Theoretische Informatik

Berechenbarkeit

Technische Hochschule Rosenheim

SS 2019

Prof. Dr. J. Schmidt

Inhalt

Technische Hochschule Rosenheim

- Algorithmen
- Entscheidungsproblem und Church-Turing-These
- Halteproblem
- LOOP/WHILE/GOTO Berechenbarkeit
- primitiv rekursive Funktionen
- µ-rekursive Funktionen und die Ackermann-Funktion
- Busy-Beaver-Funktion

ALGORITHMEN

Definition Algorithmus (nach Knuth)

Technische Hochschule **Rosenheim**

- **Endlichkeit**
  - ein Algorithmus muss immer nach einer endlichen Anzahl Schritten terminieren
- **Bestimmtheit**
  - jeder Schritt eines Algorithmus ist in jedem Fall eindeutig definiert
- **Eingabe**
  - ein Algorithmus hat 0 oder mehr Eingabeparameter (statisch oder dynamisch)
- **Ausgabe**
  - ein Algorithmus hat mindestens einen Ausgabewert, der sich aus den Eingabeparametern ableitet
- **Effektivität**
  - Anweisungen müssen grundlegend genug sein, so dass sie prinzipiell
    - exakt und
    - in endlicher Zeit ausgeführt werden können

Algorithmus
Anmerkungen

- es gibt Beispiele für **nicht-terminierende** Algorithmen (Regelschleifen in eingebetteten Systemen, Betriebssysteme). Streng genommen kein Algorithmus, Knuth schlägt hierfür „Computational Method“ vor.
- effektiv ≠ effizient
- Endlichkeit ist für praktische Zwecke ein schwaches Kriterium; ein Algorithmus soll auch effizient sein
- Begriff „Algorithmus“: Al-Khwarizmi, Name eines berühmten persischen Buchautors (825 n.Ch.)

Schritte

Technische Hochschule **Rosenheim**

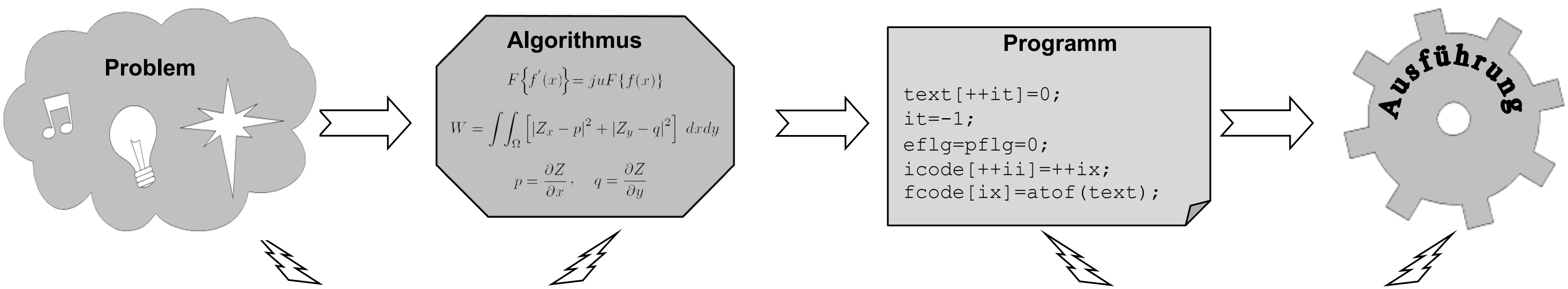
- Algorithmierung
  - finde einen Algorithmus zur prinzipiellen Lösung des anstehenden Problems
  - wesentlich: Beschreibung der funktionalen Abhängigkeit zwischen den Ein- und Ausgabedaten

- Programmierung
  - Formuliere den Algorithmus als Programm
  - Programmiersprache Bindeglied zwischen Mensch und Maschine
  - Abstraktion von Maschinendetails
  - Unterstützung einer möglichst einfachen und vollständigen Formulierung beliebiger Algorithmen

- Ausführung
  - Interpretation der formulierten Anweisungen
  - Umsetzung in endlich viele, einfache, direkt ausführbare Einzelaktionen
  - wesentlich: benötigte Zeit und Speicherplatz

Grundsätzliche Fragen

- Kann **jedes** Problem durch einen Algorithmus beschrieben werden?
  - zumindest prinzipiell, bei genügend großer Anstrengung?
- Kann **jeder** Algorithmus in ein Programm übertragen werden?
  - Welchen Anforderungen muss eine Programmiersprache genügen, damit jeder Algorithmus damit formuliert werden kann?
- Ist ein Computer grundsätzlich in der Lage, einen bekannten, als Programm formulierten Algorithmus auszuführen?



Berechenbarkeit Church-Turing-These Formale Sprachen Komplexitätstheorie

Turing-Maschinen Halteproblem LOOP/WHILE-Programme

ENTSCHEIDUNGSPROBLEM UND CHURCH-TURING-THESE

Das Gödel’sche Unvollständigkeitstheorem

Technische Hochschule Rosenheim

- Kurt Gödel (1906 – 1978)
- ursprüngliche Ansicht: jede mathematische Aussage ist **algorithmisch entscheidbar**
  - man kann prinzipiell beweisen, ob sie wahr oder falsch ist
- Unvollständigkeitstheorem (Gödel 1931)
  - Beweis, dass alle widerspruchsfreien axiomatischen Formulierungen der Zahlentheorie unentscheidbare Aussagen enthalten
  - es gibt Aussagen, die wahr, aber nicht beweisbar sind
  - nicht jede Aussage ist also algorithmisch entscheidbar
  - daher gibt es Probleme, die **prinzipiell** nicht von Computern gelöst werden können

