Theorie der Berechenbarkeit (Teil 1)

Hans U. Simon (RUB)

Email: simon@lmi.rub.de

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

Intuitive und formale Berechenbarkeit
Church'sche These

Partiell definierte Funktionen

Wir betrachten im Folgenden partiell definierte Funktionen der Form

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

Für x außerhalb des Definitionsbereiches gilt dann

$$f(x) = \text{„undefiniert“} \ .$$

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

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

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

Turing–Berechenbarkeit

Eine Funktion

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

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

- Falls $f(x) = y$, dann gilt $z_0 x \vdash^* zy$ für eine Endkonfiguration zy.
- Falls $f(x) =$ „undefiniert“, dann erreicht M bei der Rechnung auf Eingabe x keine Stoppkonfiguration (Endlosrechnung).

Turing–Berechenbarkeit (fortgesetzt)

Die Turing–Berechenbarkeit von Funktionen der Form

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

ist analog definiert, wobei wir im Falle von

$$f(x) = y \text{ mit } x = (n_1, n_2, \ldots, n_k)$$

anstelle von $z_0 x \vdash^* zy$ nun

$$z_0 \text{bin}(n_1)\#\text{bin}(n_2)\#\cdots\#\text{bin}(n_k) \vdash^* z\text{bin}(y)$$

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

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 gestoppt wird,
  - bei Nicht–Auffinden einer Ableitung eine unendliche Schleife betreten wird.
- Dann wird χ'_L berechnet durch die deterministische Simulation M' von M.

Eine „höhere Programmiersprache“ für DTMs

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: Mehrspurenmaschinen 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

Ziel: Entwurf von DTMs zur Ausführung von Befehlen einer „höheren Programmiersprache“ (mit bedingten Anweisungen, while-Schleifen etc.)

Methode: Baukastensystem

Veränderung des Inhaltes von einem der Bänder

- 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

„Band i := Band i $\dot{-}$1“

„Band i := Band j

„Band i := 0

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

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

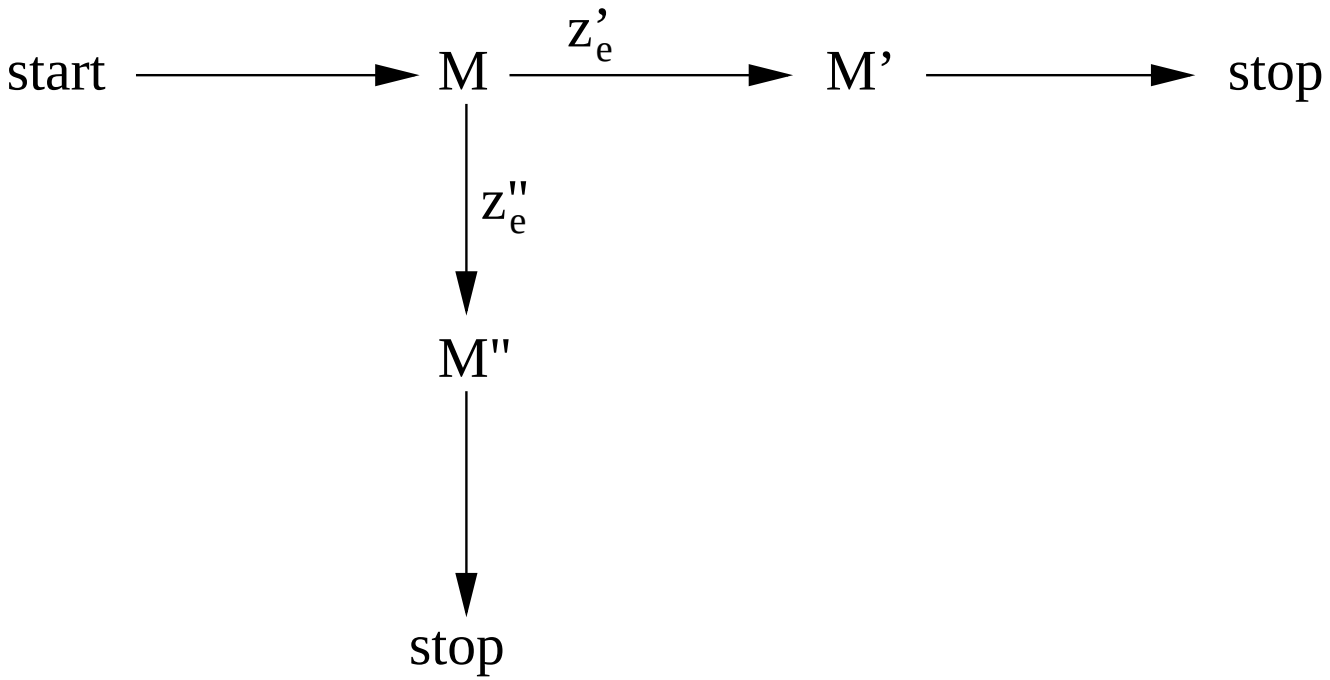
addiert zu einer gegebenen natürlichen Zahl die Konstante 3 hinzu.

