Ein Einblick in die Theorie der Berechnungen

Eidgenössische Technische Hochschule Zürich

Berichte des Instituts für Informatik

E. Engeler
E. Wiedmer
E. Zachos

Ein Einblick in die Theorie der Berechnungen

74

8

Ein Einblick in die Theorie der Berechnungen

von E. Engeler, E. Wiedmer und E. Zachos

Inhaltsverzeichnis

Die vorliegende Ausarbeitung nahm ihren Ursprung in drei Vorlesungen, welche E. Engeler im Rahmen der Einführungsvorlesung "Informatik an Mittelschulen" im Wintersemester 1973/74 gehalten hat.

§1 Formeln und deren Bedeutung

1.1. Problemstellung

Beispiele:

Formeln	Bedeutung
$(x^2-2\cdot\sin y)/(y-x)$	partielle Funktion: $\mathbb{R}\times\mathbb{R} \to \mathbb{R}$
$(x>0)\supset(3x-x^3y>x^2-2xy)$	boolesche Funktion: $\mathbb{R}\times\mathbb{R} \to \mathbb{B}=\{false,true\}$
var x,y: integer; **begin** y := 0; **while** sqr(y) ≠ y **do** y := y+1 **end**.	partielle Funktion: integer x integer → integer x integer

Wir haben gelernt, wie man gewissen Formeln deren Bedeutung zuordnet. Diese Zuordnung ist langwierig und, wie Sie täglich erfahren, mit Fehlschlägen verbunden. Im Prinzip ist der Mensch aber fähig, die Bedeutung richtig herzustellen. Woran liegt es, dass auch die Maschine dies tun kann? Wir sehen zwei Gründe:

1) Es handelt sich um formale Sprachen.

2) Die Formeln der Sprache sind eindeutig in Komponenten zerlegbar und die Bedeutung jeder Formel ist aus den Bedeutungen ihrer Komponenten ersichtlich.

Diese Bedingungen sind für die üblichen Formelsprachen der Mathematik erfüllt. Dies ist Ihnen bewusst, obwohl es vielleicht nie explizit gesagt worden ist. Für die Formelsprachen der Informatik aber, d.h. für Programmsprachen, sind sie von zentraler Wichtigkeit. Es lohnt sich also, etwas darauf einzugehen.

1.2. Formale (generative) Beschreibung einer Sprache

Dazu gehört erstens einmal eine Festlegung der Symbole der Sprache. Die Menge aller Symbole der Sprache nennen wir Alphabet. Eine endliche Folge von Symbolen aus einem Alphabet A nennen wir ein Wort (oder einen Ausdruck) über A. Wir bezeichnen mit A^* die Menge aller Wörter über A .

Zweitens gehört dazu eine Beschreibung der Grammatik (etwa mittels Produktionen oder Syntaxdiagramme), die uns erlaubt, diejenigen Wörter über A zu charakterisieren, die korrekte Formeln (Aussagen) sind.

Beispiel: Polynome in x,y,z (ohne Koeffizienten)

Alphabet A: $A = T \cup N$

$T = \{x,y,z,+,*,(,)\}$: Terminale Symbole

$N = \{P\}$: Nichtterminale Symbole

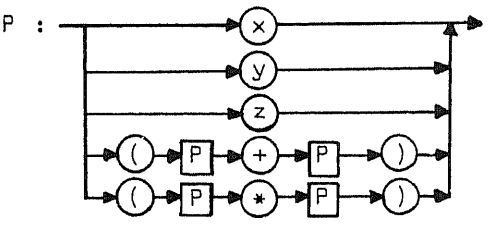
Unter den nichtterminalen Symbolen wird ein Startsymbol ausgezeichnet. Im vorliegenden Fall hat N nur ein Symbol P ; es ist das Startsymbol.

Grammatik G:

1. Beschreibung durch Produktionen

P → x
P → y
P → z
P → (P+P)
P → (P*P)

2. Beschreibung durch Syntaxdiagramme



Das obige ist ein Beispiel einer kontextfreien Grammatik. Eine Grammatik erzeugt die ihr zugeordnete Sprache in folgender Weise:

Seien $\alpha, \beta \in (T \cup N)^*$

Definition: $\alpha \Rightarrow \beta$: es gibt $\gamma, \delta, \theta \in (T \cup N)^*$ und $B \in N$ so dass: $\begin{cases} \alpha = \gamma B \delta \\ \beta = \gamma \theta \delta \\ (B \to \theta) \in G \end{cases}$

z.B. $(P+P) \Rightarrow (P+y)$ [mittels $(P \to y) \in G$]

Definition: $\alpha \overset{*}{\Rightarrow} \beta$: es gibt eine endliche Folge $\alpha_0, \ldots, \alpha_n$ von Wörtern über A, so dass $\begin{cases} \alpha_0 \equiv \alpha \\ \alpha_n \equiv \beta \\ \alpha_i \Rightarrow \alpha_{i+1} \\ \text{für } i = 0,1,\ldots,n-1 \end{cases}$

z.B. $P \overset{*}{\Rightarrow} (x+y)$

[Bemerkung: $\overset{*}{\Rightarrow}$ ist die transitive Hülle von $\Rightarrow$ unter Komposition von binären Relationen.]

Sei $B \in N$

Definition: $L(G,B) := \{\delta \in T^* \mid B \overset{*}{\Rightarrow} \delta\}$

Sei P das Startsymbol.

Definition: Sprache $L(G) := L(G,P)$

z.B. $L(G) = \{x, y, z, (x+x), (x+y), (x+z), (y+x), \ldots, (x*x), (x*y), \ldots, ((x*y)+(z*(z+x))), \ldots\ldots\}$.

1.3. Algebraische Struktur einer Sprache

Die obige Methode (der kontextfreien Grammatiken) erlaubt uns, formale Sprachen exakt zu definieren. Alle modernen Programmsprachen sind solcher Art. Die eindeutige Zerlegbarkeit von Formeln der Sprache in ihre Komponenten ist aber durch eine solche Definition noch nicht automatisch gewährleistet. (Gerade dafür brauchen wir ja z.B. die Klammern.) Betrachten Sie folgendes Beispiel:

Alphabet A: dasselbe wie oben, ohne Klammern.

Grammatik $\bar{G}$: $P \to x$

$P \to y$

$P \to z$

$P \to P+P$

$P \to P*P$

Beispiel einer Formel $\delta \in L(\bar{G})$: x*y+z

Zwei mögliche Herleitungen:

$$P \Rightarrow P*P \Rightarrow P*\widehat{P+P} \overset{*}{\Rightarrow} x*\widehat{y+z}$$

$$P \Rightarrow P+P \Rightarrow \widehat{P*P}+P \overset{*}{\Rightarrow} \widehat{x*y}+z$$

An diesem Beispiel zeigt sich, wie es bei einigen formalen Sprachen unmöglich sein kann (ohne zusätzliche Vereinbarungen), die Bedeutung von Formeln eindeutig zu eruieren. Um die Frage der eindeutigen Zerlegbarkeit der Formeln präziser zu behandeln, studieren wir nun die Algebraische Struktur einer Sprache.

Man kann die Menge der Polynome auch beschreiben, indem man erklärt, wie man aus einfachen Polynomen komplizierte Polynome zusammensetzen kann. Dies soll nun erläutert werden:

1) x,y,z sind Polynome: entspricht den Produktionen: $P \to x, P \to y, P \to z$

2) sind P_1 und P_2 Polynome, so sind auch (P_1+P_2) und (P_1*P_2) Polynome:

Bildungsregeln	entsprechend	Produktionen
$\oplus: L(G)xL(G) \to L(G)$		$P \to (P+P)$
z.B.: $\langle x,((x*y)+z)\rangle \to (x+((x*y)+z))$		
$\otimes: L(G)xL(G) \to L(G)$		$P \to (P*P)$
z.B.: $\langle x,((x*y)+z))\rangle \to (x*((x*y)+z))$		

3) Jedes Polynom erhält man aus x,y,z durch endlich viele Anwendungen der Regeln 2).

Eine Grammatik ist eindeutig lesbar (oder "strukturiert") genau dann, wenn:

1. man immer entscheiden kann, welche Bildungsregel zuletzt angewandt wurde (d.h. die Bilder der Bildungsregeln sind disjunkt).

2. die Bildungsregeln eineindeutig sind;

z.B. G^- ist nicht eindeutig lesbar, denn $x*y+z$ ist sowohl Bild von $\oplus$ als auch von $*$.

1.4. Bedeutung von Formeln

Die Tatsache, dass eine Grammatik eindeutig lesbar ist, gestattet uns, auf der durch sie definierten Sprache Funktionen rekursiv zu definieren. Insbesondere können wir die Bedeutung von Formeln als eine Funktion rekursiv einführen. In unserem Beispiel geschieht das wie folgt:

Bedeutung der Polynome: Die Bedeutung ist gegeben durch eine Abbildung $S: L(G) \to (\mathbb{R}\times\mathbb{R}\times\mathbb{R} \to \mathbb{R})$: "Semantik"

Seien $a,b,c\in\mathbb{R}, \delta\in L(G)$. Dann ist $S(\delta)[a,b,c]\in\mathbb{R}$.