Universelle Turingmaschine

- Formalisierung des Algorithmus-Begriffs
- jeder Algorithmus kann als Turing-Maschine dargestellt werden

- Universelle Turing-Maschine U
  - TM, die jede andere TM T simulieren kann
  - Computer entspricht einer universellen TM
  - Programmierung von U: Schreibe auf Eingabeband
    - Beschreibung der TM T
    - Eingabe x, die von T verarbeitet werden soll

Church-Turing-These

- jeder Algorithmus kann dargestellt werden als
  - Turing-Maschine („Turing-Berechenbarkeit“)
  - formale Sprache (Typ 0)
  - Programm einer Registermaschine
  - Schaltwerk
  - μ-rekursive Funktion
  - WHILE bzw. GOTO-Programm
  - …
- **alle** diese Darstellungen sind äquivalent

- **Church-Turing-These**
  - die durch die formale Definition der Turing-Berechenbarkeit erfasste Klasse von Funktionen stimmt genau mit der Klasse der intuitiv berechenbaren Funktionen überein

Church-Turing-These

- **These**: nicht beweisbar, aber allgemein akzeptiert
- Indizien für Korrektheit
  - niemand konnte bisher einen umfassenderen Berechenbarkeitsbegriff finden, als den der TM
  - die Äquivalenz vieler verschiedener Formalismen ist ein starkes Indiz dafür, dass man mit der TM tatsächlich **den Berechenbarkeitsbegriff an sich** gefunden hat
- Resultat: wenn von einer Funktion nachgewiesen ist, dass sie nicht Turing-berechenbar ist, so ist sie überhaupt nicht berechenbar

Berechenbarkeit/Entscheidbarkeit

Technische Hochschule Rosenheim

- Berechenbare Funktionen
  - Funktion f: $\mathbb{N}^k \to \mathbb{N}$ heißt **berechenbar**, wenn es einen Algorithmus gibt, der bei Eingabe von $\boldsymbol{x} \in \mathbb{N}^k$ f($\boldsymbol{x}$) berechnet
  - d.h., Algorithmus stoppt nach endlich vielen Schritten
  - bei partiellen Funktionen (an manchen Stellen undefiniert): Funktion eingeschränkt auf Definitionsbereich

- Entscheidbarkeit
  - Menge M heißt **entscheidbar**, wenn ihre charakteristische Funktion $\chi(m)$ berechenbar ist
  - $\chi(m)$ berechnet, ob ein Element m in der Menge M enthalten ist oder nicht:

$$\chi(m) = \begin{cases} 1 & \text{wenn } m \in M \\ 0 & \text{sonst} \end{cases}$$

Nicht-berechenbare Funktionen

- ein Algorithmus
  - muss durch ein Alphabet A mit einem endlichen Zeichenvorrat dargestellt werden können
  - im Falle einer TM reicht das binäre Alphabet A = {0, 1}
  - hat endliche Länge
- Nachrichtenraum A* besteht aus abzählbar unendlich vielen Zeichenketten
- daher gibt es auch nur abzählbar viele Algorithmen
  - d.h. alle Algorithmen könnten unter Verwendung der natürlichen Zahlen im Prinzip durchnummeriert werden
  - es gibt nur endlich viele Quelltexte mit bestimmter Länge in einer bestimmten Programmiersprache
  - man könnte diese also alle hinschreiben und ordnen

Nicht-berechenbare Funktionen – Beweis

Technische Hochschule Rosenheim

- Es gibt nicht-berechenbare Funktionen
- bereits die Menge der Funktionen f(n): $\mathbb{N} \rightarrow \mathbb{N}$ ist überabzählbar
- Beweis
  - Annahme: Menge f(n), $\mathrm{n} \in \mathbb{N}$ ist abzählbar (und damit **komplett berechenbar**)
  - dann kann man die Funktionen sortiert in eine Tabelle schreiben:

	1	**2**	**3**	**4**	**…**
$\mathbf{f_1}$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	…
$\mathbf{f_2}$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	…
$\mathbf{f_3}$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	…
$\mathbf{f_4}$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	…
…	…	…	…	…	…

Nicht-berechenbare Funktionen – Beweis

Technische Hochschule Rosenheim

- Konstruiere Funktion g wie folgt
  - $g(1) = f_1(1) + 1$ somit unterscheidet sich g von f_1
  - $g(2) = f_2(2) + 1$ somit unterscheidet sich g von f_2
  - $g(3) = f_3(3) + 1$ somit unterscheidet sich g von f_3
  - usw.
- g unterscheidet sich von allen Funktionen f_i
- g ist offensichtlich berechenbar
- damit müsste g in der Tabelle enthalten sein
- das ist aber nicht der Fall

- Fazit: **Widerspruch** – Annahme, dass Tabelle alle Funktionen $f(n)\colon \mathbb{N} \to \mathbb{N}$ enthält, ist falsch

Nicht-berechenbare Funktionen – Fazit

- es gibt nicht berechenbare Funktionen
- es gibt überabzählbar viele arithmetische Funktionen
- von diesen sind nur abzählbar viele berechenbar
- verglichen mit dem, was ein Computer **nicht** kann, ist das was er kann vernachlässigbar klein

- Anmerkung:
  - nicht-berechenbar bedeutet **nicht**, dass es Probleme gibt, für die einfach noch kein Algorithmus gefunden wurde
  - es bedeutet: es gibt Probleme, für die es **prinzipiell** keinen Algorithmus zur Lösung geben kann
    - unabhängig von der zukünftigen Entwicklung der Computer-Hardware
    - …und viele davon wären praktisch interessant

HALTEPROBLEM

Halteproblem

- wichtigstes Beispiel für unentscheidbares Problem
- Frage:
  Gibt es einen Algorithmus bzw. ein Programm HALT, mit dem man für ein **beliebiges** Programm P ermitteln kann, ob es mit beliebigen Eingabedaten jemals stoppen wird oder nicht?

