BEMERKUNGEN ZUR ÜBERSETZBARKEIT REKURSIVER IN NICHTREKURSIVE UNTERPROGRAMMSTRUKTUREN

von Herbert Kopp

A 74 /12

BEMERKUNGEN ZUR ÜBERSETZBARKEIT REKURSIVER UNTERPROGRAMMSTRUKTUREN IN NICHTREKURSIVE

von Herbert Kopp

Die wesentlichen Merkmale einer Programmiersprache sind

a) die Elementaroperationen
b) die Unterprogrammstruktur
c) die Datenstruktur

Man nimmt im allgemeinen an, daß neben der Datenstruktur und den Elementaroperationen auch die Unterprogrammstruktur einen entscheidenden Einfluß auf die Leistungsfähigkeit einer Programmiersprache hat. Dabei denkt man vor allem an rekursive und nicht-rekursive Aufruftechniken.

Eingehende Untersuchungen in dieser Richtung sind bisher noch nicht angestellt worden. die vorliegende Arbeit ist ein Versuch, die Fragestellung zu präzisieren und zu ersten Ergebnissen zu gelangen. Wir wollen hier vor allem auf die folgenden Problemstellungen eingehen:

- Wann kann man rekursive Programme in nichtrekursive verwandeln? Gibt es generelle Verfahren dafür?
- Unter welchen Voraussetzungen über die Elementaroperationen und die Datenstruktur einer Programmiersprache kann man mit rekursiven Programmen mehr Funktionen berechnen als mit nichtrekursiven?

I. Ein einfaches Modell für Programmiersprachen

a) Definition der Syntax

Es sei $A = \{a_1,\ldots,a_r\}$ eine endliche Menge von Anweisungssymbolen,

$F = \{f_1,f_2,\ldots\}$ eine abzählbare, reguläre Menge von Unterprogrammnamen und

$P = \{p_1,\ldots,p_s\}$ eine endliche Menge von Prädikatssymbolen.

Wir bauen die Programme aus den folgenden Zeilentypen auf:

Die Menge der Startzeilen sei

$$\alpha = \{ \underline{\text{begin}}\ \underline{\text{goto}}\ i\ ; \mid i \in \mathbb{N} \}$$

Die Menge der Anweisungszeilen sei

$$\beta = \{ i : \underline{\text{do}}\ a\ ; \underline{\text{goto}}\ j\ ; \mid i, j \in \mathbb{N},\ a \in A \}$$

Die Menge der Unterprogrammaufrufe sei

$$\beta_1 = \{ i : \underline{\text{do}}\ f\ ; \underline{\text{goto}}\ j\ ; \mid i, j \in \mathbb{N},\ f \in F \}$$

Die Menge der Unterprogrammstartzeilen sei

$$\alpha_1 = \{ f : \underline{\text{goto}}\ j\ ; \mid j \in \mathbb{N},\ f \in F \}$$

Die Menge der bedingten Anweisungen sei

$$\gamma = \left\{ i : \underline{\text{if}}\ p\ \underline{\text{then goto}}\ j\ \underline{\text{else goto}}\ k\ ; \;\middle|\; \begin{matrix} i,j,k \in \mathbb{N} \\ p \in P \end{matrix} \right\}$$

Die Menge der Unterprogrammrücksprünge sei

$$\omega_1 = \{ i : \underline{\text{return}}\ ; \mid i \in \mathbb{N} \}$$

Die Menge der Programmschlußzeilen sei

$$\omega = \{ i : \underline{end} ; \mid i \in \mathbb{N} \}$$

Wir setzen der Einfachheit voraus, daß die Menge

$$\alpha \cup \alpha_1 \cup \beta \cup \beta_1 \cup \gamma \cup \omega \cup \omega_1$$

frei ist, d.h. ein freies Erzeugendensystem des von ihren Elementen erzeugten Untermonoids im Basisalphabet.

