Prof. Dr. Hans-Jörg Kreowski
Studiengang Informatik
MZH 3260
Tel.: 2956, 3697 (Sekr.), Fax: 4322
email: kreo@informatik.uni-bremen.de

Sommersemester 1997

Theoretische Informatik 1

Das schriftliche Material zur gleichnamigen Lehrveranstaltung besteht vorerst aus den folgenden Kapiteln:

1. Szenarien – theoretisch
   begründet den Sinn von Theorie im allgemeinen und von Berechenbarkeits- und Komplexitätstheorie im besonderen (vgl. Studienordnung Informatik vom 1.10.1993, Anlagen, S. 6).
2. Lauter Wörter
   führt wichtige Eigenheiten von Wörtern ein, die zum Entwurf und zur Analyse von Algorithmen auf dieser Datenstruktur benötigt werden.
3. Berechnung von Operationen auf Zeichenketten mit Hilfe bedingter Gleichungen – die Sprache CE-S
   erweitert die Überlegungen des zweiten Kapitels zu einer Spezifikationssprache für Operationen auf Zeichenketten und liefert damit einen Ansatz, in dem Spezifikationen eine formale Semantik besitzen und Korrektheitseigenschaften nachgewiesen werden können. Fast nebenbei ergibt sich eine Präzisierung des Berechenbarkeitsbegriffs.
5. Matritzenmultiplikation schneller und schneller
   vergleicht den Zeitaufwand verschiedener Algorithmen am Beispiel von Matritzenmultiplikation.
6. PASCALchen macht Funktionen berechenbar
   präzisiert den Begriff der berechenbaren Funktion anhand der imperativen Programmiersprache PASCALchen.
7. Programme sind aufzählbar – doch ihr (Ver-)Halten macht Kummer
   zeigt einerseits, daß alle berechenbaren Funktionen/PASCALchen-Programme mittels eines Algorithmus aufgezählt werden können und führt andererseits anhand des Halteproblems von Programmen an die Grenzen der Berechenbarkeit.

8. Ein Programm rechnet für alle
   stellt eine berechenbare universelle Funktion vor und kommentiert die Grenzlinie, welche diese vom Halteproblem trennt.
9. Turing-Maschinen
   führt den Begriff der Turingsmaschinen ein und stellt einen Bezug zwischen diesem Maschinenmodell und der Sprache CE-S her.
10. Literaturhinweise
    enthält eine Liste von Büchern, die einen Einblick in die Gebiete der Theoretischen Informatik bieten bzw. das dafür benötigte Basiswissen liefern.

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 Berechenbarkeit- und Komplexitätstheorie angerissen.

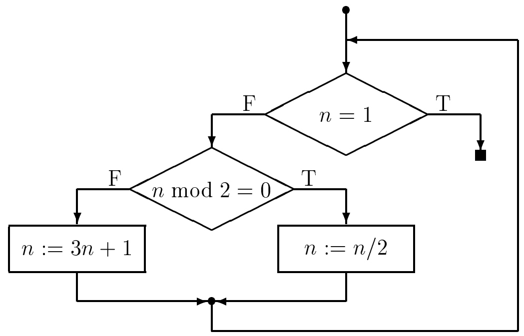
1.1 Terminationstest endlich lieferbar

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 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 DM 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 folgendes Flußdiagramm 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

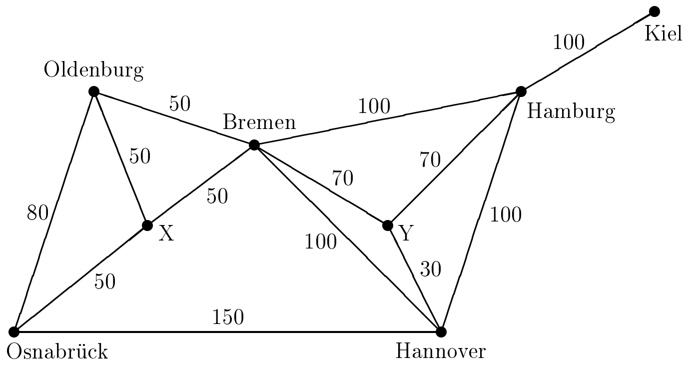
In einem Geldtransportunternehmen, das auch andere Wertsachen befördert, wird erwogen, die tägliche Fahrtroutenplanung zu automatisieren. Bei dem dafür zu erstellenden Softwaresystem muß von folgenden Vorgaben ausgegangen werden:

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



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

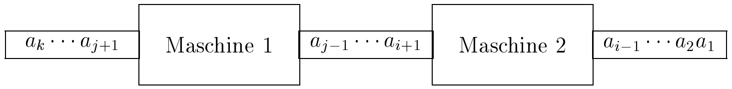
Ein mittelständisches Unternehmen, das Spezialanfertigungen von elektronischen Geräten herstellt, möchte seine Produktion weiter automatisieren. Dazu benötigt es ein Maschinenbelegungsprogramm, das für täglich durchschnittlich 75 Aufträge mit unterschiedlichen Fertigungsplänen den Durchlauf durch 128 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 Zuverlässigkeitgarantien 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. DM 100.000,-; optimale Maschinenbelegung; ungünstigste Laufzeit: m^k.
2. DM 200.000,-; optimale Maschinenbelegung; durchschnittliche Laufzeit: $k \cdot m!$.
3. DM 150.000,-; höchstens 100% über optimaler Maschinenbelegung; Laufzeit: $k^2 \cdot m^3$.
4. DM 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. DM 80.000,-; Optimum; Laufzeit: $c \cdot k$, wobei c zwischen 10.000 und 100.000 liegt. Allerdings muß dafür noch ein Zusatzprogramm für rund DM 10.000,- beschafft werden, das für einen Auftrag, der als nächstes auf mehreren Stationen gefertigt werden kann, die aktuell günstigste auswählt.
7. DM 250.000,-; Optimum; durchschnittliche Laufzeit: $k \cdot m^2$.
8. Das Angebot 7 zum doppelten Preis, dafür aber Laufzeit für den schlechtesten Fall.
9. DM 300.000,-; optimale Lösung mit 55% Garantie; Laufzeit: $128 \cdot k \cdot log(k)$. (Anmerkung: Programm benötigt einen guten Zufallsgenerator.)
10. DM 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?

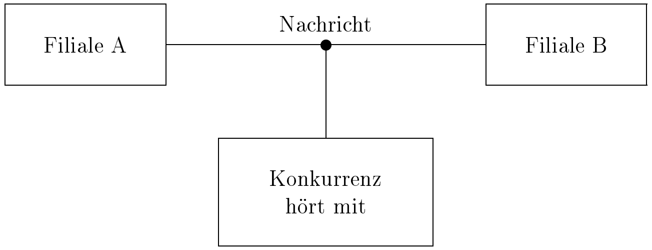
Das Fließband, bei dem Aufträge in einer vorgegebenen Folge durch die festinstallierte Sequenz der Maschinen geschleust werden, hat – so scheint es – als Maschinenbelegungsplan ausgedient.



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

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. Um das Bankgeheimnis zu wahren und mißbräuchliche Verwendung der Informationen zu verhindern, ist eine Verschlüsselung unumgänglich.



Unrealistisch wäre ein Konzept, in dem 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 ein übereifriger Vorstandsassistent, daß er 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 der Vorstandsassistent 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:

$$- - u_1 - u_2 - \cdots - u_k - \qquad -v_1 - v_2 - \cdots - v_k - -$$

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

$$- - aac - a - bb - aac - b - \qquad -a - baa - acb - ca - acb - -$$

Die Dechiffrierexperten 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 Systemanalytiker und Programmierer 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,– DM 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 leldiglich 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 seine 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 Agenten, 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 Politiker 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 Vertreter 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 simple 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 die folgende Tabelle:

n	n^2	2^n
10	100	1024
20	400	1048576
30	900	1073741824
40	1600	1099511627776
50	2500	1125899906842624
60	3600	1152521504606846976
70	4900	1180192260717411303424
		(ohne Gewähr)

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 Bereich einführende Aspekte behandelt.

2 Lauter Wörter

Informatik ist ohne Zeichenketten 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.

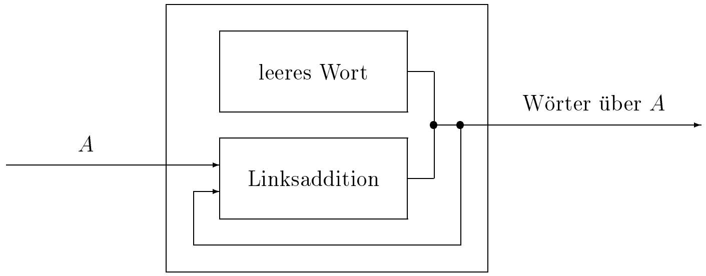
In diesem Abschnitt sind einige wichtige Informationen zum Umgang mit Zeichenketten zusammengestellt. Vieles davon wird verschiedentlich in die Überlegungen zur theoretischen Informatik während des kommenden Jahres eingehen, ohne daß diese Tatsache immer ausführlich gewürdigt wird.

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" 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 läßt sich folgendermaßen illustrieren:



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 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 letzte 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):

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

   Und 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:

   $$\lambda\lambda \underset{1}{=} \lambda \quad \text{und} \quad (xv)\lambda \underset{1}{=} x(v\lambda) \underset{*}{=} xv,$$

   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 nun 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 $\mathit{head}(v) \equiv \mathit{head}(w)$ und $\mathit{tail}(v) \equiv \mathit{tail}(w)$, 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) $\mathit{length}(\lambda) = 0$
   (b) $\mathit{length}(xv) = \mathit{length}(v) + 1 \qquad \text{für} \quad x \in A, v \in A^*$
   (c) $\mathit{count}(x, \lambda) = 0$
   (d) $\mathit{count}(x, yv) = \mathit{if} \ \ x \equiv y \ \ \mathit{then} \ \mathit{count}(x, v) + 1 \ \ \mathit{else} \ \ \mathit{count}(x, v)$
   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*: $A^* \to \mathbb{N}$ definiert wird, läßt sich mit Hilfe des Induktionsprinzips zeigen:

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

   IV: Für ein Wort v sei $\mathit{length}(v)$ 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 *count*: $A \times A^* \to \mathbb{N}$ 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):

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

$$= length(v) + 1 + length(w)$$
$$= length(av) + length(w).$$

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 $w = x_1 \cdots x_n$ ist 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. $\mathbb{Z}$ 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 (konstanten) 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 $f : D_1 \times \cdots \times D_m \to D$ wobei f ein Name ist und $D, D_1, \ldots, D_m$ Typen sind. Das schließt den Fall $m = 0$ ein, durch den Konstanten $c : \to D$ 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) $(c : \to D) \in DECL$ impliziert $c \in T_D$;
   (ii) $(x \in D) \in X$ impliziert $x \in T_D$;
   (iii) $(f : D_1 \times \cdots \times D_k \to D) \in DECL$ 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 Konstante und Variable 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:

- $f_1, \ldots, f_n : D_1 \times \cdots \times D_m \to D$ für $f_1 : D_1 \times \cdots \times D_m \to D, \ldots, f_n : D_1 \times \cdots \times D_m \to D$.
- $x_1, \ldots, x_p \in D$ 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 ($p = 0$).
- Den *falls*-Teil einer bedingten Gleichung kann man weglassen, wenn keine Bedingung folgt ($n = 0$).
- Infix-Notation $t_1\, f\, t_2$ für $f(t_1, t_2)$, wenn f eine zweistellige Grundoperation ist.
- t für $t = T$.
- $\neg t$ für $t = F$; $t_1 \not\equiv t_2$ für $\neg(t_1 \equiv t_2)$; $t_1 > t_2$ für $\neg(t_1 \leq t_2)$.
- $t = \mathit{if}\ \ b\ \ \mathit{then}\ \ t_1\ \ \mathit{else}\ \ t_2$ 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: $\mathit{search}: A \times A^* \to \mathit{BOOL}$
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: $\mathit{count}: A \times A^* \to \mathbb{N}$
vars: $x, y \in A, v \in A^*$
eqns: $\mathit{count}(x, \lambda) = 0$
$\mathit{count}(x, yv) = \mathit{if}\ \ x \equiv y\ \ \mathit{then}\ \ \mathit{count}(x, v) + 1\ \ \mathit{else}\ \ \mathit{count}(x, v)$

diff
opns: $\mathit{diff}: A \times A \times A^* \to \mathbb{Z}$
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: $\mathit{is\text{-}sorted}: A^* \to \mathit{BOOL}$
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: $\mathit{sort}_0: A^* \to A^*$
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 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 $\overset{*}{\underset{0}{\longleftrightarrow}}$ 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) $(L = R$ *falls* $L_1 = R_1, \ldots, L_n = R_n$ *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) $(f : D_1 \times \cdots \times D_k \to D) \in DECL$ und $t_i \overset{*}{\longleftrightarrow} t'_i$ mit $t_i, t'_i \in T_{Di}$ 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 $t' \overset{*}{\longleftrightarrow} t$; $t \overset{*}{\longleftrightarrow} t'$ und $t' \overset{*}{\longleftrightarrow} t''$ implizieren $t \overset{*}{\longleftrightarrow} t''$.

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) $(c : \to D) \in DECL$ impliziert $c[a] = c$,

(ii) $(x \in D) \in X$ impliziert $x[a] = a(x)$,

(iii) $(f : D_1 \times \cdots \times D_k \to D) \in DECL$ $(k > 0)$ 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.

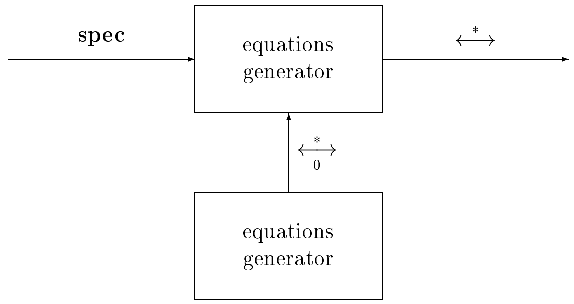
Man mag sich nun noch wundern, wo die vorgegebene Relation herkommen kann. Sie ist beliebig wählbar, aber es gibt eine besonders interessante und sinnvolle Wahl. Man kann annehmen, daß alle in der aktuellen Spezifikation über die Grundoperationen hinaus verwendeten Operationen vorher bereits in CE-S 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 eines der anderen Typen ist, nimmt man deren 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.

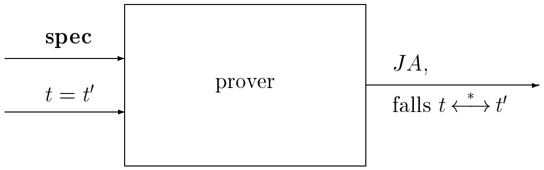
3.4 Operationelle Semantik von CE-S

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 läßt sich folgendermaßen veranschaulichen:



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 gleichwertig 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:

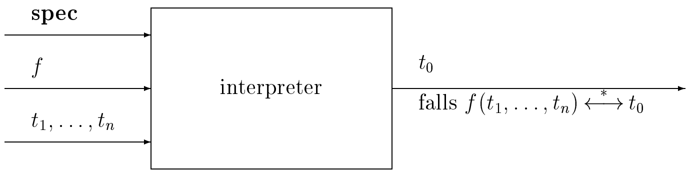


Gibt man einen Term vor und läßt wieder den Gleichungsgenerator laufen, kann man auf Gleichungen warten, die den gegebenen Term als linke Seite haben. Die rechten Seiten kann man dann als Berechnungsergebnisse des Terms ansehen. Das gilt insbesondere, wenn der Ausgangsterm eine Operation auf bekannte 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 *Wert* genannt.
2. Sei $f : D_1 \times \cdots \times D_k \to D_0$ eine in **spec** deklarierte Operation. Seien $t_i \in T_{D_i}$ für $i = 0, \ldots, k$ Werte. Dann ist t_0 ein *berechneter Wert* von $f(t_1, \ldots, t_k)$, wenn gilt:
$$f(t_1, \ldots, t_k) \xleftrightarrow{*} t_0.$$

Die Idee hinter der operationellen Semantik ist folgende: Die Grundterme repräsentieren die Werte der verfügbaren Typen, d.h. alle Zeichen, alle Zeichenketten, alle natürlichen und ganze Zahlen bzw. die beiden Wahrheitswerte. Die spezifizierten Operationen sollen berechnet werden. Wenn $f : D_1 \times \cdots \times D_k \to D$ 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 Werte $t_i \in T_{D_i}$ gegeben sind, ist also die Frage, zu welchen Grundtermen 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:



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:
$$swap(abc) \xleftrightarrow{*} \begin{cases} cba \\ bac \\ acb \end{cases}$$

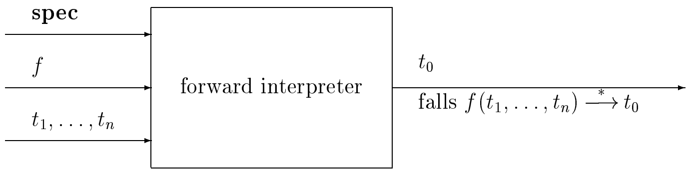
Die CE-S-spezifizierten Operationen sind also im allgemeinen relational. Um funktionale Operationen zu erhalten, muß man zusätzlich Vorkehrungen treffen. Ähnlich verhält es sich, wenn man vermeiden möchte, daß eine Operation unvollständig definiert ist. Denn in CE-S kann man leicht Spezifikation schreiben, 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 $\xrightarrow{*}$ 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:



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 $L, R \in T_D$ mit 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 $t' = C[a_R]$.

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.

3.7 Lemma (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 m.

IS: Betrachte $C = f(t_1, \ldots, t_k)$ mit Rekursionstiefe $m+1$. Dann haben die Argumentterme 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 $t_{i_0}[a_L] \to t_{i_0}[a_R]$, 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:

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

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

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 aufeinanderfolgende Zeichen in $>$-Beziehung stehen. Um einzusehen, daß das spezifische Verfahren sortiert, muß also nur noch sichergestellt werden, daß keine Zeichen verloren gegangen oder hinzugekommen sind. Das wird exemplarisch gezeigt. Dabei wird eine gegenüber der früheren Definition leicht modifizierte Spezifikation des Zeichenzählens verwendet:

count

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

Beobachtung

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

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

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*:

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

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:

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

Beobachtung

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

(1) $left(w)right(w) = w,$

(2) $\mathit{length}(\mathit{right}(w)) \le \mathit{length}(\mathit{left}(w)) \le \mathit{length}(\mathit{right}(w)) + 1$,

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

$$\begin{aligned}
&\mathit{left}(\lambda)\mathit{right}(\lambda) = \lambda\, \mathit{right}(\lambda) = \lambda\lambda = \lambda \text{ sowie}\\
&\mathit{left}(x)\mathit{right}(x) = x\, \mathit{right}(x) = x\lambda = x.
\end{aligned}$$

$$\begin{aligned}
&\mathit{length}(\mathit{left}(\lambda)) = \mathit{length}(\lambda) = \mathit{length}(\mathit{right}(\lambda)) \le \mathit{length}(\mathit{right}(\lambda)) + 1 \text{ sowie}\\
&\mathit{length}(\mathit{left}(x)) = \mathit{length}(x) = \mathit{length}(\lambda) + 1\\
&\quad = \mathit{length}(\mathit{right}(x)) + 1 \ge \mathit{length}(\mathit{right}(x)).
\end{aligned}$$

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

IS: Betrachte ein Wort w der Länge $n \ge 2$. 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:

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

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

Analog ergibt sich die Behauptung für die Längen:

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

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

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 $x \le y$, so ergibt sich die Behauptung wie folgt:

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

Die Anwendung der Induktionsvoraussetzung ist möglich wegen:

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

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

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:

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

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:

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

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 $\mathit{length}(t_{i_j}) = n_j$ 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 $g : \mathbb{N}^k \to \mathbb{R}^+$ 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 $n_0 \in \mathbb{N}$ gegeignet 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.

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 $\mathit{length}(t_{i_j}) = n_j$ 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.

4.4 Beispiel

Als weiteres Beispiel wird der Aufwand des Sortierens durch Mischen ermittelt.

< Skriptfassung folgt >

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 meist verwendeten 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 $A = (a_{ij})_{i=1\cdots m, j=1\cdots n}$, das in PASCAL und ähnlichen Sprachen etwa als zweidimensionales Feld vereinbart werden könnte:

type matrix=*array*$[1 \cdots m, 1 \cdots n]$ *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 $A = (a_{ij})_{i=1\cdots m, j=1\cdots n}$ und $B = (b_{jk})_{j=1\cdots n, k=1\cdots p}$, 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, wie viele 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 $i = 1 \cdots m$ und $k = 1 \cdots p$ 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 $\cdots$ und $b_{nk}\ Z_n$;
Z_j braucht $a_{ij}\ R_i$ für $j = 1 \cdots n$;
E_k braucht also $a_{i1} \cdot b_{1k}$ und $\cdots$ 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 $C = (c_{ik})_{i=1\cdots m, k=1\cdots p}$. 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:

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

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 miteingerechnet 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 klassiche 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 Wingograd 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:

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

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:

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

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:

$$T^{\sum}_{MULT}(n) = m = \frac{1}{2}n, \quad T^{\sum}_{ADD}(n) = 2m + m - 1 = \frac{3}{2}n - 1.$$

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:

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

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:

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

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

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

wobei die letzte Gleichung gilt wegen

$$\begin{aligned} t_1 &= m_1 + m_2 = s_2 s_6 + m_2 = (s_1 - a_{11})(b_{22} - s_5) + m_2 \\ &= (a_{21} + a_{22})b_{22} - (a_{21} + a_{22})(b_{12} - b_{11}) - a_{11}b_{22} + a_{11}(b_{12} - b_{11}) + m_2 \\ &= a_{21}b_{22} + a_{22}b_{22} - a_{21}b_{12} + a_{21}b_{11} - a_{22}b_{12} + a_{22}b_{11} - a_{11}b_{22} + a_{11}b_{12} \end{aligned}$$

und

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

(3) $$\begin{aligned} c_{12} &= t_3 + m_6 = t_1 + m_5 + m_6 = t_1 + m_5 + s_4 b_{22} \\ &= t_1 + m_5 + (a_{12} - s_2)b_{22} = t_1 + m_5 + a_{12}b_{22} - (s_1 - a_{11})b_{22} \\ &= t_1 + m_5 + a_{12}b_{22} - (a_{21} + a_{22})b_{22} - a_{11}b_{22} \\ &= t_1 + m_5 + a_{12}b_{22} - a_{21}b_{22} - a_{22}b_{22} - a_{11}b_{22} \\ &= a_{11}b_{12} + a_{12}b_{22}, \end{aligned}$$