Definition von S : (rekursiv nach dem Aufbau von δ)

$$S(x)[a,b,c] = a$$
$$S(y)[a,b,c] = b$$
$$S(z)[a,b,c] = c$$

$$S((\gamma+\delta))[a,b,c] = S(\gamma\oplus\delta)[a,b,c] = S(\gamma)[a,b,c]+S(\delta)[a,b,c]$$
$$S((\gamma*\delta))[a,b,c] = S(\gamma\otimes\delta)[a,b,c] = S(\gamma)[a,b,c]\cdot S(\delta)[a,b,c]$$

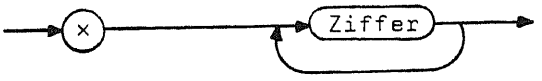
In der Theorie der Berechnungen sind Programmsprachen durch strukturierte Grammatiken gegeben. Man kann daher die Semantik von Programmsprachen entsprechend rekursiv definieren:

Idee:

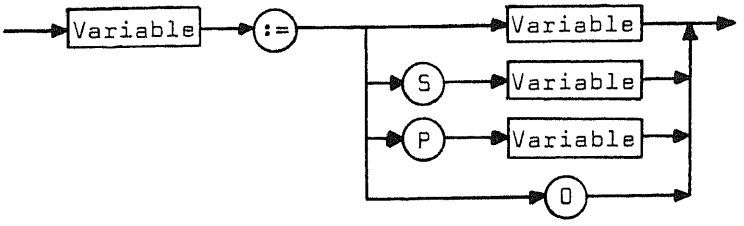
$S: \{\text{Programme in } x_1,\ldots,x_n\} \to \{\text{partielle Funktionen: } \mathbb{N}^n \to \mathbb{N}^n\}$

1.5. Eine rudimentäre Programmsprache: PASIC

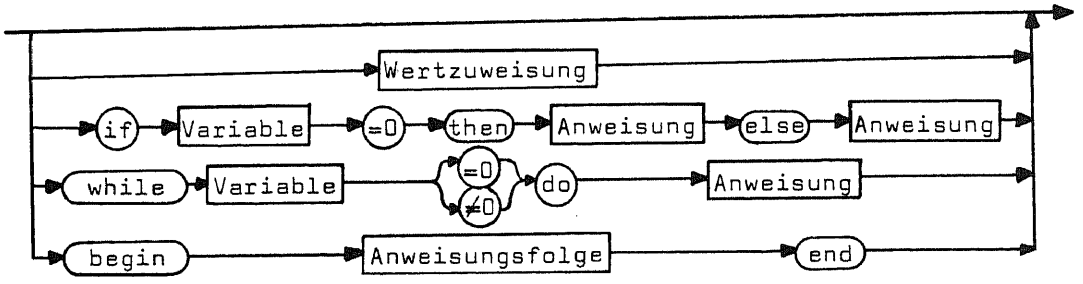
Variable



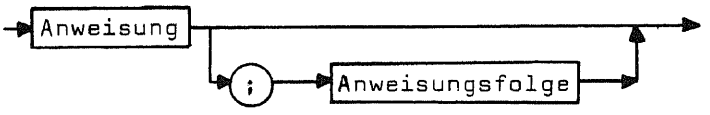
Wertzuweisung



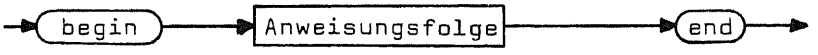
Anweisung



Anweisungsfolge



Programm



Ein PASIC Programm π ist also eine endliche Folge von Symbolen aus dem Alphabet T

$$T = \{:=,S,P,0,\text{if},\text{then},\text{else},\text{while},=0,\neq 0,\text{do},\text{begin},\text{end},;\} \cup \{x,0,1,\ldots,9\}$$

für die gilt: Programm $\overset{*}{\Rightarrow}$ π .

Bemerke: PASIC ist durch eine strukturierte Grammatik gegeben.
(Man sagt auch: "PASIC ist eine strukturierte Sprache".)

1.6. Die Semantik von PASIC

Jetzt wollen wir die Bedeutung eines Programms π erklären.

Seien alle Variablen,welche in π vorkommen, in der Menge $\{x_1,\ldots,x_n\}$ enthalten. Die Semantik von π soll als partielle Funktion $S(\pi)\colon \mathbb{N}^n \to \mathbb{N}^n$ rekursiv eingeführt werden. Dazu müssen wir auch entsprechende Bedeutungsfunktionen für die grammatischen Bestandteile (Wertzuweisungen, Anweisungen und Anweisungsfolgen) herstellen.

$$S_W(x_i := x_j)[a_1,\ldots,a_n] \quad \overset{=}{\text{def}}. [a_1,\ldots,a_{i-1},a_j,a_{i+1},\ldots,a_n]$$
$$S_W(x_i := Sx_j)[a_1,\ldots,a_n] \quad \overset{=}{\text{def}}. [a_1,\ldots,a_{i-1},a_j+1,a_{i+1},\ldots,a_n]$$
$$S_W(x_i := Px_j)[a_1,\ldots,a_n] \quad \overset{=}{\text{def}}. [a_1,\ldots,a_{i-1},a_j\dot{-}1,a_{i+1},\ldots,a_n]$$
$$S_W(x_i := 0)[a_1,\ldots,a_n] \quad \overset{=}{\text{def}}. [a_1,\ldots,a_{i-1},0,a_{i+1},\ldots,a_n]$$

wobei

$$a\dot{-}1 \quad \overset{=}{\text{def}}. \begin{cases} a-1, & \text{falls } a \geq 1 \\ 0, & \text{sonst.} \end{cases}$$

$$S_A(\ )[a_1,\ldots,a_n] \quad \overset{=}{\text{def}}. [a_1,\ldots,a_n]$$
$$S_A(\text{Wertzuweisung})[a_1,\ldots,a_n] \quad \overset{=}{\text{def}}. S_W(\text{Wertzuweisung})[a_1,\ldots,a_n]$$
$$S_A(\underline{\text{if}}\ x_i = 0\ \underline{\text{then}}\ \text{Anweisung}_1\ \underline{\text{else}}\ \text{Anweisung}_2)[a_1,\ldots,a_n]$$
$$\overset{=}{\text{def}}. \begin{cases} S_A(\text{Anweisung}_1)[a_1,\ldots,a_n] , & \text{falls } a_i = 0 \\ S_A(\text{Anweisung}_2)[a_1,\ldots,a_n] , & \text{sonst.} \end{cases}$$

$$S_A(\underline{\text{while}}\ x_i = 0\ \underline{\text{do}}\ \text{Anweisung}_1)[a_1,\ldots,a_n]$$
$$\stackrel{=}{\text{def.}} \begin{cases} (S_A(\text{Anweisung}_1) \circ S_A(\underline{\text{while}}\ x_i = 0\ \underline{\text{do}}\ \text{Anweisung}_1))[a_1,\ldots,a_n], & \text{falls } a_i = 0 \\ [a_1,\ldots,a_n]\ , \text{ sonst.} \end{cases}$$

$$S_A(\underline{\text{while}}\ x_i \neq 0\ \underline{\text{do}}\ \text{Anweisung}_1)[a_1,\ldots,a_n]$$
$$\stackrel{=}{\text{def.}} \begin{cases} (S_A(\text{Anweisung}_1) \circ S_A(\underline{\text{while}}\ x_i \neq 0\ \underline{\text{do}}\ \text{Anweisung}_1))[a_1,\ldots,a_n], & \text{falls } a_i \neq 0 \\ [a_1,\ldots,a_n]\ , \text{ sonst.} \end{cases}$$

$$S_A(\underline{\text{begin}}\ \text{Anweisungsfolge}\ \underline{\text{end}})[a_1,\ldots,a_n] \stackrel{=}{\text{def.}} S_{AF}(\text{Anweisungsfolge})[a_1,\ldots,a_n]$$

wobei $\circ$ die Komposition von Funktionen (von links nach rechts) ist.

$$S_{AF}(\text{Anweisung}) \qquad \stackrel{=}{\text{def.}} S_A(\text{Anweisung})$$

$$S_{AF}(\text{Anweisung};\text{Anweisungsfolge}) \qquad \stackrel{=}{\text{def.}} S_A(\text{Anweisung}) \circ S_{AF}(\text{Anweisungsfolge}_1)$$

$$S(\text{Programm}) = S(\underline{\text{begin}}\ \text{Anweisungsfolge}\ \underline{\text{end}}) \stackrel{=}{\text{def.}} S_{AF}(\text{Anweisungsfolge}) .$$

Man beachte, dass es sich hier um eine simultane Rekursion handelt.

Wir stellen uns nun die Frage, was wir überhaupt mit der Definition von S erreichen können. Offenbar interessiert uns der Wertebereich von S: die in PASIC berechenbaren Funktionen.

1.7. Berechenbare und rekursive Funktionen

Ein Hauptanliegen der Theorie der Berechnungen ist die Frage: "Welche Funktionen sind berechenbar?" oder "Was kommt überhaupt als Bedeutung von Programmen in Frage?"

Bemerkung: Es gibt nur abzählbar viele berechenbare Funktionen, während es überabzählbar viele zahlentheoretische Funktionen gibt.

Zur Behandlung dieser Frage brauchen wir vorerst einmal eine Präzisierung des Begriffs "berechenbar".

Definition: Eine Funktion $f: \mathbb{N}^n \to \mathbb{N}$ wollen wir (PASIC) berechenbar nennen, wenn ein (PASIC) Programm $\pi(x_1,\ldots,x_n,\ldots,x_m)$ und ein $j \leq m$ existiert, so dass für alle $a_1,\ldots,a_n \in \mathbb{N}$
$f[a_1,\ldots,a_n] = S_j(\pi)[a_1,\ldots,a_n,0,..,0]$
(wobei S_j die j-te Komponente von S ist).

