

Berechnungsmodelle

Proseminar Theoretische Informatik

Marcel Jünemann, 11. Januar 2011

1 Berechenbarkeit

Intuitiver Berechenbarkeitsbegriff

Definition Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist intuitiv-berechenbar, falls ein Algorithmus existiert, der diese Funktion berechnet.

Beispiele:

$$f_\pi(n) = \begin{cases} 1 & \text{wenn } n \text{ Anfangsabschnitt von } \pi \\ 0 & \text{sonst} \end{cases}$$

Zum Beispiel $f(314) = 1$ und $f(2) = 0$. f_π ist **berechenbar**, da ein Algorithmus zur Berechnung einer Näherung von π mit beliebiger Genauigkeit existiert

$$f_r(n) = \begin{cases} 1 & \text{wenn } n \text{ Anfangsabschnitt von } r \\ 0 & \text{sonst} \end{cases}$$

f_r ist für $r \in \mathbb{R}$ im Allgemeinen **nicht berechenbar**, da es überabzählbar viele reelle Zahlen gibt, aber nur abzählbar viele Algorithmen.

$$f_{part}(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Auch partielle Funktionen können **berechenbar** sein. Für undefinierte Funktionswerte soll der Algorithmus dabei nicht terminieren.

$$g(n) = \begin{cases} 1 & \text{falls } P = NP \\ 0 & \text{falls } P \neq NP \end{cases}$$

Auch g ist **berechenbar**: Wir können zwar zur Zeit keinen Algorithmus angeben, der g berechnet, aber es ist sicher, dass einer existiert.

Turing-Berechenbarkeit

Definition Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist Turing-berechenbar, falls eine deterministische Turingmaschine existiert, die aus der Startkonfiguration

$$z_0 \text{bin}(n_1) \# \text{bin}(n_2) \# \dots \# \text{bin}(n_k)$$

in die Endkonfiguration

$$z_e \text{bin}(f(n_1, n_2, \dots, n_k))$$

übergeht. Dabei bezeichnet $\text{bin}(x)$ die Binärdarstellung der Zahl x .

Churchsche These

Die Klasse der Turing-berechenbaren Funktionen entspricht der Klasse der intuitiv-berechenbaren Funktionen.

Anmerkung: Diese These kann nicht bewiesen werden, da der Begriff der intuitiven Berechenbarkeit nicht exakt formalisiert werden kann. Die These wird aber üblicherweise als wahr angenommen.

Berechnungsmodelle

Ein Berechnungsmodell ist ein (abstraktes) Modell einer Maschine, die Algorithmen zur Berechnung von Funktionen ausführen kann. Die Algorithmen müssen dabei in einer von der Maschine vorgeschriebenen Sprache formuliert sein. Berechnungsmodelle, die so mächtig sind wie die Turingmaschine, werden universell genannt. Im folgenden untersuchen wir drei Berechnungsmodelle und deren Mächtigkeit.

2 LOOP, WHILE und GOTO

Definition (Registermaschine) Eine Registermaschine ist eine Maschine, die über (unendlich viele) Register x_0, x_1, x_2, \dots verfügt und ein LOOP-, WHILE- oder GOTO-Programm ausführen kann. In den Registern können natürliche Zahlen gespeichert werden.

Definition (LOOP-Programm) Folgendes sind LOOP-Programme:

1. $x_i := x_i + 1$, falls $i \in \mathbb{N}$
2. $x_i := x_i - 1$, falls $i \in \mathbb{N}$
3. $P_1; P_2$, falls P_1 und P_2 LOOP-Programme

4. LOOP x_i DO P END
falls $i \in \mathbb{N}$ und P LOOP-Programm

Definition (Semantik von LOOP-Programmen) Eine Registermaschine führt ein LOOP-Programm wie folgt aus:

1. $x_i := x_i + 1$: Register x_i inkrementieren
2. $x_i := x_i - 1$: Register x_i dekrementieren, 0 dekrementiert bleibt 0
3. $P_1; P_2$: erst P_1 ausführen und danach P_2
4. LOOP x_i DO P END: P n -mal ausführen, wobei n der Inhalt von x_i zu Beginn der Schleife ist

Definition (LOOP) Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist LOOP-berechenbar, falls ein LOOP-Programm P mit folgender Eigenschaft existiert: Wenn wir die Register x_1, \dots, x_k einer Registermaschine mit den Parametern n_1, \dots, n_k befüllen und die restlichen auf 0 setzen, so steht im Register x_0 nach Ausführung von P der Wert von $f(n_1, \dots, n_k)$.