- Aufruf HALT(P) würde liefern:
  - P stoppt
  - P stoppt nicht
  - ohne dass man P selbst laufen lassen müsste
- HALT könnte also prüfen, ob ein Programm in eine Endlosschleife geraten wird

Stein der Weisen

Technische Hochschule **Rosenheim**

- Bedeutung des Halteproblems ist unübertroffen essentiell
- wäre es entscheidbar, hätte man einen **Stein der Weisen**, mit dem man sämtliche als Programm formulierbaren Probleme der Welt sofort lösen könnte
- Beispiel: Goldbachsche Vermutung
  - jede gerade Zahl g > 2 ist als Summe zweier Primzahlen darstellbar
  - bisher unbewiesen – auf jeden Fall richtig für alle Zahlen $g < 2 \cdot 10^{18}$
  - schreibe Programm, dass alle geraden Zahlen g durch Probieren testet, ob g die Summe zweier Primzahlen ist
  - das Programm hält an, falls dies für ein bestimmtes g nicht zutrifft
  - wenn die Goldbachsche Vermutung zutrifft: Programm wird nie anhalten
  - dies könnte man aber durch HALT(GOLDBACH) vorab testen
  - damit wäre die Goldbachsche Vermutung eindeutig bewiesen oder widerlegt

Beweis – spezielles Halteproblem

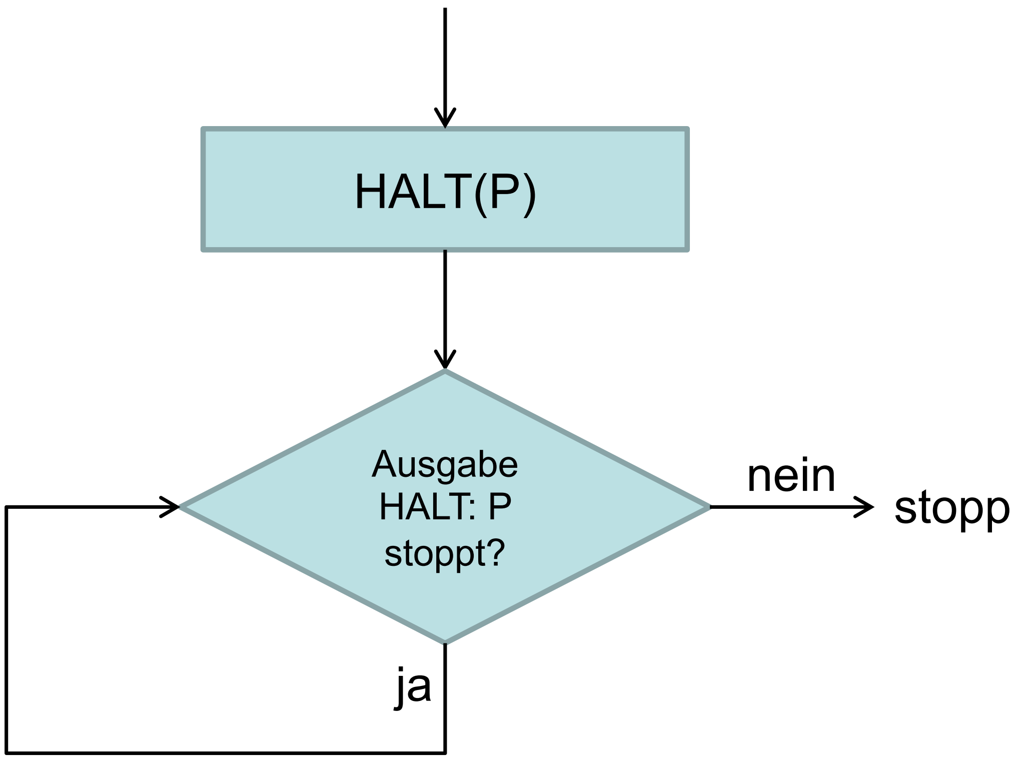
- Annahme: es existiert ein Algorithmus zur Lösung des Halteproblems
- Es gibt also ein Programm HALT
  - Eingabe:
    - beliebiges zu testendes Programm P
    - inklusive dessen Eingabedaten
  - Ausgabe: P „stoppt“ oder „stoppt nicht“

- bei beliebigen Eingabedaten für P:
  allgemeines Halteproblem
- P verwendet seinen eigenen Code als Eingabe:
  spezielles Halteproblem oder
  Selbstanwendbarkeitsproblem