Satz: Die Klasse der berechenbaren Funktionen B enthält Z, S, U_i^n und ist abgeschlossen gegen primitive Rekursion, μ-Operator und Komposition.

Beweise:

1) Z: $a_1 \mapsto 0$ "Nullfunktion"
Programm π dazu: begin $x_1 := 0$ end

2) S: $a_1 \mapsto a_1+1$ "Nachfolgerfunktion"
π dazu: begin $x_1 := Sx_1$ end

3) $U_i^n = [a_1,\ldots,a_n] \to a_i$, $i \leq n$ "Projektionsfunktionen"
π: begin end
Bemerkung: $S_i(\pi) = U_i^n$

4) Primitive Rekursion

$$\begin{cases} f(x_1,\ldots,x_n,0) = g(x_1,\ldots,x_n) \\ f(x_1,\ldots,x_n,x_{n+1}+1) = h(x_1,\ldots,x_n,x_{n+1},f(x_1,\ldots,x_{n+1})) \end{cases}$$

Bemerkung: y für x_{n+2}, z für x_{n+3}.

```
π:    begin
          y := 0;
          z := g(x1,...,xn);
          while x(n+1) ≠ 0 do
            begin
              z := h(x1,...,xn,y,z);
              x(n+1) := Px(n+1);
                 y := Sy
            end
      end
```

Bemerkung (für Puristen): Anstelle von `.....` sollten entsprechende Programme eingesetzt werden. Damit es richtig funktioniert, müsste man eigentlich dafür sorgen, dass die Input-Werte nicht zerstört werden und dass Variablenkollisionen vermieden werden.

5) μ-Operator

$f(x_1,\ldots,x_n) = (\mu z)[h(x_1,\ldots,x_n,z)=0]$

d.h. die kleinste natürliche Zahl z für die $h(x_1,\ldots,x_n,z)=0$ gilt, falls es eine solche gibt (undefiniert sonst).

Bemerkung: y für x_{n+1}, z für x_{n+2}.

```
π:    begin
          z := 0;
          y := h(x1,...,xn,0);
          while y ≠ 0 do
            begin
                z := Sz;
                y := h(x1,...,xn,z)
            end
      end
```

6) Komposition (Substitution):

$f(x_1,\ldots,x_n) = h(g_1(x_1,\ldots,x_n),g_2(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n))$

Bemerkung: y_i für x_{n+i}, y für x_{n+m+1}.

```
π:    begin
          y1 := g1(x1,...,xn);
          y2 := g2(x1,...,xn);
           :
          ym := gm(x1,...,xn);
          y := h(y1,...,ym)
      end
```

q.e.d.

Definition: Klasse der partiell rekursiven Funktionen P :
die kleinste Klasse von Funktionen: $\mathbb{N}^n \to \mathbb{N}$ welche Z, S, U_i^n enthält und abgeschlossen ist gegen primitive Rekursion, μ-Operator und Komposition.

Wir haben also bewiesen $P \subseteq B$.

Umgekehrt kann man durch Induktion nach der Struktur der Programme zeigen, dass alle berechenbaren Funktionen auch partiell rekursiv sind.

Also: $\boxed{P = B}$

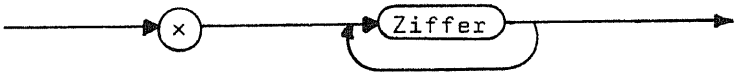
1.8. Die Sprache der Prädikatenlogik 1. Stufe

Analog zu Programmsprachen (formale Sprachen für Algorithmen) kann man auch Sprachen für mathematische Sätze formalisieren. Wir werden nun eine generative Grammatik für die Sprache der Prädikatenlogik 1. Stufe angeben.

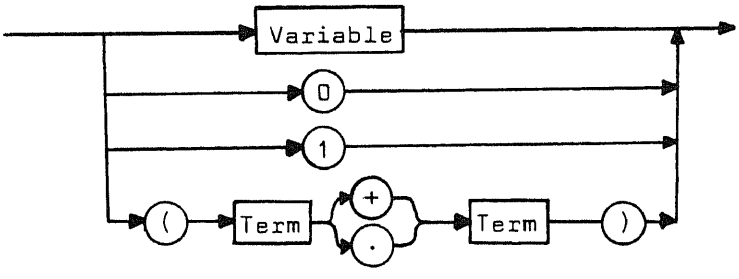
Alphabet $T = \{(,), \wedge, \vee, \neg, \exists, \forall\} \cup \{0,1,+,\cdot,=\} \cup \{x,0,1,\ldots,9\}$

Syntaxdiagramme

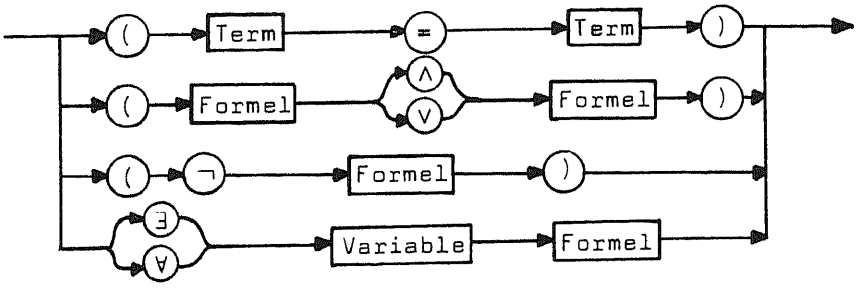
Variable



Term



Formel



Die obige Grammatik ist strukturiert. Für jede (richtig gebildete) Formel φ gilt: Formel $\overset{*}{\Rightarrow} \varphi$.

1.9. Die Semantik der Prädikatenlogik 1. Stufe

Wir wollen nun die Bedeutung einer Formel φ der Prädikatenlogik erklären. Seien alle Variablen, welche in φ vorkommen, in der Menge $\{x_1,\ldots,x_n\}$ enthalten. Die Semantik von φ soll als boolesche Funktion $S(\varphi)\colon \mathbb{N}^n \to \mathbb{B}$ eingeführt werden. (Dabei sei $\mathbb{B} = \{0,1\}$. 0 wird mit false und 1 mit true identifiziert.) Die Bedeutung von einem Term t soll eine Funktion $S_T(t) = \mathbb{N}^n \to \mathbb{N}$ sein.

$$S_T(x_i)[a_1,\ldots,a_n] \underset{\text{def.}}{=} a_i$$

$$S_T(0)[a_1,\ldots,a_n] \underset{\text{def.}}{=} 0$$

$$S_T(1)[a_1,\ldots,a_n] \underset{\text{def.}}{=} 1$$

$$S_T((\text{Term}_1+\text{Term}_2))[a_1,\ldots,a_n] \underset{\text{def.}}{=} S_T(\text{Term}_1)[a_1,\ldots,a_n] \oplus S_T(\text{Term}_2)[a_1,\ldots a_n]$$

wobei $\oplus$ die gewöhnliche Addition von natürlichen Zahlen ist.

$$S_T((\text{Term}_1\cdot\text{Term}_2))[a_1,\ldots,a_n] \underset{\text{def.}}{=} S_T(\text{Term}_1)[a_1,\ldots,a_n] \odot S_T(\text{Term}_2)[a_1,\ldots a_n]$$

wobei $\odot$ die gewöhnliche Multiplikation von natürlichen Zahlen ist.

$$S((\text{Term}_1=\text{Term}_2))[a_1,\ldots,a_n] \underset{\text{def.}}{=} \begin{cases} 1, & \text{falls } S_T(\text{Term}_1)[a_1,\ldots,a_n]=S_T(\text{Term}_2)[a_1,\ldots,a_n] \\ 0, & \text{sonst.} \end{cases}$$

$$S((\text{Formel}_1 \wedge \text{Formel}_2))[a_1,\ldots a_n] \underset{\text{def.}}{=} \begin{cases} 1, & \text{falls } S(\text{Formel}_1)[a_1,\ldots,a_n] = 1 \text{ und} \\ 0, & \text{sonst.} \quad S(\text{Formel}_2)[a_1,\ldots,a_n]=1 \end{cases}$$

$$S((\text{Formel}_1 \vee \text{Formel}_2))[a_1,\ldots a_n] \underset{\text{def.}}{=} \begin{cases} 1, & \text{falls } S(\text{Formel}_1)[a_1,\ldots a_n] = 1 \text{ oder} \\ 0, & \text{sonst.} \quad S(\text{Formel}_2)[a_1,\ldots,a_n]=1 \end{cases}$$

$$S((\neg\text{Formel}))[a_1,\ldots,a_n] \underset{\text{def.}}{=} 1 - S(\text{Formel})[a_1,\ldots,a_n]$$

$$S(\exists x_i \text{Formel})[a_1,\ldots,a_n] \underset{\text{def.}}{=} \max_{m\in\mathbb{N}} \{S(\text{Formel})[a_1,\ldots,a_{i-1},m,a_{i+1},\ldots,a_n]\}$$

$$S(\forall x_i \text{Formel})[a_1,\ldots,a_n] \underset{\text{def.}}{=} \min_{m\in\mathbb{N}} \{S(\text{Formel})[a_1,\ldots,a_{i-1},m,a_{i+1},\ldots,a_n]\}$$

Sei φ eine richtig gebildete Formel. Sei $\mathbb{N} = \langle \mathbb{N}, \oplus, \odot, =, 0, 1\rangle$ die Struktur der natürlichen Zahlen (mit Addition, Multiplikation, Gleichheit, 0 und 1).

Definition: $\mathbb{N} \models \varphi$, " φ gilt in der Struktur $\mathbb{N}$":

Für alle $a_1, a_2, \ldots, a_n$: $S(\varphi)[a_1,\ldots,a_n] = 1$.

