Grundlagen Theoretischer Informatik 2

WiSe 2009/10 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Grundlagen Theoretischer Informatik 2 Gesamtübersicht

- Organisatorisches; Einführung
- Ersetzungsverfahren: Grammatiken und Automaten
- Rekursionstheorie-Einführung
- Komplexitätstheorie-Einführung

LOOP-Berechenbarkeit; Nachtrag 1

Anmerkungen:

1. alle LOOP-berechenbaren Funktionen sind total:

   Jedes LOOP-Programm hält auf *jeder* Eingabe!

   (Beweis: Induktion über den Aufbau der LOOP-Programme...)

2. Es gibt also Turing-berechenbare Funktionen,
   die nicht LOOP-berechenbar sind...

3. Sogar: Nicht einmal jede *totale* Turing berechenbare Funktion
   ist LOOP-berechenbar...

LOOP-Berechenbarkeit; Nachtrag 2

Weitere Übung: Cantor'sche Bijektion (*Paarfunktion*)

$$\langle x, y\rangle \;=\; y + \sum_{i \le x+y} i \;=\; y + \frac{(x+y)(x+y+1)}{2}$$

$\langle x, y\rangle$	0	1	2	3	4	5	...
0	0	2	5	9	14	20	...
1	1	4	8	13	19	26	...
2	3	7	12	18	25	33	...
3	6	11	17	24	32	41	...
4	10	16	23	31	40	50	...
5	15	22	30	39	49	60	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Folg.: Es gibt nur abzählbar viele ration. Zahlen.

Sowohl die Bijektion von $\mathbb{N}\times\mathbb{N} \to \mathbb{N}$ als auch die einzelnen Komponenten (Projektionen) $p_1, p_2 : \mathbb{N} \to \mathbb{N}$ mit $z = \langle p_1(z), p_2(z)\rangle$ sind LOOP-berechenbar:
Berechnung von $y + \sum_{i \le x+y} i$ benötigt i.W. LOOP-Schleife mit Additionen;
für die Komponenten $p_1(z), p_2(z)$ gilt:

- Suche (von $k = 0, ..., z$) nach größtem k mit $\sum_{i \le k} i \le z$ (also: $k = p_1(z) + p_2(z)$).
- Dann $p_2(z) = z - \sum_{i \le k} i$
  und $p_1(z) = k - p_2(z)$.

Erweiterung der LOOP-Programme zu WHILE-Programmen:

- Jedes LOOP-Programm ist ein WHILE-Programm.

- Für jede Variable x_i und jedes WHILE-Programm P ist auch
  `WHILE` $x_i \neq 0$ `DO` P `END`
  ein WHILE-Programm.

Semantik der WHILE-Schleife:
P wird so oft ausgeführt, bis der (sich ändernde!) Inhalt von x_i
zu Beginn des Schleifenrumpfes P den Wert 0 hat.

WHILE-Berechenbarkeit

Eine Funktion $f : \mathbb{N}^k \multimap \mathbb{N}$ heißt *WHILE-berechenbar*, falls

- es ein WHILE-Programm P gibt, so dass P
- gestartet mit $n_1, ..., n_k$ aus $\mathbb{N}$ in den Variablen $x_1, ..., x_k$
- und 0 in allen anderen Variablen
- mit $f(n_1, ..., n_k)$ in x_0 stoppt, falls $f(n_1, ..., n_k)$ definiert ist,
- und nie stoppt, falls $f(n_1, ..., n_k)$ nicht definiert ist.

WHILE vs. LOOP

Bei WHILE-Programmen kann man auf die LOOP-Anweisung sogar komplett verzichten, '`LOOP` x_j `DO` P `END`' wird simuliert durch

$$x_k := x_j;$$
$$\texttt{WHILE } x_k \neq 0 \texttt{ DO } x_k := x_k - 1; \ P \texttt{ END}$$

Dabei sei x_k eine Variable, die im Programm P nicht verwendet wird.
Alle LOOP-berechenbaren Funktionen sind WHILE-berechenbar.