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

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. Sondern es wird nur verwendet, daß eine Addition existiert, die assoziativ und kommuntativ 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 aus mit $n = 2m$ und unterteilt beide längs und quer in der Mitte, entstehen 2,2-Matrizen mit m,m-Matrizen als Einträge, 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:

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

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:

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

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:

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

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*):

$$T_{MULT}^{STRASSEN}(n) = 7^k = (2^{ld7})^k = (2^k)^{ld7} = n^{ld7},$$
$$T_{ADD}^{STRASSEN}(n) = \sum_{i=1}^{k} 15 \cdot 7^{i-1} \left(\tfrac{n}{2^i}\right)^2 = \sum_{i=1}^{k} 15 \cdot 7^{i-1} \cdot \tfrac{n^2}{4^i}$$
$$\underset{(1)}{=} 15 \cdot \tfrac{n^2}{4} \cdot \sum_{i=0}^{k-1} \left(\tfrac{7}{4}\right)^i \underset{(2)}{=} 15 \cdot \tfrac{n^2}{4} \cdot \tfrac{\left(\frac{7}{4}\right)^k - 1}{\frac{7}{4}-1} = 5n^2\left(\tfrac{7}{4}\right)^k - 5n^2 \underset{(3)}{=} 5n^{ld7} - 5n^2,$$

wobei sich (1) durch Ausklammern, (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:

$$n^2 \cdot \left(\frac{7}{4}\right)^k = (2^k)^2 \cdot \left(\frac{7}{4}\right)^k = 4^k \cdot \left(\frac{7}{4}\right)^k = 7^k = n^{ld7}.$$

Es bleibt anzumerken, daß $ld7$ 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 Additonen) überlegen ist:

n	klassisch: n^3	Winograd: $\frac{1}{2}n^3 + n^2$	Strassen: $n^{2,81}$
8	512	320	343
64	262 144	135 168	112 649
256	16 777 216	8 454 144	5 764 801
$\vdots$	$\vdots$	$\vdots$	$\vdots$

6 PASCALchen macht Funktionen berechenbar

In 1.9. ist begründet, warum der Berechenbarkeitsbegriff näher untersucht werden soll. Aus den Überlegungen zum Aufwand ergeben sich weitere Anlässe dazu:

(A) Um den Aufwand von Algorithmen zu bestimmen, wurden teils Additionen und Multiplikationen gezählt, teils Gleichungsanwendungen. Lassen sich so die erzielten Ergebnisse untereinander überhaupt vergleichen (oder werden "Birnen und Äpfel" gegenübergestellt)? Will man die Vergleichbarkeit sicherstellen, ist eine für alle Algorithmen gemeinsame Zählgröße und damit eine einheitliche *Berechnungsgrundlage* erforderlich, wie sie CE-S im Prinzip bereits bietet.

(B) Vielleicht kann durch eine geschickte Wahl der Berechnungsgrundlage sogar die Komplexität von Problemen, d.h. der Mindestaufwand von Lösungsalgorithmen, verbessert werden.

(C) Die bisher betrachteten Probleme hatten alle eine endliche Komplexität in dem Sinne, daß es Lösungsalgorithmen gibt, die für jede Eingabe nur endlich viele zählbare Aktionen ausführen. Gilt das für jedes auf dem Rechner ausführbare, "berechenbare" Problem? Oder gibt es Probleme, deren Lösungen für manche Eingaben über jede Schranke hinaus rechnen müssen? Mit anderen Worten: Muß erlaubt werden, daß Aufwandsfunktionen partiell definiert sein können? Kann Aufwand auch unendlich werden?

(D) Welche Probleme sind eigentlich "berechenbar"? Wie sehen Probleme aus, die nicht "berechenbar" sind? Gibt es das überhaupt?

Ziel dieses Abschnitts ist, den Begriff der berechenbaren Funktion zu präzisieren. Die Betonung liegt dabei auf einem exakten Vorgehen, um einen Teil der gestellten Fragen möglichst zuverlässig zu beantworten und den Spielraum für Einwände einzuengen. Der übliche Weg, zu berechenbaren Funktionen zu kommen, ist die Einführung einer Sprache zum Schreiben von Algorithmen (Syntax) und die Bereitstellung eines Kalküls, mit dem Algorithmen ausgeführt werden können (Semantik). Es gibt eine breite Palette an Möglichkeiten, das zu konkretisieren; sie reicht von maschinenorientierten bis zu problemorientierten Sprachen. Hier wird eine für Informatikerinnen und Informatiker naheliegende Wahl getroffen: die imperative Programmiersprache PASCALchen.

PASCALchen ist eine magere Sprache, die vage an PASCAL erinnert. PASCALchen verzichtet nahezu ganz auf Benutzungskomfort. Die wesentlichen Unterschiede zu üblichen imperativen Programmiersprachen sind, daß der einzige erlaubte Datentyp die natürlichen Zahlen sind und – vorläufig – Rekursion verboten ist. Das wesentliche Konstrukt von PASCALchen ist die Iteration; deshalb werden die Programme der Sprache *while*-Programme genannt.

6.1 Die Sprache PASCALchen

1. Die Programme sind aus folgenden *Zeichen* aufgebaut:

   (a) *Variablennamen* sind Wörter aus Großbuchstaben und Ziffern, die mit einem Buchstaben beginnen.

   (b) *Operatorensymbole*, die zum Ändern der Variablenwerte dienen, sind *succ* für die Nachfolgerfunktion, *pred* für die Vorgängerfunktion und 0 für die Nullkonstante.

   (c) Als *Relationssymbol* zum Vergleich von Variablenwerten steht $\neq$ für die Ungleichheitsabfrage zur Verfügung.

   (d) *Programmsymbole* sind := für die Zuweisung, ; für die Reihung sowie *begin*, *end*, *while*, *do*.

   (e) Als *Hilfssymbole* zum Anwenden der Operatoren gibt es runde Klammern (und).

2. Programme werden nach folgender *Syntax* gebildet:
   (a) Ein *while-Programm* ist eine Reihungsanweisung.
   (b) Eine *Reihungsanweisung* (kurz: *Reihung*) hat die Form

   $$\mathit{begin}\ S_1; S_2, \ldots; S_m\ \mathit{end},$$

   wobei $S_1, \ldots, S_m$ für $m \geq 0$ Anweisungen sind. Der Fall $m = 0$ ist so gemeint, daß die Reihungsanweisung dann einfach die Form *begin end* hat, was einem *skip* entspricht.
   (c) Eine *Wiederholungsanweisung* (kurz: *Wiederholung*) hat die Form

   $$\mathit{while}\ X \neq Y\ \mathit{do}\ S,$$

   wobei S eine Anweisung ist und X, Y zwei Variablen.
   (d) Eine *Zuweisungsanweisung* (kurz: *Zuweisung*) hat die Form

   $$X := 0 \ \text{ oder } \ X := \mathit{succ}(Y) \ \text{ oder } \ X := \mathit{pred}(Y),$$

   wobei X, Y zwei Variablen sind.
   (e) Eine *Anweisung* kann eine Reihung, eine Wiederholung oder eine Zuweisung sein. Mit den folgenden Bemerkungen werden dann auch noch Makroanweisungen als Anweisungen zugelassen.
3. Die Variablen müssen nicht deklariert werden, da sie alle vom Typ der natürlichen Zahlen $\mathbb{N}$ sind, für die es, korrespondierend zu den syntaktischen Größen, eine Konstante 0 gibt, die Nachfolger- und Vorgängerfunktionen $\mathit{succ}, \mathit{pred} : \mathbb{N} \to \mathbb{N}$ und einen Ungleichheitstest $\neq: \mathbb{N}^2 \to \{\mathit{TRUE}, \mathit{FALSE}\}$. Dabei ist *pred* definiert durch $\mathit{pred}(0) = 0$ und $\mathit{pred}(n+1) = n$ für alle $n \in \mathbb{N}$. Die anderen Funktionen arbeiten in bekannter Weise.

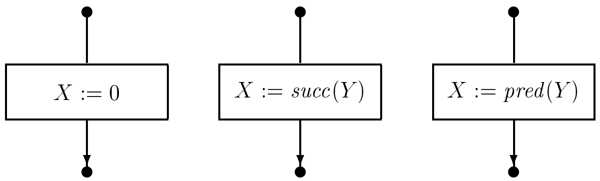
6.2 Bemerkungen

1. Daß PASCALchen so knapp zugeschnitten ist, hat einen pragmatischen und einen theoretischen – fast philosophischen – Grund.

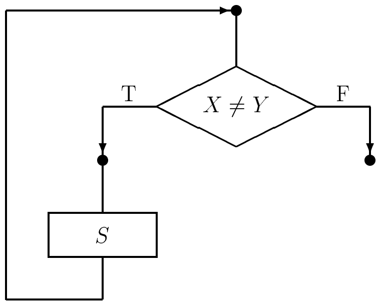
   Es muß noch definiert werden, was *while*-Programme leisten. Das ist für wenige Konstrukte oft einfacher als für ein breites Spektrum. Andererseits ist das Ziel, das Berechenbare aufzuspüren. Von Vorgaben jedoch läßt sich nicht beweisen, daß sie Berechenbares darstellen, sondern es ist allenfalls plausibel und muß geglaubt werden. Je weniger vorausgesetzt wird, desto besser sind die Chancen, daß die Berechenbarkeit davon einsichtig ist und den Erfahrungen entspricht. Aus diesem Grunde sind insbesondere auch die natürlichen Zahlen als einziger Datentyp gewählt worden, weil Menschen seit Jahrtausenden mit ihnen rechnen. Anschaulich wird erlaubt, ein leeres Blatt Papier als 0 zu nehmen, für jedes *succ* einen Strich zu machen, für jedes *pred* einen Strich auszuradieren und von zwei Blättern festzustellen, ob sie

gleich viele Striche enthalten oder nicht. Die Annahmen sind also prozeßhaft und von einfacher Art. Ist eigentlich die Unterstellung der Berechenbarkeit bei Zuweisung, Wiederholung und Reihung genauso selbstverständlich?
2. So klein, wie es aussieht, ist PASCALchen gar nicht. Da jedes *while*-Programm selbst eine Anweisung ist, kann es in anderen Programmen weiterverwendet werden. Gibt man *while*-Programmen Bezeichnungen und erlaubt diese statt der Anweisungen weiterzuverwenden, läßt sich mit diesem Mechanismus von *Makroanweisungen* sogar sehr strukturiert programmieren. Viele vertraute Operationen, Programmkonstrukte und Abfragen erweisen sich als Makroanweisungen, d.h. sie lassen sich als *while*-Programme realisieren, und können deshalb wie in PASCAL und ähnlichen Sprachen verwendet werden. Dazu gehören die Zuweisungsformen $X := Y$, $X := n$ (Konstante), $X := Y + Z$, $X := Y - Z$, $X := Y \times Z$, $X := Y \operatorname{div} Z$, $X := Y \bmod Z$, $X := 2^Y, \ldots$; die Wiederholungsanweisungen der Form *while B do S*, wobei B ein Boolescher Ausdruck ist, der aus Variablen, Konstanten, $=$, $\neq$, $<$, *and*, *or*, *not* aufgebaut ist, und die Kontrollstrukturen *if-then*, *if-then-else*, *repeat-until*.