Beispiele:

Wahre Sätze $\mathbb{N} \models \forall x_1 \forall x_2((x_1 \cdot x_2)=(x_2 \cdot x_1))$

$\mathbb{N} \models \forall x_1 \forall x_2 \forall x_3((x_1 \cdot (x_2+x_3))=((x_1 \cdot x_2)+(x_1 \cdot x_3)))$

Falsche Sätze $\mathbb{N} \not\models \forall x_1 \forall x_2 \forall x_3((x_1+(x_2 \cdot x_3))=((x_1+x_2) \cdot (x_1+x_3)))$

$\mathbb{N} \not\models \forall x_1 \exists x_2((x_2+1)=x_1)$

Es gibt Sätze, von denen man nicht so schnell (direkt) herausfinden kann, ob sie wahr oder falsch sind.

z.B. $\mathbb{N} \overset{?}{\models} \exists x_1 \exists x_2 \exists x_3$ $(((x_1 \cdot (x_1 \cdot x_1))=((x_2 \cdot (x_2 \cdot x_2))+(x_3 \cdot (x_3 \cdot x_3))))$

$\wedge$ $((\neg(x_1=0)) \wedge ((\neg(x_2=0)) \wedge (\neg(x_3=0)))))$

(Fermat'sches Problem für 3)

§2 Lösbarkeit algorithmischer Probleme

2.1. Hauptaufgabe des Programmierens:
"Verwandlung einer abstrakten Funktionsdefinition in das Input-Outputverhalten eines realen Computers".

2.1.1. Uebersetzen einer abstrakt definierten Funktion auf IN in ein Pasicprogramm

In §1.7. wurde gezeigt, dass man alle partiell-rekursiven Funktionen durch Pasicprogramme berechnen kann.

Beispiel:

$$\begin{cases} f(x_1,0) = x_1 \\ f(x_1,x_2+1) = f(x_1,x_2) + 1 \end{cases}$$

abstrakte Definition der Addition mit Rekursionsgleichungen.

zugehöriges Pasicprogramm

```
πA:  begin
        while x2 ≠ 0 do
        begin
            x2 := Px2;
            x1 := Sx1
        end
     end
```

Ebenso kann man Programme für die Multiplikation, Subtraktion, Potenzierung, Signumfunktion, GGT-Funktion, usw. konstruieren.

In 2.3.2. werden wir Funktionen auf IN definieren, die man nicht durch ein Pasicprogramm berechnen kann.

2.1.2. Funktionen auf den reellen Zahlen

Analog zu den Pasicprogrammen für die natürlichen Zahlen kann man auch Programme für die reellen Zahlen schreiben. Stehen die Operationen und Relationen $\{+,-,*,/; \geq; 0,1\}$ zur Verfügung, so kann man zu jeder rationalen Funktion f ein Programm schreiben, welches f berechnet. Aber schon für die Wurzelfunktion gibt es kein Programm mehr. Stehen die Operationen und Relationen $\{+,-,*,/,\sqrt{};\geq;0,1\}$ zur Verfügung (dies entspricht den möglichen Konstruktionen mit Zirkel und Lineal), so kann man die dritte Wurzel nicht berechnen (Deli'sches Problem der Würfelverdoppelung).

Der übliche Ausweg besteht nun darin, dass man die Funktionswerte nicht exakt auszurechnen versucht, sondern sie lediglich approximiert.

Zum Beispiel kann man $\sqrt{x}$ für $x>1$ beliebig gut durch die Funktionenfolge $a_n(x)$ mit $a_0(x) = x$

$$a_{n+1}(x) = (a_n(x) + \frac{x}{a_n(x)}) \cdot \frac{1}{2}$$

approximieren (Newton-Methode).

Zugehöriges Programm (Pascal-Notation):

```
begin
    a := x;Abbruchkriterium
    while abs(a*a-x) > eps do
        a := (a+x/a)/2
end
```

Allgemein kann man sagen: ist die Funktionenfolge $g(n,x)$ berechenbar und $F(x) = \lim_{n\to\infty} g(n,x)$, so kann man $F(x)$ approximativ berechnen.

2.2. Ideale und praktische Berechenbarkeit

In den vorangehenden Ueberlegungen haben wir von der Bedeutung von Programmen gesprochen, ohne uns zu fragen, ob es die ideale Maschine gibt, welche die Programme im gewünschten Sinne ausführen kann. Man

muss darum korrekterweise zwischen der mathematischen idealisierenden Bedeutung S^{id} und der realen Bedeutung S^{re}, welche von der vorliegenden, leider imperfekten Maschine abhängt, unterscheiden.

2.2.1. $\mathbb{N}$

Mit einem idealen Computer könnte man die Funktion

$$f(n) = 2^{(2^{\cdot^{\cdot^{\cdot^{(2^2)}}}})} \quad \left.\right\} n + 6 \text{ mal}$$

leicht ausrechnen, aber mit den üblichen realen Computern geht es nicht, denn die Funktionswerte sind viel zu gross. Auf der CDC 6000 kann man nur natürliche Zahlen n mit $n < 2^{48}$ darstellen.

Schema:

abstrakte Definition von $f: \mathbb{N}^n \to \mathbb{N}$

⇓

Pasic Programm π

⇙ ⇘

ideale Semantik

$S_1^{id}(\pi)[a_1,\ldots,a_n,0,\ldots 0]=f[a_1,\ldots,a_n]$

praktisch berechnete Funktion

$f': \tilde{\mathbb{N}}^n \to \tilde{\mathbb{N}}$ mit

$f'[a_1,\ldots,a_n]=S_1^{re}(\pi)[a_1,\ldots,a_n,0,\ldots 0]$

wobei $\tilde{\mathbb{N}} = \{n \in \mathbb{N} \mid n$ ist auf der realen Maschine darstellbar$\}$

2.2.2. $\mathbb{R}$ und der Datentyp "real" (= endliche Menge der auf der vorhandenen Maschine darstellbaren reellen Zahlen)

Beim Rechnen mit reellen Zahlen ist der Unterschied zwischen der idealen Semantik und der realen Semantik viel schwerwiegender. Dies soll nun an der Berechnung der Zahl π demonstriert werden.

Man weiss, dass man π mit den Operationen und Relationen $\{+,-,*,/,\sqrt{\ };\geq;1\}$ nicht berechnen kann (Quadratur des Kreises), aber man kann π approximieren. In der Schule geschieht dies oft durch die Berechnung des halben Umfanges des eingeschriebenen 2^n Eckes.

Zugehöriges Programm (Pascal-Notation):

a_n = Seite des eingeschriebenen 2^n Eckes

g_n = halber Umfang

$a_{n+1} = \sqrt{2 - 2.\sqrt{1 - (a_n/2)^2}}$

```
var a,sqa,g: real; p,n: integer;
begin p := 4; a := sqrt(2); g := 2*a; sqa := sqr(a);
      for n := 1 to 20 do
      begin sqa := 2 - 2*sqrt(1-sqa/4);
            a := sqrt(sqa); g := a*p;
            p := p*2
      end;
end.
```

Resultat auf CDC 6000:

n	P_n	g_n
13	8192	3.141592605919
14	16384	3.141592699173
15	32768	3.141592947853
16	65536	3.141593942571
17	131072	3.141591953135
exakter Wert von π		3.141592653589793....

Unerwarteterweise ist g_{14} grösser als π . Für $n \geq 14$ scheinen die Werte fast wie zufällig verteilt in der Umgebung von π zu liegen. Die Fehlerursache liegt vor allem in der Subtraktion 2 - 2 * sqrt(1 - sqa/4) , die ein sehr ungenaues Resultat liefert, da man die Differenz von zwei fast gleich grossen Zahlen bildet.

Ein anderes Beispiel für unerwünschte numerische Effekte entsteht bei der Berechnung der harmonischen Reihe

$$H = 1 + 1/1 + 1/2 + 1/3 + 1/4 + \ldots$$

Berechnet man die harmonische Reihe auf dem idealen Computer,so

divergiert sie, aber auf dem realen Computer konvergiert sie.

Resultat	Genauigkeit
3.7	nach der 1. Stelle werden die Ziffern abgeschnitten
5.82	nach der 2. Stelle werden die Ziffern abgeschnitten
8.069	nach der 3. Stelle werden die Ziffern abgeschnitten
10.3668	nach der 4. Stelle werden die Ziffern abgeschnitten

2.3. Nichtberechenbare zahlentheoretische Funktionen

Es gibt überabzählbar viele Funktionen über den natürlichen Zahlen, aber nur abzählbar viele berechenbare Funktionen (denn es gibt ja auch nur abzählbar viele Pasic-Programme). Somit müssen also auch nicht berechenbare Funktionen existieren.

Fragen:
- Kann man ein Beispiel einer nicht berechenbaren Funktion angeben?
- Gibt es interessante nicht berechenbare Funktionen?

Antworten:
- Ein erstes Beispiel einer nicht berechenbaren Funktion wird im Abschnitt 2.3.2. gegeben.
- Es wäre zum Beispiel interessant, wenn man ein Programm hätte, welches herausfinden würde, welche Sätze über der Struktur der natürlichen Zahlen $\langle \mathbb{N},+,\cdot,=,0,1\rangle$ wahr sind und welche falsch. Ein solches Programm würde (falls es durch eine Maschine in vernünftiger Zeit ausgeführt werden könnte) dem Mathematiker ziemlich viel Arbeit abnehmen. In 3.5 wird gezeigt, dass es kein solches Programm geben kann.

