine natürliche Zahl $n \geq 1$. Ist $n = 1$, so ist das auch das Ergebnis. Ist $n \neq 1$, so wird n halbiert, falls es gerade ist, oder sonst verdreifacht und um 1 erhöht. Mit dem jeweiligen Ergebnis (als neue Eingabe n) wird der Gesamtvorgang wiederholt. Für jede Zahl gibt es offenbar nur zwei Möglichkeiten: Entweder es kommt 1 als Ergebnis heraus oder der Berechnungsvorgang wird unendlich fortgeführt. Für welche Zahlen tritt welcher Fall ein? Zur Warnung sei gesagt, dass man sich jedoch nicht allzu sehr in die Beantwortung der Frage verbeißen sollte, weil sie seit Jahrzehnten ein offenes Problem ist.

1.2 Auf dem schnellsten Wege

In einem Geldtransportunternehmen, das auch andere Wertsachen befördert, wird erwogen, die tägliche Fahrtroutenplanung zu automatisieren. Bei dem dafür zu erstellenden Softwaresystem muss von folgenden Vorgaben ausgegangen werden:

(a) Berechnungsgrundlage ist eine "Landkarte", in der alle bundesdeutschen Ortschaften mit Geldinstituten als Knoten eingetragen sind. Je zwei Orte sind durch eine Kante

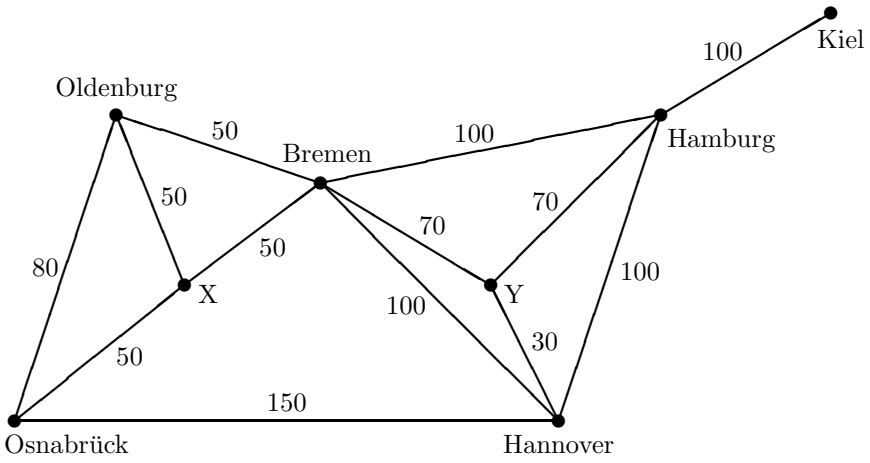

Abbildung 2: Eine "Landkarte"

miteinander verbunden, wenn es zwischen ihnen eine direkte (über keinen anderen Ort aus der Karte führende) Straßenverbindung gibt. Die Kanten sind mit der Kilometerzahl beschriftet.

(b) Die Eingabe besteht aus einer sich täglich ändernden Liste geplanter Fahrten, bei denen jeweils Start und Ziel angegeben sind. Aus Sicherheitsgründen werden keine Zwischenstationen gemacht und auch nicht mehrere Transporte zusammengefasst.

(c) Bestimmt werden soll für jeden Eintrag der eingegebenen Liste der kürzeste Weg in der Landkarte, das ist eine Folge von Kanten, die Start und Ziel miteinander verbindet und deren Kilometersumme unter allen Verbindungswegen die kleinste ist.

Betrachte zur Illustration die Karte in Abbildung 2. Von Bremen nach Hannover gibt es viele Wege (sogar unendlich viele, wenn man erlaubt, beliebig oft hin und her oder im Kreis zu fahren), aber nur der direkte und der über Y sind mit 100 km die kürzesten.

Prinzipiell ist es nicht schwierig, einen Algorithmus zu entwerfen, der diese Aufgabe bewältigt; kritisch jedoch ist sein Laufzeitverhalten. Die Eingabeliste ist aus organisatorischen Gründen nicht vor 0.00 Uhr verfügbar, die ersten Fahrzeuge müssen jedoch spätestens um 6.00 Uhr morgens abfahren. Für die tägliche Einsatzplanung stehen also nur wenige Stunden zur Verfügung.

Das Transportunternehmen besitzt einen Matrixrechner, dessen Architektur insbesondere die Multiplikation von Matrizen unterstützt. Diese Operation kann deshalb besonders schnell ausgeführt werden. Die Automatisierung der Fahrtroutenplanung kommt aus den geschilderten zeitlichen Gründen nur dann in Frage, wenn das algorithmische Problem mit möglichst wenigen Matrizenmultiplikationen gelöst werden kann.

Ist das Vorhaben unter diesen Randbedingungen realisierbar?

1.3 Maschinenbelegung im Sonderangebot

Ein mittelständisches Unternehmen, das Spezialanfertigungen von elektronischen Geräten herstellt, möchte seine Produktion weiter automatisieren. Dazu benötigt es ein Maschinenbelegungsprogramm, das für täglich durchschnittlich 75 Aufträge mit unterschiedlichen Fertigungsplänen den Durchlauf durch 128 Fertigungsstationen organisiert, wobei ein einzelner Auftrag meist nur auf einem Teil der Stationen erledigt wird, das jedoch mit wechselnden Reihenfolgen und Verweilzeiten. Das Unternehmen schreibt ein derartiges Programm aus, wobei es neben den Kosten insbesondere Zuverlässigkeitsgarantien und Angaben über die Laufzeit in der Zeiteinheit μsec verlangt. Folgende Angebote gehen ein, wobei k die Zahl der Aufträge und m die Zahl der insgesamt eingesetzten Stationen ist.

1. 100.000 €; optimale Maschinenbelegung; ungünstigste Laufzeit: m^k.
2. 200.000 €; optimale Maschinenbelegung; durchschnittliche Laufzeit: $k \cdot m!$.
3. 150.000 €; höchstens 100% über optimaler Maschinenbelegung; Laufzeit: $k^2 \cdot m^3$.
4. 120.000 €; höchstens das Dreifache des Optimums; Laufzeit: $128 \cdot k^2$.
5. Das Angebot 4 zum halben Preis, wenn die Zeiteinheit auf *sec* gesetzt werden darf.
6. 80.000 €; Optimum; Laufzeit: $c \cdot k$, wobei c zwischen 10.000 und 100.000 liegt. Allerdings muss dafür noch ein Zusatzprogramm für rund 10.000 € beschafft werden, das für einen Auftrag, der als nächstes auf mehreren Stationen gefertigt werden kann, die aktuell günstigste auswählt.
7. 250.000 €; Optimum; durchschnittliche Laufzeit: $k \cdot m^2$.
8. Das Angebot 7 zum doppelten Preis, dafür aber Laufzeit für den schlechtesten Fall.
9. 300.000 €; optimale Lösung mit 55% Garantie; Laufzeit: $128 \cdot k \cdot \log(k)$. (Anmerkung: Programm benötigt einen guten Zufallsgenerator.)
10. 75.000 €; bis 10% über optimaler Lösung; ungünstigste Laufzeit: $\binom{128}{m} \cdot \binom{m}{k}$.

Es gab noch einige weitere Angebote, die aber zu teuer, offenbar schlechter oder unseriös waren.

Welche Lösung soll das Unternehmen kaufen? Und warum? Wären alternative Angebote denkbar, die die zehn vorgestellten deutlich übertreffen?

Das Fließband, bei dem Aufträge in einer vorgegebenen Folge durch die festinstallierte Sequenz der Maschinen geschleust werden, hat – so scheint es – als Maschinenbelegungsplan ausgedient. In der Illustration für zwei Maschinen in Abbildung 3 werden gerade

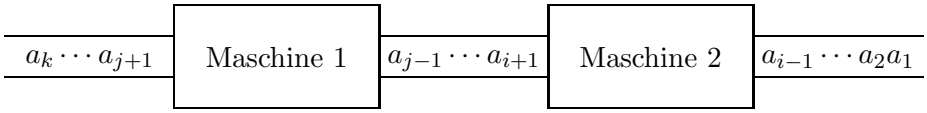

Abbildung 3: Fertigung am Fließband

zwei Aufträge bearbeitet (a_j auf Maschine 1 und a_i auf Maschine 2). Was ist an einer Fließbandlösung für die Maschinenbelegung eigentlich schlecht?

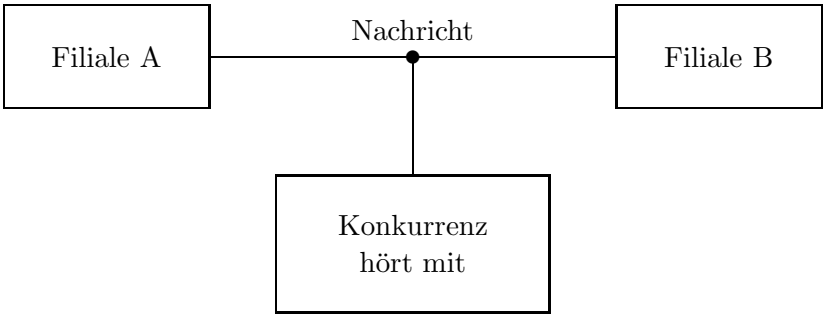

Abbildung 4: Nachrichtenaustausch

1.4 Mit Netz und doppeltem Boden

Eine große internationale Bank mit einem weltumspannenden Netz von über 3000 Filialen möchte ihren Kundendienst verbessern. Dazu sollen alle Zweig- und Geschäftsstellen in ein Kommunikationssystem einbezogen werden, das je zwei Filialen erlaubt, in wenigen Minuten untereinander Informationen auszutauschen. Kostengünstig lässt sich das jedoch nur bewerkstelligen, wenn in Kauf genommen wird, dass Unbefugte ohne großen technischen Aufwand übermittelte Nachrichten auffangen können, wie dies in Abbildung 4 skizziert ist. Um das Bankgeheimnis zu wahren und miss bräuchliche Verwendung der Informationen zu verhindern, ist eine Verschlüsselung unumgänglich.

Unrealistisch wäre ein Konzept, in dem je zwei Filialen für ihre Kommunikation einen eigenen geheimen Schlüssel hätten. Dafür wären 10.000.000 Schlüssel erforderlich. Auch die jeweilige Vergabe eines Schlüssels durch eine Zentrale, wenn zwei Filialen Verbindung aufnehmen wollen, ist umständlich und schwierig, zumal auch die Übersendung der Schlüssel verschlüsselt werden müsste. Da behauptet eine übereifrige Vorstandsassistentin, dass sie einen Artikel gelesen hätte über die Verwendung "öffentlicher" Schlüssel: Jede Filiale bildet nach einem festen, einfachen Prinzip aus nur ihr bekannten Bestandteilen einen Schlüssel, der in eine allen Zweigstellen zugängliche Liste verzeichnet wird. Will nun Filiale A Filiale B eine Mitteilung machen, sucht sie den Schlüssel von B aus dem Verzeichnis heraus, chiffriert nach einem bestimmten Verfahren die Nachricht und sendet sie an B. Die Zweigstelle B entschlüsselt diese Nachricht mit Hilfe der nur ihr bekannten Bestandteile ihres Schlüssels nach einem allen bekannten Verfahren. Ohne die Bestandteile zu kennen, ist die Dechiffrierung jedoch praktisch unmöglich.

Will sich die Vorstandsassistentin nur wichtig machen? Oder sind nicht knackbare Codes möglich, selbst wenn die verwendeten Verfahren für Ver- und Entschlüsseln sowie die Schlüssel allgemein bekannt sind?

1.5 Geheimnisvoller Auftrag

Ein Geheimdienst hört seit Monaten einen gegnerischen Sender ab. Die aufgefangenen Buchstaben-, Zahlen-, Silben- und Wörterfolgen sind jedoch bisher nicht entschlüsselt.

Die erste wichtige Beobachtung ist, dass jede Viertelstunde Informationen von besonderer Form übermittelt werden. Es handelt sich immer um zwei gleichlange Sequenzen von Zeichenfolgen, die durch Hupsignale voneinander getrennt sind. Wenn das Hupen durch einen Gedankenstrich symbolisiert wird, haben die Nachrichten folgendes Aussehen:

$$- \; - \, u_1 - u_2 - \cdots - u_k - \; - \, v_1 - v_2 - \cdots - v_k - \; -$$

Dabei sind u_i, v_j irgendwelche Zeichenfolgen aus Buchstaben und Ziffern. Ein Beispiel dafür ist:

$$- \; - \, aac - a - bb - aac - b - \; - \, a - baa - acb - ca - acb - \; -$$

Die DechiffrierexpertInnen hatten schon längere Zeit vermutet, dass sich in diesen Doppelsequenzen der Schlüssel für die Entschlüsselung aller aufgefangenen Sendungen befindet. Aber wie?

In dieser Situation kam dem Geheimdienst der Zufall zu Hilfe. Ein gegnerischer Agent ging ihm ins Netz und verriet in langen Verhören so manches Geheimnis. Insbesondere bestätigte er, dass sich unter den viertelstündlich gesendeten Folgen u_i und v_j für $i, j = 1, \ldots, k$ der Schlüssel befindet – jedoch nur einmal alle zwei bis drei Tage. Alle anderen Doppelsequenzen sind Tarnung. Die einzige signifikante Doppelsequenz unterscheidet sich von den anderen dadurch, dass es für sie allein keine Indexfolge $i_1 \cdots i_n$ gibt, derart dass die zusammengesetzten Zeichenketten $u_{i_1} u_{i_2} \cdots u_{i_n}$ und $v_{i_1} v_{i_2} \cdots v_{i_n}$ gleich sind. Diese Figuren können nach der Art ihres Zustandekommens *Zwillingspuzzles* genannt werden.

Der Rest schien einfach. Der EDV-Abteilung wurde der Auftrag erteilt, ein Programm zu entwickeln, das für eine eingegebene Doppelfolge testet, ob sie ein Zwillingspuzzle ergibt. Doch die SystemanalytikerInnen und ProgrammiererInnen des Geheimdienstes brachten ein solches Programm nicht zustande. Deshalb soll die Aufgabe einem Softwarehaus übertragen werden, das bei erfolgreicher Erledigung des Auftrags pauschal 120.000 € erhält.

Ein gutes Geschäft? Was ändert sich an der Situation, wenn der jeweils aktuelle Schlüssel in einer Doppelsequenz mit Zwillingspuzzle versteckt wäre und die Tarnsequenzen keine Zwillingspuzzles haben?

1.6 Die Antworten der Theorie

Was haben diese (und ähnliche) Situationen und Problemstellungen mit theoretischer Informatik zu tun? – Die Theorie beantwortet die gestellten Fragen. Aber das allein wäre nichts Besonderes; denn solche Fragen treten häufig bei Softwarentwicklungen auf und werden auch irgendwie beantwortet. Jedoch nur die Theorie gibt *exakte* Antworten (das sind Antworten, die in einem bestimmten Kontext zweifelsfrei und nachweisbar gültig sind). Etwas vereinfachend könnte man sogar sagen: Theoretische Informatik ist gerade dadurch definiert, dass derartige Fragen in einem formalen Rahmen mit mathematischen Methoden und daher exakt beantwortet werden. Wann immer und wo immer in der Informatik so vorgegangen wird, entsteht theoretische Informatik.

Theoretische Informatik ist Informatik mit mathematischen Mitteln.

Die Szenarien münden in Fragen, die ein gemeinsames Grundmuster besitzen: Geht das und das (oder lohnt es sich) mit den und den Methoden unter den und den Bedingungen? Es sollte klar sein, dass in der Informatik (und in manchen anderen Wissenschaften) an vielen Stellen so gefragt werden kann. Es sollte auch klar sein, dass exakte Antworten nicht nur von wissenschaftlichem, sondern auch ökonomischem Wert sein können. Denkt man an computer-gesteuerte Kernkraftwerke, Flugzeuge, Frühwarnsysteme und ähnlich brisante Aufgaben, die mit Hilfe von Computertechnik und Informatikwissen erledigt werden, können (un-)gesicherte Kenntnisse offenbar über Leben und Tod entscheiden. In solchen lebenswichtigen Fragen sind jedoch leider die Antworten der Theorie eher deprimierend. Wegen der technischen Komponenten, in die die Berechnungsprobleme lediglich eingebettet sind (Stichwort: embedded system), sind oft nur statistische Aussagen möglich. Beispielsweise wird sich unter "geeigneten" Annahmen vielleicht herausstellen, dass das russische Frühwarnsystem in einer Million Jahren wahrscheinlich nur 17mal fälschlich einen Atomangriff von Westeuropa aus meldet, ohne dass der Fehler vor Auslösen des Gegenschlags festgestellt wird. Doch die Theorie könnte nicht verhindern, dass die erste dieser Fehlmeldungen bereits morgen erfolgt und das atomare Inferno in Europa auslöst.

1.7 Die Schwierigkeiten der Theorie

Wenn Theorie so wichtig und wertvoll ist wie behauptet, warum ist sie dann in der Informatik und bei vielen Informatikerinnen und Informatikern so umstritten und unbeliebt, ja teilweise verpönt? Ein Grund dafür ist wahrscheinlich, dass es oft nicht einfach ist, grundlegende Fragen der geschilderten Art korrekt zu beantworten. Theorie ist kein Frage-Antwort-System, in das man eine Frage nur hineinstecken muss, um nach einiger Zeit automatisch die richtige Antwort zu erhalten. Oft müssen Fragen und Antworten in einem vorhandenen Rahmen entwickelt werden. Oft fehlt der geeignete theoretische Rahmen und muss erst noch geschaffen werden. Oft sind zwar Rahmen, Frage und Antwort prinzipiell bereits verfügbar, doch ist das schwer zu erkennen, weil die vorgefertigte Theorie anders formuliert ist, als es das aktuelle Problem erfordert. Meist sind deshalb Zeit, Phantasie, Intuition, Wissen und Erfahrung nötig, um existierende Theorie nutzbringend zu verwenden und fehlende zu ergänzen.

Um ein Problem exakt lösen zu können, muss schrittweise und systematisch eine Theorie aufgebaut werden. Die einzelnen Überlegungen müssen aber nicht nur verstanden werden, sondern auch nachprüfbar richtig sein. Das macht Theoriebildung zu einem langsamen, mühsamen Unternehmen. Wie umständlich dieser Vorgang sein kann, mag folgender Zeitplan verdeutlichen: Das Thema dieser Lehrveranstaltung ist Berechenbarkeit und Komplexität; alle Szenarien haben direkt und zentral mit diesem Thema zu tun; dennoch wird nur eins der Probleme in diesem Semester gelöst (vielleicht noch ein zweites).

Theorie ist ein – häufig aufwendiger – sprachlicher Rahmen,
in dem grundsätzliche Probleme exakt gelöst werden können.

1.8 Das Prinzip der Abstraktion

Damit der hohe Aufwand der Theoriebildung gerechtfertigt ist, werden Theorien meist so allgemein und breit entwickelt, dass möglichst viele Probleme damit gelöst werden können. In der Regel haben sogar mehrere verschiedene konkrete Probleme innerhalb einer entsprechenden Theorie eine gemeinsame Lösung. Erreicht wird dieser "ökonomische" Effekt durch einen Abstraktionsprozess, dem die konkreten Probleme unterzogen werden, bevor sie im Rahmen einer Theorie behandelt werden. Abstrahieren heißt, von unwichtigen Details abzusehen und sich auf das Wesentliche zu konzentrieren. Beispielsweise ist im fünften Szenario die "aufregende" Spionagegeschichte für das Problem völlig irrelevant; interessant ist allein, wie festgestellt werden kann, ob eine Doppelsequenz aus Zeichenketten ein Zwillingspuzzle besitzt. Oder im vierten Szenario könnte es sich statt um eine Bank auch um die Mafia, den CIA, die Botschaften eines Staates u.ä. handeln, statt um über 3000 Filialen, die miteinander kommunizieren wollen, auch um 428 Mafia-Organisationen, um 7000 AgentInnen, um 113 diplomatische Vertretungen u.ä. In einer Theorie, in der das Problem der öffentlichen Schlüssel diskutiert werden kann, wird also nicht definiert werden, was eine Bank ist und wie viele Filialen sie hat; davon wird abstrahiert. Stattdessen wird man von der Annahme ausgehen, dass es n Instanzen gibt, wobei n eine beliebige natürliche Zahl ist, und dass je zwei von ihnen Verbindung miteinander aufnehmen können sollen.

Das Prinzip der Abstraktion hat jedoch seine Tücken. Die ursprünglichen konkreten Probleme sind in der abstrakten Theorie nicht ohne weiteres erkennbar. Beim Abstrahieren kann es passieren, dass ein entscheidender Aspekt vernachlässigt wird; dann spiegeln die theoretischen Ergebnisse die wirkliche Situation nicht mehr adäquat wider. Zu einem konkreten Problem mag es längst eine geeignete Theorie geben, die die fertige Antwort parat hält; aber das nutzt nichts, wenn diejenigen, die das Problem haben, die Theorie nicht kennen oder ihnen die richtige Abstraktion nicht gelingt. Ist es da verwunderlich, dass viele Theorien nutzlos und wie abstrakte Spinnerei erscheinen?

1.9 Probleme – unberechenbar

Als allgemeine Vorbemerkungen zur Theorie mag das erst einmal ausreichen. Aber worum geht es nun speziell in dieser Lehrveranstaltung?

Informatik löst spezifische Aufgaben mit Hilfe von Rechenanlagen, untersucht die Gesetzmäßigkeit eines Lösungsvorgangs und stellt Methoden für Lösungsschritte bereit. Der Schlüssel zu all dem ist der Begriff des Algorithmus, worunter eine maschinell ausführbare Lösung eines vorgegebenen Problems verstanden wird. Die theoretische Informatik

verfolgt dieselben Ziele, soweit sie sich mathematisch untermauern lassen. Im Zentrum stehen dabei einige grundsätzliche Fragen, was beispielsweise ein Computer berechnen kann oder welche Probleme algorithmisch lösbar sind.

Beide Fragen sind so noch sehr unscharf formuliert. Denn es gibt ja viele verschiedene Rechnertypen mit verschiedenen Ausstattungen. Man müsste also genauer fragen: Was kann der Computer X, was der Computer Y? Aus praktischer Sicht wäre es fatal, wenn für je zwei Rechner Unterschiedliches und Unvergleichbares herauskäme. Dann wüsste niemand, der ein Problem nicht lösen kann, ob er nicht einfach an der falschen Maschine sitzt. In der zweiten Frage wird unterstellt, dass es nur *ein* "algorithmisch" gibt. Tatsächlich könnte aber die Lösungsfähigkeit und -vielfalt von der gewählten (Programmier-)Sprache oder von anderen Faktoren abhängen. Gibt es also ein "Modula-algorithmisch", das grundverschieden ist von "ML-algorithmisch"? Können in C++ völlig andere Probleme gelöst werden als mit SmallTalk? Bejahende Antworten hätten die unangenehme Konsequenz, dass allein schon die Wahl der Programmiersprache über Erfolg oder Misserfolg eines Problemlösungsversuchs entscheiden könnte. Schließlich könnten sich auch die Möglichkeiten von Rechner und Problemlösungssprache in die Quere kommen. Denn was nützt die beste Sprache, wenn kaum eine Lösung darin auf dem vorhandenen Rechner laufen kann? Was bringt der phantastischste Computer, wenn die Programmiersprache seine Rechenfähigkeit nicht voll ausschöpfen kann?

Die *Berechenbarkeitstheorie* bietet eine These an, deren Gültigkeit viele der gerade angenommenen Schwierigkeiten beseitigte. Die These wurde in den 40er Jahren von A. Church formuliert und lautet:

> Alle genügend allgemeinen Berechnungsmodelle erlauben, dieselbe Klasse von Problemen zu lösen.

In der Theorie der Berechenbarkeit werden vielfältige Belege zusammengetragen, die die Churchsche These stützen. Ein repräsentativer Ausschnitt davon wird in dieser Lehrveranstaltung entwickelt.

Überlegungen und Ansätze zur Berechenbarkeit lassen sich in zwei Klassen einteilen. Eine davon bilden die Rechner- oder Maschinenmodelle wie Turingmaschinen und Random-Access-Maschinen, die als abstrakte Versionen von Computern gesehen werden können. Mathematische Modelle, die stellvertretend für Rechner untersucht werden, haben den Vorteil, nicht so schnell zu veralten und vom Markt zu verschwinden. Ihr Nachteil allerdings besteht darin, dass oft viel Phantasie erforderlich ist, ihre Verwandtschaft mit konkreten Rechnern zu erkennen. Jedes Maschinenmodell MM definiert eine Klasse der von MM berechenbaren Probleme. Die zweite tragende Säule der Berechenbarkeitstheorie beschäftigt sich mehr mit der Problemseite. Untersucht werden rechnerunabhängige Beschreibungen von algorithmischen Problemlösungen, z.B. als rekursive oder iterative Funktionen. Insgesamt wird eine Vielzahl von Berechnungsmodellen eingeführt; jedes einzelne stellt eine Präzisierung des Begriffs "Algorithmus" dar und bestimmt eine Klasse lösbarer Probleme.