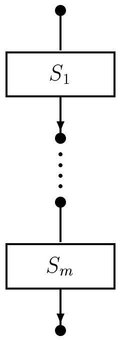
   Näheres dazu finden die Leserinnen und Leser im Abschnitt 2.2 des Buches von Kfoury, Moll, Arbib: A Programming Approach to Computability, Springer, New York/Heidelberg/Berlin 1982. Dieses Buch ist auch sonst für das Thema Berechenbarkeit zu empfehlen.
3. In Bemerkung 1 wird begründet, daß das begrenzte Repertoire von PASCALchen die Gefahr mindert, mit den gewählten Beschreibungshilfsmitteln über das Berechenbare hinauszuschießen. In Bemerkung 2 wird erläutert, daß das einzige, was gegenüber Sprachen wie PASCAL wirklich fehlt, das Konzept der Rekursion ist. Damit ist auch die Gefahr klein, daß PASCALchen hinter den Möglichkeiten des Berechenbaren zurückbleibt. Denn eine Programmiersprache wie FORTRAN kennt auch keine Rekursion; trotzdem sind keine Klagen bekannt, daß damit bestimmte algorithmische Probleme nicht lösbar seien. Tatsächlich gilt, daß Rekursion auf Iteration zurückgeführt werden kann, wie in nahezu jedem Buch über Berechenbarkeit nachlesbar ist (z.B. Abschnitt 5.2. von Kfoury, Moll, Arbib).
4. Um die Semantik von Programmen anzugeben, wird ausgenutzt, daß jede Anweisung durch ein Flußdiagramm dargestellt werden kann:
   (a) Zuweisung



(a) Wiederholung



(c) Reihung



Dabei werden die drei Zuweisungen und der Test *Instruktionen* genannt. Die Definition ist rekursiv. Die mit $S, S_1, \ldots, S_m$ bezeichneten Kästen müssen solange durch eine der fünf Diagramme ersetzt werden, bis nur noch Instruktionen vorkommen. Jedes Flußdiagramm hat genau einen Eingang und einen Ausgang. Die fünf Diagramme sind genau nach den Syntaxregeln gebildet. Darüber hinaus läßt sich zeigen, daß jedes *while*-Programm genau ein Flußdiagramm besitzt.

Mit Hilfe der Flußdiagramme soll nun geklärt werden, was *while*-Programme berechnen. Das Abarbeiten eines Programms mit bestimmten Anfangswerten für die Variablen entspricht einem Durchlauf durch das zugehörige Flußdiagramm. Wird dabei eine Zuweisung überquert, ändert sich der Wert der betroffenen Variablen entsprechend. Wird ein Test erreicht, bleiben alle Werte unverändert. Aber anhand der aktuellen Werte entscheidet sich, wie weitergegangen wird (in Richtung *TRUE* oder *FALSE*). Die jeweiligen Werte der Variablen während der Auswertung werden im Zustandsvektor festgehalten. Um dessen Definition zu vereinfachen, wird folgende Konvention getroffen.

6.3 Voraussetzung

Mit jeder Anweisung wird eine Zahl $k \geq 1$ spezifiziert, derart daß die Zahl der in der Anweisung vorkommenden Variablen k nicht übersteigt. Es kann angenommen werden, daß eine solche Anweisung höchstens die Variablen $X1, X2, \ldots, Xk$ verwendet. Sie wird dann eine *k-variable Anweisung* genannt.

Nutzt man die Reihenfolge der Variablen aus, können ihre Werte als k Zahlen $x_1, \ldots, x_k$ notiert werden, wobei x_i der Wert von Xi ist.

6.4 Definition (Berechnungszustand)

Ein *Berechnungszustand* eines k-variablen *while*-Programms ist ein k-dimensionaler Vektor über den natürlichen Zahlen

$$a = (x_1, \ldots, x_k) \in \mathbb{N}^k,$$

der auch *Zustand* oder *Zustandsvektor* genannt wird.

Gezielte Veränderungen der Berechnungszustände während eines Durchlaufs durch das Flußdiagramm ergeben eine Berechnung eines Programms.

6.5 Definition (Berechnung)

Eine *Berechnung* durch ein k-variables *while*-Programm ist eine (möglicherweise unendliche) Sequenz der Form

$$a_0 A_1 a_1 A_2 a_2 \cdots a_{i-1} A_{i-1} a_i A_{i+1} a_{i+1} \cdots,$$

wobei die a_i Berechnungszustände und die A_i Instruktionen mit folgenden Eigenschaften sind:

(a) Es gibt einen Weg durch das zugehörige Flußdiagramm, der beim Eingang beginnt und genau die Instruktionen $A_1, A_2, \ldots, A_i, \ldots$ in dieser Reihenfolge durchläuft.

(b) a_0 ist frei wählbar.

(c) Für alle $i \geq 1$ ergibt sich a_i aus a_{i-1} und A_i nach folgenden Regeln:

- Ist A_i ein Test, dann $a_i = a_{i-1}$.
- Ist A_i eine Zuweisung der Form $Xu := g(Xv)$, wobei $g(Xv)$ für $succ(Xv)$ oder $pred(Xv)$ oder 0 steht und $u, v \in \{1, \ldots k\}$, ist ferner $a_{i-1} = (x_1, \ldots, x_k)$, dann gilt:

$$a_i = (x_1, \ldots, x_{u-1}, g(x_v), x_{u+1}, \ldots, x_k).$$

Ist A_i die letzte Instruktion der Berechnung, so erreicht man hinter A_i den Ausgang. Ansonsten ist A_{i+1} die nächste Instruktion hinter A_i.

(d) Ist A_i ein Test der Form $Xu \neq Xv$, $a_{i-1} = (x_1, \ldots, x_k)$ sowie $x_u \neq x_v$, dann ist A_{i+1} die erste Instruktion in der Anweisung der zugehörigen Wiederholung. Ist jedoch $x_u = x_v$, muß man dem *FALSE*-Zweig folgen. Ist A_i die letzte Instruktion der Berechnung, erreicht man den Ausgang. Ansonsten ist A_{i+1} die nächste Instruktion nach dieser Wiederholung.

(e) Ist die Sequenz endlich, endet sie mit einem Berechnungszustand und hat deshalb die Form:

$$a_0 A_1 a_1 \cdots a_{n-1} A_n a_n \qquad (n \geq 0).$$

Dabei ist A_n die letzte Instruktion vor dem Ausgang des Flußdiagramms. Ist A_n ein Test $Xu \neq Xv$, muß ferner $x_u = x_v$ gelten für $a_{n-1} = a_n = (x_1, \ldots, x_k)$.

(f) Im Falle von (*e*) wird n die *Länge* der Berechnung genannt.

Unmittelbar aus der Konstruktion ergibt sich das folgende Ergebnis.

6.6 Beobachtung

Zu jedem k-variablen *while*-Programm und jedem Berechnungszustand a_0 gibt es genau eine Berechnung mit a_0 als Anfang.

6.7 Bemerkungen

1. Für den Umgang mit Berechnungen sind einige Sprechweisen hilfreich:
   (a) Fängt eine Berechnung mit a_0 an, wird a_0 *Eingabe* genannt.
   (b) Für eine endliche Berechnung $a_0 A_1 a_1 \cdots A_n a_n$ sagt man, daß sie (in n Schritten) *terminiert*; a_n wird dann *Ausgabe* genannt.
   (c) Entsprechend *terminiert* eine unendliche Berechnung *nicht*; ihre Ausgabe ist *undefiniert*.
2. Die hier beschriebene Auswertung von *while*-Programmen ist ein weiteres typisches Beispiel für eine operationelle Semantik, wobei anhand des Programmtextes eine Eingabe schrittweise zur Ausgabe umgeformt wird.
3. In der Länge von Berechnungen besitzen Algorithmen, die als *while*-Programme geschrieben sind, ein einheitliches Zählmaß für die Aufwandsbestimmung. Die Länge sagt, wie oft die Aktionen 0, *succ*, *pred*, $\neq$ ausgeführt werden, d.h. es werden elementare Operationen gezählt.

6.8 Beobachtung

Beobachtung 6 legt nahe, die Semantik eines Programms als partielle Funktion auf den natürlichen Zahlen zu definieren, denn Berechnungen sind pro Eingabe eindeutig, liefern jedoch nur bei Termination Ergebnisse. Aus technischen Gründen wird dabei von den verfügbaren k Variablenwerten in modifizierter Form Gebrauch gemacht. Während als Funktionswert nur der Wert der Variablen $X1$ betrachtet wird, wird die Funktion auf j Argumente angewendet. Ist $j < k$, bekommen die ersten j Variablen die Argumente als Eingangswerte, während die restlichen durch Nullen aufgefüllt werden. Ist $j \geq k$, fallen die eventuell überschüssigen Argumente bei der Berechnung weg. Durch diesen "Trick" wird die Zahl der Argumente unabhängig von der Zahl der Variablen.

6.9 Definition (Semantikfunktion)

Sei P ein k-variables *while*-Programm und $j \in \mathbb{N}$. Dann ist die j-stellige *Semantikfunktion* von P

$$SEM_P : \mathbb{N}^j \to \mathbb{N}$$

für die Argumente $(x_1, \ldots, x_j) \in \mathbb{N}^j$ nach folgenden Regeln definiert:

(a) Aus den Argumenten wird eine Eingabe $a \in \mathbb{N}^k$ hergestellt, wobei $a = (x_1, \ldots, x_k)$ ist für $j \geq k$ und $a = (x_1, \ldots, x_j, 0, \ldots, 0)$ mit $k - j$ Nullen für $j < k$.

(b) P wird mit Eingabe a berechnet.

(c) Terminiert die Berechnung mit der Ausgabe $(y_1, \ldots, y_k)$, so ist $SEM_P(x_1, \ldots, x_j) = y_1$.

(d) Terminiert sie nicht, ist $SEM_P(x_1, \ldots, x_j)$ undefiniert.

6.10 Bemerkungen

1. Beachte, daß die Semantikfunktion wegen (d) im allgemeinen nicht immer definiert ist. Sie ist genau dann eine totale Funktion, wenn jede Berechnung terminiert.
2. Jedes Programm kann wegen der Wahlfreiheit von j unendlich viele Funktionen berechnen, die sich allerdings nur wenig voneinander unterscheiden.
3. Soll das j betont werden, wird $SEM_P^{(j)}$ statt SEM_P geschrieben.
4. Ist $SEM_P(x_1, \ldots, x_j)$ undefiniert, wird im folgenden manchmal die Bezeichnung $SEM_P(x_1, \ldots, x_j) = \perp$ dafür verwendet.

Nach diesen Vorbereitungen ist es nun möglich, den zentralen Begriff der Berechenbarkeitstheorie einzuführen. Berechenbare Funktionen präzisieren durch Computer lösbare Aufgaben.

6.11 Definition (Berechenbare Funktion)

Eine partielle Funktion $f : \mathbb{N}^j \to \mathbb{N}$ heißt *berechenbar*, wenn ein *while*-Programm existiert mit

$$f = SEM_P^{(j)}.$$

Mit Hilfe dieser exakten Version von Berechenbarkeit läßt sich nun auch die CHURCHSCHE THESE für die Zwecke dieser Lehrveranstaltung neu formulieren. Der folgende Abschnitt zeigt, daß sie sehr bequem ist, auch wenn man ihr nicht unbedingt glaubt. Es muß allerdings darauf hingewiesen werden, daß die Berechenbarkeitstheorie und mathematische Logik bisher nur Belege für ihre Richtigkeit gefunden hat und daß sie auch den Erfahrungen der meisten Informatikerinnen und Informatiker entspricht.