Die Klasse der LOOP-berechenbaren Funktionen heißt LOOP.

Beispiel: $f(n, m) = n + m$

```
LOOP  $x_1$  DO  $x_0 := x_0 + 1$  END;
LOOP  $x_2$  DO  $x_0 := x_0 + 1$  END
```

Beispiel: $f(n, m) = n \cdot m$

```
LOOP  $x_1$  DO
  LOOP  $x_2$  DO  $x_0 := x_0 + 1$  END
END
```

Wir benutzen im folgenden die Befehle $x_i := x_j \pm x_k$, $x_i := x_j \cdot x_k$ und ähnliche im Bewusstsein, dass wir diese Befehle in LOOP-Programmen simulieren können.

Beispiel: $f(n) = n$ -te Fibonacci-Zahl

```
 $x_1 := x_1 - 1$ ;
 $x_2 := 1$ ;
 $x_3 := 1$ ;
LOOP  $x_1$  DO
   $x_0 := x_2 + x_3$ ;
   $x_2 := x_3$ ;
   $x_3 := x_0$ 
END
```

Satz: IF $x_i > c$ THEN P_1 ELSE P_2 END ist simulierbar. **Beweis:**

```
 $x_n := x_i - c$ ;
 $x_{n+1} := 0$ ;
 $x_{n+2} := 1$ ;
LOOP  $x_n$  DO  $x_{n+1} := 1$ ;  $x_{n+2} := 0$  END;
LOOP  $x_{n+1}$  DO  $P_1$  END;
LOOP  $x_{n+2}$  DO  $P_2$  END
```

Satz: IF $x_i = c$ THEN P_1 ELSE P_2 END ist simulierbar. **Beweis:**

```
 $x_n := x_i + 1 - c$ ;
IF  $x_n > 1$  THEN  $P_2$  ELSE
  IF  $x_n > 0$  THEN  $P_1$  ELSE  $P_2$  END
END
```

Definition (WHILE) Syntax und Semantik von WHILE-Programmen sind analog zu LOOP-Programmen definiert, mit einer Ausnahme: Anstelle des LOOP-Befehls gibt es WHILE $x_i \neq 0$ DO P END. Die Anzahl der Schleifendurchläufe steht also zu Beginn der Schleife nicht fest.

Satz (LOOP \subseteq WHILE) Jede LOOP-berechenbare Funktion ist auch WHILE-berechenbar.

Beweis: Wir können das LOOP-Programm LOOP x_i DO P END in WHILE-Programmen simulieren. Wir wählen dazu zwei noch unbenutzte Register x_n und x_{n+1} .

```
WHILE  $x_i \neq 0$  DO
   $x_n := x_n + 1$ ;
   $x_{n+1} := x_{n+1} + 1$ ;
   $x_i := x_i - 1$ 
END
WHILE  $x_{n+1} \neq 0$  DO
   $x_i := x_i + 1$ ;
   $x_{n+1} := x_{n+1} - 1$ 
END
WHILE  $x_n \neq 0$  DO
   $P$ ;
   $x_n := x_n - 1$ 
END
```

P wandeln wir dabei rekursiv in ein WHILE-Programm um.

Folgerung: Wir dürfen die simulierbaren Befehle (Addition, Multiplikation, IF-THEN-

ELSE) in WHILE-Programmen benutzen.

Satz (WHILE $\not\subseteq$ LOOP) Es gibt WHILE-Programme, die nicht mit einem LOOP-Programm simuliert werden können.

Beweis: LOOP-Programme können nur totale Funktionen berechnen, da sie immer terminieren. Zum Beispiel kann die folgende Funktion nur als WHILE-Programm formuliert werden:

$$f(n) = \begin{cases} 0 & \text{wenn } n = 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

WHILE $x_1 \neq 0$ DO $x_1 := x_1 + 1$ END

Satz (WHILE_{total} $\not\subseteq$ LOOP_{total}) Es gibt sogar totale Funktionen, die nur von WHILE-Programmen, nicht aber von LOOP-Programmen berechnet werden können.