Ein wichtiger Teil der Berechenbarkeitstheorie ist dem Vorhaben gewidmet, die Übereinstimmung der verschiedenen Problemklassen im Sinne der Churchschen These zu zeigen. In dieser Lehrveranstaltung wird beispielsweise vorgeführt, dass die von Turingmaschinen berechenbaren Probleme gerade mit den rekursiven Funktionen übereinstimmen.

Berechenbarkeit teilt die "Welt" in zwei Teile: das Berechenbare und das Unberechenbare. Beispiele, die zur ersten Kategorie gehören, entwickelt jede Informatikerin und jeder Informatiker in der täglichen Arbeit. Aber es kann auch von Nutzen sein, etwas über das mit Rechnern nicht Leistbare zu wissen. Das ist nicht allein von philosophischem Belang, sondern kann praktisch auch viel unnütze, vergebliche Anstrengung vermeiden helfen. Einige Exemplare nicht berechenbarer Probleme werden in dieser Lehrveranstaltung diskutiert. Nicht geklärt werden kann, was PolitikerInnen meinen, wenn sie sagen: "Unsere Politik muss berechenbar bleiben!"

1.10 Schnelles Rechnen – viel zu langsam

Es ist beruhigend, dass ein einziger Algorithmusbegriff prinzipiell auszureichen scheint, um das mit Rechnern Machbare untersuchen zu können. In vielen Fällen jedoch ist das Wissen um die Berechenbarkeit nicht sehr hilfreich. Zu diversen Problemen findet man mühelos Algorithmen, doch die Programme laufen nicht zufriedenstellend. Ein Grund dafür ist, dass Turingmaschinen beispielsweise (als Vertreterinnen eines allgemeinen Algorithmus- und Rechnerkonzepts) mit zwei bei wirklichen Computern knappen Ressourcen äußerst verschwenderisch umgehen: Platz und Zeit. Beides steht einer Turingmaschine unbegrenzt zur Verfügung. Vielleicht gilt das prinzipiell sogar für reale Maschinen (die Physik legt sich in diesen Punkten nicht ganz fest), de facto aber können wir nicht immer genügend Speichermedien heranschaffen und schon gar nicht lange warten.

Ein konkretes Beispiel soll die Rolle der Zeit beim Rechnen veranschaulichen. (Analoges lässt sich für den Speicherplatz überlegen.) Computer sind berühmt dafür, dass sie stupideste Aufgaben mit umfangreichem Informationsmaterial in unglaublich kurzer Zeit erledigen können. Aber selbst dabei stoßen sie schnell an ihre Grenzen. Betrachte etwa folgendes simples Problem: Eingegeben werden soll eine beliebige natürliche Zahl $n \in \mathbb{N}$, ausgegeben werden soll eine Liste aller Teilmengen der Menge $[n] = \{0, 1, \ldots, n-1\}$ der ersten n natürlichen Zahlen, d.h. eine Liste der Elemente der Potenzmenge $\mathcal{P}([n])$. Nun hat diese Potenzmenge bekanntlich 2^n Elemente. Die Länge der Ausgabeliste wächst also exponentiell mit der Größe der Eingabe. Einen Eindruck vom schnellen Anwachsen der Funktion 2^n gegenüber der Funktion n^2 vermittelt Tabelle 1. Es ist nicht sonderlich schwierig, einen Algorithmus zu finden, der die Aufgabe löst. Aber selbst wenn der Rechner nur eine Instruktion bräuchte, um eine Teilmenge zu bestimmen und aufzulisten, und 25 Millionen Instruktionen pro Sekunde schaffte (ein Supercomputer) und wenn ich mich nicht verrechnet habe, rechnete das Gerät bei Eingabe von 70 nach einer Million Jahren immer noch.

Das theoretische Gebiet, das sich mit dem Zeit- und Platzbedarf von Algorithmen beschäftigt, ist die *Komplexitätstheorie*. In dieser Lehrveranstaltung werden einige in diesen Be-

n	n^2	2^n
10	100	1024
20	400	1048576
30	900	1073741824
40	1600	1099511627776
50	2500	1125899906842624
60	3600	1152921504606846976
70	4900	1180591620717411303424

(ohne Gewähr)

Tabelle 1: Vergleich der Funktionen n^2 und 2^n

reich einführende Aspekte behandelt.

2 Lauter Wörter

Informatik ist ohne Zeichenketten, die in der Literatur oft auch Wörter genannt werden, undenkbar. Auch die Theorie ist stark auf sie angewiesen. Im Beispiel 1.5 etwa spielen sie verschiedene Rollen: Buchstaben-, Zahlen-, Silben-, Wörter-, Zeichenfolgen, Sequenzen, Nachrichten; einige von ihnen sollen gleichlang sein; andere müssen zusammengesetzt werden; ihre Gleichheit ist gefragt. Den Benutzerinnen und Benutzern von Programmiersprachen sind Zeichenketten als Namen und Ausdrücke, als Dateien und Felder (Ketten konstanter Länge) u.ä. sowie in Gestalt der Programme selbst geläufig. Wörter und Sätze einer natürlichen Sprache wie Deutsch oder Englisch sind weitere wichtige Beispiele.

In diesem Abschnitt sind einige wichtige Informationen zum Umgang mit Zeichenketten zusammengestellt. Vieles davon wird verschiedentlich in die Überlegungen zur theoretischen Informatik während des kommenden Semesters eingehen, ohne dass diese Tatsache immer ausführlich gewürdigt wird.

2.1 Erzeugung von Wörtern

1. Um nicht bei jedem einzelnen Wort den Rahmen festlegen zu müssen, wird vorweg ein Zeichenvorrat A ausgewählt. A wird auch *Alphabet*, die Elemente von A werden *Zeichen* oder *Symbole* genannt.
2. *Wörter* (über A) sind rekursiv definiert durch:
   (a) λ ist ein *Wort*,
   (b) mit $x \in A$ und einem Wort v ist auch xv ein *Wort*.
3. Das initiierende Wort in (a) wird *leeres Wort*, der wiederholbare Vorgang in (b) *Linksaddition* (von x zu v) genannt. Für Wörter sind auch die Bezeichnungen Zeichenketten, Listen, Folgen, Sequenzen, Sätze, "files", Texte, Nachrichten, "strings"

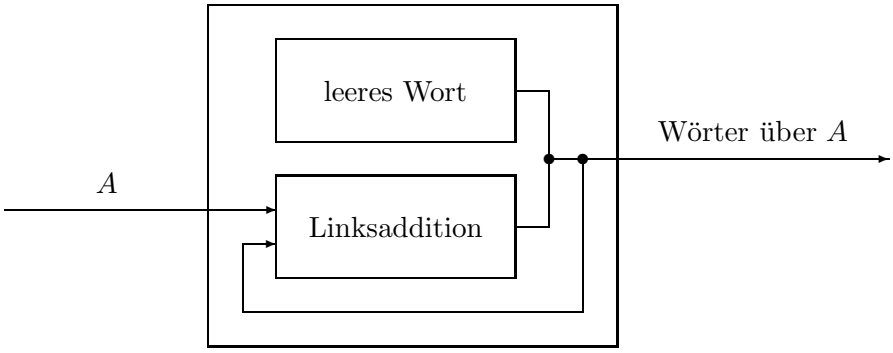

Abbildung 5: Rekursive Erzeugung von Wörtern

u.v.a.m. gebräuchlich. Die erste Bezeichnung wird im folgenden synonym für Wörter verwendet.
Die Menge aller Wörter über A wird mit A^* bezeichnet. Für die Linksaddition $x\lambda$ von x zu λ wird kurz auch x geschrieben. In diesem Sinne gilt: $A \subseteq A^*$.
Der Erzeugungsprozess für Wörter ist in Abbildung 5 illustriert.

4. Beispielsweise entsteht für das Alphabet $\{0, 1\}$ anfangs nur das leere Wort λ, weil der Teil (b) des Worterzeugungsprozesses nur verwendet werden kann, wenn schon Wörter vorhanden sind. Der Teil (a) muss nicht wieder angewendet werden, weil dadurch keine neuen Wörter mehr entstehen können. Die Anwendung von Teil (b) liefert nun zwei neue Wörter: $0\lambda, 1\lambda$ (bzw. $0, 1$ nach der Konvention zur Linksaddition mit dem leeren Wort). Darauf kann der Teil (b) erneut angewendet werden, was vier neue Wörter liefert: 00, 01, 10, 11. Durch Fortsetzung des Verfahrens (ad infinitum) entstehen nach und nach alle (endlichen) Bitstrings.
5. Die Erzeugung von Wörtern ist so gemeint, dass nur Gebilde, die durch den in Punkt 2 gegebenen rekursiven Prozess entstehen, Wörter über A sind und dass jedes Wort in eindeutiger Weise aus diesem Prozess hervorgeht. Letzteres bedeutet, dass für jedes Wort w entweder $w = \lambda$ gilt oder eindeutig $x \in A$ und ein Wort v mit $w = xv$ existieren. Im zweiten Fall wird x mit $head(w)$ und v mit $tail(w)$ bezeichnet.
Bei dem Erzeugungsprozess wird stillschweigend vorausgesetzt, dass man durch die Bezeichnung λ ein Gebilde erhält, das sich von allen anderen Wörtern unterscheiden lässt, und dass das Nebeneinanderschreiben von Zeichen und Wörtern möglich ist. Beides mag intuitiv klar sein; es wird hier einfach vorausgesetzt.
Wenn das Alphabet selbst wieder Wörter enthält, wie z.B. $\{E, I, EI\}$, muss man mit dem Nebeneinanderschreiben aufpassen. Denn sonst kann beispielsweise EI ein Wort aus einem wie zwei Zeichen sein, was wegen der verlangten Eindeutigkeit verboten ist. In solchen Fällen muss man extra Vorsorge treffen, indem man die Zeichen in Hochkommata oder sonstige Klammern einschließt oder Zeichen und Wort beim Nebeneinanderschreiben durch ein Blank oder ein sonstiges Trennzeichen auseinanderhält.

Das durch diese Vereinbarungen verfügbare Textsystem ist noch recht bescheiden. Beginnend mit dem leeren Wort, kann jedes Wort von rechts nach links geschrieben werden; das

zuletzt geschriebene Symbol kann gelesen (*head*) oder gelöscht (*tail*) werden. Wünschenswert wäre mehr Komfort, was mit den Punkten 2.2 und 2.4 ein Stück weit erreicht wird und sich analog noch wesentlich weitertreiben ließe.

2.2 Konkatenation

1. Analog zur Linksaddition und allgemeiner als diese lassen sich auch zwei Wörter v, w zu einem neuen Wort $v \cdot w$ zusammensetzen. $v \cdot w$ wird *Konkatenation* von v und w genannt und ist folgendermaßen rekursiv definiert:
   (a) für $v = \lambda$ gilt $\lambda \cdot w = w$,
   (b) für $v = xu$ mit $x \in A$ gilt $(xu) \cdot w = x(u \cdot w)$.
2. Die Linksaddition erweist sich als Spezialfall der Konkatenation:

$$xv \underset{1a}{=} x(\lambda \cdot v) \underset{1b}{=} (x\lambda) \cdot v \underset{2.1.3}{=} x \cdot v.$$

   Das rechtfertigt die Schreibweise vw für $v \cdot w$.
3. Die Konkatenation fügt zwei Wörter zusammen. Meist werden jedoch mehrere Konkatenationen kombiniert, z.B. ((((BE)I)(SP))((IE)L)). Das sieht hässlich aus; doch glücklicherweise kann man alle Klammern auch weglassen, weil die Reihenfolge, in der Wortteile verknüpft werden, keinen Einfluss auf das resultierende Wort hat. (Dagegen ist die Reihenfolge der Wortteile untereinander entscheidend, wie Punkt 2.5 zeigt).
   Für alle Wörter u, v, w gilt $(uv)w = u(vw)$. Diese Eigenschaft ist auch als Assoziativität bekannt.
   Die Behauptung ergibt sich für $u = \lambda$ nach Punkt 1a (in Verbindung mit der in Punkt 2 eingeführten Schreibweise):

$$(\lambda v)w = vw = \lambda(vw).$$

   Für den Fall $u = xt$ mit $x \in A$ erhält man

$$((xt)v)w \underset{1b}{=} (x(tv))w \underset{1b}{=} x((tv)w) \underset{(*)}{=} x(t(vw)) \underset{1b}{=} (xt)(vw),$$

   wenn die Gültigkeit der Aussage für t vorausgesetzt wird $(*)$. Das ist zulässig, da t um ein Zeichen kürzer ist als u, so dass die Eigenschaft für t als Induktionsvoraussetzung formuliert werden kann.

2.3 Induktionsprinzip

1. Wie in der vorangegangenen Überlegung liefert die rekursive Definition der Wörter ein für viele Situationen brauchbares Induktionsprinzip. Um eine Aussage THEOREM über Wörter zu beweisen, funktioniert oft folgendes Vorgehen:

Induktionsanfang (IA):	Zeige	THEOREM für $v = \lambda$.
Induktionsvoraussetzung (IV):	Nimm	THEOREM für v an.
Induktionsschluss (IS):	Zeige	THEOREM für xv mit $x \in A$.

2. Dieses Prinzip kann benutzt werden, um nachzuweisen, dass λ keinen Einfluss auf die Konkatenation hat. (Vergleiche Punkt 2.2.1a.)
   Für alle Wörter v gilt $v\lambda = v$.
   Denn es gilt:
   $$\lambda\lambda \underset{2.2.1}{=} \lambda \text{ und } (xv)\lambda \underset{2.2.1}{=} x(v\lambda) \underset{(*)}{=} xv,$$
   wobei $(*)$ die Anwendung der Induktionsvoraussetzung anzeigt.
   Die oben gezeigte Assoziativität der Konkatenation beruhte bereits auf dem Induktionsprinzip. So lässt sich auch nachweisen, dass die Konkatenation eindeutig ist.

   IA: Die Konkatenation von λ mit einem beliebigen Wort w liefert nach Definition dieses Wort, ist also eindeutig.

   IV: Sei vw für zwei Wörter v, w ein eindeutig bestimmtes Wort.

   IS: Betrachte nun $(xv)w$ für $x \in A$ und Wörter v, w.

   Nach Definition gilt: $(xv)w = x(vw)$. Nach IV ist vw ein bestimmtes Wort, so dass nach den Festlegungen in Punkt 2.1.5 auch $x(vw)$ eindeutig bestimmt ist.

2.4 Gleichheitstest, Länge und Zeichenzählen

1. Die eindeutige Zerlegbarkeit von Wörtern gemäß Punkt 2.1.5 gestattet auch, in einfacher Weise die Gleichheit zweier Wörter rekursiv festzustellen.
   Zwei Wörter v, w sind *gleich* (in Zeichen: $v \equiv w$), wenn sie beide leer sind: $v = \lambda = w$, oder wenn sie beide nicht leer sind sowie $head(v) \equiv head(w)$ und $tail(v) \equiv tail(w)$, wobei ein Gleichheitstest auf dem Alphabet, der ebenfalls mit $\equiv$ bezeichnet wird, vorausgesetzt ist.
2. Ebenfalls rekursiv bestimmt werden kann, wie lang ein Wort ist und wie oft ein bestimmtes Zeichen darin vorkommt:

   (a) $length(\lambda) = 0$
   (b) $length(xv) = length(v) + 1$ für $x \in A, v \in A^*$
   (c) $count(x, \lambda) = 0$
   (d) $count(x, yv) =$ if $x \equiv y$ then $count(x, v) + 1$ else $count(x, v)$
   für $x, y \in A, v \in A^*$.

   Die in (d) verwendete Fallunterscheidung funktioniert wie üblich: Ist die Abfrage wahr, so wird der then-Teil wirksam, sonst der else-Teil.
   Dass durch (a) und (b) eine Abbildung $length\colon A^* \to \mathbb{N}$ definiert wird, lässt sich mit Hilfe des Induktionsprinzips zeigen:

   IA: $length(\lambda) = 0$ ist genau ein Wert aus $\mathbb{N}$ für λ.

   IV: Für ein Wort v sei $length(v)$ genau eine natürliche Zahl.

IS: Betrachte nun xv für $x \in A$. Nach (b) gilt $\mathit{length}(xv) = \mathit{length}(v) + 1$. Nach IV ist $\mathit{length}(v)$ genau eine natürliche Zahl, so dass der Nachfolger auch genau eine natürliche Zahl ist.

Analog erweist sich auch $\mathit{count}\colon A \times A^* \to \mathbb{N}$ als Abbildung.

3. Auf dieser Basis lassen sich auch diverse weitere Eigenschaften der eingeführten Operationen zeigen. Als Beispiel wird mit vollständiger Induktion bewiesen, dass die Länge einer Konkatenation gerade die Summe der Längen der Einzelwörter ist, d.h. es gilt für alle $v, w \in A^*$:

$$\mathit{length}(vw) = \mathit{length}(v) + \mathit{length}(w)$$

IA: $\mathit{length}(\lambda w) = \mathit{length}(w) = 0 + \mathit{length}(w) = \mathit{length}(\lambda) + \mathit{length}(w)$.

Dabei werden nacheinander die Definition der Konkatenation, eine bekannte arithmetische Eigenschaft und die Definition der Länge ausgenutzt.

IV: Die Behauptung gelte für v und beliebige w.

IS: Betrachte av mit $a \in A$ (und beliebiges w):

$$\begin{aligned}\mathit{length}((av)w) &= \mathit{length}(a(vw))\\ &= \mathit{length}(vw) + 1\\ &= \mathit{length}(v) + \mathit{length}(w) + 1\\ &= \mathit{length}(v) + 1 + \mathit{length}(w)\\ &= \mathit{length}(av) + \mathit{length}(w).\end{aligned}$$

Dabei werden nacheinander die Definition der Konkatenation, die Definition der Länge, die Induktionsvoraussetzung, eine arithmetische Eigenschaft und wieder die Definition der Länge ausgenutzt.

2.5 Iterative Darstellung

Die eindeutige Zerlegbarkeit von Wörtern nach Punkt 2.1.5 ergibt zusammen mit dem Induktionsprinzip eine sehr wichtige, gebräuchliche und vertraute Darstellung von Wörtern.

Für jedes Wort w existieren eindeutig $n \in \mathbb{N}$ und $x_i \in A$ für $i = 1, \ldots, n$ mit $w = x_1 \cdots x_n$.

Das schließt das leere Wort mit ein. In diesem Falle ist $n = 0$, und $x_1 \cdots x_0$ steht für λ. Insgesamt lässt sich also jedes Wort eindeutig in Zeichen als elementare Bausteine zerlegen. Auf den einfachen Beweis wird verzichtet.

2.6 Ansichten von der Menge aller Wörter

Während die vorausgegangenen Informationen über Wörter unverzichtbar für die weiteren Überlegungen sind, dient dieser Abschnitt zur Abrundung. Es wird gezeigt, dass sich die Menge aller Wörter in verschiedenen Formen darstellen lässt. In Punkt 1 wird ein interessanter Zusammenhang zur Algebra und universellen Algebra hergestellt. Punkt 2

charakterisiert A^* durch eine sogenannte Bereichsgleichung. Solche Gleichungen sind in der denotationellen Semantik von Programmiersprachen gebräuchlich. Die Punkte 3 und 4 liefern eine iterative Darstellung von Wörtern, die in der Literatur am häufigsten anzutreffen ist. Die in diesem Kapitel gewählte Einführung wird *axiomatisch* genannt. In Punkt 5 wird die Nähe zu den berühmten Peano-Axiomen der natürlichen Zahlen aufgedeckt. Diese Art des Zugangs eignet sich besonders für den weiteren Umgang mit Wörtern nach Art der funktionalen Programmierung.

1. Die Konkatenation gemäß Punkt 2.2.1 bestimmt eine Abbildung $\cdot : A^* \times A^* \to A^*$, die nach Punkt 2.2.3 assoziativ ist und deshalb A^* zu einem *Monoid* macht mit dem leeren Wort als neutralem Element (vgl. Punkte 2.2.1a und 2.3.2). Da jedes Wort nach Punkt 2.5 eindeutig in "Atome" zerfällt, erweist sich A^* sogar als *freies Monoid*.
2. Die in Punkt 2.1.5 formulierte Zerlegbarkeitseigenschaft der Linksaddition lässt sich explizit dadurch erreichen, dass xv als Paar (x, v) geschrieben wird. Unter Verwendung der Mengenoperationen *disjunkte Vereinigung* $+$ und *kartesisches Produkt* $\times$ erfüllt demnach die Menge A^* aller Wörter über A die Gleichung

   $$A^* = \{\lambda\} + (A \times A^*).$$

   A^* ist sogar die kleinste Menge mit dieser Eigenschaft. Deshalb ließe sich diese Mengengleichung auch zur Definition von A^* und damit von Wörtern über A heranziehen.
3. Eine weitere Möglichkeit, A^* zu definieren, die häufig in der Literatur zu finden ist, besteht darin, Wörter direkt als beliebige Tupel von atomaren Zeichen zu wählen:

   (a) λ ist ein Wort,
   (b) für $n > 0$ und $x_i \in A$ $(i = 1, \ldots, n)$ ist $w = x_1 \cdots x_n$ ein Wort.

   In Tupelschreibweise lautet diese Definition:

   (c) das leere Tupel () ist ein Wort,
   (d) das n-Tupel $(x_1, \ldots, x_n)$ mit $n > 0$, $x_i \in A$, $i = 1, \ldots, n$ ist ein Wort.
4. Beachtet man, dass n-Tupel wie in (b) bzw. (d) von Punkt 3 Elemente des n-fachen kartesischen Produkts A^n sind, ergibt sich für die Menge aller Wörter über A folgende Darstellung

   $$A^* = \sum_{i=0}^{\infty} A^i,$$

   wobei Σ die disjunkte Vereinigung bezeichnet.
   Korrespondierend zur Linksaddition lassen sich die Potenzen A^i iterieren:

   $$A^0 = \{\lambda\} \text{ und } A^{i+1} = A \times A^i \text{ für } i \in \mathbb{N}.$$

5. Betrachtet man speziell das einelementige Alphabet $\{|\}$, so entstehen als Wörter Strichfolgen beliebiger Länge, wobei jedes Wort bereits eindeutig durch seine Länge festgelegt ist. Das leere Wort kann also als 0 gesehen werden, und die Linksaddition

entspricht der Nachfolgerfunktion natürlicher Zahlen. Die Strichdarstellung natürlicher Zahlen ist als Bierdeckel-Arithmetik bekannt. Spezialisiert man außerdem das Induktionsprinzip für Wörter auf diesen Fall, erhält man ein bekanntes Induktionsprinzip für natürliche Zahlen. Insgesamt erweist sich also der in diesem Kapitel gewählte Zugang zu Wörtern als Verallgemeinerung der durch die Peano-Axiome definierten natürlichen Zahlen.
Es sei noch folgendes angemerkt. Ergänzte man die Erzeugung von Wörtern explizit um ihre Länge:

(a) λ ist ein Wort der Länge 0,
(b) xv ist ein Wort der Länge $n+1$, falls $x \in A$ und v ein Wort der Länge $n \in \mathbb{N}$ ist,

so ließe sich das Induktionsprinzip in 2.3 durch vollständige Induktion über die Länge von Wörtern beweisen.

3 Berechnung von Operationen auf Zeichenketten mit Hilfe bedingter Gleichungen – die Sprache CE-S

Wie man an den Beispielen Konkatenation, Länge, Gleichheitstest, Zeichenzählen u.v.a.m. sehen kann, lassen sich Operationen auf Zeichenketten durch endlich viele Gleichungen spezifizieren. Manchmal ist es auch günstig, die Gleichungen noch von Bedingungen abhängig zu machen. Da sich diese Art der Spezifikation sowohl für eine Präzisierung von Berechenbarkeit eignet als auch in eleganter und einfacher Weise erlaubt, den Aufwand beim Berechnen abzuschätzen, wird dieses Konzept als Spezifikationssprache CE-S (Conditional Equations – Strings) eingeführt.

3.1 Syntax von CE-S

1. Als *Typen* sind verfügbar: Ein beliebiges Alphabet A und die Menge A^* aller Zeichenketten über A, die natürlichen Zahlen $\mathbb{N}$ bzw. ganzen Zahlen $\mathbb{Z}$ sowie die Wahrheitswerte *BOOL*. Syntaktisch gesehen sind das fünf Namen. Darüber hinaus ist das Alphabet ein variabler Typ, d.h. man kann mehrere Alphabete gleichzeitig wählen, die dann aber unterschiedlich benannt sein müssen. Demgemäß ist A^* parametrisiert, so dass für jede Wahl des Alphabets ein anderer Zeichenkettentyp entsteht.
2. Als *Grundoperationen* sind verfügbar: der Gleichheitstest $\equiv$ und eine totale Ordnung $\leq$ auf A; das leere Wort λ, die Linksaddition und die Konkatenation sowie der Gleichheitstest $\equiv$ und die Länge *length* auf A^*; die üblichen Konstanten, arithmetischen und Vergleichsoperationen auf $\mathbb{N}$ und $\mathbb{Z}$; die Wahrheitswerte T und F und die üblichen aussagenlogischen Operationen auf *BOOL*. Wenn die Zeichen aus A explizit gebraucht werden, wird angenommen, dass A ein Aufzählungstyp ist, dessen endlich viele (konstante) Elemente namentlich bekannt sind. Syntaktisch werden davon nur die Deklarationsteile verwendet (vgl. Punkt 4).