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

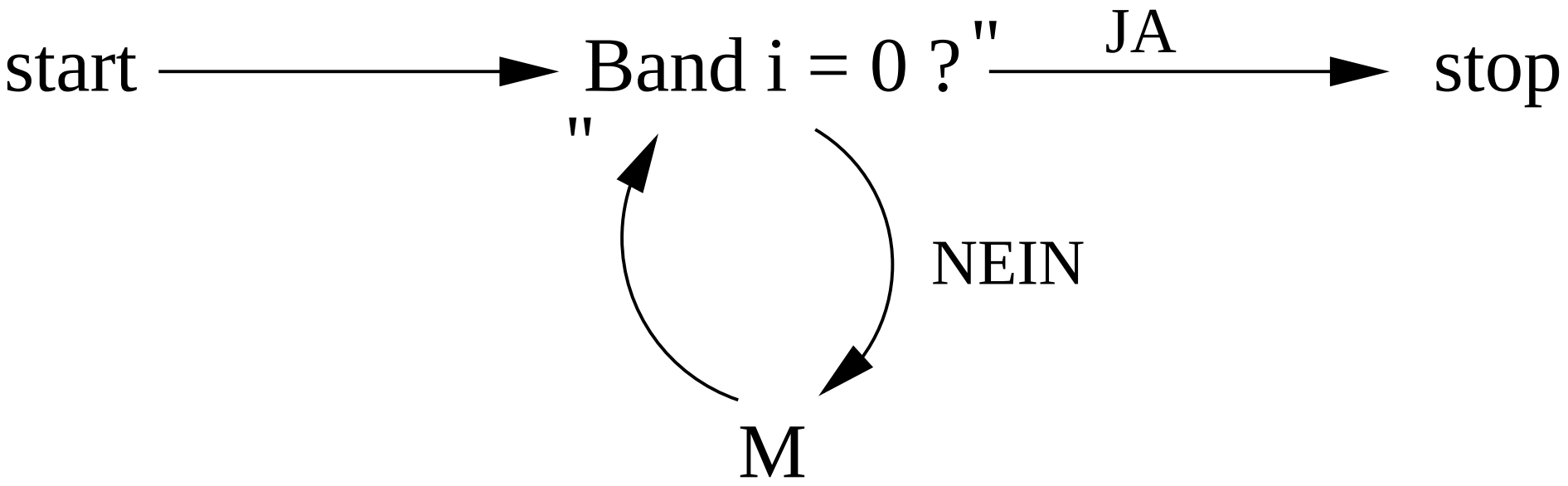
Statt „Band=0?“(i) schreiben wir einfach

„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.

LOOP- WHILE- und GOTO-Programme

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 $f : \mathbb{N}^k \to \mathbb{N}$ 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 dieselbe Klasse von Funktionen.

GOTO-Programme

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

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

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.

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

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 (um IF-THEN oder LOOP-Anweisungen erweitertes) WHILE-Programm mit lediglich einer WHILE-Schleife berechnet werden.

Exkurs: DIV und MOD

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

$$x \text{ DIV } y = \left\lfloor \frac{x}{y} \right\rfloor \ .$$

$$x \text{ MOD } y = x - y \left\lfloor \frac{x}{y} \right\rfloor$$

Zum Beispiel:

$$75 \text{ DIV } 20 = \left\lfloor \frac{75}{20} \right\rfloor = \lfloor 3.75 \rfloor = 3 \ .$$

$$75 \text{ MOD } 20 = 75 - 20 \left\lfloor \frac{75}{20} \right\rfloor = 75 - 20 \cdot 3 = 15 \ .$$

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)	$198\|4$	$198 = 1984 \text{ DIV } 10; 4 = 1984 \text{ MOD } 10$
PASTE(198|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$(\overbrace{i_1 \cdots i_{p-1} i_p}^{x})$	$\overbrace{i_1 \cdots i_{p-1}}^{x'} \| i_p$	$x' = x \text{ DIV } b; i_p = x \text{ MOD } b$
PASTE$(\underbrace{i_1 \cdots i_{p-1}}_{x'} \| j)$	$\underbrace{i_1 \cdots i_{p-1} j}_{\hat{x}}$	$\hat{x} = x' * b + j$

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

$$\overbrace{a_{i_1} \cdots a_{i_p}}^{x} z_l \overbrace{a_{j_1} \cdots a_{j_q}}^{y} \qquad (x, y \text{ } b\text{-när kodiert})$$

(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 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 $z_0\text{bin}(n_1)\#\cdots\#\text{bin}(n_k)$ 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 $z_e\text{bin}(f(n_1, \ldots, n_k))$ 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.

Primitiv rekursive und μ-rekursive Funktionen

Primitiv rekursive Funktionen

Basisfunktionen:

konstante Funktionen:	$c(n_1, \ldots, n_k) = c$
Projektionen:	$\pi_i^k(n_1, \ldots, n_k) = n_i$
Nachfolgerfunktion:	$s(n) = n + 1$

Beachte: $\pi_1^1(n) = n$ ist die identische Funktion.

Die Basisfunktionen sind primitiv rekursiv sowie alle Funktionen, die sich induktiv wie folgt ergeben:

Einsetzungsschema Jede Funktion, die durch „Komposition“ von primitiv rekursiven Funktionen entsteht, ist primitiv rekursiv.

primitives Rekursionsschema Jede Funktion, die sich durch „primitive Rekursion (Induktion)“ aus primitiv rekursiven Funktionen ergibt, ist primitiv rekursiv.