Beweis: Die Ackermann-Funktion wächst stärker als jede LOOP-berechenbare Funktion. Wir benutzen eine etwas abgewandelte Version der Ackermann-Funktion.

$$\begin{aligned} a(0, y) &= 1 \\ a(x, 0) &= 1 \\ a(1, y) &= 3y + 1 \\ a(x, y) &= \underbrace{a(x-1, a(x-1, \dots, a(x-1, y) \dots))}_{y\text{-mal}} \end{aligned}$$

Als nächstes definieren wir eine Funktion f_P , die zu jedem LOOP-Programm P die maximale Größe der Ausgabe für beliebige Eingaben berechnet. Das Argument von f_P gibt dabei an, wie groß die Eingabe maximal sein darf.

$$f_P(n) = \max\{\sum_{i=0}^{\infty} x'_i \mid \sum_{i=0}^{\infty} x_i \leq n\}$$

x_i bezeichnet dabei den Inhalt von Register x_i vor Ausführung von P und x'_i den Inhalt nach Ausführung des Programms.

Lemma $\forall P \exists k \forall n \geq k : f_P(n) < a(k, n)$

Beweis: Wir beweisen das Lemma mittels Induktion über den Aufbau von LOOP-Programmen.

Induktionsanker: P hat die Form $x_i := x_i \pm 1$
 $f_P(n) \leq n + 1 < 3n + 1 = a(1, n)$

Es existiert also ein k , z.B. $k = 1$.

Induktionsvoraussetzung: Für zwei LOOP-Programme P_1 und P_2 gibt es Konstanten k_1

und k_2 , sodass $\forall n \geq k_1 : f_{P_1}(n) < a(k_1, n)$ und $\forall n \geq k_2 : f_{P_2}(n) < a(k_2, n)$.

Induktionsschritt: Es bleiben zwei Fälle, für die ein passendes k gefunden werden muss:

1. P hat die Form $P_1; P_2$

$$\begin{aligned} f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\ &< a(k_2, a(k_1, n)) \text{ nach I.V.} \\ &\leq a(k_3, a(k_3, n)) \text{ mit } k_3 = \max(k_1, k_2) \\ &\leq \underbrace{a(k_3, a(k_3, \dots, a(k_3, n) \dots))}_{k_3\text{-mal}} \\ &= a(k_3 + 1, n) \end{aligned}$$

2. P hat die Form LOOP x_i DO P_1 END

$$\begin{aligned} f_P(n) &\leq \underbrace{f_{P_1}(f_{P_1}(\dots f_{P_1}(n) \dots))}_{x_i\text{-mal}} \\ &< \underbrace{a(k_1, a(k_1, \dots, a(k_1, n) \dots))}_{x_i\text{-mal}} \text{ (nach I.V.)} \\ &\leq \underbrace{a(k_1, a(k_1, \dots, a(k_1, n) \dots))}_{n\text{-mal}} \left(\sum_{i=0}^{\infty} x_i \leq n \right) \\ &= a(k_1 + 1, n) \end{aligned}$$

Damit ist das Lemma bewiesen. Wir wenden uns nun wieder dem eigentlichen Satz zu.

Satz (WHILE_{total} $\not\subseteq$ LOOP_{total})

Beweis durch Widerspruch: Angenommen die Ackermann-Funktion a ist LOOP-berechenbar, dann ist auch die Funktion $g(n) = a(n, n)$ LOOP-berechenbar. Nach Definition von f_P gilt $g(n) \leq f_P(n)$ für ein Programm P, das g berechnet. Nach dem Lemma gibt es ferner ein k mit $f_P(n) < a(k, n) \forall n \geq k$. Wir betrachten nun den Sonderfall $n = k$:

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

Dies ist ein Widerspruch. □

Definition (GOTO) Ein GOTO-Programm ist eine endliche Folge von GOTO-Anweisungen $A_1; A_2; \dots; A_n$ wobei eine Anweisung eine der folgenden Formen hat:

1. $x_i := x_i \pm 1$, mit $i \in \mathbb{N}$
2. IF $x_i = 0$ GOTO j , mit $i, j \in \mathbb{N}$

Semantik: Wenn in Register x_i eine 0 steht, führt die Registermaschine die Ausführung des Programms bei der j -ten Anweisung fort.

3. HALT

Beispiel: $f(n) = 2n$

- (1) IF $x_1 = 0$ GOTO 6;
- (2) $x_0 := x_0 + 1$;
- (3) $x_0 := x_0 + 1$;
- (4) $x_1 := x_1 - 1$;
- (5) IF $x_2 = 0$ GOTO 1;
- (6) HALT

Beispiel: $f(n, m) = n + m$

- (1) IF $x_1 = 0$ GOTO 5;
- (2) $x_0 := x_0 + 1$;
- (3) $x_1 := x_1 - 1$;
- (4) IF $x_3 = 0$ GOTO 1;
- (5) IF $x_2 = 0$ GOTO 9;
- (6) $x_0 := x_0 + 1$;
- (7) $x_2 := x_2 - 1$;
- (8) IF $x_3 = 0$ GOTO 5;
- (9) HALT

3 WHILE = GOTO = TM

Satz (WHILE = GOTO = TM) Die Menge der WHILE-berechenbaren Funktionen und die Menge der GOTO-berechenbaren Funktionen stimmen mit der Menge der Turing-berechenbaren Funktionen überein.