Die Umkehrung gilt nicht, z.B. für die partielle Funktion f mit

$$f(x) := \begin{cases} 42 & \text{falls } x = 0 \\ \text{undefiniert} & \text{falls } x > 0 \end{cases}$$

f ist sicher nicht LOOP-, aber WHILE-berechenbar:

$$\texttt{WHILE } x_1 \neq 0 \texttt{ DO } x_1 := x_1 + 1 \texttt{ END; } x_0 := 42$$

WHILE vs. LOOP

Es gibt sogar *totale* Funktionen, die nicht LOOP-, aber WHILE-berechenbar sind, z.B. die *Ackermann-Funktion*. Vereinfachte Definition:

$$\begin{aligned} a(0,m) &= m+1 \\ a(n+1,0) &= a(n,1) \\ a(n+1,m+1) &= a(n,a(n+1,m)) \end{aligned}$$

Werte von $a(n,m)$:

n/m	0	1	2	3	4	m
0	1	2	3	4	5	$m+1$
1	2	3	4	5	6	$m+2$
2	3	5	7	9	11	$2m+3$
3	5	13	29	61	125	$8 \cdot 2^m - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$	$a(3, 2^{65536} - 3)$	$a(3, a(4,3))$	$2^{2^{\cdot^{\cdot^{2}}}} - 3$ (m + 3 Terme im Turm)

$a(n,n)$ wächst schneller als jede LOOP-berechenbare Funktion.
Die Inverse α hiervon ist “praktisch konstant”, z.B. für UNION/FIND-Algorithmus Abschätzung.
Originalarbeit von W. Ackermann siehe `doi:10.1007/BF01459088`.

Vergleich WHILE-Programm mit Mehrband-Turingmaschinen 1

- Per Induktion über Aufbau von WHILE-Programmen und unter Benutzung der Grundprogrammierbausteine für MTMs folgt:
  Für jedes WHILE-Programm gibt es eine MTM, die die gleiche Funktion berechnet.
- Hierbei: Verwende pro benutzter Variable x_i ein eigenes Band i der MTM, nutze die bereits betrachteten MTM-Konstruktionen

$\rightsquigarrow$ Satz: Jede WHILE-berechenbare Zahlenfunktion ist Turing-berechenbar.

Vergleich WHILE-Programm mit Mehrband-Turingmaschinen 2

Zu zeigen: Satz: Turing-berechenbare Zahlenfunktionen sind WHILE-berechenbar.

Gegeben Turingmaschine

$$TM = (S, E, A, \delta, s_0, \square, F)$$

zur Berechnung einer Zahlenfunktion f.

TM wird durch WHILE-Programm aus drei Teilen simuliert:

1. Eingabe $n_1, ..., n_l$ für f wird in drei Zahlen x, y, z umgerechnet, die die Startkonfiguration der TM darstellen,
   x, y, z werden dabei in Variablen gespeichert!
2. Die Berechnung durch TM wird Schritt für Schritt durch entsprechende Änderung der drei Zahlen x, y, z nachvollzogen.
3. Aus den Zahlen x, y, z, die die Endkonfiguration beschreiben, wird der Ausgabewert extrahiert und in x_0 gespeichert.

Vergleich WHILE-Programm mit Mehrband-Turingmaschinen 3 Details

Es sei $S = \{s_0, ..., s_k\}$ und $A = \{a_1, ..., a_m\}$.

Wähle $b \in \mathbb{N}$ mit $b > m$.

Beschreibe eine Konfiguration

$$a_{i_1}...a_{i_p}\ s_l\ a_{j_1}...a_{j_q}$$

durch drei Zahlen x, y, z:

$$x = (i_1...i_p)_b \quad y = (j_q...j_1)_b \quad z = l$$

$$(i_1...i_p)_b = \sum_{l=1}^{p} i_l \cdot b^{p-l}$$

$\rightsquigarrow$ Notiere Worte als Zahlen zur Basis b.

Wichtig: Bei y ist die Reihenfolge der Ziffern vertauscht!

