Berechenbarkeit und Church'sche These

Hans U. Simon (RUB)

Email: simon@lmi.rub.de

Homepage: http://www.ruhr-uni-bochum.de/lmi

Partiell definierte Funktionen

Wir betrachten im Folgenden partiell definierte Funktionen der Form

$$f : \mathbb{N}^k \rightarrow \mathbb{N} \text{ bzw. } f : \Sigma^* \rightarrow \Sigma^* \quad .$$

Für x außerhalb des Definitionsbereiches gilt dann

$$f(x) = \text{„undefiniert“} \quad .$$

Intuition: Rechenprogramme mit Eingaben aus $\mathbb{N}^k$ bzw. Σ^* werden

entweder nach endlich vielen Schritten mit einem (durch eine Ausgabekonvention festgelegten) Ergebnis stoppen,

oder in eine unendliche Schleife geraten.

Die von einem Programm berechnete Funktion ist also i.A. nur partiell definiert.

Intuitive Berechenbarkeit

Informelle Definition: Eine Funktion

$$f : \mathbb{N}^k \rightarrow \mathbb{N} \text{ bzw. } f : \Sigma^* \rightarrow \Sigma^*$$

heißt „(intuitiv) berechenbar“, wenn es eine „mechanisch anwendbare“ Rechenvorschrift gibt, die bei Eingabe x

- nach „endlich vielen Schritten“ zur Ausgabe $f(x)$ führt, falls $f(x)$ definiert ist,
- in eine „unendliche Schleife“ führt, falls $f(x)$ undefiniert ist.

Beispiele

Die total undefinierte Funktion

$$\Omega(n) = \text{„undefiniert“}$$

für alle $n \in \mathbb{N}$ ist berechenbar:

Erstelle ein Programm (in Deiner Lieblingssprache), das sich (ungeachtet der Eingabe) in eine Endlosschleife begibt !

Beispiele (fortgesetzt)

Abkürzung: DBE = Dezimalbruchentwicklung

Die Funktion

$$f(n) = \begin{cases} 1 & \text{falls die DBE von } \pi \text{ mit den Ziffern der DBE von } n \text{ beginnt} \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar:

Benutze ein Verfahren, das beliebige genaue Approximationen von π erstellen kann und jeweils eine Fehlerabschätzung mitliefert.

Beispiele (fortgesetzt)

Der Status (berechenbar versus nicht berechenbar) der Funktion

$$g(n) = \begin{cases} 1 & \text{falls die DBE von } \pi \text{ die DBE von } n \text{ als Teilstring enthält} \\ 0 & \text{sonst} \end{cases}$$

ist ungeklärt:

Unser bisheriges Wissen über die Zahl

$$\pi$$

reicht zur Beantwortung dieser Frage nicht aus.

Man kann nicht einmal ausschließen, dass **jede** endliche Ziffernfolge irgendwo in π als Teilstring vorkommt. In diesem Falle wäre g die konstante Einsfunktion (undd dann trivialerweise berechenbar).

Subtil, subtil ...

Zu verlangen, dass eine mechanisch anwendbare Rechenvorschrift **existiert**, bedeutet **nicht**, dass wir wissen, um welche Rechenvorschrift es sich handelt.

Beispiele (fortgesetzt)

Die Funktion

$$h(n) = \begin{cases} 1 & \text{falls die DBE von } \pi \text{ den Teilstring } 7^n \text{ enthält} \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar:

Fall 1: π enthält beliebig lange 7er–Sequenzen als Teilstrings.
Dann ist h die (trivial berechenbare) konstante Einsfunktion.

Fall 2: Es gibt eine längste in π als Teilstring enthaltene 7er–Sequenz, sagen wir der Länge n_0.
Dann ist h die einfach berechenbare Funktion

$$h(n) = \begin{cases} 1 & \text{falls } n \leq n_0 \\ 0 & \text{falls } n > n_0 \end{cases}$$

Beim gegenwärtigen Wissensstand über π ist unklar, welcher der beiden Fälle vorliegt.

Beispiele (fortgesetzt)

Die Funktion

$$i(n) = \begin{cases} 1 & \text{falls das LBA–Problem eine positive Antwort hat} \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar:

Fall 1: Jede kontextsensitive Sprache kann durch einen DLBA erkannt werden.
Dann ist i die konstante Einsfunktion.

Fall 2: Es gibt eine kontextsensitive Sprache, die von keinem DLBA erkannt wird.
Dann ist i die konstante Nullfunktion.

Beim gegenwärtigen Wissensstand ist unklar, welcher der beiden Fälle vorliegt.