Beweis – spezielles Halteproblem

Technische Hochschule Rosenheim

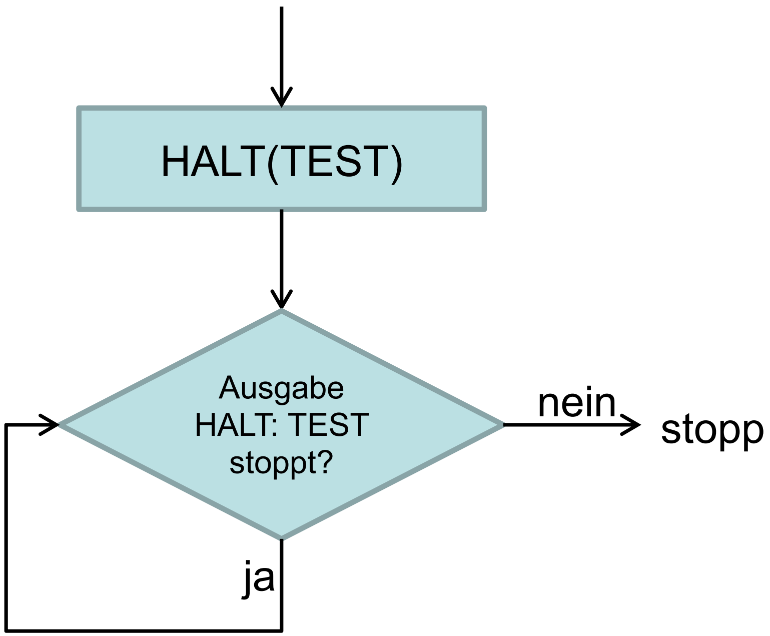
- Konstruiere nun ein Programm TEST wie folgt:



Beweis – spezielles Halteproblem

Technische Hochschule Rosenheim

- Nun: P = TEST
- 2 Fälle
  - TEST(TEST) stoppt
    - Ausgabe von HALT(TEST): TEST stoppt nicht
  - TEST(TEST) stoppt nicht
    - Ausgabe von HALT(TEST): TEST stoppt
- Widerspruch!
- Schlussfolgerung:
  - **HALT existiert nicht!**
  - **Das spezielle Halteproblem ist unentscheidbar**



Allgemeines Halteproblem

- Beweis von vielen weiteren unentscheidbaren Problemen durch Reduktion auf das spezielle Halteproblem möglich
  - d.h., Einbettung des speziellen Halteproblems als Spezialfall in das neue Probleme
  - dann muss das allgemeinere Problem erst recht unentscheidbar sein

- Allgemeines Halteproblem
  - Entscheide, ob P mit beliebiger Eingabe stoppt
  - Reduktion offensichtlich: bereits mit Spezialfall P als Eingabe unentscheidbar
  - **Das allgemeine Halteproblem ist unentscheidbar**

Halteproblem auf leerem Band

- Entscheide, ob P angesetzt auf leerem Band (also mit keiner Eingabe) stoppt
- Reduktion:
  - schreibe nach dem Start zunächst Code von P aufs Band
  - anschließend Verhalten wie bei speziellem Halteproblem
- **Das Halteproblem auf leerem Band ist unentscheidbar**

Weitere unentscheidbare Probleme

- Berechnen zwei TM/Programme die gleiche Funktion?
  - Äquivalenzproblem
  - lässt sich nicht auf das Halteproblem zurückführen
  - ist also „noch unentscheidbarer“ als dieses
- Berechnet TM eine konstante Funktion?
- Game of Life: gegeben 2 Konfigurationen – gibt es eine Zugfolge, so dass die eine aus der anderen entsteht?

- Satz von Rice:
  - Es ist hoffnungslos von einer TM **irgendeinen** Aspekt ihres funktionalen Verhaltens algorithmisch bestimmen zu wollen

Wort-/Leerheits-/Schnitt-/ Äquivalenzproblem

Technische Hochschule **Rosenheim**

- für welche Sprachklassen/Automatenmodelle ist das Problem entscheidbar (lösbar)?
- Einträge
  - ja: es gibt einen Algorithmus, der das Problem löst
  - nein: das Problem ist unlösbar, es gibt keinen Algorithmus dafür

Sprache	Wortproblem	Leerheits-/ Endlichkeitsproblem	Äquivalenzproblem	Schnittproblem
Typ 3	ja	ja	ja	ja
det.kf.	ja	ja	ja	nein
Typ 2	ja	ja	nein	nein
Typ 1	ja	nein	nein	nein
Typ 0	nein	nein	nein	nein

Typ 2 Sprachen

- Gegeben: kontextfreie Grammatiken G, G_1 und G_2
- Unentscheidbar sind
  - ist G mehrdeutig?
  - ist $\overline{L(G)}$ kontextfrei?
  - ist L(G) deterministisch kontextfrei?
  - ist L(G) regulär?
  - ist $L(G_1) \cap L(G_2) = \emptyset$?
  - ist $L(G_1) \cap L(G_2)$ kontextfrei?
  - ist $| L(G_1) \cap L(G_2) | = \infty$?
  - ist $L(G_1) \subseteq L(G_2)$?
  - ist $L(G_1) = L(G_2)$?

Deterministisch-kontextfreie Sprachen

Technische Hochschule Rosenheim

- Gegeben: det. kontextfreie Grammatiken G_1 und G_2
- Unentscheidbar sind
  - ist $L(G_1) \cap L(G_2) = \emptyset$?
  - ist $L(G_1) \cap L(G_2)$ kontextfrei?
  - ist $| L(G_1) \cap L(G_2) | = \infty$?
  - ist $L(G_1) \subseteq L(G_2)$?