Die syntaktische Grundstruktur unserer Programme wird durch die folgende (reguläre) Menge von Programmformen definiert:

$$\mathbb{P} = \alpha \cdot \left(\beta \cup \beta_1 \cup \gamma \cup \alpha_1 \cdot (\beta \cup \beta_1 \cup \gamma)^* \cdot \omega_1 \right) \cdot \omega$$

Eine Programmform $\pi \in \mathbb{P}$ ist somit eindeutig zerlegbar in ein Produkt der Form

$$\pi = u_1 \cdot \ldots \cdot u_n$$

wobei die Faktoren u_i entweder Zeilen vom Typ α, β, β_1, γ oder ω sind oder aber $u_i \in \alpha_1 \cdot (\beta \cup \beta_1 \cup \gamma)^* \cdot \omega_1$ ist.

Im zweiten Fall nennen wir den Faktor u_i eine Unterprogrammform, im ersten Fall heißt u_i eine Zeile auf der Programmebene.

Definition: Eine Programmform $\pi \in \mathbb{P}$ heißt ein Programm der Programmiersprache $\prod_r$, wenn π die folgenden Eigenschaften besitzt:

i) Jede Sprunganweisung "goto i ;" ist eindeutig ausführbar.

ii) Sprünge in Unterprogamme hinein oder aus Unterprogrammen heraus sind verboten.

iii) Zu jedem Unterprogrammaufruf "do f ;" gibt es genau ein Unterprogramm mit dem Namen f .

Bemerkung: Die Bedingungen der obigen Definition kann man auch leicht streng formal angeben, wie dies zum Beispiel in [ ] durchgeführt worden ist.

b) Definition der Semantik

Zur formalen Festlegung des Berechnungsablaufes von Programmen benötigen wir den Begriff des Programmzustandes:

Definition: Es sei $\pi \in \overline{\Pi}_r$.

i) 0 ist ein Programmzustand von π.

ii) Ist i die Marke einer Zeile auf der Programmebene, dann ist auch i ein Programmzustand von π.

iii) Wenn es auf der Programmebene eine Zeile gibt

i : do f ; goto j ;

dann ist auch fj ein Programmzustand von π.

iv) Ist $f_n j_n \ldots . f_1 j_1$ ein Programmzustand und j_{n+1} die Nummer einer Zeile im Unterprogramm f_n, dann ist auch $j_{n+1} f_n j_n \ldots . f_1 j_1$ ein Programmzustand.

v) Ist $if_{n-1} j_{n-1} \ldots . f_1 j_1$ ein Programmzustand und gibt es im Unterprogramm mit dem Zeile f_{n-1} eine Zeile

i : do f_n ; goto j_n ;

dann ist auch $f_n j_n f_{n-1} j_{n-1} \ldots . f_1 j_1$; ein Programmzustand von π.

Die Menge der Programmzustände eines Programms $\pi \in \prod_r$ bezeichnen wir mit $Z(\pi)$. $Z(\pi)$ ist eine reguläre Menge.

Definition: Ein Programm $\pi \in \prod_r$ heißt rekursiv, falls $|Z(\pi)| = \infty$ ist. Im andern Fall heißt π nichtrekusiv. Die Menge der nichtrekursiven Programme bezeichnen wir mit $\prod_1$.

Hilfssatz 1: Es ist entscheidbar, ob ein Programm π rekursiv ist.

Hilfssatz 2: Jedem Programmzustand $\pi \in Z(\pi)$ läßt sich in eindeutiger Weise eine Programmzeile von π zuordnen.

Definition: Unter einer Interpretation für die Sprache $\prod_r$ verstehen wir ein Tripel $\Phi = (\Delta, \varphi_1, \varphi_2)$

mit: Δ ist eine beliebige Menge

$\varphi_1 : A \longrightarrow \{\delta : \Delta \longrightarrow \Delta\}$ ist eine Abbildung

$\varphi_2 : P \longrightarrow \{\delta : \longrightarrow \{0,1\}\}$ ist eine Abbildung

Durch Δ wird die Datenstruktur und durch φ_1, φ_2 wird die Interpretation der Anweisungen und der Prädikate festgelegt.

Definition: Unter einer Semantik für die Sprache $\prod_r$ verstehen wir nun eine Zuordnung

$$\sigma : \prod_r \longrightarrow \{\sigma(\pi) : Z(\pi) \times \Delta \longrightarrow Z(\pi) \times \Delta \mid \pi \in \prod_r\}$$

$$\pi \longmapsto \sigma(\pi) : Z(\pi) \times \Delta \longrightarrow Z(\pi) \times \Delta$$

Durch wird jedem Programm eine Abbildung auf $Z(\pi) \times \Delta$ zugeordnet mit den folgenden Eigenschaften:

Es sei $d \in \Delta$.

