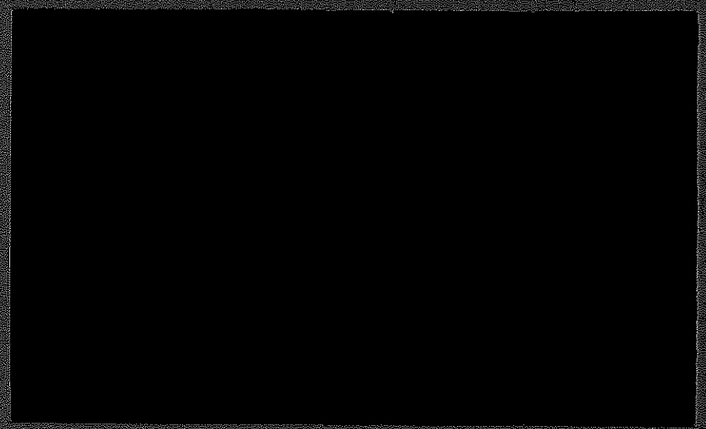


SCHRIFTEN ZUR INFORMATIK

UND ANGEWANDTEN MATHEMATIK

Arbeitstagung

"Theorie der Programmierung"

13.12.-16.12.1982 in Altenahr

veranstaltet von K. Indermark

Bericht Nr. 87 April 1983

Herausgeber : K. Indermark, J. Merkwitz, W. Oberschelp,
B. Schinzel, W. Thomas
RWTH Aachen, Templergraben 55, D-5100 Aachen

INHALT

ÄQUIVALENZ AKZEPTIERBARER UND ALGOL-ARTIGER PROGRAMMIERSPRACHEN

CH. CRASEMANN
H. LANGMAACK

KIEL

Im Satz von Lipton ([Li 77]) über effektive Aufzählung von partiell korrekten Hoareschen Zusicherungen ist von akzeptablen Programmiersprachen die Rede. In Lipton's vager Formulierung handelt es sich dabei um Programme, die von einem Interpretierer ausgeführt werden können. Clarke, German und Halpern ([CGH 82]) präzisieren Lipton's Formulierung dahingehend, daß Programme zu Funktionen äquivalent sein müssen, die über einer algebraischen Struktur effektiv durch einen deterministischen Algorithmus berechnet werden können. Die Klasse derartiger Funktionen kann im Sinne einer geeignet verallgemeinerten Church-Turing These durch verschiedene Ansätze definiert werden.

Beispiele sind H. Friedman's "effective definitional schemes" ([Fr 69], [Ti 79]), seine verallgemeinerten Turing Algorithmen (ibid.) oder while-Programme mit unendlichem ($\mathbb{N}_o$-indiziertem) Array und Arithmetik ([CG 72], [Gr 75]).

Die letztere Charakterisierung wird der Ausgangspunkt für den Beweis, daß eine ALGOL-artige Programmiersprache mit vollem Prozedurkonzept ohne Arithmetik und ohne Array, also nur mit einfachen Variablen, die gleiche Klasse von Funktionen berechnet, also universell im Sinne einer verallgemeinerten Church-Turing These ist.

Ein Schritt ist einfach. Jedes ALGOL-artige Programm mit Prozeduren kann durch ein while-Programm mit Array und Arithmetik simuliert werden, indem ein Laufzeitsystem durch rekursive Funktionen codiert wird und das Array die Aufgabe der (unbeschränkt vielen)

beim Programmablauf benutzten (dynamisch erzeugten) Variablen erfüllt.

Die umgekehrte Richtung des Äquivalenzbeweises hat zwei Teile. Ausgehend von einem while-Programm mit Array und Arithmetik wird zunächst das Array ersetzt durch geschachtelte Prozedurdeklarationen, welche vom PASCAL-Typ sind (Arttiefe 2) und indizierte Variable durch globale Variable ihrer direkten Umgebung ersetzen. Dies erfordert die Überführung eines Programmes in eine iterativ-artige (tail-recursion-artige) Form ([BaW 81]). In einer weiteren Transformation wird sodann die Wirkung rekursiver Funktionen durch geeignete Prozeduraufrufe simuliert. Dabei wird insbesondere von Prozedurschachtelungen und Selstapplikation Gebrauch gemacht.

Als Folgerung aus diesem Ergebnis wird eine offene Frage aus dem Gebiet der vergleichenden Programmschematologie ([CG 72]) beantwortet. Zuletzt bemerkt A. Critcher in [(C 82)]: "It is not clear even with passing parameters by name, whether recursion has the same power as infinite arrays" (mit Arithmetik). Unser Resultat zeigt nun die Gleichmächtigkeit beider Konzepte. Wesentlich ist dabei das Zusammenspiel von Prozeduren und Prozedurschachtelungen: Programme ohne letzeres Konzept (und ohne das äquivalente Konzept höherer Funktionalitäten ([Cr 83])) sind in der Kontrollstruktur gleichmächtig zu Nivat's rekursiven Programmschemata RPS ([Ni 74]), sogar bei erlaubter Selbstanwendung von Prozeduren ([Cr 83]).

Als weitere Folgerung der angegebenen effektiven Äquivalenz ergibt sich die Möglichkeit, ohne Einschränkung der Allgemeinheit ALGOL-artige Programmiersprachen zur Grundlage von Untersuchungen über die Existenz Hoarescher Logiken im Sinne von Lipton zu machen. Dabei können vorhande Hilfsmittel in solchen Programmiersprachen als explizites Maß für die Grenzen des Machbaren benutzt werden.
Eine detaillierte Fassung dieses Vortrages findet sich in [CrLa 83].

Literaturverzeichnis

[BaW 81] Bauer, F., Wössner, H.: Algorithmische Sprache und Programmentwicklung. Springer-Verlag 1981

[Cr 83] Crasemann, Ch.: On the power of higher functionalities and procedure nesting in ALGOL-like programming languages. Bericht des Inst.f.Inform. u.Prakt.Math., Christian-Albrechts-Universität Kiel, April 1983

[CrLa 83] Crasemann, Ch., Langmaack, H.: The characterization of acceptable by ALGOL-like programming languages. Bericht des Inst.f.Inform.u.Prakt.Math., Christian-Albrechts-Universität Kiel, April 1983

[C 82] Critcher, A.: On the ability of structures to store and access information. In: Proceedings of the 9th annual ACM symposium on principles of programming languages, pp. 366-378 (1982)

[CG 72] Constable, R.L., Gries, D.: On classes of program schemata. SIAM J. Comp. 1, pp. 66-118 (1972)

[CGH 82] Clarke, E.M., German, S.M., Halpern, J.Y.: On effective axiomatizations of Hoare logics. In: Proceedings of the 9th annual ACM symposium on principles of programming languages, pp. 309-321 (1982)

[Fr 69] Friedman, H.: Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. In: R.O. Gandy, C.M.E. Yates (eds.): Logic colloquium '69. North Holland Publishing Company, studies in logic and the foundations of mathematics, vol. 61, pp. 361-389 (1971)

[Gr 75] Greibach, S.A.: Theory of program structures: schemes, semantics, verification. LNCS 36, Springer-Verlag 1975

[Li 77] Lipton, R.J.: A necessary and sufficient condition for the existence of Hoare logics. In: 18th IEEE symposium on foundations of computer science, Providence, Rhode Island, pp. 1-6, New York, IEEE 1977

[Ni 74] Nivat, M.: On the interpretation of recursive polyadic program schemes. Symposia Mathematica, Vol 15, 1974

[Ti 79] Tiuryn, J.: Logic of effective definitions. Bericht Nr. 55 der RWTH Aachen, Juli 1979

IMPLEMENTATION TECHNIQUES FOR RECURSIVE TREE TRANSDUCERS ON HIGHER-ORDER DATA TYPES *

W. Damm
Lehrstuhl für Informatik II
RWTH Aachen

I. Guessarian
LITP, UER de Mathématiques
Université Paris VII

*this research was partially supported by the DAAD

INTRODUCTION

This paper aims in investigating the run-time behaviour of *finite-mode* ALGOL68 programs in an abstract setting. It has been demonstrated previously, that results obtained by abstracting from this programming language to the class of *level-n grammars* [Da] can be successfully applied to investigating various properties of the original programming language, ranging from equivalence of denotational - and copy-rule semantics [Da], decidability questions [DaFe], to a completeness theorem for a Hoare-logic [DaJo].

It was suggested in [DaGo] to model the run-time behaviour of ALGOL68 programs with finite mode using a canonical generalization of the pushdown automata to higher levels. The *higher-order datatype* n-PD of level-n pushdown stores can be defined by taking pushdown lists of pairs <pushdown-symbol , level n-1 pushdown store> .

In this paper we extend the characterization in [DaGo] to *level-n tree languages* using a generalization of the concept of *return-addresses* to applicative terms, which - together with substitution - form the underlying datatype of level-n tree grammars. This method is both more general and can be directly applied to the grammar (without constructing a normal-form as required in [DaGo], which would spoil the application in syntax-directed semantics).

We formulate the simulation result in the framework of *recursive-automata theory* recently devellopped by Engelfriet in [Eng 2], which is a streamlined generalization of Scott's approach [Sco] to the inherently recursive situation on trees: a recursive tree automata over an automaton data-type D uses the control-structure of top-down tree automata to generate trees using D as auxiliary storage. This framework allows for uniform definitions and easy comparison of various tree concepts (c.f. [Eng 2]).

In particular we will be interested in *recursive tree-transducers* over D , which are essentially top-down tree transducers with auxiliary storage D . This stems from the fact, that a stack-oriented denotational semantics for finite-mode ALGOL68 programs can be given using recursive tree transducers over n-PD . (This application will be included in a full version of this paper.) In general, recursive tree transducers can be viewed as formal model for storage-oriented syntax-directed semantics. Moreover, as was shown in [Eng 1 , Eng 2] recursive tree transducers over D are very close to *alternating* D-automata : the parallel checking required by alternation

can be modelled in the tree transducer by copying the input, since the input is only accepted if *all* copies are accepted. Thus the domain of recursive tree-transdurcers over D coincides with the alternating D-languages. We will prove a general decomposition result for recursive tree transducers which characterize such translations as composition of (finite state) top-down translations followed by a recursive D-automaton (viewed as a partial identity) and a linear homomorphism . The decomposition technique used is inspired from [Eng 2].

Since the datatypesunderlying level-n grammars and n-PD define the same family of tree languages, it follows from the decomposition result, that also the corresponding classes of tree-translations are equal, and, by the hierarchy result proved in [Eng 3] for alternating n-PD automata, form increasingly powerfull models for syntax directed semantics with increasing level.

Acknowledgements

This paper profits from numerous discussions and the inspiring work of Joost Engelfriet.

BASIC NOTIONS

Let S be a set of *sorts* (or : *base types*). An S-*set* A is a family of sets $(A_s | s\in S)$. For S-set A and B we define the union $A\cup B$ to be the S-set $(A_s \cup B_s | s\in S)$.

The *derived set of sorts* over S is $D(S):=S^* \times S$. An S-*sorted alphabet* Σ is a $D(S)$-set . If $\sigma\in\Sigma_{(w,s)}$, we say that *type* $\sigma=(w,s)$. For the special case where $S=\{s\}$, we call Σ a *ranked alphabet*, abbreviate (s^n,s) to n (for $n\in\omega$, the set of natural numbers), and call n the *rank* (or: *arity*) of $\sigma\in\Sigma_n$. In case Σ contains only unary symbols and one constant, we call Σ a *monadic* alphabet.

For an S-set Y of *parameters* and an S-sorted alphabet Σ we denote by $T_\Sigma(Y)$ the S-set of Σ-trees generated by Y . If $t\in T_\Sigma(Y)^s$, we let *type* $t = s$. We assume that the reader is familiar with the notions of *initial subtree* and *substitution* of trees. We denote the tree obtained by substituting in parallel t_y for y in t (for $y\in Y'\subseteq Y$) by $t[y/t_y]_{y\in Y'}$. If $Y' = \{y_1,\ldots,y_n\}$ is clear from the context we abbreviate this notation to $t(t_{y_1},\ldots,t_{y_n})$ or $t(\vec{t})$.

A *tree-language* is a subset $L\subseteq T_\Sigma$. Throughout this paper we will identify L with the partial identity $\{(t,t) \in T_\Sigma\times T_\Sigma | t\in L\}$, thus viewing it as a particular *tree-translation*. For tree-translations $T_1\subseteq T_{\Sigma_1}\times T_{\Sigma_2}$, $T_2\subseteq T_{\Sigma_2}\times T_{\Sigma_3}$, we denote by $T_1 \circ T_2$ their *relational product* (while for *functions* we use $f_1 \circ f_2$ to denote the composition $x \mapsto f_1(f_2(x))$; we write $f(x) =\uparrow$ to denote the fact the f is undefined on the argument x) . The *domain* of a tree translation T is denoted *dom*(T) .

The construction of derived types can be iterated: $D^\circ(S) := S$, $D^{n+1}(S) := D(D^n(S))$. We call $D^*(S) := \bigcup_n D^n(S)$ the set of *(higher-order) derived types over* S .

For a $D^*(S)$ set X (of *nonterminals*) the $D^*(S)$-set T_X of *applicative terms over* X is defined by $X_\tau \subseteq T_X^\tau$ (for $\tau\in D^*(S)$) , $t\in T_X^{(\alpha,\tau)}$, $t_j\in T_X^{\alpha_j}$ for $j\in[r],\alpha=\alpha_1\ldots\alpha_r \Rightarrow t(t_1,\ldots,t_r)\in T_X^\tau$. If $t\in T_X^\tau$, we let *type* $t = \tau$. We say that an atom x occurs *applied* in t iff t contains a subterm $x(t_1,\ldots,t_n)$; similarly x occurs *non-applied* iff t contains a subterm $t_o(t_1,\ldots,t_r)$ s.t. $x \equiv t_j$ for some $j\in[r]$. We assume that the reader is familiar with the notion of substitution of applicative terms, which is denoted as in the special case of trees.
If *type* $t = \tau\in D^n(S)$, than n = *level* t .

1 . RECURSIVE AUTOMATA

With the diversity of tree-automata model coming up in the recent literature, we view it as mandatory to use a common notational framework, which allows for easy comparison and clear exposition of the essential concepts. We hope that this paper contributes in advocating the framework of *recursive automata* as such a tool, which is a streamlined generalization of Scott's approach [Sco] to the inherently recursive situation on trees. This framework was developped by Engelfriet in as yet unpublished notes [Eng 2]; in order to make this paper self-contained we review the basic concepts as they are needed.

Recall from [Sco], that an *automaton-datatype* D consists of

- a *domain* (or: set of *storage configurations*) C
- a set of *initial* storage configurations $C_o \subseteq C$
- a set of *predicatesymbols* *Pred*
- a set of *instructionsymbols* *Inst*
- a meaning function $[\![\ ]\!]$ which assigns to any
  - predicatesymbol π a predicate $C \to \{true,false\}$
  - instructionsymbol ι a partial function $C \dashrightarrow C$

The following "standard" datatype will be used throughout the paper.

1.1 Example [Eng 2]

Let Σ be an S-sorted alphabet (of *input symbols*).
Define the datatype TREE by

- $C = C_o = T_\Sigma$
- *Pred* $= \{root = \sigma? \ | \sigma\in\Sigma\}$
- *Inst* $= \{select\ (i,s) | i\in\omega, s\in S\}$
- $[\![root=\sigma?]\!](t) =$ *true* iff $t = \sigma(t_1,\ldots,t_r)$ for some $t_j\in T$
- $[\![select\ (i,s)]\!](t) = \begin{cases} t_i & \text{iff } t = \sigma(t_1,\ldots,t_r), \\ & i\in[r],\ type\ t_i = s, \sigma\in\Sigma \\ \uparrow & \text{otherwise} \end{cases}$ □

A more storage oriented view of trees is obtained by generalizing the notion of pushdown to trees.

1.2 Example

Let Γ be an S-sorted alphabet (of *pushdown symbols*) with a designated element γ_o of *arity* 0 . Define the datatype TREEPD by

- $C = T_\Gamma$, $C_o = \{\gamma_o\}$
- $Pred = \{top = \gamma ? \mid \gamma \in \Gamma\}$
- $Inst = \{push\ t \mid t \in T_\Gamma(\omega \times S)\}$
- $top = \gamma ?$ is just a different notation for asking about the label of the root, which is the top of the pushdown
- $[\![push\ t]\!](pd) = t[(i,s)/[\![select\ (i,s)]\!](pd)]_{(i,s)\in\omega\times S}$

□

The semantics (or even pragmatics) of *push* t is easy to understand by viewing strings as monadic trees: pushing a string w then corresponds to substituting the current pd for the (only) parameter (1,s) at the "leave" (i.e. the right end) of w .

Given automaton datatypes D , D' , we can canonically define the *parallel product* $D \times D'$ of D and D' by executing the tests (or operations) of D and D' in parallel (notion: $\pi \times \pi'$, $\iota \times \iota'$). The datatype obtained from D by adding an *identity instruction* will be denoted D_{id} .

Let us now describe the "programming language" used to control the operations on such datatypes. In the current context of tree-language theory these datatypes will be used as auxiliary storage to top-down tree automata (viewed as generators).

1.3 Definition [Eng 2]

(1) A *recursive tree automaton over* D is a set of (possibly nondeterministic) recursive procedures with one parameter of type D , which compute trees. More formally, such an automaton consists of

- a set of *states* (or:procedurenames) Q
- a set of *initial states* $Q_o \subseteq Q$
- a (ranked) *output alphabet* Δ (of *terminal symbols*)
- a finite set of *rules* (*transitions*, *productions*)
  $R \subseteq Q[Pred^*] \times T_\Delta(Q[Inst])$
  written $q(\pi) \to t$.

(2) The class of recursive tree automata over D will be denoted RTA(D).

□

Note that over the trivial datatype 1 (with a single test which is true on the only element ofthe domain and an identity instruction) this definition specializes to regular tree grammars (with terminal alphabet Δ and nonterminals Q). Hence rewriting generalizes from regular rewriting by allowing control and manipulation of the underlying datatype D .

1.4 Definition [Eng 2]

Let $A \in RTA(D)$.

(1) Consider *sentential forms* $t_1, t_2 \in T_\Delta(Q[C])$.
The *move relation* is defined by $t_1 \Rightarrow_A t_2$ iff there is a rule $q(\pi) \to t$ in R s.t. t_2 is obtained from t_1 by replacing an occurrence of $q(c)$ in t_1 by $t[\iota/[\![\iota]\!](c)]_{\iota\in Inst}$ and for all tests π_i (with $\pi = \pi_1 \ldots \pi_n$) we have $[\![\pi_i]\!](c) = true$.

(2) A *generates* (*accepts*) the tree language
$L(A) = \{t \in T_\Delta \mid \exists c_o \in C_o\ \exists q_o \in Q_o\ q_o(c_o) \overset{*}{\Rightarrow}_A t\}$. The class of languages generated by RTA's over D will be denoted $\mathcal{L}_{RTA}(D)$ (or simply $\mathcal{L}(D)$ if no confusion arises).

□

To appreciate this definition as a unifying framework (and not as yet another model) we urge the reader to consult [Eng 2]. In this paper this framework will essentially be applied to generalizations of context-free rewriting and pushdowns. As a first example in the direction, note that RTA(TREEPD) coincides with the *creative dendogrammars* of Rounds [Rou], i.e. with state-controlled context-free tree grammars (recall that the finite-state control can be coded into the tree-pd). Thus TREEPD "is the datatype of" context-free tree languages: it captures the substitution power inherent in context-free tree rewriting.

We now give a generalization of the datatype TREEPD to *higher levels* (c.f. [Da]).

The generalization is obtained by working with *applicative terms* rather than with trees.

1.5 Example

Let X be a D*(S)-sorted alphabet (of *nonterminals*) with maximal *level* n , and let S = {s} .
Define the datatype n-TREE by

- $C = C_o = T^S_X$
- *Pred* = {*top* = x ? | x ∈ X}
- *Inst* = {*select* (j,τ) | j ∈ ω , τ ∈ D*(S)}
- Note that any n-TREE t can be uniquely written in the form $t \equiv x\,(\overrightarrow{m - t})\ldots(\overrightarrow{o - t})$ with x of *level* m , $\overrightarrow{k - t}$ a vector of applicative terms of *level* k .
  - $[\![top = x\ ?]\!](t) = true$ iff $t \equiv x\,(\overrightarrow{m - t})\ldots(\overrightarrow{o - t})$ for some $\overrightarrow{k - t}$, k ∈ {o,...,m}
  - $[\![select\ (j,\tau)]\!]\ (t) = \begin{cases} (k - t)_j & \text{iff } t \equiv x(\overrightarrow{m - t})\ldots(\overrightarrow{o - t}) \\ & \text{with } type\ (k - t)_j = \tau \in D^k(S) \\ \uparrow & \text{otherwise} \end{cases}$

Again we are interested in the storage-oriented view, which will generalize the operation of pushing to applicative terms.

1.6 Example

Let X be as in 1.5, and let X contain a *level* o element x_o .
Define the datatype n-TREEPD by

- $C = T^S_X$, $C_o = \{x_o\}$
- *Pred* = {*top* = x ? | x ∈ X}
- *Inst* = $\{push\ t \mid t \in T^S_{X \cup \omega \times D^*(S)}\}$
  (where ω×D*(S) is viewed as the obvious D*(S)-set)
- • the semantics of *top* = x ? is carried over from n-TREE .
  • $[\![push\ t]\!](pd) = t[(j,\tau)/[\![select(j,\tau)]\!](pd)]_{(j,\tau) \in \omega \times D^*(S)}$

□

Note that $\mathcal{L}$ (n-TREEPD) coincides with the *level-n stack automata* in [DaGu], i.e. with state controled *level-n tree grammars*. It has been shown in [DaGu] that one can restrict oneself to a single state, hence $\mathcal{L}$ (n-TREEPD) = n - $\mathcal{L}_{OI}$, the class of *level-n tree languages* [Da]. Thus n-TREEPD captures exactly the substitution power inherent in higher-level tree grammars.

2. RECURSIVE TREE TRANSDUCERS

In this chapter we introduce *recursivetree transducers* over a datatype D , which are essentially top-down tree transducers with auxiliary storage D . The motivation of studying this model is twofold:

- we will apply the results of this section in chapter to a particular datatype obtained by forming iterated pushdowns and show - in an abstract setting - how recursive tree transducers over this datatype can be used to model the run-time behaviour of ALGOL-68 procedures with finite mode
- recursive tree transducers over D are closely related to alternating D-automata.

Following the definition and some examples we will prove a decomposition result which shows that the top-down transducer underlying the recursive tree automata can be separated from the datastructure by doing first a top-down translation and then checking the resulting tree (of instructions) using a D-automaton as acceptor. The output of the original transducer is than recovered by a linear homomorphism. This technique is inspired from [Eng 1], where it was illustrated in connection with alternating pda's.

Consider the datatype TREE×D for a given datatype D , then a recursive tree automaton over TREE×D can be viewed as a top-down tree transducer with auxiliary storage D by keeping track of the initial tree in TREE . This motivates the following definition.

2.1 Definition

(1) We denote the class of *recursive tree transducers over* D by
RTT(D) := RTA(TREE×D) .

(2) For $M \in RTT(D)$ with input alphabet Σ and output alphabet Δ we define the *tree-translation induced by* M to be
$T(M) = \{(t,t') \in T_\Sigma \times T_\Delta \mid \exists c_o \in C_o \ \exists q_o \in Q_o \ q_o(t,c_o) \overset{*}{\Rightarrow}_M t'\}$.

(3) The class of recursive tree translations over D is denoted by $\mathfrak{T}_{RTT}(D)$. We omit the subscript RTT if no confusion arises .

□

2.2 Example

(1) RTT(1) coincides with the class of *top-down tree transducers* (and will henceforth be denoted T) . In specifying T's we stick to the usual notion. The class of top-down tree translations will be denoted $\mathfrak{T}$.

(2) RTT(TREEPD) coincides with the class of level-1 stack transducers of [DaGu] .

□

Note that ε-moves can be described by working with $TREE_{id}$ rather than with TREE . The corresponding notations will be indexed by ε .

To state the decomposition result in its sharpest form, we introduce the following notation.

Notation

(1) L_tHOM denotes the class of linear non-deleting tree-homomorphisms (for this terminology c.f [Eng 4]). Note that such a homomorphism may only erase monadic symbols.

(2) $\mathfrak{T}_{\leq 1}$ (resp. $\mathfrak{L}_{\leq 1}$) denotes the class of translations of RTT's (resp. languages of RTA's) which output at most one symbol at a time. Similarly we use the subscript = 1 .

□

The following normalform result is implicit in many applications.

2.3 Lemma

If D contains an identity instruction then $\mathfrak{L}_{\leq 1}(D)$ is closed under L_tHOM and $\mathfrak{L}_{\leq 1}(D) = \mathfrak{L}(D)$.

Proof:

Let $h \in L_t HOM, A \in RTA_{\leq 1}(D)$. Define $A_h \in RTA(D)$ by applying h (canonically extended to $Q[Inst]$ as identity) to the right-hand-sides of the rules of A . Since h is linear and nondeleting

$$q_o(c_o) \overset{*}{\Rightarrow}_A t \text{ iff } q_o(c_o) \overset{*}{\Rightarrow}_{A_h} h(t)$$

for all $t \in T_\Delta(Q[C])$, hence $h(L(A)) = L(A_h)$.
Now $A' \in RTA_{\leq 1}(D)$ equivalent to A_h is constructed by memorizing the occurrences of a right-hand-side in the state and keeping the storage-

configuration unchanged until reaching the instruction-leaves of the simulated right-hand-side (see [Gue] for the construction with D=TREEPD) . □

For the rest of this section we will assume that all derivations are leftmost. Moreover, for the automaton datatype TREE we will abbreviate

root = σ ?	for $\sigma\in\Sigma_k$	into $\sigma(x_1,\ldots,x_k)$
select (k,s)	into	x_k
id	into	x

We now proceed to prove the decomposition result in a series of steps.

2.4 Theorem

(1) $\mathcal{T}_\varepsilon\ (D) \subseteq \mathcal{T}_{\varepsilon,=1} \circ \mathcal{L}_{=1}\ (D) \circ L_t HOM$

(2) $\mathcal{T}\ (D) \subseteq \mathcal{T}_{=1} \circ \mathcal{L}_{=1}\ (D) \circ L_t HOM$

Proof: Let $S = (Q,Q_\circ,\Delta,R) \in RTT_\varepsilon\ (D)$.
Its rules are of one of the following forms

(i) $r : q(\sigma(x_1,..,x_n),\pi) \to t_r\ (q_1(x_{j_1},\iota_1),\ldots,q_n(x_{j_n},\iota_n))$
(ii) $r' : q(x,\pi) \to t_{r'}(q_1(x,\iota_1),\ldots,q_n(x,\iota_n))$
(iii) $r'' : q(x,\pi) \to q'\ (x,\iota)$
(iv) $r''' : q(\sigma(x_1,\ldots,x_n),\pi) \to q'(x_j,\iota)$

Note that type (iii) rules are ε-rules or ε-moves and type (ii) rules are ε-moves with output.

S can be decomposed into the following recursive tree automata:

$M = (Q,Q_\circ,\Delta_R,R_M) \in T_{\varepsilon,=1}$

where $\Delta_R = \{\rho_r / r \in R\}$ is a ranked alphabet disjoint from existing alphabets; the rank of ρ_r is the number of occurrences of elements in $Q[Inst]$ in the right hand side of rule r .

R_M contains the rules: for r (resp. r',r'',r''') a type (i) (resp. (ii),(iii), (iv)) rule in R :

$r_M : q\ (\sigma(x_1,\ldots,x_n)) \to \rho_r(q_1(x_{j_1}),\ldots,q_n(x_{j_n}))$
$r'_M : q\ (x) \to \rho_{r'}(q_1(x),\ldots,q_n\ (x))$
$r''_M : q\ (x) \to \rho_{r''}(q'(x))$
$r'''_M : q(\sigma(x_1,\ldots,x_n)) \to \rho_{r'''}\ (q'(x_j))$

Notice that M is a "depth preserving" transducer, i.e. $M \in T_{\leq 1}$.