LOOP/WHILE/GOTO BERECHENBARKEIT

LOOP-Programme

Technische Hochschule Rosenheim

- einfache Programmiersprache
- Komponenten
  - Variablen: $x_0, x_1, x_2, x_3, \ldots$
  - Konstanten: 0, 1, 2, …
  - Trennsymbole: ; :=
  - Operatoren: + –
  - Schlüsselwörter: LOOP DO END
- Syntax
  - $x_i := x_j + c$ bzw. $x_i := x_j - c$ ist ein LOOP-Programm (mit c Konstante)
  - wenn P_1 und P_2 LOOP-Programme, dann auch $P_1 ; P_2$
  - wenn P ein LOOP Programm und x_i eine Variable, dann ist auch
    LOOP x_i DO P END
    ein LOOP-Programm

LOOP-Programme

- Semantik
  - Programm wird gestartet mit Parametern in den Variablen $x_1, \ldots, x_n$
  - alle anderen haben den Wert 0
  - es sind nur natürliche Zahlen zulässig
  - Berechnungsergebnis steht am Ende in x_0
  - Zuweisungen
    - + wie üblich
    - –: wenn Wert kleiner Null werden würde, wird der Wert auf Null gesetzt
  - P_1 ; P_2 bedeutet: erst wird P_1, dann P_2 ausgeführt
  - LOOP x_i DO P END bedeutet
    - P wird x_i mal ausgeführt
    - Änderung der Variablen im Schleifenrumpf hat keinen Effekt

Anmerkungen

- die alleinige Verwendung von natürlichen Zahlen ist keine Einschränkung
- jedes Alphabet kann auf die natürlichen Zahlen abgebildet werden
- ebenso jede rationale Zahl
- nicht: reelle Zahlen – die kann eine TM/Rechner aber sowieso nicht verarbeiten

1: 1/1	*2:* 1/2	*6:* 1/3	*7:* 1/4	*15:* 1/5	*16:* 1/6	*28:* 1/7	.....
3: 2/1	*5:* 2/2	*8:* 2/3	*14:* 2/4	*17:* 2/5	*27:* 2/6	.....	
4: 3/1	*9:* 3/2	*13:* 3/3	*18:* 3/4	*26:* 3/5	.....		
10: 4/1	*12:* 4/2	*19:* 4/3	*25:* 4/4	.....			
11: 5/1	*20:* 5/2	*24:* 5/3	.....				
21: 6/1	*23:* 6/2	.....					
22: 7/1	.....						
.....							

LOOP-Programme Eigenschaften

Technische Hochschule **Rosenheim**

- alle LOOP-berechenbaren Funktionen sind totale Funktionen
  - die Umkehrung gilt nicht: Ackermann-Funktion
- jedes LOOP-Programm stoppt immer in endlicher Zeit
- Wertzuweisungen
  - x_i := c durch x_i := x_j + c mit Verwendung einer nicht benutzten Variablen x_j, die noch den Wert Null hat
  - x_i := x_j durch Verwendung von c = 0
- IF-THEN
  - IF x = 0 THEN P END kann simuliert werden durch
  - y := 1;
LOOP x DO y := 0 END;
LOOP y DO P END

LOOP-Programme
Beispiel

- Addition ist LOOP-berechenbar: $x_0 := x_1 + x_2$
- $x_0 := x_1;$
  LOOP x_2 DO $x_0 := x_0 + 1$ END

WHILE-Programme

- Erweiterung der LOOP-Syntax durch
  - wenn P ein WHILE-Programm und x_i eine Variable, dann ist auch WHILE $x_i \neq 0$ DO P END ein WHILE-Programm
- Semantik: Führe P so lange aus, wie der Variablenwert nicht Null ist

- Anmerkung: LOOP wird nun eigentlich nicht mehr benötigt
  - LOOP x DO P END entspricht
  - y := x;
    WHILE y ≠ 0 DO y := y – 1; P END

WHILE-Programme

- es können nun auch partielle Funktionen dargestellt werden
  - Endlosschleifen sind möglich
- jede WHILE-berechenbare Funktion ist auch Turing-berechenbar
  - TM können WHILE Programme simulieren
  - die Umkehrung gilt ebenfalls
- für **jedes** beliebige Programm genügt **eine einzige** WHILE-Schleife – Beweis folgt

GOTO-Programme

- Sequenz von Anweisungen mit Marke
  $M_1 : A_1 ; M_2 : A_2 ; \ldots; M_n : A_n$

- zulässige Anweisungen:
  - Zuweisung: $x_i := x_j + c$ bzw. $x_i := x_j - c$
  - unbedingter Sprung: GOTO M_i
  - bedingter Sprung: IF $x_i = c$ THEN GOTO M_i
  - Stopp: HALT

- jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar und umgekehrt

WHILE durch GOTO

- WHILE $x_i \neq 0$ DO P END
  entspricht
- M_1 : IF $x_i = 0$ THEN GOTO M_2;
  P;
  GOTO M_1 ;
  M_2 : …