3. Neue Operationen lassen sich dann durch *Spezifikationen* der folgenden Form einführen:

   spec
   opns: $decl_1, \ldots, decl_k$
   eqns: $ce_1, \ldots, ce_l$

   wobei **spec** ein Name ist, "opns" und "eqns" Schlüsselwörter, $decl_i$ für $i = 1, \ldots, k$ Deklarationen gemäß Punkt 4 und ce_j für $j = 1, \ldots, l$ bedingte Gleichungen gemäß Punkt 5 sind.
4. Eine *Deklaration* hat die Form $f\colon D_1 \times \cdots \times D_m \to D$, wobei f ein Name ist und $D, D_1, \ldots, D_m$ Typen sind. Das schließt den Fall $m = 0$ ein, durch den Konstanten $c\colon \to D$ deklariert werden.
5. Eine *bedingte Gleichung* hat die Form eq falls $eq_1, \ldots, eq_n$ für $tv_1, \ldots, tv_p$, wobei eq und $eq_1, \ldots, eq_n$ Gleichungen gemäß Punkt 7 und $tv_1, \ldots, tv_p$ Deklarationen getypter Variablen gemäß Punkt 6 sind.
6. Die Deklaration einer *getypten Variablen* hat die Form $x \in D$, wobei x ein Name und D ein Typ ist.
7. Eine Gleichung hat die Form $L = R$, wobei L und R Terme eines Typs D über einer Deklarationsmenge $DECL$ und einer Menge X von getypten Variablen gemäß der folgenden Definition sind.
8. Die Menge T_D aller *Terme* des Typs D über $DECL$ und X ist rekursiv definiert durch:
   (i) $(c\colon \to D) \in DECL$ impliziert $c \in T_D$;
   (ii) $(x \in D) \in X$ impliziert $x \in T_D$;
   (iii) $(f\colon D_1 \times \cdots \times D_k \to D) \in DECL$ und $t_i \in T_{D_i}$ für $i = 1, \ldots, k$ implizieren $f(t_1, \ldots, t_k) \in T_D$.

Es macht Sinn anzunehmen, dass die Terme, die in einer bedingten Gleichung einer Spezifikation vorkommen, nur die Variablen der bedingten Gleichung und die Deklarationen der Grundoperationen, der aktuellen Spezifikation sowie anderer bereits bekannter Spezifikationen verwenden.

Die Definition der Syntax von CE-S ist so zu verstehen, dass eine Zeichenkette genau dann eine CE-S-Spezifikation ist, wenn sie den Bedingungen der Punkte 1 bis 8 genügt. Die textuellen Formen in den Punkten 1 bis 7 sind explizit angegeben, der Termbegriff in Punkt 8 ist rekursiv definiert. Die einfachste Sicht auf diese Beschreibung ist wohl, die Menge der Terme T_D als die kleinste Menge von Wörtern über den Namen von deklarierten Operationen und Variablen sowie dem Komma und den zwei runden Klammern zu sehen, die die drei Eigenschaften (i), (ii) und (iii) besitzt. Prozesshaft gesehen ist T_D also die Menge von Wörtern, die dadurch entsteht, dass ausgehend von den Namen von Konstanten und Variablen des Typs D die formale Operationsanwendung beliebig oft wiederholt wird.

Da bei der Definition von Termen des Typs D auch Terme anderer Typen eingehen können, ist die Rekursion so gemeint, dass sie für alle Typen, die in $DECL$ und X vorkommen, gleichzeitig ausgeführt wird.

Mit der rekursiven Definition von Termen ist analog zu Wörtern ein Induktionsprinzip verbunden: Eine Eigenschaft gilt für alle Terme, wenn sie als Induktionsanfang für Konstanten und Variablen und im Induktionsschluss für einen zusammengesetzten Term (einer bestimmten Länge) unter der Voraussetzung gezeigt werden kann, dass die Eigenschaft für kürzere Terme und damit insbesondere für Teilterme bereits gilt.

Einige Konventionen erleichtern außerdem das Schreiben von CE-S-Spezifikationen:

- $f_1, \ldots, f_n \colon D_1 \times \cdots \times D_m \to D$ für $f_1 \colon D_1 \times \cdots \times D_m \to D, \ldots, f_n \colon D_1 \times \cdots \times D_m \to D$.
- $x_1, \ldots, x_p \in D$ für $x_1 \in D, \ldots, x_p \in D$.
- Haben alle bedingten Gleichungen denselben für-Teil $tv_1, \ldots, tv_p$, so kann man diesen dort weglassen und dafür $tv_1, \ldots, tv_p$ vor die bedingten Gleichungen hinter dem Schlüsselwort "vars" und einem Doppelpunkt schreiben.
- Den für-Teil kann man auch weglassen, wenn keine Variablen verwendet werden, d.h. wenn $p = 0$.
- Den falls-Teil einer bedingten Gleichung kann man weglassen, wenn keine Bedingung folgt, d.h. wenn $n = 0$.
- Infix-Notation $t_1 \, f \, t_2$ für $f(t_1, t_2)$, wenn f eine zweistellige Grundoperation ist.
- t für $t = \mathsf{T}$.
- $\neg t$ für $t = \mathsf{F}$; $t_1 \not\equiv t_2$ für $\neg(t_1 \equiv t_2)$; $t_1 > t_2$ für $\neg(t_1 \leq t_2)$.
- $t =$ if b then t_1 else t_2 für zwei bedingte Gleichungen der Form $t = t_1$ falls b und $t = t_2$ falls $\neg b$.

Soviel zur Syntax.

3.2 Beispiele

Als Beispiele werden eine Zeichensuche, das Zeichenzählen aus 2.4, eine Zeichenzähldifferenz, ein Sortiertheitstest und eine Sortieroperation spezifiziert:

search

opns:	$\mathit{search} \colon A \times A^* \to \mathit{BOOL}$
vars:	$x, y \in A$, $v \in A^*$
eqns:	$\neg \mathit{search}(x, \lambda)$
	$\mathit{search}(x, yv) = x \equiv y \vee \mathit{search}(x, v)$

count

opns:	$\mathit{count} \colon A \times A^* \to \mathbb{N}$
vars:	$x, y \in A$, $v \in A^*$
eqns:	$\mathit{count}(x, \lambda) = 0$
	$\mathit{count}(x, yv) =$ if $x \equiv y$ then $\mathit{count}(x, v) + 1$ else $\mathit{count}(x, v)$

diff

opns:	$\mathit{diff} \colon A \times A \times A^* \to \mathbb{Z}$
eqns:	$\mathit{diff}(x, y, w) = \mathit{count}(x, w) - \mathit{count}(y, w)$ für $x, y \in A$, $w \in A^*$

is-sorted

opns: $\mathit{is\text{-}sorted}\colon A^* \to \mathit{BOOL}$
vars: $x, y \in A,\ v \in A^*$
eqns: $\mathit{is\text{-}sorted}(\lambda)$
$\mathit{is\text{-}sorted}(x)$
$\mathit{is\text{-}sorted}(xyv) = x \leq y \wedge \mathit{is\text{-}sorted}(yv)$

sort$_0$

opns: $\mathit{sort}_0\colon A^* \to A^*$
vars: $x, y \in A,\ u, w \in A^*$
eqns: $\mathit{sort}_0(w) = w$ falls $\mathit{is\text{-}sorted}(w)$
$\mathit{sort}_0(uxyw) = \mathit{sort}_0(uyxw)$

Bei der Differenzspezifikation wird vorausgesetzt, dass $\mathbb{N} \subseteq \mathbb{Z}$ gilt und dass deshalb Terme des Typs $\mathbb{N}$ automatisch Terme des Typs $\mathbb{Z}$ sind. Ansonsten handelt es sich um eine typische Funktionsdefinition, indem für alle Argumente ein geschlossener Ausdruck als Wert angegeben wird. Lässt sich eine Funktion nicht für alle Argumente direkt definieren, kann man versuchen, verschiedene Fälle zu unterscheiden. Bei Wörtern beispielsweise bietet sich an, das leere Wort von den nichtleeren Wörtern zu trennen, weil alle nichtleeren Wörter durch die Variablen $x \in A$ und $v \in A^*$ als xv darstellbar sind. Diese Fallunterscheidung ist in **search** und **count** genau so verwendet. In **is-sorted** ist sie zweimal angewendet, indem nichtleere Wörter noch einmal in $x\lambda = x$ und xyv unterschieden sind, also in Wörter der Länge 1 und längere Wörter. In **sort$_0$** wird ein Fall durch eine Bedingung ausgedrückt, ein weiterer wieder durch die Form des Arguments, die hier irgendwo im Wort zwei aufeinanderfolgende Zeichen verlangt, egal was vorher und nachher kommt.

Intuitiv sollte klar sein, wie mit den Gleichungen in Spezifikationen gerechnet werden kann und wie insbesondere Operationen ausgerechnet werden können. Denn das Prinzip ist von der funktionalen Programmierung her bekannt und wurde im vorigen Kapitel bereits mehrmals verwendet, ohne näher darauf einzugehen. Findet man innerhalb eines Terms einen Teilterm, auf den die linke Seite einer Gleichung passt, so darf dieser Teilterm durch die zugehörige rechte Seite der Gleichung ersetzt werden. Die linke Seite passt, wenn man ihre Variablen so mit Termen belegen kann, dass genau der Teilterm entsteht. Vor dem Ersetzen des Teilterms müssen die Variablen der rechten Seite genau wie die der linken Seite belegt werden. Das sei an einem Beispiel illustriert:

$$\mathit{count}(0, 100) = \mathit{count}(0, 00) = \mathit{count}(0, 0) + 1 = \mathit{count}(0, \lambda) + 1 + 1 = 0 + 1 + 1 = 2$$

Im ersten Schritt wird die zweite **count**-Gleichung angewendet, wobei x mit 0, y mit 1 und v mit 00 belegt werden; im zweiten Schritt die zweite Gleichung, wobei x, y und v mit 0 belegt werden; im dritten Schritt die zweite Gleichung, wobei x und y mit 0 und v mit λ belegt werden; und im vierten Schritt die erste Gleichung, wobei x mit 0 belegt wird.

3.3 Gleichwertigkeit von Termen

Spezifizierte Gleichungen drücken gewünschte oder erwartete Gleichheiten aus. Um die Wirkung von Gleichungen exakt zu fassen, wird eine binäre Relation auf Termen namens Gleichwertigkeit eingeführt. Sie bildet das Grundkonzept der operationellen Semantik von CE-S.

Sei **spec** eine Spezifikation mit der Menge CE von bedingten Gleichungen und $DECL$ eine Menge von Deklarationen, die die von **spec** und die aller Grundoperationen und aller in CE verwendeten Operationen enthält. $DECL_0$ bezeichne den Teil von $DECL$, der keine Deklaration aus **spec** enthält. Sei X eine Menge getypter Variablen, die insbesondere alle in **spec** vorkommenden enthält. Sei schließlich für alle Terme über $DECL_0$ und X bereits eine binäre Relation $\underset{0}{\overset{*}{\longleftrightarrow}}$ gegeben.

1. Dann lässt sich die binäre Relation $\overset{*}{\longleftrightarrow}$ auf Termen über $DECL$ und X, die *Gleichwertigkeit* genannt wird, als Fortsetzung der gegebenen Relation rekursiv definieren, wobei Wertzuweisungen gemäß Punkt 2 und die Substitution gemäß Punkt 3 verwendet werden:
   (i) $\underset{0}{\overset{*}{\longleftrightarrow}} \subseteq \overset{*}{\longleftrightarrow}$;
   (ii) $(L = R \text{ falls } L_1 = R_1, \ldots, L_n = R_n \text{ für } x_1 \in D_1, \ldots, x_p \in D_p) \in CE$ und eine Wertzuweisung a implizieren $L[a] \overset{*}{\longleftrightarrow} R[a]$, vorausgesetzt, dass bereits $L_i[a] \overset{*}{\longleftrightarrow} R_i[a]$ für $i = 1, \ldots, n$ gilt;
   (iii) $(f\colon D_1 \times \cdots \times D_k \to D) \in DECL$ und $t_i \overset{*}{\longleftrightarrow} t_i'$ mit $t_i, t_i' \in T_{D_i}$ für $i = 1, \ldots, k$ implizieren $f(t_1, \ldots, t_k) \overset{*}{\longleftrightarrow} f(t_1', \ldots, t_k')$;
   (iv) $t \overset{*}{\longleftrightarrow} t$ (für alle Terme t); $t \overset{*}{\longleftrightarrow} t'$ impliziert $t' \overset{*}{\longleftrightarrow} t$ (für alle Terme t, t'); $t \overset{*}{\longleftrightarrow} t'$ und $t' \overset{*}{\longleftrightarrow} t''$ implizieren $t \overset{*}{\longleftrightarrow} t''$ (für alle Terme t, t', t'').
2. Eine *Wertzuweisung* a ordnet jeder getypten Variablen $(x \in D) \in X$ einen Term $a(x) \in T_D$ zu.
3. Zu jeder Wertzuweisung a gibt es eine *Substitution*, die für jede in einem Term vorkommende Variable den zugewiesenen Term einsetzt und alles andere lässt, wie es ist, was wieder einen Term ergibt:
   (i) $(c\colon \to D) \in DECL$ impliziert $c[a] = c$,
   (ii) $(x \in D) \in X$ impliziert $x[a] = a(x)$,
   (iii) $(f\colon D_1 \times \cdots \times D_k \to D) \in DECL$ (wobei $k > 0$) und $t_i \in T_{D_i}$ für $i = 1, \ldots, k$ implizieren $f(t_1, \ldots, t_k)[a] = f(t_1[a], \ldots, t_k[a])$.

Die Definition der Gleichwertigkeit ist so zu verstehen wie die Definition von Termen und Spezifikationen. Sie ist die kleinste binäre Relation auf Termen, die die Eigenschaften (i) bis (iv) besitzt. Insbesondere ist die Gleichwertigkeit mit Eigenschaft (iv) reflexiv, symmetrisch und transitiv und damit eine Äquivalenzrelation. Da sie nach Eigenschaft (iii) auch gegenüber Operationsanwendung abgeschlossen ist, erweist sie sich sogar als Kongruenzrelation. Die Eigenschaften (i) und (ii) sagen, dass sie von der vorgegebenen Relation und den spezifizierten bedingten Gleichungen erzeugt wird. Man beachte, dass die dabei verwendete Substitution eine reine textuelle Ersetzung von Variablen durch Terme ist.

Man mag sich nun noch wundern, wo die vorgegebene Relation herkommen kann. Sie ist beliebig wählbar, aber es gibt eine besonders interessante und sinnvolle Wahl. Man kann annehmen, dass alle in der aktuellen Spezifikation über die Grundoperationen hinaus verwendeten Operationen vorher bereits in CE-S spezifiziert sind. Somit kann induktiv vorausgesetzt werden, dass für sie die Gleichwertigkeit bereits bekannt ist. Bleibt als Induktionsanfang sicherzustellen, dass es für die aus den Grundoperationen aufgebauten Terme eine geeignete Gleichwertigkeit gibt. Für den Typ *BOOL* wird dafür die übliche aussagenlogische Äquivalenz genommen. Für die Typen $\mathbb{N}$ und $\mathbb{Z}$ kann man die bekannten arithmetischen Gesetze verwenden. Ist das Alphabet ein Aufzählungstyp, nimmt man die Gleichheit. Für den Zeichenkettentyp erhält man mit den für Grundoperationen in Kapitel 2 eingeführten Gleichungen durch die obige Definition eine Gleichwertigkeit, wenn man in Teil (i) mit der leeren Relation beginnt. Falls das Alphabet selbst wieder einer der anderen Typen ist, nimmt man dessen Gleichwertigkeitsrelation.

Um die Rolle der Gleichwertigkeit suggestiv zu betonen, darf statt $t \overset{*}{\longleftrightarrow} t'$ auch $t = t'$ geschrieben werden, wenn keine Missverständnisse möglich sind.

3.4 Operationelle Semantik von CE-S

Das im vorigen Abschnitt eingeführte Verfahren generiert algorithmisch Gleichungen zu einer Spezifikation, wenn man annimmt, dass die entsprechende Gleichung für alle verwendeten Grundoperationen und anderweitig spezifizierten Operationen sowie für alle Variablen bereits von einem entsprechenden Verfahren geliefert werden. Die Situation ist in Abbildung 6 veranschaulicht.

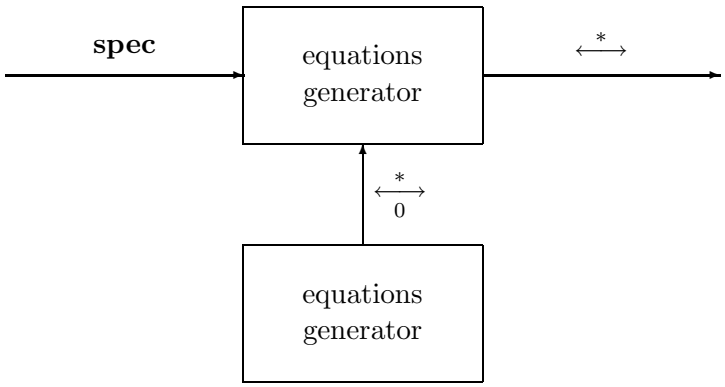

Abbildung 6: Erzeugung von Gleichungen

Dieses Verfahren hat einen Haken und taugt deshalb für sich genommen nur bedingt als operationelle Semantik. Die Gleichwertigkeit ist fast immer eine unendliche Relation, so dass der Generator i.a. nicht anhält. Außerdem ist es völlig ungewiss, wann interessante Gleichungen erzeugt werden. Meist ist es so, dass man sich für bestimmte Gleichungen interessiert. Oder man hat sogar nur einen Term und wüsste gern, zu welchen er gleich-

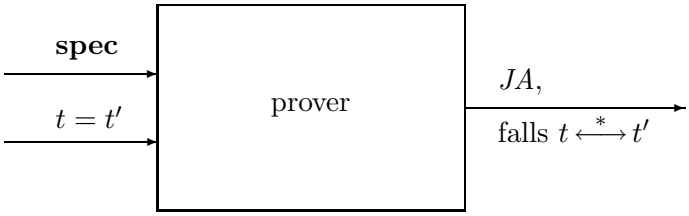

Abbildung 7: Ein Gleichungsbeweiser

wertig ist. Das letztere ist die typische Situation beim Auswerten von Funktionen bzw. Operationen. Man wendet eine Operation auf Argumente an und wüsste gern den Wert. Beispielsweise mag die Länge eines Wortes interessieren oder die Frage, ob eine Zeichenkette ein bestimmtes Zeichen enthält und sortiert ist. Das erstere kommt vor, wenn man zwei Größen vergleichen möchte. Beispielsweise könnte man sich dafür interessieren, ob die Länge einer Konkatenation mit der Längensumme der Teilwörter übereinstimmt. In beiden Fällen liefert die Gleichwertigkeit algorithmische Lösungen.

Gibt man eine Gleichung vor und wartet, bis der Gleichungsgenerator sie produziert, hat man die Gültigkeit der Gleichung automatisch bejaht. Das Ergebnis ist ein Gleichungsbeweiser wie in Abbildung 7 skizziert.

Gibt man einen Term vor und lässt wieder den Gleichungsgenerator laufen, kann man auf Gleichungen warten, die den gegebenen Term als linke Seite haben. Die rechten Seiten kann man dann als Berechnungsergebnisse des Terms ansehen. Das gilt insbesondere, wenn der Ausgangsterm eine Operation auf bekannte Größen anwendet und der Ergebnisterm ebenfalls eine bekannte Größe ist. Das lässt sich präzise so ausdrücken.

1. Ein Term, der nur aus Grundoperationen und Variablen aufgebaut ist, wird *Werteterm* oder kurz *Wert* genannt.
2. Sei $f\colon D_1 \times \cdots \times D_k \to D_0$ eine in **spec** deklarierte Operation. Seien $t_i \in T_{D_i}$ für $i = 0, \ldots, k$ Werteterme. Dann ist t_0 ein *berechneter Wert* von $f(t_1, \ldots, t_k)$, wenn gilt:
$$f(t_1, \ldots, t_k) \overset{*}{\longleftrightarrow} t_0.$$

Die Idee hinter der operationellen Semantik ist folgende: Die Werteterme repräsentieren die Werte der verfügbaren Typen, d.h. alle Zeichen, alle Zeichenketten, alle natürlichen und ganzen Zahlen bzw. die beiden Wahrheitswerte. Die spezifizierten Operationen sollen berechnet werden. Wenn $f\colon D_1 \times \cdots \times D_k \to D$ eine solche Operation ist, will man also wissen, welchen Wert aus D für Argumente aus $D_1, \ldots, D_k$ sie liefert. Da die Argumente durch Werteterme $t_i \in T_{D_i}$ gegeben sind, ist also die Frage, zu welchen Wertetermen des Typs D der auszuwertende Term $f(t_1, \ldots, t_k)$ gleichwertig ist, welche berechneten Werte dieser Term besitzt. Das Attribut *berechnet* ist gerechtfertigt, weil Punkt 1 der Gleichwertigkeit ein Verfahren liefert, bei dem nach und nach alle Paare gleichwertiger Terme entstehen, so dass man nur warten muss, bis für eine Operation, angewendet auf bestimmte Argumente, ein Wert ermittelt ist (falls man nicht vergeblich wartet, weil es gar keinen Wert gibt).

Was man mit diesen Überlegungen erhält, ist ein Interpreter für CE-S, siehe Abbildung 8.

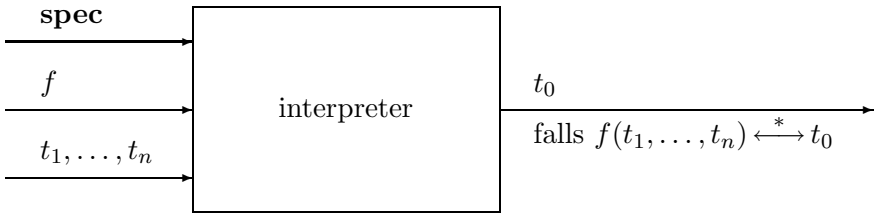

Abbildung 8: Ein Interpreter für CE-S

Man beachte, dass CE-S einige Ähnlichkeiten mit einer funktionalen Sprache aufweist. Die syntaktischen Gemeinsamkeiten sollten ins Auge springen, aber auch die operationelle Semantik von CE-S kann durchaus so gedeutet werden, dass sie nicht völlig von den Interpretern funktionaler Programmiersprachen verschieden ist. Aber es gibt auch signifikante Unterschiede. So ist die Gleichung $swap(uxvyw) = uyvxw$ für $x, y \in A$, $u, v, w \in A^*$ zulässig, die für *swap*, angewendet auf Zeichenketten länger als 2, jeweils mehrere Werte liefert:

$$swap(abc) \overset{*}{\longleftrightarrow} \begin{cases} cba \\ bac \\ acb \end{cases}$$

Die CE-S-spezifizierten Operationen sind also im allgemeinen relational. Um funktionale Operationen zu erhalten, muss man zusätzlich Vorkehrungen treffen. Ähnlich verhält es sich, wenn man vermeiden möchte, dass eine Operation unvollständig definiert ist. Denn in CE-S kann man leicht Spezifikation schreiben, bei denen Operationen für manche Argumente keine berechneten Werte besitzen. Das ist allerdings in funktionalen Sprachen bekanntlich ebenfalls möglich.

3.5 Gerichtete Auswertung

Neben der möglichen Mehrwertigkeit von CE-S-Operationen besteht ein anderer wichtiger Unterschied zur funktionalen Programmierung in der Symmetrie-Eigenschaft der Gleichwertigkeit, während ein Interpreter einer funktionalen Programmiersprache die Gleichungen bzw. Regeln, die ein Programm bilden, immer nur von links nach rechts anwendet.

Diese Variante der operationellen Semantik lässt sich für CE-S sehr einfach definieren, weil man in der Definition 3.3.1 der Gleichwertigkeit nur die Symmetrieforderung in Unterpunkt (iv) streichen muss. Die resultierende Relation wird *Termersetzung* genannt und mit $\overset{*}{\longrightarrow}$ bezeichnet. Es ist dann naheliegend, dass man auch die vorgegebene Relation für die vorausgesetzten Operationen als Termersetzung wählt. Für die Grundoperationen ist es allerdings ratsam, bei der Gleichwertigkeit zu bleiben. Da Termersetzung eine Teilrelation der Gleichwertigkeit ist, bleibt es gerechtfertigt, ein Gleichheitszeichen zwischen

Termen zu setzen, die durch Termersetzung auseinander hervorgehen.