Let now $A = (\{\bar q\},\{\bar q\},\Delta_R,R_A) \in RTA_{=1}\ (D)$ be defined as follows. A has a single state $\bar q$, and its rules are, for each type (i), (ii) or (iii) rule of S , defined respectively by:

$r_A : \bar q(\pi) \to \rho_r(\bar q(\iota_1),\ldots,\bar q(\iota_n))$ for a type (i) or type (ii) rule r in R
$r''_A : \bar q(\pi) \to \rho_{r''}(\bar q(\iota))$ for a type (iii) or type (iv) rule r'' in R

Then $T(S) \subseteq T_\Sigma \times T_\Delta$, $T(M) \subseteq T_\Sigma \times T_{\Delta_R}$ and $L(A) \subset T_{\Delta_R}$.

Define finally the linear homomorphism $h : T_{\Delta_R}(Y) \to T_\Delta(Y)$, where Y is some set of parameters, by $h(y) = y$ for y in Y
$h(\rho_r(t_1,\ldots,t_n)) = t_r(h(t_1),\ldots,h(t_n))$ if r is a type (i) or (ii) rule

$h(\rho_r(t)) = h(t)$ if r is a type (iii) or (iv) rule.
Type (iii) and (iv) rules correspond to erasing rules for h .

Notice that when S is real time (i.e. has neither ε-moves nor output on ε-moves,i.e. has only type (i) rules), then M is a usual transducer without ε-moves or ε-outputs, i.e. $M \in T_{=1}$, A is also realtime, hence $A \in RTA_{=1}\ (D)$. This shows that it suffices to prove part (1) of the theorem, then part (2) will follow.

Notation: a sequence of derivations applying rules $r^1\ldots r^n$ will be denoted by $r^1 \ldots r^n$. Then if $q_\circ(t) \overset{r^1_M \ldots r^n_M}{\Rightarrow} t_1$, by definition of M , $r^1_M\ldots r^n_M$ will be the denotation in polish prefix notation of t_1 ; more precisely let φ_M be the morphism $R_M^* \to \Delta_R^*$ defined $\varphi_M(r^i_M) = \rho_{r_i}$, then $\varphi_M(r^1_M\ldots r^n_M) = \rho_{r_1}\ldots\rho_{r_n}$ is the polish notation for t_1 . Similarly, if $\bar q(c_\circ) \overset{r^1_A \ldots r^p_A}{\Rightarrow} t_1$, letting $\varphi_A : R_A^* \to \Delta_R$ be defined by $\varphi_A(r^i_A) = \rho_{r_i}$, $\varphi_A(r^1_A\ldots r^p_A) = \rho_{r_1}\ldots\rho_{r_p}$ is the polish notation for t_1 .
(1) is then a consequence of the following lemmata:

Lemma 1: If $q(c_\circ,t) \overset{r^1\ldots r^n}{\Rightarrow} t'$ for $c_\circ \in C_\circ$, $q_\circ \in Q_\circ$ $t \in T_\Sigma$, $t' \in T_\Delta$ and $r^1,\ldots,r^n \in R$, then : $\exists\ t_1$ in t_{Δ_R} such that $q_\circ(t) \overset{r^1_M \ldots r^n_M}{\Rightarrow} t_1$, $\bar q_\circ(c_\circ) \overset{r^1_A\ldots r^n_A}{\Rightarrow} t_1$, $h(t_1) = t'$, and $\varphi(r^1\ldots r^n) = \rho_{r_1}\ldots\rho_{r_n}$ is the polish prefix notation for t_1 .

Proof: any computation sequence $c = r^1\ldots r^n$ of S can obviously be projected into the corresponding computation sequences $r^1_M\ldots r^n_M$ of M and $r^1_A\ldots r^n_A$ of A ; if moreover c is terminating, $h(t_1) = t'$. □

Lemma 2: If for $q_o \in Q_o$, $c_o \in C_o$, $t \in T_\Sigma$ $t'_1 \in T_{\Delta_R}(Y)$,

$q_o(t) \overset{r_M^1 \ldots r_M^n}{\Rightarrow} t'_1(q_1(t_1),\ldots,q_n(t_n))$ and $\bar{q}(c_o) \overset{r_A^{j_1} \ldots r_A^{j_p}}{\Rightarrow} t'_1(\bar{q}(c_1),\ldots,\bar{q}(c_n))$

then: $p = n$, $j_k = k$ for $k = 1,\ldots,p$ and

$q_o(c_o,t) \overset{r^1 \ldots r^n}{\Rightarrow} h(t'_1)(q_1(t_1,c_1),\ldots,q_n(t_n,c_n))$. If moreover $t'_1 \in T_{\Delta_R}$, then $q_o(c_o,t) \overset{r_1 \ldots r_n}{\Rightarrow} h(t'_1)$ is a terminating computation sequence of S .

Proof: By the construction of M and A ; let $\varphi_A(r_A^{j_1},\ldots,r_A^{j_p}) = \rho_{r_{j_1}} \ldots \rho_{r_{j_p}}$ and $\varphi_M(r_M^1,\ldots,r_M^n) = \rho_{r_1} \ldots \rho_{r_n}$ then both $\rho_{r_1} \ldots \rho_{r_n}$ and $\rho_{r_{j_1}} \ldots \rho_{r_{j_p}}$ are the encodings of t'_1 in polish prefix notation hence they are equal .

□

The reverse inclusions will be proved separately.

2.5 Theorem

$$\mathcal{T}(D) \supseteq \mathcal{T}_{=1} \circ \mathcal{L}_{=1}(D) \circ L_t HOM$$

Proof: Let $M = (Q,Q_o,\Sigma,\Sigma',R_M) \in T_{=1}$, $A = (\bar{Q},\bar{Q}_o,\Sigma',R_A) \in RTA_{=1}(D)$, and the tree-homomorphism $h : T_{\Sigma'}(Y) \to T_\Delta(Y)$ be given by $h(y) = y$, $h(\sigma(t_1,\ldots,t_n)) = t_\sigma(h(t_1),\ldots,h(t_n))$ for σ in Σ'_n , $t_\sigma \in T_\Delta(Y_n)$.

Let $Tr = T(M) \circ L(A) \circ h$, thus $(t,t') \in Tr$ iff
$\exists q_o \in Q_o, \bar{q}_o \in \bar{Q}_o \ \exists c_o \in C_o \ \exists t_1 \in T_{\Sigma'}$, $q_o(t) \overset{*}{\Rightarrow}_M t_1 \wedge \bar{q}_o(c_o) \overset{*}{\Rightarrow}_A t_1 \wedge h(t_1) = t'$.
We show that $Tr = T(S)$, where $S = (Q\times\bar{Q},Q_o\times\bar{Q}_o,\Delta,R) \in RTT(D)$ is given by to each rule $r_M : q(t(x_1,\ldots,x_p)) \to \sigma(q_1(x_{j_1}),\ldots,q_n(x_{j_n}))$ of M and rule $r_A : \bar{q}(\pi) \to \sigma(\bar{q}_1(\iota_1),\ldots,\bar{q}_n(\iota_n))$ of A having the same right-hand side root σ as r_M , we associate the rule
$r_M \times r_A : \langle q,\bar{q}\rangle(\tau(x_1,\ldots,x_p),\pi) \to t_\sigma(\langle q_1,\bar{q}_1\rangle(x_{j_1},\iota_1),\ldots,\langle q_n,\bar{q}_n\rangle(x_{j_n},\iota_n))$.
$T(S) \subset Tr$: project any computation sequence of S onto the corresponding computation sequences of M and A . Conversely, $Tr \subset T(S)$ is also obvious since both M and A are real-time. If moreover we replace $L_t HOM$ by $L_s HOM$, i.e. if the homomorphism h is non erasing, then S is also real-time on output, i.e. $S \in RTT_{\geq 1}(D)$.

□

The same proof applies if we replace $T_{=1}$ by $T_{\varepsilon,=1}$, namely if the transducer always outputs a symbol in any move; we then get
$\mathcal{T}_\varepsilon(D) \supseteq \mathcal{T}_{\varepsilon,=1} \circ \mathcal{L}_{=1} \circ L_t HOM$.

2.6 Theorem

If D contains an identity instruction, then (3) $\mathcal{T}_\varepsilon(D) \supseteq \mathcal{T}_\varepsilon \circ \mathcal{L}(D) \circ L_t HOM$

Proof: by lemma 2.3 we can replace $\mathcal{L}(D) \circ L_t HOM$ by $\mathcal{L}_{\leq 1}(D)$. Note that the automaton datatype corresponding to $\mathcal{T}_\varepsilon$ also contains an identity instruction, hence (3) is equivalent to (4) $\mathcal{T}_\varepsilon(D) \supseteq \mathcal{T}_{\varepsilon,\leq 1} \circ \mathcal{L}_{\leq 1}(D)$.

Let now $M = (Q,Q_o,\Sigma,\Delta,R_M) \in T_{\varepsilon,\leq 1}$, and $A = (\bar{Q},\bar{Q}_o,\Delta,R_A) \in RTA_{\leq 1}(D)$. The rules of M are of the form

(i) $q(\sigma(x_1,\ldots,x_p)) \to t(q_1(x_{j_1}),\ldots,q_n(x_{j_n}))$

(ii) $q(x) \to t(q_1(x),\ldots,q_n(x))$

where either $t \equiv \delta(y_1,\ldots,y_n)$ (for $\delta\in\Delta$) or $t \equiv y_i$ (i.e. a move with no output). Similarly, the rules of A are at the form

($\bar{\text{i}}$) $\bar{q}(\pi) \to \delta(\bar{q}_1(\iota_1),\ldots,\bar{q}_n(\iota_n))$
($\bar{\text{ii}}$) $\bar{q}(\pi) \to \bar{q}'(\iota)$ (ε-move) .

Let then $S = (Q\times\bar{Q},Q_o\times\bar{Q}_o,\Delta,R) \in RTT_\varepsilon(D)$ be defined by the following set of rules:

- for each type (i) rule r_M of M and type ($\bar{\text{i}}$) rule r_A of A having the same right-hand-side root δ we associate the rule $r_M \times r_A$:
  $\langle q,\bar{q}\rangle(\sigma(x_1,\ldots,x_p),\pi) \to \delta(\langle q_1,\bar{q}_1\rangle(x_{j_1},\iota_1),\ldots,\langle q_n,\bar{q}_n\rangle(x_{j_n},\iota_n))$

- for each type (ii) and ($\bar{\text{i}}$) rules r_M and r_A having the same right-hand-sides root δ corresponds the rule $r_M \times r_A$:
  $\langle q,\bar{q}\rangle(x,\pi) \to \delta(\langle q_1,\bar{q}_1\rangle(x,\iota_1),\ldots,\langle q_n,\bar{q}_n\rangle(x,\iota_n))$

- to each type (i) rule r_M of M with right-hand-side root $q_i(x_i)$ correspond the rules $r_M \times id_{\bar{q}} : \langle q,\bar{q}\rangle(\sigma(x_1,\ldots,x_p),\varepsilon) \to \langle q_i,\bar{q}\rangle(x_i,id)$ for each $\bar{q} \in \bar{Q}$ (and $\varepsilon \in Pred^*$)

- to each type (ii) rule r_M of M : $q(x) \to q_i(x)$ correspond the rules $r_M \times id_{\bar{q}}$: $\langle q,\bar{q}\rangle(x,\varepsilon) \to \langle q_i,\bar{q}\rangle(x,id)$ for each $\bar{q} \in \bar{Q}$

- to each type ($\overline{\text{ii}}$) rule r_A of A correspond rules $id_q \times r_A$
  $\langle q,\bar{q}\rangle(x,\pi) \to \langle q,\bar{q}'\rangle(x,\iota')$

Obviously, $T(S)$ is included in the translation
$Tr = \{(t,t') \in T_\Sigma \times T_\Delta \mid \exists q_o \in Q_o, \bar{q}_o \in \bar{Q}_o \ \exists c_o \in C_o \ q_o(t) \overset{*}{\Rightarrow}_M t' \wedge \bar{q}_o(c_o) \overset{*}{\Rightarrow}_A t'\}$,
Since any computation sequence of S can be projected into computation sequences of M and A .

Conversely, $Tr \subseteq T(S)$ stems from the following lemma.

Lemma

Let $t_1 \in T_\Delta(Y)$, $t_o \in T_\Sigma$, $c_o \in C_o$ and assume
$q_o(t_o) \overset{r_M^1 \ldots r_M^n}{\Rightarrow} t_1(\overrightarrow{q(t)})$, $\bar{q}_o(c_o) \overset{r_A^1 \ldots r_A^p}{\Rightarrow} t_1(\overrightarrow{\bar{q}(c)})$,

then there exists a computation sequence $s = r_S^1 \ldots r_S^q$ of S s.t. $\langle q_o, \bar{q}_o\rangle \overset{s}{\Rightarrow} t_1(\overrightarrow{\langle q,\bar{q}\rangle(t,c)})$.

The proof of this lemma proceeds by induction on (n,p) .

- If $n = p = o$ the result is obviously true.
- Suppose that the result holds for $(n',p') \leq (n,p)$, and consider a computation sequence $q_o(t_o) \overset{n+1}{\Rightarrow}_M t_1(\overrightarrow{q(t)})$ (the case where $q_o(c_o) \overset{p+1}{\Rightarrow}_A t_1(\overrightarrow{\bar{q}(c)})$ is similar and omitted). Then *either*
  - $q_o(t_o) \overset{n'}{\Rightarrow}_M t_1(\overrightarrow{q'(t)}) \overset{s_M}{\Rightarrow}_M t_1(\overrightarrow{q(t)})$ for some $n' \leq n$, where $s_M \equiv r_M^{n'+1} \ldots r_M^{n+1}$ is a sequence of ε-moves without output, hence by the induction hypothesis $\langle q,\bar{q}_o\rangle(t_o,c_o) \overset{*}{\Rightarrow}_S t_1(\overrightarrow{\langle q',\bar{q}\rangle(t,c)}) \equiv \Theta$ and for $s := r_M^{n'+1} \times id_{\bar{q}_{n'+1}} \ldots r_M^{n+1} \times id_{\bar{q}_{n+1}}$, where $\bar{q}_j$ is the state labelling the *occurrence* in $t_1(\overrightarrow{\bar{q}(c)})$ to which rule r_M^j is applied (for $j \in \{n'+1,\ldots,n+1\}$) , we have $\Theta \overset{s}{\Rightarrow}_S t_1(\overrightarrow{\langle q,\bar{q}\rangle(t,c)})$; *or*
  - $q_o(t_o) \overset{n}{\Rightarrow}_M t_1'(q''(t''))$ for some prefix t_1' of t_1 and $q_i''(t'') \overset{r_M^{n+1}}{\Rightarrow} \overrightarrow{\delta(q'(t'))}$ with $t_1(\overrightarrow{q(t)}) \equiv t_1'(q_1''(t_1''),\ldots,\delta(\overrightarrow{q'(t')}),\ldots,q_n''(t_n''))$. All the derivations being leftmost, there is some $p' \leq p$ with $\bar{q}_o(c_o) \overset{p'-1}{\Rightarrow}_M t_1'(\overrightarrow{\bar{q}''(c'')}) \overset{r_M^{p'}}{\Rightarrow} t_1(\overrightarrow{\bar{q}'(c')})$. Then, by the induction hypothesis for $(n,p'-1)$, $\langle q_o,\bar{q}_o\rangle(t_o,c_o) \overset{*}{\Rightarrow}_S t_1'(\overrightarrow{\langle q'',\bar{q}''\rangle(t'',c'')}) \equiv \Theta$. By the definition of the rules of S $\Theta \; r_M^{n+1} \times r_A^{p'} \overset{}{\Rightarrow} t_1(\overrightarrow{\langle q,\bar{q}'\rangle(t,c')}) \equiv \Theta'$, and by the same reasoning as above, for $s := id_{q_{p'+1}} \times r_A^{p'+1} \ldots id_{q_p} \times r_A^p$, $\Theta' \overset{s}{\Rightarrow} (\overrightarrow{\langle q,\bar{q}\rangle(t,c)})$; *or*
  - $q_o(t_o) \overset{n}{\Rightarrow}_M t_1(\ldots,q'(t'),\ldots) \overset{r_M^{n+1}}{\Rightarrow} t_1(\ldots,q(t),\ldots)$ where $r_M^{n+1} \equiv q'(\sigma(x_1,\ldots,x_n)) \to q(x_i)$ and $t' \equiv \sigma(t_1',\ldots,t_n')$ and $t \equiv t_i'$. Then by the induction hypothesis for (n,p) $\langle q_o,\bar{q}_o\rangle(t_o,c_o) \overset{*}{\Rightarrow}_S t_1(\ldots,\langle q',\bar{q}\rangle(t',c),\ldots) \equiv \Theta$ and $\Theta \; r_M^{n+1} \times id_{\bar{q}} \overset{}{\Rightarrow} t_1(\ldots,\langle q,\bar{q}\rangle(t,c),\ldots)$. This finishes the proof of the lemma and thus of the theorem.

□

Summarizing the above theorems gives us the promised decomposition result.

2.6 Corollary

(1) $\mathfrak{T}(D) = \mathfrak{T}_{=1} \circ \mathcal{L}_{=1}(D) \circ L_t HOM$

(2) if D contains an identity instruction then $\mathfrak{T}_\varepsilon(D) = \mathfrak{T}_\varepsilon \circ \mathcal{L}(D) \circ L_t HOM$

(3) $\mathfrak{T}_\varepsilon(D) = \mathfrak{T}_{\varepsilon,=1} \circ \mathcal{L}_{=1}(D) \circ L_t HOM$

□

As an immediate corollary it follows, that in many cases acceptor-equivalence implies transducer equivalence.

2.7 Corollary

(1) D_1, D_2 contain an identity instruction. Then $\mathcal{L}(D_1) = \mathcal{L}(D_2) \Rightarrow \mathfrak{T}(D_1) = \mathfrak{T}(D_2)$ and $\mathfrak{T}_\varepsilon(D_1) = \mathfrak{T}_\varepsilon(D_2)$

(2) $\mathcal{L}_{=1}(D_1) = \mathcal{L}_{=1}(D_2) \Rightarrow \mathfrak{T}(D_1) = \mathfrak{T}(D_2)$ and $\mathfrak{T}_\varepsilon(D_1) = \mathfrak{T}_\varepsilon(D_2)$

□

In particular, the top-down auxiliary D-automaton are so closely related to the D-acceptors, that strictness of a transducer hierarchy (over $D_1,\ldots,D_n,D_{n+1},\ldots$) implies this result for the acceptor hierarchy. The reverse implication holds (only?) for transducers with ε-moves : if $L \in \mathcal{L}(D_1)$, then $\{a\} \times L \in \mathfrak{T}_\varepsilon(D_1)$ for any constant symbol a (since the corresponding RTT_ε can simulate the RTA independent of the input). Moreover it can be shown, that moves of the kind $q(root = \sigma\,?\,\times\,\pi) \to t$ (for $\sigma \neq a$) "don't pay" for the equivalent RTT over D_2 unless *all* instructions in t leave the input-tree unchanged. Thus an RTA over D_2 generating L can be constructed from this RTT by forgetting the input component in all such rules (and not simulating any other rule).

Now recall from [Eng 1], that recursive D-transducers are *very close* to *alternating* D-*automata*: the parallel checking required by alternating automata correspond to copying the input by the top-down tree transducer, since a tree is in its domain iff *all* copies are accepted. Note that the transducer will have to make ε-moves of the alternating D-automaton. This justifies to take the following proposition as *definition* of the class $\mathcal{L}_{ALT}(D)$ of alternating D languages. (Recall that *m* means monadic input trees).

2.8 Proposition [Eng 1]

$\mathcal{L}_{ALT}(D) = dom(\mathfrak{T}_{\varepsilon,m}(D))$

□

Now the following characterization of alternating D-languages is immediate from the decomposition theorem (note that the $L_t HOM$ is irrelevant with respect to domains), thus giving a detailed proof of this result.

2.9 Corollary [Eng 1]

(1) if D contains an identity instruction, then
$\mathcal{L}_{ALT}(D) = T^{-1}_{\varepsilon,m}(\mathcal{L}(D))$

(2) $\mathcal{L}_{ALT}(D) = T^{-1}_{\varepsilon,m}(\mathcal{L}_{=1}(D))$.

3. A RETURN-ADDRESS IMPLEMENTATION OF HIGHER TYPE PROCEDURES

In this section we will extend the classical return-address implementation of (parameterless) procedures on pda's to procedures with procedure parameters of finite mode (in the sense of ALGOL68). In this paper we will only give the construction on the level of abstraction of implementing level-n tree languages; the run-time behaviour of a subset of ALGOL68 will be modelled explicitly in a forthcoming paper.

The technique used for the implementation is generalized from the case of procedures with base-type parameters, which on the abstract level correspond to (OI-) context-free tree languages. For this case the implementation was given in [Gue, theorem 3], see also [Gal]. The basic idea is to store the *occurrence* of a recursive call on the pushdown; when reaching a parameter in simulating a right-hand-side, this occurrence is popped and -updated by the argument position - used as new state, indicating that the simulation is proceeding at the occurrence "following" the recursive call.

To extend this idea to higher levels, we will generalize the notion of occurrences to applicative terms. The canonical generalization of the pda to higher levels is the *level-n pda*, obtained by taking pushdown lists of pairs <pushdown-symbol, level-(n - 1) pda> . This model was first mentioned in [Gre] and then used in [Mas] to obtain an automata-theoretic characterization of generalized indexed languages. We recall here the formal definition used in [DaGo]. For a motivation of this defintion we refer to this paper.

3.1 Definition

Let Γ be a set of *pushdown-symbols* and $n \in \omega$. The datatype n-PD is given as follows.

- To define the set of configurations we define inductively an [n+1]-set n-*pds* (Γ). Intuitively pd $\in$ n-*pds* $(\Gamma)^j$ describes a level-n pd-store, where only the levels $\geq j$ are specified.

  Let n-*pds*$(\Gamma)^{n+1} = \{\varepsilon\}$, n-*pds*$(\Gamma)^j = (\Gamma[\text{n-}pds(\Gamma)^{j+1}])^*$.
  Then $C = \text{n-}pds(\Gamma)^\circ$, $C_\circ = \{\varepsilon\}$. Note that each pds $\in C$ has a unique representation $A_\circ [A_1[\ldots A_m\ m\text{-}rest]\ldots 1\text{-}rest]\ o\text{-}rest]$
  with $A_j \in \Gamma$, $j\text{-}rest \in \text{n-}pds(\Gamma)^j$.

- *Pred* = {j-*top* = A ? | A ∈ Γ , o≦j<n}
- *Inst* = {j-*pop* | o≦j<n}∪{j-*push* γ | γ ∈ Γ* , o≦j<n}
- • ⟦ j-*top* = A ? ⟧ (pds) = *true* iff
  pds ≡ A_o [... A_m m-*rest*]...]o-*rest* and A ≡ A_j
  • ⟦ j-*pop* ⟧ (pds) =

$$\begin{cases} A_o\,[A_1\,[\ldots j\text{-}rest\ldots]1\text{-}rest]o\text{-}rest \\ \quad \text{iff pds} \equiv A_o\,[\ldots[A_j\ \ldots[A_m\ m\text{-}rest]\ldots\ j\text{-}rest]\ldots]o\text{-}rest \\ \quad\quad \text{and}\ \ j \leqq m \\ \uparrow\ \text{otherwise} \end{cases}$$

• ⟦ j-*push* γ ⟧ (pds) = (for γ = $\gamma_1 \ldots \gamma_r$)

$$\begin{cases} A_o\,[\ldots\gamma_1[j+1\text{-}flag]\ldots\gamma_r[j+1\text{-}flag]A_j[j+1\text{-}flag]j\text{-}rest\ldots]o\text{-}rest \\ \quad \text{iff pds} \equiv A_o\,[\ldots[A_j\ \ldots[A_m\ m\text{-}rest]\ldots\ j\text{-}rest]\ldots]o\text{-}rest \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad j+1\text{-}flag \\ \quad\quad \text{and}\ \ j \leqq m \\ \\ A_o\,[\ldots\,[A_m[\gamma_1\ldots\gamma_r]m\text{-}rest]\ldots]o\text{-}rest \\ \quad \text{iff pds} \equiv A_o\,[\ldots[A_m\ m\text{-}rest]\ldots]o\text{-}rest] \text{ and } j=m+1 \\ \uparrow\ \text{otherwise} \end{cases}$$

Note, that this datatype contains an identity instruction *id* := o-*push* ε .

□

It was shown in [DaGo] that the class of string languages accepted by n-PDA's coincides with the class of level-n OI-string languages. (We note that the implementation technique of this paper, which is essentially a generalization of the construction in [Fi] , cannot be directly adopted to the tree-case.) Engelfriet proves in [Eng 3] that *alternating* n-PDA's coincide with the class of languages accepted by deterministc Turing machines in EXP^n(LIN)-TIME and thus form a strict hierarchy. It thus follows, by considering domains (c.f. 2.8), that recursive n-PD transducers define a strict hierarchy of tree-translations with increasing level.

3.2 Corollary

$$\forall n \geq 1 \quad \mathfrak{T}_\varepsilon\ (n\text{-PD}) \subsetneq \mathfrak{T}_\varepsilon\ (n+1\text{-PD})$$

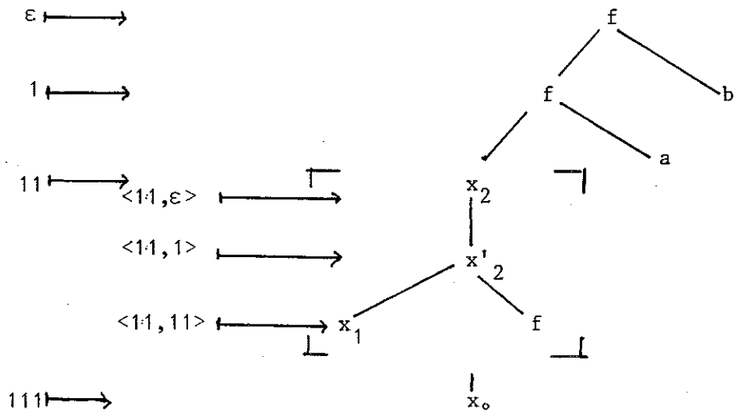
□

We now describe how the notion of occurrence is generalized to applicative terms. Recall that occurrences in a *tree* can be denoted by a string in ω* . Now note that applicative terms can be viewed as trees of trees of... the atom set. Then an occurrence of an atom in such a "complex node" of the tree is just the occurrence of the atom within the node indexed by the occurrence of that node. The following example illustrates this idea.

3.3 Example

Consider the applicative expression
$f(f(x_2(x_2'(x_1,f))(x_o),a),b) \in T_X^s$
where $\{x_o,a,b\}$, $\{x_1,f\}$, $\{x_2,x_2'\}$ are level o (resp. 1, resp. 2) -atoms in X .

The occurrences of this expression viewed as a tree are shown below



□

This motivates the following formal definition.

3.4 Definition

(1) Let $o\text{-}occ = \omega^*$, $n+1\text{-}occ = n\text{-}occ \times \omega^*$.
We denote (string) concatenation from left (i.e. with the leftmost string in an n-occurrence) (resp. from right) by $j*o$ (resp. $o*j$) .

(2) We define (by induction on applicative terms) the set $occ(t)$ of *occurrences* of t and a mapping $lab(t)$ which associates to an occurrence the corresponding subexpression of t .

- for $x \in X$, $occ(x) = \{\varepsilon\}$ and $lab(x)(\varepsilon) = x$
- for $t \equiv t_o\,(t_1,\ldots,t_r)$
  - $occ(t) = \{\varepsilon\}$
    $\cup\{<\varepsilon,o> \mid o \in occ\,(t_o),\ t_o \text{ no atom}\}$
    "lifted occurrences" of the node t_o
    $\cup\{j*o \mid o \in occ\,(t_j),\ j \in [r]\}$
    "shifted" occurrences of t_j
  - $lab(t)(\varepsilon) = t_o$
    $lab(t)(<\varepsilon,o>) = lab\,(t_o)\,(o)$
    $lab(t)(j*o) = lab\,(t_j)\,(o)$

□

We now have the tools available to give the construction. Let $A \in$ RTA (n-TREEPD). Without loss of generality (see [DaGu]) we may assume that A has a single state q . We can then write a rule (with production number j,say) $q(top = x\ ?) \to t(q(push\ t_1),\ldots,q(push\ (t_r))$ as
$j : x\downarrow \to t(t_1,\ldots,t_r)$
and define $rhs(j) \equiv t(t_1,\ldots,t_r)$. Thus the right hand sides consist of applicative terms over terminals (which thus may only occur within the initial subtree t), nonterminals in X , and parameters $<j,\tau> \in \omega \times D^*(S)$. The construction is such that it will store m-occurrences on level m .