Komposition und primitive Rekursion

Das Einsetzungsschema:

Gegeben seien primitiv rekursive Funktionen $h : \mathbb{N}^r \to \mathbb{N}$ und $g_i : \mathbb{N}^k \to \mathbb{N}$ für $i = 1, \ldots, r$. Dann ist auch die Funktion $f : \mathbb{N}^k \to \mathbb{N}$ mit

$$f(x) = h\left(g_1(x), \ldots, g_r(x)\right)$$

primitiv rekursiv.

Das Schema der primitiven Rekursion:

Gegeben seien primitiv rekursive Funktionen $g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$. Dann ist die folgende (induktiv definierte) Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv:

$$\begin{aligned} f(0, x) &= g(x) \\ f(n+1, x) &= h(f(n, x), n, x) \end{aligned}$$

Vertauschen, Identifikation und Konstantsetzung von Variablen

In Verbindung mit den Projektionsabbildungen π_i^k und den konstanten Funktionen kann das Einsetzungsschema benutzt werden, um Variablen zu vertauschen, zu identifizieren oder konstant zu setzen, ohne die Klasse der primitiv rekursiven Funktionen zu verlassen.

Beispiel: Wenn $f(u,v,w,x,y)$ primitiv rekursiv ist, dann ist auch

$$\begin{aligned} g(a,b,c) &= f(b,b,c,a,1) \\ &= f\left(\pi_2^3(a,b,c), \pi_2^3(a,b,c), \pi_3^3(a,b,c), \pi_1^3(a,b,c), 1\right) \end{aligned}$$

primitiv rekursiv.

Addition und Multiplikation sind primitiv rekursiv

Addition Nutze aus, dass $(n+1)+x = (n+x)+1$:

$$\begin{aligned} \mathrm{add}(0,x) &= x \\ \mathrm{add}(n+1,x) &= s(\mathrm{add}(n,x)) \end{aligned}$$

Dies entspricht dem primitiven Rekursionsschema mit $g = \pi_1^1$ (identische Funktion) und $h = s \circ \pi_1^3$ wegen

$$s(\mathrm{add}(n,x)) = s(\pi_1^3(\mathrm{add}(n,x),n,x)) \ .$$

Ab jetzt geben wir g und h bei Verwendung des primitiven Rekursionsschema nicht immer explixit an.

Multiplikation Nutze aus, dass $(n+1)x = nx + x$:

$$\begin{aligned} \mathrm{mult}(0,x) &= 0 \\ \mathrm{mult}(n+1,x) &= \mathrm{add}(\mathrm{mult}(n,x),x) \end{aligned}$$

Modifizierte Differenz ist primitiv rekursiv

Funktion $\mathrm{sub}(x,y)$ soll die modifizierte Differenz

$$x \dot{-} y = \max\{0, x-y\}$$

darstellen. Funktion

$$u(n) = \max\{0, n-1\}$$

ist entsprechend die modifizierte Vorgängerfunktion. Wegen

$$\begin{aligned} u(0) &= 0 \\ u(n+1) &= n \end{aligned}$$

ist u primitiv rekursiv. Wegen

$$\begin{aligned} \mathrm{sub}(x,0) &= x \\ \mathrm{sub}(x,y+1) &= u(\mathrm{sub}(x,y)) \end{aligned}$$

ist dann auch sub primitiv rekursiv.

Exkurs: Bijektion zwischen $\mathbb{N}^2$ und $\mathbb{N}$

Folgende Funktion $c(x,y)$, unten angegeben als Matrix, liefert eine (bijektive) Abzählung aller Paare $(x,y) \in \mathbb{N} \times \mathbb{N}$:

0	2	5	9	14	$\cdots$
1	4	8	13	19	$\cdots$
3	7	12	18	25	$\cdots$
6	11	17	24	32	$\cdots$
10	16	23	31	40	$\cdots$

$\cdots$

Nach diesem Schema gilt (Denksportaufgabe!)

$$c(x,y) = \binom{x+y+1}{2} + x \ .$$

Primitive rekursive Zahlkodierung von Tupeln

Wegen

$$\binom{0}{2} = 0$$

$$\binom{n+1}{2} = \binom{n}{2} + n$$

ist die Funktion $\binom{n}{2}$ primitiv rekursiv. Mit dem Einsetzungsschema folgt dann leicht, dass erstens

$$c(x,y) = \binom{x+y+1}{2} + x$$

und zweitens

$$\langle n_0, n_1, \ldots, n_k \rangle = c(n_0, c(n_1, \ldots, c(n_k, 0)\ldots))$$

primitiv rekursiv ist. Abbildung $\langle \cdots \rangle$ bettet $\mathbb{N}^{k+1}$ (injektiv) in $\mathbb{N}$ ein. (Kodierung eines Zahlentupels durch **eine** Zahl).

Rückgewinnung des Zahlentupels

Wie können wir aus der Zahl $n = \langle n_0, n_1, \ldots, n_k \rangle$ das Tupel $(n_0, n_1, \ldots, n_k)$ zurückgewinnen (mathematisch die Frage nach der Umkehrfunktion)?