GOTO durch WHILE

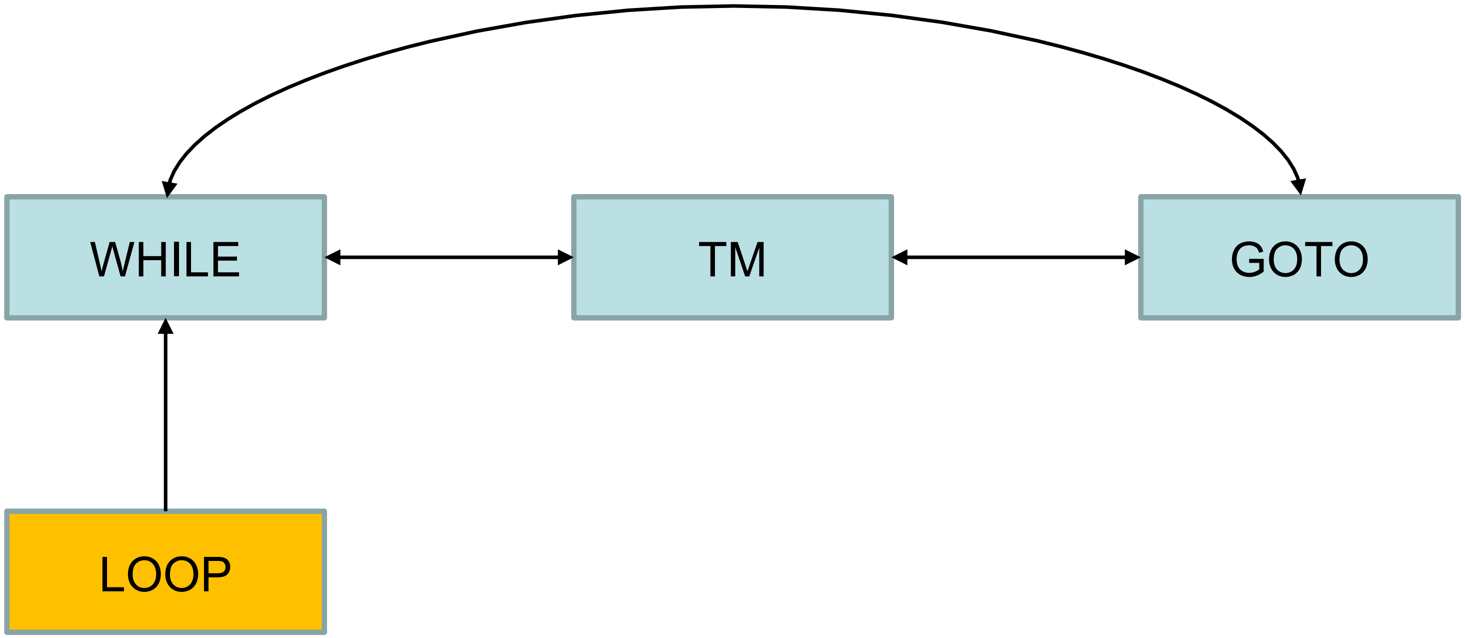
Technische Hochschule **Rosenheim**

- GOTO Programm: $M_1 : A_1 ; M_2 : A_2 ; \ldots ; M_n : A_n$
- umgeformt in WHILE:

```
z := 1;
WHILE z ≠ 0 DO
      IF z = 1 THEN A'1 END;
      IF z = 2 THEN A'2 END;
      ...
      IF z = n THEN A'n END;
END
```

- mit A'_i =
  - $x_j := x_k \pm c \, ; \, z := z + 1$ falls $A_i =$ $x_j := x_k \pm c$
  - $z := n$ falls $A_i =$ GOTO M_n
  - IF $x_i = c$ THEN $z := n$ ELSE $z := z + 1$ END falls $A_i =$ IF $x_i = c$ THEN GOTO M_n
  - $z := 0$ falls $A_i =$ HALT
  - wobei IF-THEN-ELSE durch LOOP darstellbar ist
- es entsteht nur eine einzige WHILE-Schleife!

LOOP-GOTO-WHILE-TM



PRIMITIV REKURSIVE FUNKTIONEN

Primitive Rekursion

- Berechenbarkeitsmodell, entstanden parallel zu Turing
- es werden wenige Grundfunktionen definiert, aus denen neue Funktionen erstellt werden können

Primitive Rekursion
Definition

- Die folgenden Basisfunktionen sind primitiv rekursiv:
  - alle **konstanten Funktionen**
    $f: \mathbb{N}_0^n \to \mathbb{N}_0,\ f(\boldsymbol{x}) = c\ \ , c \in \mathbb{N}_0\ , \forall x \in \mathbb{N}_0^n$
  - **Projektion**
    $p_i^n: \mathbb{N}_0^n \to \mathbb{N}_0, p_i^n(x_1, x_2, \dots, x_n) = x_i, \quad 1 \le i \le n$
  - **Nachfolgerfunktion**
    $s: \mathbb{N}_0\ \to \mathbb{N}_0,\ s(x) = x + 1$

- Die wie folgt konstruierten Funktionen sind primitiv rekursiv:
  - **Funktionskomposition** (Einsetzung)
    seien $g: \mathbb{N}_0^n \to \mathbb{N}_0$ und $h_1, h_2, \dots, h_n: \mathbb{N}_0^n \to \mathbb{N}_0$ primitiv rekursiv
    dann ist auch $f(\boldsymbol{x}) = g(h_1(\boldsymbol{x}), \dots, h_n(\boldsymbol{x}))$ primitiv rekursiv
  - **primitive Rekursion**
    seien $g: \mathbb{N}_0^n \to \mathbb{N}_0$ und $h: \mathbb{N}_0^{n+2} \to \mathbb{N}_0$ primitiv rekursiv
    dann ist auch $f: \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ primitiv rekursiv mit

$$f(0, \boldsymbol{y}) = g(\boldsymbol{y})\,, \qquad \boldsymbol{y} \in \mathbb{N}_0^n$$
$$f(x+1, \boldsymbol{y}) = h\big(x, \boldsymbol{y}, f(x, \boldsymbol{y})\big)\,, \qquad x \in \mathbb{N}_0\ , \boldsymbol{y} \in \mathbb{N}_0^n$$

Primitive Rekursion
Beispiele

- Addition

$$
\begin{aligned}
add(0, y) &= g(y) = p_1^1(y) = y \\
add(x + 1, y) &= h(x, y, add(x, y)) \\
&= s(p_3^3(x, y, add(x, y))) \\
&= add(x, y) + 1
\end{aligned}
$$