3.5 Construction

For $A \in$ RTA (n-TREEPD) as above, we define $A' \in$ RTA (n-PD) by
- A' has pushdown alphabet
  $\Gamma = \{[j,o] \mid o \in occ\,(rhs(j))$, j a production number$\}$

- states $Q = \Gamma \dot{\cup} \{q\}$ with initial state q ; if A' is in state $[j,o]$ then it is currently simulating occurrence o in the j-th right-hand-side

- the set of rules is given by
  - *initialization*
    $q(\varepsilon) \to [j,\varepsilon]$ (*id*)
    for each production j with left-hand-side $x_o\downarrow$
    "start simulating some body of the main procedure x_o"
  - *check input*
    for $[j,o] \in Q$ with $lab\,(rhs(j))\,(o) = f \in \Delta_r$
    $[j,o](\varepsilon) \to f([j,1*o](id),\ldots,[j,r*o](id))$
  - *recursive call*
    for $[j,o] \in Q$ with $lab(rhs(j))\,(o) = x$ of *level* m
    - if x occurs in an *applied* position (c.f. basic concepts)
      $[j,o](\varepsilon) \to [k,\varepsilon](m\text{-}1\text{-}push\ [j,o])$
      for each production k with left-hand-side $x\downarrow$ "start simulating some body of x and store the return address at level $m-1$"
    - if x occurs in a non-applied position
      $[j,o](\varepsilon) \to [k,\varepsilon](id)$
      "start simulating some body of x "
  - *return*
    for $[j,o] \in Q$ with $lab\,(rhs(j))(o) = <k,\tau>$ of *level* m
    - if $<k,\tau>$ occurs in a non-applied position
      $[j,o](m\text{-}top = [r,o']?) \to [r,o'*k](m\text{-}pop)$
      for all $[r,o'] \in \Gamma$
      "simulation of this level of the right-hand-side of j is completed; return to the k-th son of the return address stored at level m "
    - if $<k,\tau>$ occurs in an applied position
      $[j,o](m\text{-}top = [r,o']?) \to [r,o'*k](m\text{-}1\text{-}push\ [j,o] \circ m\text{-}pop)$
      for all $[r,o'] \in \Gamma$
      "in addition memorize where to continue after completing level m "
  - *split*
    for $[j,o] \in Q$ with $lab\,(rhs(j))(o) = t(t_1,\ldots,t_r)$ of *level* m
    $[j,o]\ \varepsilon \to [j,<o,\varepsilon>](m\text{-}1\text{-}push\ [j,o])$

"the simulated occurrence is a complex call; split the call and start simulating t; store the return address of this level".

□

3.6 Example

Consider the level-2 grammar A (i.e. one-state RTA(2-TREEPD)) with main nonterminal x_o, level-1 nonterminal $\bar{f}$, level-2 nonterminals x_2, x_2' and terminals a,b,f, and productions

1 : $x_o\downarrow \to x_2\ (\bar{f})\ (a,b)$
2 : $x_2\downarrow \to f\ (x_2(x_2'(\langle 1,(ss,s)\rangle))(\langle 1,s\rangle,\langle 2,s\rangle),\ \langle 1,s\rangle)$
3 : $x_2\downarrow \to \langle 1,(ss,s)\rangle(\langle 1,s\rangle,\ \langle 2,s\rangle)$
4 : $x_2'\downarrow \to \langle 1,(ss,s)\rangle(\langle 1,(ss,s)\rangle(\langle 2,s\rangle,\langle 1,s\rangle),\ \langle 2,s\rangle)$
5 : $\bar{f}\downarrow \to f(\langle 1,s\rangle,\ \langle 2,s\rangle)$

A sample derivation in A is

$x_o \overset{1}{\Rightarrow} x_2(\bar{f})\ (a,b)$
$\overset{2}{\Rightarrow} f(x(x'(\bar{f}))(b,a),\ a)$
$\overset{3}{\Rightarrow} f(x'(\bar{f})\ (b,a),a)$
$\overset{4}{\Rightarrow} f(\bar{f}(\bar{f}(a,b),a),a)$
$\overset{5}{\Rightarrow} f(f(\bar{f}\ (a,b),a),a)$
$\overset{5}{\Rightarrow} f(f(f\ (a,b),a),a) \in T_\Delta$.

We give the corresponding derivation sequence for the RTA(2-PD) constructed according to 3.5 . Recall that the initial storage configuration of 2-PD is ε .

q(ε) *initialization*
⇒ $[1,\varepsilon](\varepsilon)$ *split*
⇒ $[1,\langle\varepsilon,\varepsilon\rangle]([1,\varepsilon])$ *recursive call* (applied)
⇒ $[2,\varepsilon]([1,\varepsilon][[1,\langle\varepsilon,\varepsilon\rangle]])$ *check input*
⇒ $f([2,1](pd_1),\ [2,2](pd_1))$ *return* at level o
where $pd_1 \equiv [1,\varepsilon][[1,\langle\varepsilon,\varepsilon\rangle]]$
⇒ $f([2,1](pd_1),[1,1](\varepsilon))$ *check input*
⇒ $f([2,1](pd_1),\ a)$ *split*

⇒ $f([2,\langle 1,\varepsilon\rangle]([2,1][[1,\langle\varepsilon,\varepsilon\rangle]]\ pd_1),\ a)$ *recursivecall*
⇒ $f([3,\varepsilon](2,1][[2,\langle 1,\varepsilon\rangle][1,\langle\varepsilon,\varepsilon\rangle]]\ pd_1),\ a)$ *return* at level 1 (applied)
⇒ $f([2,\langle 1,1\rangle]([3,\varepsilon][[1,\langle\varepsilon,\varepsilon\rangle]]\ pd_2\ pd_1),\ a)$
where $pd_2 \equiv [2,1][[1,\langle\varepsilon,\varepsilon\rangle]]$ *recursive call* (applied)
⇒ $f([4,\varepsilon]([3,\varepsilon][[2,\langle 1,1\rangle][1,\langle\varepsilon,\varepsilon\rangle]]pd_2\ pd_1),\ a)$ *return* at level 1 (applied)
⇒ $f([2,\langle 1,11\rangle]([4,\varepsilon][[1,\langle\varepsilon,\varepsilon\rangle]]pd_3\ pd_2\ pd_1),\ a)$
where $pd_3 \equiv [3,\varepsilon][[1,\langle\varepsilon\ \varepsilon\rangle]]$
return at level 1 (non-applied)
⇒ $f([1,\langle\varepsilon,1\rangle]([4,\varepsilon]pd_3\ pd_2\ pd_1),\ a)$ *recursive call* (non-applied)
⇒ $f([5,\varepsilon]([4,\varepsilon]pd_3\ pd_2\ pd_1),\ a)$ *check input*
⇒ $f(f([5,1]([4,\varepsilon]pd_3\ pd_2\ pd_1),[5,2]([4,\varepsilon]pd_3\ pd_2\ pd_1)),\ a)$
(2) *return* at level o (non-applied)
⇒ $f(f([4,1](pd_3\ pd_2\ pd_1),[4,2](pd_3\ pd_2\ pd_1)),\ a)$
⋮
⇒ f(f(f(a,b), a), a) .

□

We hope that the informal comments together with the example help in understanding the construction. The correctness proof is complex and given in a full version of this paper.

We note that the techniques used in [DaGo] to show how level-n pda's can be simulated by level-n grammars can be adopted straightforwardly to the tree case. Thus we obtain the following theorem.

3.7 Theorem

$\mathcal{L}$ (n-TREEPD) = $\mathcal{L}$ (n-PD)

□

Now note that both datatypes have an identity instruction (for n-TREEPD, *id* = *push* <1,s>) , thus by 2.7 the class of translations induced by RTT's over these datatypes are equal.

3.8 Corollary

$\mathcal{T}_\varepsilon$ (n-TREEPD) = $\mathcal{T}_\varepsilon$ (n-PD)

□

In particular this proves the result conjectured in [DaGu], that level-n stack transducers induce a strict hierarchy of tree-translations.

REFERENCES

[Da] W. DAMM, *The IO -and OI-hierarchies* TCS 20 (2) 1982, 95-207.

[DaGo] W. DAMM, A.GOERDT, *An automata- theoretic characterization of the OI hierarchy*, Proc.9th ICALP, LNCS 140 (1982), 141-153.

[DaGu] W. DAMM, I. GUESSARIAN, *Combining T and level-N*, Proc. MFCS'81 LNCS 118 (1981), 262-270.

[Eng 1] J. ENGELFRIET, *Some open questions and recent results on tree transducers and tree languages*, in Formal languages: perspectives and open problems, Academic Press, London (1980), 241-286.

[Eng 2] J. ENGELFRIET, *Recursive automata*, unpublished notes (1982).

[Eng 3] J. ENGELFRIET, *Iterated pushdown automata and complexity classes*, to appear in Proc. S.T.O.C. Conference (1983).

[Eng 4] J. ENGELFRIET, *Tree automata and tree grammars*, Aarhus Univ. Tec. Report # DAIMI FN-IO (1975).

[Fi] M.J. FISCHER, *Grammars with macro-like productions*, 9th SWAT (1968), 131-142.

[Gal] J.H. GALLIER, *DPDA's in "atomic normal form" and applications to equivalence problems* ICS (1981), 155-186.

[Gre] S.A. GREIBACH, *Full AFLs and nested iterated substitution*, Inf. and Control 16 (1976), 7-35.

[Gue] I. GUESSARIAN, *Pushdown tree automata*, to appear.

[Mas] A.N. MASLOV, *Multilevel stack automata*, Problemy Peredachi Informatsii 12 (1976), 55-62.

[Rou] W.C. ROUNDS, *Mappings and grammars on trees*, MST 4 (1970) 257-287.

[Sco] D. SCOTT, *Some definitional suggestions for automata theory* JCSS 1 (1967), 187-212.

[DaFe] W. DAMM, E.FEHR *A schematological approach to the analysis of the procedure concepts in ALGOL-languages*, Proc. 5ième colloque sur les Arbes en Algebre et en Programmation Lille, (1980), 130-134.

[DaJo] W. DAMM, B. JOSKO *A sound and Relatively* Complete Hoare-Logic for a Language with higher type Procedures*, to appear in Acta Informatica

On some problems with operational semantics of functional programming languages

Elfriede Fehr

(extended abstract)

Functional (applicative) programming languages, such as LISP (McCarthy et.al., 1965), KRC (Turner, 1981), PCF (Plotkin, 1977) etc., are in general based upon the lambda-calculus (Church, 1941). Although operational and denotational semantics of the lambda-calculus are by now well understood (Gordon, 1975), most of the existing implementations of the lambda-calculus correspond to incomplete versions or inconsistent extensions of the axioms of the lambda-calculus.

The reason for this is mainly the fact that the standard reduction of leftmost-outermost β-redexes with preceeding tests on variable conflicts and appropriate renaming is highly inefficient, when implemented on or simulated by a machine.

Interpreters of LISP and LIPS-like languages as the one given in (McCarthy et.al., 1965) use in general a dynamic binding mechanism, which violates the semantics of the underlying lambda-calculus as shown in (Eick and Fehr, 1983).

Another well known implementation of the lambda-calculus is the SECD-machine (Landin, 1964), which supports the correct scope-rules of the lambda-calculus but as shown in (McGowan, 1970) fails the reduce all expressions having a normal form, because functional arguments cannot be treated appropriately. Similar problems arise with implementation on a cellular computer architecture as introduced in (Magó, 1979), which are suitable only for a restricted class of functional languages as e.g. FP and FFP (Backus, 1978) because objects of higher functional types cannot be handled. Other implementations such as the graph-machine introduced in (Wadsworth, 1971) or the combinator reduction introduced in (Turner, 1979) make a radical change in the representation of lambda-expressions. They work on labelled graphs or purely combinatory expressions, with the effect that the original lambda-expressions gets lost and intermediate results can hardly be understood by the programmer.

All these deficiencies can be overcome by implementing a variant of the lambda-calculus, which maintains the original structure and naming of an expression. This variant is obtained by adding an unbinding mechanism lambda-bar (#) to the language, which neutralizes the effect of one preceding lambda-binding. For example the variable x occurs free in the expression $\lambda x.\#x$ but bound in the expression $\lambda x.\lambda x.\#x$.

The benefit of this extension is that β-conversion can be perfomed without renaming of variables by systematically using the lambda-bar mechanism.

As a result machine models of languages based upon this extension have an uncomplicated machine structure and run very efficiently.

The corresponding formalism was first introduced in (De Bruijn, 1972). De Bruijn uses an implementation of this mechanism in his AUTOMATH-project (De Bruijn, 1976) and shows that it is very efficient for automatic formula manipulation. In (Berkling 1976a) the same mechanism was introduced independently. Berkling developed in (Berkling, 1976b) a reduction language BRL which is an extension of the lambda-calculus not only by a certain set of base operations, such as conditionals, arithmetical, boolean, and list operations, but also by the unbinding mechanism lambda-bar. A machine implementation of BRL was first simulated in PL/I (Hommes, 1977) and then a hardware-model was built (Kluge, 1979), which started operating in 1978 and has since the shown a satisfactory performance. In particular the machine supports different reduction strategies, such as a finite number of call-by-value reductions followed by a call-by-name reduction. Hence the efficiency of the BRL-machine can again be increased with more efficient strategies.

The semantic effect of the lambda-bar operation on the lambda-calculus was until now not very clear, since there existed only the syntactical and operational descriptions of it. In (Berkling and Fehr, 1982) the denontational semantics and a proof that it consistently extends the lambda-calculus is presented.

REFERENCES

[1] Backus, J. (1978), Can Programming be Liberated from the von Neumann Style?, Comm. of the Assoc. Comput. Mach. 21, 613-641.

[2] Berkling, K.J. (1976 a), A Symmetric Complement to the Lambda-Calculus, Interner Bericht ISF-76-7, GMD, D-5205 St. Augustin-1.

[3] Berkling, K.J. (1976 b), Reduction Languages for Reduction Machines, Interner Bericht ISF-76-8, GMD, D-5205 St. Augustin-1.

[4] Berkling, K.J. and Fehr, E. (1982), A Modification of the λ-calculus as a Base for Functional Programming Languages *in* "Automata, Languages and Programming" (M. Nielsen and E.M. Schmidt, Eds.), pp. 35-47, Lecture Notes in Computer Science No. 140, Springer-Verlag, Berlin.

[5] De Bruijn, N.G. (1972), Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem, *Indag. Math.* 34, 381-392.

[6] McCarthy, J. et al. (1965), "LISP 1.5 Programmer's Manual", M.I.T. Press, Mass.

[7] Church, A. (1941), "The Calculi of Lambda-Conversion", Princeton University Press, Princton.

[8] Eick, A. and Fehr, E. (1983), Inconsistencies of Pure LISP, *in* "Theoretical Computer Science, 6th GI-Conference" (A.B. Cremers and H.P. Kriegel, Eds.), pp. 101-110, Lecture Notes in Computer Science No. 145, Springer Verlag, Berlin.

[9] Gordon, M. (1975), Operational Reasoning and Denotational Semantics, Memo AIM-264, Stanford Artificial Intelligence Laboratory, Stanford.

[10] McGowan, C.L. (1970), The Correctness of a Modified SECD-machine, *in* Second ACM-Symposium on Theory of Computing, pp. 149-157, ACM, New York.

[11] Hommes, F. (1977), The Internal Structure of the Reduction Machine, Interner Bericht ISF-77-3, GMD, D-5205 St. Augustin-1.

[12] Kluge, W.E. (1979), The Architecture of a Reduction Language Machine Hardware Model, Interner Bericht ISF-79-3, GMD, D-5205 St. Augustin-1.

[13] Landin, P.J. (1964), The Mechanical Evaluation of Expressions, *Comp. J.* 6, 308-320.

[14] Magó, G. (1979), A Network of Microprocessors to Execute Reduction Languages, *Int. J. of Comp. and Inform. Sciences* 8, 349-385.

[15] Plotkin, G.D. (1977), LCF Considered as a Programming Language, *Theoret. Comput. Sci.* 5, 223-255.

[16] Turner, D.A. (1979), A New Implementation Technique for Applicative Languages, *Software - Practice and Experience* 9, 31-49.

[17] Turner, D.A. (1981), The Semantic Elegance of Applicative Languages, *in* "Functional Programming Languages and Computer Architecture (Arvind, J. Dennis, Eds.), pp. 85-92, ACM, New York

[18] Wadsworth, C. (1971), Semantics and Pragmatics of the Lambda-Calculus, Ph.D. thesis, Oxford University, Oxford.

COMPLEXITY OF INFINITE TREES

K. Indermark
Lehrstuhl für Informatik II, RWTH Aachen
Büchel 29 - 31, 5100 Aachen
W.-Germany

Abstract

Rational schemes interpreted over derived algebras permit a simple algebraic analysis of higher type recursion. Their equivalence is characterized by infinite trees. Measuring their complexity by the size of finite subtrees we obtain a direct proof of the recursion hierarchy.

Introduction

Recursion is certainly a fundamental control construct of programming languages. The problem whether its auxiliary use on higher functional domains adds computational power to a language has been investigated by W. Damm in great detail. He solved this problem successfully using the typed λ-calculus with fixed-point operators and constructing hierarchies of formal languages [Dam 82].
A simplified algebraic framework for proving the recursion hierarchy was suggested in [Ind 80] by means of rational schemes with rank-free interpretations. It is the purpose of this paper to demonstrate that this suggestion in fact allows an appropriate treatment of higher type recursion.
Denotational semantics supports the view that the function computed by a program is a certain combination of given base functions. Under the influence of D. Scott's work the advantage of continuous base functions was observed : their use avoids predicates and partial functions. As a consequence, complete algebras were recognized as a basis for an algebraic theory of programming [Niv 72, 75].
In a complete algebra there is a natural class of operations definable as the closure of projections and base operations under composition and resolution. The fundamental character of resolution (regular equations with parameters) already became evident in [Bek 69], [Wag 71 a,b] and [Wan 72].
Of course, resolution is simpler than recursion - their difference corresponds to that of regular and context-free languages. However, the simple control mechanism of regular equations turns out to be powerful enough in order to model not only recursion but also higher type recursion.
Technically, this is achieved by taking derived algebras (algebras of operations) as interpretations. This technique has first been used by Maibaum in formal language theory [Mai 74].
Since the derived algebra is heterogeneous according to the arities of operations, we allow an operation to have an arbitrary number of arguments. The resulting theory of complete algebras without rank is not very different from the ranked case, but it avoids many-sortedness.
Secondly, we modify the derivation insofar as base operations of the underlying algebra are not represented by nullary constants but by left-composition. This gives some additional simplifications.
The main advantage of this approach is that the semantics of a rational scheme interpreted by a derived algebra splits into several parts which can be analyzed seperately. Since there exist initial algebras for complete algebras representable by infinite trees [Gog 77], the equivalence class of a n-rational scheme can be characterized by an infinite tree which originates from a regular infinite tree by successive elimination of composition and projection symbols. Using an appropriate complexity measure we shall see that the resulting infinite trees become more and more complex. This easily demonstrates the recursion hierarchy.

1. Ω-algebras without rank

We consider algebras whose operations have an arbitrary number of arguments, including the empty list ε . This modification is of technical advantage for dealing with higher type objects because we thus circumvent many-sortedness.

Let A be a set.
Then $\underline{Ops}(A) := \{f \mid f : A^* \longrightarrow A\}$ is the set of *operations on A*.
Let Ω be a set of *operation symbols* (without arities).
Then $\psi : \Omega \longrightarrow \underline{Ops}(A)$ determines an Ω-*algebra* :

$$A := \langle A;\psi \rangle \in \underline{Alg}_\Omega$$

For any set X there exists

$$F_\Omega(X) \in \underline{Alg}_\Omega \text{ freely generated by } X .$$

Hence, any assignment $\alpha : X \longrightarrow A$ with $A = \langle A;\psi \rangle \in \underline{Alg}_\Omega$ extends uniquely to a homomorphism

$$\bar{\alpha} : F_\Omega(X) \longrightarrow A .$$

If $X = \emptyset$, we simply write F_Ω instead of $F_\Omega(\emptyset)$, and we denote the unique homomorphism by h_A .
The same situation arises when starting from a partially ordered or complete partially ordered set A -complete with respect to directed sub-

sets - and restricting $\underline{Ops}(A)$ to monotone or continuous operations on A . The corresponding objects are denoted by

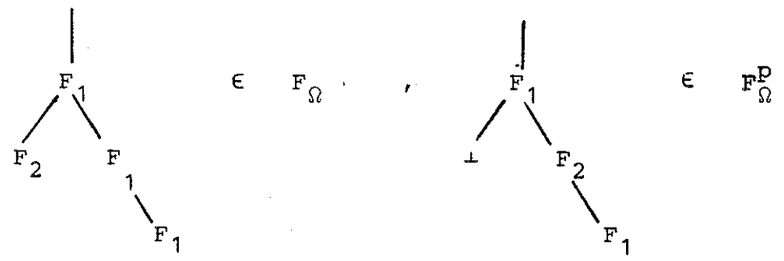
$$\underline{Alg}_\Omega^p \quad , \quad F_\Omega^p(X) \quad , \quad h_A^p$$

and $$\underline{Alg}_\Omega^c \quad , \quad F_\Omega^c(X) \quad , \quad h_A^c \quad .$$

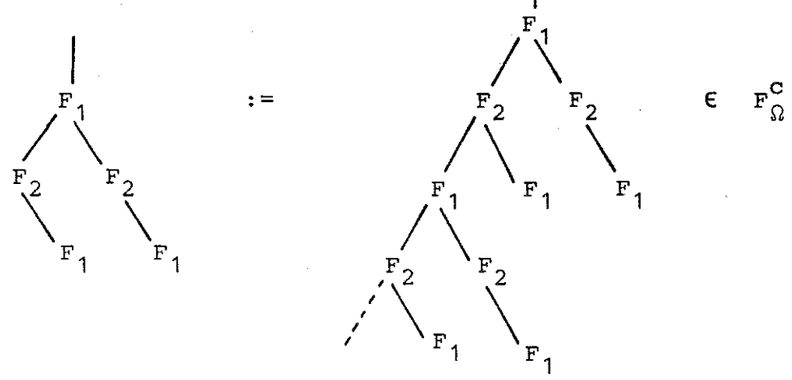
Although we do not need any representation of a freely generated algebra, we may of course view its elements as trees. In particular, we may assume set inclusion for the carrier sets :

$$F_\Omega(X) \subseteq F_\Omega^p(X) \subseteq F_\Omega^c(X)$$

Examples With $\Omega = \{F_1, F_2, \ldots\}$ we have



and



Note that the operation symbols have no arity and can therefore occur at arbitrary nodes.

Derived operations

If we choose a standard alphabet

$$X = \{x_1, x_2, \ldots\} \quad ,$$

each infinite tree $t \in F_\Omega^c(X)$ determines for $A \in \underline{Alg}_\Omega^c$ an operation.

For its definition we associate with

$$a = a_1 .. a_r \in A^*$$

the assignment

$$[a,\bot] : X \longrightarrow A$$

$$x_i \longmapsto \underline{proj}_i(a) := \begin{cases} a_i & \underline{if} \quad i \leqslant \underline{length}(a) \\ \bot & \underline{else} \end{cases}$$

and get the *derivation operator*

$$\underline{derop}_A : F_\Omega^c(X) \longrightarrow \underline{Ops}(A)$$

$$t \longmapsto (a \longmapsto \overline{[a,\bot]}(t)) \quad .$$

Substitution

If $A = F_\Omega^c(X)$, $\underline{derop}_A$ describes *substitution of infinite trees* :

$$\underline{derop}_{F_\Omega^c(X)} =: \underline{sbst}^\bot : F_\Omega^c(X) \longrightarrow \underline{Ops}(F_\Omega^c(X))$$

The index $\bot$ indicates that in $\underline{sbst}^\bot(t_o)(t_1..t_r)$ the variables $x_{r+1}, x_{r+2}, \ldots$ are replaced by $\bot$.

Iteration

In order to define *iteration of infinite trees* we shall also consider another kind of substitution which only substitutes given arguments and leaves the remaining arguments unchanged :

for $t_1, \ldots, t_r \in F_\Omega^c(X)$ we define the assignment

$$[t_1..t_r, x_{r+1}] : X \longrightarrow F_\Omega^c(X)$$

$$x_i \longmapsto \underline{if} \quad i \leqslant r \ \underline{then} \ t_i \ \underline{else} \ x_i$$

and get

$$\underline{sbst} : F_\Omega^c(X) \longrightarrow \underline{Ops}(F_\Omega^c(X))$$

$$t \longmapsto (t_1..t_r \longmapsto \overline{[t_1..t_r, x_{r+1}]}(t)) \quad .$$

This is used to introduce the *iteration of infinite trees at* i , where $i \geqslant 1$:

$$\underline{iter}_i : F_\Omega^c(X)^* \longrightarrow F_\Omega^c(X) \quad \text{is defined by}$$

$$\underline{iter}_i(t_1..t_r) :=$$

$$\underline{proj}_i(\underline{fix}(u_1..u_r \longmapsto \underline{sbst}(t_1)(u_1..u_r)..\ \underline{sbst}(t_r)(u_1..u_r)))$$

Here, $\underline{fix}$ gives the least fixed-point of a continuous transformation of $F_\Omega^c(X)^r$.

2. Rational Ω-schemes

For an arbitrary complete algebra $A \in \underline{Alg}^c_\Omega$ there exists a natural class of operations, called *rational operations*, which can be obtained uniformly from the class of projections by means of left-composition with base operations of A, composition and resolution. The essential construction is that of resolution : it corresponds to the least solution of a system of regular equations with parameters.
For defining rational operations we choose rational schemes in their general inductive form. This is of technical advantage as we can introduce at the same time the derivation mechanism.