Wie in 3.4 die Gleichwertigkeit induziert auch die Termersetzung einen Interpreter für CE-S, der zur Unterscheidung *Vorwärtsinterpreter* genannt wird, siehe Abbildung 9.

Diese gerichtete Art der Auswertung von Termen wird in der Literatur am häufigsten für diesen Zweck eingesetzt und wird auch im folgenden meist eingesetzt, wenn der Wert einer Operationsanwendung ermittelt werden soll.

3.6 Gleichungsanwendung auf Terme

Wenn ein bestimmter Term ausgewertet werden soll, und das ist häufig der Fall, sind die bisherigen Überlegungen, algorithmisch gesehen, noch ziemlich unbefriedigend. Denn nach Definition von Gleichwertigkeit und Termersetzung werden mehr und mehr Paare von Termen generiert, die mit dem interessierenden Term oft gar nichts zu tun haben werden. Man muss dann warten, bis ein Termpaar erscheint, dessen linke Seite mit dem gegebenen Term übereinstimmt, so dass dessen rechte Seite als Wert oder Zwischenergebnis in Frage kommt. Geschickter ist es in diesem Zusammenhang, die Termersetzung (oder Gleichwertigkeit) so zu konstruieren, dass der gegebene Term direkt und zielgerichtet ausgewertet wird, möglichst ohne überflüssige Termpaare zu produzieren.

Ansatzpunkte für ein solches Vorgehen sind bereits in den Unterpunkten (ii) und (iii) von 3.3.1 angelegt. Stimmt ein gegebener Term t mit einer substituierten linken Regelseite $L[a]$ überein, so lässt sich $t = L[a]$ zu $R[a]$ auswerten (falls die Bedingungen erfüllt sind). Außerdem darf in einem Argumentterm einer Operationsanwendung ausgewertet werden. Rekursiv fortgesetzt darf dann auch in einem Argumentterm eines Argumentterms eines Argumentterms ... gerechnet werden, d.h. in einem beliebigen Teilterm des gegebenen Terms. Man kann also $L[a]$ durch $R[a]$ ersetzen, wenn $L[a]$ Teilterm von t ist, wobei sich das dadurch ausdrücken lässt, dass $L[a]$ beim Einsetzen in einen geeigneten Kontext t ergibt. Genauer lässt sich die Gleichungsanwendung auf Termen folgendermaßen definieren:

Sei $L = R$ eine Gleichung mit $L, R \in T_D$ und Variablen aus X; sei a eine Wertzuweisung. Sei C ein *Kontextterm*, der neben möglichen Variablen aus X genau eine Extravariable $-$ (des Typs D) enthält. Betrachte für diese Variable die beiden Wertzuweisungen $a_L(-) = L[a]$ und $a_R(-) = R[a]$.

Dann entsteht ein Term t' durch *Anwendung* von $L = R$ aus dem Term t, in Zeichen: $t \longrightarrow t'$, falls $t = C[a_L]$ und $t' = C[a_R]$.

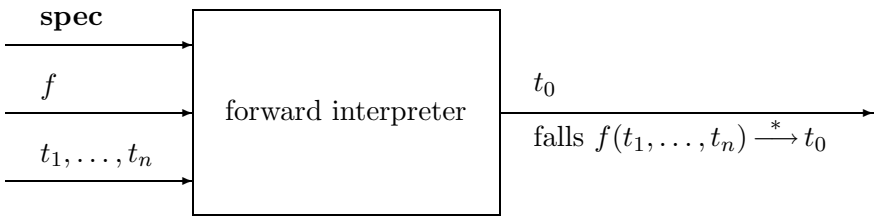

Abbildung 9: Ein Vorwärtsinterpreter für CE-S

Wie die folgende Beobachtung zeigt, gehen zwei Terme durch Termersetzung auseinander hervor, falls sie durch Gleichungsanwendung ineinander überführbar sind. Das rechtfertigt, Operationsanwendungen mit Hilfe von Gleichungsanwendungen auszuwerten und dabei eine Gleichungsanwendung als elementaren Berechnungsschritt zu deuten.

Lemma 1 (Gleichungsanwendung als Termersetzung)
Sei CE eine Menge bedingter Gleichungen zur Menge der Deklarationen DECL und Variablen X. Sei $\xrightarrow{*}$ *die zugehörige Termersetzung. Sei* $L = R$ falls $L_1 = R_1, \ldots, L_m = R_m$ *eine bedingte Gleichung aus CE. Seien C ein Kontextterm und a eine Wertzuweisung, so dass* $C[a_L] \longrightarrow C[a_R]$ *definiert ist. Gelte außerdem* $L_i[a] \xrightarrow{*} R_i[a]$ *für* $i = 1, \ldots, m$.

Dann folgt: $C[a_L] \xrightarrow{*} C[a_R]$.

Beweis (durch Induktion über den Aufbau von C)**.**

IA: Hier ist C nicht zusammengesetzt, muss aber die Extravariable enthalten, so dass $C = -$ gelten muss. Man erhält dann nach Definition der Substitution und der Wertzuweisungen a_L und a_R: $-[a_L] = a_L(-) = L[a]$ und $-[a_R] = a_R(-) = R[a]$, was nach 3.3.1(ii) und 3.5 $L[a] \xrightarrow{*} R[a]$ ergibt.

IV: Die Behauptung gelte für Terme mit Rekursionstiefe $\leq m$.

IS: Betrachte $C = f(t_1, \ldots, t_k)$ mit Rekursionstiefe $m + 1$. Dann haben die Argumentterme höchstens Rekursionstiefe m. Da C die Extravariable genau einmal enthält (und f keine Variable ist), enthält genau ein t_{i_0} die Extravariable, ist also ein Kontextterm. Insbesondere gilt damit $t_{i_0}[a_L] \rightarrow t_{i_0}[a_R]$, so dass nach Induktionsvoraussetzung $t_{i_0}[a_L] \xrightarrow{*} t_{i_0}[a_R]$ folgt. Die anderen sind normale Terme. Deshalb gilt nach Definition der Substitution:

$$C[a_L] = f(t_1, \ldots, t_k)[a_L] = f(t_1[a_L], \ldots, t_k[a_L]) = f(t_1, \ldots, t_{i_0-1}, t_{i_0}[a_L], t_{i_0+1}, \ldots, t_k),$$

und analog:

$$C[a_R] = f(t_1, \ldots, t_{i_0-1}, t_{i_0}[a_R], t_{i_0+1}, \ldots, t_k).$$

Dabei wird ausgenutzt, dass ein Term bei der Substitution unverändert bleibt, wenn er die Variablen, die substituiert werden, gar nicht enthält. Für die Terme t_i ($i \neq i_0$) gilt außerdem nach 3.3.1(iv): $t_i \xrightarrow{*} t_i$, so dass sich nach 3.3.1(iii) wunschgemäß ergibt:

$$f(t_1, \ldots, t_{i_0-1}, t_{i_0}[a_L], t_{i_0+1}, \ldots, t_k) \xrightarrow{*} f(t_1, \ldots, t_{i_0-1}, t_{i_0}[a_R], t_{i_0+1}, \ldots, t_k). \qquad \square$$

3.7 Beispiele: Sortieren durch Einsortieren und Mischen

Sortieren gehört zu den wichtigsten und häufig vorkommenden algorithmischen Problemen. Es ist daher kein Wunder, dass in der Literatur viele Verfahren zu finden sind. Aus

der Vielzahl sollen hier zwei vorgestellt werden, um mit ihrer Hilfe weitere Beispiele für den Umgang mit CE-S bereitzustellen. Diese Beispiele werden später bei der Aufwandsanalyse von Algorithmen wieder aufgegriffen.

Die Aufgabe des Sortierens für Zeichenketten besteht darin, die Zeichen jeder eingegebenen Zeichenkette so umzuordnen, dass je zwei aufeinanderfolgende Zeichen in der $\leq$-Beziehung stehen.

Beim Sortieren durch Einsortieren wird eine Hilfsoperation verwendet. Ein Zeichen wird in ein Wort einsortiert, indem es solange nach rechts geschoben wird, bis das nächste Zeichen nicht mehr kleiner ist oder das Ende erreicht ist. Sortieren lässt sich damit, indem die Zeichen eines Eingabewortes von rechts nach links in das jeweilige Zwischenergebnis einsortiert werden, wobei man mit dem leeren Wort beginnt. Die folgenden Spezifikationen des Einsortierens und Sortierens durch Einsortieren setzen diese Ideen rekursiv um.

insort

opns:	$\mathit{insort}\colon A \times A^* \to A^*,\ \mathit{sort}\colon A^* \to A^*$
vars:	$x, y \in A,\ u, v \in A^*$
eqns:	$\mathit{insort}(x, \lambda) = x$
	$\mathit{insort}(x, yv) = \text{if } x \leq y \text{ then } xyv \text{ else } y\,\mathit{insort}(x, v)$
	$\mathit{sort}(\lambda) = \lambda$
	$\mathit{sort}(xu) = \mathit{insort}(x, \mathit{sort}(u))$

Durch vollständige Induktion kann man sich ohne viel Mühe davon überzeugen, dass *insort* und *sort* Abbildungen auf den angegebenen Datenbereichen definieren, die durch Anwendung der Gleichungen für jede Wahl der Argumente ausgerechnet werden können. Es ist auch nicht schwer zu sehen, dass in einem Ergebnis von *sort* keine zwei aufeinanderfolgenden Zeichen in $>$-Beziehung stehen. Um einzusehen, dass das spezifische Verfahren sortiert, muss also nur noch sichergestellt werden, dass keine Zeichen verlorengegangen oder hinzugekommen sind. Das wird exemplarisch gezeigt. Dabei wird eine gegenüber der früheren Definition leicht modifizierte Spezifikation des Zeichenzählens verwendet:

count

opns:	$\mathit{count}\colon A \times A^* \to \mathbb{N},\ \delta\colon A \times A \to \mathbb{N}$
vars:	$x, y \in A,\ v \in A^*$
eqns:	$\mathit{count}(x, \lambda) = 0$
	$\mathit{count}(x, yv) = \mathit{count}(x, v) + \delta(x, y)$
	$\delta(x, y) = \text{if } x \equiv y \text{ then } 1 \text{ else } 0$

Beobachtung 2
Für alle $a, b \in A$ und $w \in A^*$ sind folgende Terme gleichwertig:

(1) $\mathit{count}(a, \mathit{insort}(b, w)) = \mathit{count}(a, w) + \delta(a, b)$
(2) $\mathit{count}(a, \mathit{sort}(w)) = \mathit{count}(a, w)$

Beweis (mit Induktion über den Aufbau von w)**.**

IA: (1) $count(a, insort(b, \lambda)) = count(a, b) = count(a, \lambda) + \delta(a, b)$, wobei zuerst die Definition von *insort* und dann von *count* verwendet wird.
(2) $count(a, sort(\lambda)) = count(a, \lambda)$ nach Definition von *sort*.

IV: Die Behauptungen gelten für $v \in A^*$.

IS: (1) Für $count(a, insort(b, yv))$ gibt es nach Definition von *insort* zwei Fälle:
$b \leq y$: Dafür ergibt sich nach Definition von *insort* und *count* wunschgemäß:

$$count(a, insort(b, yv)) = count(a, byv) = count(a, yv) + \delta(a, b).$$

$b > y$: In diesem Fall kann man nacheinander die Definition von *insort* und *count*, die Induktionsvoraussetzung, das Kommutativgesetz für natürliche Zahlen und wieder die Definition von *count* anwenden, um die Behauptung zu erhalten:

$$\begin{aligned} count(a, insort(b, yv)) &= count(a, y\, insort(b, v)) \\ &= count(a, insort(b, v)) + \delta(a, y) \\ &= count(a, v) + \delta(a, b) + \delta(a, y) \\ &= count(a, v) + \delta(a, y) + \delta(a, b) \\ &= count(a, yv) + \delta(a, b). \end{aligned}$$

(2) Hier folgt die Behauptung nach Definition von *sort*, dem gerade Gezeigten für *insort*, der Induktionsvoraussetzung und der Definition von *count*:

$$\begin{aligned} count(a, sort(xv)) &= count(a, insort(x, sort(v))) \\ &= count(a, sort(v)) + \delta(a, x) \\ &= count(a, v) + \delta(a, x) \\ &= count(a, xv). \end{aligned}$$

□

Das Sortieren durch Mischen funktioniert ähnlich, aber statt eines Zeichens wird ein ganzes Wort "einsortiert". Um ein Wort zu sortieren, wird es in eine linke und rechte Hälfte geteilt, die beide sortiert und danach zusammengemischt werden. Um das zu erreichen, werden Hilfsoperationen *left*, *right* und *merge* eingeführt.

mergesort

opns: $left, right, sort\colon A^* \to A^*$, $merge\colon A^* \times A^* \to A^*$
vars: $x, y \in A$, $u, v, w \in A^*$
eqns: $left(\lambda) = \lambda$
$left(x) = x$
$left(xvy) = x\, left(v)$
$right(\lambda) = \lambda$
$right(x) = \lambda$
$right(xvy) = right(v)y$
$merge(\lambda, v) = v$
$merge(u, \lambda) = u$
$merge(xu, yv) =$ if $x \leq y$ then $x\, merge(u, yv)$ else $y\, merge(xu, v)$
$sort(\lambda) = \lambda$
$sort(x) = x$
$sort(w) = merge(sort(left(w)), sort(right(w)))$ falls $length(w) \geq 2$

Einzusehen, dass dieses Verfahren wirklich sortiert, bleibt der Intuition überlassen. Aber es werden einige Eigenschaften bewiesen, die vor allem die Länge betreffen und später gebraucht werden. Dabei wird folgender Satz über die Länge von Wörtern vorausgesetzt (vgl. 2.4.3):

$$\mathit{length}(vw) = \mathit{length}(v) + \mathit{length}(w) \text{ für alle } v, w \in A^*.$$

Beobachtung 3
Für alle $u, v, w \in A^*$ sind folgende Terme gleichwertig:

(1) $\mathit{left}(w)\mathit{right}(w) = w$,
(2) $\mathit{length}(\mathit{right}(w)) \leq \mathit{length}(\mathit{left}(w)) \leq \mathit{length}(\mathit{right}(w)) + 1$,
(3) $\mathit{length}(\mathit{merge}(u, v)) = \mathit{length}(u) + \mathit{length}(v)$,
(4) $\mathit{length}(\mathit{sort}(w)) = \mathit{length}(w)$.

Beweis.
Die ersten beiden Punkte werden zusammen mit vollständiger Induktion über die Länge von Wörtern bewiesen.

IA: Sei $\mathit{length}(w) \leq 1$, d.h. $w = \lambda$ oder $w = x \in A$. Dann gilt nach Definition:

$\mathit{left}(\lambda)\mathit{right}(\lambda) = \lambda\, \mathit{right}(\lambda) = \lambda\lambda = \lambda$ sowie
$\mathit{left}(x)\mathit{right}(x) = x\, \mathit{right}(x) = x\lambda = x$.
$\mathit{length}(\mathit{left}(\lambda)) = \mathit{length}(\lambda) = \mathit{length}(\mathit{right}(\lambda)) \leq \mathit{length}(\mathit{right}(\lambda)) + 1$ sowie
$\mathit{length}(\mathit{left}(x)) = \mathit{length}(x) = \mathit{length}(\lambda) + 1 = \mathit{length}(\mathit{right}(x)) + 1 \geq \mathit{length}(\mathit{right}(x))$.

IV: Die Behauptungen (1) und (2) gelten für alle Wörter mit Länge kleiner $n \geq 2$.

IS: Betrachte ein Wort w der Länge $n \geq 2$. Dann gibt es $x, y \in A$ und $v \in A^*$ mit $w = xvy$.

Nach Definition von *left* und *right* sowie der Induktionsvoraussetzung ergibt sich dann:

$$\begin{aligned}\mathit{left}(w)\mathit{right}(w) &= \mathit{left}(xvy)\mathit{right}(xvy)\\ &= x\, \mathit{left}(v)\mathit{right}(xvy)\\ &= x\, \mathit{left}(v)\mathit{right}(v)y\\ &= xvy\\ &= w.\end{aligned}$$

Die Induktionsvoraussetzung ist anwendbar, weil v die Länge $n - 2$ hat.

Analog ergibt sich die Behauptung für die Längen:

$$\begin{aligned}\mathit{length}(\mathit{right}(w)) &= \mathit{length}(\mathit{right}(xvy))\\ &= \mathit{length}(\mathit{right}(v)y)\\ &= \mathit{length}(\mathit{right}(v)) + \mathit{length}(y)\end{aligned}$$

$$
\begin{aligned}
&= \mathit{length}(\mathit{right}(v)) + \mathit{length}(\lambda) + 1 \\
&= \mathit{length}(\mathit{right}(v)) + 0 + 1 \\
&= \mathit{length}(\mathit{right}(v)) + 1 \\
&\leq \mathit{length}(\mathit{left}(v)) + 1 \\
&= \mathit{length}(x\,\mathit{left}(v)) \\
&= \mathit{length}(\mathit{left}(xvy)) \\
&= \mathit{length}(\mathit{left}(w)).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{length}(\mathit{left}(w)) &= \mathit{length}(\mathit{left}(xvy)) \\
&= \mathit{length}(x\,\mathit{left}(v)) \\
&= \mathit{length}(\mathit{left}(v)) + 1 \\
&\leq \mathit{length}(\mathit{right}(v)) + 1 + 1 \\
&= \mathit{length}(\mathit{right}(v)) + 0 + 1 + 1 \\
&= \mathit{length}(\mathit{right}(v)) + \mathit{length}(\lambda) + 1 + 1 \\
&= \mathit{length}(\mathit{right}(v)) + \mathit{length}(y) + 1 \\
&= \mathit{length}(\mathit{right}(v)y) + 1 \\
&= \mathit{length}(\mathit{right}(xvy)) + 1 \\
&= \mathit{length}(\mathit{right}(w)) + 1.
\end{aligned}
$$

Der Nachweis der dritten Eigenschaft wird mittels Induktion über die Längensumme der beiden Argumentwörter geführt.

IA: Die Summe ist 0, wenn beide Argumente von *merge* leer sind. Dann gilt:

$$\mathit{length}(\mathit{merge}(\lambda, \lambda)) = \mathit{length}(\lambda) = \mathit{length}(\lambda) + 0 = \mathit{length}(\lambda) + \mathit{length}(\lambda).$$

IV: Die Behauptung gelte für alle Wörter, deren Längensumme n ist.

IS: Seien u und v Wörter mit $\mathit{length}(u) + \mathit{length}(v) = n + 1$. Falls $u = \lambda$ ist, gilt:

$$\mathit{length}(\mathit{merge}(\lambda, v)) = \mathit{length}(v) = 0 + \mathit{length}(v) = \mathit{length}(\lambda) + \mathit{length}(v).$$

Analog folgt die Behauptung für $v = \lambda$. Bleibt der Fall, dass u und v beide nicht leer sind, so dass $x, y \in A$ und $\overline{u}, \overline{v} \in A^*$ existieren mit $u = x\overline{u}$ und $v = y\overline{v}$. Gilt außerdem $x \leq y$, so ergibt sich die Behauptung wie folgt:

$$
\begin{aligned}
\mathit{length}(\mathit{merge}(u, v)) &= \mathit{length}(\mathit{merge}(x\overline{u}, y\overline{v})) \\
&= \mathit{length}(x\,\mathit{merge}(\overline{u}, y\overline{v})) \\
&= \mathit{length}(\mathit{merge}(\overline{u}, y\overline{v})) + 1 \\
&= \mathit{length}(\overline{u}) + \mathit{length}(y\overline{v}) + 1 \\
&= \mathit{length}(\overline{u}) + \mathit{length}(v) + 1 \\
&= \mathit{length}(\overline{u}) + 1 + \mathit{length}(v) \\
&= \mathit{length}(x\overline{u}) + \mathit{length}(v) \\
&= \mathit{length}(u) + \mathit{length}(v).
\end{aligned}
$$

Die Anwendung der Induktionsvoraussetzung ist möglich wegen:

$$
\begin{aligned}
&\mathit{length}(\overline{u}) + \mathit{length}(y\overline{v}) \\
&\quad = \mathit{length}(\overline{u}) + 1 - 1 + \mathit{length}(y\overline{v}) \\
&\quad = \mathit{length}(x\overline{u}) + \mathit{length}(y\overline{v}) - 1 \\
&\quad = \mathit{length}(u) + \mathit{length}(v) - 1 \\
&\quad = n + 1 - 1 \\
&\quad = n.
\end{aligned}
$$

Es bleibt noch die Alternative, dass $x > y$ ist, für die sich die Behauptung aber völlig analog ergibt.

Die vierte Behauptung ergibt sich wieder durch vollständige Induktion über die Länge von w, wobei die anderen drei Beobachtungen ausgenutzt werden. □

4 Wie beim Rechnen die Zeit vergeht

Lässt man ein Programm laufen, wertet einen Algorithmus aus, berechnet eine Operation, so vergeht dabei Zeit. Alle, die regelmäßig am Rechner sitzen, müssen warten können. Der Zeitaufwand beim Auswerten und Berechnen von Programmen ist ein Schlüsselfaktor, der Informations- und Kommunikationstechnik begrenzt und der in der Informatik als wissenschaftliche Disziplin untersucht wird. Dabei geht es theoretisch darum, das Phänomen richtig zu verstehen, und praktisch muss das Ziel sein, den Zeitaufwand bei angewandten Algorithmen möglichst gering zu halten. In der Komplexitätstheorie werden die Gesetzmäßigkeiten des Zeitaufwandes studiert und Methoden entwickelt, mit deren Hilfe für einzelne Algorithmen und algorithmische Probleme der Aufwand bestimmt oder zumindest abgeschätzt werden kann.

Will man den Zeitaufwand ermitteln, muss man sich zuerst überlegen, was überhaupt beim Berechnen von Algorithmen Zeit kostet und wie oft diese zeitverbrauchenden Ereignisse eintreten. Auf der untersten Rechnerebene misst man das meist in der Zahl der Instruktionen, die ausgeführt werden müssen. Kennt man die Zahl der Instruktionen, die ein bestimmter Rechner pro Sekunde schafft, erhält man dann eine echte Zeitabschätzung. Diese Art der Betrachtung hängt dann natürlich stark vom Stand der Rechnertechnik ab, in der es seit Jahrzehnten bemerkenswerte Fortschritte gibt. Während Wolfgang Coy in seinem Buch *Aufbau und Arbeitsweise von Rechenanlagen* 1992 noch 1 Mrd. Instruktionen pro Sekunde als Bestmarke einer *Cray* angibt, brüsten sich japanische Wissenschaftler nach einer kleinen Notiz in der *Frankfurter Rundschau* vom 27./28.5.1996 damit, 300 Mrd. erreicht zu haben.

Allerdings ist die Ebene der Rechnerinstruktionen direkt nur sehr bedingt brauchbar, weil Algorithmen höchst selten in Maschinencode geschrieben werden. Programme in höheren Programmiersprachen setzen sich dagegen aus Konstrukten und Operationen zusammen,

deren unmittelbarer Bezug zu Maschineninstruktionen oft unbekannt ist. Für Algorithmen, die auf einer hohen Abstraktionsebene formuliert und entwickelt werden, muss man sich deshalb eine andere geeignete Größe ausdenken, mit deren Hilfe sich Zeitaufwand erfassen lässt. Diese Zählgröße sollte mindestens zwei Eigenschaften besitzen:

(1) Sie spiegelt alles wider, was Zeit verbraucht.

(2) Sie lässt sich durch eine konstante Zahl von Instruktionen auf jedem vernünftigen Rechner implementieren.

Die erste Forderung stellt sicher, dass die Zählgröße keine Zeit-relevanten Faktoren außer acht lässt und somit wirklich Zeit erfasst wird. Die zweite Forderung garantiert, dass die Zählgröße wenigstens in der Größenordnung mit messbarem Zeitverbrauch übereinstimmt. Ein konkreter Versuch in die skizzierte Richtung wird in diesem Abschnitt unternommen, ein anderer am Beispiel der Matrizenmultiplikation im nächsten.

4.1 Zeitaufwand in CE-S

Der Versuch soll mit Hilfe von CE-S-Spezifikationen durchgeführt werden, was zu der Frage führt, inwiefern beim Berechnen in CE-S überhaupt Zeit vergeht. Betrachtet man an dieser Stelle den Vorwärtsinterpreter, gibt es nur eine naheliegende Antwort, weil eine Berechnung aus nichts anderem besteht als aus einer Sequenz von Gleichungsanwendungen. Bei der Anwendung einer Gleichung kann man einen Zeitverbrauch unterstellen (den man auch erleben kann, wenn man Gleichungsanwendungen selbst ausführt), weil sich der auszuwertende Term ändert. Die gesamte Berechnungszeit ist dann offenbar die Summe der Zeiten, die die einzelnen Schritte brauchen. Nimmt man an, dass jeder Schritt eine konstante Zeiteinheit braucht, ist die Berechnungszeit proportional zur Länge der Berechnungssequenz.