1. $\sigma(\pi)(0,d) = (i,d)$ falls "begin goto i ; " eine Zeile von π ist.

2. $\sigma(\pi)(iz,d) = (iz,d)$ falls " i : end ; " eine Zeile von π ist.

3. $\sigma(\pi)(iz,d) = (jz,\varphi_1(a)(d))$ falls die durch iz eindeutig bestimmte Zeile die Gestalt hat: "i : do a ; goto j ; " mit $a \in A$

4. $\sigma(\pi)(iz,d) = (fjz,d)$ falls die durch iz eindeutig bestimmte Zeile die Gestalt hat : "i : do f ; goto j ; " mit $f \in F$

5. $\sigma(\pi)(fz,d) = (ifz,d)$ falls die Startzeile des Unterprogramms mit dem Namen f lautet : "f : goto i ; "

6. $$\sigma(\pi)(iz,d) = \begin{cases} (jz,d) & \text{falls die durch iz bestimmte Zeile lautet : "if p then goto j else goto k;" und } \varphi_2(p)(d) = 1 \text{ ist} \\ (kz,d) & \text{wie oben, aber } \varphi_2(p)(d) = 0 \end{cases}$$

7. $\sigma(\pi)(iz,d) = (z,d)$ falls die durch iz bestimmte Zeile lautet : "i : return ;"

Unter der Berechnung eines Programmes $\pi \in \prod_r$ verstehen wir nun eine Folge

$$(0,d),\ \sigma(\pi)(0,d),\ldots\ldots\ldots$$

Falls diese Folge einen Fixpunkt (j,d') besitzt, wobei

j : end ;

die Schlußzeile im Programm π ist, dann sagen wir, d' sei das Resultat der Berechnung und schreiben

$$Res_\pi(d) = d'$$

π berechnet dann die Funktion $f_\pi : \Delta \longrightarrow \Delta$ bzgl. einer Interpretation ϕ falls für alle $d \in \Delta$ gilt $f_\pi(d) = Res_\pi(d)$ sofern $f_\pi(d)$ definiert ist, und andernfalls auch $Res_\pi(d)$ nicht existiert.

Definition: Sind $\widehat{\prod}$ eine weitere Programmiersprache und

$$\ddot{u} : \prod_r \longrightarrow \widehat{\prod}$$

eine rekursiv berechenbare Abbildung, dann heißt $\ddot{u}$ eine korrekte Übersetzung, falls für alle Interpretationen ϕ von $\prod_r$ und für alle Interpretationen ϕ von $\widehat{\prod}$ gilt:

$$f_\pi = f_{\ddot{u}(\pi)}$$

Bemerkung: Die eingangs gestellte Frage kann damit folgendermaßen präzisiert werden:

- Für welche Interpretationen ϕ ist

$$\left\{ f_\pi \mid \pi \in \prod_r \right\} = \left\{ f_\pi \mid \pi \in \prod_1 \right\} \; ?$$

- $\mathcal{J}$ sei eine Klasse von Interpretationen mit

$$\left\{ f_\pi \mid \pi \in \prod_r \right\} = \left\{ f_\pi \mid \pi \in \prod_1 \right\}$$

für alle $\phi \in \mathcal{J}$. Unter welchen Voraussetzungen kann man eine relativ zu $\mathcal{J}$ korrekte Übersetzung angeben, d.h. für die

$$f_\pi = f_{\ddot{u}(\pi)}$$

ist für alle $\phi \in \mathcal{J}$.

II. Zum Problem der Übersetzbarkeit von Π_r in Π_l

Ob eine Abbildung $ü : \Pi_r \longrightarrow \Pi_l$ eine korrekte Übersetzung ist, hängt nun wesentlich davon ab, bezüglich welcher Klasse von Interpretationen man die Korrektheit verlangt. Wir betrachten zunächst den Fall, daß Δ eine endliche Menge ist.