Abstract syntax

We enlarge the set Ω of operation symbols first to the set

$$D(\Omega) := \{F' \mid F \in \Omega\} \cup \{\mathbb{P}_i \mid i > 0\} \cup \{\mathbb{C}\} \text{ of } \textit{derived symbols}$$

and next to the set

$$R(\Omega) := D(\Omega) \cup \{\mathbb{R}_i \mid i > 0\} \text{ of } \textit{rational symbols} \text{ over } \Omega.$$

The *algebra of rational Ω-schemes* is then defined as

$$Rat_\Omega := F_{R(\Omega)} \ .$$

Initial algebra semantics

Let $A = \langle A;\varphi\rangle \in \underline{Alg}^c_\Omega$ be an *interpretation* of Ω . For the semantics of rational Ω-schemes it suffices to construct an $R(\Omega)$-algebra because of initiality of Rat_Ω in $\underline{Alg}_{R(\Omega)}$. Again we proceed in two steps : first we define the *derived algebra* of A

$$D(A) := \langle \underline{Ops}(A);\bar{\varphi}\rangle \in \underline{Alg}^c_{D(\Omega)}$$

where

$$\begin{aligned}
&\bar{\varphi}(F')(f_1..f_r)(a) := \varphi(F)(f_1(a)..f_r(a)) && \textit{left-composition}\\
&\bar{\varphi}(\mathbb{P}_i)(f_1..f_r)(a) := \underline{proj}_i(a) && \textit{projection}\\
&\bar{\varphi}(\mathbb{C})(f_o..f_r)(a) := f_o(f_1(a)..f_r(a)) && \textit{composition}\\
&\bar{\varphi}(\mathbb{C})(\varepsilon)(a) := \perp_A
\end{aligned}$$

with $r,i \in \mathbb{N}$.
Next, we augment $D(A)$ to the *rational algebra* of A

$$R(A) := \langle \underline{Ops}(A);\bar{\varphi}\rangle \in \underline{Alg}^c_{R(\Omega)}$$

where in addition

$$\bar{\varphi}(\mathbb{R}_i)(f_1..f_r)(a) := \underline{proj}_i(\underline{fix}(f)) \text{ with } f : A^r \longrightarrow A^r,\ b \longmapsto f_1(ba)..f_r(ba)$$

Now, the semantics of rational Ω-schemes with interpretation $A \in \underline{Alg}^c_\Omega$ is given by the initial $R(\Omega)$-homomorphism

$$[\![\ ]\!]_A := h_{R(\Omega)} : Rat_\Omega \longrightarrow R(A)$$

Equivalence, infinite trees

As it may happen that different schemes define the same rational operation in each interpretation we have a natural equivalence relation on $\underline{Rat}_\Omega$ (the carrier set of Rat_Ω) :

$$S \sim T :\Leftrightarrow (\forall\, A \in \underline{Alg}^c_\Omega)\ [\![S]\!]_A = [\![T]\!]_A$$

We shall characterize this equivalence by infinite Ω-trees. For this purpose we extend the algebra of infinite Ω-trees

$$F^c_\Omega(X) = \langle F^c_\Omega(X);\varphi\rangle \in \underline{Alg}^c_\Omega$$

to an $R(\Omega)$-algebra such that the derivation operator turns out to be an $R(\Omega)$-homomorphism. For technical reasons, we proceed again in two steps.

$$F^c_\Omega(X)^D = \langle F^c_\Omega(X);\hat{\varphi}\rangle \in \underline{Alg}^c_{D(\Omega)}$$

is defined by

$$\begin{aligned}
&\hat{\varphi}(F') && := \varphi(F)\\
&\hat{\varphi}(\mathbb{P}_i)(t_1..t_r) && := x_i\\
&\hat{\varphi}(\mathbb{C})(t_o..t_r) && := \underline{sbst}^{\perp}(t_o)(t_1..t_r)\\
&\hat{\varphi}(\mathbb{C})(\varepsilon) && := \perp
\end{aligned}$$

and extended to

$$F^c_\Omega(X)^R = \langle F^c_\Omega(X);\hat{\varphi}\rangle \in \underline{Alg}^c_{R(\Omega)}$$

by $\hat{\varphi}(\mathbb{R}_i)(t_1..t_r) := \overline{[\perp^r,x_1]}(\underline{iter}_i(t_1..t_r))$

where the assignment $[\perp^r,x_1] : X \longrightarrow F^c_\Omega(X)$ is a shift of variables : $x_i \longmapsto \underline{if}\ i > r\ \underline{then}\ x_{i-r}\ \underline{else}\ \perp$.
It was shown in [Ind 80] that the derivation operator now is an $R(\Omega)$-homomorphism

$$\underline{derop}_A : F^c_\Omega(X)^R \longrightarrow R(A)$$

Moreover, since Rat_Ω is initial in $\underline{Alg}_{R(\Omega)}$ there is a unique $R(\Omega)$-homomorphism $\underline{tree}_\Omega : Rat_\Omega \longrightarrow F^c_\Omega(X)^R$

and we conclude the coincidence

$$[\![\ ]\!]_A = \underline{derop}_A \circ \underline{tree}_\Omega \ .$$

Now, the characterization of equivalence by infinite trees is easily

obtained : $S \sim T \Leftrightarrow \underline{tree}_\Omega(S) = \underline{tree}_\Omega(T)$

3. Derived interpretations

We are now well prepared for modelling higher type recursion : we leave the control structure represented by rational schemes unchanged, but take repeatedly derived algebras as interpretations.

Abstract syntax of (n+1)-rational schemes

By induction on $n \in \mathbb{N}$ we construct the set

$D^n(\Omega)$ of *derived symbols of degree n* :

$D^0(\Omega) := \Omega$

$D^{n+1}(\Omega) := D(D^n(\Omega))$

The *algebra of (n+1)-rational Ω-schemes* is then simply defined as

$$Rat_\Omega^{(n+1)} := Rat_{D^n(\Omega)} = F_{R(D^n(\Omega))}$$

Semantics

Let $A = \langle A;\varphi\rangle \in \underline{Alg}_\Omega^c$ be an interpretation of Ω .
By induction we get *the derived algebra of degree n*

$D^n(A) \in \underline{Alg}^c_{D^n(\Omega)}$:

$D^0(A) := A$

$D^{n+1}(A) := D(D^n(A))$

From initiality we conclude the unique $R(D^n(\Omega))$-homomorphism

$$h_{R(D^n(A))} : Rat_\Omega^{(n+1)} \longrightarrow R(D^n(A))$$

which gives us the semantics of an (n+1)-rational Ω-scheme S by

$$[\![S]\!]_{D^n(A)} := h_{R(D^n(A))}(S) : \underline{Ops}^n(A)^* \longrightarrow \underline{Ops}^n(A)$$

where $\underline{Ops}^0(A) := A$ and $\underline{Ops}^{n+1}(A) := \underline{Ops}(\underline{Ops}^n(A))$.

O-semantics

Our interest in using fixed-points on higher functional domains originated from the question whether the possibility of high level recursion increases the relative computational power of a language - in other words : can we define more elements of a complete algebra by auxiliary fixed-point constructions on higher functional levels.
In order to prove a positive answer it suffices to consider repeated applications of higher level functions to empty argument lists ε thus producing a low level object.
Therefore we modify the semantics.

For $S \in Rat_\Omega^{(n+1)}$ we define *O-semantics* of S with interpretation A by

$$[\![S]\!]^O_A := [\![S]\!]_{D^n(A)}(\varepsilon)(\varepsilon)\ldots(\varepsilon)^{n+1} \in A$$

Now, it becomes possible to compare schemes of different functional levels.
Let $S \in Rat_\Omega^{(n+1)}$ and $T \in Rat_\Omega^{(m+1)}$.
Their *O-equivalence* is defined by

$$S \,\&\, T :\Leftrightarrow (\forall A \in \underline{Alg}^c_\Omega)[\![S]\!]^\circ_A = [\![T]\!]^\circ_A \ .$$

Again, this equivalence can be characterized by infinite Ω-trees.
If $S \in Rat_\Omega^{(n+1)}$, we compute its Ω-tree by taking its rational $D^n(\Omega)$-tree followed by successive elimination of derived symbols.
The initial $D(\Omega)$-homomorphism

$$\underline{yield}_\Omega : F^c_{D(\Omega)} \longrightarrow F^c_\Omega(X)^D$$

describes this elimination. Since O-semantics is given by repeated application to empty argument lists, variables in infinite trees do not have any influence and will be replaced by ⊥ . Therefore, we define the Ω-homomorphisms

$$\underline{\bot\text{-yield}}_\Omega : F^c_{D(\Omega)} \longrightarrow F^c_\Omega \quad \text{by} \quad \overline{[\varepsilon,\bot]} \circ \underline{yield}_\Omega$$

and $$\underline{\bot\text{-tree}}_\Omega : F_{R(\Omega)} \longrightarrow F^c_\Omega \quad \text{by} \quad \overline{[\varepsilon,\bot]} \circ \underline{tree}_\Omega$$

Taking into account that $[\![\ ]\!]_{D^n(A)} = \underline{derop}_{D^n(A)} \circ \underline{tree}_{D^n(\Omega)}$ we get the indicated splitting of O-semantics.

(1) Lemma For $S \in Rat_\Omega^{(n+1)}$ we have

$$[\![S]\!]^O_A = h^c_A(\underbrace{\underline{\bot\text{-yield}}_\Omega(\ldots\underline{\bot\text{-yield}}_{D^{n-1}(\Omega)}(\underline{\bot\text{-tree}}_{D^n(\Omega)}(S))\ldots)}_{=:\ \underline{O\text{-tree}}_\Omega(S)})$$

This implies as an immediate consequence the desired characterization of O-equivalence.

(2) Theorem Let $S \in Rat_\Omega^{(n+1)}$ and $T \in Rat_\Omega^{(m+1)}$.

Then : $S \,\&\, T \Leftrightarrow \underline{O\text{-tree}}_\Omega(S) = \underline{O\text{-tree}}_\Omega(T)$

In this way, higher type recursion leads to classes of infinite trees.
Let $\underline{Rattree}_\Omega^{(n+1)} := \underline{O\text{-tree}}_\Omega(Rat_\Omega^{(n+1)})$

be the set of *(n+1)-rational Ω-trees*. In the sequel, we shall investigate their complexity and prove the hierarchy theorem :

(3) Theorem $(\forall\, n \in \mathbb{N})\quad \underline{\mathrm{Rattree}}_{\Omega}^{(n+1)} \subsetneq \underline{\mathrm{Rattree}}_{\Omega}^{(n+2)}$

4. Representation of (n+1)-rational trees

We have seen that an (n+1)-rational Ω-tree is constructable from an 1-rational $D^n(\Omega)$-tree by successive elimination of derived symbols using $\underline{\perp\text{-yield}}$. In order to prove the hierarchy we shall investigate the impact of derived symbols on the structure of these trees. Therefore, we first choose an appropriate representation of derived symbols. In particular, we have to distinguish between different derivation levels. This will be indicated by a *level index* p and a *derivation index* q in

$$G^{p.q} \in D^n(\Omega) \qquad \text{where p.q is short for } (p,q) :$$

$$\begin{aligned} D^0(\Omega) &:= \{F^{1.0} \mid F \in \Omega\} \\ D(D^n(\Omega)) &:= \{G^{p.q+1} \mid G^{p.q} \in D^n(\Omega)\} \\ &\quad \cup \{\mathbb{P}_i^{n+2.0} \mid i > 0\} \qquad (\mathbb{P} := \mathbb{P}_1) \\ &\quad \cup \{\mathbb{C}^{n+2.0}\} \end{aligned}$$

It follows that p+q = n+1 which is the "semantic level" since $\bar{\varphi}(G^{p.q}) \in \underline{\mathrm{Ops}}^{n+1}(A)$.

Now we can explicitly describe the construction of an (n+1)-rational Ω-tree from an 1-rational $D^n(\Omega)$-tree, $n \geq 1$.

$$\underline{\perp\text{-yield}}_{D^{n-1}(\Omega)} : F^c_{D^n(\Omega)} \longrightarrow F^c_{D^{n-1}(\Omega)} \quad \text{splits into}$$

$$\overline{[\varepsilon,\perp]} \circ \underline{\mathrm{yield}}_{D^{n-1}(\Omega)}$$

$$\text{with} \quad \underline{\mathrm{yield}}_{D^{n-1}(\Omega)} : F^c_{D^n(\Omega)} \longrightarrow F^c_{D^{n-1}(\Omega)}(X)^D$$

Since this mapping, $\downarrow$ for short , is a $D^n(\Omega)$-homomorphism , we have

$$\begin{aligned} \downarrow(G^{p.q+1}(t_1..t_r)) &= G^{p.q}(\downarrow(t_1)..\downarrow(t_r)) \\ \downarrow(\mathbb{P}_i^{n+2.0}(t_1..t_r)) &= x_i \\ \downarrow(\mathbb{C}^{n+2.0}(t_0..t_r)) &= \underline{\mathrm{sbst}}^{\perp}(\downarrow(t_0))(\downarrow(t_1)..\downarrow(t_r)) \\ \downarrow(\mathbb{C}^{n+2.0}(\varepsilon)) &= \perp \end{aligned}$$

Moreover, as $\downarrow$ is continuous we may carry out the computation on finite subtrees (approximations).

At this point we realize the advantage of representing a base operation on level n by its left-composition on level n+1 : the tree structure remains unchanged. As an immediate consequence we get the inclusion part of the hierarchy theorem (3) :

$$(\forall\, n \in \mathbb{N})\quad \underline{\mathrm{Rattree}}_{\Omega}^{(n+1)} \subseteq \underline{\mathrm{Rattree}}_{\Omega}^{(n+2)}$$

For the proof we only have to replace in the defining 1-rational $D^n(\Omega)$-tree a symbol $G^{p.q}$ by $G^{p.q+1}$. The resulting 1-rational $D^{n+1}(\Omega)$-tree yields the same Ω-tree.

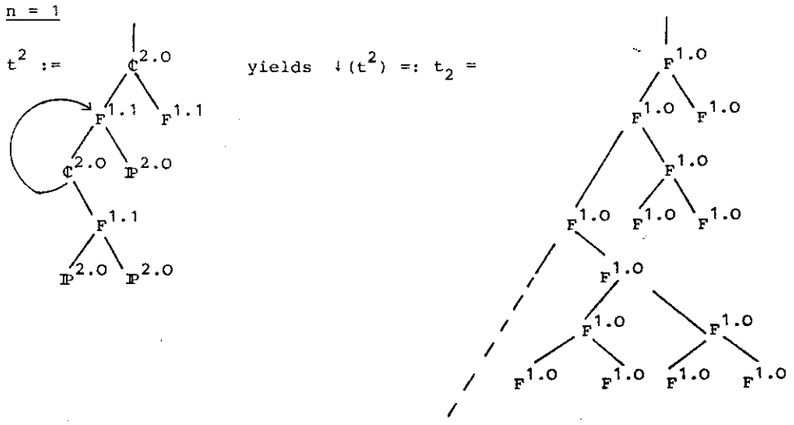
Next, we shall present for $n \geq 1$ "hierarchy candidates"

$$t_{n+1} \in \underline{\mathrm{Rattree}}_{\Omega}^{(n+1)}$$

such that $t_{n+1} \notin \underline{\mathrm{Rattree}}_{\Omega}^{(n)}$.

They are constructed from $t^{n+1} \in \underline{\mathrm{Rattree}}_{D^n(\Omega)}^{(1)}$ as explained below.

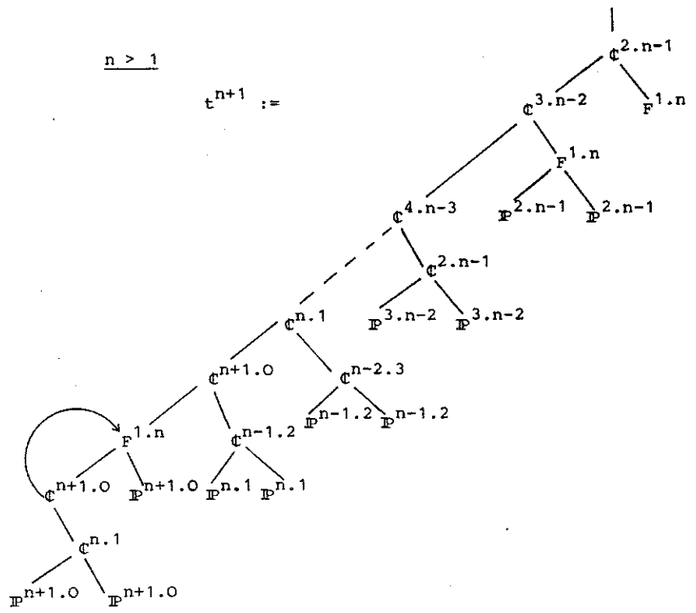
n = 1



Note that during $\downarrow$-computation $\mathbb{P}^{2.0}$ is replaced by x_1 which disappears within the substitution produced by elimination of $\mathbb{C}^{2.0}$.

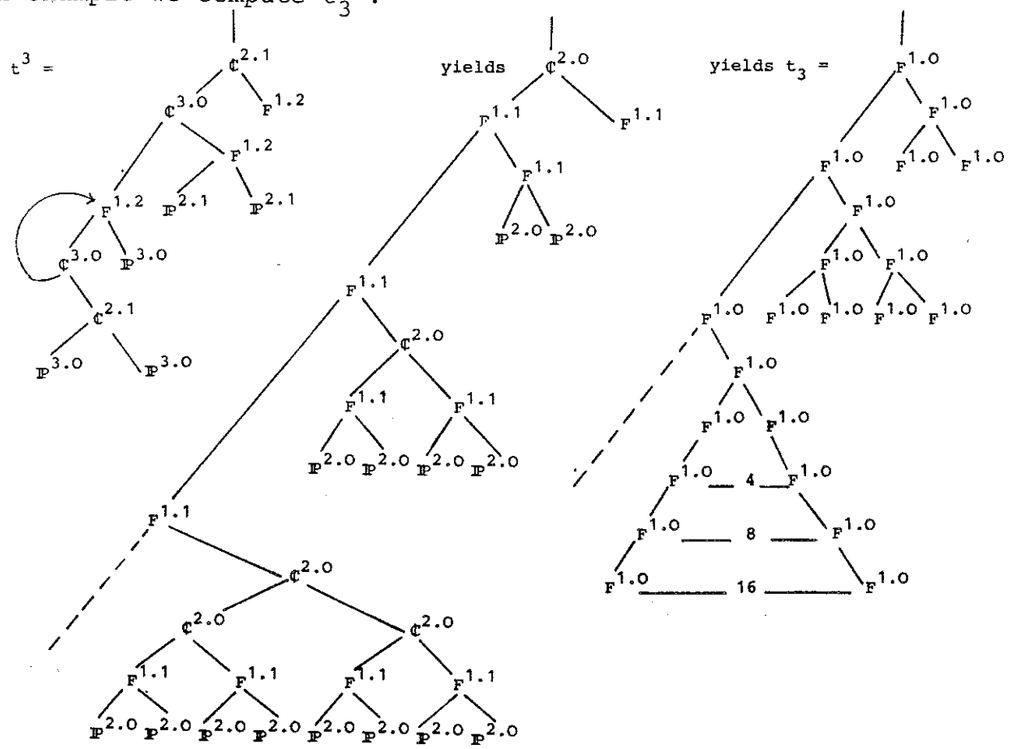
This first example already demonstrates the influence of derived symbols : iterated composition yields finite subtrees of growing size.

n > 1

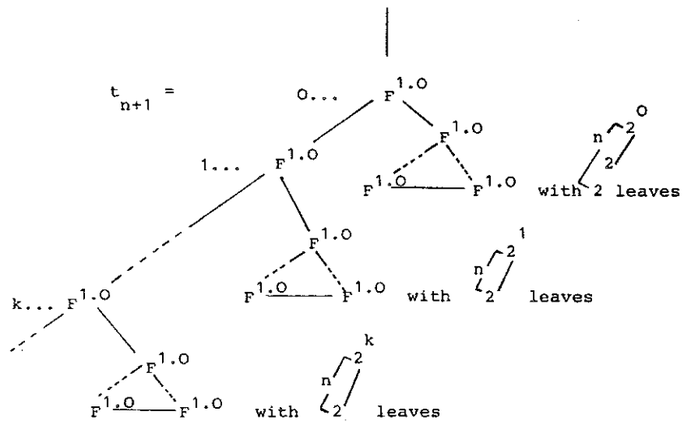


On computing t_{n+1} we carry out substitution of substitutions of ... so that the resulting tree becomes more and more complex. The growth of finite subtree size increases with level n.

As an example we compute t_3 :



For arbitrary $n > 1$ we can prove by induction that



5. Complexity of (n+1)-rational Ω-trees

Obviously, it holds for our example trees that $t_{n+1} \in \underline{Rattree}_\Omega^{(n+1)}$.
For a hierarchy proof it remains to verify that $t_{n+1} \notin \underline{Rattree}_\Omega^{(n)}$.
Therefore, we introduce a complexity measure on infinite trees following our previous observations : we choose the breadth of finite subtrees with respect to the depth of their roots.

Let $\mathbb{N}^\top$ be the cpo of natural numbers with top element such that

$$0 < 1 < 2 < \ldots < \top$$

and $M := \{f \mid f : \mathbb{N} \longrightarrow \mathbb{N}^\top\}$ be the cpo of *measures* with pointwise ordering.

The *measure algebra*

$$M = \langle M; \sim \rangle \in \underline{Alg}_\Omega^c$$

is then defined by

$$\tilde{F}(\varepsilon) := \underline{\text{if}}\ k = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0$$

$$\tilde{F}(f_1..f_r) := \underline{\text{if}}\ k = 0\ \underline{\text{then}}\ \underline{\text{if}}\ \prod_{i=1}^{r} f_i(0) \neq 0\ \underline{\text{then}}\ \sum_{i=1}^{r} f_i(0)\ \underline{\text{else}}\ 0\ \underline{\text{else}}\ \max_{i=1}^{r}(f_i(k-1))$$

Clearly, F does not depend on F. Moreover, one can easily check that $\tilde{F}$ is continuous. Hence, there is a unique Ω-homomorphism

$$\Gamma : F_\Omega^c(X) \longrightarrow M$$

which extends $x_i \longmapsto \underline{\text{if}}\ k = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0$.
$\Gamma(t)$ is called *finite subtree complexity* of t.
We see that $\Gamma(t)(k) \neq 0$ iff t contains a node of depth k being the root of a finite subtree $t' \in F_\Omega(X)$. Of these subtrees $\Gamma(t)(k)$ takes the maximal breadth.

Computing the complexity of our example trees t_{n+1} shows :

(4) Lemma $\quad \Gamma(t_{n+1})(0) = 0$, $\Gamma(t_{n+1})(k+1) = 2^{2^{n}}$ for all $n \geq 1$

Now, we determine upper bounds for the complexity of $t \in \underline{Rattree}_\Omega^{(n)}$.
First, we can prove that each regular tree has bounded complexity :

(5) Lemma For each $t \in \underline{Rattree}_\Omega^{(1)}$ there exists $c \in \mathbb{N}$ such that $\Gamma(t)(k) \leq c$ for all $k \in \mathbb{N}$.

This relies on the fact that a regular tree has only a finite number of different subtrees.

(6) Corollary $\quad \underline{Rattree}_\Omega^{(1)} \subsetneq \underline{Rattree}_\Omega^{(2)}$

Next, we analyse the increase of complexity produced by $\perp$-yield. So, let $t \in F_{D(\Omega)}^c$ be an arbitrary infinite $D(\Omega)$-tree such that $\Gamma(t) \leq f$.
A look at the computation of our example trees shows what happens on elimination of derived symbols : composition symbols produce substitution of subtrees causing exponential growth of the breadth of a finite subtree. However, if f is constant, as in the case of regular trees, $2^{f(n)}$ remains constant ! But, since composition symbols can also occur on infinite branches they can produce more and more substitutions of finite subtrees :

(7) Theorem Let $t \in F^{c}_{D(\Omega)}$ such that $\Gamma(t) \leqslant f$.
Then there are c_1 , $c_2 \in \mathbb{N}$ such that $\Gamma(\underline{1\text{-yield}}(t))(k) \leqslant \prod_{i=1}^{c_1 \cdot k} 2^{c_2 \cdot f(i)}$

The proof proceeds by algebraic induction on the structure of finite approximations of t in $F^{p}_{D(\Omega)}$ and exploits continuity of yield .
From (5) and (7) it follows :

(8) Corollary For $n \geqslant 1$ and $t \in \underline{Rattree}^{(n+1)}_{\Omega}$ there are $c_1, c_2 \in \mathbb{N}$ such that

$$\Gamma(t)(k) \leqslant \left(2^{2^{n}}\right)^{c_1 \cdot k + c_2}$$

Therefore : $t_{n+1} \notin \underline{Rattree}^{(n)}_{\Omega}$.

Conclusion

The algebraic analysis of recursion on higher functional domains shows that its computational power grows with increasing functional level. Moreover, the approach by combinators (derived symbols) clearly demonstrates the reason for this phenomenon : it is composition of higher functionals which in connection with recursion causes growing complexity.

Acknowledgement

It should be clear that this paper would not have been written without the long-standing efforts W. Damm spent on solving the recursion hierarchy problem. In numerous discussions with him I learnt a lot about the nature of higher type recursion. I gratefully acknowledge his help.

References

[Bek 69] Bekić, H : Definable operations in general algebras, and the theory of automata and flowcharts ,Research Report,IBM Lb., Vienna, 1969

[Dam 82] Damm, W. : The IO- and OI-hierarchies
Theoret. Comput. Sci. 20 (1982) 2, 95 - 208

[Gog 77] Goguen, J.A. et. al. : Initial algebra semantics and continuous algebras
Journal ACM 24 (1977) 1, 68 - 95

[Ind 80] Indermark, K. : On rational definitions in complete algebras without rank
Theoret. Comput. Sci. 21(1982), 281 - 313

[Mai 74] Maibaum, T.S.E. : A generalized approach to formal languages
Journal Comp. Syst. Sci. 8 (1974), 409 - 439

[Niv 72] Nivat, M. : Langages algébriques sur le magma libre et sémantique des schémas de programme
in : Automata, Languages, and Programming (M. Nivat, ed.) North-Holland P.C. (1972) , 293 - 308

[Niv 75] Nivat, M. : On the interpretation of recursion polyadic program schemes - Symposia Matematica 15, Rome 1975, 255 - 281

[Wag 71a] Wagner, E.G. : An algebraic theory of recursive definitions and recursive languages - Proc. 3rd ACM Symp.
Theory of Computing (1971) ,12 - 23

[Wag 71b] Wagner, E.G. : Languages for defining sets in arbitrary algebras - Proc. IEEE Conf. SWAT 12 (1971), 192 -201

[Wan 72] Wand, M. : A concrete approach to abstract recursive definitions - in : Automata, Languages, and Programming (M. Nivat, ed.) North-Holland P.C. (1972), 331 - 344

AUTOMATED CONSTRUCTION OF VERIFICATION CONDITIONS

Thomas Käufl
Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1

The question if the correctness of programs can be established automatically was raised as early as 1967 when Floyd's paper appeared. His foundational work led to the program verifier of King [9], to the Stanford verifier [6] and others.

In this paper we present the rules for generating verification conditions which guarantee the correctness of a program in such a way that they specify an algorithm performing the task. A system generating verification conditions is currently implemented [8].

1. The Function VC Generates Verification Conditions

In this chapter we define verification conditions and the function VC generating them.

Suppose we are given a precondition P and a postcondition Q and some means to evaluate the truth of correctness formulae like P|pc|Q where pc is a construct of a programming language.

Definition:
A *verification condition* for a language construct pc, a precondition P and a postcondition Q is a logical formula whose truth implies that of P|pc|Q.

If pc does not contain while or if-statements, then verification conditions are implications P→R such that the truth of R|pc|Q is implied by P→R.

The function VC yielding verification conditions has the functionality

$$F \times PC \times F \rightarrow F$$

where F is the set of well formed logical formulae and PC is a construct of the programming language. The first argument is used as precondition and the third as postcondition.

The following equations define VC for the common program constructs:

Assignment

$$VC(P,sta;\ x := e,\ R) = VC(P,sta,[R]^{e}_{x})$$

$$VC(P,sta;\ x[e_1] := e_2,R) = VC(P,sta,\ [R]^{x[e_1/e_2]}_{x})$$

Here $[R]^{e}_{x}$ denotes R where every free occurence of x is replaced by e. $x[e_1/e_2]$ is the modification of the array x: $x_{[e_1/e_2]}[i] = x[i]$ if $i \neq e_1$ and $x_{[e_1/e_2]}[e_1] = e_2$.

Assert

$$VC(P,sta;\ \underline{assert}\ Q,R) = VC(P,sta,Q) \wedge (Q \rightarrow R)$$

Assume

$$VC(P,sta;\ \underline{assume}\ Q,R) = VC(P,sta,Q \rightarrow R)$$

Empty-Statement

$VC(P,\varepsilon,R) = P \rightarrow R$

While-Statement

$VC(P,sta;\ \underline{invariant}\ Q;\ \underline{while}\ e\ \underline{do}\ sta',R)$
$= VC(P,sta,Q) \wedge VC(Q \wedge e, sta'; Q) \wedge (Q \wedge \neg e \rightarrow R)$

We require every while-statement to be preceded by an invariant: The designer of the program is assumed to know the invariant assertion being the basis for the behaviour of the loop. Thus the definition of VC forces him to write the invariant into his program.

If-Statements

$VC(P,sta;\ \underline{if}\ e\ \underline{then}\ sta_1\ \underline{else}\ sta_2\ \underline{fi},R)$
$= VC(P,sta;\ \underline{assume}\ e;\ sta_1,R)$
$\wedge$
$VC(P,sta;\ \underline{assume}\ \neg e;\ sta_2,R)$

$VC(P,sta;\ \underline{if}\ e\ \underline{then}\ sta'\ \underline{fi},R)$
$= VC(P,sta;\ \underline{assume}\ e;\ sta',R)$
$\wedge\ VC(P,sta;\ \underline{assume}\ \neg e,R)$

Compound Statement

$VC(P,sta_1;\ \underline{begin}\ sta_2\ \underline{end},Q)$
$= VC(P,sta_1;\ sta_2,Q)$

Procedure call

$VC(P,sta;\ p(e,v),S)$
$= VC(P,sta,[Q]^{e\ v}_{a\ r} \wedge ([R]^{e\ v}_{a\ r} \rightarrow S))$

where: e is the actual call-by-value parameter
v is the actual call-by-reference parameter
$Q|p(a,r)|R$ procedure hypothesis
a is the formal call-by-value parameter
r is the formal call-by-reference parameter

It is obvious that this equation for procedure calls can be generalized to an arbitrary number of parameters.