6.12 Churchsche These (2. Version)

Jede partielle Funktion $f : \mathbb{N}^j \to \mathbb{N}$, die durch irgendeinen Mechanismus oder auf Grund irgendeiner Überlegung algorithmisch berechnet werden kann, ist bereits berechenbar (durch ein *while*-Programm).

7 Programme sind aufzählbar – doch ihr (Ver-)Halten macht Kummer

Zu den technologischen Durchbrüchen der Computertechnik in den 40er Jahren zählt das von Neumann entdeckte Prinzip der Programmspeicherung, durch das Programme jederzeit abrufbar und wie Daten bearbeitbar werden. Heutzutage ist allen Informatikerinnen und Informatikern angesichts von Editoren, Compilern u.ä. vertraut, daß Programme Programme verarbeiten können. Theoretisch wurde die Tatsache, daß Algorithmen Daten von Algorithmen sein können, noch einige Jahre früher von Gödel erkannt und hat in Logik und Berechenbarkeitstheorie zu fundamentalen Erkenntnissen geführt. Um einiges davon vorstellen zu können, ist noch eine Schwierigkeit zu überwinden: Der Algorithmusbegriff des vorigen Kapitels arbeitet auf natürlichen Zahlen; damit *while*-Programme Daten werden und umgekehrt, müssen natürliche Zahlen als Programme und Programme als natürliche Zahlen aufgefaßt werden können. Zu diesem Zweck erhält jedes *while*-Programm einen Index. Außerdem wird ein algorithmisches Verfahren angegeben, wie aus einer natürlichen Zahl ein Programm entsteht, dessen Index sie ist. Auf diese Weise werden *while*-Programme "effektiv" aufgezählt.

7.1 Bestimmung des Indexes eines Programms

1. Nach Punkt 6.1.1 in Verbindung mit der Konvention in Punkt 6.3 besteht der Zeichensatz A von PASCALchen aus 22 Zeichen.
2. Für jedes Zeichen $a \in A$ wird nun eine eindeutige 6-Bit-Darstellung $code(a) = b_1 \cdots b_6$ mit $b_1 = 1$ und $b_2, \ldots, b_6 \in \{0,1\}$ ausgewählt. Es wird also eine injektive Abbildung $code\colon A \to \{0,1\}^*$ fixiert. Eine solche Wahl ist möglich, weil es 32 verschiedene 6-Bit-Muster mit 1 am Anfang gibt.
3. *code* läßt sich auf A^* fortsetzen, indem jedes Zeichen eines Wortes von links nach rechts durch seine 6-Bit-Darstellung ersetzt wird. Das ergibt eine Abbildung $code^*\colon A^* \to \{0,1\}^*$, die definiert ist durch
   (i) $code^*(\lambda) = \lambda$
   (ii) $code^*(av) = code(a)code^*(v)$ für $a \in A, v \in A^*$.
4. Jedes *while*-Programm ist ein Wort über A und besitzt deshalb ein Bitmuster. Jedes Bitmuster läßt sich als Binärdarstellung einer natürlichen Zahl auffassen. Das erlaubt folgende Definition.
5. Der *Index* eines *while*-Programms P ist die natürliche Zahl, deren Binärdarstellung $code^*(P)$ ist.
6. Diese Verwandlung syntaktischer Objekte in Zahlen wird *Arithmetisierung* oder *Gödelnumerierung* genannt. Sie macht es prinzipiell möglich, mit Programmen zu rechnen.
7. Es ist äußerst wichtig, daß verschiedene Programme verschiedene Indizes erhalten. Das jedoch folgt unmittelbar aus folgendem Lemma.

7.2 Lemma

Die Gödelnumerierung $code^*$ ist injektiv.

Beweis: Betrachte die Abbildung $decode\colon \{0,1\}^* \to A^*$, die definiert ist für alle $B \in \{0,1\}^*$ und $b_1, \ldots, b_6 \in \{0,1\}$ durch

(i) $decode(B) = \lambda$ für alle B kürzer als 6,

(ii) $decode(b_1 \cdots b_6 B) = \begin{cases} a\,decode(B) & \text{wenn } code(a) = b_1 \cdots b_6 \\ \lambda & \text{sonst.} \end{cases}$

Dann gilt $decode(code^*(w)) = w$ für alle $w \in A^*$, wie sich durch vollständige Induktion beweisen läßt.

IA: $decode(code^*(\lambda)) = decode(\lambda) = \lambda$.

IV: Die Aussage gelte für $v \in A^*$.

IS: $decode(code^*(av)) = decode(code(a)code^*(v)) = a\,decode(code^*(v)) = av$.

Daraus ergibt sich die behauptete Injektivität:
$code^*(u) = code^*(v)$ impliziert $decode(code^*(u)) = decode(code^*(v))$ also $u = v$. □

Umgekehrt kann jeder Zahl algorithmisch ein Programm zugeordnet werden, dessen Index sie ist.

7.3 Bestimmung des Programms eines Indexes

1. Betrachte zu jeder natürlichen Zahl n ihre Binärdarstellung $B(n)$, und wende darauf die Abbildung *decode* aus dem vorangegangenen Beweis an. Ist $decode(B(n))$ ein *while*-Programm, so wird es als von n aufgezählt betrachtet:

   $$\text{AUFZÄHLUNG}(n) = decode(B(n)).$$

   Andernfalls wird n ein Programm zugeordnet, das niemals hält:

   $$\begin{aligned} \text{AUFZÄHLUNG}(n) = \; & begin \\ & \quad X1 := 0;\ \ X2 := 1; \\ & \quad while\ \ X1 \neq X2\ \ do\ \ X1 := X1 \\ & end. \end{aligned}$$

   Damit ist insgesamt eine Abbildung AUFZÄHLUNG definiert, die jeder Zahl ein Programm zuordnet.
2. **Beobachtung**: Durch AUFZÄHLUNG wird insbesondere der Index n eines Programms P auf P abgebildet.

   Beweis: Nach Definition des Indexes n eines *while*-Programms P gilt: $B(n) = code^*(P)$. Daraus folgt nach Lemma 2:

   $$decode(B(n)) = decode(code^*(P)) = P.$$

   Nach obiger Definition von AUFZÄHLUNG ergibt sich wie behauptet:
   $\text{AUFZÄHLUNG}(n) = decode(B(n)) = P$. □
3. Im folgenden wird eine natürliche Zahl n *Index* des *while*-Programms P genannt, wenn $\text{AUFZÄHLUNG}(n) = P$. Wegen der obigen Beobachtung umfaßt diese Definition die bisherigen Indizes. Statt $\text{AUFZÄHLUNG}(n)$ wird oft kurz P_n geschrieben. P_n bezeichnet also das Programm mit dem Index n.

Jede natürliche Zahl ist nun Index, so daß *while*-Programme mühelos Indizes manipulieren können. Von AUFZÄHLUNG erfährt man dann, wie sich das auf die indizierten Programme auswirkt. Beachte, daß dadurch alle Programme erfaßt werden, weil nach Punkt 1 jedes Programm einen Index hat.

4. Da jedes Programm P für jede Argumentzahl j eine berechenbare Funktion $SEM_P^{(j)}$ bestimmt, kann jede Zahl auch als Index einer berechenbaren Funktion aufgefaßt werden:

$$SEM_n^{(j)} := SEM_{P_n}^{(j)} \quad \text{für alle } n \in \mathbb{N}.$$

Statt $SEM_n^{(1)}$ wird kürzer SEM_n geschrieben, denn dieser Fall kommt am häufigsten vor.

Beim "Aufzählen" der natürlichen Zahlen werden also gleichzeitig alle *while*-Programme und damit insbesondere alle berechenbaren Funktionen mit einem Argument "mitaufgezählt".

0	1	2	...	n	...	$\mathbb{N}$
$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$		$\downarrow$ AUFZÄHLUNG
P_0	P_1	P_2	...	P_n	...	*while*−Programme
$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$		$\downarrow$ *SEM*
SEM_0	SEM_1	SEM_2	...	SEM_n	...	berechenbare Funktionen

Für jede andere Argumentzahl kann die Aufzählung entsprechend durchgeführt werden, wird aber im folgenden nicht gebraucht.

5. Beachte, daß die Indizes nicht für das praktische Rechnen mit Programmen geeignet sind, weil sie sehr groß ausfallen. Ein Programm aus nur 10 Zeichen beispielsweise hat bereits einen kleinsten Index zwischen 2^{59} und 2^{60}.

7.4 Bemerkung (abzählbar, aufzählbar, effektiv aufzählbar)

1. Man nennt eine Menge M von Objekten *abzählbar* oder *aufzählbar*, wenn es eine surjektive Abbildung $num: \mathbb{N} \to M$ gibt. M heißt *effektiv aufzählbar*, wenn diese Numerierung durch einen Algorithmus vorgenommen wird.
2. Da die Binärdarstellung einer Zahl berechnet werden kann, da die Werte von *decode* algorithmisch bestimmt sind, da es schließlich einen Algorithmus gibt, der entscheidet, ob ein Text über A ein *while*-Programm ist oder nicht, erweist sich AUFZÄHLUNG als effektiv. Graphisch sei das ausgedrückt durch:

$$\xrightarrow{\;n\;} \boxed{\text{AUFZÄHLUNG}} \xrightarrow{\;P_n\;}$$

3. Daß die effektive Aufzählung etwas Besonderes ist, zeigt folgende Überlegung:

   Abzählbar ist insbesondere jede Teilmenge der natürlichen Zahlen, von denen es überabzählbar viele gibt. Effektiv aufzählbar sind jedoch nur abzählbar viele Teilmengen, weil es nach Abschnitt 3 nur abzählbar viele Algorithmen gibt. Die dabei verwendete Überabzählbarkeit der Teilmengen von $\mathbb{N}$ ergibt sich analog zur folgenden Überlegung. Die Konstruktionen in Abschnitt 3 zeigen, daß es nur abzählbar viele berechenbare Funktionen gibt. Mit einem bekannten Argument (dem Diagonalisierungsverfahren von Cantor, mit dem dieser gezeigt hat, daß es überabzählbar viele reelle Zahlen gibt) stellt sich heraus, daß die (partiellen) Funktionen auf $\mathbb{N}$ nicht abzählbar sind. Es muß also Funktionen geben, die nicht berechenbar sind.

7.5 Theorem (Existenz nicht-berechenbarer Funktionen)

Es gibt eine Funktion $f: \mathbb{N} \to \mathbb{N}$, die nicht berechenbar ist.