2.3.1. Das universelle Programm

Gemäss dem Vorschlag von Babbage-Neumann werden in den meisten Maschinen die Programme intern gespeichert. Das heisst, man konstruiert nicht für jeden Algorithmus einen speziellen Computer, sondern man sucht eine Maschine, die mehrere (wenn möglich alle) Programme ausführen kann (universelle Maschine). Ein Programm für eine solche universelle Maschine soll im folgenden entwickelt werden.

a Name des Speichers, in welchem die jeweils zur Ausführung kommende PASIC-Anweisung gespeichert wird (immer von der Form << begin Anweisungsfolge end >>).

y Name des Speichers, wo die jeweiligen Werte der Variablen $x_1,\ldots,x_n$ abgespeichert werden.

1. Schritt {Bemerkung: R steht für Rest; AF für Anweisungsfolge; A, A1, A2 für Anweisungen}

```
begin
   "lese in a das Programm ein";
   "lese in y die Daten ein";
   while "a verschieden von << begin end >>"do
   begin
      if "a von der Form << begin Wertzuweisung R end >>" then
      begin "führe Wertzuweisung aus";
            "a := << begin R end >>";
      end else
      if "a von der Form << begin; AF end >>" then
            "a := << begin AF end >>
      else
      if "a von der Form << begin if X_i = 0 then A1 else A2 R end >>"
          then
      begin
         if "i-te Komponente von y gleich 0" then
            "a := << begin A1 R end >>"
         else "a := << begin A2 R end >>"
      end else
      if "a von der Form << begin while X_i = 0 do A R end >>"
          then
      begin
         if "i-te Komponente von y gleich 0" then
            "a := << begin A; while X_i = 0 do A R end >>"
         else "a := << begin R end >>"
      end else
      if "a von der Form << begin while X_i ≠ 0 do A R end >>" then
      begin
           ......
      end else
      if "a von der Form <<begin begin AF end R end >>" then
         "a := <<begin AF R end >>"
```

2. Schritt Gödelnumerierung

Da a,y eigentlich Speicher für natürliche Zahlen sind, muss man die Pasicprogramme und die Daten $\langle x_1,\ldots,x_n\rangle$ verschlüsseln. Dabei kann man zum Beispiel die Primzahlkodierung anwenden.

Daten:

$$\langle x_1,\ldots,x_n\rangle \mapsto y = 2^{x1}\cdot 3^{x2}\cdot 5^{x3}\ldots p_n^{x_n} = p_1^{x1}\cdot p_2^{x2}\ldots\cdot p_n^{x_n}$$

(p_i = i-te Primzahl)

Bezeichnung: $y = \overline{\langle x_1,\ldots,x_n\rangle}$

Anweisungen: (rekursiv)

leer $\mapsto$ 29

$x_i := x_j \mapsto 2^i\cdot 3^j$

$x_i := Sx_j \mapsto 2^i\cdot 5^j$

$x_i := Px_j \mapsto 2^i\cdot 7^j$

$x_i := 0 \mapsto 2^i\cdot 11$

if $x_i = 0$ then A1 else A2 $\mapsto 2^i\cdot 13^{\overline{A1}}\cdot 17^{\overline{A2}}$

($\overline{A1}$ = Gödelnummer der Anweisung A1 ,
$\overline{A2}$ = Gödelnummer der Anweisung A2)

while $x_i = 0$ do A $\mapsto 2^i\cdot 19^{\overline{A}}$

while $x_i \neq 0$ do A $\mapsto 2^i\cdot 23^{\overline{A}}$

begin A1; ,,,; AN end $\mapsto 31\cdot 2^{\overline{A1}}\cdot 3^{\overline{A2}}\ldots\cdot p_N^{\overline{AN}}$

Beispiel:

$$q_1 = 31\cdot 2^{(2^2\cdot 23^{(31\cdot 2^{(2^2\cdot 7^2)}\cdot 3^{(2^1\cdot 5^1)})})}$$ = Gödelnummer von

π_A: begin while x2 ≠ 0 do begin x2 := px2; x1 := sx1 end end

Während der Ausführung von π_A mit der universellen Maschine kann also das Register a folgende Werte annehmen.

q_1 = Gödelnummer von $\pi_A = 31 \cdot 2^{(2^2 \cdot 23^{(31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)})})}$

q_2 = Gödelnummer von: begin begin x2 := px2; x1 := sx1 end; while x2 ≠ 0 do begin x2 := px2; x1 := sx1 end end

$= 31 \cdot 2^{31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)} \cdot 3^{(2^2 \cdot 23^{(31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)})})}}$

q_3 = Gödelnummer von: begin x2 := px2; x1 := sx1; while x2 ≠ 0 do begin x2 := px2; x1 := sx1 end end

$= 31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)} \cdot 5^{(2^2 \cdot 23^{(31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)})})}$

q_4 = Gödelnummer von: begin; x1 := sx1; while x2 ≠ 0 do begin x2 := px2; x1 := sx1 end end

$= 31 \cdot 2^{29} \cdot 3^{(2^1 \cdot 5^1)} \cdot 5^{(2^2 \cdot 23^{(31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)})})}$

q_5 = Gödelnummer von: begin x1 := sx1; while x2 ≠ 0 do begin x2 := px2; x1 := sx1 end end

$= 31 \cdot 2^{(2^1 \cdot 5^1)} \cdot 3^{(2^2 \cdot 23^{(31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)})})}$

q_6 = Gödelnummer von: begin; while x2 ≠ 0 do begin x2 := px2; x1 := sx1 end end

$= 31 \cdot 2^{29} \cdot 3^{31 \cdot 2^{(2^2 \cdot 23^{(31 \cdot 2^{(2^2 \cdot 7^2)} \cdot 3^{(2^1 \cdot 5^1)})})}}$

q_7 = Gödelnummer von: begin end = $31 \cdot 2^{29}$

3. Schritt Umschreiben

Man schreibt die Instruktionen, welche in "..." geschrieben wurden, in PASIC-Anweisungen um. Dies ist äusserst langwierig, aber nicht besonders schwierig.

Bemerkung: Für die gebräuchlichen Maschinen wäre die obige Verschlüsselung völlig undurchführbar. Man verwendet dort eine der

Speicherstruktur und eine den zur Verfügung stehenden Operationen angepasste Kodierung.

Folgerung: Es gibt eine berechenbare Funktion $\Omega: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

$$\Omega\ (\overline{\pi},\ \overline{\langle a_1,\ldots,a_n\rangle}\ ) = \begin{cases} b & \text{falls } S_1(\pi)[a_1,\ldots,a_n] = b \\ \text{undefiniert} & \text{falls } S_1(\pi)[a_1,\ldots,a_n] \text{ undef.} \end{cases}$$

Ω nennt man universelle Funktion.

2.3.2. Eine nicht berechenbare Funktion

Man kann die Gödelisierungsmethode von 2.3.1. auch benutzen, um eine nicht berechenbare Funktion anzugeben.

Lemma: Vor.: $\delta: \mathbb{N} \to \mathbb{N}$ partielle Funktion mit

für alle Programme π gilt

$$\delta(\overline{\pi}) = \begin{cases} \text{undef.} & \text{falls } S(\pi)[\overline{\pi},0,\ldots,0] \text{ definiert} \\ 0 & \text{sonst.} \end{cases}$$

Beh.: Für alle Programme π gilt
$S_1(\pi)[\overline{\pi},0,\ldots,0]$ ist verschieden von $\delta(\overline{\pi})$
d.h. δ ist nicht berechenbar.

Beweis: Die Behauptung folgt direkt aus der Voraussetzung für δ .

q.e.d.

Die Funktionen, welche der Voraussetzung des Lemmas genügen, scheinen etwas künstlich zu sein. Von grösserem Interesse ist das folgende Resultat.

Satz: Vor.: $\Delta: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

$$\Delta(\overline{\pi},\ \overline{\langle a_1,\ldots,a_n\rangle}\ ) = \begin{cases} 1 & \text{falls } S(\pi)[a_1,\ldots,a_n] \text{ definiert} \\ 0 & \text{falls } S(\pi)[a_1,\ldots,a_n] \text{ undef.} \end{cases}$$

Beh.: Δ ist nicht berechenbar,

d.h. man kann im allgemeinen nicht entscheiden, ob ein Programm für gegebene Eingabedaten terminiert.

Beweis: Annahme: Δ ist berechenbar.

Man betrachte das folgende Programm π_Δ :

```
begin
     "x2 := <x1,0,...,0> ";
      x3 := Δ(x1,x2);
      if x3 = 1 then "ewige Schleife"
end
```

Das obige Programm π_Δ berechnet eine Funktion δ (Wert in x1), welche der Voraussetzung des Lemmas genügt.
Widerspruch! q.e.d.

§3 Berechnungen und ihre Zeitkomplexität

3.1. Die Church'sche These

Jetzt wollen wir uns der folgenden Frage widmen: Darf man überhaupt die Klasse der intuitiv (irgendwie) berechenbaren Funktionen mit der Klasse der PASIC berechenbaren Funktionen identifizieren?

Wir haben schon (in § 1.7) bemerkt, dass die Klasse der PASIC berechenbaren Funktionen mit derjenigen der partiell rekursiven Funktionen zusammenfällt. Mit beiden Methoden versucht man, die intuitiv berechenbaren Funktionen zu charakterisieren. Es gibt andere Ansätze (z.B. die Turing Maschinen, die Kleene Rekursionsgleichungen (Herbrand-Gödel), die Post Systeme, die λ-Konversion von Church, die Markov Algorithmen, die Shepherdson-Sturgis Registermaschine etc.), die aber alle genau die gleiche Klasse von Funktionen erfassen. Diese Klasse wollen wir mit P bezeichnen.