Construct definition

$VC(P,\underline{const}\ x = y;\ dec;\ sta,Q)$
$= VC(P,[dec;\ sta]^{y}_{x},Q)$

Variable declaration

$VC(P,\underline{var}\ x;\ dec;\ sta,Q)$
$= VC(P,[dec;\ sta]^{x'}_{x},Q)$

where x' is a new identifier not occuring in P, Q and dec; sta.

Initials

$VC(P,initial\ x' = x,Q) = VC(P,\varepsilon,[Q]^{x}_{x},)$

Let us consider the equations for constant and variable declarations. A declaration changes the environment, which is a textual notion. Defining the equations for VC we wish to be as close as possible to predicate transformers (see [10]) yielding weakest liberalized predicates. Hence following Milne [10] we prefer to rename the new declared indentifiers instead of renaming identifiers in the pre- and postconditions in order to avoid name clashes.

Procedure declaration

$VC(P,procdec;\ dec;\ sta,Q)$
$= VC(P,dec;\ sta,Q) \wedge$
$\wedge\ VC(R,\underline{value}\ x;\ \underline{initial}\ x' = x;\ dec_p;\ sta_p,S)$
where procdec
$\underline{procedure}\ p(x:type;\ \underline{var}\ y:type);$
$\underline{entry}\ R;$
$\underline{exit}\ S;$
$\underline{initial}\ x' = x;$
$dec_p;\ sta_p$

Value specification

$VC(P,\underline{value}\ x;\ \underline{initial}\ x' = x;\ dec;\ sta,Q)$
$= VC(P,\{[dec;\ \underline{initial}\ x_o = x;\ sta]^{x_o}_{x'}\}^{x_o}_{x},Q)$

where x_o is a new identifier neither occuring in P, Q nor in the programming language construct and $\{sta\}^{x_o}_{x}$ is a substitution not applied to the asserts, invariants etc. contained in sta.

Initials

$VC(P,\underline{initial}\ x' = x,Q) = VC(P,\varepsilon,[Q]^{x'}_{x})$

The treatment of value specifications allows assignments to a value parameter in the body of the procedure. In order to motivate the definition let us consider the Hoare-like [7] proof rule for call-by-value

$$\frac{P|[sta]^{y}_{x}|Q}{[P]^{x}_{y}|sta|Q}$$

where y does not occur free in Q.

If a while-statement occurs in $[sta]^{y}_{x}$ the designer may use y to denote the current content of the parameter and x to denote the content of the parameter when the procedure is called. To make this explicit he has to specify this in the initial equation initial y = x. The equation dealing with value specifications together with the equation for initials treats the call-by-value specification as usual. The formal parameter becomes a variable local to the procedure body. The value of the corresponding actual parameter is then assigned to this local variable.

We conclude this chapter by noting a lemma which shows that the result of VC is a conjunction of logical formulae, a fact used frequently in the following chapter.

Lemma 1: $VC(P,pc,Q) = R_1 \wedge \ldots \wedge R_n$ where pc is a construct of the programming language and the R_i are logical formulae.

2. The Correctness of VC

In order to prove the correctness of VC we shall show:

Whenever

$$VC(P,pc,Q) = R_1 \wedge \ldots \wedge R_n$$

then there is a derivation using Hoare-like proof rules

$$R_1,\ldots,R_n,\Sigma_1,\ldots,\Sigma_k,P|\overline{pc}|Q$$

where $\overline{pc}$ is obtained from pc by cancelling all asserts, assumes etc. and $\Sigma_1,\ldots,\Sigma_k$ are correctness formulae occuring in the derivation.

In the following $R_1,\ldots,R_n \vdash P|\overline{pc}|Q$ denotes the existence of such a derivation.

The subsequent lemma shows how the splitting of statements is mirrored by VC.

Lemma 2:

$$VC(P,sta_1; sta_2,Q) = VC(P,sta_1,S_1' \wedge \ldots \wedge S_n') \wedge R_1 \wedge \ldots \wedge R_m$$

where $VC(\underline{true},sta_2,Q) = S_1 \wedge \ldots \wedge S_n \wedge R_1 \wedge \ldots \wedge R_m$

and $S_i = True \rightarrow S_i'$

In order to state the correctness of VC we need an auxiliar notion:

Definition:

Every construct beginning with assert, assume, invariant, initial, entry or exit is called *annotation*.

Theorem 1:

Assume $VC(P,sta,Q) = R_1 \wedge \ldots \wedge R_n$ such that sta does not contain an assume.

Then there exists a derivation

$$R_1,\ldots,R_n \vdash P|\overline{sta}|Q$$

where $\overline{sta}$ is obtained from sta by deleting all annotations.

Proof:

The proof is done by induction on the structure of the constructs of the language. We only show the induction steps for the procedure call and the value specification. The proofs for the other language constructs (except for declarations) and the proof of the induction base may be found in [6].

Value specifications:

$VC(P,\underline{value}\ x;\ \underline{initial}\ x' = x;\ dec;\ sta,Q)$

$= VC(P,\{[dec;\ \underline{initial}\ x_o = x;\ sta]^{x_o}_{x'}\}^{x_o}_{x},Q)$

by definition

$= VC(P,\ \underline{initial}\ x_o = x;\ sta',Q)$

where sta' is obtained from $\{[sta]^{x_o}_{x'}\}^{x_o}_{x}$ by substitutions according to the declarations dec

$= VC(P,\ \underline{initial}\ x_o = x,\ S'_1 \wedge \ldots \wedge S'_n) \wedge R_1 \wedge \ldots \wedge R_m$

because of Lemma 2.

$= P \rightarrow [S'_1 \wedge \ldots \wedge S'_n]^{x}_{x_o} \wedge R_1 \wedge \ldots \wedge R_m .$

Again by Lemma 2:

$VC(True,sta',Q) = S_1 \wedge \ldots \wedge S_n \wedge R_1 \wedge \ldots \wedge R_m$.

Using the induction hypothesis:

$S_1,\ldots,S_n,R_1,\ldots,R_m \vdash True|\overline{sta'}|Q$ 〉

$S_1,\ldots,S_n,R_1,\ldots,R_m \vdash True|\{[dec;\ \overline{sta}]^{x_o}_{x},\}^{x_o}_{x}|Q$ 〉

$\overline{sta}$ does not contain asserts etc. 〉

$S_1,\ldots,S_n,R_1,\ldots,R_m \vdash True|[dec;\ \overline{sta}]^{x_o}_{x}|Q$ 〉

$S_1 \wedge \ldots \wedge S_n \rightarrow True$ 〉

$R_1,\ldots,R_m \vdash S_1 \wedge \ldots \wedge S_n|[dec;\ \overline{sta}]^{x_o}_{x}|Q$ 〉

$R_1,\ldots,R_m \vdash [S_1 \wedge \ldots \wedge S_n]^{x}_{x_o}|\underline{value}\ x;\ dec;\ \overline{sta}|Q$ 〉

$P \rightarrow [S_1 \wedge \ldots \wedge S_n]^{x}_{x_o},R_1,\ldots,R_m \vdash P|\underline{value}\ x;\ dec;\ \overline{sta}|Q$

(using the rule of consequence)

Procedure calls:

$VC(P,sta;\ p(e,v),S)$

$= VC(P,sta,[Q]^{e\ v}_{a\ r} \wedge \forall r([R]^{e}_{a} \rightarrow [S]^{r}_{v}))$

where R is the exit- and
Q the entry-assertion of p

$= P_1 \wedge \ldots \wedge P_n$.

By the induction hypothesis we have

$P_1,\ldots,P_r \vdash P|\overline{sta}|[Q]^{e\ v}_{a\ r} \wedge \forall r([R]^{e}_{a} \rightarrow [S]^{r}_{v})$

In order to obtain $P|\overline{sta};\ p(e,v)|S$ we execute the following derivation:

(1) $P|\overline{sta}|[Q]^{e\ v}_{a\ r} \wedge \forall r([R]^{e}_{a} \rightarrow [S]^{r}_{v})$

(2) $Q|p(a,r)|R$ procedure hypothesis

(3) $[Q]^{e\ v}_{a\ r}|p(e,v)|[R]^{e\ v}_{a\ r}$ Call (2)

(4) $\forall r([R]^{e}_{a} \rightarrow [S]^{r}_{v}) \wedge [Q]^{e\ v}_{a\ r}$

$|p(e,v)|[R]^{e\ v}_{a\ r} \wedge \forall r([R]^{e}_{a} \rightarrow [S]^{r}_{v})$

Invariance (3)

(5) $[R]^{e\ v}_{a\ r} \wedge \forall r([R]^{e}_{a} \rightarrow [S]^{r}_{v}) \rightarrow S$

Note: R does not contain
free occurences of r

(6) $P|\overline{sta};\ p(e,v)|S$

by applying the rules of
consequence and composition
to (1, 4, 5)

□

In the derivations presented in the proof we have used the following proof rules:

Procedure hypothesis

$Q|p(a,r)|R$ where each identifier occuring free in Q or R must be a or r

Call

$$\frac{Q|p(a,r)|R}{[Q]^{e\ v}_{a\ r}|p(e,r)|[R]^{e\ v}_{a\ r}}$$

Invariance

$$\frac{Q|p(e,r)|R}{Q \wedge P|p(e,r)|R \wedge P}$$

where each identifier free in P is different from r

Call-by-value

$$\frac{P|[sta]^{y}_{x}|Q}{[P]^{x}_{y}|sta|Q}$$

where y must not occur free in Q

In order to prove the induction step for variable- and constant declarations in the proof of the theorem one has to use the following proof rules.

$$\frac{P|[dec;\ sta]^{c}_{id}|Q}{P|\underline{const}\ id = c;\ dec;\ sta|Q}$$

where c is a constant

and

$$\frac{P|[dec;\ sta]^{y}_{x}|Q}{P|\underline{var}\ x;\ dec;\ sta|Q}$$

where y must not occur free in P, Q and dec; sta

The rule for constant declarations is an adaptation of Milne's results [10].

The rule for variable declarations may be found in [1] too. Many authors (see [2], [3] and [11]) reject this rule. They

prefer an appropriate renaming of the pre- and postcondition in order to avoid name clashes possibly connected with variable declarations.

The question of the completeness of VC is open: We have to show that VC(P,sta,Q) yields true verification conditions whenever P|sta|Q is derivable using Hoare-like proof rules. This question is currently investigated. Note that this problem also arises when one studies the proof rules in presented in [4]. In this paper the question whether the proof rules have the same power as traditional Hoare-like proof rules as far as the same language constructs are concerned is not answered too.

References

[1] Apt, K.R.: Ten Years of Hoare's Logic, a Survey
Techn. Report, Faculty of Economics, Erasmus Univ., Rotterdam: 1979

[2] Cartwright R., Oppen D.: Unrestricted Procedure Calls in Hoare's Logic
Proc. of the 5th ACM Symposion on Principles of Programming Languages

[3] Cook, S.A.: Soundness and Completeness of an Axiom System for Program Verification
SIAM Journal on Computing 7, pp. 70-90: 1978

[4] Ernst G.W., Jainendra K.N., Ogden W.F.: Verification of Programs with Procedure-Type Parameters
Acta Informatica 18, pp. 149-169: 1982

[5] Floyd R.W.: Assigning Meanings to Programs
Proc. of the Symposia in Applied Mathematics 19, pp. 19-32: 1967

[6] Igarashi S., London R.L., Luckham D.C.: Automatic Program Verification I: A Logical Basis and its Implementation
Acta Informatica 4, pp. 145-182: 1975

[7] Käufl Th.: Proof Rules for Procedures
Workshop on Reliable Software
Proceedings of the 2nd Meeting of the German Chapter of the ACM, München: 1979

[8] Käufl Th., Heisel M., Reif W.: The Verification Condition Generator "Tatzelwurm"
Reference Manual and Users Guide - In Preparation

[9] King J.C.: A Program Verifier
Carnegie-Mellon University, Pittsburgh, Ph.D. Thesis: 1969

[10] Milne R.: Transforming Predicate Transformers
Proc. of the IFIP Working Conference on Formal Description of Programming Concepts, IFIP: 1979

[11] Olderog E.-R.: Charakterisierung Hoarescher Systeme für ALGOL-ähnliche Programmiersprachen
Bericht 5/81, Institut für Informatik und Praktische Mathematik, Universität Kiel: 1981

Dr. H.-J. Klein, Kiel

Thema des Vortrags: "Eine Methode zur Konstruktion von Automaten für die Simulation des Laufzeitverhaltens von Programmen"

Zusammenfassung: Die durch Anwendung der Kopierregel auf Programme ALGOL-ähnlicher Programmiersprachen erzeugbaren Ausführungsbäume sind i.a. nicht endlich. Für Programme bestimmter Teilklassen solcher Sprachen konnten jedoch endliche Unterbäume der Ausführungsbäume angegeben werden, die die Ausführungsbäume vollständig charakterisieren. Wir definieren eine sogenannte erweiterte Kopierregel und betrachten spezielle Ausführungsbäume, zu denen stets derartige charakterisierende endliche Unterbäume gefunden werden können. Über diesen speziellen Ausführungsbäumen lassen sich "Durchläufe" definieren, die den Pfaden in den gewöhnlichen Ausführungsbäumen entsprechen. Für dieses Durchlaufen der Bäume werden Informationen über gewisse statische Verweise und über Zuordnungen von aktuellen zu formalen Parametern benötigt. Der zur Verwaltung dieser Information und zur Realisierung der Durchläufe auf dem charakterisierenden Unterbaum des speziellen Ausführungsbaumes eines Programms notwendige Speichertyp bestimmt den Typ des Automaten, der zur Simulation des Laufzeitverhaltens dieses Programms geeignet ist. Wir demonstrieren diese Konstruktionsmethode an einigen Beispielen.

M. Krause, W.-M. Lippe, F. Simon

ON ALGEBRAIC SEMANTICS OF IMPERATIVE PROGRAMMING LANGUAGES

Abstract

We give a survey of algebraic semantics and it's problems with imperative program constructs. Generalized recursive program schemes (GRPS) are used to cope with these problems. Thereby, the solvability of so called local variables is a major point. A method of L. Kott /Ko76/ to show the solvability is discussed. Due to technical reasons Kott makes restrictions which do not allow to define algebraic semantics for programs with mutually recursive procedures. We show that it is possible to solve local variables without Kott's restrictions by using some elementary ideas of the copy-rule-semantics. Thus we are able to define algebraic semantics for a more powerful set of imperative programs.

I. Introduction

For the definition of a programming language one distinguishes between two fields: syntax, defining the structure of programs, and semantics, giving meaning to programs. Formal description of the syntax are well known. But for a long time semantics were usually given informally by describing the effects of syntatic structures on a particular computer or on an abstract model of a computer.

But since one considers a program as a definition of a mathematical function /McC 63,Pa 69/ it is possible to describe the semantics of a program formally, i.e. by mathematical means. This leads to the so called "fixed point semantics" or "denotational semantics" introduced by D. Scott and C. Strachey /Sc 70/. Z. Manna and J. Vuillemin /Ma 72,Ma 74/ applied this method to ALGOL-like languages.

Another method is the "algebraic semantics" introduced by M. Nivat /Ni 72, Ni 74/. In this approach programs are simple rewriting systems defined by recursive equations, called recursive program schemes (RPS). This method is immediately applicable to functional languages. But applied to ALGOL-like languages there arise problems with some imperative program constructs, i.e. assignments, value parameters, block nesting and local variables. To handle these structures correctly, the RPS were augmented by L. Kott /Ko 76,Ko 77/ to the generalized recursive program schemes (GRPS). This concept is described in chapter III. The theory of GRPS yields "computing processes" as they are needed for a formal description of the semantics of variables in imperative programming languages.

Because of the chosen proof technique, however, Kott can only handle programs which do not have mutual recursive procedures with value-parameters. This happens, because he regards procedures as parameterized blocks, which is generally not correct.

The main point of this paper is to show how mutual recursion of procedures with value parameters can be done in the theory of GRPS. For this we use some ideas from the "copy-rule-semantics" /La 73/ for programs with procedures. Then Kott's theorems and

definitions (see part "Interpretation and Semantics of a SSG") are to be adapted to our framework.

II. Algebraic semantics

In this section the theory of algebraic semantics will be roughly outlined. Let F be a finite set of function symbols. With every $f \in F$ is associated a positiv integer $\rho(f)$, called the arity of f. Let V be a set of variables. The free F-magma generated by V, M(F,V), is given inductively by:

1. $V \subseteq M(F,V)$
2. If $f \in F$ and $t_1,\ldots,t_{\rho(f)} \in M(F,V)$, then $f(t_1,\ldots,t_{\rho(f)}) \in M(F,V)$.

Thus M(F,V) is the set of all wellformed terms which can be constructed by function symbols $f \in F$ and by variables $v \in V$.

A recursive program scheme (RPS) Σ on M(F,V) is a system of equations

$$\Sigma: \varphi^i(v_1,\ldots,v_{\rho_i}) = \tau^i,\ 1 \leq i \leq N,$$

where $\Phi = \{\varphi^1,\ldots,\varphi^N\}$ is a set of unknown function symbols, τ^i is an element of $M(F \cup \Phi, \{v_1,\ldots,v_{\rho_i}\})$ and $\rho_i = \rho(\varphi^i)$.

We extend a RPS Σ to a schematic rewriting systems $\overline{\Sigma}$ by adding equations $\varphi^i(v_1,\ldots,v_{\rho_i}) = \Omega$ to Σ for all $\varphi^i \in \Phi$. These equations are abbreviated by $\varphi^i(v_1,\ldots,v_{\rho_i}) = \tau^i + \Omega$.

On the magma $M(F \cup \Phi, V)$ a derivation is defined by $\overline{\Sigma}$:

For all $t, t' \in M(F \cup \Phi, V)$ t' directly derives from z with regard to $\overline{\Sigma}$ ($t \xrightarrow[\Sigma]{} t'$) iff $t = \alpha\ \varphi^i(t_1,\ldots,t_{\rho_i})\ \beta$ and

$$t' = \alpha\ \tau^i(t_1,\ldots,t_{\rho_i}/v_1,\ldots,v_{\rho_i})\beta \text{ or } t' = \alpha\ \Omega\ \beta$$

where $\alpha,\beta \in X^*$, $X = F \cup V \cup \{'(',')',' ,'\}$ and $\tau^i(t_1,\ldots,t_{\rho_i}/v_1,\ldots,v_{\rho_i})$ results from τ^i by replacing all occurrences or v_j by t_j for $j=1,\ldots,\rho_i$.

$\xrightarrow[\Sigma]{+}$ ($\xrightarrow[\Sigma]{*}$) denotes the transitive (reflexive, transitive) closure of $\xrightarrow[\Sigma]{}$.

By $L(\overline{\Sigma},\varphi^1) =_{Df} \{t \in M(F,\ ) / \varphi^1(v_1,\ldots,v_{\rho_1}) \xrightarrow[\Sigma]{*} t\}$ we denote the set of all terms which are derived from $\varphi^1(v_1,\ldots,v_{\rho_1})$ with regard to $\overline{\Sigma}$.

Example: Let $\bar{\Sigma}$ be given by

$$\bar{\Sigma}: \varphi^1(v_1) = f(g(v_1),a,h(v_1,\varphi^1(k(v_1)))) + \Omega.$$

$$L(\bar{\Sigma},\varphi^1) = \{\Omega,$$

$$f(g(v_1),a,h(v_1,\Omega)),$$

$$f(g(v_1),a,h(v_1,f(g(k(v_1)),a,h(k(v_1),\Omega)))),\ldots\}.$$

Notice that in the theory of RPS no procedure parameters or formal procedure calls are allowed. Variables are always interpreted over simple domains, like integers, Booleans or strings.

A discrete interpretation I of a RPS Σ on $M(F,V)$ is given by

1. a discrete domain, i.e. a partially ordered set (D_I,E) with $x \sqsubseteq y$ iff $x=\omega$ or $x=y$, and $\omega \in D_I$ is the smallest element of D_I,
2. a set of monotone functions called magma operations $F_I = \{f_I: D_I^{\rho(f)} \longrightarrow D_I / f \in F\}$.

Thus a discrete interpretation fixes the set of values and the basic functions. Let I be a decrete interpretation.

A valued interpretation of a RPS Σ on $M(F,V)$

$(I,\nu)/M(F,V) \longrightarrow D_I$ is inductively defined by

$$(I,\nu)(\Omega) = \omega$$
$$(I,\nu)(v_i) = \nu(v_i), \text{ where } \nu/V \longrightarrow D_I \text{ is an environment}$$
$$(I,\nu)(f(t_1,\ldots,t_{\rho(f)})) = f_I((I,\nu)(t_1),\ldots,(I,\nu)(t_{\rho(f)})).$$

Now we can define the semantics of a RPS. Let Σ be a RPS on $M(F,V)$ and let I be a (discrete) interpretation. Then the semantics $Val_I(\Sigma,\varphi^1)/D_I^{\rho} \longrightarrow D_I$ of Σ under I is defined by

$$Val_I(\Sigma,\varphi^1)(d_1,\ldots,d_{\rho_1}) = \begin{cases} d' & \text{if there is a } t \in L(\bar{\Sigma},\varphi^1) \text{ with} \\ & (I,\nu)(t)=d', d' \neq \omega \\ \omega & \text{otherwise} \end{cases}$$

where $d_j = \nu(v_j)$ for $j=1,\ldots,\rho_1$.

The language $L(\bar{\Sigma},\varphi^1)$ can be considered as a set of finite approximations of the function which is "computed" by Σ.

Because of the following result $Val_I(\Sigma,\varphi^1)$ is well defined.

Lemma /Ni 74/:

Let be t, $t' \in L(\bar{\Sigma},\varphi^1)$ and let (I,ν) be valued interpretation. If $(I,\nu)(t) \neq \omega$ and $(I,\nu)(t') \neq \omega$, then $(I,\nu)(t) = (I,\nu)(t')$.

Using RPSs to describe the semantics of ALGOL-like programs causes problems with some imperative program constructs which cannot be regarded as basic functions over some simple domains. An example is the assignment statement.

Example /Ko 76/:

```
Π1 = begin integer procedure fact(x); integer x;
                fact:= if x=0 then 1 else x * fact(x-1);
          integer a,b;
          read(a);
          b:= fact(a);
          a:= if a =(a÷2) *2 then 1 else a↑b + b;
          write(a)
     end
```

By the transformation described in /Ma 72/ this program is associated with the following RPS:

$$\varphi^1(v_1) = f(q(v_1),a,s(r(v_1,\varphi^2(v_1)),\varphi^2(v_1))) \qquad (I)$$
$$\varphi^2(v_1) = f(g(v_1),a,h(v_1,\varphi^2(k(v_1)))). \qquad (II)$$

Equation (II) is the transformation of the declaration of the procedure fact. It is not so obvious how the main part of the program is transformed into equation (I). First of all the four statements of the main program are transformed into

```
integer procedure prog(a);
          prog:= if a=(a÷2) *2 then 1
                                else a↑fact(a) + fact(a);
out  prog(in)
```

where in and out denote I/O-operations.
Now it is obvious that equation (I) is the transformation of the declaration of the procedure prog. Unfortunately the substitution of b by fact (a) is only correct if the computation of fact terminates.

Let I be the following interpretation:

$D_I = \mathbb{Z} \cup \{\omega\}$.

For every $m, n \in D_I$ and $t \in \{false, true\}$ is

$a_I = 1$; $k_I(n) = n-1$; $h_I(m,n) = m \cdot n$;

$S_I(m,n) = m+n$; $r_I(m,n) = m^n$, if $n>0$;

$$q_I(n) = \begin{cases} true & \text{if } n \text{ is pair} \\ false & \text{if } n \text{ is odd} \\ \omega & \text{otherwise} \end{cases}$$

$$g_I(n) = \begin{cases} true & \text{if } n=0 \\ false & \text{if } n \neq 0 \text{ and } n \neq \omega \\ \omega & \text{otherwise;} \end{cases}$$

$$f_I(t,m,n) = \begin{cases} m & \text{if } t=true \\ n & \text{if } t=false \\ \omega & \text{otherwise.} \end{cases}$$

Let us take $f(q(v_1),a,s(r(v_1,\Omega),\Omega))$, which is a term of $L(\Sigma,\varphi^1)$ and let ν be an environment with $\nu(v_1) = -2$. Then

$$\begin{aligned} Val_I(\Sigma,\varphi^1)(-2) &= (I,\nu)(f(q(v_1),a,s(r(v_1,\Omega),\Omega)) \\ &= f_I((I,\nu)(q(v_1)),(I,\nu)(a),(I,\nu)(s(r(v_1,\Omega)\Omega))) \\ &= f_I(q_I(-2),1,S_I(r_I(-2,\Omega),\Omega)) \\ &= 1. \end{aligned}$$

It is easily verified that this is not the output of the given program for input a= -2. This example shows that elimination of assignments is not the proper way to capture the notion of a store. But by means of generalized recursive program schemes we are able to cope correctly with imperative language elements.

III. Generalized recursive program schemes

Definitions. Let F, Φ and V be defined as in chapter II and let W be a finite set of local variables with $W \cap (F \cup \Phi \cup V) = \emptyset$.

A generalized recursive program scheme (GRPS) Σ is a system of equations on the magma $M(F \cup \Phi, V \cup W)$

$$\Sigma: \begin{cases} \varphi^i(v_1,\ldots,v_{\rho_i}) = \tau^i, & i=1,\ldots,N \\ w^{i,j} = \tau^{i,j}, & j=1,\ldots,\sigma_i, \end{cases}$$

where $\tau^i \in M(F \cup \Phi, \{v_1,\ldots,v_{\rho_i}\} \cup \{w^{i,1},\ldots,w^{i,\sigma_i}\})$

and $\tau^{i,j} \in M(F \cup \Phi, \{v_1,\ldots,v_{\rho_i}\})$.

σ_i is the number of local variables belonging to φ^i.

Remember that all elements of $X = F \cup \Phi \cup V \cup W \cup \{'(',')',',','\}$ are symbols of an alphabet. A string $t \in M(F \cup \Phi, V \cup W)$ is called a term. We obtain a generalized schematic rewriting system $\overline{\Sigma}$ by writing the equations $\varphi^i(v_1,\ldots,v_{\rho_i}) = \tau^i + \Omega$ instead of $\varphi^i(v_1,\ldots,v_{\rho_i}) = \tau^i$. In this paper we always use generalized schematic rewriting systems (système schèmatique généralisé, shortly SSG) and we do not distinguish them from GRPS.

Due to technical reasons we devide the set Φ into Ψ and Φ^W: Ψ is the set of all unknown function symbols from which no terms with occurrences of local variables can be derived. $\Phi^W = \Phi \setminus \Psi$ is the complement of Ψ with respect to Φ.

Example (continued):

π_1 will be transformed to the SSG Σ^{π_1} on $M(F_1 \cup \Phi_1 V_1 \cup W_1)$

$$\Sigma^{\pi_1}: \begin{cases} \varphi^1(v_1) = \eta_2^2(\varphi^2(v_1,w^{1,1})) + \Omega_1 \\ w^{1,1} = \varphi^3(v_1) \\ \varphi^2(v_1,v_2) = id_2(w^{2,1},v_2) + \Omega_2 \\ w^{2,1} = f(q(v_1),a,s(r(v_1,v_2),v_2)) \\ \varphi^3(v_1) = f(g(v_1),a,h(v_1,\varphi^3(k(v_1)))) + \Omega_1 \end{cases}$$

$F_1 = \{a,f,g,h,id_2,k,q,r,s,\eta_2^2,\Omega_1,\Omega_2\}$, where

$\Omega_1 = \Omega$, $\Omega_2 = (\Omega,\Omega)$, $I(id_2)(\nu_1,\nu_2) = (\nu_1,\nu_2)$ for all $\nu_1,\nu_2 \in D_I$ and for all I,

$I(\eta_2^2)(\nu_1,\nu_2) = \nu_1$ for all $\nu_1,\nu_2 \in D_I$ and for all I.

$\Phi_1 = \{\varphi^1,\varphi^2,\varphi^3\}$, $\Phi_1^W = \{\varphi^1,\varphi^2\}$, $\Psi_1 = \{\varphi^3\}$

$V_1 = \{v_1,v_2\}$

$W_1 = \{w^{1,1}, w^{2,1}\}$.