Beweis: $\text{WHILE} = \text{GOTO} = \text{TM} \Leftrightarrow \text{GOTO} \subseteq \text{WHILE} \subseteq \text{TM} \subseteq \text{GOTO}$

(i) GOTO \subseteq WHILE

Wir haben ein GOTO-Programm $A_1; \dots; A_n$ gegeben und möchten zeigen, dass dieses durch ein WHILE-Programm simuliert werden kann. Wir wählen ein noch unbenutztes Register x_m und simulieren wie folgt:

```

 $x_m := 1$ ;
WHILE  $x_m \neq 0$  DO
  IF  $x_m = 1$  THEN  $\phi(A_1)$  END;
  IF  $x_m = 2$  THEN  $\phi(A_2)$  END;
  ...
  IF  $x_m = n$  THEN  $\phi(A_n)$  END
END
    
```

Die Funktion ϕ wandelt dabei eine GOTO-Anweisung in ein WHILE-Programm um. Wir definieren sie wie folgt:

$$\phi(\text{HALT}) = x_m := 0$$

$$\phi(x_i := x_i \pm 1) = x_i := x_i \pm 1; x_m := x_m + 1$$

$$\phi(\text{IF } x_i = 0 \text{ GOTO } j) = \text{IF } x_i = 0 \text{ THEN } x_m := j \text{ ELSE } x_m := x_m + 1 \text{ END}$$

x_m arbeitet dabei wie ein Programcounter in einer CPU. □

(ii) WHILE \subseteq TM

Ohne Beschränkung der Allgemeinheit nutzt das zu simulierende WHILE-Programm nur die Register x_0, \dots, x_n . Wir simulieren dieses Programm mit einer $(n+1)$ -Band-Turingmaschine. Auf jedem Band speichern wir den Inhalt des entsprechenden Registers in Binärcodierung. Unser Bandalphabet ist folglich $\{0, 1, \#\}$. Es ist zu zeigen, dass alle WHILE-Programme von einer Turingmaschine simuliert werden können.

Induktionsanker $x_i := x_i \pm 1$

Inkrementierung und Dekrementierung lassen sich leicht als Turingmaschine ausdrücken. Wir operieren dabei nur auf dem i -ten Band. Die Dekrementierung von 0 benötigt ggf. wieder eine Sonderbehandlung.

Induktionsvoraussetzung T_1 und T_2 sind Turingmaschinen, die P_1 bzw. P_2 simulieren.

Induktionsschritt Zwei Fälle sind zu zeigen:

- $P_1; P_2$
Wir konstruieren T' wie folgt: T' startet im Startzustand von T_1 . Wenn sie in einen Endzustand von T_1 gerät, geht sie in den Startzustand von T_2 über.
- WHILE $x_i \neq 0$ DO P_1 END
Wir konstruieren T' wie folgt: T' startet im Zustand S und prüft, ob das i -te Band 1en enthält. Falls ja, geht sie in den Startzustand von T_1 über. Sobald sie in einen Endzustand von T_1 gerät, geht sie wieder in Zustand S über. Falls das i -te Band keine 1 enthält geht sie in einen Endzustand über.

□

(iii) TM \subseteq GOTO

O.B.d.A. ist eine deterministische Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_1, \#, E)$ gegeben mit $Z = \{z_1, z_2, \dots, z_k\}$ und $\Gamma = \{a_1, a_2, \dots, a_m\}$. Es ist zu zeigen, dass die Turingmaschine mit einem GOTO-Programm simuliert werden kann. Dazu müssen wir uns zunächst überlegen, wie wir eine Konfiguration der Turingmaschine der Form

$$a_{i_1} \dots a_{i_p} z_l a_{j_1} \dots a_{j_q}$$

in Registern speichern können. Wir vereinbaren eine Darstellung, die eine Konfiguration in nur

drei Registern x , y und z codieren kann:

$$x = (i_1, i_2, \dots, i_p)_b = \sum_{\gamma=1}^p i_\gamma \cdot b^{p-\gamma}$$

$$y = (j_q, \dots, j_2, j_1)_b = \sum_{\gamma=1}^q j_\gamma \cdot b^{\gamma-1}$$

$$z = l$$

Dabei ist b eine beliebige Basis mit $b > |\Gamma|$. Um mit dieser Darstellung umgehen zu können, benötigen wir insbesondere Multiplikation und Division.