Am – noch sehr einfachen – Beispiel der Transposition wird die Idee sofort klar:

transposition
opns: $trans\colon A^* \to A^*$
vars: $x \in A, u \in A^*$
eqns: $trans(\lambda) = \lambda$
$trans(xu) = trans(u)x.$

Die Berechnung von *trans* mit dem Eingabewort *abc* sieht so aus:

$$trans(abc) = trans(bc)a = trans(c)ba = trans(\lambda)cba = \lambda cba.$$

Dabei wird die zweite *trans*-Gleichung dreimal und die erste einmal angewendet. Allgemein wird bei Anwendung der zweiten Gleichung das Argument der Transposition um 1 kürzer, was bei einem Eingabewort der Länge n gerade n-mal möglich ist. Dann muss einmal die erste Gleichung angewendet werden, so dass insgesamt $n + 1$ Schritte nötig sind.

Wenn man die Zahl der Gleichungsanwendungen, die nötig sind, um eine Operation *op* für ein Eingabewort der Länge n zu berechnen, mit $T^{op}(n)$ bezeichnet, kann man die obige Überlegung so ausdrücken:

$$T^{trans}(n) = n + 1 \text{ für alle } n \in \mathbb{N}.$$

Allgemein ausgedrückt, ist die Grundidee also, als Maß für den Zeitaufwand die Zahl der Gleichungsanwendungen zu bestimmen, die für die Auswertung eines Operationsaufrufs in einer CE-S-Spezifikation gebraucht werden, wobei das Ergebnis von der Länge der Eingabewörter abhängen kann.

4.2 Beispiel

Am (schon etwas komplizierteren) Beispiel des Sortierens durch Einsortieren, das in Punkt 3.8 spezifiziert ist, wird die skizzierte Vorgehensweise etwas genauer betrachtet und verfeinert.

Die erste Gleichung verrät, dass das Sortieren des leeren Wortes, das einzig die Länge 0 hat, in einem Schritt erfolgt:

(1) $\quad T^{sort}(0) = 1.$

Die zweite Gleichung behandelt alle nicht-leeren Wörter, die nach Definition jeweils aus einem Zeichen $x \in A$ und einem Restwort $u \in A^*$ durch Linksaddition xu entstehen. Dieses Wort hat genau dann die Länge $n + 1$, wenn u die Länge n hat. Die Anwendung der zweiten Gleichung auf $sort(xu)$ liefert:

(2) $\quad insort(x, sort(u)).$

Beim Weiterrechnen muss das Wort u sortiert und x in das Ergebnis einsortiert werden. Für den Aufwand ergibt sich also:

(3) $\quad T^{sort}(n + 1) = 1 + T^{insort}(m) + T^{sort}(n),$

wobei $m = length(sort(u))$ ist. Beachte, dass die Zeichen, die einsortiert werden, beim Aufwand nicht berücksichtigt werden, weil nach Vereinbarung nur Zeichenkettenargumente eingehen und Zeichen eines endlichen Alphabets als konstant groß angenommen werden können. Analog zu der Überlegung in Punkt 3.8, dass das Zeichenzählen vor und nach dem Sortieren gleiche Werte liefert, kann gezeigt werden, dass sich die Länge beim Sortieren durch Einsortieren nicht ändert. Somit folgt aus (3):

$$(4) \qquad T^{sort}(n+1) = 1 + T^{insort}(n) + T^{sort}(n).$$

Um die Eingabelänge um 1 zu verkürzen, muss man also eine Gleichung anwenden und den Aufwand des Sortierens und Einsortierens für ein Wort der Länge n in Kauf nehmen. Das führt für alle $n \in \mathbb{N}$ zu der nicht mehr rekursiven Formel:

$$(5) \qquad T^{sort}(n) = n + 1 + \sum_{i=1}^{n} T^{insort}(i-1),$$

wobei $\sum_{i=1}^{0} T^{insort}(i-1) = 0$ vereinbart wird.

Die aufgestellte Behauptung lässt sich durch vollständige Induktion über n beweisen:

IA: Die Aussage (1) und ein wenig Arithmetik zusammen mit der obigen Vereinbarung ergibt für $n = 0$ wunschgemäß:

$$T^{sort}(0) = 1 = 0 + 1 + 0 = 0 + 1 + \sum_{i=1}^{0} T^{insort}(i-1).$$

IV: Die Behauptung gelte für n.

IS: Für $n+1$ ergibt sich aus Aussage (4), der Induktionsvoraussetzung und wieder etwas Arithmetik die gewünschte Gleichheit:

$$\begin{aligned} T^{sort}(n+1) &= 1 + T^{insort}(n) + T^{sort}(n) \\ &= 1 + T^{insort}(n) + n + 1 + \sum_{i=1}^{n} T^{insort}(i-1) \\ &= (n+1) + 1 + \sum_{i=1}^{n+1} T^{insort}(i-1). \end{aligned}$$

Solange man den Aufwand für das Einsortieren nicht kennt, ist die Aussage (5) immer noch nicht allzu aussagekräftig. Um mehr zu erfahren, wird dasselbe Schema für *insort* durchgeführt. Die erste *insort*-Gleichung liefert:

$$(6) \qquad T^{insort}(0) = 1.$$

Die zweite Gleichung beschreibt, wie das Einsortieren bei Wörtern der Länge $n+1$ funktioniert. Es gibt zwei Fälle, die nach den Vereinbarungen zur Schreibweise mit *if-then-else* eigentlich auf zwei Gleichungen hinauslaufen. Im ersten Fall ist die Berechnung sofort abgeschlossen. Im zweiten Fall muss man nach der Gleichungsanwendung noch ein Zeichen in ein Wort der Länge n einsortieren. Das Ergebnis ist also 1 oder $1 + T^{insort}(n)$. In solchen Situationen wird mit dem größeren Wert weitergerechnet, d.h. mit dem schlechtesten Fall:

$$(7) \qquad T^{insort}(n+1) = max(1, 1 + T^{insort}(n)) = 1 + T^{insort}(n).$$

Wie beim Transponieren ergibt das für alle $n \in \mathbb{N}$:

$$(8) \qquad T^{insort}(n) = n + 1,$$

was sich mit Hilfe der Aussagen (6) und (7) leicht durch vollständige Induktion beweisen lässt.

Setzt man nun das Ergebnis (8) in die Aussage (5) ein und beachtet die bekannte Formel für die Summe der ersten $n + 1$ natürlichen Zahlen, so erhält man:

$$(9) \qquad T^{sort}(n) = n + 1 + \sum_{i=1}^{n}((i-1)+1)) = \sum_{i=1}^{n+1} i = \tfrac{(n+1)(n+2)}{2} = \tfrac{1}{2}n^2 + \tfrac{3}{2}n + 1.$$

Diese Formel drückt den Aufwand des Sortierens durch Einsortieren recht klar aus, dass man nämlich höchstens polynomiell viele Schritte braucht, um ein Eingabewort der Länge n zu sortieren, wobei das Polynom $\frac{1}{2}n^2 + \frac{3}{2}n + 1$ vom Grad 2 ist. Weil das quadratische Glied dabei am stärksten wächst, kann man kurz und einprägsam auch sagen, dass das Sortieren durch Einsortieren einen quadratischen Aufwand hat.

4.3 Aufwandsermittlung

Aus den bisherigen Überlegungen lässt sich eine Vorgehensweise bei der Aufwandsermittlung herausfiltern. Betrachtet wird eine CE-S-Spezifikation mit einer Operation $f : D_1 \times \cdots \times D_m \to D$. Jede definierende Gleichung hat als linke Seite einen Term der Form $f(t_1, \ldots, t_n)$, wobei $t_1, \ldots, t_n$ Grundterme sind, sich also nur aus Variablen und Grundoperationen zusammensetzen. Die bisher betrachteten Operationen haben gerade ein Argument aus A^*. Im folgenden dürfen auch mehrere Argumente von Operationen Zeichenketten sein. Es wird deshalb angenommen, dass k Argumentbereiche von f als Zeichenkettenbereiche deklariert sind. Seien das die Bereiche mit den Indizes $i_1, \ldots, i_k$ mit $1 \leq i_1 < i_2 < \cdots < i_k \leq m$. Mit $T^f(n_1, \ldots, n_k)$ wird dann die Zahl der Gleichungsanwendungen bezeichnet, die im schlechtesten Fall nötig ist, um einen Term $f(t_1, \ldots, t_n)$ mit $length(t_{i_j}) = n_j$ für $j = 1, \ldots, k$ und variablenfreien Grundtermen $t_1, \ldots, t_n$ mit Hilfe des Vorwärtsinterpreters auszuwerten, wobei die Berechnung erfolgreich ist, wenn man einen Grundterm als Wert erreicht. $T^f : \mathbb{N}^k \to \mathbb{N}$ wird *Aufwandsfunktion* von f genannt, wenn T^f für alle k-Tupel von natürlichen Zahlen definiert ist.

Für die Beispiele *trans*, *sort* und *insort* ist es gelungen, die Aufwandsfunktion genau zu bestimmen. Um die Arbeit zu erleichtern, wird im folgenden auch gestattet, T^f nach oben abzuschätzen. Dabei kann eine Funktion $g\colon \mathbb{N}^k \to \mathbb{R}^+$ verwendet werden, die bis auf endlich viele Ausnahmen und bis auf einen konstanten Faktor und einen konstanten

Summanden eine obere Schranke von T^k bilden soll. Es soll also für alle $n_1, \ldots, n_k \geq n_0$, wobei $n_0 \in \mathbb{N}$ geeignet gewählt sei, und für geeignet gewählte Konstanten $c, c_0 \in \mathbb{R}^+$ gelten:

$$(*) \qquad T^f(n_1, \ldots, n_k) \leq c \cdot g(n_1, \ldots, n_k) + c_0.$$

Wenn das der Fall ist, spricht man davon, dass T^f in der *Aufwandsklasse* von g liegt und schreibt $T^f \in O(g)$ (sprich: groß O von g). Der Trick ist die geschickte Wahl von g, n_0, c und c_0. Typische Fälle sind für $k = 1$ die Funktionen g_0, g_1, g_2, g_3, g_4, g_5 mit $g_0(n) = 1$, $g_1(n) = ld\ n$, $g_2(n) = n$, $g_3(n) = n^2$, $g_4(n) = n^3$, $g_5(n) = 2^n$ für alle $n \in \mathbb{N}$. Von einer Operation, deren Aufwandsfunktion in die Klasse $O(1)$ ($O(ld\ n)$, $O(n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$) fällt, sagt man dann, sie habe *konstanten* (*logarithmischen, linearen, quadratischen, kubischen* bzw. *exponentiellen*) Aufwand. Demnach haben *trans* und *insort* linearen Aufwand und *sort* ist ein Sortierverfahren mit quadratischem Aufwand (wobei ausgenutzt wird, dass $n \leq n^2$ gilt und somit $T^{sort}(n) \leq 2n^2 + 1$).

Bei der Aufwandsabschätzung geht es darum, eine Behauptung der Form $(*)$ aufzustellen, wobei die obere Schranke möglichst dicht am exakten Ergebnis liegen sollte. Der Beweis der Behauptung kann normalerweise mit Hilfe vollständiger Induktion über ein Argument oder mehrere Argumente oder eine geeignet abgeleitete Größe durchgeführt werden. Bei der Aufstellung der Behauptung kann es helfen, die definierenden Gleichungen genau anzusehen. Denn wenn man variablenfreie Grundterme in die Variablen einsetzt, entsteht aus der linken Seite ein variablenfreier Term der Form $f(t_1, \ldots, t_n)$, dessen Unterterme $t_1, \ldots, t_n$ Grundterme sind. Insbesondere sind die Grundterme aus Zeichenkettenbereichen nur aus Zeichen, dem leeren Wort, der Linksaddition und der Konkatenation aufgebaut und damit Zeichenketten in üblichen Sinne. Der Übergang von der linken zur rechten Seite ist eine Gleichungsanwendung und trägt den Summanden 1 zur Bestimmung von $T^f(n_1, \ldots, n_k)$ mit $length(t_{i_j}) = n_j$ für $j = 1, \ldots, k$ bei. Dann muss die rechte Seite ausgerechnet werden, was für jede dort vorkommende Operation, die nicht Grundoperation ist, einen bestimmten Aufwand bedeutet, der additiv dazukommt. Das klingt kompliziert, ist im Einzelfall manchmal jedoch recht einfach, wie die Aussagen (1), (3), (6) und (7) in 4.2 zeigen.

Da man den Aufwand üblicherweise nach Rechenschritten zählt, darf man getrost annehmen, dass zumindest ab n_0 $g(n_1, \ldots, n_k) \geq 1$ ist. Dann ergibt sich aus $(*)$:

$$\begin{aligned} T^f(n_1, \ldots, n_k) &\leq c \cdot g(n_1, \ldots, n_k) + c_0 \\ &\leq c \cdot g(n_1, \cdots, n_k) + c_0 \cdot g(n_1, \ldots, n_k) \\ &= (c + c_0) \cdot g(n_1, \ldots, n_k). \end{aligned}$$

Statt $(*)$ ist also $T^f \in O(g)$ auch dadurch charakterisiert, dass für geeignet gewählte $n_0 \in \mathbb{N}$ und $\overline{c} \in I\!R^+$ und für alle $n_1, \ldots, n_k \geq n_0$ gilt:

$$(**) \qquad T^f(n_1 \cdots n_k) \leq \overline{c} \cdot g(n_1 \cdots n_k).$$

Manche Abschätzungen für Aufwandsklassen lassen sich in dieser Form leichter vornehmen.

4.4 Beispiel

Als weiteres Beispiel wird der Aufwand des Sortierens durch Mischen ermittelt, das in Punkt 3.7 spezifiziert ist. Um eine Zeichenkette w in **mergesort** zu sortieren, muss für die Längen 0 und 1 je eine Gleichung angewendet werden. Ab der Länge 2 muss die dritte Gleichung angewendet werden und dann müssen die Operationsaufrufe auf der rechten Seite der Gleichung ausgewertet werden. Das heißt

$$T^{sort}(0) = T^{sort}(1) = 1$$

$$T^{sort}(n) = 1 + T^{merge}(\overline{n}_1, \overline{n}_2) + T^{left}(n) + T^{right}(n) + T^{sort}(n_1) + T^{sort}(n_2),$$

wobei $n_1 = \mathit{length}(\mathit{left}(u))$, $n_2 = \mathit{length}(\mathit{right}(u))$, $\overline{n}_1 = \mathit{length}(\mathit{sort}(\mathit{left}(u)))$ und $\overline{n}_2 = \mathit{length}(\mathit{sort}(\mathit{right}(u)))$, falls $n = \mathit{length}(u) \geq 2$. Nach 3.7 gilt $n_1 = n_2 = \overline{n}_1 = \overline{n}_2 = m$, falls $n = 2m$, sowie $n_1 = \overline{n}_1 = m + 1$ und $n_2 = \overline{n}_2 = m$, falls $n = 2m + 1$, d.h.

$$T^{sort}(2m) = 1 + T^{merge}(m, m) + T^{left}(2m) + T^{right}(2m) + 2T^{sort}(m)$$

$$T^{sort}(2m + 1) = \\ 1+T^{merge}(m+1, m)+T^{left}(2m+1)+T^{right}(2m+1)+T^{sort}(m+1)+T^{sort}(m).$$

Um weiterzukommen, muss man den Aufwand von *merge*, *left* und *right* kennen. Bei diesen drei Operationen sind die Verhältnisse ähnlich zu *trans* und *insort*. Bei der Rekursion von *merge* wird ein Argument in jedem Schritt um Eins kürzer. Ist ein Argument leer (geworden), muss noch eine Gleichung angewendet werden, so dass sich im schlechtesten Fall

$$T^{merge}(m, n) = m + n$$

ergibt, wenn nicht beide Argumente 0 sind. Bei *left* und *right* muss man beachten, dass in der Rekursion die Argumente um 2 kleiner werden, was für $m \geq 0$

$$T^{left}(2m) = T^{right}(2m) = T^{left}(2m + 1) = T^{right}(2m + 1) = m + 1$$

ergibt. Diese Behauptungen lassen sich analog zu den Überlegungen bei T^{trans} und T^{insort} beweisen, weshalb hier darauf verzichtet wird. Setzt man nun die Ergebnisse in die obigen Gleichheiten für T^{sort} ein, so erhält man für $m \geq 1$

$$\begin{aligned}
\text{(a)}\quad T^{sort}(2m) &= 1 + 2m + 2(m+1) + 2T^{sort}(m) \\
&= 3 + 4m + 2T^{sort}(m) \\
\text{(b)}\quad T^{sort}(2m+1) &= 1 + 2m + 1 + 2(m+1) + T^{sort}(m+1) + T^{sort}(m) \\
&= 4 + 4m + T^{sort}(m+1) + T^{sort}(m).
\end{aligned}$$

In beiden Rekursionen wird das linke Argument halbiert, was bekanntlich nur logarithmisch oft mit ganzzahligem Ergebnis geht. Außerdem wird ein Vielfaches des Arguments aufaddiert. Das führt zu der Vermutung, dass der Aufwand des Sortierens durch Mischen für $n \geq 2$ folgendermaßen abschätzbar ist:

$$(*) \qquad T^{sort}(n) \leq 5 \cdot n \cdot k$$

wobei k eindeutig so gewählt ist, dass $2^{k-1} < n \leq 2^k$ ist, also als kleinste natürliche Zahl die größer oder gleich dem dualen Logarithmus von n ist.

Die Behauptung kann mit vollständiger Induktion über k gezeigt werden.

Induktionsanfang für $k = 1$, d.h. $n = 2$:

$$T^{sort}(2) \underset{(a)}{=} 3 + 4 \cdot 1 + 2 \cdot T^{sort}(1) = 7 + 2 \cdot 1 = 9 \leq 5 \cdot 2 \cdot 1.$$

Als Induktionsvoraussetzung (IV) gelte die Behauptung für $k \geq 1$, d.h. für alle n mit $2^{k-1} < n \leq 2^k$.

Induktionsschluss : Betrachte $k+1$ und damit ein n mit $2^k < n \leq 2^{k+1}$. Dann können die Fälle (i) $n = 2m$ und (ii) $n = 2m + 1$ unterschieden werden.

(i) Nach (a) und IV gilt dann wegen $2^{k-1} < m \leq 2^k$

$$\begin{aligned}
T^{sort}(2m) &= 3 + 4m + 2T^{sort}(m) \\
&\leq 3 + 4m + 2 \cdot 5m \cdot k \\
&\leq 2 \cdot 5m + 2 \cdot 5m \cdot k \\
&= 5 \cdot 2m(k+1).
\end{aligned}$$

(ii) Da $n = 2m+1$ ungerade ist, gilt $2^k < 2m+1 < 2^{k+1}$, so dass $2^{k-1} \leq m < m+1 \leq 2^k$ folgt. Nach (b) und IV erhält man deshalb für $2^{k-1} < m$

$$\begin{aligned}
T^{sort}(2m+1) &= 4 + 4m + T^{sort}(m+1) + T^{sort}(m) \\
&= 4 + 4m + 5(2m+1)k \\
&\leq 5 + 10m + 5(2m+1)k \\
&= 5(2m+1) + 5(2m+1)k \\
&= 5(2m+1)(k+1).
\end{aligned}$$

Insgesamt ist damit gezeigt, dass

$$T^{sort} \in O(n\lceil ld\, n \rceil)$$

ist, wenn $\lceil x \rceil$ für $x \in \mathbb{R}^+$ die nächst größere natürliche Zahl bezeichnet, bzw. genauer gilt: $\lceil x \rceil \in \mathbb{N}$ mit $\lceil x \rceil - 1 < x \leq \lceil x \rceil$.

5 Matrizenmultiplikation schneller und schneller

Die Matrizenrechnung als das Herzstück der linearen Algebra kann für die exakte Beschreibung, Planung und Berechnung vieler technischer und ökonomischer Prozesse eingesetzt werden. Insbesondere die Matrizenmultiplikation gehört zu den meistverwendeten Algorithmen in der numerischen Datenverarbeitung, so dass selbst eine geringfügige Verbesserung des Laufzeitverhaltens einen ökonomischen Nutzeffekt haben könnte. Die Multiplikation von Matrizen zu beschleunigen hat aber nicht nur eine eminent praktische Bedeutung, sondern eignet sich auch für erste Aufwandsbetrachtungen, weil vergleichsweise leicht herausgefunden werden kann, was den Aufwand dieses Algorithmus ausmacht.

5.1 Datenstruktur Matrix

Zur Erinnerung: Eine (m, n)-Matrix mit m Zeilen und n Spalten ist ein rechteckiges Zahlenschema

$$A = (a_{ij})_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix},$$

das in PASCAL und ähnlichen Sprachen etwa als zweidimensionales Feld vereinbart werden könnte:

> *type* matrix = *array*$[1..m, 1..n]$ *of real* (oder *integer*);
> *var* A: matrix (mit $A[i, j] = a_{ij}$).

5.2 Beispielsweise Materialverflechtung

Bevor in Punkt 3 die klassische Version der Matrizenmultiplikation diskutiert wird, soll kurz ein kleines Beispiel der Materialverflechtung gezeigt werden, damit sich die praktische Bedeutung von Matrizen und ihren Produkten erahnen lässt.

Gegeben seien zwei Matrizen $A = (a_{ij})_{i=1,\ldots,m,\, j=1,\ldots,n}$ und $B = (b_{jk})_{j=1,\ldots,n,\, k=1,\ldots,p}$, deren Einträge so zu interpretieren sind:

> a_{ij} ist die Zahl der Einheiten des Rohstoffs i, die benötigt wird, um eine Einheit des Zwischenprodukts j herzustellen;
>
> b_{jk} ist die Zahl der Einheiten des Zwischenprodukts j, die benötigt wird, um eine Einheit des Endprodukts k herzustellen.

Ein Zahlenbeispiel dafür:

$$A = \begin{pmatrix} 4 & 2 \\ 0 & 5 \\ 1 & 3 \end{pmatrix} \text{ und } B = \begin{pmatrix} 0 & 7 & 1 & 1 \\ 2 & 3 & 2 & 0 \end{pmatrix},$$

wobei die konkrete Natur der betroffenen Produkte der Phantasie überlassen bleibt.

Für eine Gesamtplanung wäre es wichtig, aus den Einzelinformationen zu bestimmen, wieviele Einheiten des Rohstoffs i erforderlich sind, um eine Einheit des Endprodukts k herzustellen. Wird diese Größe mit c_{ik} bezeichnet, ist also die Frage: Wie lässt sich c_{ik} für $i = 1, \ldots, m$ und $k = 1, \ldots, p$ aus A und B berechnen?

Seien R_i, Z_j, E_k Abkürzungen für "Rohstoff i", "Zwischenprodukt j" beziehungsweise "Endprodukt k". Dann ergibt sich die Antwort für E_3 und R_1 im Zahlenbeispiel folgendermaßen:

E_3 braucht $b_{13} = 1$ Z_1 und $b_{23} = 2$ Z_2;
Z_1 braucht $a_{11} = 4$ R_1;
Z_2 braucht $a_{12} = 2$ R_1;
E_3 braucht also $a_{11} \cdot b_{13} = 4 \cdot 1$ und $a_{12} \cdot b_{23} = 2 \cdot 2$ R_1;
d.h. $c_{13} = a_{11} \cdot b_{13} + a_{12} \cdot b_{23} = 4 \cdot 1 + 2 \cdot 2 = 8$.

Allgemein erhält man entsprechend für E_k und R_i:

E_k braucht b_{1k} Z_1 und b_{2k} Z_2 und $\ldots$ und b_{nk} Z_n;
Z_j braucht a_{ij} R_i für $j = 1, \ldots, n$;
E_k braucht also $a_{i1} \cdot b_{1k}$ und $\ldots$ und $a_{in} \cdot b_{nk}$ R_i;
d.h. $c_{ik} = a_{i1} \cdot b_{1k} + a_{i2} \cdot b_{2k} + \cdots + a_{in} \cdot b_{nk} = \sum_{j=1}^{n} a_{ij} \cdot b_{jk}$.

Beachte, dass in die Berechnung von c_{ik} lediglich die i-te Zeile von A und die k-te Spalte von B eingehen, indem korrespondierende Einträge (die in Zeile und Spalte an gleicher Stelle stehen und dasselbe Zwischenprodukt betreffen) multipliziert und die Produkte aufsummiert werden.