Remarks:

- The notion "local variable" refers to local variables in blocks known from imperative languages.
- The interpretation of a SSG is different from that given in chapter II, as the values of the local variables $w^{i,1},\ldots,w^{i,\sigma_i}$ occurring in τ^i will be given "dynamically" in the interpretation. In order to compute the value of τ^i first of all it is necessary to compute the values of the local variables $w^{i,1},\ldots,w^{i,\sigma_i}$. Following Kott, we say, it is necessary to solve the local variables.
- Different rewritings of a symbol φ^i may differ by different values of it's local variables. So it is necessary to distinguish the equations with local variables by subscription, so that the equations may be considered as "static objects". So every equation $\varphi^i(v_1,\ldots,v_{\rho_i}) = \tau^i + \Omega$ with $\varphi^i \in \Phi^W$ resolves in a system of equations $\varphi^i_m(v_1,\ldots,v_{\rho_i}) = \tau^i_m + \Omega$, $w^{i,j}_m = \tau^{i,j}_m$, $j=1,\ldots,\sigma_i$, where $m\in\Lambda$, Λ is a set of indices and τ^i_m rsp $\tau^{i,j}_m$ is obtained from τ^i rsp $\tau^{i,j}$ by subscription of all $\varphi\in\Phi^{W^m}$ and all $w\in W$ occurring in τ^i rsp. $\tau^{i,j}$. Thus, for all SSGs we have in general an infinite system of equations which simulates the SSG. Semantics of SSGs will be defined by means of semantics of the simulating infinite system.

IV. On the semantics of SSGs

Using Kott's definition of semantics of a SSG it is impossible to give semantics for the following program, because it has mutual recursion of two procedures with value parameters and local variables.


```
π = begin integer a;
        integer procedure f(x); value x; integer x;
            begin integer y; y:= h(x);
                f:= y
            end;
        integer procedure h(z); value z; integer z;
            begin integer t; t:= f(z);
                h:= t
            end;
        read(a);
        a:= f(a)
    end
```

π will be transformed to the SSG Σ^π:

Σ^π:

$\varphi^1(v_1) = id_1(w^{1,1}) + \Omega_1$
$w^{1,1} = \varphi^2(v_1)$
$\varphi^2(v_1) = \varphi^3(v_1) + \Omega_1$ - call of f in the main program
$\varphi^3(v_1) = \varphi^4(v_1,w^{3,1}) + \Omega_1$ - declaration of f
$w^{3,1} = v_1$
$\varphi^4(v_1,v_2) = \varphi^5(v_1,v_2,w^{4,1}) + \Omega_1$ - body of f
$w^{4,1} = \varphi^6(v_2)$
$\varphi^5(v_1,v_2,v_3) = v_3 + v_3 + \Omega_1$ - f:= y
$\varphi^6(v_1) = \varphi^7(v_1) + \Omega_1$ - call of h
$\varphi^7(v_1) = \varphi^8(v_1,w^{7,1}) + \Omega_1$ - declaration of h
$w^{7,1} = v_1$
$\varphi^8(v_1,v_2) = \varphi^9(v_1,v_2,w^{8,1}) + \Omega_1$ - body of h
$w^{8,1} = \varphi^{10}(v_2)$
$\varphi^9(v_1,v_2,v_3) = v_3 + \Omega_1$ - h:= t
$\varphi^{10}(v_1) = \varphi^3(v_1) + \Omega_1$ - call of f in the body of h

$F = \{id_1, \Omega_1\}$ $\Phi = \{\varphi^1,\ldots,\varphi^{10}\}$, $V = \{v_1,v_2,v_3\}$,
$W = \{w^{1,1}, w^{3,1}, w^{4,1}, w^{7,1}, w^{8,1}\}$.

Using Kott's method to solve local variables $w^{i,j}$ belonging to φ^i it is not necessary to solve further local variables $w^{i',j'}$ with $i'<i$ because he defines an order on the unknown function symbols such that no local variable $w^{i,j}$ will depend on any local variable $w^{i',j'}$ with $i'<i$. To define such an order, however, there must be no mutual recursion as it is given in π. There must not exist a sequence of procedure calls where two procedures call themselves mutually recursively such that the transformation of the procedure in a SSG have both local variables. The transformation of a procedure has local variables if it has value parameters or if it has local variables in it's body.

This restriction was made as a consequence of the wrong assumption, that procedures can be considered as parameterized blocks.

With some elementary ideas of the copy-rule-semantics we are able to solve local variables without Kott's restriction and to define algebraic semantics for a more powerful set of SSGs and thus for a more powerful set of imperative programs. In Kott's paper the theory of the solution of local variables already affords some mathematical efforts which unfortunately are not less in our presentation.

In /Kr 82/ transformations are described which translate programs of a simple imperative block-structured programming language with assignments, conditional statements, while-loops and procedures into SSGs. This provides a method to define semantics of simple imperative programming languages without higher functionality, i.e. without procedures as values. More complex constructions with procedures were studied with algebraic methods by Damm /Da 80/, Fehr /Fe 80/ and Indermark /In 80/.

On solving local variables

Using SSGs the solvability of the local variables is Kott's main problem as it is our's. If we have shown that all local variables are solvable, we can adapt Kott's propositions to our method of solving local variables.

In our framework we have to handle with distinguished SSGs, i.e.

every unknown function symbol in Φ^W has at most one occurrence on a right side of an equation. In /Kr 82/ it is described how one generates a distinguished SSG Σ' from any SSG Σ with $L(\Sigma') = L(\Sigma)$. Thus Σ' and Σ have the same semantics /Gu 80/.

Analogously to the copy-rule-semantics /La 73/ we define a rewriting tree eb for any distinguished SSG Σ. The root of eb is labelled by $\varphi^1(v_1,\ldots,v_{\rho_1})$, i.e. the axiom of Σ. If $n \in \mathbb{N}^*$ is a node of eb, then ni, $i \in \mathbb{N}$ is a son of n, if there is a $\varphi^j \in \Phi^W$ that occures in the label of n and there are i-1 occurrences of symbols of Φ^W left from φ^j. ni is labelled by $\tau^j(t_1,\ldots,t_{\rho_j}/v_1,\ldots,v_{\rho_j})$ where $t_1,\ldots,t_{\rho_j}$ are the arguments of φ^j in the label of n. Thus a node of eb with a label without an occurrence of any symbol of Φ^W is a leaf.

Furthermore for every $w^{i,j}$ we define a rewriting tree $eb_{i,j}$. The root of $eb_{i,j}$ is labelled by $\tau^{i,j}$. The construction of all other nodes is as described above.

Example (cont.):

The rewriting tree for Σ is eb^π:

$$\begin{array}{c} \varphi^1(v_1) \\ | \\ id_1(\omega^{1,1}). \end{array}$$

The rewriting tree for $w^{3,1}$ is $eb^\pi_{3,1}$: v_1

The rewriting tree for $w^{4,1}$ is $eb^\pi_{4,1}$:

$$\begin{array}{l} \varphi^6(v_2) \\ | \\ \varphi^7(v_2) \\ | \\ \varphi^8(v_2, w^{7,1}) \\ | \\ \varphi^9(v_2, w^{7,1}, w^{8,1}) \\ | \\ w^{8,1} \end{array}$$

The reader may construct the other rewriting trees.

With these definitions all rewriting trees are regular /La 73, Kr 82/ w.r.t. relations $\sim$ in eb rsp. $\sim_{i,j}$ in $eb_{i,j}$. $\sim$ rsp. $\sim_{i,j}$ say that two different nodes are in relation if they are in the same path and if they are generated by the

same unknown function symbol.

Thus we only need finite initial subtrees tb rsp. $tb_{i,j}$ to know the whole structure of the eventually infinite rewriting trees. Due to technical reasons all symbols V,W,Ψ and F are substituted by "-". Let a node be labelled by $\tau^i(t_1,..,t_{\rho_i}/v_1,...,v_{\rho_i})$, then the unknown function symbols in Φ^W occurring in this term are not substituted by "-" iff they already occur in τ^i. Now all remaining unknown function symbols in tb rsp. tb_{ij} are indexed according to their position in tb rsp. $tb_{i,j}$. Thereby in $tb_{i,j}$ every index gets the prefix (i,j). The set of indices is $\mathbb{N}_e^*$, where $\mathbb{N}_e = \mathbb{N} \cup \{(i,j) \mid i\in\{1,...,N\},\ j\in\{1,...,\sigma_i\}\}$.

Example (cont.):

All rewriting trees eb^π and $eb^\pi_{i,j}$ are finite. Thus we know the whole structure of all of them. The rewriting trees which are modified and indexed are

eb^π $\varphi^1(-)$

$eb^\pi_{3,1}$ -

$eb^\pi_{4,1}$ $\varphi^6_{(4,1)}(-)$

$\varphi^7_{(4,1)1}(-)$

$\varphi^8_{(4,1)11}(-,-)$

$\varphi^9_{(4,1)111}(-,-,-)$

-

The reader may construct the other trees.

Now we define an infinite system of equations Σ_{ebw} associated with Σ. Again we associate rewriting trees with Σ_{ebw}, which all have the same structure as the corresponding rewriting trees of Σ.

Σ_{ebw} on $M(FU\Psi U\Phi^W_{ind}, VUW_{ind})$ is obtained from Σ by

- adding the equation $\varphi^i_n(v_1,...,v_{\rho_i}) = \tau^i_n + \Omega$, $n\in\mathbb{N}_e^*$ to Σ_{ebw} for every equation $\varphi^i(v_1,...,v_{\rho_i}) = \tau^i + \Omega$ in Σ with $\varphi^i\in\Phi^W$, where $\tau^i_n = \tau^i(\varphi_{n\ ind'(n,i)}/\varphi)(w^{i,j}_n/w^{i,j})$ for all $\varphi\in\Phi^W$, $j\in\{1,..,\sigma_i\}$;
- for all equations $w^{i,j} = \tau^{i,j}$, $j\in\{1,...,\sigma_i\}$ associated to $\varphi^i\in\Phi^W$ adding $w^{i,j}_n = \tau^{i,j}_{(n(i,j)}$ to Σ_{ebw}, where $\tau^{i,j}_{n(i,j)} = \tau^{i,j}(\varphi_{n(i,j)}/\varphi)$ for all $\varphi\in\Phi^W$.
- other equations in Σ are added to Σ_{ebw} unchanged.

$\Phi^W_{ind} = \{\varphi_n/\varphi\in\Phi^W,\ n\in\mathbb{N}_e^*\}$, $W_{ind} = \{w_n \mid w\in W,\ n\in\mathbb{N}_e^*\}$.

The partially defined mapping $ind' \mid \mathbb{N}_e^* \times \{1,...,N\} \longrightarrow \mathbb{N}$ computes the extension of the indices with the aid of the finite initial trees tb rsp. $tb_{i,j}$. Thereby the rightmost occurrence of a pair $(i,j)\in\mathbb{N}_e\setminus\mathbb{N}$ in an index n indicates in which tree $tb_{i,j}$ we have to compute the extension of n.

Example (cont.): From Σ^π we get Σ^π_{ebw}:

Σ^π_{ebw}:

$\varphi^1(v_1) = id_1(w^{1,1}) + \Omega_1$
$w^{1,1} = \varphi^2_{(1,1)}(v_1)$
$\varphi^2_{(1,1)}(v_1) = \varphi^3_{(1,1)ind'((1,1),2)}(v_1) + \Omega_1$ - Here the extensions of
$\varphi^3_n(v_1) = \varphi^4_{n1}(v_1, w^{3,1}_n) + \Omega_1$ - the indices are trivial.
$w^{3,1}_n = v_1$ - So we write them explicitly.
$\varphi^4_n(v_1,v_2) = \varphi^5_{n1}(v_1,v_2,w^{4,1}_n) + \Omega_1$
$w^{4,1}_n = \varphi^6_{n(4,1)}(v_2)$
$\varphi^5_n(v_1,v_2,v_3) = v_3 + \Omega_1$
$\varphi^6_n(v_1) = \varphi^7_{n1}(v_1) + \Omega_1$
$\varphi^7_n(v_1) = \varphi^8_{n1}(v_1, w^{7,1}_n) + \Omega_1$
$w^{7,1}_n = v_1$
$\varphi^8_n(v_1,v_2) = \varphi^9_{n1}(v_1,v_2,w^{8,1}_n) + \Omega_1$
$w^{8,1}_n = \varphi^{10}_{n(8,1)}(v_2)$
$\varphi^9_n(v_1,v_2,v_3) = v_3 + \Omega_1$
$\varphi^{10}_n(v_1) = \varphi^3_{n1}(v_1) + \Omega_1$
$n\in\mathbb{N}_e^*$

We now construct an idexed rewriting tree for Σ_{ebw} which is equal to the rewriting tree for Σ except for the indication of the symbols from Φ^W and the local variables. Therefore we call it eb too. For every local variable $w_n^{i,j} \in W_{ind}$ we construct an indexed rewriting tree $eb_{n(i,j)}$, which is equal to $eb_{i,j}$ except for the indication of symbols from Φ^W and the local variables. We have to distinguish $eb_{i,j}$ and $eb_{n(i,j)}$ carefully!

Example (cont): The indexed rewriting trees to Σ^{π}_{ebw} and to the local variables have the following form:

eb^{π}:

$$\varphi^1(v_1)$$
$$id_1(w^{1,1})$$

$eb^{\pi}_{(1,1)}$:

$$\varphi^2_{(1,1)}(v_1)$$
$$\varphi^3_{(1,1)1}(v_1)$$
$$\varphi^4_{(1,1)11}(v_1, w^{3,1}_{(1,1)1})$$
$$\varphi^5_{(1,1)111}(v_1, w^{3,1}_{(1,1)1}, w^{4,1}_{(1,1)11})$$
$$w^{4,1}_{(1,1)11}$$

$eb^{\pi}_{n(3,1)}$: v_1 , $n \in \mathbb{N}_e^*$

$eb^{\pi}_{n(4,1)}$: , $n \in \mathbb{N}_e^*$

$$\varphi^6_{n(4,1)}(v_2)$$
$$\varphi^7_{n(4,1)1}(v_2)$$
$$\varphi^8_{n(4,1)11}(v_2, w^{7,1}_{n(4,1)1})$$
$$\varphi^9_{n(4,1)111}(v_2, w^{7,1}_{n(4,1)1}, w^{8,1}_{n(4,1)11})$$
$$w^{8,1}_{n(4,1)11}$$

$eb^{\pi}_{n(7,1)}$: v_1 , $n \in \mathbb{N}_e^*$

$eb^{\pi}_{n(8,1)}$: , $n \in \mathbb{N}_e^*$

$$\varphi^{10}_{n(8,1)}(v_2)$$
$$\varphi^3_{n(8,1)1}(v_2)$$
$$\varphi^4_{n(8,1)11}(v_2, w^{3,1}_{n(8,1)1})$$
$$\varphi^5_{n(8,1)111}(v_2, w^{3,1}_{n(8,1)1}, w^{4,1}_{n(8,1)11})$$
$$w^{4,1}_{n(8,1)11}$$

Now we take an arbitrary finite initial tree of eb. For all $eb_{i,j}$ we take an arbitrary finite initial tree of $eb_{i,j}$. These trees are called eeb and $eeb_{i,j}$ respectively. The set of all these trees $\mathcal{F} = \{eeb\} \cup \bigcup_{i\in\{1,\dots,N\},\ j\in\{1,\dots,\sigma_i\}} \{eeb_{i,j}\}$ is called a forest for the SSG Σ.

The number of all nodes of eeb and all $eeb_{i,j}$ is denoted by $||\mathcal{F}||$. We define a derivation $\xrightarrow[\mathcal{F},\kappa]{}$ on Σ_{ebw} depending on a forest $\mathcal{F}$ and a monotone mapping $\kappa | \mathbb{N} \longrightarrow \mathbb{N}$ that gives us the number ob allowed open, i.e. not finished, supporting computations. – A supporting computation has to be opened, if a local variable has to be solved. – Let be $\varphi_n \in \Phi^W_{ind}$ with $n=n'(i,j)n''$ or $n=n''$, $n,n' \in \mathbb{N}_e^*$, $n'' \in \mathbb{N}^*$. If $n'' \in eeb_{i,j}$ rsp. $n'' \in eeb$ and the number of open supporting computations (i.e. the number of occurrences of elements of $\mathbb{N}_e \setminus \mathbb{N}$ in the index n) is not greater than $\kappa(||\mathcal{F}||)$, then φ_n will be expanded. Otherwise, φ_n is rewritten by Ω. The structure and the quantity of the finite initial tree $eb_{i,j}$ is transfered to $eb_{n(i,j)}$ by the derivation $\xrightarrow[\mathcal{F},\kappa]{}$, so we call such a tree $eeb_{n(i,j)}$.

Now we will define, what we mean by saying a local variable is solvable:

Let Σ_{ebw} and $\mathcal{F},\kappa$ be given, let be $w_n^{i,j} \in W_{ind}$; φ_n^i may have the arguments $t_1,\dots t_{\rho_i} \in M(F\cup\Psi\cup\Phi^W_{ind}, V\cup W_{ind})$. $w_n^{i,j}$ is called solvable iff there exists $t \in M(F\cup\Psi, V\cup W_{ind})$ with $\tau^{i,j}_{n(i,j)}(t_1,\dots,t_{\rho_i}/v_1,\dots,v_{\rho_i}) \xrightarrow[\mathcal{F},\kappa]{*} t$ and $t(lt(w)/w) \in M(F\cup\Psi, V)$ exists. A term $lt(w_n^{i,j}) = t(lt(w)/w)$ is called solution term of $w_n^{i,j}$. If we can give $lt(w_n^{i,j})$ explicitly, we say that $w_n^{i,j}$ is solved.

A solution of a local variable may depend on solutions of other local variables. Given $\mathcal{F}$ and κ the following two lemmata and the theorem show that all occurring local variables are solvable.

Lemma 1: Let Σ_{ebw}, $\mathcal{F}$ and κ be given. $\varphi_n^i \in \Phi^W_{ind}$ may have an occurrence in eeb(n) and all arguments of φ_n^i may be terms, in which no $\varphi \in \Phi^W_{ind}$ has an occurrence. Also unknown function symbols which had to be rewritten to generate eeb(n) from φ^1, may have only terms as arguments in which no $\varphi \in \Phi^W_{ind}$ has an occurrence. Then all local variables $w_n^{i,j}, j \in \{1,\dots,\sigma_i\}$ of φ_n^i are solvable.

Lemma 1 is proved by induction on the length $|n|$ of n. Induction start and induction step are proved by induction on $\kappa(||\mathcal{F}||)$.

Lemma 2: Let $n \in eeb$ be the greatest node w.r.t. to lexicographic ordering in eeb. Then for all $\varphi_n^i \in \Phi_{ind}^W$ having an occurrence in eeb(n) all local variables $w_n^{i,j}$ are solvable.

Outline of proof: We vary eeb(n) stepwise: In eeb(n) there exists a φ_n^i which satisfies all premises of Lemma 1. After rewriting φ_n^i by the term which is generated from φ_n^i by $\xrightarrow{\mathcal{F},\kappa}$, another unknown function symbol satisfies the premises of Lemma 1 or there is no other unknown function symbol of Φ_{ind}^W in eeb(n) anymore. In the last case Lemma 2 is proved.

From these two lemmata we get the

Theorem: All local variables with an index in eeb are solvable.

Outline of proof: Let $n \in$ eeb be the greatest node w.r.t. to lexiographic ordering. By Lemma 2 all local variables of all $\varphi_n^i \in \Phi_{ind}^W$ occurring in eeb(n) are solvable. Therefore we may substitute the unknown function symbol that generates eeb(n) by a term in which all occurring local variables are solvable. This term is generated from eeb(n) by $\xrightarrow{\mathcal{F},\kappa}$. We now remove n from eeb and we have a smaller tree eeb'. This can be repeated until we have a tree that only consists of it's root.

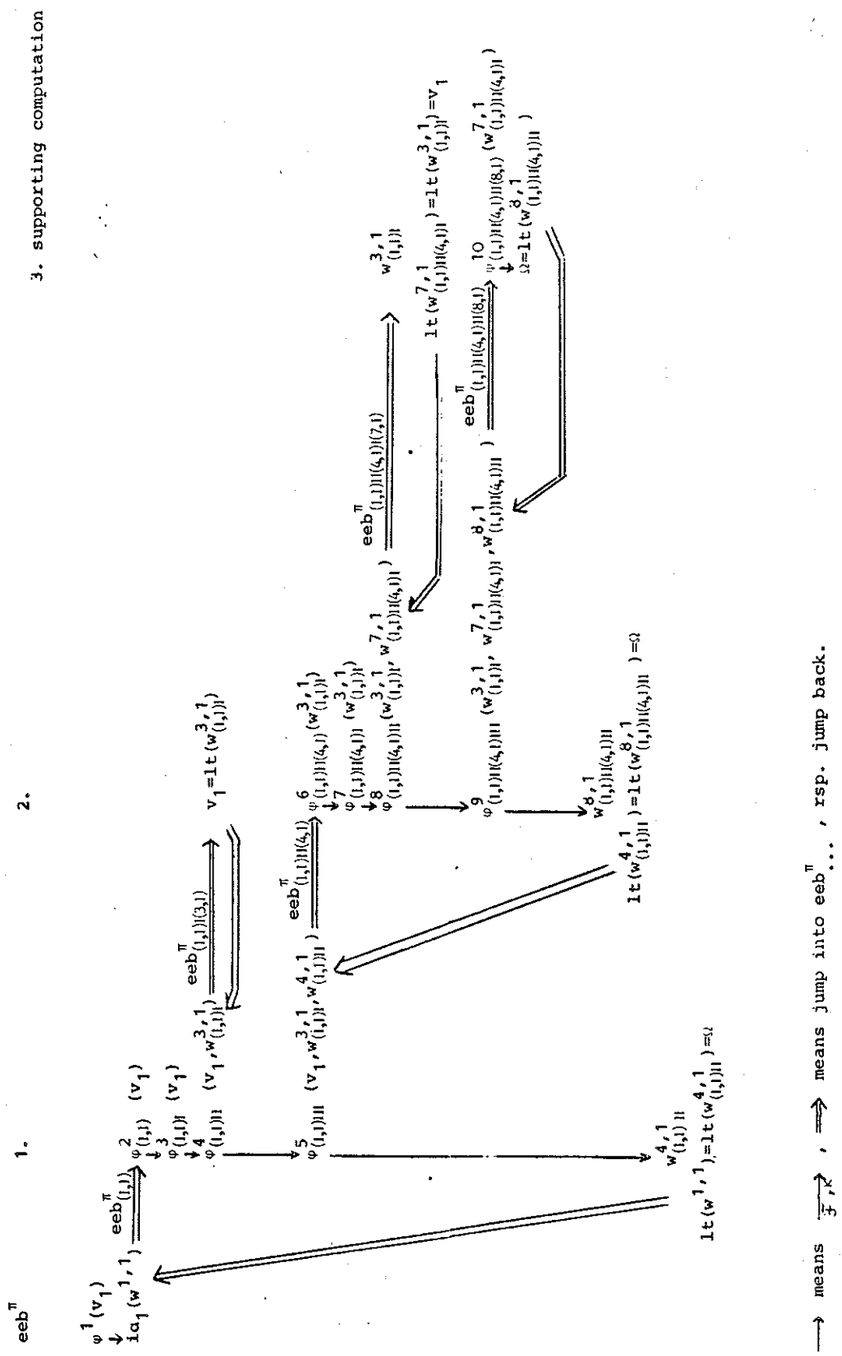
Example (cont.): Let be $eeb^\pi = eb^\pi$ and for all i,j let be $eeb^\pi_{i,j} = eb^\pi_{i,j}$. Then $\mathcal{F}^\pi$ is determined. Let κ^π be the constant mapping $\kappa^\pi(m) = 2$ for all $m \in \mathbb{N}$. We write $\mathcal{F}$ and κ instead of $\mathcal{F}^\pi$ and κ^π.

There is only $w^{1,1}$ in eeb^π. To solve $w^{1,1}$ we "jump" into the tree $eeb^\pi_{(1,1)}$. The root of $eeb_{(1,1)}$ is $\tau^{1,1}_{(1,1)}$. So the first supporting computation is opened:

$\tau^{1,1}_{(1,1)} = \varphi^2_{(1,1)}(v_1) \xrightarrow{\mathcal{F},\kappa} \varphi^3_{(1,1)1}(v_1) \xrightarrow{*}_{\mathcal{F},\kappa} w^{4,1}_{(1,1)11}$.

In $eeb^\pi_{(1,1)}$ the local variables $w^{3,1}_{(1,1)1}$ and $w^{4,1}_{(1,1)11}$ have occurrences. The solution of $w^{3,1}_{(1,1)1}$ is trivial. We jump into $eeb^\pi_{(1,1)1(3,1)}$ and get v_1 as solution term. To solve $w^{4,1}_{(1,1)11}$ we jump into

3. supporting computation



$eeb^{\pi}_{(1,1)11(4,1)}$:

$\varphi^{6}_{(1,1)11(4,1)}(w^{3,1}_{(1,1)1} \xrightarrow{\mathcal{F},\kappa} \varphi^{7}_{(1,1)11(4,1)1}(w^{3,1}_{(1,1)1}) \xrightarrow{*}_{\mathcal{F},\kappa} w^{8,1}_{(1,1)11(4,1)11}$.

In $eeb^{\pi}_{(1,1)11(4,1)}$ the local variables $w^{7,1}_{(1,1)11(4,1)1}$ and $w^{8,1}_{(1,1)11(4,1)11}$ have occurrences. The solution of $w^{7,1}_{(1,1)11(4,1)1}$ is simple: We jump into $eeb^{\pi}_{(1,1)11(4,1)1(7,1)}$ and we get $w^{3,1}_{(1,1)1}$ for which we know the solution term. Thus $lt(w^{7,1}_{(1,1)11(4,1)1}) = v_1$.

To solve $w^{8,1}_{(1,1)11(4,1)11}$, we jump into $eeb^{\pi}_{(1,1)11(4,1)11(8,1)}$, its root is $\varphi^{10}_{(1,1)11(4,1)11(8,1)}(w^{7,1}_{(1,1)11(4,1)1})$. Now the number of occurrences of elements in $\mathbb{N}_e \setminus \mathbb{N}$ is greater than $\kappa(||\mathcal{F}||) = 2$. So by definition of $\xrightarrow{\mathcal{F},\kappa}$: $\varphi^{10}_{(1,1)11(4,1)11(8,1)}(w^{7,1}_{(1,1)11(4,1)1}) \xrightarrow{\mathcal{F},\kappa} \Omega_1$. That is already the solution term of $w^{8,1}_{(1,1)11(4,1)11}$: $lt(w^{8,1}_{(1,1)11(4,1)11}) = \Omega_1$. Now we have to complete the supporting computation of $w^{4,1}_{(1,1)11}$: $lt(w^{4,1}_{(1,1)11}) = lt(w^{8,1}_{(1,1)11(4,1)11}) = \Omega_1$. Furtheron, we have to complete the supporting computation of $w^{1,1}$: $lt(w^{1,1}) = lt(w^{4,1}_{(1,1)11}) = \Omega_1$. Thus $w^{1,1}$ is solved.

Interpretation and semantics of a SSG

Let Σ be a distinguished SSG. By Σ_{eb} we assign the infinite SSG on $M(F \cup \Psi \cup \Phi^{W}_{eb}, V \cup W_{eb})$ associated with Σ. In Σ_{eb} all unknown function symbols are only indicated by indices of eb, i.e. by Σ_{eb} no supporting computations are opened.

A discrete interpretation $(I,\mathcal{F},\kappa)$ of a infinite SSG Σ_{eb} is given by

- a discrete domain, i.e. a partially ordered set $(D_I,\sqsubseteq)$ with $x \sqsubseteq y$ iff $x = \omega$ or $x = y$ where $\omega \in D_I$ is the smallest element of D_I,
- a set $F_I = \{f_I : D_I^{\rho(f)} \longrightarrow D_I \mid f \in F,\ f_I \text{ monotone w.r.t. } \sqsubseteq\}$,
- a forest $\mathcal{F}$
- a monotone mapping $\kappa \mid \mathbb{N} \longrightarrow \mathbb{N}$ with $||\mathcal{F}|| \mapsto \kappa(||\mathcal{F}||)$.

We define the following set of local variables

$W(t) = \{w \in W_{eb} \mid t \in L(\Sigma_{eb}\ \varphi^1)$ and there is a t' with $\varphi^1(v_1,\ldots,v_{\rho_1}) \xrightarrow[\Sigma_{eb}]{*} t' \xrightarrow[\Sigma_{eb}]{*} t$ and w occurs in $t'\}$.