- x: Bandinschrift links des Lese-Schreibkopfes
- y: Bandinschrift rechts des Lese-Schreibkopfes, inklusive dem Zeichen unter dem Kopf
- z: aktueller Zustand der TM

Vergleich WHILE-Programm mit Mehrband-Turingmaschinen 4 Details

Simulation der Schritte der TM durch Wiederholung einer großen Verzweigung:

```
WHILE Zustand s ist kein Endzustand DO
    Bestimme Zeichen a unter dem Kopf;
    IF (s = s_0) und (a = a_1):
       simuliere δ(s_0, a_1) in x, y, z;
    ELSE IF...
       ...
    ELSE IF (s = s_k) und (a = a_m):
       simuliere δ(s_k, a_m) in x, y, z;
END WHILE;
```

Vergleich WHILE-Programm mit Mehrband-Turingmaschinen 5 Details

- Bestimmung des *Zeichens* a durch Berechnung der *Zahl* 'y MOD b'
- Vergleich '$a = a_j$' entspricht Test 'y MOD $b = j$'
- Vergleich '$s = s_i$' entspricht Test '$z = i$'.

Insgesamt $(k+1) \cdot m$ verschiedene Fälle:

- für $k+1$ Zustände s und
- für m Symbole a im Arbeitsalphabet A (unter dem Kopf).

Simulation der Übergangsfunktion $\delta(s_i, a_j) = (s_{i'}, a_{j'}, B)$

Erstellung der Folgekonfiguration in den Variablen x, z, y

Hier nur $B = L$ bei $x \neq 0$ als Beispiel:

```
z := i';
y := y DIV b;
y := b * y + j';
y := b * y + (x MOD b);
x := x DIV b;
```

- Variable für z direkt passend für Zustand $s_{i'}$ ändern,
- $a_{j'}$ als neues letztes Zeichen in y vermerken (statt a_j) und
- letztes Zeichen aus x in y übertragen (Kopf nach links!)
- bei $x = 0$ statt $y := b * y + (x \text{ MOD } b)$ nun $y := b * y + j_\square$ (bei $a_{j_\square} = \square$)

Erster Teil (Kodierung der Eingabe)
und dritter Teil (Extraktion der Ausgabe):
i.W. analog...

Als Folgerung aus der Konstruktion ergibt sich:

Satz: (Kleenesche Normalform für WHILE-Programme)

Jedes Turing-Programm kann durch ein WHILE-Programm mit nur einer einzigen WHILE-Schleife berechnet werden.

Wörter und Zahlen

Sei $E = \{e_1, e_2, \ldots, e_n\}$ ein endliches Alphabet.

Definiere Bijektion $\nu : \mathbb{N} \to E^*$ durch Sortieren (liefert *Längenlexikographische Ordnung*):

- Wörter in E^* nach Länge sortiert
- Wörter gleicher Länge alphabetisch sortiert

i	0	1	2	$\ldots$	n	$n+1$	$n+2$	$\ldots$	n^2+n	n^2+n+1	$\ldots$
$\nu(i)$	λ	e_1	e_2	$\ldots$	e_n	e_1e_1	e_1e_2	$\ldots$	e_ne_n	$e_1e_1e_1$	$\ldots$

Wort- und Zahlfunktionen

- Ist $f : \mathbb{N} \multimap \mathbb{N}$ einstellige Zahlfunktion,
  so ist $\Gamma(f) := \nu \circ f \circ \nu^{-1}$ eine Wortfunktion:

$$\Gamma(f) = \nu \circ f \circ \nu^{-1} : \Sigma^* \xrightarrow{\nu^{-1}} \mathbb{N} \overset{f}{\multimap} \mathbb{N} \xrightarrow{\nu} \Sigma^*$$

- Ist $g : \Sigma^* \multimap \Sigma^*$ eine Wortfunktion,
  so ist $\Delta(f) = \nu^{-1} \circ g \circ \nu$ einstellige Zahlfunktion
- Es ist $\Delta \circ \Gamma(f) = \nu^{-1} \circ \left(\nu \circ f \circ \nu^{-1}\right) \circ \nu = f$ und analog $\Gamma \circ \Delta(g) = g$.

$\Rightarrow \Delta$ und Γ definieren Bijektion zwischen:
Menge aller Zahlenfunktionen $f : \mathbb{N} \multimap \mathbb{N}$ und
Menge aller Wortfunktionen $g : \Sigma^* \multimap \Sigma^*$.