Die These von Church: P fällt mit der Klasse der intuitiv berechenbaren Funktionen zusammen.

Diesen "Satz" kann man natürlich nicht formal beweisen, denn in einem Beweis müsste man den Begriff "intuitiv berechenbar" formalisieren, und dadurch hätte man einfach das Problem verschoben. Wir werden aber doch versuchen, die These plausibel zu machen.

Die erste Richtung (falls $f \in P$, so f intuitiv berechenbar) ist ziemlich evident, denn alle Charakterisierungen von P wurden so gewählt, dass man die Funktionen in einem intuitiven Sinne "im Prinzip" berechnen kann.

Die zweite Richtung (falls f intuitiv berechenbar, so $f \in P$) ist viel schwieriger zu begründen. Wir bringen folgende Plausibilitätsargumente vor:

1. Alle bis heute erfundenen Charakterisierungen stimmen überein.
2. Zu jeder konkreten Angabe einer berechenbaren Funktion hat man immer eine entsprechende Definition in irgendeiner dieser Charakterisierungen angeben können.

3.2. PASIC Berechnungen

Was versteht man überhaupt unter dem Begriff "Berechnung"?

Eine erste Antwort dazu kann folgendermassen lauten: Eine Berechnung setzt sich zusammen aus endlich vielen Berechnungsschritten auf gewissen (formalen) Objekten: Daten, gemäss einer festgelegten (deterministischen, also nicht zufälligen) Vorschrift: Algorithmus (Programm).

Wir wollen im folgenden diesem Begriff etwas Raum schenken, indem wir über seine wichtigen Bestandteile - Daten, Berechnungsschritte, Algorithmen - diskutieren. Stellen wir uns also einen "Prozessor" vor, der eben eine Funktion "berechnet".

Elemente des Argumentenbereiches der berechneten Funktion nennen wir Eingabedaten, solche des Wertebereiches Ausgabedaten. Im Unterschied zu Elementen irgendeines mengentheoretischen Bereiches sind Daten aus einer höchstens abzählbaren Menge von wohlunterscheidbaren Objekten auszuwählen. Wir denken uns einfachheitshalber solche Daten - z.B. natürliche Zahlen in ihrer Strichdarstellung - auf endlich vielen Datenträgern (Variablen) - z.B. Papierblättern - aufgeschrieben. Auf jedem dieser Blätter, welche mit Namen $x_1,\ldots,x_n$ versehen sind, stehen also irgendwelche Elemente aus dem (abzählbaren) Datenbereich. Eine Berechnung wird nun die Inschriften auf diesen Blättern schrittweise abändern, bis schliesslich auf einem Blatt - etwa x_1 - der errechnete Wert der Funktion steht.

Jeder Berechnungsschritt entspricht einer elementaren, d.h. möglichst einfachen Operation auf den Daten. Der "Prozessor", welcher diese Operationen ausführt, hat ausserdem die Fähigkeit, die Daten zu lesen

und sich daraufhin zu entscheiden, was er zu tun hat. Rechnen wir z.B. auf der Strichdarstellung der natürlichen Zahlen, so lassen wir die folgenden elementaren Operationen zu: Anhängen resp. Wegstreichen eines Striches am Anfang der Strichfolge auf dem Blatt x_i, kopieren der Inschrift von Blatt x_i auf das Blatt x_j, löschen von x_i. [Entsprechende PASIC Zuweisungen: $x_i := Sx_i$, $x_i := Px_i$, $x_i := x_j$, $x_i := 0$].

Eine Berechnung setzt sich also zusammen aus solchen elementaren Operationen; allerdings nicht in einer beliebigen oder zufälligen Reihenfolge, sondern gemäss einer Vorschrift: Programm. Die einfachsten Vorschriften sind Befehle zur Ausführung einer elementaren Operation. Man kann aber Vorschriften auch zusammensetzen. Wir wollen die Art und Weise der Zusammensetzung von Vorschriften möglichst einfach und verständlich festlegen. Dann ergeben sich zwei Möglichkeiten:

1) Es ist eine Reihe von einfacheren Vorschriften unbedingt (der Reihe nach) auszuführen. Dies entspricht der Anweisungsfolge von PASIC.

2) Die Ausführung wird von Eigenschaften der Daten abhängig gemacht. Wir lassen im wesentlichen eine elementare Abfrage zu: hat es einen Strich auf dem Blatt x_i [PASIC: $x_i = 0$?, $x_i \neq 0$?]. Im Fall 2) der bedingten Ausführung wollen wir weiter unterscheiden: a)"Ausführung, falls"b)"repetierte Ausführung, solange"die Daten eine gewisse Eigenschaft besitzen. Diese beiden Varianten werden in PASIC in der bekannten Weise dargestellt:

   a) if ... then ... else ... b) while ... do ... .

Die obige formale Präzisierung einer solchen "Vorschrift" für das Berechnen mit natürlichen Zahlen in Strichdarstellung führt unmittelbar zur Programmsprache PASIC. Damit erfassen wir eine genügend grosse Klasse von Berechnungen auf eine recht einfache Art, denn:

1. Die Ausführung der elementaren Operationen ist wirklich trivial, und
2. Die Berechnungsvorschrift, d.h. ein PASIC-Programm, lässt sich linear aufschreiben und leicht von links nach rechts lesen und verstehen.

Zu einem PASIC Programm π über $x_1,\ldots,x_n$ könnte man (analog zur Semantikfunktion $S(\pi)$, siehe § 1.6) eine (partielle) Berechnungsfunktion $C(\pi)$ rekursiv definieren, welche jedem n-Tupel $[a_1,\ldots,a_n]\in\mathbb{N}^n$ von Eingabedaten eine endliche Folge von natürlichen Zahlen ($\in\mathbb{N}^{\underline{\omega}}$) die sogenannte Berechnungsfolge zuordnet.

$$C(\pi) : \mathbb{N}^n \rightarrow \mathbb{N}^{\underline{\omega}}$$

$$[a_1,\ldots,a_n] \rightarrow [y_{11},y_{12},\ldots,y_{1n},p_1,y_{21},\ldots,p_m]$$

wobei y_{ij} der Wert der Variable x_j vor der Ausführung des i-ten Schrittes und p_i die Gödelnummer des noch auszuführenden Programmteils ist. (Insbesondere ist p_1 die Gödelnummer des ganzen Programms und p_m die Gödelnummer des leeren Programms: begin end.)

Bemerkung:

$$S(\pi)[a_1,\ldots,a_n] = [b_1,\ldots,b_n] \Leftrightarrow$$

$$\text{ex. } m\in\mathbb{N} \text{ , ex. } y_{21},\ldots,y_{m-1,n},\ p_{m-1}$$

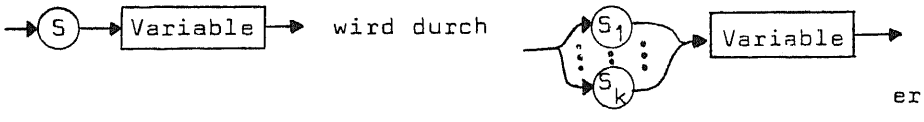
$$C(\pi)[a_1,\ldots,a_n] = [a_1,\ldots,a_n,\overline{\pi},y_{21},\ldots,p_{m-1},b_1,\ldots,b_n,\underline{\text{begin end}}]$$

3.3. Datendarstellung und Zeitkomplexität

Im Abschnitt 3.2 haben wir uns auf die Strichdarstellung der natürlichen Zahlen beschränkt. Es ist jedoch möglich, Berechnungen mit anderen Objekten durchzuführen. Wie aber schon erwähnt, kann die Anzahl der betrachteten, voneinander verschiedenen Daten sicher nicht überabzählbar sein. Folglich kann man jeweils die Daten mit natürlichen Zahlen kodieren und darauf (z.B. gemäss PASIC Programm) operieren.

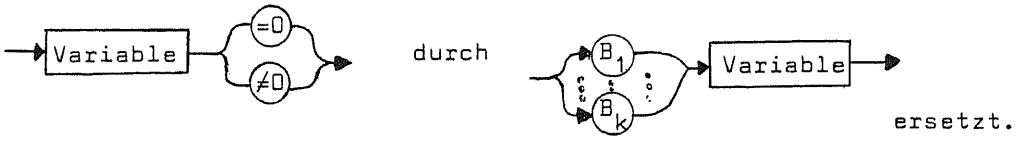
Man könnte aber ebensogut, statt $\mathbb{N}$ eine andere Struktur wählen, um die Daten zu kodieren. Diese Struktur würde man durch eine Sprache über einem endlichen Alphabet darstellen. Die Daten werden also durch Wörter über einem Alphabet $A = \{a_1,\ldots,a_k\}$ repräsentiert. Betrachten wir die folgenden elementaren Operationen: Ein Symbol a_i $\{i=1,\ldots,k\}$ am Anfang eines Wortes anhängen (Konkatenation), das erste Symbol

eines Wortes wegstreichen, kopieren, das ganze Wort löschen. Analog elementare Abfragen: Testen, ob das erste Symbol eines Wortes das Symbol a_i ist (i=1,...,k). Entsprechend diesen elementaren Operationen und Abfragen können wir ein verallgemeinertes PASIC über dem Alphabet A einführen. Dazu müssen wir sehr wenig abändern:



ersetzt;

Ebenso werden die Abfragen



ersetzt.