Beweis: Angenommen, jede Funktion $f: \mathbb{N} \to \mathbb{N}$ wäre berechenbar. Dann gäbe es zu f einen Index i mit $f = SEM_i$. Die (totalen) Funktionen lassen sich dann folgendermaßen aufzählen: f_0 ist die erste Funktion in der Aufzählung $SEM_0, SEM_1, SEM_2, \ldots$. Sind $f_0, \ldots, f_k$ bereits bestimmt, dann ist f_{k+1} die nächste Funktion hinter f_k in $SEM_0, SEM_1, SEM_2, \ldots$. Betrachte dann die Funktion $plus: \mathbb{N} \to \mathbb{N}$, die definiert ist durch

$$plus(n) = f_n(n) + 1.$$

Faßt man die Argumente als Spaltennummern und die Indizes der Funktionen als Zeilennummern auf und trägt in die Felder (m, n) immer gerade $f_m(n)$ ein, dann entsteht *plus* dadurch, daß jede Zahl in der Diagonalen dieser unendlichen Matrix um 1 hochgezählt wird.

Nach Annahme existiert ein Index l mit $plus = f_l$. Das aber führt zu einem Widerspruch, denn *plus* erweist sich als verschieden zu f_l:

$$plus(l) = f_l(l) + 1 \neq f_l(l).$$

Da die Annahme falsch ist, muß die Behauptung richtig sein. □

Die allgemeine Überlegung zur Nicht-Berechenbarkeit hat den Nachteil, daß sich eine ziemlich uninteressante Funktion als nicht-berechenbar erweist. Es ist nicht übermäßig wahrscheinlich, daß irgendjemand auf die Idee gekommen wäre, gerade diese Funktion zu berechnen. Anders verhält es sich mit dem folgenden Problem, bei dem versucht wird, eine äußerst wichtige Eigenschaft von Programmen zu ermitteln, nämlich ob bestimmte

Berechnungen terminieren oder nicht. In dem hier betrachteten Fall des "Halteproblems" wird jedes Programm auf den eigenen Index angewendet, weshalb es in der Literatur auch häufig "Selbstanwendungsproblem" genannt wird. Es stellt sich heraus, daß es unmöglich ist, zu berechnen, ob ein Programm bei Eingabe seines Indexes hält oder nicht.

7.6 Theorem (Unlösbarkeit des Halteproblems)

Die (totale) Funktion $\mathit{HALTEPROBLEM}\colon \mathbb{N} \to \mathbb{N}$, die definiert ist für alle $i \in \mathbb{N}$ durch

$$\mathit{HALTEPROBLEM}(i) = \begin{cases} 1 & \text{wenn } \mathit{SEM}_i(i) \text{ definiert ist} \\ 0 & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Beweis: Der Beweis wird durch Widerspruch geführt.
Angenommen, die Funktion $\mathit{HALTEPROBLEM}$ wäre durch das *while*-Programm HALT berechenbar. Dann ist auch die partielle Funktion $\mathit{konfus}\colon \mathbb{N} \to \mathbb{N}$ mit

$$\mathit{konfus}(i) = \begin{cases} 1 & \text{für } \mathit{HALTEPROBLEM}(i) = 0 \\ \perp & \text{sonst} \end{cases}$$

berechenbar durch das Programm:

```
begin
  HALT;
  while X1 ≠ 0 do X1 := X1;
  X1 := 1
end
```

Es existiert also ein Index j mit $\mathit{konfus} = \mathit{SEM}_j$. Betrachte nun die Anwendung von SEM_j auf j.

1. Fall: $\mathit{SEM}_j(j)$ ist definiert.
Dann ist $\mathit{HALTEPROBLEM}(j) = 1$ und damit $\mathit{konfus}(j) = \mathit{SEM}_j(j)$ undefiniert.

2. Fall: $\mathit{SEM}_j(j)$ ist undefiniert.
Dann ist $\mathit{HALTEPROBLEM}(j) = 0$ und damit $\mathit{konfus}(j) = \mathit{SEM}_j(j) = 1$.

Mehr Fälle gibt es nicht, und $\mathit{SEM}_j(j)$ kann nicht gleichzeitig definiert und undefiniert sein. Also muß schon die Annahme falsch und damit die Behauptung des Theorems richtig sein. □

Wem es immer noch "exotisch" vorkommt, daß man Programme auf sich selbst bzw. ihre Indizes anwendet, den mag die Unlösbarkeit des allgemeinen Halteproblems mehr überzeugen, bei dem es darum geht, ob ein Programm für irgendeine Eingabe hält oder nicht. Das ist auch ein erstes Beispiel für das hilfreiche Reduktionsprinzip, bei dem die Nicht-Berechenbarkeit eines Problems dadurch gezeigt wird, daß das Problem auf eines zurückgeführt wird, dessen Nicht-Berechenbarkeit bereits bekannt ist .

7.7 Corollar (Unlösbarkeit des allgemeinen Halteproblems)

Die Funktion $\mathit{HALTEPROBLEM2}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

$$\mathit{HALTEPROBLEM2}(i, j) = \begin{cases} 1 & \text{wenn } \mathit{SEM}_i(j) \text{ definiert ist} \\ 0 & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Beweis: Angenommen, $\mathit{HALTEPROBLEM2}$ wäre durch das *while*-Programm $\mathit{HALT2}$ berechenbar. Dann wäre auch $\mathit{HALTEPROBLEM}$ berechenbar durch das Programm

```
begin
  X2 := X1;
  HALT2
end
```

im Widerspruch zum obigen Theorem. □

7.8 Bemerkung

Für die Nicht-Berechenbarkeit von Problemen, die auch als Unlösbarkeit angesprochen wird, kann man häufig auch die Bezeichnung *Nichtentscheidbarkeit* finden, wenn das Problem eine ja/nein-Frage ist, d.h. wenn die zugehörige Funktion für jede Eingabe einen von zwei Werten (in der Regel 0 und 1) liefert.

8 Ein Programm rechnet für alle

Die Nicht-Berechenbarkeit des Halteproblems wird verursacht durch den Wunsch, auch erfahren zu wollen, wenn Programme nicht halten. Verzichtet man darauf und begnügt sich damit, die Werte von Programmberechnungen zu bekommen, soweit sie existieren, so erweist sich diese abgeschwächte Aufgabe als berechenbar. Um das einzusehen, muß ein Interpreter konstruiert werden, der bei Eingabe eines Programms P und der Argumente $(x_1, \ldots, x_j)$ der von P berechneten Funktion SEM_P den Wert $\mathit{SEM}_P(x_1, \ldots, x_j)$ liefert, vorausgesetzt P terminiert für diese Eingabe:

Ein solcher Algorithmus $INTERPRETER_0$ ergibt sich gerade aus den im 6. Kapitel eingeführten Konzepten und Konstruktionen:

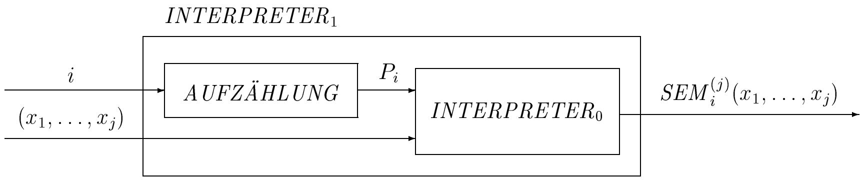
(1) Wandle das k-variable Programm P in sein Flußdiagramm um.

(2) Wandle $(x_1, \ldots, x_j)$ in eine Eingabe a um.

(3) Berechne P für Eingabe a.

(4) Gib bei Termination den Wert von $X1$ aus.

Zwei Klippen sind noch zu umschiffen, bevor dieser Vorgang sich als berechenbar herausstellen kann. Der Interpreter verarbeitet Programme, *while*-Programme und berechenbare Funktionen nur natürliche Zahlen. Diese Diskrepanz läßt sich beseitigen, wenn der Interpreter mit der Aufzählung des 7. Kapitels gekoppelt wird, da die Aufzählung ja gerade dazu dient, Zahlen stellvertretend für Programme als Eingabe zu akzeptieren:



Diese neue Interpreterfunktion hat jetzt nur noch natürlichzahlige Argumente, und zwar genau ein Argument mehr als die Funktionen, die berechnet werden. Es gibt nach den bisherigen Überlegungen einen Algorithmus, der die Werte dieser Funktion ermittelt; aber gibt es auch ein *while*-Programm, das das leistet? Die bejahende Antwort ergibt sich aus der CHURCHSCHEN THESE. Das ist ein interessantes Beispiel für ihren vorteilhaften Gebrauch: Um Berechenbarkeit sicherzustellen, muß nicht immer ein *while*-Programm geschrieben werden; irgendein Algorithmus tut es auch. Beachte, daß es gerade auch für die Konstruktionen des 6. und 7. Kapitels reichlich mühsam wäre, simulierende *while*-Programme zu entwickeln.

Zusammengefaßt hat sich folgende Erkenntnis ergeben:

8.1 Theorem (universelle Funktion)

Für alle $j \in \mathbb{N}$ ist die partielle Funktion $interpret\colon \mathbb{N}^{j+1} \to \mathbb{N}$, die definiert ist durch

$$interpret(i, x_1, \ldots, x_j) = SEM_i^{(j)}(x_1, \ldots, x_j),$$

berechenbar.

8.2 Bemerkung

Da *interpret eine* Funktion ist, die *alle* berechenbaren Funktionen mit j Argumenten ausrechnet, wird sie *universelle Funktion* genannt. So verblüffend dieser Sachverhalt klingen mag, eigentlich passiert etwas Simples. Die Indizes der berechenbaren Funktionen werden zu einem zusätzlichen Argument. Und die Berechnung eines *while*-Programms ist ein algorithmischer Vorgang.

Zum Abschluß dieses Kapitels wird an fünf Beispielen die Grenzlinie zwischen Berechenbarem und Unberechenbarem kommentiert, die universelle Funktionen vom Halteproblem trennt. Vier der Beispiele nutzen dabei die Existenz universeller Funktionen in spezieller Form:

Es gibt ein k-variables *while*-Programm *INTERPRETER* mit

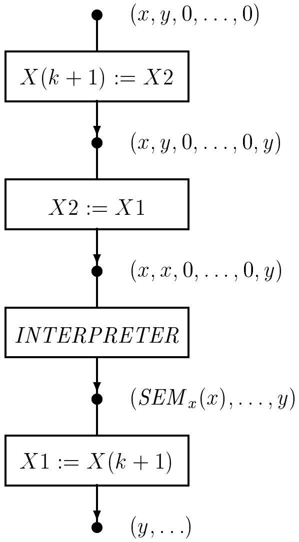
$$SEM_{INTERPRETER}(i, x) = SEM_i(x) \qquad \text{für alle } i, x \in \mathbb{N}.$$

8.3 Beispiele

1. Abhängig von den Berechnungen des Interpreters, lassen sich auch andere Aufgaben erledigen. Ein einfaches Beispiel dafür ist die partielle Funktion $halt_1\colon \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_1(x, y) = \begin{cases} y & \text{wenn } SEM_x(x) \text{ definiert ist} \\ \perp & \text{sonst.} \end{cases}$$

Die Funktion $halt_1$ erweist sich als berechenbar durch das folgende $(k+1)$-variable *while*-Programm:



Rechts sind die wichtigsten Änderungen der Zustandsvektoren verzeichnet, wobei der vorletzte Zustand nur entsteht, wenn der Interpreter hält. Die Berechnungen zeigen, daß die partielle Funktion $halt_1$ berechnet wird.

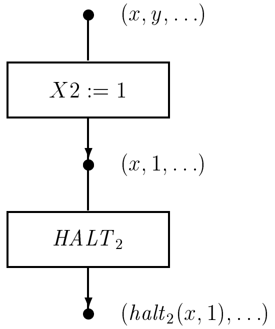
2. Die Grenze zur Unberechenbarkeit wird wieder gerade überschritten, wenn über $halt_1$ hinaus auch Werte produziert werden sollen, wo der Interpreter nicht hält.