Beispiele (fortgesetzt)

Ähnlich wie bei π können wir zu jeder reellen Zahl $r \in \mathbb{R}$ die Funktion

$$f_r(n) = \begin{cases} 1 & \text{falls die DBE von } r \text{ mit den Ziffern der DBE von } n \text{ beginnt} \\ 0 & \text{sonst} \end{cases}$$

zuordnen.

Frage: Ist f_r für jedes $r \in \mathbb{R}$ berechenbar ?

Antwort: Nein !

Begründung: Eine Rechenvorschrift sollte durch einen endlichen Text über einem endlichen Alphabet beschreibbar sein. Daher gibt es nur abzählbar viele Rechenvorschriften, wohingegen $\mathbb{R}$ überabzählbar unendlich ist.

Die Berechenbarkeit von f_r ist demnach eher die Ausnahme.

Wozu ein formales Rechenmodell ?

- Zum Nachweis der Berechenbarkeit genügt (in der Regel) die Angabe einer konkreten Rechenvorschrift (und somit ein intuitives Verständnis des Berechenbarkeitsbegriffes).

- Zum Nachweis der Unberechenbarkeit ist hingegen zu zeigen, dass keine passende Rechenvorschrift existiert. Dazu brauchen wir eine klare Vorstellung über die Gesamtheit aller Rechenvorschriften.

Ausblick: Formale Definitionen der Berechenbarkeit

- durch Turing–Programme berechenbar
- durch WHILE–Programme berechenbar
- durch GOTO–Programme berechenbar
- μ–rekursiv

Alle diese Vorschläge (von Turing, Church und anderen Mathematikern Mitte der 1930er unterbreitet) haben sich als äquivalent erwiesen. Zudem wurde bislang keine intuitiv berechenbare Funktion gefunden, die nicht auch Turing–berechenbar wäre. Dies führte zur (formal nicht beweisbaren)

Church'schen These: Die Klasse der intuitiv berechenbaren Funktionen stimmt überein mit der Klasse der durch Turing–berechenbaren (bzw. WHILE–berechenbaren, GOTO–berechenbaren, μ–rekursiven, ...) Funktionen.

Turing–Berechenbarkeit

Eine Funktion

$$f : \Sigma^* \to \Sigma^*$$

heißt Turing–berechenbar **gdw** eine DTM M existiert mit folgenden Eigenschaften:

- Die Überführungsfunktion δ ist auf Endzuständen (und nur auf diesen) undefiniert, so dass die Rechnung von M abbricht, sowie ein Endzustand erreicht ist.
- Falls $f(x) = y$, dann gibt es einen Endzustand $z_e \in E$ mit $z_0 x \vdash^* z_e y$.
- Falls $f(x) =$ „undefiniert“, dann erreicht M bei der Rechnung auf Eingabe x keinen Endzustand (Endlosrechnung).

Turing–Berechenbarkeit (fortgesetzt)

Die Turing–Berechenbarkeit von Funktionen der Form

$$f : \mathbb{N}^k \rightarrow \mathbb{N}$$

ist analog definiert, wobei wir im Falle von

$$f(x) = y \text{ mit } x = (n_1, \ldots, n_k)$$

anstelle von $z_0 x \vdash^* z_e y$ nun

$$z_0 \mathrm{bin}(n_1) \cdots \mathrm{bin}(n_k) \vdash^* z_e \mathrm{bin}(y)$$

fordern. Hierbei bezeichnet $\mathrm{bin}(\cdot)$ die Binärdarstellung ohne führende Nullen.

Varianten dieser Definition

Bemerkung: Wir können auch erlauben, dass die Überführungsfunktion δ einer TM M auf beliebigen Zuständen aus Z undefiniert sein darf.

- Wir sagen, dass M stoppt, wenn M im Zustand $z \in Z$ Symbol A liest (bzw. mehrere Symbole bei Mehrband–Maschinen) und $\delta(z, A)$ undefiniert ist.

- Bei der Turing–Berechenbarkeit einer Funktion f durch M können wir dann im Falle $f(x) =$„undefiniert" erlauben, dass M auf Eingabe x eine Endlosrechnung durchführt oder aber in einem Zustand aus $Z \setminus E$ stoppt.

Definition: Der Haltebereich $H_T(M)$ einer DTM M ist definiert als die Menge aller Eingabewörter, auf denen M nach endlich vielen Rechenschritten stoppt.

Beispiele

Die Nachfolgerfunktion

$$s(n) = n + 1$$

ist Turing–berechenbar:

Siehe unsere frühere Implementierung eines Binärzählers.

Beispiele (fortgesetzt)

Die total undefinierte Funktion Ω ist Turing–berechenbar durch die DTM mit

$$\delta(z_0, a) = (z_0, a, R) \ ,$$

die auf jeder Eingabe eine unendliche Schleife durchläuft.