Als bijektive Abbildung hat $c(x, y)$ eine Umkehrfunktion (e, f) mit

$$e(c(x, y)) = x \text{ und } f(c(x, y)) = y \ .$$

Wegen

$$n = \langle n_0, n_1, \ldots, n_k \rangle = c(n_0, c(n_1, \ldots, c(n_k, 0) \ldots))$$

ergibt sich

$$\begin{array}{lcccl} d_0(n) & := & e(n) & = & n_0 \\ d_1(n) & := & e(f(n)) & = & n_1 \\ & & \ldots & & \\ d_k(n) & := & e(\underbrace{f(f(\cdots f}_{k-mal}(n)\cdots)) & = & n_k \end{array}$$

Primitive Rekursivität der Dekodierung

Hilfssatz: Funktionen e, f mit

$$e(c(x,y)) = x \text{ und } f(c(x,y)) = y$$

sind primitiv rekursiv.
Den etwas kniffligen Beweis lassen wir aus.

Mit dem Einsetzungsschema ergibt sich dann sofort die
Folgerung Funktionen $d_0, d_1, \ldots, d_k$ mit

$$d_i(\langle n_0, \ldots, n_k \rangle) = n_i$$

für $i = 0, 1 \ldots, k$ sind primitiv rekursiv.

Primitive Rekursivität LOOP–berechenbarer Funktionen

Wir betrachten ein LOOP-Programm P, das eine Teilmenge der Variablen $x_0, x_1, \ldots, x_k$ verwendet. Das Verhalten von P ist vollständig beschrieben durch die Funktion $g_P : \mathbb{N} \to \mathbb{N}$ mit:

$$g_P(\langle \overbrace{a_0, a_1, \ldots, a_k}^{Anfangsbelegung} \rangle) = \langle \overbrace{b_0, b_1, \ldots, b_k}^{Endbelegung} \rangle \ .$$

Satz Funktion g_P zu einem LOOP-Programm P ist primitiv rekursiv.

Folgerung Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

Denn: Wenn P mit Variablen $x_0, x_1, \ldots, x_k$ die Funktion $f : \mathbb{N}^r \to \mathbb{N}$ berechnet, dann gilt

$$f(n_1, \ldots, n_r) = d_0(g_P(\langle 0, n_1, \ldots, n_r, \underbrace{0, \ldots, 0}_{(k-r)-mal} \rangle)) \ .$$

Primitive Rekursivität von g_P

Q durchlaufe die „Teilprogramme“ von P, beginnend bei einfachen Wertzuweisungen und fortschreitend zu zunehmend komplexeren Teilprogrammen. Das zuletzt durchlaufene „Teilprogramm“ ist P selbst.

zu zeigen: Für jedes Teilprogramm Q ist g_Q eine primitiv rekursive Funktion.

Fall 1 Q hat die Form $x_i := x_j \pm c$.
Dann gilt $g_Q(\langle a_0, a_1, \ldots, a_k \rangle) = \langle b_0, b_1, \ldots, b_k \rangle$ mit

$$b_l = \begin{cases} a_l & \text{falls } l \neq i \\ a_j \pm c & \text{falls } l = i \end{cases} .$$

Wegen $a_l = d_l(\langle a_0, a_1, \ldots, a_k \rangle)$ folgt die primitive Rekursivität von g_Q unmittelbar aus dem Einsetzungsschema.

Komplexere Teilprogramme

Fall 2 Q hat die Form „$Q'; Q''$“
Da dann Q', Q'' zuvor schon betrachtet wurden ist die primitive Rekursivität von $g_{Q'}$ und $g_{Q''}$ schon geklärt. Wegen $g_Q = g_{Q''} \circ g_{Q'}$ ist nach dem Einsetzungsschema auch g_Q primitiv rekursiv.

Komplexere Teilprogramme (fortgesetzt)

Fall 3 Q hat die Form „LOOP x_i DO Q' END“.
Da dann Q' zuvor schon betrachtet wurde, ist die primitive Rekursivität von $g_{Q'}$ bereits geklärt. Betrachte Hilfsfunktion

$$h(n,x) = \underbrace{g_{Q'}(g_{Q'}(\cdots(g_{Q'}}_{n-mal}(x)\cdots) \ .$$

Mit dem Schema der primitiven Rekursion

$$\begin{aligned} h(0,x) &= x \\ h(n+1,x) &= g_{Q'}(h(n,x)) \end{aligned}$$

folgt die primitive Rekursivität von h. Die primitive Rekursivität von g_Q ergibt sich dann mit dem Einsetzungsschema aus

$$g_Q(\langle a_0,a_1,\ldots,a_k\rangle) = h(a_i,\langle a_0,a_1,\ldots,a_k\rangle) \ .$$

LOOP-Berechenbarkeit von primitiv rekursiven Funktionen

Induktion über den Aufbau von primitiv rekursiven Funktionen:

1. Die Basisfunktionen (Konstanten, Projektionen, Nachfolgerfunktion) sind offensichtlich LOOP-berechenbar.

2. Betrachte eine Funktion f der Form

$$f(x) = h(g_1(x), \ldots, g_r(x)) \ ,$$

wobei gemäß Induktionsvoraussetzung $h, g_1, \ldots, g_r$ LOOP-berechenbar sind. Dann kann ein LOOP-Programm $f(x)$ nach folgendem Schema berechnen:

$$y_1 := g_1(x) \ ; \ \cdots \ ; \ y_r := g_r(x) \ ; \ x_0 := h(y_1, \ldots, y_r)$$

LOOP-Berechenbarkeit von primitiv rekursiven Funktionen (fortgesetzt)

3. Betrachte eine Funktion f der Form

$$f(0,x) = g(x) \text{ und } f(n+1,x) = h(f(n,x),n,x) \ ,$$

wobei gemäß Induktionsvoraussetzung g und h LOOP-berechenbar sind. Dann kann ein LOOP-Programm $f(n,x)$ nach folgendem Schema berechnen:

$$z := 0 \ ; \ x_0 := g(x) \ ; \ \text{LOOP } x_1 \text{ DO } z := z+1 \ ; \ x_0 := h(x_0,z,x) \text{ END}$$

Kommentare:

- Variable x_1 enthält den Eingabeparameter n.
- Nach i Iterationen hat x_0 den Wert $f(i,x)$.