Bemerkung: Bei umfangreichen Datenstrukturen bereitet es im allgemeinen keine Schwierigkeiten, ein rekursives Programm dadurch in ein nichtrekursives zu verwandeln, daß man die Rücksprungorganisation mit in die Datenkomponente aufnimmt; die Verwaltung der Rücksprünge wird dann explizit vom Programm übernommen. Da ein solches Vorgehen nicht in unserem Sinne ist, erscheint es sinnvoll, dies durch eine Beschränkung von Δ zu verhindern.

Dazu wurde von G.Hotz der Begriff der k-universellen Übersetzung eingeführt:

Definition: Eine Abbildung $ü : \Pi_r \longrightarrow \Pi_l$ heißt eine k-universelle Übersetzung von Π_r in Π_l, falls $ü$ eine korrekte Übersetzung ist für jede Interpretation $\phi = (\Delta, \varphi_1, \varphi_2)$ mit $|\Delta| \leq k$ und $k \in \mathbb{N}$.

Satz 1: [3] Zu jedem $k \in \mathbb{N}$ gibt es eine k-universelle Übersetzung von Π_r in Π_l.

Beweis: [3]

Es ist entscheidbar, welche Unterprogramme eines Programms π rekursiv aufgerufen werden. Führt man die Kopierregel so lange durch, bis jedes dieser Unterprogramme mindestens k-mal eingesetzt worden ist, und ersetzt man dann alle weiteren Aufrufe eines rekursiv aufgerufenen Unterprogramms durch eine unendliche Schleife, dann hat man ein Π_l-Programm, das die gleiche Funktion wie π berechnet.

Der folgende Satz wurde ebenfalls in [3] bewiesen:

Satz 2: Es gibt keine Übersetzung von Π_r in Π_1, die k-universell ist für jedes $k \in \mathbb{N}$.

Beweis:

Wir betrachten eine Folge von Interpretationen

$$\phi_n = (\Delta_n, \varphi_{1,n}, \varphi_{2,n})$$

für $A = \{N, c_1\}$

und $P = \{p\}$

Dabei sei $\Delta_n = \{0,1,.....,n-1\}$

und $\varphi_{1,n}(N)(i) = i+1 \bmod n$ für alle $i \in \mathbb{N}$

$\varphi_{1,n}(c_1)(i) = 1$ für alle $i \in \mathbb{N}$

$\varphi_{2,n}(p)(i) = 1$ gdw $i = 0$

Wir betrachten nun ein Programm, welches für jeden Startwert $i \in \{0,1,....,n\}$ den Wert n liefert, wenn man die Interpretation ϕ_{n+1} zugrundelegt.

π_o :

```
begin  goto 1 ;
  1 :  do  c1  ; goto 2 ;
  2 :  do  f   ; goto 3 ;
       f : goto 1 ;
       1 : do  N  ; goto 2 ;
       2 : if  p  then goto  4  else goto  3 ;
       3 : do  f  goto 4 ;
       4 : do  N  goto 5 ;
       5 : return ;
  3 :  end ;
```

Ist nun $\ddot{u} : \prod_r \longrightarrow \prod_1$ eine Abbildung und hat das Programm $\ddot{u}(\pi_o)$ n_o Zeilen, dann gilt für jede der Interpretationen ϕ_{n+1}:

$$f_{\ddot{u}(\pi_o)}(i) \leq i + n_o$$

Setzt man nun $n-1 > n_o$, dann gilt

$$n = f_{\pi_o}(1) > n_o+1 \geq f_{\ddot{u}(\pi_o)}(1)$$

$\ddot{u}$ ist daher bezüglich ϕ_{n+1} keine korrekte Übersetzung.

Bemerkung: Übersetzungen die rekursive in nichtrekursive Programme transformieren sind demnach nicht zu erwarten, wenn man verlangt, daß sie für alle Interpretationen $\phi = (\Delta, \varphi_1, \varphi_2)$ mit endlichem Δ korrekt sein sollen.

Andererseits heißt dies, daß jede solche Übersetzung wesentlich die Eigenschaften der Datenstruktur mit einbeziehen muß.

Bei der Untersuchung der Übersetzbarkeit von $\prod_r$ in $\prod_1$ ist stets die Vorfrage zu klären, ob in den zugrundegelegten Interpretationen überhaupt in $\prod_r$ die gleichen Funktionen berechnet werden können wie in $\prod_1$. Erst wenn diese Frage positiv beantwortet werden kann, ist es sinnvoll nach Übersetzungen zu suchen, d.h. nach effektiven Verfahren, die die Transformation bewirken.