Lemma: Wir dürfen Multiplikation, IF-THEN-ELSE usw. in GOTO-Programmen benutzen, da $\text{WHILE} \subseteq \text{GOTO}$.

Beweis: $\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$ kann in GOTO-Programmen simuliert werden. Dabei wird P rekursiv in ein GOTO-Programm umgewandelt und die Sprungmarken entsprechend angepasst.

(j)	IF $x_i = 0$ GOTO $k + 1$;
$(j + 1)$	P ;
	...
(k)	IF $x_n = 0$ GOTO j ;
$(k + 1)$	$x_n := x_n - 1$

Lemma: Wir können die Operationen $x_i \text{ DIV } x_j$ und $x_i \text{ MOD } x_j$ simulieren.

Beweis:

```

result := 0;
rest := 0;
LOOP  $x_i$  DO
  rest := rest + 1;
  IF rest =  $x_j$  THEN
    result := result + 1;
    rest := 0
  END
END

```

Beweis (TM \subseteq GOTO): Wir teilen das GOTO-Programm, mit dem wir die Turingmaschine simulieren werden, in drei Teile, die hintereinander ausgeführt werden.

Teil 1 Zunächst müssen wir die Parameter, die in den Registern x_1 bis x_k liegen, in die Startkonfiguration der Turingmaschine umwandeln, sprich in die Register x und y codieren. Mit Hilfe von Division und Multiplikation beherrschen wir die Umrechnung von

Zahlen in andere Basen, sodass dieser Teil nur eine Fleißarbeit ist. Insbesondere ist er unabhängig von Übergangsfunktion der Turingmaschine. Wichtig ist nur, in welcher Reihenfolge wir die Zeichen in Γ aufzählen und wie viele Parameter es gibt.

Teil 2 Im zweiten Teil des GOTO-Programmes simulieren wir die Turingmaschine. Dies machen wir mittels einer Fallunterscheidung von Zustand z und gelesenen Zeichen a :

(L_1)	$a := y \text{ MOD } b$; IF $z = 1$ THEN IF $a = 1$ GOTO $L_{1,1}$ END; ... IF $a = m$ GOTO $L_{1,m}$ END; END; ... IF $z = k$ THEN IF $a = 1$ GOTO $L_{k,1}$ END; ... IF $a = m$ GOTO $L_{k,m}$ END; END; $\phi(1, 1)$; GOTO L_1 ; ... $(L_{k,m})$ $\phi(k, m)$; GOTO L_1 ; ...
---------	--

Dabei wird $\phi(i, j)$ durch ein Codesegment ersetzt, das den Zustandsübergang $\delta(z_i, a_j)$ simuliert. Wenn z_i ein Endzustand ist, geht das Programm über zu Teil 3. Wir betrachten hier ϕ für den Fall $\delta(z_i, a_j) = (z_{i'}, a_{j'}, L)$, der Fall $\delta(z_i, a_j) = (z_{i'}, a_{j'}, R)$ ist analog.

```

 $z := i'$ ;  

 $y := y \text{ DIV } b$ ;  

 $y := b \cdot y + j'$ ; // Zeichen schreiben  

 $y := b \cdot y + (x \text{ MOD } b)$ ; // Kopfbewegung  

 $x := x \text{ DIV } b$ 

```

Teil 3 Nun müssen wir noch die Endkonfiguration, die in y steht, in eine natürliche Zahl umwandeln, die wir in x_0 schreiben.

```

WHILE  $y \neq 0$  DO
   $a := y \text{ MOD } b$ ;  

   $y := y \text{ DIV } b$ ;  

   $x_0 := x_0 \text{ MUL } 2$ ;  

  IF  $a = '1'$  THEN  $x_0 := x_0 + 1$  END
END

```

4 Weitere Berechnungsmodelle

Berechnungsuniverselle Modelle:

- PDA mit zwei Stacks
- Registermaschine mit einem Register sowie DIV, MOD und MUL
- Registermaschine mit zwei Registern
- Nicht-löschende Turingmaschine
- λ -Kalkül
- μ -rekursive Funktionen

Ferner entspricht die Klasse LOOP der Klasse der primitiv-rekursiven Funktionen.

Literaturverzeichnis

- (1) ERK, PRIESE: Theoretische Informatik, Springer 2008
- (2) UWE SCHÖNING: Theoretische Informatik - kurzgefaßt, Spektrum Akad. Verlag 1995