- Multiplikation

$$
\begin{aligned}
mult(0, y) &= g(y) = 0 \\
mult(x + 1, y) &= h(x, y, mult(x, y)) \\
&= add(\, p_2^3(x, y, mult(x, y)), \\
&\qquad\quad p_3^3(x, y, mult(x, y))\ ) \\
&= add(y, mult(x, y))
\end{aligned}
$$

Primitive Rekursion

- alle primitiv rekursiven Funktionen sind
  - berechenbar
  - total
- die Umkehrung gilt nicht
- die Klasse der primitiv rekursiven Funktionen stimmt genau mit der Klasse der LOOP-berechenbaren Funktionen überein
- Schlussfolgerung
  - jede Iteration lässt sich durch eine Rekursion darstellen
  - und umgekehrt

μ-REKURSIVE FUNKTIONEN

μ-Rekursion

- Erweiterung des Konzepts der primitiven Rekursion
- hinzunehmen des μ-Operators
- sei $f: \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ eine μ-rekursive Funktion, dann ist auch $\mu f: \mathbb{N}_0^{n} \to \mathbb{N}_0$ μ-rekursiv mit

$$\mu f(x_1, \dots, x_n) = \begin{cases} \min M & \text{falls } M \neq \emptyset \\ \text{undefiniert} & \text{falls } M = \emptyset \end{cases}$$

  mit

  $$\mathrm{M} = \{\mathrm{k} \mid f(k, x_1, \dots, x_n) = 0 \text{ und} f(l, x_1, \dots, x_n) \text{ ist def. } \forall l < k\}$$

- somit sind jetzt auch partielle Funktionen darstellbar

Ackermannfunktion

- totale berechenbare Funktion, die nicht primitiv rekursiv ist (und damit auch nicht LOOP-berechenbar)
- entdeckt von ***Wilhelm Ackermann*** 1928
- einfachste bekannte Funktion, die schneller wächst als jede primitiv rekursive Funktion
  - also auch schneller als die Fakultät und jede Exponentialfunktion