Beispiel: Es sei $A = \{\oplus, +, c_o\}$

$P = \{p\}$

$= \mathbb{N}_o$

Weiter sei $\varphi_1(\oplus)(n) \equiv n+1 \bmod 10^{[\log n]+1}$

$\varphi_1(+)(n) = n+1$

$\varphi_1(c_o)(n) = 0$

und außerdem sei $\varphi_2(p)(n) = 1 \iff n = 0$

Behauptung: Mit der obigen Interpretation $\phi = (\Delta, \varphi_1, \varphi_2)$ kann man in $\prod_r$ mehr Funktionen berechnen als in $\prod_1$.

Zum Beweis geben wir ein Programm an, das die Funktion

$$\mu : \mathbb{N}_o \longrightarrow \mathbb{N}_o$$
$$n \longmapsto 10^{[\log n]+1} - n$$

berechnet und in $\prod_r$ ist.

(Mit $[z]$ bezeichnen wir die größte ganze Zahl, die kleiner oder gleich z ist.)

π_1 :

```
begin  goto 1 ;
   1 : do  f  ; goto 2 ;
       f: goto 1 ;
       1: do ⊕ ; goto 2 ;
       2: if  p  then goto  3  else goto 4 ;
       3: do  +  ; goto 5 ;
       4: do  f  ; goto 3 ;
       5: return ;
   2 : end ;
```

π_1 berechnet offenbar die angegebene Funktion. Es gibt jedoch kein π_1-Programm, das μ berechnet, wie man aus dem nachfolgenden Hilfssatz 4 schließen kann.

Die in diesem Beispiel zugrundeliegende Situation soll nun verallgemeinert werden zu einem Kriterium dafür, wann man mit $\prod_r$ mehr Funktionen berechnen kann als mit $\prod_1$.

Zunächst benötigen wir jedoch noch einen weiteren Hilfssatz:

Hilfssatz 3: Bezeichnen wir mit $\prod_o$ die Menge der Programme $\pi \in \prod_r$, die keine Unterprogramme besitzen, so gibt es eine Übersetzung

$$\ddot{u} : \prod_1 \longrightarrow \prod_o$$

die korrekt ist bezüglich jeder Interpretation ϕ .

Beweis: Für nicht-rekursive Programme führt die Iteration der Kopierregel zu einem $\prod_0$-Programm, da der Einsetzungsprozeß abbricht.

Hilfssatz 4: $\phi = (\Delta, \varphi_1, \varphi_2)$ sei eine Interpretation mit folgender Eigenschaft:

Für alle $p \in P$ ist entweder $\left|\varphi_2(p)^{-1}(0)\right| < \infty$

oder $\left|\varphi_2(p)^{-1}(1)\right| < \infty$

Dann gibt es zu jedem $\pi \in \prod_1$ Operationen $\alpha_1, \ldots, \alpha_r \in A$ und eine endliche Zerlegung von Δ:

$$\Delta = \Delta_0 \cup \Delta_1 \cup \ldots\ldots \cup \Delta_s$$

so daß gilt $f_\pi \left|\Delta_0 = \varphi_1(\alpha_1) \circ \ldots\ldots \circ \varphi_1(\alpha_s)\right| \Delta_0$

und $f_\pi \left|\Delta_i = d_i \in \Delta \right.$ für $i = 1, \ldots, s$

Bemerkung: Unter den genannnten Voraussetzungen läßt sich also die Funktion f_π im wesentlichen, d.h. bis auf endlich viele Funktionswerte durch einen Ausdruck über den Elementaroperationen der Programmiersprache darstellen. Die obige Funktion μ ist nicht so darstellbar.

Beweis:

Wegen Hilfssatz 3 können wir ohne Beschränkung der Allgemeinheit annehmen, daß $\pi \in \Delta_0$ ist.

