Berechenbarkeit

Informatik I – Automatisierungstechnik in der Produktion

Stefan Zimmer
22.12.2004

1. Grenzen unserer Programmierkunst

- Die Menge der Programme, die wir bisher schreiben können, wird durch drei Umstände eingeschränkt:
  - In den Ada-Büchern finden wir noch Kapitel über bisher nicht behandelte Sprachelemente (z.B. für das Abfangen von Laufzeitfehlern) und Bibliotheksfunktionen (z.B. für das Lesen und Schreiben von Dateien oder um unsere Programme bunter und klickbarer zu machen).

    Unsere Ada-Kenntnisse sind aber mittlerweile soweit fortgeschritten, dass wir diese Dinge bei Bedarf selber erarbeiten können.
  - Wir wissen wenig darüber, wie man ein größeres Softwareprojekt angehen würde.

    Dieses Problem ist ernsthafter als das erste, leider würde seine Behandlung den Rahmen der Vorlesung erheblich sprengen. . .
  - Uns schließlich kann man nach prinzipiellen Grenzen der Probleme fragen, für die sich Programme schreiben lassen.

    Darum wird es heute gehen.

2. Programme realisieren Abbildungen

2.1. Modell der Wirkungsweise eines Programms

- Um die theoretischen Grenzen auszuloten, lösen wir uns von der speziellen Programmiersprache Ada95 und betrachten stattdessen ein sehr allgemeines Modell für Programme.
- Ein Programm setzt einen Algorithmus um, indem es in einer endlichen Zahl wohldefinierter Schritte mit endlichem Speicherplatzbedarf für gegebene Eingabedaten zugehörige Ausgabedaten produziert.

  Das Programm selber bestehe aus einer endlichen Anzahl von Symbolen einer Sprache, die deren Wirkungsweise festlegt.
- Abstrakt beschreiben wir ein Programm mittels Funktionen, die k-Tupel natürlicher Zahlen (die Eingabedaten) auf eine natürliche Zahl (die Ausgabe) abbilden:

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

2.2. Abzählbare Mengen

- Um einzusehen, dass die Beschränkung auf Funktionen $\mathbb{N}^k \to \mathbb{N}$ keine Einschränkung darstellt, erinnern wir uns an das Konzept der abzählbaren Mengen.
- Eine Menge M heißt abzählbar, wenn es eine Abbildung $f : \mathbb{N} \to M$ gibt mit

$$M = \{f(1), f(2), f(3) \ldots\},$$

  d.h., wenn man die Elemente mit den natürlichen Zahlen nummerieren kann (wenn ein Element mehrere Nummern erhält, schadet das nicht).
- Insbesondere ist natürlich $\mathbb{N}$ selber abzählbar, aber auch die ganzen Zahlen $\mathbb{Z}$ und die rationalen Zahlen $\mathbb{Q}$.
- Nicht abzählbar hingegen sind die reellen Zahlen $\mathbb{R}$
- Die k-Tupel natürlicher Zahlen $\mathbb{N}^k$ sind ebenfalls abzählbar (nicht aber die Menge aller Folgen natürlicher Zahlen).
- Die Menge aller Worte Σ^* über einer abzählbaren Menge Σ ist abzählbar, insbesondere die über endlichen Mengen (Wörter haben eine endliche Länge!).

- Da unsere Eingabedaten Wörter über einer endlichen Menge (z.B. dem Zeichenvorrat eines Datentyps Character) sind, können wir sie nummerieren und geben bei der Eingabe nur noch diese (möglicherweise sehr große) Zahl an: aus einem Programm, dass z.B. Zeichenketten als Eingabe verlangt, haben wir nun ein äquivalentes mit einer natürlichen Zahl als Eingabe erzeugt.
- Analog nummerieren wir die möglichen Ausgaben und sehen ein, dass wir uns zum Modellieren auf Abbildungen $\mathbb{N} \to \mathbb{N}$ beschränken können.
- Die „Erweiterung“ auf Abbildungen $\mathbb{N}^k \to \mathbb{N}$ dient nur der Erleichterung beim Aufschreiben, nötig wäre sie nicht.
- Wir rechnen mit den ganzen natürlichen Zahlen — es ist kein prinzipielles Problem, mit beliebig großen Zahlen zu rechnen; die Bereichsgrenzen der Ada-Datentypen entsprangen ja dem Wunsch nach Effizienz, über solche profanen Details sind wir diese Woche erhaben (wenn der Speicher ausgeht, kaufen wir neuen nach).
- Da Programme selber eine endliche Länge besitzen, kommen sie — geeignet nummeriert — auch als Eingabedaten infrage (z.B. für einen Übersetzer).