- Definition

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

  $a(x + 1, 0) = a(x, 1)$

  $a(x + 1, y + 1) = a(x, a(x + 1, y))$

Ackermannfunktion

$a(0, y) = y + 1$
$a(x + 1, 0) = a(x, 1)$
$a(x + 1, y + 1) = a(x, a(x + 1, y))$

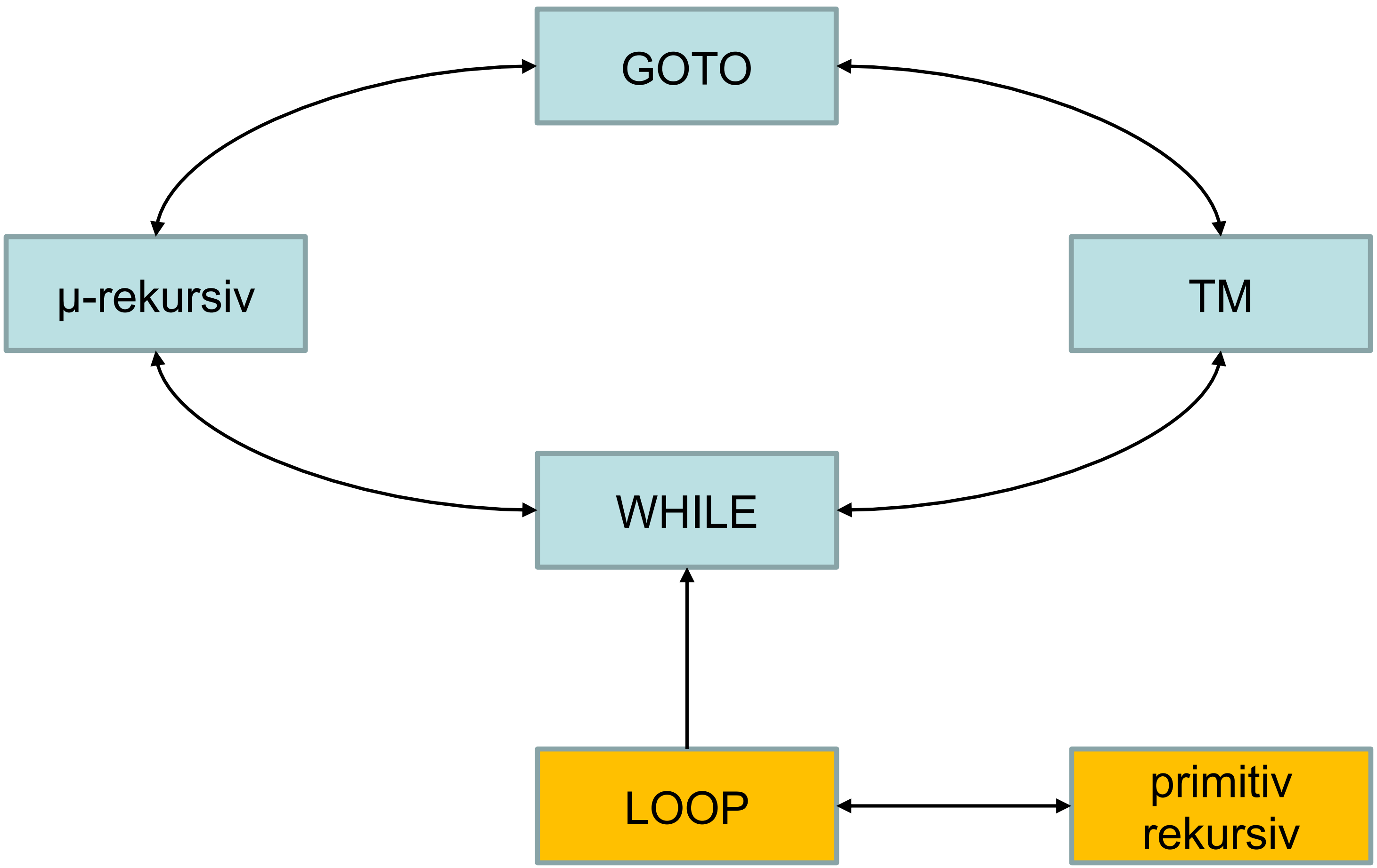
- Berechnung von $a(1, 2)$

$$
\begin{aligned}
a(1, 2) &= a(0, a(1, 1)) \\
&= a(0, a(0, a(1, 0))) \\
&= a(0, a(0, a(0, 1))) \\
&= a(0, a(0, 2)) \\
&= a(0, 3) \\
&= 4
\end{aligned}
$$

- Wachstum von $a(x, y)$

$a(1, 1) = 3$
$a(1, 2) = 4$
$a(2, 2) = 7$
$a(3, 3) = 61$
$a(4, 4) > 10^{10^{10^{2100}}}$
Anzahl Atome im Universum: ca. 10^{80}

Zusammenfassung



Turing-Vollständigkeit

- Eine Programmiersprache heißt **Turing-vollständig**, wenn damit alles berechenbar ist, was auch eine TM berechnen kann

- Turing-vollständig sind z.B.
  - WHILE, GOTO und μ-Rekursion
  - alle verbreiteten prozeduralen, objektorientierten oder funktionalen Programmiersprachen

- Nicht Turing-vollständig sind z.B.
  - LOOP und primitive Rekursion
  - reguläre Ausdrücke

BUSY BEAVER

Definition

- ***Tibor Radó*** 1962
- wächst schneller als jede μ-rekursive Funktion
  - und damit schneller als jede berechenbare Funktion
  - ist also auch nicht durch WHILE-/GOTO-Programme oder TM darstellbar
  - daher nicht berechenbar – es gibt keinen allgemeinen Algorithmus zur Lösung des Problems

- **Definition**
  bb(0) = 0
  bb(n) = die maximale Anzahl von Strichen (Einsen), die eine Turing-Maschine mit n Zuständen (Anweisungen) und Alphabet {0, 1} auf ein leeres Band schreibt **und hält**

Bestimmung von bb(n)

1. Liste alle Turing-Maschinen mit T = {0,1} mit n Anweisungen auf.
   - jede Anweisung besteht aus zwei Teilen: ergibt 2n Teilanweisungen
   - für jede gibt es zwei Möglichkeiten für das zu schreibende Zeichen
   - und zwei Möglichkeiten für den nächsten Schritt (L, R)
   - und n+1 mögliche Anweisungsnummern (einschließlich HALT) für den folgenden Schritt
   - Anzahl der Turing-Maschinen mit n Anweisungen: $[4(n+1)]^{2n}$

   - Für n=5: ca. $6{,}3 \cdot 10^{13}$ Möglichkeiten

Bestimmung von bb(n)

2. suche alle **haltenden** Turing-Maschinen aus, die auf ein mit Nullen vorbesetztes Band Einsen schreibt
   - solche gibt es für jedes n auf jeden Fall (wird hier nicht bewiesen)
   - das auftretende Halteproblem ist zwar ein Indiz für die Nichtberechenbarkeit von bb(n), aber als Beweis nicht ausreichend
   - es wird **nicht** vorausgesetzt, dass ein **allgemeines** Verfahren zur Lösung des Halteproblems existieren muss
   - das Problem ist sehr speziell
   - man könnte verschiedene angepasste Verfahren für jede zu prüfende TM entwickeln – die Anzahl ist für jedes n ja endlich

3. Prüfe für jede der so ausgewählten Turing-Maschinen, wie viele Striche sie auf das Band schreibt, bevor sie anhält. Die größte Anzahl geschriebener Striche ist bb(n).

bb(n)

- der Beweis, dass bb(n) tatsächlich nicht berechenbar ist, soll hier nicht geführt werden
- das bedeutet nicht, dass bb(n) nicht für einzelne Werte bestimmbar ist
- es existiert nur kein allgemeingültiger Algorithmus, der bb(n) für jedes beliebige n berechnen könnte

n	0	1	2	3	4	5	6	>6
bb(n)	0	1	4	6	13	≥ 4098	$\geq 3{,}5 \cdot 10^{18267}$	?

Zusammenfassung

- Berechenbarkeit
  - Es gibt einen Algorithmus zur Berechnung der Funktion
  - dieser stoppt nach endlich vielen Schritten
- Entscheidbarkeit
  - Berechenbarkeit der charakteristischen Funktion
- Problem unentscheidbar
  - es gibt keinen allgemeingültigen Algorithmus, der das Problem löst
  - es kann aber durchaus für manche Fälle oder auch mit speziell auf bestimmte Fälle angepassten Algorithmen Lösungen geben
- es existieren unendlich viel mehr nicht-berechenbare Funktionen als berechenbare

Quellen

Die Folien entstanden auf Basis folgender Literatur

- H. Ernst, J. Schmidt und G. Beneken: Grundkurs Informatik. Springer Vieweg, 6. Aufl., 2016.
- Schöning, U.: *Theoretische Informatik - kurz gefasst*. Spektrum Akad. Verlag (2008)
- Hopcroft, J.E., Motwani, R. und Ullmann, J.D.: *Einführung in die Automatentheorie, formalen Sprachen und Komplexitätstheorie*. Pearson Studium (2002)