Es gibt nun höchstens einen Weg durch das zu π gehörende Programmschema, der bei der Startzeile beginnt und bei der Programmschlußzeile endet und der bei jeder

bedingten Anweisung " i : if p then goto j else goto k ; " diejenige Alternative durchläuft, für die gilt

$$\left|\varphi_2(p)^{-1}(b)\right| < \infty$$

Daraus folgt nun aber: Es gibt eine endliche Menge $\Delta_1 \subset \Delta$, so daß jede Berechnung mit einem Startwert $d \in \Delta$ den angegebenen Weg durch das Programm nimmt oder es ist $Res_\pi(d) \in \prod_1$.

Damit ist der Hilfssatz aber bewiesen.

Der folgende Satz gibt nun ein einfach zu überprüfendes Kriterium dafür an, daß man in $\prod_r$ mehr Funktionen berechnen kann als in $\prod_1$.

Satz 3: ϕ sei eine Interpretation mit den folgenden beiden Eigenschaften:

i) für alle $p \in P$ ist
   entweder $\left|\varphi_2(p)^{-1}(0)\right| < \infty$
   oder $\left|\varphi_2(p)^{-1}(1)\right| < \infty$

ii) In $\prod_r$ ist eine Funktion berechenbar, die sich nicht bis auf endlich viele Funktionswerte durch einen Ausdruck darstellen läßt im Sinne von Hilfssatz 4.

Dann sind in $\prod_r$ mehr Funktionen berechenbar als in $\prod_1$.

III. Der Zusammenhang mit automatentheoretischen Betrachtungsweisen

Wenn die Interpretation ϕ so beschaffen ist, daß wir die für den rekursiven Aufruf notwendige Organisation in die Datenstruktur mit übernehmen können, dann bereitet eine Übersetzung von $\prod_r$ in $\prod_1$ keine Schwierigkeiten.

Anders wird die Situation jedoch, wenn wir Datenstrukturen Δ betrachten, bei denen dies nicht ohne weiteres möglich ist, etwa wenn der Speicher in irgendeiner Weise durch die Eingabe beschränkt ist. In diesem speziellen Fall führt das Problem der Übersetzbarkeit von $\prod_r$ in $\prod_1$ auf bekannte Probleme der Automatentheorie.

Eine Interpretation zur Simulation von Automaten in $\prod_r$

Es seien B und Z disjunkte, endliche Mengen. Als Datenstruktur wählen wir

$$\Delta = \{\dashv\} \cdot B^* \cdot Z \cdot B^* \{\vdash\}$$

Unter B kann man sich das Bandalphabet und unter Z die Zustandsmenge eines endlichen Automaten vorstellen.

Als Operationssymbole wählen wir

moveL

moveR

print(b,z) für $b \in B$

Die Prädikatssymbole seien von der Form $\underline{\text{state}} = (\beta, z)$ für $\beta \in B$ und $z \in Z$.

Die Interpretierenden Abbildungen definieren wir folgendermaßen:

$\varphi_1(\underline{\text{moveL}})(\dashv b_1 \ldots b_k z b_{k+1} \ldots b_r \vdash) = \dashv b_1 \ldots b_{k-1} z b_k \ldots b_r \vdash$

falls $z \in Z$ und $k \geq 1$ ist.
für $k = 1$ ist $\varphi_1(\underline{\text{moveL}})$ nicht definiert.

$\varphi_1(\underline{\text{moveR}})(\dashv b_1 \ldots b_k z b_{k+1} \ldots b_r \vdash) = \dashv b_1 \ldots b_{k+1} z b_{k+2} \ldots b_r \vdash$

falls $z \in Z$ und $k+1 < r$ ist.
Sonst ist $\varphi_1(\underline{\text{moveR}})$ nicht definiert.

$\varphi_1(\text{print}(b,z))(\ldots\ldots z'b'\ldots\ldots) = \ldots\ldots zb\ldots\ldots$

$\varphi_2(\underline{\text{state}}=(\beta,z'))(\ldots\ldots zb\ldots\ldots) = 1 \iff z = z' \text{ und } \beta = b$

Es gilt nun der folgende Satz:

Satz 4: Mit der oben angegebenen Interpretation kann man in $\prod_r$ genau den 'writing pushdown'-Automaten [4] simulieren und in $\prod_1$ kann man genau den deterministischen, linear beschränkten Automaten simulieren.