Diese verallgemeinerte Programmsprache genügt dann, um Berechnungen auf Wörtern über A zu beschreiben.

Da wir einerseits die natürlichen Zahlen durch Wörter über einem beliebigen endlichen Alphabet darstellen können (z.B. über dem üblichen Dezimalalphabet {0,1,...,9} oder dem binären {0,1}), anderseits Wörter über einem endlichen Alphabet mit natürlichen Zahlen kodieren können, sind diese zwei Betrachtungsweisen (vorläufig noch) äquivalent.

Von grosser Bedeutung ist die Anzahl der elementaren Operationen, die während einer Berechnung ausgeführt werden. Sie gibt nämlich ein Mass für die Zeitkomplexität. Mit diesem Begriff eröffnet sich ein weiterer Fragenkreis der Berechnungstheorie. Die zentrale Frage lautet nämlich nicht mehr, ob eine Funktion berechenbar ist, sondern wie leicht kann man sie berechnen. Zu diesem Zweck kann man natürlich auch andere oder allgemeinere Komplexitätsmasse einführen.

Beispiele von Problemen über Zeitkomplexität:

1) Welche Funktionen kann man in vernünftiger Zeit berechnen?
   (In Abschnitt 3.6 werden wir z.B. beweisen, dass man in der Prädikatenlogik 1. Stufe für $\langle \mathbb{N}; +, =, 0, 1 \rangle$ für viele Formeln nicht

in vernünftiger Zeit entscheiden kann, ob sie wahr oder falsch sind.)

2) Wie hängt die Zeitkomplexität von der Darstellung der Daten ab? (Man beweist z.B. leicht, dass man in der Dualdarstellung 2^n mit einem Aufwand $< c \cdot n \cdot \log_2(n)$ berechnen kann, währenddem man in der Strichdarstellung dafür mindestens $2^n - n$ Schritte benötigt.)

3) Wie hängt die Zeitkomplexität von der Wahl der elementaren Operationen ab?
(Z.B. was gewinnt man, wenn man elementar multiplizieren kann?)

4) Wie hängt die Zeitkomplexität von der Art der Zusammensetzung der Vorschriften ab?
(Z.B. was gewinnt man, wenn man Programme mit Prozeduren oder "goto" oder gar selbstmodifizierende Programme zulässt?)

3.4. Formalisierung des Berechnungsbegriffes in der Prädikatenlogik 1. Stufe (Vorbereitung zu 3.5 und 3.6)

Wir betrachten als Beispiel das Programm π_A

```
(q1)   begin
           while x2 ≠ 0 do
(q3)       begin
               x2 := Px2;
(q5)           x1 := Sx1
           end
(q7)   end
```

q_1, q_3, q_5, q_7 sind die in 2.3.1 im 2. Schritt berechneten Gödelnummern von eventuell zur Ausführung gelangenden Anweisungen.

Die Folge $y_{11}, y_{12}, p_1, y_{21}, y_{22}, p_2, \ldots, y_{m1}, y_{m2}, p_m$ ist eine Berechnung des Programmes $\pi_A(x_1, x_2)$ genau wenn:

p_1 = Gödelnummer von π_A {d.h. $p_1 = q_1$} und für alle i mit $1 \leq i \leq m-1$ gilt:

$$[p_i = q_1 \text{ und } y_{i2} \neq 0 \text{ und } p_{i+1} = q_3 \text{ und } y_{i+1,1} = y_{i,1} \text{ und } y_{i+1,2} = y_{i,2}]$$

oder $[p_i = q_1 \text{ und } y_{i2} = 0 \text{ und } p_{i+1} = q_7 \text{ und } y_{i+1,1} = y_{i,1} \text{ und } y_{i+1,2} = y_{i,2}]$

oder $[p_i = q_3 \text{ und } p_{i+1} = q_5 \text{ und } y_{i+1,1} = y_{i,1} \text{ und } y_{i+1,2} = y_{i,2}{}^{-1}]$

oder $[p_i = q_5 \text{ und } p_{i+1} = q_1 \text{ und } y_{i+1,1} = y_{i,1}{}^{+1} \text{ und } y_{i+1,2} = y_{i,2}]$

und p_m = Gödelnummer des leeren Programmes {d.h. $p_m = q_7$}

Bemerkung: m-1 ist also die Anzahl der Berechnungsschritte.

Frage: Wie drückt man in der Prädikatenlogik aus:

"$\langle y_{11},\ldots,y_{mn},p_m\rangle$ ist eine Berechnung von $\pi(x_1,\ldots,x_n)$" ?

Zuerst muss man Folgen von natürlichen Zahlen verschlüsseln können. Dies soll hier mit Hilfe der Funktion $f: \mathbb{N}^3 \to \mathbb{N}$ geschehen.

$$f(b,c,h) = b \text{ mod } (1+(h+1)\cdot c) \qquad \text{(Gödel 1931).}$$

Man kann nämlich mit Hilfe des chinesischen Restsatzes beweisen:

"Für alle Folgen $K_1,\ldots,K_l$ existieren $b,c\in\mathbb{N}$ mit $f(b,c,h)=K_h$ für $0 \leq h \leq l$."

Jede Berechnungsfolge $\langle y_{11},\ldots,y_{mn},p_m\rangle$ von $\pi(x_1,\ldots,x_n)$ wird nun durch drei natürliche Zahlen b,c,m verschlüsselt, sodass

$$\left.\begin{aligned} y_{ij} &= f(b,c,(i-1)\cdot(n+1)+j) \\ p_i &= f(b,c,i\cdot(n+1)) \end{aligned}\right\} \text{ für alle } 1 \leq i \leq m,\ 1 \leq j \leq n$$

1. Schritt $L(\cdot,+,-,f,=,<,\leq,0,1)$

Zum Programm π soll nun eine Formel $\tilde{B}_\pi$ angegeben werden mit

$$\tilde{B}_\pi \in L(\cdot,+,-,f,=,<,\leq,0,1)$$

und $\mathbb{N} \models \tilde{B}_\pi(b,c,m) \Leftrightarrow$ b,c,m Verschlüsselung einer Berechnung von π .

Das Beispiel π_A ist genügend allgemein, um zu sehen, wie man zu beliebigem Programm π eine entsprechende Formel $\tilde{B}_\pi$ finden kann.

$\tilde{B}_{\pi_A}$: (in diesem Fall ist $n = 2$)

$$
\begin{aligned}
& f(b,c,3) = q_1 \\
\wedge\ & \forall i\ (1 \leq i \leq m-1 \rightarrow \\
& \quad [f(b,c,i\cdot 3) = q_1 \wedge f(b,c,(i-1)\cdot 3+2) \neq 0 \\
& \quad \wedge f(b,c,(i+1)\cdot 3) = q_3 \wedge f(b,c,i\cdot 3+1) = f(b,c,(i-1)\cdot 3+1) \\
& \quad \wedge f(b,c,i\cdot 3+2) = f(b,c,(i-1)\cdot 3+2)] \\
& \vee [f(b,c,i\cdot 3) = q_1 \wedge f(b,c,(i-1)\cdot 3+2) = 0 \wedge f(b,c,(i+1)\cdot 3) = q_7 \\
& \quad \wedge f(b,c,i\cdot 3+1) = f(b,c,(i-1)\cdot 3+1) \wedge f(b,c,i\cdot 3+2) = f(b,c,(i-1)\cdot 3+2)] \\
& \vee [f(b,c,i\cdot 3) = q_3 \wedge f(b,c,(i+1)\cdot 3) = q_5 \wedge \\
& \quad f(b,c,i\cdot 3+1) = f(b,c,(i-1)\cdot 3+1) \wedge f(b,c,i\cdot 3+2) = f(b,c,(i-1)\cdot 3+2)-1] \\
& \vee [f(b,c,i\cdot 3) = q_5 \wedge f(b,c,(i+1)\cdot 3) = q_1 \wedge \\
& \quad f(b,c,i\cdot 3+1) = f(b,c,(i-1)\cdot 3+1)+1 \wedge f(b,c,i\cdot 3+2) = f(b,c,(i-1)\cdot 3+2)])
\end{aligned}
$$

$$\wedge\ f(b,c,m\cdot 3) = q_7$$

2. Schritt $L(\cdot,+,=,0,1)$

In $L(\cdot,+,=,0,1)$ kann man $f,-,<,\leq$ definieren, z.B.

$$f(b,c,h) = r \Leftrightarrow \exists y(b=(1+(h+1)\cdot c)\cdot y+r \wedge \exists x(r+x=1+(h+1)\cdot c))$$

Also kann man eine Formel $B_\pi \in L(\cdot,+,=,0,1)$ finden mit

$\mathbb{N} \models B_\pi$ genau wenn $\mathbb{N} \models \tilde{B}_\pi$ (für alle π) .

3.5. Die Unentscheidbarkeit der elementaren Zahlentheorie

Wir betrachten die Struktur der natürlichen Zahlen, versehen mit der Addition, Multiplikation, Gleichheit, Konstanten 0 und 1 .
$\mathbb{N} = \langle \mathbb{N},+,\cdot,=,0,1\rangle$. Wir fragen uns nun:
Gibt es einen Algorithmus, der herausfindet, welche Formeln aus $L(+,\cdot,=,0,1)$ (Prädikatenlogik 1. Stufe, siehe 1.8 - 1.9) in der Struktur $\mathbb{N}$ gelten?