Let $\nu \mid V \longrightarrow D_I$ be an environment, $\nu(v_k) = \nu_k$, $\nu^1 = (\nu_1,\ldots,\nu_{\rho_1})$.

A valued interpretation $(I,\mathcal{F},\kappa,\nu)$ of Σ_{eb} is a mapping

$(I,\mathcal{F},\kappa,\nu) \mid M(F,V \cup W_{eb}) \longrightarrow D_I$ defined by

$(I,\mathcal{F},\kappa,\nu)(\Omega) = \omega$

$(I,\mathcal{F},\kappa,\nu)(v_k) = \nu_k,\ k \in \{1,\ldots,\rho_1\}$

$(I,\mathcal{F},\kappa,\nu)(w) = Val_I\ lt(\omega)(\nu^1)$

$$(I,\mathcal{F},\kappa,\nu)(f(t_1,\ldots,t_{\rho(f)})) = \begin{cases} \omega, \text{ if there is a } t_m,\ m \in \{1,\ldots \rho(f)\} \text{ and a } w_n^{i,j} \in W_{eb} \text{ with } w_n^{i,j} \text{ occurs in } t_m \text{ and } n \notin eeb \\ \omega, \text{ if there is a } w_n^{i,j} \in W(f(t_1,\ldots,t_{\rho(f)})) \text{ with } n \in eeb \text{ and } Val_I\ lt(w_n^{i,j})\ (\nu^1) = \omega \\ f_I((I,\mathcal{F},\kappa,\nu)(t_1),\ldots,(I,\mathcal{F},\kappa,\nu)(t_{\rho(f)})) \text{ otherwise} \end{cases}$$

We have to explain $Val_I\ t(\nu^1)$ with $t \in M(F \cup \Psi, V)$:
Having a rewriting system Σ' that is constructed by taking all equations of the unknown function symbols of Ψ from Σ, Σ' is a rewriting system without local variables. Then we define

$Val_I\ t(\nu^1) = Val_I(\Sigma',t)(\nu^1)$.

Now the following theorem holds:

Theorem: Let Σ be a SSG, Σ_{eb} the associated infinite SSG, $(I,\mathcal{F},\kappa,\nu)$ a valued interpretation of Σ_{eb}. Then for all $t,s \in L(\Sigma_{eb},\varphi^1)$:
If $(I,\mathcal{F},\kappa,\nu)(t) \neq \omega$ and $(I,\mathcal{F},\kappa,\nu)(s) \neq \omega$, then
$(I,\mathcal{F},\kappa,\nu)(t) = (I,\mathcal{F},\kappa,\nu)(s)$.

With this theorem the semantics of an infinite SSG Σ_{eb} is defined:

The semantics of an infinite SSG Σ_{eb} is a mapping

$Val_{I,\mathcal{F},\kappa}(\Sigma_{eb},\varphi^1) \mid D_I^{\rho_1} \longrightarrow D_I$ with

$$Val_{I,\mathcal{F},\kappa}(\Sigma_{eb},\varphi^1)(\nu^1) = \begin{cases} d, & \text{if there is a } t \in L(\Sigma_{eb},\varphi^1) \text{ with } (I,\mathcal{F},\kappa,\nu)(t) = d,\ d \neq \omega \\ \omega & \text{otherwise} \end{cases}$$

To define the semantics of a SSG Σ by means of the semantics of the associated infinite SSG Σ_{eb} the following is shown.

Let Σ be a SSG, Σ_{eb} the associated infinite SSG, $\mathcal{F}$, $\mathcal{F}'$ forests, κ,κ' monotone mappings, I an interpretation. Then the following holds:

1) If $\mathcal{F} \leq \mathcal{F}'$, then $\kappa(||\mathcal{F}||) \leq \kappa(||\mathcal{F}'||)$

2) If $\kappa \leq \kappa'$, then $Val_{I,\mathcal{F},\kappa}(\Sigma_{eb},\varphi^1) \sqsubseteq Val_{I,\mathcal{F},\kappa'}(\Sigma_{eb},\varphi^1)$

3) If $\mathcal{F} \leq \mathcal{F}'$, then $Val_{I,\mathcal{F},\kappa}(\Sigma_{eb},\varphi^1) \sqsubseteq Val_{I,\mathcal{F}',\kappa}(\Sigma_{eb},\varphi^1)$,

where $\mathcal{F} \leq \mathcal{F}'$ iff eeb $\leq$ eeb' and for all i,j $eeb_{i,j} \leq eeb'_{i,j}$.

The <u>semantics of a SSG</u> Σ is a mapping $Val_I(\Sigma,\varphi^1)\ D_I^{\Omega_1} \longrightarrow D_I$ with

$$Val_I(\Sigma,\varphi^1)(\nu^1) = \begin{cases} d, \text{ if there are } \mathcal{F} \text{ and } \kappa, \text{ so that} \\ \quad Val_{I,\mathcal{F},\kappa}(\Sigma_{eb},\varphi^1)(\nu^1) = d,\ d \neq \omega \\ \omega, \text{ otherwise.} \end{cases}$$

<u>References</u>

/Da 80/ W. Damm: "The IO- and OI-hierarchies", Schriften zur Informatik und Angewandten Mathematik Nr. 41, RWTH Aachen, 1980

/Fe 80/ E. Fehr: "Lambda-calculus as control structures of programming languages", Schriften zur Informatik und Angewandten Mathematik Nr. 57, RWTH Aachen, 1980

/Gu 80/ I. Guessarian: "Algebraic Semantics", LITP Nr. 80-13, Université Paris VII, 1980

/In 80/ K. Indermark: "On rational definitions in complete algebras without rank", Schriften zur Informatik und Angewandten Mathematik Nr. 64, RWTH Aachen, 1980

/Ko 76/ L. Kott: "Approche par le magma d'un langage de programmation type ALGOL; Semantique et verification de programmes", Thèse 3eme Cycle Université Paris VII, 1976

/Ko 77/ L. Kott: "Semantique algebrique d'un langage de programmation type ALGOL", R.A.I.R.O. Vol. 11, Nr. 3, S. 237-263, 1977

/Kr 82/ M. Krause: "Algebraische Semantik für ALGOL-ähnliche Programmiersprachen", Diplomarbeit, Universität Kiel, 1982

/La 73/ H. Langmaack: "On Correct Procedure Parameter Transmission in Higher Programming Language", Acta Informatica 2, S. 110-142, 1973

/Ma 72/ Z. Manna, J. Vuillemin: "Fixpoint Approach to the Theory of Computation", CACM 15, S. 528-536, 1972

/Ma 74/ Z. Manna: "Mathematical Theory of Computation", McGraw-Hill, New York, 1974

/McC 63/ J. McCarthy: "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems, P. Braffort, D. Hirschberg (eds.), North Holland, S. 33-70, 1963

/Ni 72/ M. Nivat: "Sur l'interpretation des schémas de programme monadiques", Rapport IRIA-Laboria Nr. 1, 1972

/Ni 74/ M. Nivat: "On the Interpretation of Recursive Polyadic Schemes", Instituto Nazionale di Alta Matematica, Symposia Matematica, Vol XV, S. 255-281, 1974

/Pa 69/ D. Park: "Fixpoint Induction and Proof of Program Properties", Machine Intelligence 5, B. Meltzer, D. Michie (eds.), S. 59-78, 1969

/Sc 70/ D. Scott, C. Strachey: "Towards a Mathematical Semantics for Computer Languages", Technical Memo PRC- , Oxford University, Oxford, 1970

Nicht-deterministischer Lambda-Kalkül

Erich Meyer

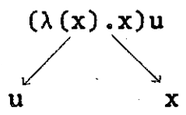
Institut für Informatik, Universität Kiel

Im klassischen Lambda-Kalkül reduzieren die Terme $(\lambda x.x)u$ zu u und $(\lambda y.x)u$ zu x. Wir wollen den Kalkül so erweitern, daß es in ihm einen Term gibt, der sowohl zu u als auch zu x reduziert.

Wir erweitern dazu die Termmenge, indem wir zulassen, daß Variable, die unmittelbar einem Lambda folgen, geklammert werden dürfen. Die Bedeutung einer geklammerten Variablen wollen wir so festlegen, daß sie einerseits wie eine ungeklammerte wirkt und andererseits wie eine Variable, von der überhaupt keine Bindung ausgeht.

Betrachten wir den Term $(\lambda(x).x)u$; er soll also einerseits die Bedeutung von $(\lambda x.x)u$ haben und andererseits die von $(\lambda y.x)u$.

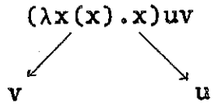
(1)



Welche der möglichen Reduktion genommen wird, soll nicht determiniert sein.

Nehmen wir den Term $(\lambda x(x).x)uv$. Er soll einerseits die Bedeutung von $(\lambda xx.x)uv$ haben, welches zu v reduziert, und andererseits die von $(\lambda xy.x)uv$, welches zu u reduziert.

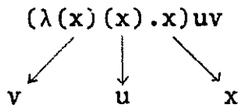
(2)



Wir können die Wirkung von $\lambda(x)$ auch so auffassen, daß $\lambda(x)$ nicht die Bindung von einem weiter außenstehenden λx oder $\lambda(x)$ an ein weiter innenstehendes x abschneidet. Eine Variable soll also im Unterschied zum klassischen Kalkül mehrfach gebunden sein können. In (2) ist x an λx gebunden, welches zur Einsetzung von u führt, und an $\lambda(x)$, wodurch v eingesetzt werden kann. (2) stellt also eine nicht-deterministische Alternative von v und u dar.

Betrachten wir den Term $(\lambda(x)(x).x)uv$, so kommt gegenüber (2) noch die Möglichkeit hinzu, für x garnichts einzusetzen, wie es schon bei (1) der Fall war.

(3)



Nach dem bisher Gesagten ist die Bedeutung eines Terms, wie $(\lambda(x).xx)u$ noch nicht vollständig bestimmt. Würden wir $\lambda(x)$ zum einen durch λx und zum anderen durch λy ersetzen, so erhielten wir die Reduktionen zu (uu) und zu (xx). Dieses entspricht aber nicht ganz unserer Intention. Wir möchten die Einsetzungen für jedes x unabhängig von den Einsetzungen für jedes andere x ausführen können, so daß wir auch zu (ux) und zu (xu) reduzieren können. Dieses erreichen wir, wenn wir von $(\lambda(x).xx)u$ zunächst zu $(\lambda(x).x)u((\lambda(x).x)u)$ übergehen. Wir benutzen also eine analoge Regel zu der im klassichen Kalkül zulässigen Ableitungsregel:

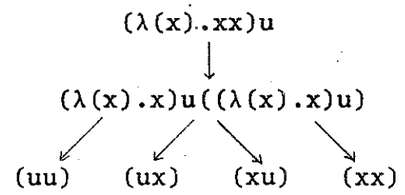
(4) $(\lambda x_1 \ldots x_n.PQ)R_1 \ldots R_n$ reduziert zu

$(\lambda x_1 \ldots x_n.P)R_1 \ldots R_n((\lambda x_1 \ldots x_n.Q)R_1 \ldots R_n)$

Als zulässig sehen wir eine Regel an, wenn sie Terme in solche der gleichen Bedeutungsklasse überführt. Wir wollen die Reduk-

tionsschritte von (λ(x).xx)u noch einmal zusammen darstellen durch

(5)



Um unsere Überlegungen verallgemeinern zu können, fehlen uns noch weitere Details. Wie sieht zum Beispiel die Reduktion von λx.(λ(x)y.xx)y aus? Hier treten zwei Schwierigkeiten auf: zum einen haben wir bisher nur reduziert, wenn für jeden Parameter ein Argument vorhanden war, zum andern würde man einen Bindungskonflikt erhalten, wenn man y für x einsetzte.

Zunächst eine Zwischenbetrachtung:

Im klassischen Kalkül muß man gebundene Variablen umbenennen können. Zum Beispiel darf der Term (λxy.x)y nicht zu λy.y reduziert werden. Normalerweise geht man zu einem Term (λxz.x)y über, den man dann zu λz.y reduzieren kann. Die Umbenennung von Variablen kann man durch eine eingeschränkte inverse η-Reduktion vermeiden, indem man von (λxy.x)y zu λz.(λxy.x)yz übergeht.
Ein Term der Form (λx1...xn.x)P1...Pn läßt sich mit den folgenden klassisch zulässigen Regeln ohne Umbenennungen reduzieren:

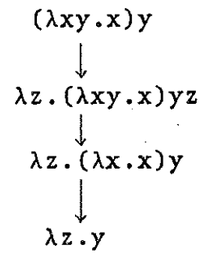
(6) $(\lambda x_1 \ldots x_n . x_n) P_1 \ldots P_n$ reduziert zu P_n

(7) $(\lambda x_1 \ldots x_n y . x) P_1 \ldots P_n P$ reduziert zu

$(\lambda x_1 \ldots x_n . x) P_1 \ldots P_n$

Wir gehen also davon aus, daß man mit Hilfe der Regeln (4), (6), (7) und der eingeschränkten inversen η-Reduktion einen klassischen Term mit Normalform zu dieser reduzieren kann. Für den Term (λxy.x)y erhält man also die folgenden Reduktionsschritte

(8)

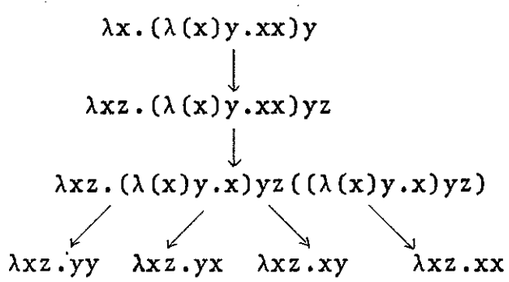


Kommen wir zurück zur Reduktion des nicht-klassischen Terms λx.(λ(x)y.xx)y .

Wenden wir das Analogen zur eingeschränkten inversen η-Reduktion an, so erhalten wir λxz.(λ(x)y.xx)yz. Diesen Term können wir analog zu (4) reduzieren und erhalten λxz.(λ(x)y.x)yz((λ(x)y.x)yz). Die Teilterme (λ(x)y.x)yz reduzieren wir analog zu (7) zuerst zu (λ(x).x)y und dann wie in (1) zu y und zu x.

Insgesamt erhalten wir

(9)



Nachdem wir nun alle Reduktionsmöglichkeiten beispielhaft angewandt haben, sollen sie nun systematisch und allgemein dargestellt werden.

Wir gehen aus von einer Menge VAR von Variablen und einer Menge KONST von Konstanten. Die Mengen sollen nicht leer sein und die Zeichen (,) und λ nicht enthalten und sie sollen disjunkt sein. Die Konstanten nehmen wir hinzu, damit wir später Einsetzungen von Termen in Terme beschreiben können. Insbesondere sollen Konstanten nicht gebunden werden können.

Definition 1

TERM ist die kleinste Menge von Termen, für die gilt:

(i) VAR + KONST $\subseteq$ TERM

(ii) seien P,Q $\in$ TERM, dann ist (PQ) $\in$ TERM

(iii) sei P $\in$ TERM und x $\in$ VAR, dann sind λxP, λ(x)P $\in$ TERM .

Um die Terme übersichtlicher schreiben zu können, führen wir einige Notationen ein; dabei werden wir Terme durch die Buchstaben P, Q, R, S und T bezeichnen und Variable und geklammerte Variable durch X, Y und Z.

Notation

Statt $\lambda X_1 \ldots \lambda X_n x$ und $\lambda X_1 \ldots \lambda X_n (PQ)$ schreiben wir
$(\lambda X_1 \ldots X_n . x)$ bzw. $(\lambda X_1 \ldots X_n . PQ)$.
Statt $(\ldots(PR_1)\ldots R_n)$ schreiben wir $(PR_1 \ldots R_n)$.
Ferner lassen wir äußere Klammerpaare fort.

Auf den Termen wollen wir nun Reduktionen definieren. Sei ρ eine solche, dann wird die ρ-Reduktion eines Termes P zu einem Term Q durch $(P,Q) \in \rho$ dargestellt, also ρ als Relation aus TERM x TERM aufgefaßt.

Definition 2

$\beta 1$ ist die Relation aus TERM x TERM mit

$$\beta 1 = \{(\ (\lambda X_1 \ldots X_n . PQ) R_1 \ldots R_n, \ (\lambda X_1 \ldots X_n . P) R_1 \ldots R_n ((\lambda X_1 \ldots X_n . Q) R_1 \ldots R_n) \ ) \mid \}$$

Die $\beta 1$-Reduktion ist analog zur Regel (4) gebildet.

Sei $(P,Q) \in \beta 1$ und seien uPv und uQv Terme, wobei also u und v Worte aus (VAR+KONST+{(,),λ})* sein müssen, dann können wir den Übergang von uPv zu uQv als eine von $\beta 1$ bewirkte Reduktion

auffassen. Um dieses allgemein formulieren zu können, definieren wir für jede Konstante a eine Einsetzungsfunktion κa mit der Wirkung, daß $\kappa a(uav,P) = uPv$ ist.

Definition 3

Sei a $\in$ KONST, dann ist
$\kappa a : \text{TERM} \times \text{TERM} \rightarrow \text{TERM}$
die Abbildung mit

(i) $\kappa a(a,S) = S$

(ii) $\kappa a(p,S) = p$, falls $a \neq p$ und $p \in$ VAR + KONST ist.

(iii) $\kappa a(\lambda X P,S) = \lambda X \kappa a(P,S)$

(iv) $\kappa a((PQ),S) = (\kappa a(P,S) \kappa a(Q,S))$

Mit Hilfe der Einsetzungsfunktionen wollen wir zu jeder Relation ρ aus TERM x TERM eine Einsetzungsrelation $\kappa\rho$ definieren.
Um beim Beispiel zu bleiben: uPv $\kappa\beta 1$-reduziert zu uQv, wenn P zu Q $\beta 1$-reduziert.

Definition 4

TERMa = {T $\in$ TERM | a kommt in T genau einmal vor}
Sei $\rho \subseteq$ TERM x TERM, dann ist
$\kappa\rho$ die durch ρ erzeugte Einsetzungsrelation aus
TERM x TERM mit
$\kappa\rho = \{(\kappa a(P,Q), \kappa a(P,R)) \mid P \in \text{TERMa}, (Q,R) \in \rho \}$.

Im klassischen Lambda-Kalkül gilt das CHURCH-ROSSER-Theorem, das besagt, daß wenn man einen Term P zu Q und R reduzieren kann, man diese zu einem Term S reduzieren kann. Wir wollen sagen, die Reduktion habe die Diamanteigenschaft.

Definition 5

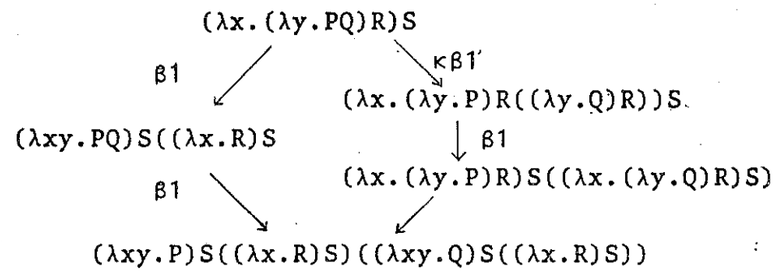
Sei SET eine Menge und $\rho \subseteq$ SET x SET eine Relation.
ρ hat die Diamanteigenschaft genau dann, wenn es zu (P,Q) und (P,R) aus ρ stets (Q,S) und (R,S) aus ρ gibt.
Ist ρ reflexiv und transitiv und hat ρ die Diamanteigenschaft, so heißt ρ ein Diamant.
ρ^* ist die reflexive und transitive Hülle von ρ.

Satz 1

$(\kappa\beta 1)^*$ ist ein Diamant.

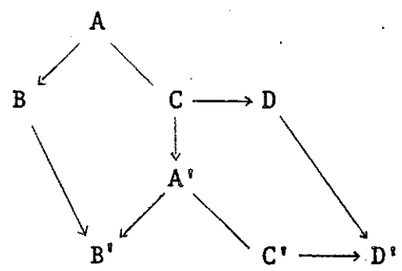
Die Komplikationen, die beim Beweis des Satzes auftreten, macht das folgende Beispiel deutlich:

Beispiel



Die Pfeile bedeuten $\beta 1$- oder $\kappa\beta 1$-Reduktionen, abgesehen vom Pfeil rechts unten, wo zwei $\beta 1$-Reduktionen parallel auf Teiltermen ausgeführt sind.

Bestehen für eine Relation ρ Fünfecke (die Terme im Beispiel bilden ein solches), so kann man im allgemeinen nicht auf die Diamanteigenschaft von ρ^* schließen, wie durch die Skizze angedeutet sei.



Um die störenden Fünfecke zu beseitigen, definiert man eine φ-Reduktion, die im allgemeinen mehrere $\beta 1$-Reduktionen zusammenfaßt.

Definition 6

Φ : TERM → TERM ist die Abbildung mit

$$\Phi(T) = \begin{cases} \Phi((\lambda X_1 \ldots X_m.P)R_1 \ldots R_m \Phi((\lambda X_1 \ldots X_m.Q)R_1 \ldots R_m) \ldots R_n) \\ \quad \text{falls } T = (\lambda X_1 \ldots X_m.PQ)R_1 \ldots R_m \ldots R_n \\ T \text{ sonst} \end{cases}$$

$\varphi \subseteq$ TERM x TERM ist die Relation mit

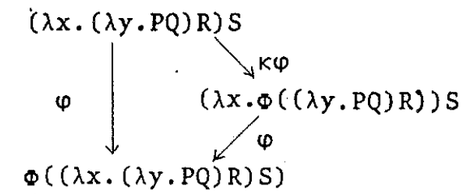
$\varphi = \{(S,T) \mid \Phi(S) = T\}$

Lemma 2

Es ist für $m < n$

$\Phi((\lambda X_1 \ldots X_m.P)R_1 \ldots R_n) = \Phi((\lambda X_1 \ldots X_m.\Phi(P))R_1 \ldots R_n)$.

Betrachten wir den Ausgangsterm des Beispiels $(\lambda x.(\lambda y.PQ)R)S$, so können wir $\Phi((\lambda x.(\lambda y.PQ)R)S)$ und $(\lambda x.\Phi((\lambda y.PQ)R))S$ bilden. Auf grund von Lemma 2 besteht dann das Diagramm



Den Beweis von Lemma 2 kann man durch vollständige Induktion über die Stufe von P führen. Die Stufe eines Terms ist die minimale Anzahl der Rekursionsschritte, die zur Berechnung des Bildes bei Φ benötigt werden, wobei in jedem Schritt alle Vorkommen von Φ einmal auszuwerten sind.

In dem Beispiel hatten wir die Reduktion von
$(\lambda x.(\lambda y.P)R)S((\lambda x.(\lambda y.Q)R)S)$ zu
$(\lambda xy.P)S((\lambda x.R)S)((\lambda xy.Q)S((\lambda x.R)S))$
als von zwei parallelen $\beta 1$-Reduktionen auf Teiltermen herrührend bezeichnet. Wir können die Reduktion durch zwei Einsetzungen in den Term $(a_1 a_2)$ darstellen:
$\kappa a_2(\kappa a_1((a_1 a_2), (\lambda x.(\lambda y.P)R)S), (\lambda x.(\lambda y.Q)R)S)$
reduziert zu
$\kappa a_2(\kappa a_1((a_1 a_2), (\lambda xy.P)S((\lambda x.R)S)), (\lambda xy.Q)S((\lambda x.R)S))$

Allgemein haben wir

Definition 7

Sei $\rho \subseteq$ TERM x TERM, dann ist $\pi\rho$ die Relation mit
$\pi\rho = \{(\kappa a_n(\ldots\kappa a_1(P,S_1),\ldots S_n), \kappa a_n(\ldots\kappa a_1(P,T_1),\ldots T_n)) \mid$
$P \in TERMa_i, (S_i,T_i) \in \rho$ für alle i $\} \cup \{(P,P)\mid\}$

Satz 3

$\pi\varphi$ hat die Diamanteigenschaft.

Zum Beweis dieses Satzes benötigt man das Lemma 2 .
Mit Hilfe von Satz 3 kann man dann Satz 1 beweisen.

Als nächstes wollen wir die eingeschränkte inverse η-Reduktion definieren. Da sie die Wirkung einer impliziten Umbenennung von Variablen hat, wollen wir sie, wie im klassischen Kalkül üblich, mit α bezeichnen.

Zuvor müssen wir noch definieren, wann eine Variable nicht frei in einem Term vorkommt. Statt "x ist nicht frei in T", werden wir $x \notin T$ schreiben.

Definition 8

Für $(x,T) \in$ VARxTERM ist $x \notin T$ genau dann, wenn eine der folgenden Bedingungen erfüllt ist.

(i) $T = (RS)$, $x \notin R$ und $x \notin S$
(ii) $T = \lambda xS$
(iii) $T = \lambda YS$, $x \neq Y$ und $x \notin S$
(iv) $T \in$ VAR+KONST und $T \neq x$.

Nach dieser Definition ist zum Beispiel $x \notin (\lambda xy.x)y$ aber nicht $x \notin (\lambda(x)y.x)y$

Definition 9

$\alpha \subseteq$ TERM x TERM ist die Relation mit
$\alpha = \{((\lambda X_1\ldots X_n Y_1\ldots Y_m.P)R_1\ldots R_n,$
$\lambda z.(\lambda X_1\ldots X_n Y_1\ldots Y_m.P)R_1\ldots R_n z)\mid$
$z \notin (\lambda X_1\ldots X_n Y_1\ldots Y_m.P)R_1\ldots R_n \}$

Es sind also $((\lambda xy.x)y, \lambda x.(\lambda xy.x)yx)$ und $((\lambda(x)y.x)y, \lambda z.(\lambda(x)y.x)yz)$ aus α, $((\lambda(x)y.x)y, \lambda x.(\lambda(x)y.x)yx)$ aber nicht.

Satz 4

$\kappa\alpha$ und $\pi\alpha$ haben die Diamanteigenschaft.

Satz 5

$(\kappa\alpha \cup \kappa\beta 1)^*$ ist ein Diamant.

Der Beweis von Satz 4 ist leicht zu führen. Zum Beweis von Satz 5 braucht man den

Satz 6

$(\pi\alpha \cup \pi\varphi)^*$ ist ein Diamant.

Zum Beweis dieses Satzes braucht man das

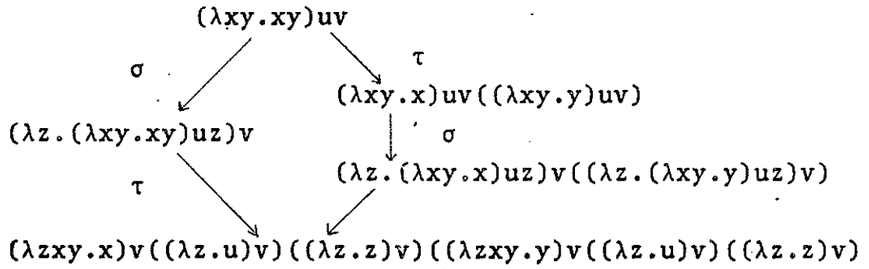
Lemma 7

Seien $(P,Q) \in \pi\alpha$ und $(P,R) \in \pi\varphi$,
dann gibt es
$(Q,S) \in \pi\varphi$, $(R,T) \in \pi\alpha$ und $(T,S) \in \pi\varphi$.

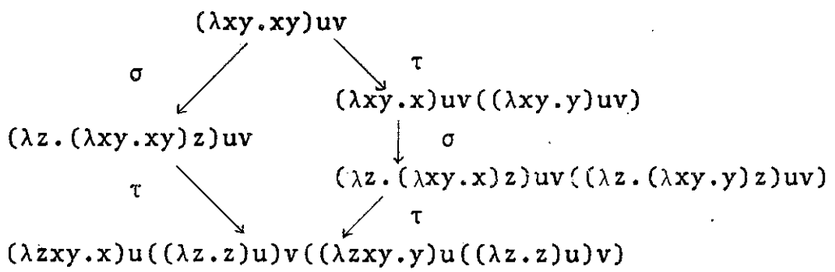
Dieses Lemma behauptet also das Bestehen spezieller Fünfecke. Zusammen mit der Diamanteigenschaft von $\pi\alpha$ und von $\pi\varphi$ kann man dann diese von $(\pi\alpha \cup \pi\varphi)^*$ beweisen.

Lemma 7 demonstrieren wir an zwei Beispielen:

(1)



(2)



dabei sind $\sigma \in \pi\alpha$ und $\tau \in \pi\varphi$.