Beispiele (fortgesetzt)

Zu einer Sprache L vom Typ 0 betrachte die Funktion

$$\chi'_L(w) = \begin{cases} 1 & \text{falls } w \in L \\ \text{„undefiniert“} & \text{falls } w \notin L \end{cases}$$

Die Turing–Berechenbarkeit von χ'_L kann folgendermaßen eingesehen werden:

- Es gibt eine Grammatik G vom Typ 0, welche L generiert.
- Es gibt eine NTM M, welche L erkennt, indem Ableitungen $S \Rightarrow_G^* w$ geraten werden. M kann so implementiert werden, dass
  - nach Auffinden einer Ableitung Ausgabe 1 produziert und in einem Endzustand gestoppt wird,
  - bei Nicht–Auffinden einer Ableitung eine unendliche Schleife betreten wird.
- Dann wird χ'_L berechnet durch die deterministische Simulation M' von M.

Mehrspurenmaschinen

Zu einem gegebenen Alphabet Γ können wir „Supersymbole" aus Γ^k betrachten. Wenn das Arbeitsalphabet einer TM ein Supersymbol $(A_1, \ldots, A_k)$ enthält, dann ist es anschaulich sich vorzustellen, dass

- das Band in k „Spuren" zerlegt werden kann,
- und beim Abspeichern von $(A_1, \ldots, A_k)$ in einer Zelle, das Symbol A_i in der i–ten Spur der Zelle steht.

Beachte: Mehrspurnmaschinen haben zwar ein unkonventionelles Arbeitsalphabet (welches k-Tupel enthält), entsprechen aber unserer Standarddefinition einer TM (**kein** neues Modell).

Mehrbandmaschinen

Definition: Unter einer k-Band TM verstehen wir eine TM mit k Bändern und einem Kopf pro Band. Die insgesamt k Köpfe können sich in einem Rechenschritt in verschiedene Richtungen bewegen. Die Überführungsfunktion δ hat nun die Form

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, N\}^k$$

mit der offensichtlichen Interpretation.

Mehrbandmaschinen sind nicht mächtiger als das Standardmodell wie der folgende sogenannte Bandreduktionssatz zeigt:

Satz: Eine k-Band TM M kann von einer 1-Band TM M' simuliert werden. Ist dabei M eine DTM, so auch M'.

Beweis

- M' besitzt für jeden Zustand z von M einen entsprechenden Zustand z' (und weitere Zustände).
- M' simuliert
  - einen Schritt von M mit Zustandswechsel von z_1 nach z_2
  - durch eine Folge von Schritten, welche im Zustand z'_1 startet und im Zustand z'_2 endet

Nach diesem Schema verlaufende Simulationen heißen „Schritt für Schritt Simulation“.

Beweis (fortgesetzt)

Die wesentliche Schwierigkeit besteht darin, die Beschriftung der k-Band TM M auf einem einzigen Band unterzubringen. M' benutzt dazu ein Band mit k Spuren. Dabei soll stets gelten:

(1) Spur i des Bandes von M' enthält die Beschriftung von Band i von M $(1 \leq i \leq k)$.

(2) Zelle 1 von M' enthält genau die k Symbole, auf denen die k Köpfe von M positioniert sind.

(3) Zu Beginn der Simulation des nächsten Rechenschrittes von M befindet sich der Kopf von M' auf Zelle 1.

Bedingungen (2) und (3) sorgen dafür, daß M' die von M gelesenen k Symbole kennt.

Beweis (fortgesetzt)

Um einen Schritt von M zu simulieren, geht M' vor wie folgt:

- Wenn M Symbole $a_1, \ldots, a_k$ durch $b_1, \ldots, b_k$ ersetzt, ersetzt M' in Zelle 1 $(a_1, \ldots, a_k)$ durch $(b_1, \ldots, b_k)$.
- Wenn M Kopf i nach rechts (bzw. links) bewegt, so verschiebt M' die Inschrift von Spur i um eine Position in die entgegengesetzte Richtung (positioniert aber im Anschluss den Kopf wieder auf Zelle 1).
- Wenn M in Zustand z übergeht, geht M' in Zustand z' über.

Hierdurch bleiben Bedingungen (1), (2) und (3) erhalten und die Simulation ist korrekt.

Offensichtlich arbeitet M' deterministisch, falls M deterministisch arbeitet.