Da man mit PASIC-Programmen nur auf den natürlichen Zahlen operieren kann, muss man die Formeln aus $L(+,\cdot,=,0,1)$ in natürliche Zahlen verschlüsseln. Dies geschieht ganz analog zur Verschlüsselung von Programmen (2.3.1, 2. Schritt). Ist φ eine Formel aus $L(+,\cdot,=,0,1)$, so bezeichnen wir auch hier mit $\overline{\varphi}$ die Verschlüsselung von φ.

Satz: Vor.: $f: \mathbb{N} \to \mathbb{N}$

für alle Formeln φ auf $L(+,\cdot,=,0,1)$ gilt

$$f(\overline{\varphi}) = \begin{cases} 1, & \text{falls } \underline{\mathbb{N}} \models \varphi \text{ (d.h. } \varphi \text{ gilt in der Struktur } \underline{\mathbb{N}}) \\ 0, & \text{falls } \underline{\mathbb{N}} \not\models \varphi \text{ (d.h. } \varphi \text{ ist falsch in der Str. } \underline{\mathbb{N}}) \end{cases}$$

Beh.: f ist nicht berechenbar. D.h. die elementare Zahlentheorie ist nicht entscheidbar.

Beweis: Zu jedem Programm ρ kann man eine Formel $D_\rho \in L(+,\cdot,=,0,1)$ ausrechnen mit

$\underline{\mathbb{N}} \models D_\rho$ genau wenn $S(\rho)[\overline{\rho},0,\ldots,0]$ definiert.

D_ρ kann man aus B_ρ (siehe 3.4, 2. Schritt) leicht bestimmen!

Jetzt kann man das Lemma von § 2.3.2 anwenden.

Annahme: f sei berechenbar.

Es existiert also ein Programm π mit $f(x) = S_1(\pi)[x,0,..,0]$ für alle x.

Aus π kann man das folgende Programm p konstruieren:

```
begin
  if "x1 Gödelnummer eines Programmes ρ" then
    "x1 := Gödelnummer der Formel Dρ";
  π;
  if x1 = 1 then "ewige Schleife"
end
```

p berechnet eine Funktion δ, die den Voraussetzungen des Lemmas von § 2.3.2 genügt. Widerspruch! q.e.d.

3.6. $\langle N,+,=,0,1\rangle$ Rabin-Fischer 1973

Mit der Sprache $L(+,=,0,1)$, das heisst ohne das Multiplikationszeichen, kann man nicht mehr soviel ausdrücken. Insbesonders kann man die Gödelisierungsfunktionen für beliebige endliche Folgen nicht mehr definieren.

Tatsächlich hat man schon früh Verfahren gefunden, die für jede Formel aus $L(+,=,0,1)$ angeben, ob sie wahr oder falsch ist. (Pressburger 1929, mit Hilfe der Quantorenelimination.) Versucht man nun aber ein solches Verfahren auf dem Computer zu implementieren, so muss man feststellen, dass für lange Formeln der Rechnungsaufwand unmöglich gross wird.

Rabin-Fischer (1973) haben nun bewiesen, dass jedes noch so gute Entscheidungsverfahren für unendlich viele Formeln φ mehr als $2^{2^{c\cdot|\varphi|}}$ Rechenschritte benötigt. (Mit $|\varphi|$ bezeichnet man die Länge der Formel. c ist eine nicht näher bestimmte Konstante.) Der Beweis soll im folgenden angedeutet werden.

Lemma:

Vor.: ρ ist ein Programm mit:
für alle Programme π , für alle $K \in \mathbb{N}$ gilt

$$S_1(\rho)[\overline{\pi},K,0,\ldots,0] = \begin{cases} 1 \text{ , falls } S_1(\pi)[\overline{\pi},K,0,\ldots,0] = 0 \text{ und } \pi \text{ terminiert auf } [\overline{\pi},K,0,\ldots,0] \text{ in } \leq 2^{2^K} \\ \qquad \text{Schritten.} \\ 0 \text{ , sonst.} \end{cases}$$

Beh.: ρ terminiert auf $[\overline{\rho},K,0,\ldots,0]$ erst nach mehr als 2^{2^K} Schritten (für alle K).

Beweis: Sei ρ ein Programm, welches den Voraussetzungen genügt.
Sei $K \in \mathbb{N}$ beliebig.
Da die Annahme $S_1(\rho)[\overline{\rho},K,0,\ldots,0] = 1$ sofort auf einen Widerspruch führt, gilt: $S_1(\rho)[\overline{\rho},K,0,\ldots,0] = 0$ und somit $\neg(S_1(\rho)[\overline{\rho},K,0,\ldots,0] = 0$ in $\leq 2^{2^K}$ Schritten). Also terminiert ρ erst nach mehr als 2^{2^K} Schritten.

q.e.d.

Satz: Vor.: Σ ein Programm mit

$$S_1(\Sigma)[\overline{\varphi},0,\ldots,0] = \begin{cases} 1 \text{ , falls } \underline{\mathbb{N}} \models \varphi \\ 0 \text{ , falls } \underline{\mathbb{N}} \not\models \varphi \end{cases}$$

für alle Formeln φ aus $L(+,=,0,1)$.

Beh.: Es existiert c , sodass für unendlich viele Formeln φ , das Programm Σ mehr als $2^{2^{c\cdot|\overline{\varphi}|}}$ Schritte benötigt, wobei $|K|$ die Länge der Darstellung der natürlichen Zahl K im Dualsystem bezeichnen soll, d.h. $|K| \approx \log_2 K \cdot \text{const}$.

Beweisskizze

1. Schritt: Zu jedem Programm π und zu jeder natürlichen Zahl K , kann man eine Formel $\varphi_{\pi,K} \in L(+,=,0,1)$ finden mit:

$\varphi_{\pi,K}$ wahr, genau wenn $S_1(\pi)[\overline{\pi},K,0,\ldots,0] = 0$

und π terminiert auf $[\overline{\pi},K,0,\ldots,0]$ in höchstens 2^{2^K} Schritten.

und $|\overline{\varphi_{\pi,K}}| \leq c_\pi \cdot K + 1_\pi = O(K)$

(d.h. die Länge der Gödelnummer von φ wächst linear mit K).

Das Auffinden der Formel $\varphi_{\pi,K}$ ist die eingentliche Schwierigkeit des ganzen Beweises. Der Haupttrick besteht darin, dass man jedem i eine Formel $M_i^*(x,y,z) \in L(+,=,0,1)$ und $r_i \geq 2^{2^{2^i}}$ finden kann mit

$\underline{\mathbb{N}} \models M_i^*[x,y,z] \Leftrightarrow x = y\cdot z$ und $x,y,z \leq r_i$ und $|M_i^*| \leq c_1\cdot i + c_2 = O(i)$.

Da in einer Berechnungsfolge $\langle y_{11},\ldots,y_{1n},a_1,\ldots,y_{m1},\ldots,y_{mn},a_m\rangle$ mit $m \leq 2^{2^K}$ und $y_{11} = K$ und $y_{12} = 0,\ldots,y_{1n} = 0$ (d.h. Input = $[K,0,0,\ldots,0]$) nur Elemente $\leq 2^{2^K} + K \leq 2^{2^{K+1}}$ vorkommen, und damit auch die Gödelnummer b,c,m der Berechnungsfolge $\ldots \leq 2^{2^{2^{K+4}}}$ werden, kann man aus B_π (siehe 3.4, 2. Schritt) mit Hilfe von M_i^* eine Formel $\varphi_{\pi,K} \in L(+,=,0,1)$ erhalten, mit $|\varphi_{\pi,K}| \leq c'_\pi\cdot K + 1'_\pi$. Wählt man eine der üblichen Gödelnumerierungen, so folgt $|\overline{\varphi_{\pi,K}}| \leq c_\pi \cdot K + 1_\pi$ für geeignete $c_\pi, 1_\pi$.

2. Schritt: Man betrachte nun das folgende Programm ρ

```
begin
if "X1 Gödelnummer eines Programmes π" then "X1 := φ_{π,x2}";
{Aufwand bis hierher ≤ a_π + x_2^(b_π); a_π, b_π sind nicht näher
 bestimmte Konstanten}
X2 := 0;
Σ
end
```

Das Programm ρ genügt den Voraussetzungen des obigen Lemmas.

ρ terminiert somit auf $[\overline{\rho},K,0,\ldots,0]$ erst nach mehr als 2^{2^K} Schritten. Dies impliziert (denn δ wurde extra so konstruiert)

Σ terminiert auf $[\overline{\varphi_{\overline{\rho},K}},0,\ldots,0]$ erst nach mehr als $2^{2^K}-a_\rho-K^{(b_\rho)}-1$ Schritten, während dem die Länge des Inputs $[\overline{\varphi_{\overline{\rho},K}},0,\ldots,0]$, wie im 1. Schritt gezeigt wurde, höchstens mit $c_\rho \cdot K + 1_\rho$ wächst.

q.e.d.

Einführende Literatur

Engeler, E.: Introduction to the Theory of Computation (Academic Press, 1973)

Hermes, H.: Aufzählbarkeit Entscheidbarkeit Berechenbarkeit (Heidelberger Taschenbücher, Springer, 1971: 2. Auflage)

Hopcroft, J.-Ullmann, J.: Formal Languages and their Relation to Automata (Addison-Wesley, 1969)

Minsky, M.: Computation: Finite and Infinite Machines (Prentice-Hall, 1967)

[dasselbe in Deutsch:] Berechnung: Endliche und Unendliche Maschinen (Berliner Union, 1971)

Nivat, M. (editor): Automata, Languages and Programming (Proceedings of a symposium (1972) organized by IRIA, North-Holland, 1973).