Berechenbare Wort- und Zahlfunktionen

Satz:

1. Ist $f : \mathbb{N} \multimap \mathbb{N}$ eine (Turing-)berechenbare Zahlfunktion, so ist $\nu \circ f \circ \nu^{-1}$ eine (Turing-)berechenbare Wortfunktion.

2. Ist $g : \Sigma^* \multimap \Sigma^*$ eine (Turing-)berechenbare Wortfunktion, so ist $\nu^{-1} \circ g \circ \nu$ eine (Turing-)berechenbare Zahlfunktion.

Beweis von (1):

- Sei $f : \mathbb{N} \multimap \mathbb{N}$ berechenbare Zahlfunktion.
- Betrachte entsprechende Turingmaschine TM:

  TM berechne aus $bin(n)$ den Wert $bin(f(n))$
  (für n aus dem Definitionsbereich von f)
- Aufgabe: konstruiere Turingmaschine TM',
  die aus w das Wort $\nu \circ f \circ \nu^{-1}(w)$ berechnet,
  (für w im Definitionsbereich von $\nu \circ f \circ \nu^{-1}$)

Vorgehensweise von $\mathcal{TM}'$:

- Erst aus Wort w das Wort $\mathrm{bin}(\nu^{-1}(w))$ bestimmen
- Auf $\mathrm{bin}(\nu^{-1}(w))$ die Turingmaschine $\mathcal{TM}$ anwenden.
- Auf (binär kodiertes) $f(\nu^{-1}(w))$ wiederum ν anwenden,
  mit Resultat $\nu(f(\nu^{-1}(w)))$.
- Das Ganze geht mit Turingmaschinen, da die Funktion, die Wörter w auf $\mathrm{bin}(\nu^{-1}(w))$ abbildet, sicher berechenbar ist, ebenso ihre Umkehrung.

Beweis von (2): analog...

$\rightsquigarrow$ Berechenbarkeitsbegriffe für Zahlen / für Wörter gleichwertig.

$\rightsquigarrow$ *Church(-Turing) These für Wortfunktionen*

Im Folgenden auch analog Funktionen $\mathbb{N} \multimap \Sigma^*$ und $\Sigma^* \multimap \mathbb{N}$ verwendet...

Wieviele berechenbare Funktionen gibt es?

Satz: Die Menge aller von Turingmaschinen berechneten Funktionen über einem gegebenen Alphabet Σ ist abzählbar, es gibt zu jedem Σ eine totale Funktion h mit Definitionsbereich $\{0,1\}^*$, deren Bild alle(!) berechenbaren Funktionen $g : \Sigma^* \multimap \Sigma^*$ umfasst.

Konstruiere dazu Liste aller Turing-berechenbaren Funktionen, beispielsweise wie folgt:

- Beschränkung auf deterministische Turingmaschinen $TM = (S, \Sigma, A, \delta, s_0, \square, F)$
- Festes Alphabet Σ mit $\{0,1,\#\} \subseteq \Sigma$
- $\delta(s,a)$ undefiniert für alle $a \in A$ und Endzustände $s \in F$
- $\delta(s,a)$ definiert für alle $a \in A$ und Nicht-Endzustände $s \notin F$
- Falls bei Eingabe w mit Konfiguration us_fv hält:

$$f_{TM}(w) := \text{ das längste Präfix } \in \Sigma^* \text{ von } v$$

Fortsetzung der Konstruktion O.B.d.A.:

- $S = \{s_0, s_1, \ldots, s_n\}$ mit Startzustand s_0
- Endzustände am Listenende: $F = \{s_{n-e+1}, \ldots, s_n\}$ mit $e = |F|$
- $A = \{a_0, a_1, \ldots, a_k\}$
- $E \subset A$ mit $E = \{a_0, a_1, \ldots, a_{l-1}\}$ für $l = |E|$
- $\square = a_d$ für ein d

Übergänge der Form '$\delta(s_i, a_j)$ enthält (s_t, a_m, B)' beschreibbar durch

$$\Delta_{i,j} := \#\#bin(i)\#bin(j)\#bin(t)\#bin(m)\#bin(b)$$

mit $b = 0$, wenn $B = L$, $b = 1$, wenn $B = N$ und $b = 2$, wenn $B = R$.