Insgesamt entsteht durch diesen Vorgang aus den beiden Matrizen A und B eine neue Matrix $C = (c_{ik})_{i=1,\ldots,m,\ k=1,\ldots,p}$. Da das nicht nur für die diskutierte ökonomische Situation, sondern auch in vielen anderen Anwendungen sinnvoll interpretiert werden kann, wird die Berechnung von C aus A und B als eigenständige Matrizenoperation eingeführt.

5.3 Matrizenmultiplikation

Sei A eine (m, n)-Matrix und B eine (n, p)-Matrix. Dann ist das *Produkt* von A und B eine (m, p)-Matrix C, deren Einträge definiert sind durch:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \cdot b_{jk} \quad \text{für } i = 1, \ldots, m,\ k = 1, \ldots, p.$$

Die Produktmatrix wird auch mit $A \circ B$ bezeichnet.

Aus der i-ten Zeile von A und der k-ten Spalte von B ergibt sich also der Eintrag von C an der (i,k)-ten Stelle als Produktsumme:

$$\begin{pmatrix} \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots \end{pmatrix} \circ \begin{pmatrix} \cdots & b_{1k} & \cdots \\ \cdots & b_{2k} & \cdots \\ & \vdots & \\ \cdots & b_{nk} & \cdots \end{pmatrix} = \begin{pmatrix} & \vdots & \\ \cdots & \sum_{j=1}^{n} a_{ij} \cdot b_{jk} & \cdots \\ & \vdots & \end{pmatrix}$$

Die Definition des Produkts legt einen Algorithmus nahe, der im wesentlichen aus drei *for*-Schleifen besteht. Die ersten beiden erlauben den Zugriff auf jede Stelle des Produktfeldes, in der dritten Schleife wird die Summe iteriert:

```
for i := 1 to m do
    for k := 1 to p do
    begin
        C[i,k] := A[i,1] · B[1,k];
        for j := 2 to n do
            C[i,k] := C[i,k] + A[i,j] · B[j,k]
    end.
```

Anhand der Definition und dieses Programmstücks lässt sich nun ermitteln, welche Aktionen bei der Ausführung des Produktalgorithmus einen zeitlichen Aufwand verursachen. Um einen Eintrag zu berechnen, müssen die Werte der zugewiesenen Ausdrücke bestimmt werden, was $n-1$ Additionen und n Multiplikationen erfordert. Dieser Vorgang muss insgesamt $(m \cdot p)$-mal wiederholt werden. Zugriffe und Zuweisungen, die in der vorliegenden Form der Matrizenmultiplikation die einzigen zusätzlichen Aktivitäten darstellen, müssen nicht extra gezählt werden, weil sie immer nur im Zusammenhang mit dem Addieren und Multiplizieren von Zahlen stehen und deshalb dort mit eingerechnet werden können.

Für quadratische Matrizen mit $m = n = p$ ergibt sich also speziell folgender Rechenaufwand:

$$n^2 \cdot (n-1) = n^3 - n^2 \text{ Additionen}, \quad n^3 \text{ Multiplikationen}.$$

Diese Angabe lässt sich auch etwas anders schreiben:

$$T_{ADD}^{klassisch}(n) = n^3 - n^2, \quad T_{MULT}^{klassisch}(n) = n^3.$$

Dabei steht T für "time" und erinnert an den im Moment interessierenden Zeitaufwand von Algorithmen; verweist die obere Beschriftung auf den aktuell betrachteten Algorithmus, der in diesem Fall die klassische Art der Matrizenmultiplikation ist; nennt die untere Beschriftung die jeweilige Größe, die gezählt beziehungsweise gerechnet wird; wird eine Formatangabe der Eingabe in Klammern gesetzt, die in diesem Fall die Zahl der Zeilen und Spalten der multiplizierten Matrizen ist; und wird schließlich rechts vom Gleichheitszeichen das Ergebnis geschrieben. Diese Art, den Aufwand anzugeben, wird später helfen, mit Aufwänden umzugehen und zu rechnen.

5.4 Algorithmus von Winograd

Das Produkt von Matrizen ist seit langem bekannt, und die klassische Definition in Punkt 3 war bis in die 60er Jahre hinein ausreichend, um das Produkt zu bestimmen. Als jedoch Matrizen immer häufiger und mit sehr großer Zeilen- und Spaltenzahl auf dem Computer multipliziert wurden, begannen einige Wissenschaftlerinnen und Wissenschaftler zu fragen, ob das nicht schneller geht. Winograd war einer der ersten, der einen brauchbaren Vorschlag hatte. Seine Überlegung ging davon aus, dass Multiplikationen auf vielen Rechnersystemen länger dauern als Additionen und deshalb Zeit gespart wird, wenn man mit weniger Multiplikationen auskommt, selbst wenn die Zahl der Additionen dabei etwas steigt. Das Verfahren von Winograd nutzt aus, dass bei der Ausmultiplikation von Summen $(a + b)$ und $(c + d)$ Summanden entstehen, wie sie für die Berechnung der Produkteinträge gebraucht werden:

Seien A, B zwei (n, n)-Matrizen mit $n = 2m$. Sei A_i die i-te Zeile von A und B^k die k-te Spalte von B. Sei für einen Vektor $X = (x_1, \ldots, x_n)$ hilfsweise eine Produktsumme definiert durch:

$$W(X) = \sum_{j=1}^{m} x_{2j-1} \cdot x_{2j} = x_1 \cdot x_2 + x_3 \cdot x_4 + \cdots + x_{n-1} \cdot x_n.$$

Dann lässt sich das Produkt C von A und B berechnen durch:

$$c_{ik} = \sum_{j=1}^{m} (a_{i(2j-1)} + b_{(2j)k}) \cdot (a_{i(2j)} + b_{(2j-1)k}) - W(A_i) - W(B^k) \text{ für } i, k = 1, \ldots, n.$$

Unter Ausnutzung des Distributiv- und des Kommutativgesetzes für die Addition und Multiplikation von Zahlen kann gezeigt werden, dass das Verfahren korrekt ist, d.h. dass die berechneten Einträge tatsächlich die in Punkt 3 für das Produkt definierten sind:

$$\begin{aligned}
&\sum_{j=1}^{m} (a_{i(2j-1)} + b_{(2j)k}) \cdot (a_{i(2j)} + b_{(2j-1)k}) \\
&= \sum_{j=1}^{m} (a_{i(2j-1)} \cdot a_{i(2j)} + a_{i(2j-1)} \cdot b_{(2j-1)k} + b_{(2j)k} \cdot a_{i(2j)} + b_{(2j)k} \cdot b_{(2j-1)k}) \\
&= W(A_i) + \sum_{j=1}^{m} (a_{i(2j-1)} \cdot b_{(2j-1)k} + a_{i(2j)} \cdot b_{(2j)k}) + W(B^k) \\
&= \sum_{j=1}^{n} a_{ij} \cdot b_{jk} + W(A_i) + W(B^k).
\end{aligned}$$

Zieht man nun von beiden Seiten $W(A_i)$ und $W(B^k)$ ab, so erweist sich die gewünschte Beziehung als richtig.

Es bleibt, den Aufwand zu bestimmen. Die Hilfsberechnung W muss für die n Zeilen von A und die n Spalten von B ausgeführt werden; also muss in $2n$ Fällen m-mal multipliziert und $(m - 1)$-mal addiert werden:

$$T^W_{MULT}(n) = 2n \cdot m = n^2, \;\; T^W_{ADD}(n) = 2n \cdot (m-1) = n^2 - 2n.$$

Der Summenausdruck bei der Berechnung von c_{ik} erfordert $m-1$ Additionen und 2 weitere in jedem der m Summanden sowie 1 Multiplikation in jedem Summanden:

$$T^\Sigma_{MULT}(n) = m = \tfrac{1}{2}n, \;\; T^\Sigma_{ADD}(n) = 2m + m - 1 = \tfrac{3}{2}n - 1.$$

Daraus ergibt sich die Zahl der Additionen und Multiplikationen für den Winograd-Algorithmus, wenn man bedenkt, dass n^2 Einträge bestimmt werden müssen und bei jedem Eintrag zu dem Summenausdruck $\sum$ noch 2 Additionsoperationen (Subtraktionen) hinzukommen:

$$T^{WINOGRAD}_{MULT}(n) = n^2 \cdot T^\Sigma_{MULT}(n) + T^W_{MULT}(n) = \tfrac{1}{2}n^3 + n^2,$$
$$T^{WINOGRAD}_{ADD}(n) = n^2 \cdot (T^\Sigma_{ADD}(n) + 2) + T^W_{ADD}(n) = \tfrac{3}{2}n^3 + 2n^2 - 2n.$$

Beachte, dass dieser Algorithmus gegenüber der klassischen Methode Multiplikationen zuungunsten von Additionen einspart, so dass nur ein Laufzeitgewinn eintreten kann, wenn Multiplizieren langsamer als Addieren ist.

5.5 Algorithmus von Strassen

Einen Weg, wie man beim Matrizenprodukt echt unter n^3 elementaren Rechenoperationen bleiben kann, hat Strassen 1969 gewiesen.

Für $(2,2)$-Matrizen A, B ist sein Verfahren ein verwirrendes Rechenexempel, dessen wesentliches Merkmal ist, dass nur 7 Multiplikationen statt der sonst üblichen 8 gebraucht werden. Nach Strassen erhält man die Einträge der Produktmatrix C wie folgt:

$s_1 = a_{21} + a_{22}$	$s_2 = s_1 - a_{11}$	$s_3 = a_{11} - a_{21}$	$s_4 = a_{12} - s_2$
$s_5 = b_{12} - b_{11}$	$s_6 = b_{22} - s_5$	$s_7 = b_{22} - b_{12}$	$s_8 = s_6 - b_{21}$
$m_1 = s_2 \cdot s_6$	$m_2 = a_{11} \cdot b_{11}$	$m_3 = a_{12} \cdot b_{21}$	$m_4 = s_3 \cdot s_7$
$m_5 = s_1 \cdot s_5$	$m_6 = s_4 \cdot b_{22}$	$m_7 = a_{22} \cdot s_8$	
$t_1 = m_1 + m_2$	$t_2 = t_1 + m_4$	$t_3 = t_1 + m_5$	
$c_{11} = m_2 + m_3$	$c_{21} = t_2 - m_7$	$c_{12} = t_3 + m_6$	$c_{22} = t_2 + m_5$

Programmieren ließe sich das einfach als Folge von Zuweisungen (anstelle der Gleichungen). Etwas mehr Anstrengung ist nötig, wenn man sich vergewissern will, dass die gewünschten Ergebnisse herauskommen. Neben dem wiederholten Einsetzen braucht man dazu einige Rechengesetze für Addition und Multiplikation, wobei das Multiplikationszeichen oft weggelassen wird:

(1) $c_{11} = m_2 + m_3 = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$;

$$
\begin{aligned}
(2)\quad c_{21} &= t_2 - m_7 \\
&= t_1 + m_4 - a_{22}s_8 \\
&= t_1 + m_4 - a_{22}(s_6 - b_{21}) \\
&= t_1 + m_4 - a_{22}(b_{22} - s_5) + a_{22}b_{21} \\
&= t_1 + m_4 - a_{22}b_{22} + a_{22}(b_{12} - b_{11}) + a_{22}b_{21} \\
&= t_1 + m_4 - a_{22}b_{22} + a_{22}b_{12} - a_{22}b_{11} + a_{22}b_{21} \\
&= a_{21}b_{11} + a_{22}b_{21},
\end{aligned}
$$

wobei die letzte Gleichung gilt wegen

$$
\begin{aligned}
t_1 &= m_1 + m_2 \\
&= s_2s_6 + a_{11} \cdot b_{11} \\
&= (s_1 - a_{11})(b_{22} - s_5) + a_{11} \cdot b_{11} \\
&= (a_{21} + a_{22})b_{22} - (a_{21} + a_{22})(b_{12} - b_{11}) - a_{11}b_{22} + a_{11}(b_{12} - b_{11}) + a_{11} \cdot b_{11} \\
&= a_{21}b_{22} + a_{22}b_{22} - a_{21}b_{12} + a_{21}b_{11} - a_{22}b_{12} + a_{22}b_{11} - a_{11}b_{22} + a_{11}b_{12}
\end{aligned}
$$

und

$$
m_4 = s_3 \cdot s_7 = (a_{11} - a_{21})(b_{22} - b_{12}) = a_{11}b_{22} - a_{11}b_{12} - a_{21}b_{22} + a_{21}b_{12};
$$

$$
\begin{aligned}
(3)\quad c_{12} &= t_3 + m_6 \\
&= t_1 + m_5 + s_4b_{22} \\
&= t_1 + m_5 + (a_{12} - s_2)b_{22} \\
&= t_1 + m_5 + a_{12}b_{22} - (s_1 - a_{11})b_{22} \\
&= t_1 + m_5 + a_{12}b_{22} - (a_{21} + a_{22})b_{22} + a_{11}b_{22} \\
&= t_1 + m_5 + a_{12}b_{22} - a_{21}b_{22} - a_{22}b_{22} + a_{11}b_{22} \\
&= a_{11}b_{12} + a_{12}b_{22},
\end{aligned}
$$

wobei die letzte Gleichung gilt wegen

$$
m_5 = s_1 \cdot s_5 = (a_{21} + a_{22})(b_{12} - b_{11}) = a_{21}b_{12} - a_{21}b_{11} + a_{22}b_{12} - a_{22}b_{11};
$$

$$
(4)\quad c_{22} = t_2 + m_5 = t_1 + m_4 + m_5 = a_{21}b_{12} + a_{22}b_{22}.
$$

Nach diesem Verfahren ist also der Aufwand zum Multiplizieren von $(2,2)$-Matrizen:

$$
T_{MULT}^{STRASSEN}(2) = 7, \;\; T_{ADD}^{STRASSEN}(2) = 15.
$$

Wie lässt sich diese Einsparung einer Multiplikation nun auf größere Matrizen übertragen? Dazu muss man sich vergegenwärtigen, dass die gesamte vorausgegangene Berechnung nirgendwo davon Gebrauch gemacht hat, dass mit Zahlen gerechnet wird. Vielmehr wird nur verwendet, dass eine Addition existiert, die assoziativ und kommutativ ist und durch Subtraktion aufgehoben werden kann, und dass es außerdem eine Multiplikation gibt, die sich distributiv zur Addition verhält. Mit anderen Worten muss nur vorausgesetzt werden, dass die Einträge der zu multiplizierenden Matrizen einen Ring bilden, damit das Verfahren von Strassen greift. Insbesondere können also die Einträge der $(2,2)$-Matrizen auch (m,m)-Matrizen sein, die mit der normalen komponentenweisen Addition und der Matrizenmultiplikation eine Ringstruktur haben. Geht man nun von zwei (n,n)-Matrizen

A, B mit $n = 2m$ aus und unterteilt beide längs und quer in der Mitte, entstehen $(2,2)$-Matrizen mit (m,m)-Matrizen als Einträgen, deren Produkt identisch ist mit dem Produkt von A und B, nachdem das ebenfalls längs und quer geteilt worden ist:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \circ \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Bewiesen wird das in der linearen Algebra, lässt sich aber auch leicht nachrechnen. Die Konsequenz dieser Überlegung ist, dass das Produkt von zwei (n,n)-Matrizen nach Strassen mit 7 Multiplikationen und 15 Additionen von (m,m)-Matrizen berechnet werden kann, was folgenden Aufwand ergibt:

$$T_{MULT}^{STRASSEN}(n) = 7 \cdot T_{MULT}^{STRASSEN}(m),$$
$$T_{ADD}^{STRASSEN}(n) = 15 \cdot T_{ADD}^{+}(m) + 7 \cdot T_{ADD}^{STRASSEN}(m).$$

Dabei verweist der Exponent + auf die normale Addition von Matrizen. Zur Berechnung der 7 Multiplikationen von (m,m)-Matrizen muss auch addiert werden, so dass dieser Aufwand zum gesamten Additionsaufwand zuzuzählen ist. Schließlich werden für eine Addition von (m,m)-Matrizen gerade m^2 Additionen in den m^2 Komponenten ausgeführt, so dass gilt:

$$T_{ADD}^{STRASSEN}(n) = 15 \cdot m^2 + 7 \cdot T_{ADD}^{STRASSEN}(m).$$

Für die Matrizen mit halbiertem Format lässt sich genauso verfahren, solange das Format eine gerade Zahl ist. Mit dieser Rekursion kann demnach das Matrizenprodukt vollständig ausgerechnet werden, wenn das ursprüngliche Format eine 2er-Potenz ist: $n = 2^k$ für $k \geq 1$. Der Rekursionsschritt muss dann k-mal wiederholt werden, was für den Aufwand folgendes ergibt:

$$T_{MULT}^{STRASSEN}(n) = 7^k, \;\; T_{ADD}^{STRASSEN}(n) = \sum_{i=1}^{k} 15 \cdot 7^{i-1} \cdot \left(\tfrac{n}{2^i}\right)^2.$$

Wer das nicht glaubt, kann es aus den obigen Formeln durch vollständige Induktion über k herleiten.

Die erzielten Ergebnisse lassen sich nun noch so umrechnen, dass sie mit den bisherigen Aufwänden für das Matrizenprodukt besser vergleichbar werden. Dabei werden überwiegend Gesetze der Potenzrechnung verwendet und die Definition des Logarithmus zur Basis 2, der mit ld bezeichnet ist (*logarithmus dualis*):

$$T_{MULT}^{STRASSEN}(n) = 7^k = (2^{\operatorname{ld} 7})^k = (2^k)^{\operatorname{ld} 7} = n^{\operatorname{ld} 7},$$
$$T_{ADD}^{STRASSEN}(n) = \sum_{i=1}^{k} 15 \cdot 7^{i-1} \cdot \left(\tfrac{n}{2^i}\right)^2 = \sum_{i=1}^{k} 15 \cdot 7^{i-1} \cdot \tfrac{n^2}{4^i}$$
$$\underset{(1)}{=} 15 \cdot \tfrac{n^2}{4} \cdot \sum_{i=0}^{k-1} \left(\tfrac{7}{4}\right)^i \underset{(2)}{=} 15 \cdot \tfrac{n^2}{4} \cdot \tfrac{\left(\frac{7}{4}\right)^k - 1}{\frac{7}{4} - 1} = 5n^2 (\tfrac{7}{4})^k - 5n^2 \underset{(3)}{=} 5n^{\operatorname{ld} 7} - 5n^2,$$

wobei sich (1) durch Ausklammern und eine Indexverschiebung, (2) aus der Summenformel für geometrische Reihen und (3) aus Potenzgesetzen ergibt. Letzteres sei noch explizit ausgeführt, weil es etwas Übung erfordert, darauf zu kommen:

$$n^2 \cdot \left(\tfrac{7}{4}\right)^k = (2^k)^2 \cdot \left(\tfrac{7}{4}\right)^k = 4^k \cdot \left(\tfrac{7}{4}\right)^k = 7^k = n^{\operatorname{ld} 7}.$$

Es bleibt anzumerken, dass $\operatorname{ld} 7$ ungefähr $2,81$ ist, so dass der Strassen-Algorithmus sowohl dem klassischen als auch dem Winograd-Algorithmus für große n in der Zahl der Multiplikationen (und analog der Additionen) überlegen ist; vgl. Tabelle 2.

n	klassisch: n^3	Winograd: $\frac{1}{2}n^3 + n^2$	Strassen: $n^{2,81}$
8	512	320	345
64	262 144	135 168	118 950
256	16 777 216	8 454 144	5 849 979
⋮	⋮	⋮	⋮

Tabelle 2: Aufwandsvergleich der Matrizenmultiplikations-Algorithmen

Vorbemerkung zu den folgenden Kapiteln zum Thema Formale Sprachen

Laut Curriculum sollen in Theoretischer Informatik 2 neben der Einführung in die Komplexitätstheorie die Themen Formale Sprachen und Berechenbarkeit aus Theoretischer Informatik 1 weitergeführt werden. Konkret geschieht das anhand des Wortproblems für monotone Grammatiken, weil das ein sogenanntes PSPACE-Problem ist und damit die Diskussion von Komplexitätsklassen noch etwas abgerundet werden kann, und eines Nichtentscheidbarkeitsproblems für kontextfreie Grammatiken als Überleitung zum Thema Berechenbarkeit. Um diese Betrachtungen systematisch einordnen zu können, werden Formale Sprachen von der Definition von Programmiersprachen und der Konstruktion ihrer Compiler her noch einmal ausführlich motiviert und wird die Chomsky-Hierarchie eingeführt. Dabei werden einige Passagen aus dem Skript von Theoretischer Informatik 1 wiederholt.

6 Formale Sprachen – Wieso? Weshalb? Warum?

Die Theorie formaler Sprachen stellt eines der ältesten und am weitesten entwickelten Gebiete der Theoretischen Informatik dar. Die Bedeutung dieser Theorie rührt daher, dass ihre Konzepte und Methoden die Grundlage für die Definition der Syntax von Programmiersprachen und für den Bau ihrer Compiler bilden, wobei insbesondere die Syntaxanalyse unterstützt wird (6.1). Ein zentraler Aspekt der Syntaxanalyse ist die Lösung des sogenannten Wortproblems, die sehr viel mit Berechenbarkeit zu tun hat, wie sie in der Lehrveranstaltung *Theoretische Informatik 1* einführend behandelt wird.

6.1 Syntax von Programmiersprachen und Syntaxanalyse

Wer ein Programm in einer Programmiersprache X schreiben möchte, wählt sich häufig einen Texteditor Y aus und erstellt mit dessen Hilfe eine bestimmte Zeichenkette, die dann als Eingabe für den Compiler von X dient. Dies ist in Abbildung 10 skizziert.

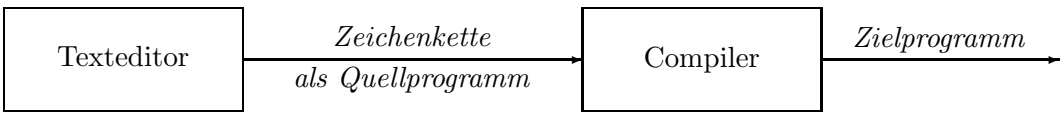

Abbildung 10: Erstellung eines Programms

Man kann nicht erwarten, dass jeder Text, der mit dem Editor Y entstehen kann, bereits ein Programm in X ist. Denn mit einem Texteditor lassen sich in der Regel beliebige Zeichenketten aufbauen, während ein Programm eine bestimmte Form haben muss.

Zeichenketten jedoch, die keine Programme sind, wird der Compiler als unübersetzbar zurückweisen. Woher weiß aber eine Programmiererin oder ein Programmierer, wie die Zeichenkette aussehen muss, um ein Programm zu sein? Wie findet der Compiler heraus, ob irgendeine Zeichenkette ein übersetzbares Programm darstellt?

Die Form von Programmen einer Programmiersprache wird durch ihre Syntax festgelegt. Die Syntaxdefinition macht meist äußerst restriktive Vorschriften über die Anordnung und Plazierung von Zeichen, damit eine Zeichenkette ein syntaktisch richtiges Programm ist. Eine Person, die ein Programm mit Hilfe eines Texteditors erstellen will, sollte die Syntax recht gut kennen, weil andernfalls sicherlich häufig Syntaxfehler auftreten. Der Compiler dagegen übersetzt die eingegebene Zeichenkette in ein Zielprogamm, falls die Eingabe ein syntaktisch korrekt gebildetes Programm ist. Um das festzustellen, besitzen Compiler eine Syntaxanalyse-Komponente, die diese Aufgabe übernimmt.

Bei der Syntaxanalyse wird für eingegebene Zeichenketten untersucht, ob sie Programme sind oder nicht. Im positiven Fall wird außerdem der syntaktische Aufbau ermittelt, weil diese Information bei der weiteren Übersetzung maßgeblich genutzt wird. Im negativen Fall werden meist noch Hinweise auf Syntaxfehler gegeben. Die Trennung in "richtige" und "falsche" Zeichenketten ist in der Regel das entscheidende algorithmische Problem, weil bei der Lösung die zusätzlichen Informationen ohne allzu große Mühe nebenbei gewonnen werden können.

Die syntaktisch richtigen Programme einer Programmiersprache bilden als Menge von Zeichenketten eine "formale Sprache". Solche Zeichenkettenmengen sind Gegenstand der Untersuchung in der Theorie formaler Sprachen, die Konzepte für die syntaktische Definition formaler Sprachen bereitstellt und Methoden liefert, um die Eigenschaften formaler Sprachen analysieren zu können. Zu den wichtigsten Anwendungsfeldern der Theorie formaler Sprachen gehören die Syntaxdefinition von Programmiersprachen und die Syntaxanalyse. Die Schlüsselfrage der Syntaxanalyse, ob eine Zeichenkette ein Programm ist oder nicht, wird auch *Wortproblem* genannt. Betrachtet man die Menge aller richtigen Programme als formale Sprache, dann besteht das Problem darin, ob eine Zeichenkette in der Sprache liegt oder nicht. Da die Lösung des Wortproblems eine zentrale Rolle bei der Implementierung von Programmiersprachen spielt, aber keineswegs immer auf der Hand liegt, zieht sich die Behandlung des Wortproblems wie ein roter Faden durch die Theorie formaler Sprachen.