Zusätzliche Beobachtung

Wenn M auf Eingaben der Länge n

- maximal $S(n)$ Zellen ihres Bandes besucht
- und maximal $T(n)$ Schritte rechnet,

dann

- besucht M' ebenfalls maximal $S(n)$ Zellen
- und rechnet maximal $O(S(n) \cdot T(n))$ Schritte (da jeder Schritt von M in $O(S(n))$ Schritten von M' simuliert werden kann).

Ein „Baukastensystem" für Turing–Maschinen

- Zu einer 1–Band–TM M bezeichne $M(i,k)$, oder einfach $M(i)$, die k–Band TM, die das „Programm" von M auf ihrem i–ten Band simuliert (und auf den anderen Bändern keine Modifikationen vornimmt).
- „Band := Band +1" bezeichne die früher bereits besprochene 1–Band DTM zur Berechnung der Funktion $s(n) = n + 1$.
- Statt „Band := Band+1"(i) schreiben wir „Band i:= Band i +1".
- Definiere die „modifizierte Differenz" wie folgt:

$$n \dot{-} m = \max\{0, n - m\} \ .$$

Die Notationen

$$\begin{array}{lcl} \text{„Band i} & := & \text{Band i } \dot{-} 1\text{"} \\ \text{„Band i} & := & \text{Band j} \\ \text{„Band i} & := & 0 \end{array}$$

sind dann analog zu verstehen.

Komposition von TMs

Die Komposition zweier TMs

$$M_i = (Z_i, \Sigma, \Gamma_i, \delta_i, z_i, \Box, E_i), i = 1,2, \; Z_1 \cap Z_2 = \emptyset$$

ist definiert als die TM

$$M = (Z_1 \cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, z_1, \Box, E_2) \; ,$$

wobei

$$\delta(z,A) = \begin{cases} \delta_1(z,A) & \text{falls } z \in Z_1 \setminus E_1 \\ (z_2, A, N) & \text{falls } z \in E_1 \\ \delta_2(z,A) & \text{falls } z \in Z_2 \end{cases} \; .$$

M führt also zuerst das Programm von M_1 aus und (falls M_1 einen Endzustand erreicht) dann das Programm von M_2.

Notation als „Flussdiagramm“: start $\rightarrow M_1 \rightarrow M_2 \rightarrow$ stop.

Notation wie bei Programmiersprachen: $M_1; M_2$

Beispiel

Die DTM

$$\begin{array}{lll} \text{start} & \rightarrow & \text{„Band := Band +1“} \\ & \rightarrow & \text{„Band := Band +1“} \\ & \rightarrow & \text{„Band := Band +1“} \rightarrow \text{stop} \end{array}$$

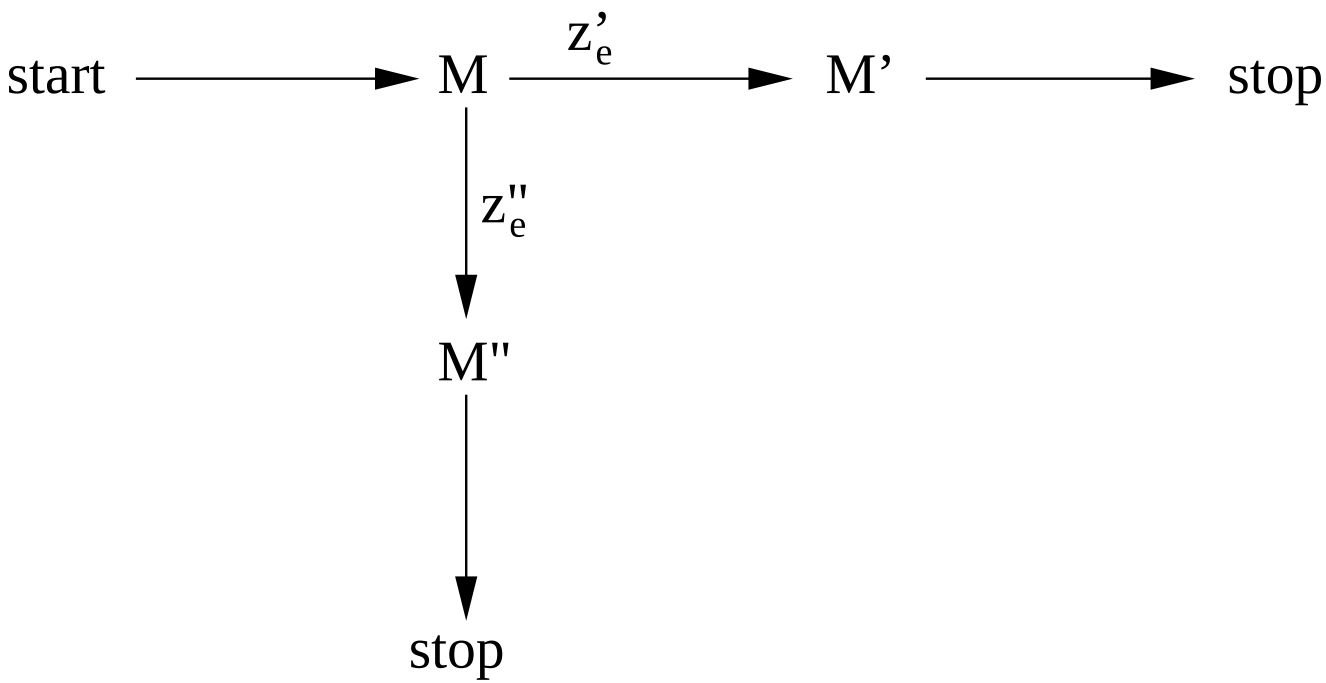
addiert zu einer gegebenen natürlichen Zahl die Konstante 3 hinzu.