Hauptresultate

Es ergibt sich der

Satz: Die Klasse der primitiv rekursiven Funktionen stimmt mit der Klasse LOOP-berechenbaren Funktionen über ein.

Wir werden die Klasse der primitiv rekursiven Funktionen durch Einführung des sogenannten μ-Operators zur Klasse der μ-rekursiven Funktionen erweitern.

Ziel: Die Klasse der μ-rekursiven Funktionen stimmt mit der Klasse WHILE-berechenbaren Funktionen über ein.

Der μ-Operator

Für eine gegebene (evtl. partielle) Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$ bezeichne $\mu f : \mathbb{N}^k \to \mathbb{N}$ die Funktion

$$\mu f(x) := \min\{n |\ f(n,x) = 0, \forall m < n : f(m,x) \text{ ist definiert}\}$$

verbunden mit der Konvention $\min \emptyset =$„undefiniert“.

Intuitive Bemerkung Falls $\mu f(x)$ definiert ist, dann liefert diese Funktion eine Art „kleinste Nullstelle“ für die Funktion $f(n,x)$ (aufgefasst als Funktion in n).

Definition Die Klasse der μ-rekursiven Funktionen ist die kleinste Klasse von (evtl. partiellen) Funktionen, die die Basisfunktionen enthält und abgeschlossen ist unter Einsetzung, primitiver Rekursion und der Anwendung des μ-Operators.

μ–Rekursivität WHILE–berechenbarer Funktionen

Wir betrachten ein WHILE-Programm P, das eine Teilmenge der Variablen $x_0, x_1, \ldots, x_k$ verwendet. Das Verhalten von P ist vollständig beschrieben durch die (evtl. partielle) Funktion $g_P : \mathbb{N} \to \mathbb{N}$ mit:

$$g_P(\langle \overbrace{a_0, a_1, \ldots, a_k}^{Anfangsbelegung} \rangle) = \langle \overbrace{b_0, b_1, \ldots, b_k}^{Endbelegung} \rangle$$

verbunden mit der Konvention, dass g_P undefiniert ist, wenn P nicht terminiert.

Satz Funktion g_P zu einem WHILE-Programm P ist μ-rekursiv.

Folgerung Jede WHILE-berechenbare Funktion ist μ-rekursiv.

μ-Rekursivität von g_P

Q durchlaufe die „Teilprogramme“ von P, beginnend bei einfachen Wertzuweisungen und fortschreitend zu zunehmend komplexeren Teilprogrammen. Das zuletzt durchlaufene „Teilprogramm“ ist P selbst.

zu zeigen: Für jedes Teilprogramm Q ist g_Q eine μ rekursive Funktion.

Die Fälle der Wertzuweisung und der Komposition zweier WHILE-Programme lassen sich abhandeln wie bei der analogen Überlegung für LOOP-Programme. Wesentlich neu ist nur der Fall der WHILE-Anweisung.

WHILE-Anweisung und μ-Operator

Neuer Fall Q hat die Form „WHILE $x_i \neq 0$ DO Q' END“.
Da dann Q' zuvor schon betrachtet wurde, ist die μ-Rekursivität von $g_{Q'}$ bereits geklärt. Betrachte die μ-rekursive (!) Hilfsfunktion

$$h(n,x) = \underbrace{g_{Q'}(g_{Q'}(\cdots(g_{Q'}}_{n-mal}(x)\cdots) \ .$$

Beachte: $d_i(h(n,x))$ ist der Wert der Variablen x_i nach n Ausführungen von Q. Die WHILE-Anweisung führt daher Q insgesamt

$$\min\{n|\ d_i(h(n,x)) = 0\} = \mu(d_i \circ h)(x)$$

mal aus (sofern sie terminiert). Mit

$$g_Q(x) = h\left(\mu(d_i \circ h)(x), x\right)$$

ergibt sich die μ-Rekursivität von g_Q.

WHILE-Berechenbarkeit von μ-rekursiven Funktionen

Induktion über den Aufbau von μ-rekursiven Funktionen: Wegen der Analogie zur LOOP-Berechenbarkeit von primitiv rekursiven Funktionen beschränken wir uns auf den

Neuen Fall: Betrachte eine (evtl. partielle) Funktion der Form

$$\mu f(x) = \min\{n |\ f(n,x) = 0, \forall m < n : f(m,x) \text{ ist definiert}\} \ .$$

Gemäß Induktionsvoraussetzung ist $f(n,x)$ WHILE-berechenbar. Dann können wir $\mu f(x)$ nach folgendem Schema berechnen:

$$x_0 := 0 \ ; \ y := f(0,x) \ ; \ \text{WHILE } y \neq 0 \text{ DO } x_0 := x_0 + 1 \ ; \ y := f(x_0,x) \text{ END}$$

Kommentar: Sofern $\mu f(x)$ definiert ist, enthält Variable x_0 am Ende die kleinste Zahl $n \in \mathbb{N}$ mit $f(n,x) = 0$ (kleinste Nullstelle).