2.3. Realisierung durch Programme

- Nun können wir die abstrakte Vorstellung von einem Programm π präzisieren: es liest die Eingabe $e \in \mathbb{N}^k$ und in Abhängigkeit von e tritt dann eine der folgenden drei Möglichkeiten ein:
  - Es hält nach endlich vielen Schritten an und liefert eine Ausgabe $a(e) \in \mathbb{N}$,
  - es hält nach endlich vielen Schritten an, ohne eine Ausgabe zu liefern (z.B., wenn bei der Berechnung ein Fehler aufgetreten ist), oder
  - es hält nicht nach endlich vielen Schritten an (z.B., weil es in eine Endlosschleife geraten ist).
- Das Programm beschreibt also eine partielle Abbildung

$$f_\pi : \mathbb{N}^k \to \mathbb{N}.$$

- Partiell deshalb, weil es ja Eingaben $e \in \mathbb{N}^k$ ohne zugeordnete Ausgaben geben kann. Wenn das Programm für jede Eingabe $e \in \mathbb{N}^k$ eine Ausgabe $a(e)$ liefert, liegt eine Abbildung im eigentlichen Sinn vor, zur Verdeutlichung hier auch totale Abbildung genannt.
- Wir sagen, dass das Programm π eine (partielle) Funktion f realisiert, falls $f_\pi = f$.

Stefan Zimmer

3. Berechenbare Funktionen

3.1. Definition und Beispiele

- Wir nennen eine Funktion f berechenbar, wenn es ein Programm π gibt, das sie realisiert: $f = f_\pi$.
- Über die Art der zugelassenen Programme (z.B. die Sprache, in der sie notiert werden), werden wir im Folgenden nur so allgemeine Annahmen machen, dass diese von jeder „vernünftigen“ Sprache erfüllt werden.
- Schöne Beispiele im Zusammenhang mit berechenbaren Funktionen liefert die Dezimalentwicklung von (der Zahl) π, also die Folge $3, 1, 4, 1, 5, 9, \ldots$.
- Die Funktion a_π, mit $a_\pi(n) = 1$, falls n Anfang der Dezimalentwicklung von π ist, und 0 sonst (also z.B. $a(31415) = 1$, $a(1415) = 0$), ist berechenbar — man kennt Algorithmen, die π mit jeder gewünschten Genauigkeit berechnen, so dass sich der Wert von $a_\pi(n)$ für beliebiges n bestimmen lässt.

- Von der Funktion m_π, mit $m_\pi(n) = 1$, falls n in der Dezimalentwicklung von π vorkommt, und 0 sonst, weiß man nicht, ob sie berechenbar ist.

  Man kennt zwar derzeit noch kein n mit $m_\pi(n) = 0$, aber es ist nicht bewiesen, dass es keins gibt.

- Interessant ist der Status der Funktionen $w_{\pi,d}$, wobei d eine Ziffer ist und $w_{\pi,d}(n) = 1$, falls es in der Dezimalentwicklung von π eine Stelle mit wenigstens n Ziffern d in ununterbrochener Folge gibt, und 0 sonst.

  Diese Funktionen sind berechenbar! Wir wissen zwar nicht, wie wir sie berechnen können, aber es gilt folgendes:

  – $w_{\pi,d}$ ist entweder konstant 1 oder
  – es gibt ein $n_0(d)$ mit $w_{\pi,d}(n) = 1$ für alle $n < n_0(d)$ und $w_{\pi,d}(n) = 0$ für alle $n \geq n_0(d)$.

  In beiden Fällen können wir ein Programm angeben, dass diese Abbildung realisiert (die $n_0(d)$ können ggf. als Konstanten im Programm stehen).

  Wir haben mithin Funktionen, von denen wir Berechenbarkeit zeigen können, ohne ein Pogramm angeben zu können.

- Dass es Funktionen gibt, die prinzipiell nicht berechenbar sind, machen wir uns dadurch klar, dass wir analog zu der Funktion a_π, die auf „Anfang von π“ überprüft hat, für jedes $x \in \mathbb{R}$ mit $x > 0$ ein a_x definieren.
- Es gibt überabzählbar viele positive reelle Zahlen; verschiedene Zahlen x und y haben verschiedene Dezimalentwicklungen und somit auch verschiedene Funktionen: es gibt ein n mit $a_x(n) \neq a_y(n)$. Somit haben wir überabzählbar viele Funktionen definiert.
- Da es aber nur abzählbar viele Programme gibt, folgt: es gibt $x \in \mathbb{R}$, deren zugeordnetes a_x nicht berechenbar ist (das sind übrigens sogar „die allermeisten“).
- Aber wir sieht es mit einer konkreten Funktion aus, die nicht berechenbar ist? Das ist schwieriger, weil hier mehr Details der Programmiersprache eingehen (für jedes x, können wir leicht eine Programmiersprache erfinden, die gerade a_x berechnen kann).
- Abhilfe schafft hier ein Trick: die Programme selber als Eingabedaten verwenden!