Bedingte Komposition von TMs

Eine TM, welche

- zunächst das Programm einer TM M ausführt,
- hernach das Programm von M', falls M im Endzustand z_e' stoppt,
- bzw. das Programm M'', falls M im Endzustand z_e'' stoppt,

notieren wir in der Form



Die Abfrage–Maschine

„Band=0?“ bezeichnet eine DTM mit folgenden Eigenschaften:

- Sie hat vier Zustände $z_0, z_1, \mathrm{JA}, \mathrm{NEIN}$ mit $\mathrm{JA}, \mathrm{NEIN}$ als Endzuständen.
- Sie verändert den Bandinhalt nicht. Zu Beginn und am Ende der Rechnung ist der Kopf auf dem ersten Zeichen der Eingabe positioniert.
- Ihre Hauptaufgabe ist zu testen, ob die Eingabe nur aus dem Zeichen 0 besteht. Falls dem so ist stoppt sie im Endzustand JA; andernfalls stoppt sie im Endzustand NEIN.

Eine solche DTM ist einfach zu entwerfen:

$$\delta(z_0, a) = \begin{cases} (z_1, a, R) & \text{falls } a = 0 \\ (\mathrm{NEIN}, a, N) & \text{sonst} \end{cases}$$

$$\delta(z_1, a) = \begin{cases} (\mathrm{JA}, a, L) & \text{falls } a = \Box \\ (\mathrm{NEIN}, a, L) & \text{sonst} \end{cases}$$

Einbettung einer TM in eine WHILE–Schleife

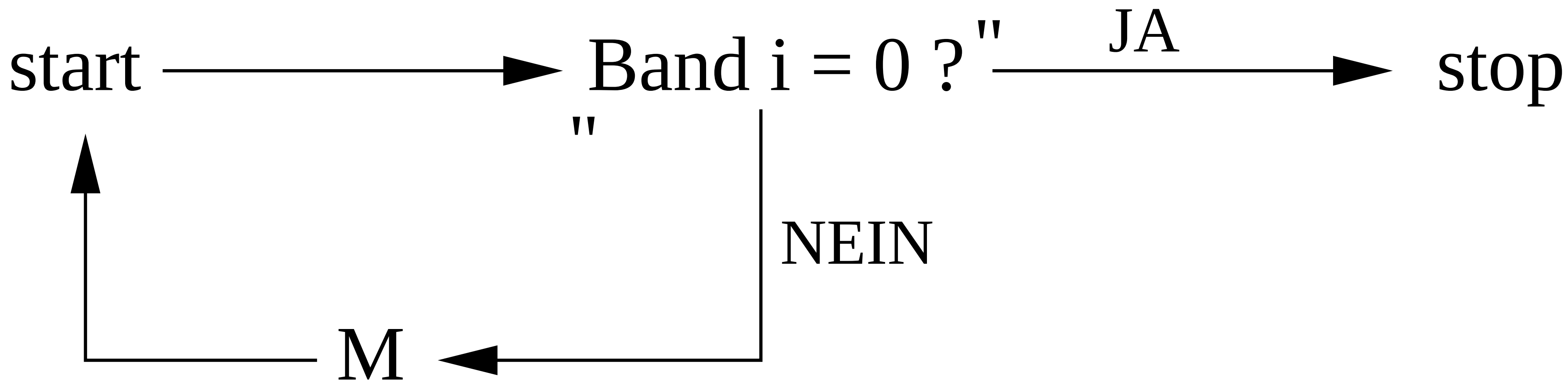
Statt „Band=0?"(i) schreiben wir einfach

„Band i = 0 ?"