Die Ackermannfunktion

Eine Frage zu Anfang

Ist jede intuitiv berechenbare totale (= total definierte) Funktion

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

LOOP–berechenbar?

Im Jahre 1928 präsentierte Ackermann ein Gegenbeispiel.

Seine Idee: Entwerfe eine Funktion, die schneller wächst als jede LOOP-berechenbare Funktion.

Die Ackermannfunktion

Betrachte folgende induktiv definierte Funktion $a : \mathbb{N}^2 \to \mathbb{N}$:

$$\begin{aligned} a(0,y) &= y+1 \text{ für alle } y \geq 0 \\ a(x,0) &= a(x-1,1) \text{ für alle } x \geq 1 \\ a(x,y) &= a(x-1,a(x,y-1)) \text{ für alle } x,y \geq 1 \end{aligned}$$

Die Definition besteht zwar nur aus drei Gleichungen, ist aber nicht ganz leicht zu durchschauen.

Ackermannfunktion (fortgesetzt)

Per „Salami–Taktik“ zerlegen wir $a(x,y)$ in „Scheiben“ $A_x : \mathbb{N} \to \mathbb{N}$:

$$A_x(y) := a(x,y)$$

und „schnuppern“ an $A_0, A_1, A_2, \ldots$.

Offensichtlich gilt $A_0(y) = y+1$. Mit Induktion ergibt sich relativ leicht (s. Übung):

$$\begin{aligned} A_1(y) &= y+2 \\ A_2(y) &= 2y+3 \\ A_3(y) &= 2^{y+3}-3 \\ A_4(y) &= \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3 \ Zweien} - 3 \end{aligned}$$

Die Folge beginnt harmlos, nimmt aber dann schnell an Fahrt auf.

Macht die Definition von Ackermann Sinn ?

Wir beweisen durch „doppelte Induktion“, dass alle Funktionen A_x Werte aus $\mathbb{N}$ als Ergebnis liefern.

Induktionsanfang: $A_0(y) = y + 1 \in \mathbb{N}$ für alle $y \in \mathbb{N}$.

Induktionsvoraussetzung: $A_{x-1}(y) \in \mathbb{N}$ für alle $y \in \mathbb{N}$.

Induktionsschritt: Zum Studium von A_x erfolgt erneut eine vollständige Induktion (diesmal nach y):

Induktionsanfang: $A_x(0) = a(x,0) = a(x-1,1) = A_{x-1}(1) \in \mathbb{N}$.

Induktionsvoraussetzung: $A_x(y-1) \in \mathbb{N}$.

Induktionsschritt:

$$A_x(y) = a(x,y) = a(x-1, a(x,y-1)) = \overbrace{A_{x-1}(\underbrace{A_x(y-1)}_{\in\mathbb{N}})}^{\in\mathbb{N}}$$

Folgerung: $a(x,y) = A_x(y) \in \mathbb{N}$ für alle $x, y \in \mathbb{N}$.

Ist die Ackermannfunktion berechenbar ?

Wir berechnen $a(3,0)$ mit Hilfe eines Kellerspeichers (Stapels):

										0	1				
				0	1				1	1	0	2			
		0	1	1	0	2		2	1	0	0	0	3		
0	1	2	1	0	0	0	3	1	0	0	0	0	0	4	
3	2	1	1	1	1	1	1	0	0	0	0	0	0	0	5

- Da wir mit „brain power“ $A_3(y) = 2^{y+3} - 3$ herausgefunden hatten, hätten wir das Ergebnis auch einfacher bestimmen können:

$$a(3,0) = A_3(0) = 2^{0+3} - 3 = 8 - 3 = 5 \ .$$

- Das Stapelverfahren (langsam wie es ist) funktioniert aber für alle möglichen Eingaben $(x,y) \in \mathbb{N}^2$.

Die allgemeine Vorgehensweise beim Stapelverfahren

Wir erinnern an die Definitionsgleichungen für die Ackermannfunktion:

$$\begin{aligned} a(0,y) &= y+1 \text{ für alle } y \geq 0 \\ a(x,0) &= a(x-1,1) \text{ für alle } x \geq 1 \\ a(x,y) &= a(x-1,a(x,y-1)) \text{ für alle } x,y \geq 1 \end{aligned}$$

Das Stapelverfahren interpretiert die zwei obersten Operanden x, y des Stapel (y oben) als $a(x,y)$ und manipuliert den Stapel wie folgt:

- Ersetze $0, y$ durch $y+1$ (Stapel wird niedriger).
- Für $x \geq 1$ ersetze $x, 0$ durch $x-1, 1$.
- Für $x, y \geq 1$ ersetze x, y durch $x-1, x, y-1$ (Stapel wird höher).

Dadurch wird $a(x,y)$ stets gemäß der Definitionsgleichungen ausgewertet.