Aber erst einmal zurück zur Syntaxdefinition. Eine grundlegende Weise, formale Sprachen, einschließlich Programmiersprachen, zu spezifizieren, besteht darin, die übliche Art der Begriffsbildung (unter *dem und dem* versteht man *das und das*) in formalisierter Form zu nutzen. Einem zu definierenden, also noch undefinierten, syntaktischen Konstrukt wird ein definierender Ausdruck zugeordnet. Beides zusammen bildet eine Syntaxregel, das (noch) Undefinierte wird linke, das Definierende rechte Regelseite genannt. Der definierende Ausdruck der rechten Seite ist eine Zeichenkette, die rekursiv auch wieder undefinierte Konstrukte enthalten darf. Ist das noch Undefinierte der linken Seite durch ein einziges Zeichen dargestellt, spricht man von einer kontextfreien Regel. Die Syntaxdefinition einer Programmiersprache besteht in einem ersten Anlauf meist in der Angabe von

kontextfreien Regeln, die dann später um Syntaxteile ergänzt werden, die sich nicht durch kontextfreie Regeln ausdrücken lassen.

Der kontextfreie Anteil der Syntax von Programmiersprachen wird häufig in der sogenannten Backus-Naur-Form geschrieben, wobei die kontextfreien Regeln als linke Seiten nichtterminale Zeichen, die zu definierende syntaktische Konstrukte der Sprache benennen, und als rechte Seiten Zeichenketten aus terminalen und nichtterminalen Zeichen besitzen. Die rechten Seiten zur selben linken Seite werden als Alternativen nebeneinandergestellt, durch einen senkrechten Strich voneinander getrennt, was die sogenannte Backus-Naur-Form (BNF) ergibt. Linke und rechte Seiten werden durch das Trennzeichen "::=" auseinandergehalten. Nichtterminale Zeichen sind in spitze Klammern eingeschlossen. Für Spezifikationen in CE-S sieht das im Ausschnitt so aus:

<*cesspec*> ::= <*name*> opns: <*decllist*> vars: <*varlist*> eqns: <*celist*>
<*decllist*> ::= <*decl*> | <*decl*>, <*decllist*>
<*celist*> ::= <*ce*> | <*ce*>, <*celist*>
<*decl*> ::= <*name*>: <*typelist*>→<*type*>
<*ce*> ::= <*eq*> falls <*eqlist*>

Die Regeln können zum Aufbau syntaktisch korrekter Programme bzw. Programmstücke verwendet werden, indem mit dem gewünschten Konstrukt begonnen wird und dann nach und nach in den aktuellen Zeichenketten auftretende nichtterminale Zeichen durch zugehörige rechte Seiten ersetzt werden; z.B.:

<*cesspec*>
⟶ <*name*>vars:<*varlist*> opns: <*decllist*> eqns: <*celist*>
⟶ <*name*> opns: <*decl*>vars:<*varlist*> eqns: <*celist*>
⟶ <*name*> opns: <*decl*>vars:<*varlist*> eqns: <*ce*>, <*celist*>
⟶ <*name*> opns: <*decl*>vars:<*varlist*> eqns: <*ce*>, <*ce*>
⟶ <*name*> opns: <*name*>: <*typelist*>→<*type*>vars:<*varlist*> eqns: <*ce*>, <*ce*>
⋮
⟶ **sort**
opns: $sort\colon A^* \to A^*$
vars: $x \in A$, $u \in A^*$
eqns: $sort(\lambda) = \lambda$,
$sort(xu) = insort(x, sort(u))$

wobei allerdings die resultierende Spezifikation mit den obigen Regeln nicht erreichbar ist. Dazu müsste die Syntax von CE-S vervollständigt werden. Ziel solchen Ableitens ist eine Zeichenkette ohne nichtterminale Zeichen, die dann bezüglich der kontextfreien Syntax ein korrektes Konstrukt darstellt. Dieses Prinzip findet sich in syntaxgesteuerten Editoren wieder.

6.2 Wortproblem und Berechenbarkeit

Wie in Abschnitt 6.1 erläutert, ist die Syntaxanalyse eine zentrale Aufgabe bei der Übersetzung von Programmen einer Programmiersprache. Den Kern bildet dabei die Lösung des Wortproblems für die Menge aller syntaktisch korrekten Programme, das darin besteht, algorithmisch festzustellen, ob ein beliebiges eingegebenes Wort ein Programm ist oder nicht.

Aber das Wortproblem ist auch für andere Sprachen signifikant, weil ein enger Zusammenhang zur Berechenbarkeit besteht. Das soll im folgenden näher erläutert werden.

6.2.1 Wortproblem einer beliebigen Sprache

Betrachtet man als *formale Sprache* eine beliebige Menge L von Wörtern aus A^* für ein Alphabet A, so definiert L ein *Wortproblem*:

Gibt es einen Algorithmus, der für jedes $x \in A^*$ bestimmt, ob $x \in L$ oder $x \notin L$ gilt?

Die Frage ist also, ob die sogenannte *charakteristische Funktion* $\chi_L \colon A^* \to \{\mathsf{T}, \mathsf{F}\}$ von $L \subseteq A^*$, die für alle $x \in A^*$ gegeben ist durch $\chi_L(x) = \mathsf{T}$, falls $x \in L$, und $\chi_L(x) = \mathsf{F}$ sonst, berechenbar ist. Die Lösbarkeit des Wortproblems von $L \subseteq A^*$ entspricht der Berechenbarkeit der zugehörigen charakteristischen Funktion.

Als Beispiel betrachte die Menge aller Palindrome aus A^*. Das Wortproblem ist in diesem Fall gerade die Frage, ob ein gegebenes Wort ein Palindrom ist oder nicht. Dass dieser Test berechenbar ist und damit das Wortproblem lösbar, wurde in einer Aufgabe zu CE-S gezeigt.

6.2.2 Entscheidungsprobleme als Wortprobleme

Aber man kann sich der Situation auch umgekehrt nähern, indem man von Entscheidungsproblemen ausgeht. Ein *Entscheidungsproblem* für einen Datenbereich D (wie beispielsweise A^*, $\mathbb{N}$, $\mathbb{Z}$ oder deren kartesische Produkte) ist durch eine totale Funktion $f \colon D \to \{\mathsf{T}, \mathsf{F}\}$ beschrieben. Es ist *lösbar*, wenn f berechenbar ist. Die Funktion f kann als charakteristische Funktion der Menge aller Eingaben, die T liefern, aufgefasst werden. Die Berechnung von $f(x)$ für $x \in D$ entspricht also gerade der Frage, ob $x \in f^{-1}(\mathsf{T})$ oder $x \notin f^{-1}(\mathsf{T})$ gilt. Falls $D = A^*$, ist das das Wortproblem von $f^{-1}(\mathsf{T})$.

Aber auch wenn die Daten in D durch Wörter repräsentiert sind, d.h. $D \subseteq A^*$ für ein geeignetes Alphabet A, kann f dadurch berechnet werden, dass man das Wortproblem für $f^{-1}(\mathsf{T})$ löst. Denn es gilt für alle $x \in A^*$:

(i) $f(x) = \mathsf{T}$, falls $x \in f^{-1}(\mathsf{T})$, und
(ii) $f(x) = \mathsf{F}$, falls $x \in D$, aber $x \notin f^{-1}(\mathsf{T})$.

Offensichtlich muss im Falle des negativen Ergebnisses auch das Wortproblem von D lösbar sein.

Ein Beispiel dieser Art ist ein Primzahltest $prim\colon \mathbb{N} \to \{\mathsf{T}, \mathsf{F}\}$ mit $prim(n) = \mathsf{T}$, falls n eine Primzahl ist, und $prim(n) = \mathsf{F}$ sonst. Für die natürlichen Zahlen in Dezimaldarstellung gilt $\mathbb{N} \subseteq \{0, \ldots, 9\}^*$. Ob eine Ziffernfolge eine Zahldarstellung ist, lässt sich leicht ermitteln, denn jede Zahl größer 0 hat keine führenden Nullen. Deshalb liefert die Lösung des Wortproblems für die Menge aller Primzahlen einen Primzahltest und umgekehrt.

6.2.3 Berechenbare Funktionen und das Wortproblem

Ähnliches gilt auch für beliebige berechenbare Funktionen $g\colon X \to Y$. Wenn $X \subseteq A^*$ gilt, dann ist für jedes $y \in Y$ die Menge der Urbilder $g^{-1}(y) \subseteq X \subseteq A^*$ eine formale Sprache. Und die Frage, ob g für eine Eingabe $x \in X$ den Wert y liefert, wird gerade durch die Lösung des Wortproblems von $g^{-1}(y)$ beantwortet.

Ein Beispiel ist die modulo-Funktion $\bmod k\colon \mathbb{N} \to \mathbb{N}$ für $k > 0$, die für jedes $n \in \mathbb{N}$ $n \bmod k$ als Wert liefert. Statt für n den Rest beim Teilen durch k zu berechnen, kann man auch nachsehen, in welcher Restklasse $[i] = \{m \in \mathbb{N} \mid m \bmod k = i\}$ für $i = 0, \ldots, k-1$ die Zahl n liegt.

6.2.4 Wortproblem selten lösbar

Wenn das unterliegende Alphabet A nicht leer ist, gibt es unendlich viele Wörter, so dass nach einer bekannten Überlegung aus der Mathematik die Menge aller Teilmengen von A^* überabzählbar unendlich ist. Da es aber nur abzählbar viele Algorithmen gibt, müssen die Wortprobleme der meisten Sprachen unlösbar sein (vgl. Abschnitt 18.3 des Skripts *Theoretische Informatik 1* [Kre04] mit der analogen Argumentation zu berechenbaren Funktionen). Aber obwohl die Lösbarkeit des Wortproblems grundsätzlich in den seltensten Fällen gegeben ist, sind doch die meisten Sprachen, denen man üblicherweise begegnet, ziemlich gutartig. So haben alle Sprachen, die in diesem Abschnitt vorkommen (die Menge der Primzahlen, die Menge der Palindrome, die Menge der durch k mit Rest i teilbaren Zahlen), ein lösbares Wortproblem. Es ist sogar relativ schwierig, Sprachen zu konstruieren, deren Wortproblem nicht lösbar ist (siehe zwei derartige Beispiele in den Abschnitten 9.1 und 9.2 in [HMU01, HMU02] und oder auch Abschnitt 8.4 in diesem Skript). Dem Wortproblem wird dennoch in den nächsten Abschnitten viel Aufmerksamkeit gewidmet, weil man nicht nur an Lösbarkeit interessiert ist, sondern auch an praktisch benutzbaren Lösungen. So muss ein Algorithmus, der in einen Compiler eingebaut werden soll, insbesondere auch schnell sein.

7 Chomsky-Grammatiken

Lässt man bei den kontextfreien Regeln, wie sie meist bei der syntaktischen Beschreibung von Programmiersprachen verwendet werden, auf der linken Seite auch beliebige Zeichenketten zu, erhält man Produktionen (Regeln) von Chomsky-Grammatiken (nach dem amerikanischen Sprachwissenschaftler Noam Chomsky, geb. 1928, einem Pionier der Theorie der formalen Sprachen).

7.1 Grammatik allgemein

Eine Chomsky-Grammatik besteht aus endlich vielen Produktionen der Form $u ::= v$ für $u, v \in A^*$, wobei A ein Alphabet ist, aus einem Startsymbol $S \in A$ und einem terminalen Alphabet $T \subseteq A$. Meist wird noch verlangt, dass $S \in A \setminus T$ ist und in den linken Seiten von Produktionen Zeichen vorkommen, die nicht terminal sind. Die Differenzmenge $A \setminus T$ wird nichtterminales Alphabet genannt und mit N bezeichnet. Es ist manchmal auch bequemer, mit einem Startwort statt mit einem Startsymbol zu beginnen.

1. Für ein gegebenes Alphabet A ist eine *Produktion* (*Regel*) ein Paar $p = (u, v) \in A^* \times A^*$, das meist als $u ::= v$ geschrieben wird. Die Zeichenkette u wird *linke Seite*, v *rechte Seite* von p genannt.
   Zur Abkürzung können mehrere Produktionen $u ::= v_1, \ldots, u ::= v_k$ $(k \geq 2)$ mit derselben linken Seite zu $u ::= v_1 \mid \cdots \mid v_k$ zusammengefasst werden.
2. Eine *Chomsky-Grammatik* ist ein System $G = (N, T, P, S)$, wobei N eine Menge *nichtterminaler Zeichen*, T eine Menge *terminaler Zeichen*, P eine endliche Menge von Produktionen und $S \in N$ ein *Startsymbol* ist.
   Soweit nichts anderes gesagt wird, nimmt man an, dass kein nichtterminales Zeichen gleichzeitig terminal ist, d.h. $N \cap T = \emptyset$, dass alle in Produktionen vorkommenden Zeichen terminal oder nichtterminal sind, d.h. $p \in (N \cup T)^* \times (N \cup T)^*$ für alle $p \in P$, und dass in jeder linken Seite mindestens ein nichtterminales Zeichen vorkommt, d.h. $u \in (N \cup T)^* N (N \cup T)^*$ (bzw. $u \notin T^*$) für jede Produktion $u ::= v \in P$.

Produktionen werden auf Zeichenketten analog zum kontextfreien Fall angewendet. Man sucht in einer Zeichenkette eine Teilkette, die die linke Seite einer Produktion ist, und ersetzt sie durch die rechte Seite.

3. Seien $w, w', x, y, u, v \in A^*$. Dann wird w' aus w *direkt* durch Anwendung der Produktion $p = (u ::= v)$ *abgeleitet*, falls $w = xuy$ und $w' = xvy$. In diesem Falle wird $w \underset{p}{\longrightarrow} w'$ geschrieben.
   Die Anwendung einer Produktion wird *direkte Ableitung* genannt. Ist P eine Menge von Produktionen und $p \in P$, so kann man statt $w \underset{p}{\longrightarrow} w'$ auch $w \underset{P}{\longrightarrow} w'$ schreiben.
4. Die Iteration direkter Ableitungen ergibt das Konzept der *Ableitung*:

$$w_0 \underset{p_1}{\longrightarrow} w_1 \underset{p_2}{\longrightarrow} \cdots \underset{p_n}{\longrightarrow} w_n$$

für $w_0, \ldots, w_n \in A^*$ und Produktionen $p_1, \ldots, p_n$ ($n \geq 1$). Stammen alle angewendeten Produktionen aus P, so kann man die obige Ableitung auch schreiben als $w_0 \underset{P}{\longrightarrow} \cdots \underset{P}{\longrightarrow} w_n$ oder $w_0 \overset{n}{\underset{P}{\longrightarrow}} w_n$. Für manche Zwecke ist es sinnvoll, auch *Nullableitungen* zuzulassen: $w \overset{0}{\underset{P}{\longrightarrow}} w$ für alle $w \in A^*$. Statt $w \overset{n}{\underset{P}{\longrightarrow}} w'$ für $n \in \mathbb{N}$ darf auch $w \overset{*}{\underset{P}{\longrightarrow}} w'$ geschrieben werden. Außerdem kann man bei Ableitungen und direkten Ableitungen das Subskript P weglassen, wenn die Produktionsmenge aus dem Kontext klar ist.

Der Ableitungsprozess bildet die operationelle Semantik, die durch eine Produktionsmenge syntaktisch beschrieben ist. Betrachtet man diejenigen Zeichenketten, die aus dem Startsymbol einer Chomsky-Grammatik $G = (N, T, P, S)$ ableitbar sind und nur aus terminalen Zeichen bestehen, so erhält man auf der Basis des Ableitungsprozesses eine erzeugte Sprache.

5. Sei $G = (N, T, P, S)$ eine Chomsky-Grammatik. Dann enthält die von G *erzeugte Sprache* alle mit Produktionen in P aus dem Startsymbol S ableitbaren terminalen Zeichenketten:
$$L(G) = \{w \in T^* \mid S \overset{*}{\underset{P}{\longrightarrow}} w\}.$$

Auf diese Weise stellen Chomsky-Grammatiken ein syntaktisches Instrument dar, um formale Sprachen zu spezifizieren. Ausführliche Darstellungen von Chomsky-Grammatiken findet man in praktisch jedem Buch über formale Sprachen (siehe z.B. Hopcroft, Motwani, Ullman [HMU01] mit der deutschen Übersetzung [HMU02], Moll, Arbib und Kfoury [MAK88] und Salomaa [Sal73] mit der deutschen Übersetzung [Sal78]). Die Verbindung von formalen Sprachen und der syntaktischen Behandlung von Programmiersprachen wird umfassend in Aho und Ullman [ASU86] abgehandelt.

7.2 Beispiele

1. Mit der Produktion $S ::= aS$ lässt sich das Zeichen a hochzählen:
$$S \to aS \to a^2S \to \cdots \to a^nS.$$
Entsprechend kann man mit $S ::= a^kS$ ($k \in \mathbb{N}$) ein Vielfaches von k hochzählen. Terminieren lässt sich dieser Vorgang mit $S ::= \lambda$, so dass
$$L((\{S\}, \{a\}, \{S ::= a^kS \mid \lambda\}, S)) = \{a^{n \cdot k} \mid n \in \mathbb{N}\}.$$

2. Fast genauso einfach ist es, zwei Größen gleichzeitig hochzuzählen:
$$L((\{S\}, \{a, b\}, \{S ::= aSb \mid \lambda\}, S)) = \{a^nb^n \mid n \in \mathbb{N}\}.$$

3. Auch das getrennte Zählen zweier (bzw. mehrerer) Größen ist kein Problem. Sei $T_l = \{a_1, \ldots, a_l\}$ ($l \geq 1$), $N_l = \{S\} \cup \{A_1, \ldots, A_l\}$ und $P_l = \{S ::= A_1 \cdots A_l\} \cup \{A_i ::= a_iA_i \mid i = 1, \ldots, l\} \cup \{A_i ::= \lambda \mid i = 1, \ldots, l\}$.

Dann gilt:

$$L((N_l, T_l, P_l, S)) = \{a_1^{n_1} \cdots a_l^{n_l} \mid n_i \geq 0,\ i = 1, \ldots, l\}.$$

4. Etwas schwieriger wird es, die zahlenmäßige Ausgewogenheit zweier Größen zu garantieren, wenn die Reihenfolge nicht mehr wie in Punkt 2 fixiert ist.
   Die Grammatik $G_{equilibrium} = (\{A, B, S\}, \{a, b\}, P_{equilibrium}, S)$, wobei die Regelmenge die Regeln

   $$S ::= ASB,\ S ::= \lambda,\ AB ::= BA,\ A ::= a,\ B ::= b$$

   enthält, erzeugt als Sprache die Menge aller Wörter, in denen nur die Zeichen a und b vorkommen und das gleich oft. Ableitungen (d.h. wiederholte Regelanwendungen) sehen z.B. so aus:

   $$S \underset{(1)}{\longrightarrow} ASB \underset{(1)}{\longrightarrow} A^2SB^2 \underset{(2)}{\longrightarrow} A^2B^2 \underset{(3)}{\longrightarrow} ABAB \underset{(3)}{\longrightarrow} ABBA \underset{(4),(5)}{\longrightarrow} \cdots \underset{(4),(5)}{\longrightarrow} abba.$$

   Dabei verweisen die Nummern auf die Produktionen, die von links nach rechts gezählt sind.
5. Noch schwieriger wird es, wenn man drei Größen gleichzeitig nach Art des Punktes 2 kontrollieren möchte. Es ist zwar leicht zu sehen, dass mit den Produktionen der Form $ab^nc ::= a^2b^{n+1}c^2$ für $n \geq 1$ aus der Zeichenkette abc alle Zeichenketten der Form $a^nb^nc^n$ abgeleitet werden können; aber das sind unendlich viele Regeln. Um dasselbe mit endlich vielen Regeln zu erreichen, bedarf es der bisher kompliziertesten Grammatik im nächsten Punkt. Oder geht es einfacher?
6. Folgende Produktionen erlauben, die Sprache $\{a^nb^nc^n \mid n \geq 1\}$ zu erzeugen, wenn man mit S startet und $\{a, b, c\}$ als terminales Alphabet wählt:

(1)	$S ::= aABc$
(2)&(3)	$A ::= aABc \mid \lambda$
(4)	$cB ::= Bc$
(5)	$aB ::= ab$
(6)	$bB ::= bb$

   Mit (1) bis (3) erhält man $S \xrightarrow{*} a^n(Bc)^n$.

   Mit (4) wird daraus: $a^nB^nc^n$.

   Mit (5) und (6) erhält man: $a^nb^nc^n$.

   Der Nachweis, dass man nichts anderes Terminales ableiten kann, erfordert diverse Fallunterscheidungen, die hier nicht im einzelnen durchgeführt werden sollen.

8 Immerhin aufzählbar

In diesem Kapitel wird der Zusammenhang zwischen Chomsky-Grammatiken und dem Konzept der Aufzählbarkeit (vgl. [Kre04, Abschnitt 18.3]) beschrieben. Der Ableitungsprozess zusammen mit einem Terminalitätstest für Zeichenketten zählt die erzeugte Sprache einer Chomsky-Grammatik auf (8.1), deren Wortproblem sich damit auch als “halb”

lösbar erweist (8.2). Es wird außerdem plausibel gemacht, dass effektiv aufzählbare Mengen von Zeichenketten von Chomsky-Grammatiken erzeugt werden (8.3), allerdings ist das Wortproblem nicht immer "ganz" lösbar (8.4). In Abschnitt 8.5 schließlich wird auf Zusammenhänge zwischen dem hier angegebenen Aufzählungsverfahren und anderen bekannten Algorithmen hingewiesen.

8.1 Aufzählbarkeit erzeugter Sprachen

Für eine beliebige Chomsky-Grammatik $G = (N, T, P, S)$ bildet der Ableitungsmechanismus einen algorithmischen Prozess, den man mit beliebigen Zeichenketten beginnen und beliebig lange laufen lassen kann. Startet man zum Beispiel mit S und erlaubt bis zu k Ableitungsschritte, führt aber alle möglichen Alternativen bei Regelanwendungen aus, so erhält man die Menge $S(G)_k$ aller aus S in bis zu k Schritten ableitbaren Wörter; d.h.

$$S(G)_k = \{w \mid S \xrightarrow[P]{l} w,\ l \leq k\}.$$

Es gilt offenbar $S(G)_k \subseteq S(G)_m$ für $k \leq m$. Es gilt genauer: $S(G)_0 = \{S\}$ und $S(G)_{k+1} = S(G)_k \cup \{w \mid v \xrightarrow[P]{} w,\ v \in S(G)_k\}$. Denn nach Definition von Ableitungen kann man aus S in 0 Schritten nur S ableiten, und eine Ableitung mit höchstens $k+1$ Schritten hat sogar höchstens k Schritte oder setzt sich aus k Schritten gefolgt von einer weiteren direkten Ableitung zusammen.

Insbesondere erweisen sich die Mengen $S(G)_k$ für $k \in \mathbb{N}$ als endlich. Denn $S(G)_0$ ist einelementig, und wenn nach Induktionsvoraussetzung $S(G)_k$ endlich ist, so muss es auch $S(G)_{k+1}$ sein. Letzteres ergibt sich daraus, dass die endlich vielen Wörter in $S(G)_k$ nur endlich viele Teilwörter besitzen, also auch höchstens endlich viele linke Regelseiten, die durch höchstens endlich viele rechte Regelseiten ersetzt werden können, da die Regelmenge endlich ist.

Darüber hinaus ist $S(G)_{k+1}$ aus $S(G)_k$ effektiv, d.h. algorithmisch herstellbar, da Suchen und Ersetzen von Teilwörtern algorithmisch machbar sind.

Bezeichnet man die Menge aller aus S ableitbaren Zeichenketten mit $S(G)$, so gilt offenbar:

$$S(G) = \bigcup_{k \in \mathbb{N}} S(G)_k,$$

denn jedes ableitbare Wort ist in einer bestimmten Schrittzahl ableitbar. Die Elemente von $S(G)$ nennt man *Satzformen* von G. Besteht eine Satzform nur aus terminalen Zeichen, so gehört sie zur erzeugten Sprache, d.h.

$$L(G) = S(G) \cap T^*.$$

Bildet man also die Mengen $S(G)_k$ für wachsende k, beginnend mit $S(G)_0$, und filtert die terminalen Wörter heraus, so erhält man einen algorithmischen Vorgang, bei dem nach und nach jedes Wort aus $L(G)$ entsteht. Dieses Verfahren ist in Abbildung 11 dargestellt.

Mit anderen Worten erweist sich die von der Chomsky-Grammatik G erzeugte Sprache als effektiv oder – wie man auch sagt – *rekursiv aufzählbar*.