Zu einer gegebenen TM M bezeichne

„WHILE Band i $\neq$ 0 DO M"

die durch folgendes Flussdiagramm gegebene TM:



Résumée

- Mit dem Baukastensystem lassen sich aus elementaren TMs komplexere TMs zusammensetzen, die Strukturen höherer Programmiersprachen wie zum Beipiel
  - bedingte Anweisungen
  - WHILE–Schleifen
  - Prozedurkonzept

  (ansatzweise) realisieren.
- Die Realisierung macht Gebrauch von Mehrband–TMs. Wie wir wissen lässt sich aber jede Mehrband–TM durch eine Einband–TM simulieren.

Zeichenvorrat für LOOP-Programme

Variablen:	x_0 x_1 x_2 $\cdots$
Konstanten:	0 1 2 $\cdots$
Trennsymbole:	; :=
Operationszeichen:	$+$ $-$
Schlüsselwörter:	LOOP DO END

Syntax von LOOP-Programmen

Induktive Definition:

1. Jede Wertzuweisung der Form

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

   (für eine Konstante c) ist ein LOOP-Programm.

2. Die Hintereinanderschaltung

$$P_1 \ ; \ P_2$$

   von LOOP-Programmen P_1, P_2 ist ein LOOP-Programm.

3. Das iterierte Durchlaufen

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

   eines LOOP-Programmes P ist ein LOOP-Programm.

Semantik von LOOP-Programmen

Kanonisch definiert bis auf:

- „$a - b$" wird interpretiert als „modifizierte Differenz" $a \dot{-} b := \max\{a - b, 0\}$.
- Bei einem LOOP-Programm der Form LOOP x_i DO P END wird P so oft ausgeführt wie der Wert der Variablen x_i zu *Beginn* angibt. (Änderung des Werte von x_i im Innern von P haben auf die Anzahl der Wiederholungen also keinen Einfluss.)
  Folgerung: LOOP-Programme terminieren stets.

Konventionen beim Berechnen von $f : \mathbb{N}^k \rightarrow \mathbb{N}$ durch ein LOOP-Programm:

- Eingabewerte $n_1, \ldots, n_k$ anfangs in $x_1, \ldots, x_k$.
  Restliche Variable initialisiert auf 0.
- Ausgabewert $f(n_1, \ldots, n_k)$ am Ende in x_0.

LOOP-Programme berechnen nur *totale* (= total definierte) Funktionen.

LOOP-Simulierbare Konstrukte

neues Konstrukt	**Simulation**
$x_i := x_j$ $x_i := c$	$x_i := x_j + 0$ $x_i := y + c$ (für ein y mit Wert 0)
IF $x = 0$ THEN A END	$y := 1;$ LOOP x DO $y := 0$ END; LOOP y DO A END
$x_i := x_j + x_k$	$x_i := x_j;$ LOOP x_k DO $x_i := x_i + 1$ END
$x_i := x_j * x_k$	$x_i := 0;$ LOOP x_k DO $x_i := x_i + x_j$ END
$x_i := x_j$ DIV x_k $x_i := x_j$ MOD x_k	s. Übung (evtl.) s. Übung (evtl.)

WHILE-Programme

Syntax wie bei LOOP-Programmen, außer dass die WHILE-Schleife an die Stelle der LOOP-Schleife tritt:

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

Semantik der WHILE-Schleife: P wird iteriert solange ausgeführt wie x_i (mit ihrem aktuellen Wert!) ungleich Null ist. Endlosschleife ist möglich.

Konventionen zum Berechnen von Funktionen wie bei LOOP-Programmen.
Berechnung partieller (= partiell definierter) Funktionen ist möglich.

WHILE-simulierbare LOOP-Schleife:

$$\text{LOOP } x \text{ DO } P \text{ END}$$

kann simuliert werden durch

$$y := x;\ \text{WHILE } y \neq 0 \text{ DO } y := y - 1; P \text{ END}\ .$$

Wechselseitige Simulationen

Da die LOOP-Schleife durch die WHILE-Schleife simulierbar ist, gilt der

Satz: Jede LOOP-berechenbare Funktion ist auch WHILE-berechenbar.

Weiter gilt:

Satz: (Beweis mündlich in der Vorlesung)
Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Wir werden (nach Einführung der GOTO-Programme) noch zeigen:
– Jede Turing-berechenbare Funktion ist auch GOTO-berechenbar.
– Jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar.

Folgerung: Turing-Maschinen, WHILE-Programme und GOTO-Programme berechnen die gleiche Klasse von Funktionen.