Berechenbarkeit der Ackermannfunktion

Das Stapelverfahren ist leicht auf einer Mehrband–DTM implementierbar, die eines ihrer Bänder als Kellerspeicher verwendet.

Folgerung: Die Ackermannfunktion ist berechenbar.

In der Folge sagen wir einfach „berechenbar“ statt „Turing–berechenbar“ (oder „WHILE“– bzw. „GOTO–berechenbar“).

Monotonie–Eigenschaften der Ackermannfunktion

Die folgenden Ungleichungen gelten für alle $x, y \in \mathbb{N}$:

$$y < a(x,y) \quad (1)$$

$$a(x,y) < a(x,y+1) \quad (2)$$

$$a(x,y+1) \leq a(x+1,y) \quad (3)$$

$$a(x,y) < a(x+1,y) \quad (4)$$

Insbesondere ist $a(x,y)$ streng monoton wachsend in x und y.

Beweis der 1. Ungleichung

Zum Beweis von $a(x,y) > y$ verwenden wir doppelte Induktion:

Induktionsanfang: $a(0,y) = y+1 > y$ für alle $y \geq 0$.

Induktionsvoraussetzung: $a(x-1,y) > y$ für alle $y \geq 0$.

Induktionsschritt: Analyse von $a(x,y)$ erfolgt mit Induktion nach y.

Induktionsanfang: $a(x,0) = a(x-1,1) > 1 > 0$.

Induktionsvoraussetzung: $a(x,y-1) > y-1$ und somit $a(x,y-1) \geq y$.

Induktionsschritt: für alle $y \geq 0$ gilt dann

$$a(x,y) = a(x-1,a(x,y-1)) > a(x,y-1) \geq y \ .$$

Beweis der 2. Ungleichung

Der Beweis von $a(x, y+1) > a(x, y)$ erfolgt durch eine Fallunterscheidung:

Fall 1: $x = 0$.

$$a(0, y+1) = y + 2 > y + 1 = a(0, y)$$

Fall 2: $x \geq 1$.

$$a(x, y+1) = a(x-1, a(x, y)) \overset{(1)}{>} a(x, y)$$

Beweis der 3. Ungleichung

Zum Beweis von $a(x+1,y) \geq a(x,y+1)$ verwenden wir Induktion nach y:

Induktionsanfang: $a(x+1,0) = a(x,1)$.

Induktionsvoraussetzung: $a(x+1,y-1) \geq a(x,y)$.

Induktionsschritt: Nun ergibt sich $a(x+1,y) \geq a(x,y+1)$ wie folgt:

$$
\begin{aligned}
a(x+1,y) &= a(x,\underbrace{a(x+1,y-1)}_{\overset{IV}{\geq} a(x,y)}) && \text{Definition der Ackermannfunktion} \\
&\geq a(x,\underbrace{a(x,y)}_{\overset{(1)}{\geq} y+1}) && \text{Monotonie in } y \\
&\geq a(x,y+1) && \text{Monotonie in } y
\end{aligned}
$$

Beweis der 4. Ungleichung

Die Ungleichung $a(x,y) < a(x+1,y)$ ergibt sich direkt wie folgt:

$$a(x,y) \overset{(2)}{<} a(x,y+1) \overset{(3)}{\leq} a(x+1,y)$$

Wachstumsfunktion zu LOOP–Programmen

Betrachte ein LOOP–Programm P. P führt Anfangswerte $n_0, n_1, n_2, \ldots$ der Variablen in Endwerte über, die wir mit $n'_0, n'_1, n'_2, \ldots$ bezeichnen.

Folgende Funktion misst gewissermaßen, wie stark die Werte der Variablen durch Ausführung von P wachsen könnnen:

$$f_P(n) := \max \left\{ \sum_{i \geq 0} n'_i \;\middle|\; \sum_{i \geq 0} n_i \leq n \right\}$$

In Worten: $f_P(n)$ ist die größtmögliche Summe der Variablenendwerte, wenn die Summe der Variablenanfangswerte durch n beschränkt ist.

Wachstumsfunktion (fortgesetzt)

Wenn P eine Funktion $g : \mathbb{N} \to \mathbb{N}$ berechnet, dann gilt offensichtlich

$$g(n) \leq f_P(n) \ ,$$

da die Anfangskonfiguration zu Eingabe n mit einer speziellen Wahl der Variablenanfangswerte, nämlich

$$n_0 = 0, n_1 = n \text{ und } n_i = 0 \text{ für alle } i \geq 2 \ ,$$

operiert, welche bei dem Maximum in der Definition von f_P berücksichtigt wird.

Verhältnis von Ackermannfunktion und LOOP–Programmen

Schlüsselresultat: Für jedes LOOP–Programm P gilt:

$$\exists k \geq 0, \forall n \geq 0 : f_P(n) < A_k(n)$$

Die einem LOOP–Programm P zugeordnete Funktion f_P wächst demnach nicht schneller als die k–te „Schicht“ $A_k(\cdot)$ der Funktion $a(\cdot, \cdot)$.

Folgerung: Die Ackermannfunktion ist nicht LOOP–berechenbar.

Widerspruchsbeweis zur Folgerung

(heuchlerische) Annahme: $a(x, y)$ ist LOOP–berechenbar.