$\Rightarrow$ TM beschreibbar durch n, e, k, l, d und Angabe der $\delta(s_i, a_j)$,
d.h. als Wort über $\{0, 1, \#\}$ in folgender Form:

$$bin(n)\#bin(e)\#bin(k)\#bin(l)\#bin(d)\#\Delta_{0,0}\ldots\Delta_{n-e,k}$$

Codiere z.B. $0 \mapsto 00$, $1 \mapsto 01$ und $\# \mapsto 11$
$\Rightarrow$ *jede* Turingmaschine durch ein $w \in \{0, 1\}^*$ beschreibbar.

Ziel: Aufzählung aller berechenbaren Wortfunktionen, dazu

- Zu jedem $w \in \{0,1\}^*$ definiere M_w durch

$$M_w := \begin{cases} M, & \text{falls } w \text{ eine Beschreibung der Maschine } M \text{ ist .} \\ M^{ud}, & \text{sonst} \end{cases}$$

- Dabei sei M^{ud} beliebig gewählt, z.B.

$$M^{ud} := (\{s_0, s_1\}, E, E \cup \{\Box\}, \delta, \Box, \{s_1\})$$

  mit $\delta(s_0, a) = (s_0, a, N)$ für $a \in A$ (d.h. M^{ud} hält nie an...)

- h_w sei die von M_w berechnete Wortfunktion.

$\Rightarrow w$ ist 'Programm' für die Funktion $h_w : E^* \multimap E^*$

$\Longrightarrow$ Abbildung $w \mapsto h_w$ ist 'Programmiersprache'

Notationen

Sei Σ ein Alphabet.
Eine *Notation* (Programmiersprachenineterpretation) für die berechenbaren Funktionen $g : \Sigma^* \multimap \Sigma^*$ ist eine surjektive Funktion

$$h' : \{0,1\}^* \to \{ \text{ berechenbare Wortfunktionen über } \Sigma^* \}$$

- Jedem $w \in \{0,1\}^*$ (also jedem Programmtext) wird eine berechenbare Wortfunktion zuordnet.
- Zu jeder berechenbaren Wortfunktion g gibt es ein Wort $w \in \{0,1\}^*$ derart, dass $h'(w)$ gerade die Funktion g ist. Jedes solche g lässt sich also programmieren.

Wir schreiben oft auch h'_w für die Funktion $h'(w)$.

Zwei wichtige Eigenschaften von Notationen

utm-Eigenschaft (universelle Turingmaschine)

Die Wortfunktion, die Wörter $w\#x$ für $w \in \{0,1\}^*$ und $x \in \Sigma^*$ auf $h'_w(x)$ abbildet, ist berechenbar.

smn-Eigenschaft (Projektion)

Ist $g : \Sigma^* \multimap \Sigma^*$ eine berechenbare Wortfunktion, so gibt es eine totale berechenbare Wortfunktion r derart, dass für alle $x \in \{0,1\}^*$ und alle $y \in \Sigma^*$ gilt:

$$g(x\#y) = h'_{r(x)}(y)$$

Beachte: Paare von Wörtern $x\#y$ lassen sich bijektiv mit der Cantorschen Paarfunktion und ν bzw. ν^{-1} auf Wörter bzw. Zahlen abbilden.

Satz: Die von uns oben definierte Funktion $\mathfrak{h}$, die jedes Wort w aus $\{0,1\}^*$ auf eine berechenbare Funktion $\mathfrak{h}_w : \Sigma^* \multimap \Sigma^*$ abbildet, ist eine Notation der berechenbaren Wortfunktionen.

$\mathfrak{h}$ hat zudem die utm-Eigenschaft und die smn-Eigenschaft.

Beweis: Nach Konstruktion ist $\mathfrak{h}$ Notation der berechenbaren Wortfunktionen:

- $\mathfrak{h}_w$ ist berechenbare Funktion für jedes w.
- Jede (normierte) Turingmaschine wird durch ein w kodiert.