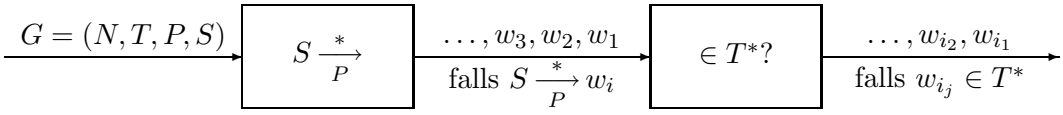

Abbildung 11: Aufzählung der von einer Grammatik erzeugten Sprache

8.2 Die halbe Miete

Will man von einer Zeichenkette wissen, ob sie zu einer erzeugten Sprache gehört oder nicht, so kann man den Aufzählungsprozess starten. Erscheint dabei irgendwann die fragliche Zeichenkette, liegt sie in der gegebenen Sprache. Der positive Fall des Wortproblems ist damit gelöst. Über den negativen Fall erfährt man auf diese Weise jedoch nichts, wenn der Aufzählungsprozess nicht anhält, was gerade bei unendlichen Sprachen passiert. Denn ist eine Zeichenkette zu einem bestimmten Zeitpunkt nicht aufgezählt, kann das noch später oder gar nicht geschehen. Das Wortproblem ist so nur "halb" gelöst, was man auch *semi-entscheidbar* nennt.

8.3 Erzeugbarkeit aufzählbarer Sprachen

Dass auch die Umkehrung gilt, dass also jede rekursiv aufzählbare Menge von Zeichenketten von einer Chomsky-Grammatik erzeugt werden kann, soll nicht bewiesen werden, da das zu viel Zeit und Mühe erforderte. Man kann sich aber die Richtigkeit dieser Behauptung mit den bisherigen Erkenntnissen der theoretischen Informatik recht gut plausibel machen.

Eine Menge von Zeichenketten ist rekursiv aufzählbar, falls es einen Algorithmus gibt, der die Elemente der Menge nach und nach "aufzählt". Nach der Churchschen These ist alles Algorithmische auch mit einer Turingmaschine machbar. Die Arbeitsweise einer Turingmaschine jedoch kann von einer Chomsky-Grammatik simuliert werden. Denn Konfigurationen sind bereits Zeichenketten, die den aktuellen Bandinhalt, den aktuellen Zustand und die Position des Lese/Schreibkopfes repräsentieren. Die Zustandsüberführung lässt sich auch durch Produktionen ausdrücken. Es ist dann nicht mehr allzu schwierig, die Grammatik erzeugen zu lassen, was die Turingmaschine aufzählt. Details zu der Verbindung zwischen Turingmaschinen und Chomsky-Grammatiken finden sich in vielen Büchern über formale Sprachen.

8.4 Unlösbarkeit des Wortproblems

In ihrer allgemeinen Form sind Chomsky-Grammatiken allerdings nur bedingt für die Definition der Syntax von Programmiersprachen geeignet, weil nicht zu jeder definierbaren Syntax auch eine Syntaxanalyse möglich ist. Auch das soll nicht formal bewiesen, sondern nur mit Hilfe von PASCALchen und seinem unentscheidbaren Halteproblem (vgl. [Kre04, Abschnitt 18.4]) plausibel gemacht werden.

Betrachte etwa folgenden Algorithmus: Generiere nach und nach alle PASCALchen-Programme und alle ihre initialen Berechnungszustände; interpretiere jedes dieser Programme für jeden dieser Berechnungszustände; terminiert die Berechnung, gilt das Paar aus Programm und initialem Berechnungszustand als aufgezählt. Die so entstehende Menge ist also rekursiv aufzählbar und lässt sich nach der vorangegangenen Überlegung von einer Chomsky-Grammatik erzeugen. Das zugehörige Wortproblem kann aber nicht lösbar sein, weil es sonst das Halteproblem löste, was bekanntlich unmöglich ist.

8.5 Ausschöpfende Suche in die Breite mit roher Gewalt

Das der Aufzählung der erzeugten Sprache in Punkt 1 zugrundeliegende Verfahren, das in Abbildung 12 skizziert ist, lässt sich bei vielen Arten regelbasierter Systeme anwenden: Man beginnt mit einem Anfangszustand oder einem Eingabezustand, wendet wiederholt auf alle entstehenden Zwischenzustände alle möglichen Regeln an, wie und wo immer es geht, und filtert dann nach einem bestimmten Kriterium Ausgabe- oder Endzustände aus den erreichten und produzierten Zwischenzuständen heraus. Die Vorwärtsinterpretation von CE-S-Spezifikationen, die Berechnung von PASCALchen-Programmen und die Turingmaschinen funktionieren nach diesem Prinzip. Man mache sich in diesen drei Fällen klar, was die Systemzustände, was die Regeln sind und nach welchem Kriterium ausgefiltert wird.

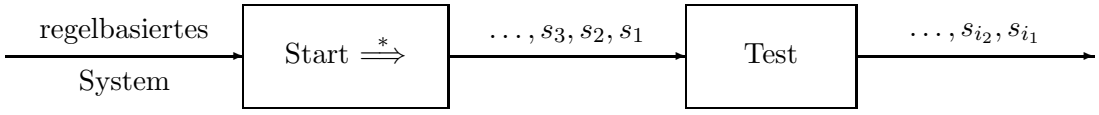

Abbildung 12: Allgemeine Funktionsweise regelbasierter Systeme

Im Zusammenhang mit regelbasierten Systemen der Künstlichen Intelligenz und mit Expertensystemen sowie in der Logistik wird das Verfahren oft *ausschöpfende Suche* genannt. In der Algorithmen- und Komplexitätstheorie ist das Verfahren ebenfalls bekannt und wird dort häufig als "rohe Gewalt" (brute force) eingesetzt, weil einfach alles durchprobiert wird. Zählt man alle Möglichkeiten nach der Systematik in Abschnitt 8.1 auf, d.h. nach wachsender Ableitungslänge bzw. wachsender Zahl von Regelanwendungen, so spricht man häufig auch von *Breitensuche* (*breadth first search*).

9 Lösbarkeit des Wortproblems für monotone Grammatiken

Nach den Überlegungen des vorigen Kapitels kann die Lösbarkeit des Wortproblems nicht für alle Chomsky-Grammatiken garantiert werden, sondern höchstens für geeignete

Spezialfälle. Außerdem hat sich gezeigt, dass die halbe Lösung, die durch den Aufzählungsprozess gegeben ist, deshalb nicht zu einer ganzen gemacht werden kann, weil man nicht weiß, wann der Prozess als erfolglos abgebrochen werden darf.

Die Situation ändert sich, wenn man monotone Grammatiken betrachtet, in deren Produktionen keine rechte Seite kürzer als die linke Seite ist. Dann können Wörter während des Ableitens auch nicht kürzer werden. Will man in einem solchen Fall von einer Zeichenkette wissen, ob sie zur erzeugten Sprache gehört oder nicht, kann man beim Aufzählungsprozess auf längere Zeichenketten verzichten. Die Zahl der Zeichenketten bis zu einer bestimmten Länge ist aber endlich, so dass jede Aufzählung solcher Zeichenketten nach endlich vielen Schritten abbrechen muss. Das ist der Schlüssel zur Lösung des Wortproblems für monotone Grammatiken. Allerdings ist der angegebene Algorithmus exponentiell, und ein polynomieller ist nicht bekannt, so dass auch monotone Grammatiken für praktische Anwendungen nicht geeignet sind, wenn die Lösbarkeit des Wortproblems wichtig ist.

9.1 Monotone Grammatiken

Eine Chomsky-Grammatik $G = (N, T, P, S)$ wird *monoton* genannt, wenn für jede Produktion $u ::= v \in P$ $|u| \leq |v|$ gilt.[1]

Für einen Ableitungsschritt $w = xuy \underset{u::=v}{\longrightarrow} xvy = w'$ gilt dann offenbar:

$$|w| = |x| + |u| + |y| \leq |x| + |v| + |y| = |w'|,$$

so dass durch einfache Induktion über die Länge von Ableitungen auch folgt:

$$|w| \leq |w'| \text{ für } w \xrightarrow[P]{*} w'.$$

Wenn man ein bestimmtes Wort der Länge n aus dem Startsymbol S ableiten will, treten zwischendurch also niemals längere Wörter auf. Wörter bis zur Länge n gibt es aber nur endlich viele, deren Zahl mit K_n bezeichnet wird. Denn es gilt für ein Alphabet A mit k Elementen:[2]

$$\#\{w \in A^* \mid |w| \leq n\} = 1 + k + k^2 + \ldots + k^n = \sum_{i=0}^{n} k^i = K_n < \infty.$$

9.2 Lösung des Wortproblems für monotone Grammatiken

Theorem 4
Für jede monotone Grammatik $G = (N, T, P, S)$ ist das Wortproblem lösbar.

[1] Für ein Wort v bezeichnet $|v|$ die Länge.
[2] Für eine endliche Menge X bezeichnet $\#X$ die Zahl der Elemente.

Beweis.
Sei $w_0 \in T^*$ mit $|w_0| = n$. Ohne Einschränkung kann man $n \geq 1$ annehmen, weil das leere Wort niemals von einer monotonen Grammatik erzeugt wird. Für jedes $k \geq 0$ sei S_k die Menge aller in höchstens k Schritten aus S ableitbaren Wörter, deren Länge n nicht überschreitet; d.h.

$$S_k = \{w \in (N \cup T)^* \mid S \xrightarrow[P]{j} w,\ j \leq k,\ |w| \leq n\}.$$

Offenbar gilt $S_0 = \{S\}$, und S_{k+1} lässt sich folgendermaßen aus S_k konstruieren:

$$S_{k+1} = S_k \cup \{w \in (N \cup T)^* \mid v \xrightarrow[P]{} w,\ v \in S_k,\ |w| \leq n\}. \qquad (*)$$

Damit ist nach Definition $S_k \subseteq S_{k+1}$, und wenn ein $m \in \mathbb{N}$ existiert mit $S_m = S_{m+1}$, so folgt

$$S_m = S_{m+1} = S_{m+2} = \ldots.$$

Da G nur endlich viele Produktionen hat, lässt sich leicht ein Algorithmus angeben, der ausgehend von $S_0 = \{S\}$ unter Verwendung von $(*)$ die Mengen S_k rekursiv konstruiert.

Falls es nun eine natürliche Zahl m gibt, so dass $S_m = S_{m+1}$ gilt, wäre unser Wortproblem entschieden, denn es gilt: $S \xrightarrow[P]{*} w_0$ gdw. $w_0 \in S_m$. Denn $S \xrightarrow[P]{*} w_0$ impliziert $w_0 \in S_k$, wobei k die Länge der Ableitungskette ist. Für $k \leq m$ gilt aber $S_k \subseteq S_m$ und für $k > m$ gilt $S_m = S_k$ nach Voraussetzung, also $w_0 \in S_m$. Die Umkehrung ist offenbar.

Es bleibt also die Existenz eines m mit $S_m = S_{m+1}$ zu zeigen. Nun gilt aber für alle $k \geq 0$ und alle $w \in S_k$, dass $|w| \leq n$ und damit

$$\#S_k \leq \{w \in (N \cup T)^* \mid |w| \leq n\} = K_n < \infty.$$

Wegen $S_k \subseteq S_{k+1}$ muss also nach spätestens $m = K_n$ Schritten $S_{m+1} = S_m$ gelten. □

9.3 So ein Aufwand

Der Algorithmus, der das Wortproblem für monotone Grammatiken löst, sammelt – beginnend mit dem Startsymbol – alle Wörter bis zu einer vorgegebenen Länge auf, die sich mit wachsender Schrittzahl ableiten lassen, bis keine neuen Wörter mehr hinzukommen. Die resultierende Menge S_m kann im schlechtesten Fall K_n Elemente enthalten, also exponentiell viele, falls das Alphabet mindestens zwei Elemente enthält. Deshalb ist der Algorithmus im schlechtesten Fall, der aber oft eintrifft, exponentiell und damit für praktische Zwecke unbrauchbar.

Es ist jedoch noch ungeklärt, ob es nicht eine polynomielle Lösung des Wortproblems monotoner Grammatiken gibt. Dies wird allerdings von Fachleuten für unwahrscheinlich gehalten.

Wenn man die jeweils nächste erfolgversprechende Regelanwendung richtig raten könnte, müsste man sich zwischendurch nur das bisher abgeleitete Wort merken, dessen Länge durch die Länge der Eingabe beschränkt ist. Probleme, die sich so lösen lassen, gehören zur Klasse NP-SPACE, wobei das "N" für *nichtdeterministisch* steht (und auf das Raten verweist) und "P-SPACE" auf den polynomiellen Platzbedarf verweist. Man weiß, dass die Klassen NP-SPACE und P-SPACE übereinstimmen, weil man den Nichtdeterminismus immer z.B. durch Backtracking beseitigen kann. Das Interessante an diesen Problemklassen ist, dass sie zu den größten bekannten gehören, die noch polynomiell lösbar sein könnten.

10 Die Chomsky-Hierarchie

Wie in Abschnitt 8.4 diskutiert wurde, haben Chomsky-Grammatiken die ungünstige Eigenschaft, dass das Wortproblem für die erzeugten Sprachen im allgemeinen unentscheidbar ist. Grammatiken, die solche Sprachen erzeugen, sind z.B. für die Definition der Syntax von Programmiersprachen offenbar ungeeignet. Es stellt sich daher die Frage, ob man durch geeignete zusätzliche Bedingungen – insbesondere an die Form der Regeln – zu eingeschränkten Klassen von Chomsky-Grammatiken gelangen kann, die bessere Eigenschaften haben, aber trotzdem noch genügend Allgemeinheit besitzen. Dies führt zur sogenannten *Chomsky-Hierarchie*. Wie der Name andeutet, handelt es sich dabei um eine Hierarchie mehr oder weniger eingeschränkter Typen von Chomsky-Grammatiken.

Eine Chomsky-Grammatik $G = (N, T, P, S)$ ist

(1) *kontext-sensitiv*, falls alle Regeln in P die Form $u_1 A u_2 ::= u_1 v u_2$ haben, wobei $u_1, u_2, v \in (N \cup T)^*$, $v \neq \lambda$ und $A \in N$;
(2) *kontextfrei*, falls $u \in N$ für alle Regeln $u ::= v \in P$;
(3) *rechtslinear* (auch *regulär* genannt), falls für alle Regeln $u ::= v \in P$ gilt, dass $u \in N$ und entweder $v \in T^*$ oder $v = v'B$ mit $v' \in T^+$ und $B \in N$. (Hierbei bezeichnet T^+ die Menge $T^* \setminus \{\lambda\}$.)

Monotone und kontext-sensitive Grammatiken werden auch *Grammatiken vom Typ 1* genannt, kontextfreie werden als *Typ-2-* und rechtslineare als *Typ-3-Grammatiken* bezeichnet. Allgemeine Chomsky-Grammatiken werden Grammatiken vom *Typ 0* genannt. Eine Sprache L ist vom Typ i, wenn es eine Grammatik dieses Typs gibt, die L erzeugt.

Der folgende Satz rechtfertigt, warum diese Typeneinteilung nicht zwischen monotonen und kontext-sensitiven Grammatiken unterscheidet.

Theorem 5
Monotone und kontext-sensitive Grammatiken erzeugen dieselbe Klasse von Sprachen.

Aus der Definition von kontext-sensitiven Grammatiken folgt sofort, dass sie monoton sind. Umgekehrt lässt sich zeigen, dass zu jeder monotonen Grammatik eine kontextsensitive konstruiert werden kann, die dieselbe Sprache erzeugt (in einem solchen Fall

spricht man auch von einer *Normalform*-Grammatik). Wer genaueres wissen will, kann den Beweis z.B. in [EP00, Satz 8.1.1] nachlesen.

Die Berechtigung, von einer *Hierarchie* zu sprechen, liefert der nächste Satz.

Theorem 6
Sei $\mathcal{L}_i$ die Menge aller Sprachen des Typs i ($i \in \{0, \ldots, 3\}$). Dann gilt:

1. $\mathcal{L}_1 \subsetneqq \mathcal{L}_0$,
   d.h. Monotonie bzw. Kontext-Sensitivität ist eine echte Einschränkung.
2. $\{L \setminus \{\lambda\} \mid L \in \mathcal{L}_2\} \subsetneqq \mathcal{L}_1$,
   d.h. bis auf die bei monotonen Grammatiken offenbar nicht bestehende Möglichkeit, das leere Wort zu erzeugen, ist Kontextfreiheit eine echte Einschränkung gegenüber Kontext-Sensitivität.
3. $\mathcal{L}_3 \subsetneqq \mathcal{L}_2$,
   d.h. Rechtslinearität ist eine echte Einschränkung gegenüber Kontextfreiheit.

Aus der in Kapitel 9 behandelten Entscheidbarkeit des Wortproblems für Typ-1-Sprachen lässt sich die erste Aussage des obigen Satzes – $\mathcal{L}_1 \subsetneqq \mathcal{L}_0$ – als Folgerung ableiten. Zusammen mit der Unentscheidbarkeit des Wortproblems im allgemeinen Fall ergibt sich nämlich aus der Entscheidbarkeit für $\mathcal{L}_1$ die Ungleichheit $\mathcal{L}_1 \neq \mathcal{L}_0$ (und $\mathcal{L}_1 \subseteq \mathcal{L}_0$ gilt ohnehin per Definition).

Der Nachweis, dass die beiden anderen Inklusionen echt sind, benötigt Werkzeuge, die erst später im Skript bereitgestellt werden.

11 Ein unentscheidbares Problem für kontextfreie Grammatiken

Was verrät die Syntax über die Semantik? Das ist eine Kernfrage der Informatik, weil die semantische Ebene das Gewünschte und Interessierende repräsentiert, während nur die syntaktischen Beschreibungen explizit verfügbar sind. Das Halteproblem für Programme (und Turingmaschinen) und das Wortproblem für Grammatiken sind typische Probleme dieser Art.

Dummerweise sind viele semantische Fragen an syntaktischen Gebilden unentscheidbar – selbst dann noch, wenn man die Syntax stark einschränkt. Ein Beispiel dieser Art wird in diesem Kapitel für kontextfreie Grammatiken vorgestellt. Während das Leerheitsproblem für kontextfreie Grammatiken ($L(G) = \emptyset\,?$) entscheidbar ist, erweist sich bereits die Frage nach der Leerheit des Durchschnitts zweier kontextfreier Sprachen ($L(G_1) \cap L(G_2) = \emptyset\,?$) als unentscheidbar.

Um das zu beweisen, werden Postsche Korrespondenzprobleme, deren Lösbarkeit bekanntlich unentscheidbar ist, auf das Durchschnittsleerheitsproblem reduziert. Solche Reduktionen sind typisch für den Nachweis von Unentscheidbarkeit.

Betrachte dazu ein Postsches Korrespondenzproblem $PCP = ((u_1, \ldots, u_n), (v_1, \ldots, v_n))$ über dem Alphabet T. PCP ist lösbar, wenn es eine nichtleere Indexfolge $i_1 \cdots i_k$ mit $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$ gibt. Die Konkatenationen aus beiden Listen zu Indexfolgen lassen sich gleichzeitig kontextfrei erzeugen, wenn man die zweite Konkatenation transponiert. Die entsprechende Grammatik G_{PCP} hat folgende Produktionen:

$$\left.\begin{array}{l} S ::= u_i A \; \mathit{trans}(v_i) \\ A ::= u_i A \; \mathit{trans}(v_i) \mid \$ \end{array}\right\} \text{ für } i = 1, \ldots, n.$$

Offensichtlich lassen sich damit aus S die terminalen Wörter der Form

$$u_{i_1} \cdots u_{i_k} \$ \; \mathit{trans}(v_{i_k}) \cdots \mathit{trans}(v_{i_1}) = u_{i_1} \cdots u_{i_k} \$ \; \mathit{trans}(v_{i_1} \cdots v_{i_k})$$

ableiten. Mit anderen Worten ist PCP genau dann lösbar, wenn $L(G_{PCP})$ ein Wort der Form $w\$ \; \mathit{trans}(w)$ enthält.

Solche Wörter lassen sich aber bekanntlich durch eine kontextfreie Grammatik G_{mirror} mit den Produktionen

$$S ::= \$ \mid xSx \text{ für } x \in T$$

erzeugen. Also ist PCP genau dann lösbar, wenn $L(G_{PCP}) \cap L(G_{mirror}) \neq \emptyset$.

Wäre nun das Durchschnittsleerheitsproblem für kontextfreie Grammatiken entscheidbar, gälte das insbesondere für die Grammatiken G_{PCP} und G_{mirror}, so dass sich die Lösbarkeit von Postschen Korrespondenzproblemen als entscheidbar erwiese. Der Widerspruch ist nur dadurch auflösbar, dass die Annahme falsch ist. Die Frage nach der Leerheit des Durchschnitts von Sprachen, die von kontextfreien Grammatiken erzeugt werden, muss also unentscheidbar sein.

Liefert eine Indexfolge eine Lösung für ein PCP, so bildet jede Wiederholung der Indexfolge ebenfalls eine Lösung. Ein PCP hat also unendlich viele Lösungen, wenn es überhaupt Lösungen hat. Mit der obigen Übersetzung ist PCP genau dann lösbar, wenn $L(G_{PCP}) \cap L(G_{mirror})$ unendlich ist. Die Frage nach der (Un-)Endlichkeit des Durchschnitts zweier kontextfrei erzeugter Sprachen erweist sich also ebenso wie die Frage nach der Leerheit des Durchschnitts als unentscheidbar.

Das Wortproblem für den Durchschnitt ist übrigens entscheidbar, denn ein Wort liegt genau dann im Durchschnitt, wenn es in beiden Sprachen liegt. Das Wortproblem für die einzelnen kontextfreien Sprachen ist aber entscheidbar.

Literatur

Die folgenden Literaturhinweise geben einen Einblick in die Theorie der formalen Sprachen und verschiedene andere Gebiete der Theoretischen Informatik. Die Bücher [Sal73, Sal78, MAK88, ST99, EP00, Hed00, HMU01, HMU02] behandeln formale Sprachen ausführlich, wobei [Sal78] und [HMU02] die deutschen Übersetzungen von [Sal73] und [HMU01] sind.

Wer noch weitergehende Informationen sucht, wird vermutlich im dreibändigen *Handbook of Formal Languages* [RS97] fündig. In [LP81, Coh86, Woo87, AU95, AB02] werden vorrangig kontextfreie Sprachen untersucht. Die Verbindung von formalen Sprachen und der syntaktischen Behandlung von Programmiersprachen wird umfassend in [ASU86] abgehandelt.

Literatur

[AB02] Alexander Asteroth and Christel Baier. *Theoretische Informatrik*. Pearson Studium, München, 2002.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles,Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[AU95] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, New York, 1995.

[Coh86] Daniel I.A. Cohen. *Introduction to Computer Theory*. John Wiley & Sons, Inc., New York, 1986.

[DSW94] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages. Fundamentals of Theoretical Computer Science*. Academic Press, San Diego, 1994.

[EP00] Katrin Erk and Lutz Priese. *Theoretische Informatik*. Springer, Berlin Heidelberg, 2000.

[Hed00] Ulrich Hedtstück. *Einführung in die Theoretische Informatik*. Oldenbourg, München, 2000.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2001.

[HMU02] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, 2002.

[Kre04] Hans-Jörg Kreowski. Theoretische Informatik 1, 2003/2004. Skript zur Veranstaltung.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[MAK88] Robert N. Moll, Michael A. Arbib, and A.J. Kfoury. *An Introduction to Formal Language Theory*. Springer, New York, 1988.

[RS97] Grzegorz Rozenberg and Arto K. Salomaa, editors. *Handbook of Formal Languages*. Springer, 1997. Vol. 1: Word, Language, Grammar. Vol. 2: Linear Modeling. Vol. 3: Beyond Words.

[Sal73] Arto K. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[Sal78] Arto K. Salomaa. *Formale Sprachen*. Springer, Berlin, 1978.

[Sch95] Uwe Schöning. *Theoretische Informatik – kurzgefaßt* (2. Auflage). Spektrum Akademischer Verlag, 1995.

[ST99] Dan A. Simovici and Richard L. Tenney. *Theory of Formal Languages with Applications*. World Scientific, Singapore, 1999.

[VW00] Gottfried Vossen and Kurt-Ulrich Witt. *Grundlagen der Theoretischen Informatik mit Anwendungen*. Vieweg, Braunschweig, 2000.

[Wag94] Klaus W. Wagner. *Einführung in die Theoretische Informatik – Grundlagen und Modelle*. Springer, Berlin Heidelberg, 1994.

[Wät94] Dietmar Wätjen. *Theoretische Informatik – Eine Einführung*. Oldenbourg, München, 1994.

[Weg93] Ingo Wegener. *Theoretische Informatik*. B.G. Teubner, Stuttgart, 1993.

[Woo87] Derick Wood. *Theory of Computation*. John Wiley & Sons, New York, 1987.