- Dann ist auch $g(n) := a(n, n)$ LOOP–berechenbar, sagen wir mit LOOP–Programm P.
- Es gilt dann $g(n) \leq f_P(n)$ für alle $n \geq 0$.
- Wähle dann k so aus, dass $f_P(n) < A_k(n)$ für alle $n \geq 0$.
- Für $n = k$ ergibt sich nun ein Widerspruch:

$$g(k) \leq f_P(k) < A_k(k) = a(k, k) = g(k)$$

Beweis des Schlüsselresultates

Der Beweis ist aufgebaut wie folgt:

- Reduktion des Schlüsselresultates auf sogenannte „einfache“ LOOP–Programme.
- Beweis des Schlüsselresultates für einfache LOOP–Programme mit Hilfe von struktureller Induktion.

Einfache LOOP–Programme

Ein LOOP–Programm heißt einfach **gdw** es die folgenden Bedingungen erfüllt:

- Anweisungen der Form „$x_i := x_j \pm c$“ verwenden nur Konstanten $c \in \{0, 1\}$.
- Bei Anweisungen der Form „LOOP x_i DO Q END“ wird Variable x_i nicht innerhalb von Q verwendet.

Jedes LOOP–Programm ist in ein äquivalents einfaches LOOP–Programm transformierbar:

- Eine Anweisung „$x_i := x_j \pm c$“ mit $c \geq 2$ kann durch c–fache Hintereinanderausführung von „$x_i := x_j \pm 1$“ simuliert werden.
- Eine Anweisung der Form „LOOP x_i DO Q END“, die x_i innerhalb Q verwendet, kann statt x_i eine bisher unbenutzte Variable y verwenden, wobei der LOOP–Anweisung mit Laufvariable y die Wertzuweisung $y := x_i + 0$ vorangeschaltet wird.

Induktionsanfang

P hat die Form „$x_i := x_j \pm c$“ mit $c \in \{0, 1\}$.

Das größte Wachstum wird erzielt für Variablenanfangswerte

$$n_i = 0, n_j = n$$

und die Anweisung „$x_i := x_j + 1$“, wobei sich Variablenendwerte

$$n'_i = n + 1, n'_j = n$$

ergeben (restliche Variablen auf Null). Somit gilt:

$$f_P(n) \leq 2n + 1 < 2n + 3 = A_2(n) \ .$$

1. Induktionsschritt

P hat die Form „$P_1; P_2$“.

Induktionsvoraussetzung: Es gibt Konstanten k_1, k_2 so dass

$$f_{P_1}(n) < A_{k_1}(n) \text{ und } f_{P_2}(n) < A_{k_2}(n)$$

für alle $n \geq 0$.

Offensichtlich gilt

$$f_P(n) \leq f_{P_2}\left(f_{P_1}(n)\right) \overset{IV}{<} A_{k_2}\left(f_{P_1}(n)\right) \overset{IV}{<} A_{k_2}\left(A_{k_1}(n)\right) = a(k_2, a(k_1, n)) \ .$$

Fall 1: $k_1 \leq k_2 + 1$.

$$a(k_2, a(k_1, n)) \leq a(k_2, a(k_2+1, n)) = a(k_2+1, n+1) \leq a(k_2+2, n) = A_{k_2+2}(n)$$

Fall 2: $k_1 > k_2 + 1$.

$$a(k_2, a(k_1, n)) < a(k_1 - 1, a(k_1, n)) = a(k_1, n+1) \leq a(k_1+1, n) = A_{k_1+1}(n)$$

Für $k := \max\{k_1 + 1, k_2 + 2\}$ gilt dann $f_P(n) < A_k(n)$ für alle $n \geq 0$.

2. Induktionsschritt

P hat die Form „LOOP x_i DO Q END“, wobei x_i in Q nicht verwendet wird.

Induktionsvoraussetzung: Es gibt eine Konstante k so dass $f_Q(n) < A_k(n)$ für alle $n \geq 0$.

Wähle Variablenanfangswerte $n_0, n_1, n_2, \ldots$ mit $f_P(n) = \sum_{j \geq 0} n'_i$ und setze $m := n_i$ (Anfangswert von x_i, der zu maximaler Summe der Variablenendwerte führt).

Dann gilt

$$f_P(n) \leq f_Q^m(n-m) + m \ .$$

Erkläre !

Ausgehend von dieser Ungleichung und $f_Q(n) < A_k(n)$ werden wir

$$f_P(n) < A_{k+1}(n) \ , \tag{5}$$

nachweisen, was den Induktionsbeweis abschließt.

„High Noon“ bei Familie Ackermann

$$
\begin{aligned}
f_P(n) &\leq f_Q^m(n-m)+m \\
&\leq A_k(f_Q^{m-1}(n-m))+(m-1) \\
&\leq A_k^2(f_Q^{m-2}(n-m))+(m-2) \\
&\cdots \\
&\leq A_k^m(n-m) \\
&= A_k^{m-2}\left(a(k,a(k,n-m))\right) \\
&< A_k^{m-2}(\underbrace{a(k,a(k+1,n-m))}_{=a(k+1,n-m+1)}) \\
&= A_k^{m-3}(\underbrace{a(k,(a(k+1,n-m+1)))}_{=a(k+1,n-m+2)}) \\
&\cdots \\
&= a(k+1,n-1) \\
&< a(k+1,n) = A_{k+1}(n)
\end{aligned}
$$