Zur utm-Eigenschaft: Bilde *universelle Turingmaschine* $\mathfrak{U}$ wie folgt:

- Bei Eingabe $w\#x$ bestimmt $\mathfrak{U}$ die von w codierte Maschine M (d.h. direkt $M = M_w$ oder aber $M = M^{ud}$)
- Danach simuliert $\mathfrak{U}$ die Maschine M auf der Eingabe x

$f_{\mathfrak{U}}$ ist dann die für die utm-Eigenschaft notwendige Funktion!

Zur smn-Eigenschaft

Zur berechenbaren Wortfunktion $g : E^* \multimap E^*$ sei M_g fixierte TM, die g berechnet.
Betrachte eine Turingmaschine R, die wie folgt arbeitet:

- R liest ihre Eingabe x und modifiziert dann (eine Codierung von) M_g zu (der Codierung einer TM) M_g^x mit folgender Arbeitsweise:

  Bei einer Eingabe y schreibt M_g^x zunächst das Wort $x\#$ links vor das Wort y und arbeitet dann weiter wie M_g.

- x wird dabei in der Zustandsmenge von M_g^x codiert.

- R erzeugt dann als Ausgabe eine Codierung von M_g^x.

- M_g wird in der Zustandsmenge von R codiert.

R erzeugt also aus einer Eingabe x (die Codierung von) M_g^x mit

$$h_{f_R(x)}(y) = f_{M_g^x}(y) = f_{M_g}(x\#y) = g(x\#y)$$

f_R ist also das in der smn-Eigenschaft geforderte (totale!) r.

Erinnerung

Berechnung der Summe zweier natürlicher Zahlen m und n als Funktion $f : \mathbb{N}^2 \to \mathbb{N}$ mit $f(m, n) = m + n$:

```
import java.util.Vector;
import java.math.BigInteger;
class Addition {
  public static void main(String args[]) {
    BigInteger m=new BigInteger(args[0]);
    BigInteger n=new BigInteger(args[1]);
    while (n.compareTo(BigInteger.ZERO) > 0) {
      n = n.subtract(BigInteger.ONE);
      m = m.add(BigInteger.ONE);
    }
    System.out.println(m.toString());
  }
}
```

Anwendung der smn-Eigenschaft in JAVA

Betrachte Funktion zu Addition zweier Zahlen in JAVA (s.o.),
wende smn-Eigenschaft für ersten Parameter m an:

```
class SMN {
public static void main(String args[]) {
System.out.println(¨import java.util.Vector;¨);
System.out.println(¨import java.math.BigInteger;¨);
System.out.println(¨class Addition {¨);
System.out.println(¨public static void main(String args[]){¨);
System.out.println(¨BigInteger m=new BigInteger(\”¨
                + args[0] + ¨\”);¨ );
System.out.println(¨BigInteger n=new BigInteger(args[0]);¨);
System.out.println(¨while(n.compareTo(BigInteger.ZERO)>0){¨);
System.out.println(¨n = n.subtract(BigInteger.ONE);¨);
System.out.println(¨m = m.add(BigInteger.ONE);¨);
System.out.println(¨}¨);
System.out.println(¨System.out.println(m.toString());¨);
System.out.println(¨}¨);
System.out.println(¨}¨);
} }
```

Veranschaulichungen der Eigenschaften

Universelle Turingmaschinen sind Interpreter, die bei Eingabe einer Notation einer berechenbaren Funktion f (also z.B. einem JAVA- oder Turingmaschinenprogrammtext für f) und dessen Argument x den Wert von $f(x)$ ausrechnet.

Die smn-Eigenschaft hat eher was mit Compilern zu tun:
Bei Eingabe der Notation einer berechenbaren Funktion f wird ein JAVA-Programm erzeugt, das bei Eingabe von x den Wert $f(x)$ ausrechnet.
Das vorige "Anwendungsbeispiel" funktioniert sogar etwas wörtlicher!
Dies entspricht dem "Currying" in funktionalen Programmiersprachen wie ML oder Haskell.

Eine typische Aufgabe

Zu zeigen ist: Es existiert eine totale und berechenbare Funktion $g\colon \mathbb{N}^2 \to \mathbb{N}$, so dass für $i, j, x \in \mathbb{N}$ gilt: $h_{g(i,j)}(x) = h_i(x) + h_j(x)$.