GOTO-Programme

Syntax: GOTO-Programme haben (bis auf Fehlen von redundanten Marken) die Form

$$M_1 : A_1 \; ; \; M_2 : A_2 \; ; \; \cdots \; ; \; M_k : A_k \quad .$$

Dabei ist A_i eine „Anweisung“ und M_i eine sogenannte „Marke“ (eindeutige Adresse für die Anweisung A_i). Als Anweisungen sind zugelassen:

Wertzuweisungen:	$x_i := x_j \pm c$
unbedingter Sprung:	GOTO M_i
bedingter Sprung:	IF $x_i = c$ THEN GOTO M_j
Stoppanweisung:	HALT

Semantik: — offensichtlich (oder?) —

Konventionen beim Berechnen von Funktionen:
analog zu LOOP- oder WHILE-Programmen.

Simulation von GOTO durch WHILE

Satz: Jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar.

$$M_1 : A_1 \; ; \; M_2 : A_2 \; ; \; \cdots \; ; \; M_k : A_k$$

kann simuliert werden durch

$y := 1;$
WHILE $y \neq 0$ DO
  IF $y = 1$ THEN A'_1 END;
  IF $y = 2$ THEN A'_2 END;
  $\cdots$
  IF $y = k$ THEN A'_k END
END

Idee: Identifiziere M_i mit Nummer i.
Wert von y = Nummer der aktuellen Marke
(bzw. 0 nach Erreichen von HALT).
A'_i realisiert A_i und aktualisiert y.

Simulation von GOTO durch WHILE (fortgesetzt)

A_i	A'_i
$x_k := x_l \pm c$	$x_k := x_l \pm c$; y:=y+1
GOTO M_j	$y := j$
IF $x_k = c$ THEN GOTO M_j	IF $x_k = c$ THEN $y := j$ ELSE $y := y + 1$ END
HALT	$y := 0$

Beobachtung: Die Simulation benötigt nur **eine** WHILE-Schleife.

Simulation von WHILE durch GOTO (fortgesetzt)

Satz: Jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

$$\text{WHILE } x_i \neq 0 \; DO \; P \text{ END}; \ldots$$

kann simuliert werden durch:

M_1: IF $x_i = 0$ THEN GOTO M_2;
P;
GOTO M_1
M_2: ...

Folgerung (Kleene-Normalform für WHILE-Programme):
Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit lediglich einer WHILE-Schleife berechnet werden.

Exkurs: DIV und MOD

DIV (ganzzahliger Quotient) und MOD (kleinster Rest) sind die folgenden Operationen:

$$
\begin{aligned}
x \text{ DIV } y &= \left\lfloor \frac{x}{y} \right\rfloor \ . \\
x \text{ MOD } y &= x - y \left\lfloor \frac{x}{y} \right\rfloor
\end{aligned}
$$

Zum Beispiel:

$$
\begin{aligned}
75 \text{ DIV } 20 &= \left\lfloor \frac{75}{20} \right\rfloor = \lfloor 3.75 \rfloor = 3 \ . \\
75 \text{ MOD } 20 &= 75 - 20 \left\lfloor \frac{75}{20} \right\rfloor = 75 - 20 \cdot 3 = 15 \ .
\end{aligned}
$$

DIV und MOD (fortgesetzt)

CUT und PASTE (Abschneiden und Ankleben) von Ziffern kann mit Hilfe von DIV, MOD und $*, +$ implementiert werden:

CUT und PASTE	Ergebnis	Simulation mit DIV,MOD,+, $*$
CUT(1984)	$198 \vert 4$	$198 = 1984$ DIV $10; 4 = 1984$ MOD 10
PASTE($198 \vert 5$)	1985	$1985 = 198 * 10 + 5$

Verallgemeinerung auf b-näre Zahlendarstellungen (Ziffern aus $\{0, 1, \ldots, b-1\}$):

CUT und PASTE	Ergebnis	Simulation mit DIV,MOD,+, $*$
CUT$\overbrace{(i_1 \cdots i_{p-1} i_p)}^{x}$	$\overbrace{i_1 \cdots i_{p-1}}^{x'} \vert i_p$	$x' = x$ DIV $b; i_p = x$ MOD b
PASTE$(\underbrace{i_1 \cdots i_{p-1}}_{x'} \vert j)$	$\underbrace{i_1 \cdots i_{p-1} j}_{\hat{x}}$	$\hat{x} = x' * b + j$

Exkurs: Konfiguration als Zahlentripel

- Zustandsmenge $Z = \{z_1, \ldots, z_s\}$: z_l hat „Nummer" l.
- Bandalphabet $\Gamma = \{a_1, \ldots, a_m\}$: a_i hat „Nummer" i

Setze $b := |\Gamma| + 1$. Eine Konfiguration

$$\overbrace{a_{i_1} \cdots a_{i_p}}^{x} z_l \overbrace{a_{j_1} \cdots a_{j_q}}^{y}$$

(mit z_l als aktuellem Zustand, $a_{i_1} \cdots a_{i_p}$ als (nichtleere) Bandinschrift links vom Kopf und $a_{j_1} \cdots a_{j_q}$ als (nichtleere) Bandinschrift ab Kopfposition) kann als b-näres Zahlentripel (x, y, z) kodiert werden:

$$z = l \ , \ x = \sum_{\rho=1}^{p} i_\rho b^{p-\rho} \ , \ y = \sum_{\rho=1}^{q} j_\rho b^{\rho-1}$$

(Nummern der Symbole sind gleichsam die Ziffern der Zahlendarstellung.)