   Deshalb ist die Funktion $halt_2 : \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_2(x,y) = \begin{cases} y & \text{wenn } SEM_x(x) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

   nicht berechenbar.

   Angenommen, $halt_2$ wird durch das *while*-Programm $HALT_2$ berechnet. Betrachte dann das Programm



   Die rechts aufgezeichneten Berechnungen zeigen (für $y = 0$), daß dieses Programm P die einstellige Funktion $SEM_P : \mathbb{N} \to \mathbb{N}$ berechnet mit

$$SEM_P(x) = halt_2(x,1) = \begin{cases} 1 & \text{wenn } SEM_x(x) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

   Das ist das Halteproblem, das gar nicht lösbar ist; also muß die Annahme falsch sein.

   Am Unterschied von $halt_1$ und $halt_2$ wird noch einmal deutlich, wodurch Unberechenbarkeit verursacht wird. Solange der Interpreter läuft, kann nicht gesagt werden, ob die Berechnung nach weiteren Schritten abbricht oder nie zum Stehen kommt. Denn die Schrittzahl von Berechnungen kann jede endliche Schranke überschreiten. Ob der Wert 0 sein muß, läßt sich beim Berechnen nicht erfahren.

3. Die Verhältnisse können aber auch wesentlich undurchsichtiger sein. Die (partielle) Funktion $halt_3 \colon \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_3(x,y) = \begin{cases} \mu(y) & \text{wenn } SEM_x(x) \text{ definiert ist} \\ \nu(y) & \text{sonst,} \end{cases}$$

   wobei $\mu, \nu : \mathbb{N} \to \mathbb{N}$ beide berechenbare Funktionen sind, hat große Ähnlichkeit mit $halt_2$. Und doch ist sie unter bestimmten Umständen berechenbar, wenn nämlich $\nu \leq \mu$ ist, d.h. wann immer $\nu(y)$ definiert ist, so ist auch $\mu(y)$ definiert, und in diesem Fall gilt: $\mu(y) = \nu(y)$.

   Ein Beispiel für solche Funktionen wäre etwa gegeben durch

$$\mu(y) = \begin{cases} y^2 \text{ für } y < 200 \\ \perp \text{ sonst} \end{cases} \quad \text{und} \quad \nu(y) = \begin{cases} y^2 \text{ für } y < 100 \\ \perp \text{ sonst.} \end{cases}$$

   Die Berechenbarkeit von $halt_3$ läßt sich folgendermaßen einsehen, vorausgesetzt $\nu \leq \mu$.

   (a) Berechne gleichzeitig $\nu(y)$ und $SEM_x(x)$.
   (b) Terminiert zuerst $SEM_x(x)$, tritt der erste Fall in Kraft. Beende deshalb die Berechnung von $\nu(y)$, und beginne die von $\mu(y)$.
   (c) Terminiert zuerst die Berechnung von $\nu(y)$. Dann ist nach Voraussetzung auch $\mu(y)$ definiert, und es gilt: $\mu(y) = \nu(y)$. Der Wert von $halt_3(x,y)$ ist damit gefunden, unabhängig davon, ob $SEM_x(x)$ terminiert.
   (d) Terminieren beide nicht, so ist auch $halt_3(x,y)$ undefiniert.

   Deshalb beschreiben die Punkte (a)-(c) einen Algorithmus, der $halt_3$ ausrechnet. Nach der CHURCHSCHEN THESE ist $halt_3$ also berechenbar.

4. Eine sehr praktische Möglichkeit, die Schwierigkeiten mit dem Halteproblem zu umgehen, besteht darin, nach bestimmter "Zeit" die Berechnung gezielt abzubrechen. Im Beispiel wird das erreicht, indem die Rechenschritte gezählt werden und bei Überschreiten einer vorgegebenen Schranke aufgehört wird.

   Die Funktion $halt_4 \colon \mathbb{N}^3 \to \mathbb{N}$ mit

$$halt_4(x,y,z) = \begin{cases} 1 & \text{wenn } P_x \text{ mit Eingabe } y \text{ nach spätestens } z \text{ Schritten hält} \\ 0 & \text{sonst} \end{cases}$$

   ist berechenbar.

   Denn ein Algorithmus für $halt_4$ kann wie folgt arbeiten:

   (a) Berechne P_x für Eingabe y.
   (b) Zähle die Schritte (Länge des Berechnungsweges im Flußdiagramm).
   (c) Gib 1 aus, wenn der Zähler bei $count \leq z$ stehen bleibt.
   (d) Sonst gib 0 aus.

Die Berechenbarkeit folgt wieder aus der CHURCHSCHEN THESE.

Diese Vorgehensweise wird bei vielen Problemen der Informatik verwendet. Ihre "Philosophie" ist, eine Frage als unbeantwortet zu betrachten, wenn die Antwort zu lange ausbleibt.

5. Das letzte Beispiel ist nicht auf den Interpreter bezogen, sondern soll zeigen, daß auch im Bereich des Berechenbaren sehr undurchsichtige Situationen entstehen können. Es rankt sich um die Dezimalentwicklung von $\pi = 3,14159265\ldots$. Gesucht werden Sequenzen aufeinanderfolgender 5en:

$$3,14\ldots\underbrace{55\ldots5}_{x-\mathrm{mal}}\ldots$$

Dabei spielen die vorausgehenden und nachfolgenden Ziffern keine Rolle (insbesondere dürfen auch sie 5en sein).

Aus diesen Elementen läßt sich eine Funktion $foggy\colon \mathbb{N} \to \mathbb{N}$ bilden mit

$$foggy(x) = \begin{cases} 1 & \text{wenn } \pi \text{ eine 5er-Sequenz der Länge } x \text{ enthält} \\ 0 & \text{sonst.} \end{cases}$$

Verglichen mit $halt_2$ könnte *foggy* für unberechenbar gehalten werden. Denn die Dezimalentwicklung von π läßt sich beliebig weit algorithmisch herstellen, wird jedoch nie fertig. So betrachtet, kann man nie wissen, ob der Wert 0 sein muß oder noch eine geeignete Sequenz beim weiteren Entwickeln der Dezimalstellen kommt.

Mit einer anderen Überlegung erweist sich *foggy* als berechenbar. Entweder gibt es in π 5er-Sequenzen jeder Länge. Dann ist $foggy(x) = 1$ für alle $x \in \mathbb{N}$ und als konstante Funktion berechenbar. Oder es existiert eine 5er-Sequenz mit maximaler Länge *max*. Dann gilt

$$\mathrm{foggy}(x) = \begin{cases} 1 & \text{für } x \leq max \\ 0 & \text{sonst.} \end{cases}$$

Für jede natürliche Zahl $max \in \mathbb{N}$ läßt sich ein *while*-Programm schreiben, das *foggy* berechnet (*if* $x \leq max$ *then* 1 *else* 0).

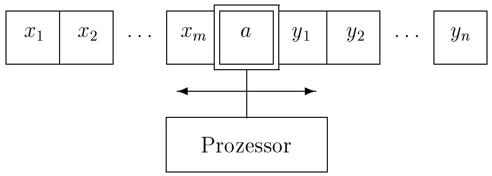
Es gibt für *foggy* damit unendlich viele Möglichkeiten; in jedem Fall aber ist die Berechenbarkeit sichergestellt. Der einzige Haken: Niemand weiß, welcher Fall zutrifft.

9 Turing-Maschinen

Das Konzept der Turing-Maschine wurde von Alan Turing in den 30er Jahren dieses Jahrhunderts eingeführt und stellt damit eines der ältesten Berechenbarkeitsmodelle dar. Die Idee war, den mechanischen Anteil des Rechnens mit Bleistift und Radiergummi auf Papier formal zu fassen.

9.1 Begriff und Arbeitsweise

Eine Turing-Maschine *TM* funktioniert so: Sie befindet sich zu jeder Zeit in einem Zustand, der aus einer vorgegebenen endlichen Menge S von Zuständen stammt, mit denen sich also endlich viele verschiedene Fälle darstellen lassen. Sie hat ein Arbeitsband, das aus einer linearen Kette von Zellen besteht, die von links nach rechts geordnet sind. Eine Zelle kann unbeschrieben oder mit einem Zeichen aus einem endlichen Alphabet A beschrieben sein. Zu jeder Zeit ist das Band endlich lang, kann aber unter bestimmten Umständen links und rechts durch unbeschriebene Zellen verlängert werden. Außerdem hat die Maschine einen Lese-Schreib-Kopf, der entweder am rechten Ende des Bandes oder auf einer der Zellen steht. Formal ist deshalb das Arbeitsband durch zwei Zeichenketten $u, v \in A'^*$ beschrieben, wobei u den Bandinhalt links vom Kopf und v den Bandinhalt rechts vom Kopf – einschließlich der aktuellen Zelle, falls der Kopf nicht ganz rechts steht – darstellt. A' umfaßt A und enthält außerdem ein Sonderzeichen $\Box$, das unbeschriebene Zellen repräsentiert. Zu Beginn befindet sich die Maschine in einem ausgezeichneten Anfangszustand, der Lese-Schreib-Kopf steht ganz links und alle Zellen sind beschrieben (d.h. $u = \lambda$ und $v \in A^*$). Die Maschine kann Arbeitsschritte vollziehen, die von ihrem "Programm" – der Zustandsüberführungsrelation d – abhängen. Diese ordnet jedem Zustand und jedem gelesenen Zeichen aus A' mögliche Folgezustände, mögliche zu schreibende Zeichen aus A' und mögliche Bewegungen des Kopfes zu. Als Bewegungen stehen zur Verfügung "*l*" für eine Zelle nach links, "*r*" für eine Zelle nach rechts und "*n*" für Nicht-Bewegen. Ein konkreter Schritt ändert dann den aktuellen Zustand sowie den Inhalt der Zelle unter dem Lese-Schreib-Kopf und führt eine Bewegung aus – und das alles gemäß der Zustandsüberführungsrelation. Solche Arbeitsschritte können beliebig wiederholt werden. Die Schrittfolge endet notwendigerweise, wenn die Zustandsüberführungsrelation keinen Folgezustand mehr vorsieht. Das tritt insbesondere ein, wenn der aktuelle Zustand ein Endzustand ist. Alles was in diesem Fall unter dem Kopf und rechts davon steht, wird als Ergebnis der Berechnung angesehen, wenn da keine Zelle unbeschrieben ist.



Diese Konzeption kann folgendermaßen formalisiert werden:

1. Eine *Turing-Maschine* ist ein System $TM = (S, A, d, s_0, F)$, wobei S eine endliche Menge von *Zuständen*, A ein endliches *Alphabet*, $s_0 \in S$ ein *Anfangszustand*, $F \subseteq S$ eine Menge von *Endzuständen* und d eine *Zustandsüberführungsrelation* ist, die jedem Zustand $s \in S$ und jedem Zeichen $a \in A' = A \cup \{\Box\}$ eine Teilmenge $d(s, a) \subseteq S \times A' \times$

$\{n, l, r\}$ zuordnet. Dabei ist $\Box$ ein Sonderzeichen (d.h. $\Box \notin A$), und n, l und r sind drei Steuerzeichen. Außerdem wird gefordert, daß $d(s', a)$ leer ist für alle $s' \in F$ und $a \in A'$.