Ein Programm zur Simulation des deterministischen 'writing pushdown'-Automaten:

Es sei $B = \{b_0,\ldots,b_m\}$ das Bandalphabet des Automaten

$Z = \{z_0,\ldots,z_n\}$ sei die Zustandsmenge und

$\Gamma = \{\gamma_0,\ldots,\gamma_p\}$ sei sein Kelleralphabet.

```
begin goto 1 ;
   1 : do  γ0 ; goto 2 ;
     γ0 : goto (0.0) ;
  (0.0) : if state=(b0,z0) then goto (0.0.1) else goto (0.1) ;
  (0.1) : if state=(b0,z1) then goto (0.1.1) else goto (0.2) ;
                    .
                    .
                    .
(m.n-1) : if state=(bm,zn-1) then goto (m.n-1.1) else goto (m.n.1) ;
(0.0.1) : do  AOP(0.0) ; goto (0.0.2) ;
(0.0.2) : KOP(0.0) ;
(0.1.1) : do  AOP(0.1) ; goto (0.1.2) ;
(0.1.2) : KOP(0.1) ;
                    .
                    .
                    .
(m.n.1) : do  AOP(m.n) ; goto (m.n.2) ;
(m.n.2) : KOP(m.n) ;
(m.n.3) : return ;
```

γ_1 : ..... ; return ;

γ_2 : ..... ; return ;

⋮

γ_p : ..... ; return ;

2 : end ;

Dabei bedeutetn AOP(i.j) eine Arbeitsfeldoperation d.h. eine Anweisung des Typs :

moveL , moveR , print(b,z)

und KOP dient zur Simulation von Kelleroperationen, das heißt an diesen Stellen stehen Anweisungen des Typs :

do γ_i ; goto (0.0) ;

bzw. goto (m.n.3) ;

Die Unterprogramme $\gamma_1,\ldots,\gamma_p$ haben die gleiche Struktur wie das Unterprogramm γ_o.

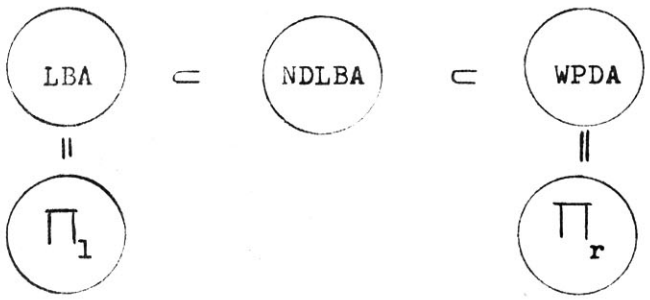
Bemerkung: Man überlegt sich leicht, daß man einen nichtdeterministischen WPDA durch ein Programm von der gleichen Struktur simulieren kann. Außerdem kann man jedes $\prod_r$-Programm so umformen, daß es die obige Gestalt erhält. Somit lassen sich in $\prod_r$ genau die WPDA simulieren.

Durch Satz 4 erhalten wir nun den folgenden Zusammenhang mit der Automatentheorie :

Faßt man den LBA, den NDLBA und den WPDA als Automaten auf die Funktionen

$$f : \Sigma^* \longrightarrow \mathbb{N}_o$$

berechnen, dann erhält man für die Klassen der in $\prod_1$, $\prod_r$, mit dem LBA, dem NDLBA sowie mit dem WPDA berechenbaren Funktionen die folgenden Inklusionen :



Ob die angegebenen Inklusionen echt sind, ist bekanntlich seit langem offen.

Literatur :

[1] Cook : Characterizations of Pushdown Machines in Terms of Time-Bounded Computers. JACM 18,Nr.1 (1971)

[2] Hotz : Grundlagen einer Theorie der Programmiersprachen
I. Rekursive und nichtrekursive Unterprogrammtechniken.
II. Typen, Operationen, Ausdrücke
III. Datenstrukturen

[3] Hotz : Vorlesung 'Theorie der Programmiersprachen' im Sommersemester 1972 in Saarbrücken.

[4] Kopp : Beiträge zur Theorie der Programmiersprachen Dissertation an der Universität des Saarlandes 1973.

[5] Mager : Writing Pushdown Acceptors Jcss, Nr.3, (1969)

[6] Monien : Komplexitätsklassen von Automatenmodellen und beschränkte Rekursion Univ. Hamburg, 1974