Erinnnere Cantorsche Paarfunktion $\langle \cdot, \cdot \rangle$.
Definiere $\psi(\langle i, j \rangle, x) = h_i(x) + h_j(x)$.
ψ ist berechenbar wegen utm-Eigenschaft.
Nach dem smn-Theorem gilt: $h_{\hat{g}(\langle i,j \rangle)}(x) = \psi(\langle i, j \rangle, x)$ für totale und berechenbare Funktion $\hat{g}\colon \mathbb{N} \to \mathbb{N}$,
aus der das gesuchte g sich mit der Inversen der Cantorschen Bijektion ergibt.

Vergleichbarkeit von Notationen

Eine Notation h'' ist in eine Notation h' *übersetzbar*, wenn es eine totale(!) berechenbare Wortfunktion (*Übersetzungsfunktion*) $u : \{0,1\}^* \to \{0,1\}^*$ gibt mit $h''_w(x) = h'_{u(w)}(x)$ für alle $w \in \{0,1\}^*$ und $x \in E^*$, d.h. $h''_w = h'_{u(w)}$.
Jedes 'Programm' w der 'Programmiersprache' h'' kann also dann automatisch in ein 'Programm' der 'Sprache' h' übersetzt werden.
Notationen h' und h'' heißen *äquivalent*, wenn man jede in die andere übersetzen kann.

Satz: (Äquivalenzsatz von Rogers) Eine Notation h' für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation h äquivalent, wenn sie die utm-Eigenschaft und die smn-Eigenschaft hat.

Beweisrichtung $\Rightarrow$:

Sei h wie oben und sei h' weitere äquivalente Notation für berechenbaren Wortfunktionen.

(a) utm-Eigenschaft für h: f_h mit $f_h(w\#x) := h_w(x)$ ist berechenbar.
Definiere f' durch $f'(w\#x) := f_h(u'(w)\#x)$
$\rightsquigarrow f'$ berechenbar und

$$h'_w(x) = h_{u'(w)}(x) = f_h(u'(w)\#x) = f'(w\#x)$$

D.h. f' zeigt utm-Eigenschaft für h'.
(b) Sei $g : E^* \multimap E^*$ irgendeine berechenbare Wortfunktion.
smn-Eigenschaft für h: r mit $h_{r(x)}(y) := g(x\#y)$ ist berechenbar
Dann ist $u \circ r$ berechenbar (und total) mit

$$g(x\#y) = h_{r(x)}(y) = h'_{u\circ r(x)}(y)$$

D.h. h' hat auch die smn-Eigenschaft.

Beweisrichtung $\Leftarrow$:

h' besitze ebenfalls die utm- und die smn-Eigenschaft:

utm-Eigenschaft für h: f_h mit $f_h(w\#x) = h_w(x)$ berechenbar

Wende smn-Eigenschaft für h' auf f_h an, mit entsprechendem r':

$$h_w(x) = f_h(w\#x) = h'_{r'(w)}(x)$$

Damit Übersetzung von h nach h' mit r'

Analog: utm-E. für h' und smn-E. für $h \Rightarrow h'$ in h übersetzbar.

Damit: h und h' äquivalent!

Ein Ersatz für smn

Satz: Eine Notation h' für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation h äquivalent, wenn sie die utm-Eigenschaft und die folgende Eigenschaft (*effektive Komposition*) hat:

> Es gibt eine totale berechenbare Wortfunktion $r : \{0,1\}^*\#\{0,1\}^* \to \{0,1\}^*$ derart, dass für alle v, w aus $\{0,1\}^*$ und alle x aus E^* gilt:
>
> $$h'_v(h'_w(x)) = h'_{r(v\#w)}(x)$$

r bestimmt also aus zwei 'Programmen' v, w ein 'Programm' $r(v\#w)$ für die Komposition der Funktionen h'_v und h'_w.

(Ohne Beweis.)

- Alle 'vernünftigen' Notationen der berechenbaren Wortfunktionen sind äquivalent (mit 'vernünftig' = 'utm+smn' = 'utm+effKomp')
- Jede zu h äquivalente Notation heißt *Standardnotation*.