Simulation von Turing-Maschine durch GOTO

Satz Jede Turing-berechenbare Funktion ist auch GOTO-berechenbar.

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

Aufbau der Simulation:

Phase 1 (Vorbereitung): Berechne aus den Werten $n_1, \ldots, n_k$ der Eingabevariablen $x_1, \ldots, x_k$ das Zahlentripel (x, y, z), welches die Startkonfiguration $z_0\mathrm{bin}(n_1)\# \cdots \#\mathrm{bin}(n_k)$ von M repräsentiert.

Phase 2 (Schritt-für-Schritt Simulation): Solange M nicht stoppt, berechne aus dem Zahlentripel (x, y, z) der aktuellen Konfiguration das Zahlentripel für die direkte Folgekonfiguration.

Phase 3 (Nachbereitung) Extrahiere aus dem Zahlentripel (x, y, z), das eine Endkonfiguration $z_e\mathrm{bin}(f(n_1, \ldots, n_k))$ von M repräsentiert, den Ausgabewert $f(n_1, \ldots, n_k)$ und belege damit die Ausgabevariable x_0.

Details zur Phase 2

„(l,j)-Aktualisierung“ von (x,y,z) bezeichne die Aktualisierung, die erforderlich ist, wenn M im Zustand z_l Symbol a_j liest (und die durch $\delta(z_l,a_j)$ beschriebene Aktion ausführt).

Das GOTO-„Unterprogramm“ (plus zugehöriger Marke), das die (l,j)-Aktualisierung durchführt (und i.A. aus mehreren Anweisungen besteht) bezeichnen wir mit

$$M_{l,j} : A_{l,j} \quad .$$

Wir präsentieren im Folgenden:

- Das Grundgerüst eines Teilprogrammes P_2, das die Verzweigung zum richtigen Unterprogramm gewährleistet,
- ein Beispiel-Unterprogramm.

Verzweigung zum richtigen Unterprogramm

Programmstück $M_2 : P_2$ für Phase 2 hat folgende Form:

M_2: $a := y$ MOD b; (CUT-Operation liefert Symbol unterm Lesekopf)
IF $z = 1$ AND $a = 1$ THEN GOTO $M_{1,1}$;
IF $z = 1$ AND $a = 2$ THEN GOTO $M_{1,2}$;
usw.
— alle sm Zustands/Symbolkombinationen —
usw.
IF $z = s$ AND $a = m$ THEN GOTO $M_{s,m}$;
$M_{1,1}$: $A_{1,1}$; GOTO M_2;
$M_{1,2}$: $A_{1,2}$; GOTO M_2;
usw.
— alle sm Zustands/Symbolkombinationen —
usw.
$M_{s,m}$: $A_{s,m}$; GOTO M_2;

Ein Beispiel-Unterprogramm

Programmzeile

$$\delta(z_l, a_j) = (z_{l'}, a_{j'}, L)$$

würde durch folgendes Unterprogramm realisiert:

$M_{l,j}$: $z := l'$; (Aktualisiere z mit Nummer des neuen Zustands)

$y := y$ DIV b (CUT)

$y := b * y + j'$ (PASTE)

Kommentar: führt $y = \langle j\ j_2\ \cdots\ j_q\rangle_b$ in $y = \langle j'\ j_2\ \cdots\ j_q\rangle_b$ über

$y := b * y + (x$ MOD $b)$ (PASTE)

$x := x$ DIV b (CUT)

Kommentar: führt $x = \langle i_1\ \cdots\ i_{p-1}\ i_p\rangle_b$, $y = \langle j'\ j_2\ \cdots\ j_q\rangle_b$
in $x = \langle i_1\ \cdots\ i_{p-1}\rangle_b$, $y = \langle i_p\ j'\ j_2\ \cdots\ j_q\rangle$ über

Falls $z_{l'}$ ein Endzustand wäre, dann würde dieses Unterprogramm noch um die Anweisung „GOTO M_3“ (Eintritt in Phase 3) erweitert.