3.2. Das Halteproblem

- Es wäre praktisch, wenn der Übersetzer uns warnen würde, wenn unser Programm bei bestimmten Eingabewerten in eine Endlosschleife gerät. Dass das leider im Allgemeinen nicht möglich ist, werden wir nun sehen.
- Die Funktion h, des Halteproblems bekommt als Parameter ein Programm π und eine Eingabe w für π (beides nach geeigneter Nummerierung als natürliche Zahlen aufgefasst).
- Es soll gelten:

$$h(\pi, w) = \begin{cases} 0 & \text{falls } \pi \text{ bei Eingabe } w \\ & \text{nach endlich vielen Schritten anhält,} \\ 1 & \text{sonst.} \end{cases}$$

  Dadurch ist h eindeutig definiert.
- Nun werden wir zeigen, dass h nicht berechenbar ist. Das machen wir als Widerspruchsbeweis und nehmen an, wir hätten ein Programm α mit $f_\alpha = h$.

Stefan Zimmer

- Daraus machen wir ein Programm β, das nur noch eine Zahl einliest, nämlich das Programm π. Es soll 0 ausgeben, wenn π mit sich selbst als Eingabe nach endlich vielen Schritten anhält, sonst 1:
$$f_\beta(\pi) = h(\pi, \pi).$$
- Und ein drittes Programm γ, das 1 liefert, falls β eine 1 liefert (die Eingabe π mit sich selbst als Eingabe nicht anhält) und andernfalls nicht anhält. In Ada sähe das etwa so aus:

  if $f_\beta(\pi) = 0$ **then loop null**; **end loop**; **end if**;
- Und nun die große Frage: was macht γ, wenn es sich selbst als Eingabe bekommt?
- Anhalten geht nicht (weil es dann ein Programm wäre, das mit sich selbst als Eingabe anhält).
- Nicht anhalten geht auch nicht (weil es mit einem derartigen Programm als Eingabe mit Ergebnis 1 anhält).
- Das ist der gesuchte Widerspruch: es kann kein Programm geben, das h realisiert!

3.3. Praktische Konsequenzen aus dem Halteproblem

- Das Halteproblem ist keineswegs nur ein Beispiel von theoretischer Bedeutung: wenn wir ein Programm hätten, dass es löste, würden wir viele Programmierfehler gar nicht mehr machen können.
- Man kann natürlich Verfahren erfinden, die viele Fälle abdecken, aber — wie wir gesehen haben — niemals alle.
- Nun kann man sich fragen, ob man nicht eine Programmiersprache erfinden kann, in der jedes Programm terminiert.
- Das ist gar nicht so schwer: wenn wir die Ada-Konstruktionen, die wir kennen gelernt haben, daraufhin untersuchen, sehen wir, dass wir nur zwei Möglichkeiten haben, nicht terminierende Programme zu schreiben: die Schleife (ohne Kontrollelement oder als **while**-Schleife; die **for**-Schleife ist in diesem Zusammenhang harmlos) und den rekursiven Prozeduraufruf.
- Wenn wir diese Möglichkeiten verbieten, würde sich das Halteproblem gar nicht erst stellen.

- Leider zeigt sich, dass man Funktionen angeben kann, die man zwar mit Rekursion und/oder **while**-Schleife berechnen kann, nicht aber nur mit **for**-Schleifen: die verbesserte Sicherheit wäre mit reduzierten Möglichkeiten erkauft.
- Andersherum kann man sich fragen, ob man mit mehr Sprachkonstruktionen als wir sie kennen auch mehr Funktionen berechnen kann.
- Die Vermutung hier ist: nein — die unterschiedlichsten Versuche, Programme zu beschreiben, enden stets bei einer Menge berechenbarer Funktionen, die nicht größer ist, als die, die wir auch (prinzipiell) berechnen können.
- Wir haben also schon allerhand gelernt. . .

4. Literatur

Zum Nachlesen diese Woche:

- Lagally-Skript, Kapitel 96 „Algorithmen“ und Kapitel 97 „Berechenbarkeit“
- Hauptfach-Vorlesung, im WS 2003/04: Kapitel 1.1.3 „Realisierte Abbildung“ sowie 1.2.1 „Charakteristika von Algorithmen“ und 1.2.2 „Grenzen der Algorithmen, Unentscheidbarkeit“.
- Und für weitergehend Interessierte: Uwe Schöning, Theoretische Informatik kurzgefasst, Spektrum Akademischer Verlag.