2. TM ist *deterministisch*, falls $d(s, a)$ für jedes $s \in S$ und $a \in A'$ höchstens ein Element enthält.

3. Eine *Konfiguration* hat die Form usv mit $u, v \in A'^*$ und $s \in S$.

4. Eine *Anfangskonfiguration* hat die Form $\lambda s_0 w$ mit $w \in A^*$.

5. Eine *Endkonfiguration* hat die Form $us'v$ mit $u \in A'^*$, $s' \in F$ und $v \in A^*$.

6. *Folgekonfigurationen* entstehen für alle $s, s' \in S$, $u, v \in A'^*$ und $a, c \in A'$ wie folgt:

$$
\begin{array}{lll}
\text{(i)} & usav \longmapsto us'bv, & \text{falls } (s', b, n) \in d(s, a) \\
\text{(ii)} & usav \longmapsto ubs'v, & \text{falls } (s', b, r) \in d(s, a) \\
\text{(iii)} & ucsav \longmapsto us'cbv & \multirow{2}{*}{$\Big\}$ falls $(s', b, l) \in d(s, a)$} \\
\text{(iv)} & \lambda sav \longmapsto s'\Box bv & \\
\text{(v)} & us\lambda \longmapsto us\Box &
\end{array}
$$

7. Die Arbeitsweise der Turing-Maschine besteht in der beliebigen Iteration solcher Konfigurationsübergänge:

$$u_1 s_1 v_1 \longmapsto u_2 s_2 v_2 \longmapsto \cdots \longmapsto u_k s_k v_k,$$

wofür kurz auch $u_1 s_1 v_1 \overset{k-1}{\longmapsto} u_k s_k v_k$ geschrieben werden kann, wenn die Zwischenschritte nicht explizit gebraucht werden. Ist auch die Schrittzahl unwesentlich, kann $k-1$ durch $*$ ersetzt werden.

8. Eine (partielle) Funktion $f\colon A^* \to A^*$ wird von einer deterministischen Turing-Maschine *TM berechnet*, falls für alle $v, w \in A^*$ gilt:

$$f(w) = v \quad \text{gdw.} \quad \lambda s_0 w \overset{*}{\longmapsto} us'v \text{ für geeignete } u \in A'^* \text{ und } s' \in F.$$

In diesem Fall wird f auch mit f_{TM} bezeichnet.

9.2 Deterministische Turing-Maschinen

Während bei einer beliebigen Turing-Maschine eine Konfiguration mehrere Folgekonfigurationen besitzen kann, ist bei einer deterministischen Maschine immer höchstens ein Übergang möglich. Jede Anfangskonfiguration kann also in genau einer Weise durch Konfigurationsübergänge ausgerechnet werden, wobei die Übergänge entweder unendlich fortsetzbar sind und die Maschine nicht hält, oder aber die Folge in einer Konfiguration endet, zu der es keine Folgekonfiguration gibt. Ist diese eine Endkonfiguration, so ist die Berechnung erfolgreich. Sonst hält die Maschine ohne Ergebnis.

Ohne Beweis sei angemerkt, daß deterministische und nichtdeterministische Turing-Maschinen dieselbe Klasse von Funktionen berechnen.

9.3 Übersetzung deterministischer Turing-Maschinen in CE-S

Als ein Beispiel, wie verschiedene Berechnungsmodelle miteinander verglichen werden können, soll gezeigt werden, daß jede von einer Turing-Maschine berechnete Funktion auch in CE-S spezifizierbar ist. Dazu wird, was in der Informatik häufig vorkommt, ein Übersetzer gebaut. Der Übersetzer, der TRANSLATOR heißen soll, verwandelt jede deterministische Turing-Maschine TM in eine Spezifikation **spec(**TM**)**, in der insbesondere die Funktion $f\colon A^* \to A^*$ deklariert ist. Es kann gezeigt werden, daß für die von TM berechnete Funktion f_{TM} und für $v, w \in A^*$ die Gleichheit $f_{TM}(w) = v$ genau dann gilt, wenn $f(w) = v$ im Sinne gleichwertiger Terme in **spec(**TM**)** gilt. TM und **spec(**TM**)** berechnen also dieselben Funktionen, was sich einprägsam als *Korrektheit* von TRANSLATOR ausdrücken läßt.

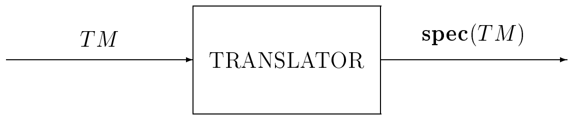
Die bei der Übersetzung verwendete Idee ist, für jeden Zustand $s \in S$ eine Funktion $f_s\colon A'^* \times A'^* \to A'^*$ einzuführen, so daß eine Konfiguration usv durch den Term $f_s(u, v)$ ausgedrückt wird und die Bildung von Folgekonfigurationen direkt als Gleichungen widergespiegelt werden kann. Weitere Gleichungen modellieren die Rolle von Anfangs- und Endkonfigurationen:

spec(TM**)**

$$
\begin{array}{lll}
\text{opns:} & f\colon A^* \to A^* & \\
 & f_s\colon A'^* \times A'^* \to A'^* & \text{für alle } s \in S \\
\text{vars:} & u, v \in A'^*, w, x \in A^*, c \in A'^* & \\
\text{eqns:} & f(w) = f_{s_0}(\lambda, w) & \\
 & f_s(u, av) = f_{s'}(u, bv) & \text{mit } (s', b, n) \in d(s, a) \\
 & f_s(u, av) = f_{s'}(ub, v) & \text{mit } (s', b, r) \in d(s, a) \\
 & f_s(uc, av) = f_{s'}(u, cbv) & \multirow{2}{*}{$\Big\}$ mit $(s', b, l) \in d(s, a)$} \\
 & f_s(\lambda, av) = f_{s'}(\lambda, \Box bv) & \\
 & f_s(u, \lambda) = f_s(u, \Box) & \text{mit } s \in S \\
 & f_{s'}(u, x) = x & \text{mit } s' \in F
\end{array}
$$

Beachte, daß dabei a und b keine Variablen, sondern Konstanten sind und daß jeder Eintrag der Form $(s', b, n) \in d(s, a)$ oder $(s', b, r) \in d(s, a)$ in der Zustandsüberführungsrelation von TM zu einer Gleichung und jeder Eintrag der Form $(s', b, l) \in d(s, a)$ zu zwei Gleichungen führt. Insbesondere enthält **spec(**TM**)** also nur – wie es der Definition von CE-S entspricht – endlich viele Gleichungen, weil A und S endlich sind.

Insgesamt erhält man einen Übersetzer, der als Diagramm so aussieht:



Er erweist sich im folgenden Sinne als korrekt:

9.4 Theorem

Sei TM eine deterministische Turing-Maschine, die die Funktion $f_{TM}: A^* \to A^*$ berechnet. Sei $\mathbf{spec}(TM)$ die zugehörige CE-S-Spezifikation mit der Deklaration $f: A^* \to A^*$. Dann gilt für alle $v, w \in A^*$:

$$f_{TM}(w) = v \quad gdw. \quad f(w) = v \text{ in } \mathbf{spec}(TM).$$

Beweis: $f_{TM}(w) = v$ bedeutet nach Definition $\lambda s_0 w \overset{*}{\longmapsto} us'v$ für geeignete $u \in A'^*$ und $s' \in F$. Nach dem Lemma unten folgt daraus $f_{s_0}(\lambda, w) = f_{s'}(u, v)$ in $\mathbf{spec}(TM)$, was mit der ersten und letzten Gleichung – wie gewünscht – ergibt:

$$f(w) = f_{s_0}(\lambda, w) = f_{s'}(u, v) = v.$$

Betrachte umgekehrt $f(w) = v$ in $\mathbf{spec}(TM)$. Nach dem Lemma unten folgt $f(w) \overset{*}{\longrightarrow} v$, was genau dann gilt, wenn $f_{s_0}(\lambda, w) \overset{*}{\longrightarrow} f_{s'}(u, v)$ für geeignete $u \in A'^*$ und $s' \in F$, da auf $f(w)$ nur die erste Gleichung anwendbar ist und auf v nur die letzte Gleichung rückwärts. Nach dem Lemma folgt $\lambda s_0 w \overset{*}{\longmapsto} us'v$, was gerade $f_{TM}(w) = v$ ergibt.

Lemma

Sei TM eine deterministische Turing-Maschine und $\mathbf{spec}(TM)$ die zugehörige CE-S-Spezifikation.

1. Dann gilt für alle $s, s' \in S$ und $u, v, u', v' \in A'^*$:

$$usv \overset{*}{\longmapsto} u's'v' \quad gdw. \quad f_s(u, v) \overset{*}{\longrightarrow} f_{s'}(u', v')$$

2. Sei $v, w \in A^*$ mit $f(w) \overset{*}{\longleftrightarrow} v$. Dann gilt $f(w) \overset{*}{\longrightarrow} v$.

Beachte, daß der zweite Teil besagt, daß die Terme $f(w)$ und v in $\mathbf{spec}(TM)$ genau dann gleichwertig sind, wenn sich $f(w)$ durch Gleichungsanwendungen von links nach rechts in v überführen läßt, daß also in diesem Fall die Symmetrie keine Rolle spielt.

Die Beweise beider Teile des Lemmas, die hier nicht aufgeführt werden, lassen sich mit vollständiger Induktion über die Zahl von Konfigurationsübergängen bzw. Gleichungsanwendungen führen. Um das in Teil 2 zu bewerkstelligen, muß eine allgemeinere Behauptung gezeigt werden, nämlich:

Sei $\overline{v} \in A^*$. Sei $x = \overline{u}$ für $\overline{u} \in A^*$ oder $x = f_s(u, v)$ für $s \in S$ und $u, v \in A^*$ oder $x = f(w)$ für $w \in A^*$. Sei ferner $x \overset{*}{\longleftrightarrow} \overline{v}$. Dann gilt $x \overset{*}{\longrightarrow} \overline{v}$.

10 Literaturhinweise

Die folgenden Bücher geben einen Einblick in die verschiedenen Gebiete der Theoretischen Informatik bzw. behandeln das dafür benötigte Basiswissen.

Alfred v. Aho, Jeffrey D. Ullman. Foundations of Computer Science. Computer Science Press, 1992.

Michael A. Arbib, A.J. Kfoury, Robert N. Moll. A Basis for Theoretical Computer Science. Springer-Verlag, 1981.

Erwin Engeler, Peter Läuchli. Berechnungstheorie für Informatiker. B.G.Teubner Stuttgart, 1992

John E. Hopcroft, Jeffrey D. Ullman. Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley, 1990.

A.J. Kfoury, Robert N. Moll, Michael A. Arbib. A Programming Approach to Computability. Springer-Verlag, 1982.

Karl Rüdiger Reischuk. Einführung in die Komplexitätstheorie. Teubner, 1990.

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

Volker Sperschneider, Barbara Hammer. Theoretische Informatik. Eine problemorientierte Einführung. Springer-Verlag, 1996.

Franz Stetter. Grundbegriffe der Theoretischen Informatik. Springer-Verlag, 1994.

Ingo Wegener. Theoretische Informatik. B.G.Teubner Stuttgart, 1993.

Dietmar Wätjen. Theoretische Informatik. Eine Einführung. Oldenbourg Verlag, 1994.