Wir benötigen noch die Reduktionen von Termen der Form $(\lambda X_1 \ldots X_n.x)R_1 \ldots R_n$. Erst durch diese wird der Kalkül zu einem Einsetzungskalkül, denn durch die β1-Reduktionen wurde nur eine Verteilung oder Weitergabe von Parametern und Argumenten auf die Teilterme vorgenommen.

Definition 10

β2 und β3 sind die Relationen aus TERM x TERM mit

$$\beta 2 = \{ ((\lambda X_1 \ldots X_n X.x)R_1 \ldots R_n R , R) \mid 0 < n , X = x \text{ oder } X = (x) \}$$

$$\beta 3 = \{ ((\lambda X_1 \ldots X_n X.x)R_1 \ldots R_n R , (\lambda X_1 \ldots X_n.x)R_1 \ldots R_n) \mid 0 < n , X = y \text{ oder } X = (x) \}$$

Im folgenden bezeichnen wir mit TERM' die Menge der Terme, die keine geklammerten Variablen enthalten und entsprechend definieren wir

$\beta i' = \beta i \cap \text{TERM}' \times \text{TERM}'$ und $\alpha' = \alpha \cap \text{TERM}' \times \text{TERM}'$.

Betrachten wir die Reduktionen nur auf den klassischen Termen, so erhalten wir den

Satz 8 (CHURCH-ROSSER-Theorem)

$(\kappa\alpha' \cup \kappa\beta 1' \cup \kappa\beta 2' \cup \kappa\beta 3')^*$ ist ein Diamant.

Stellen wir noch einmal zusammen, was wir insgesamt zur Definition des Kalküls benötigt haben:
Definition 1 ergab die Menge der Terme. Die Einsetzungsrelation war durch Definition 4 gegeben; dazu benötigen wir die Einsetzungsfunktionen $\kappa\alpha$ aus Definition 3. Die β-Reduktionen waren durch Definition 2 und Definition 10 gegeben. Die α-Reduktion war durch Definition 9 gegeben; dabei brauchten wir die Relation "kommt nicht frei vor in", die durch Definition 8 gegeben war.

Zum Schluß noch einige Bemerkungen:

Die Reduktionsstrategie, die wir verfolgten, bestand darin, nur Terme der Form $(\lambda X_1 \ldots X_n.P)R_1 \ldots R_n$ zu reduzieren. Um dieses zu erreichen, wurde die α-Reduktion neu definiert. Sie gestattet einerseits die Umbenennung von Variablen, soweit sie nicht geklammert sind, und andererseits die Ergänzung fehlender Argumente, also den Übergang von $(\lambda X_1 \ldots X_n.P)R_1 \ldots R_m$ zu $\lambda z_1 \ldots z_l.(\lambda X_1 \ldots X_n.P)R_1 \ldots R_m z_1 \ldots z_l$ mit $l = n - m$.
Es stellte sich heraus, daß die Schwierigkeiten beim Beweis des CHURCH-ROSSER-Theorems für die klassischen Terme nicht von den Einsetzungen also den β2- und β3-Reduktionen herrührten, sondern bereits beim Verteilen der Parameter und Argumente, wie es die β1-Reduktion bewirkt, auftreten.
In den Beweis der Diamanteigenschaft von $(\kappa\alpha \cup \kappa\beta 1)^*$ geht nicht ein, ob die Variablen total oder partiell gebunden sind, also ob in den Termen der Form $(\lambda X_1 \ldots X_n.P)$ die X_i Variable oder geklammerte Variable sind.

Diese Erkenntnis hatte ursprünglich zu der Idee geführt, den klassischen Kalkül zu einem nicht-deterministischen in der dargestellten Form zu erweitern.

DRAFT VERSION

AN ALGEBRAIC SEMANTICS FOR DATA-DRIVEN (BUSY) AND DEMAND-DRIVEN (LAZY) EVALUATION AND ITS APPLICATION TO A FUNCTIONAL LANGUAGE +)

Bernhard Möller
Institut für Informatik der TU München
Arcisstr. 21, D-8000 München 2, Fed. Rep. Germany

1. INTRODUCTION

In recent years ideas on "non-conventional" machines and languages have become more and more important. The aims are, on the one hand, to move towards a more flexible use of parallelism, and, on the other hand, to allow "unbounded objects" in such a way that certain problems can be solved more easily in terms of these objects while the solution remains algorithmic. These two concepts are closely related to the techniques of data-driven (or busy) and demand-driven (or lazy) evaluation (see e.g. /Treleaven et al. 82/ for these notions).

The present paper shows how the notion of an equationally defined continuous algebra can be employed for deriving these two forms of operational semantics for recursively defined objects from the least-fixpoint-semantics of recursive definitions.

The general plan used is similar to the well-known techniques for evaluating recursively defined funtions, viz. repeatedly replacing certain occurrences of the recursively defined identifiers by their definitions and subsequent simplification.

By using equationally defined continuous algebras whose equations can immediately be used as term rewriting rules, the simplification rules are correct by construction. Moreover, a syntactic criterion (safety) guarantees that the rewriting system obtained is confluent and noetherian so that finding normal forms is uncritical. The algebraic approach permits a coherent presentation of mathematical semantics (satisfaction of equations in an algebra) and operational semantics (deductions using equations as term rewrite rules). Finally, the algebraic approach allows a uniform treatment of data and control structures:

+) This work was partially sponsored by the Sonderforschungsbereich 49, Programmiertechnik, Munich, Fed. Rep. Germany

This is shown by a safe equational specification of a functional programming language à la /Backus 78/; the techniques developed in the paper provide a mathematical as well as lazy and busy operational semantics for it. Detailed proofs of the theorems and lemmas are contained in /Möller 82/.

I am grateful to F. L. Bauer and M. Broy for a number of valuable remarks.

2. CONTINUOUS ALGEBRAS

We call a partially ordered set N complete if it has a least element and every directed subset $M \subseteq N$ has a supremum $\bigsqcup M$ in N. If N and P are partially ordered, a mapping $f : N \to P$ is called monotonic if for $x, y \in N$ $x \leq y$ implies $f(x) \leq f(y)$. If N, P are complete, f is called continuous if for all directed $M \subseteq N$ $\bigsqcup f(M)$ exists and $f(\bigsqcup M) = \bigsqcup f(M)$. A continuous mapping is monotonic. We quote the well-known fixpoint theorem for continuous mappings (see e.g. /Manna 74/):

Theorem 0: Let M be complete and $f : M \to M$ continuous.

(1) (Tarski) f has a least fixpoint.

(2) (Kleene) The least fixpoint of f is $\bigsqcup\{f^i(\bot) : i \in \mathbb{N}\}$ where $\bot$ is the least element of M and $f^0(\bot) := \bot$, $f^{i+1}(\bot) := f(f^i(\bot))$.

Given a family $(M_i)_{i\in I}$ of complete sets, the direct product $P := \prod_{i\in I} M_i$ consists of all mappings $f : I \to \bigcup_{i\in I} M_i$ such that $f(i) \in M_i$ for $i \in I$. It is partially ordered by $f \leq g$ iff $f(i) \leq g(i)$ for all $i \in I$. Under this ordering, the direct product is again complete and for a directed set $G \subseteq P$ $(\bigsqcup G)(i) = \bigsqcup\{g(i) : g \in G\}$. If $I = \{1, \ldots, n\}$ we also write $M_1 \times \ldots \times M_n$ for P.

A signature (cf. /ADJ 77/) $\Sigma = (S,F)$ consists of a set S of sort symbols and a set F of operation symbols. Each $f \in F$ has a functionality $s_1 \times \ldots \times s_n \to s$ with $s_1, \ldots, s_n, s \in S$. If $n=0$ then f is called a constant symbol.

Example: $\Sigma_0 = (S_0, F_0)$ where $S = \{\underline{nat}, \underline{sequ}\}$ and F consists of

$0 : \to \underline{nat}$, $\quad succ : \underline{nat} \to \underline{nat}$,

$() : \to \underline{sequ}$, $\quad add : \underline{nat} \times \underline{sequ} \to \underline{sequ}$.

Given a signature Σ, an ordered Σ-algebra A consists of a family $(s^A)_{s\in S}$ of nonempty ordered carrier sets s^A and a family $(f^A)_{f\in F}$ of operations such that f^A is a monotonic mapping from $s^A \times \ldots \times s_n^A \to s^A$. A is called continuous if its carrier sets are complete and its operations are continuous. Homomorphisms between Σ-algebras will not be needed in this paper.

Every ordered Σ-algebra A may be embedded into a continuous Σ-algebra A^∞. The construction used is the completion by ideals (cf. e.g. /Vuillemin 75/, /Nivat 76/): wlog. we assume that all carrier sets of A have least elements. An ideal of a partially ordered set is a directed subset of it which together with an element contains all smaller ones. Now the set s^{A^∞} consists of all ideals of s^A and for ideals $I_1, \ldots, I_n$ $f^{A^\infty}(I_1, \ldots, I_n)$ is defined as the least ideal containing $f^A(I_1, \ldots, I_n)$. It is straightforward to prove that A^∞ is indeed a continuous Σ-algebra. A is embedded into A^∞ by assigning to each element the smallest ideal containing it.

A^∞ has the property of being inductive (cf. e.g. /ADJ 78/): every element of a carrier set is finitely approximable, i.e. it is the supremum of a directed set of finite elements. Here, an element x of an ordered set N is called finite if for every directed set $D \subseteq N$, $x \leq \bigsqcup D$ implies $x \leq z$ for some $z \in D$. The non-finite elements in an inductive algebra are called infinite elements or limit points.

3. TERMS, VALUATIONS, INTERPRETATION

Let $\Sigma = (S, F)$ be a signature and $X = (X_s)_{s\in S}$ a family of sets of variables. Furthermore, let for each $s \in S$ $\bot_s$ be a constant symbol of functionality $\to s$ not contained in F. Then we define a family $(W\Sigma(X)_s)_{s\in S}$ as the least family of sets (wrt. inclusion) which satisfies

(1) $X_s \subseteq W\Sigma(X)_s$ for all $s \in S$

(2) $\bot_s \in W\Sigma(X)_s$ for all $s \in S$

(3) If $f \in F$ is an operation symbol of functionality $s_1 \times \ldots \times s_n \to s$ and $t_i \in W\Sigma(X)_{s_i}$ $(i=1, \ldots, n)$ then $f(t_1, \ldots, t_n) \in W\Sigma(X)_s$.

The elements of the $W\Sigma(X)_s$ are called Σ-terms (with free variables) of sort s.

Let $\underline{\sqsubset} = (\underline{\sqsubset}_s)_{s\in S}$ be the least (wrt. inclusion) family of partial orderings on the $W\Sigma(X)_s$ satisfying

(1) $\bot_s \underline{\sqsubset}_s t$ for all $t \in W\Sigma(X)_s$

(2) If $t_i \underline{\sqsubset}_{s_i} t'_i$ $(i=1, \ldots, n)$ then $f(t_1, \ldots, t_n) \underline{\sqsubset}_s f(t'_1, \ldots, t'_n)$

for an operation symbol f of functionality $s_1 \times \ldots \times s_n \to s$.

We construct an ordered Σ-algebra $W\Sigma(X)$ by defining $s^{W\Sigma(X)} := W\Sigma(X)_s$ ordered by $\underline{\sqsubset}_s$ and $f^{W\Sigma(X)} : (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n)$. $W\Sigma(X)$ is called the Σ-term-algebra over X (cf. the notion of the free ordered magma in /Nivat 75/ or $T\Sigma(X)$ in /ADJ77/). For $W\Sigma(\emptyset)$ we simply write $W\Sigma$.

The completion $W\Sigma(X)^\infty$ of $W\Sigma(X)$ can be interpreted as the algebra of finite and infinite Σ-terms with free variables (cf. the free complete magma in /Nivat 75/ or $C T\Sigma(X)$ in /ADJ 77/).

A valuation of X in a Σ-algebra A is a mapping $v : \prod_{s\in S} X_s \to \prod_{s\in S} s^A$ such that $v(x) \in s^A$ for $x \in X_s$. Let V_A be the set of valuations of X in A. Because V_A is a direct product of carriers of A we have

Lemma 1: If A is continuous then V_A is complete.

The interpretation $t[v]$ of a Σ-term t with respect to a valuation v of X in a Σ-algebra A is defined by

(1) If t is a variable x then $t[v] := v(x)$.

(2) If t is of the form $f(t_1, \ldots, t_n)$ then $t[v] := f^A(t_1[v], \ldots, t_n[v])$.

4. TERMS OVER RECURSIVELY DEFINED OBJECTS AND THEIR MATHEMATICAL SEMANTICS

A system of recursion equations is a valuation e of X in $W\Sigma(X)$. Over a Σ-algebra A e defines a valuation transformer $e_A : V_A \to V_A$ by $e_A(v)(x) := e(x)[v]$.

Example: Let e be the system $x \mapsto add(0, x)$ over Σ_o. It will be interpreted as the recursion equation sequ $x = add(0, x)$. For an arbitrary valuation v in a Σ_o-algebra A we have $e_A(v) : x \mapsto add^A(0^A, v(x))$. Over $W\Sigma_o(X)$ we may take e itself for v and obtain $e_{W\Sigma(X)}(e) : x \mapsto add(0, add(0, x))$, i.e. the result of "unfolding" x once in e. □

Lemma 2: If A is continuous then e_A is continuous.

Therefore, if A is continuous, by (1) of theorem 0 e_A has a least fixpoint $|e_A|$. We call $|e_A|$ the solution of e in A (cf. /ADJ 77/).

Example: In $W\Sigma_o(X)^\infty$ the system e from the previous example has as its solution the ideal generated by

$$\{\bot, add(0, \bot), add(0, add(0, \bot)) \ldots\}$$

which may be interpreted as the infinite term $add(0, add(0, add(0, \ldots)))$. □

For a term t from $W\Sigma(X)$ and a system e of recursion equations we call the pair $r = (e, t)$ a representation and $r^A := t[|e_A|]$ the object represented by r in A.

Whereas this defines the mathematical semantics of terms over recursively defined objects completely, we are now going to develop an operational semantics for such terms from Kleene's approximation sequence as given in (2) of theorem 0.

Let e be a system of recursion equations over $W\Sigma(X)$, A a continuous Σ-algebra, and $\Omega^A \in V_A$ the valuation which assigns the least element of s^A to every $x \in X_s$. This valuation is used to interpret terms with (not yet unfolded) variables: forming $t[\Omega^A]$ for some term $t \in W\Sigma(X)$ means interpreting t such that the variables are considered as carrying no information.

For a term $t \in W\Sigma(X)$ and a system e of recurision equations define $\hat{t}_o := t$, $\hat{t}_{n+1} := \hat{t}_n[e]$; i.e. the $\hat{t}_i$ evolve from t by repeatedly unfolding the definitions of the variables as specified by e.

Lemma 3: $t[|e_A|] = \bigsqcup \{\hat{t}_n[\Omega^A] : n \in \mathbb{N}\}$, i.e. the object represented by (e, t) is the supremum of the interpretations of the $\hat{t}_n$.

Proof: By induction on n one shows that $t[e_A^n(v)] = \hat{t}_n[v]$ for all valuations v. Now the claim follows from the fact that $|e_A| = \bigsqcup \{e_A^n[\Omega^A] : n \in \mathbb{N}\}$ ((2) of theorem 0) and lemma 2. □

Thus, a first operational semantics for a representation $r_n = (e, \hat{t}_n)$ is given by Kleene's approximation process:

(1) (Interpretation) Form $\hat{t}_n[\Omega^A] =: u_n$.

(2) (Unfolding) Replace in $\hat{t}_n$ simultaneously all variables by their definitions according to e, i.e. form $\hat{t}_{n+1} := \hat{t}_n[e]$, and apply the process to $r_{n+1} := (e, \hat{t}_{n+1})$.

Then $||\{u_m : m \geq n\} = r^A$, the object represented by r in A.

Example: The representation (sequ x = add(0, x), x) over Σ represents the object add(0, add(0, add(0, ...))) of $W\Sigma^\infty$. The t_n and u_n which evolve during Kleene's approximation process are

n	$\hat{t}_n$	u_n
0	x	$\perp$
1	add(0, x)	add(0, $\perp$)
2	add(0, add(0, x))	add(0, add(0, $\perp$))
.	.	.
.	.	.
.	.	.

The u_n are exactly the terms generating the ideal which is the solution of the recursion equation. □

If the carrier set of the sort of $\hat{t}_n$ in A is a flat domain, i.e. if all chains in it have at most two elements, then by the continuity of the operations involved there is a $\hat{t}_k$, $k \geq n$, such that $\hat{t}_k[\Omega^A] = \hat{t}_n[|e_A|]$ and the process can stop as soon as a maximal element is reached.

Since in the process all variables are unfolded, this method is also called full computation rule (cf. /Vuillemin 74/, /Manna 74/; see also the Herbrand-Kleene-Machine in /Bauer, Wössner 82/).

This operational semantics still is somewhat vague as the interpretation step (1) in general is not "algorithmic". In the next section a class of algebras is defined for which this step can be made effective.

5. OPERATIVE ALGEBRAS

An equation over a signature Σ is a pair (t_1, t_2) of $\Sigma(X)$-terms of the same sort; we also write $t_1 = t_2$. A Σ-algebra A satisfies the equation $t_1 = t_2$ if for all valuations $v \in V_A$ $t_1[v] = t_2[v]$.

Call a family $\leq = (\leq_s)_{s \in S}$ of quasiorderings on the carrier sets of $W\Sigma(X)$

Σ-compatible if $\Gamma \subseteq \leq$ and the operations of $W\Sigma(X)$ are monotonic wrt. $\leq$, too. For a set E of equations let $\leq_E$ be the least (wrt. inclusion) Σ-compatible family of quasiorderings on $W\Sigma(X)$ such that for all equations $(t_1, t_2) \in E$ and all valuations $v \in V_{W\Sigma(X)}$ $t_1[v] \leq_E t_2[v]$ and $t_2[v] \leq_E t_1[v]$.

Theorem 1 (cf. e.g. /Bloom 76/, /Courcelle, Nivat 76/):

(1) The quotient algebra $W\Sigma(X)/\leq_E$ of $W\Sigma(X)$ by the congruence induced by $\leq_E$ satisfies E.

(2) If an ordered Σ-algebra A satisfies E, so does A^∞.

Under certain restrictions the equation defining $W\Sigma(X)/\leq_E$ can be used as term rewriting rules. This will allow to make the interpretation step in the approximation process effective.

The idea is that the set of operations is partitioned into "constructors" which suffice for generating all elements of $W\Sigma(X)/\leq_E$, and "extensions" which are defined essentially by primitive recursion over the constructors. This is related to the criterion for sufficient completeness given in /Guttag 75/. We require that every extension has certain "critical arguments" which control the primitive recursion. They will allow a precise treatment of lazy evaluation. Let us now formalize these notions.

Let $\Sigma = (S, F)$ be a signature and $C, Z \subseteq F$ form a partition of F. The operation symbols in C are called constructors, those in Z extensions. We write C also for the subsignature (S, C) of Σ. We want to impose conditions on a set E of Σ-equations under which every variable-free Σ-term can be rewritten into a pure (also variable-free) C-Term unsing E.

A ZC-combination is a term of the form $z(t_1, \ldots, t_n)$ with an extension $z \in Z$ and constructor terms $t_i \in WC(X)$, in which every variable occurs at most once.

Now let E consist only of equations (t_1, t_2) in which the t_1 is a ZC-combination. Let E be unambiguous, i.e. no two left-hand-sides in E may be identical up to a consistent renaming of the variables. For every $z \in Z$ let the number of equations whose left-hand-side begins with z be finite. Then $||z||_j$ denotes the maximal height of terms occurring as the j-th argument of z in a left-hand-side in E (the height of variables is 0). If $||z||_j > 0$ the j-th argument of z is called critical. E is called C-complete for the j-th argument of z if in all j-th arguments of z in left-

hand-sides of E variables occur only at nesting depth $||z||_j + 1$ and every variable-free C-term of height $\leq ||z||_j$ occurs as j-th argument of z in some left-hand-side of E.

An equation $z(t_1, \ldots, t_n) = t \in E$ is called reducing if no other extension than z occurs in t, all variables of t also occur in the left-hand-side, and, if z actually occurs in t, every critical argument of z in t is a proper subterm of some critical t_j.

E is called safe if all its equations are reducing, if for all $z \in Z$ E is C-complete for all arguments of z, and if it is monotonic, i.e. if for equations $t_1 = t_2$, $u_1 = u_2 \in E$, $t_1 \sqsubseteq u_1$ (possibly after consistent renaming) implies $t_1 \sqsubseteq u_2$.

For a safe set E of equations, C-completeness means that the left-hand-sides in E provide a complete case analysis; together with the unambiguity it also implies that at most one equation can be "applied" at any "place" in a given term. For Σ-terms t_1, t_2 we say that $t_1 \underset{E}{\Rightarrow} t_2$ if there is a context K, an equation $(u_1, u_2) \in E$ and a valuation v of X in $W\Sigma(X)$ such that $t_i = K[u_i[v]]$ $(i=1, 2)$.

Theorem 2 For safe E, $\overset{*}{\underset{E}{\Rightarrow}}$ is confluent and noetherian. Therefore every term $t \in W\Sigma(X)$ has a unique normal form $NF[t]$ with respect to $\overset{*}{\underset{E}{\Rightarrow}}$ and there are no nonterminating computations under $\overset{*}{\underset{E}{\Rightarrow}}$.

Theorem 3 For safe E the carrier sets of $W\Sigma/\leq_E$ are isomorphic to those of WC. This means that WC (WC^∞) can be extended to an ordered (continuous) Σ-algebra satisfying E by interpreting the extensions suitably.

For the proofs see /Möller 82/.

Thus, interpreting a Σ-term in the algebra WC means reducing it to its normal form. By theorem 2 this process is effective. Therefore, for safe E, we call $W\Sigma/\leq_E$ an operative algebra. More generally, $W\Sigma/\leq_E$ is operative, if there is a sequence $\Sigma_0 \subseteq \Sigma \subseteq \ldots$ of signatures with $\bigcup_{i\in N} \Sigma_i = \Sigma$, and a sequence $(E_i)_{i\in N}$ with $\bigcup_{i\in N} E_i = E$, such that E_i is safe w.r.t. $\Sigma_i \sim \Sigma_{i-1}$ as extensions and Σ_{i-1} as constructors.

Example: If we enrich Σ_o by the extensions

pred : nat → nat, head : sequ → nat, tail : sequ → sequ

to Σ_1, the equations

pred(⊥) = ⊥,
pred(0) = ⊥,
pred(succ(x)) = x,

head(⊥) = ⊥,
head(()) = ⊥,
head(add(x, s)) = x,

tail(⊥) = ⊥,
tail(()) = ⊥,
tail(add(x, s)) = s

are safe wrt. the operation symbols of Σ_o as constructors. □

Over an operative algebra Kleene's approximation process is fully "algorithmic"; note, however, that in the case of a carrier set with limit points nontermination may be necessary for the correctness of the operational semantics. In this case one may view the process as producing more and more output (the sequence of the u_n, for which $u_1 \sqsubseteq u_2 \sqsubseteq u_3 \sqsubseteq \ldots$ holds) which approximates the exact value "to any degree of precision desired".

The "language" of (terms over) recursive definitions over an operative algebra may be compared with the "language" of partial recursive functions of the natural numbers: the simplification rules $(\overset{*}{\underset{E}{\Rightarrow}})$ of the algebra correspond to primitive recursion, whereas taking fixpoints of recursion equations (the fixpoint operator) is more related to the μ-operator.

6. DATA-DRIVEN AND DEMAND-DRIVEN EVALUATION

In the form of Kleene's process considered so far, in every interpretation step the normal form is computed completely anew. This section aims at avoiding these repeated computations. We exploit the fact that - by construction - interpreting and term rewriting are compatible, and simplify the terms $\hat{t}_n$ before the interpretation step proper. This saves rewriting work in later interpretation steps. Ideally, the simplification eliminates all variables so that unfolding becomes unnecessary and the process can terminate.

First we consider a method which strives for "maximal increase in information". It therefore reduces the considered term as far as possible and afterwards un-

folds all variables. It can therefore be called busy evaluation /Broy 80/, or, since it starts reducing terms as soon as sufficient information about the arguments of the extensions in the term is available, also data-driven evaluation (see e.g. /Treleaven et al. 82/).

For a given representation $r_n = (e, t_n)$ it works as follows:

(0) (Simplification) Determine the normal form $t'_n := NF[t_n]$.

(1) (Interpretation) Interpret the resulting term, i.e. form $u_n := NF[t'_n[\Omega^{W\Sigma(X)}]]$.

(2) (Unfolding) Unfold the recursive definitions in t'_n, i.e. form $t_{n+1} := t'_n[e]$, and apply the process to the new representation $r_{n+1} := (e, t_{n+1})$.

We give two criteria for terminating the process:

(0a) If t'_n does not contain variables, $u_n = r_n^{WC^\infty}$ and the process may stop.

(1a) If u_n is maximal in the respective carrier set (i.e. if it does not contain $\perp$) then $u_n = r_n^{WC^\infty}$ and the process may stop.

The correctness of this operational semantics is stated in

Theorem 4 (1) For the u_m evolving during data-driven evaluation of r_n and the object $r_n^{WC^\infty}$ represented by r_n in WC^∞ one has $r_n^{WC^\infty} = \bigsqcup\{u_m : m \geq n\}$.

(2) If the carrier set of the sort r_n is flat and neither of the termination criteria is ever satisfied then $r_n^{WC^\infty} = \perp$.

Second, we consider a method which strives to avoid term manipulations which are irrelevant for the further progress of computation. Because the applicability of equations of a safe set E is determined by the critical arguments of the extensions, this method avoids reducing terms which "at the moment" are not critical (they may, however, become critical by subsequent rewriting), and unfolds only those variables which "at the moment" prevent the further evaluation of the critical arguments. Thus this method simplifies only as far as necessary and unfolds as little as possible; it can therefore be called lazy evaluation (cf. /Henderson, Morris 76/, /Friedman, Wise 76/, /Bauer 79/) or demand-driven evaluation (cf. /Treleaven et al. 82/). For its precise specification we need two further notions:

For a term $t \in W\Sigma(X)$ the set $K(t)$ of critical variables is given by

(1) $K(x) := \emptyset$ for a variable $x \in X$

(2) $K(f(t_1, \ldots, t_n)) := \bigcup_{j \in Mf} K(t_j)$ where Mf is the set of all j such that an extension occurs in t_j, if f is a constructor, and the set of critical argument indices, if f is an extension.

The normal form $KNF[t]$ of t wrt. the critical arguments is defined by

(1) $KNF[x] := x$ for $x \in X$

(2) For a term $t = f(t_1, \ldots, t_n)$, $KNF[t] := f(KNF[t_1], \ldots, KNF[t_n])$ if f is a constructor. If f is an extension, we set $\overline{t} := f(\overline{t}_1, \ldots, \overline{t}_n)$ where $\overline{t}_j := KNF[t_j]$ if t_j is critical for f and $\overline{t}_j := t_j$ otherwise. If there is an equation $(u_1, u_2) \in E$ such that $u_1[v] = \overline{t}$ for some valuation v (by the safety of E there is at most one such equation) then $KNF[t] := KNF[u_2[v]]$, otherwise $KNF[t] := \overline{t}$.

In general, $KNF[t] \neq NF[t]$, however one has always $t \overset{*}{\underset{E}{\Rightarrow}} KNF[t] \overset{*}{\underset{E}{\Rightarrow}} NF[t]$.

Finally, for the method of demand-driven evaluation it is important not to neglect the noncritical variables completely: the operations of a continuous algebra in general are not sequential in the sense of /Vuillemin 74/, so that the evaluations of their arguments have to be advanced "sufficiently uniformly", "in a fair manner in parallel".

Now for a represenation $r_n = (e, t_n)$ the process of demand-driven evaluation works as follows:

(A) (Reduction of extensions) Determine an arbitrary natural number k_n, set $v_{no} := t_n$, $l := k_n$, and apply (B) to the representation $\overline{r}_n := (e, v_{no})$ and l.

(B) If $l=0$, apply step (C). Otherwise

(B1) (Simplification) For the given representation $\overline{r}_{nm} = (e, v_{nm})$ determine the normal form $v'_{nm} := KNF[v_{nm}]$ with respect to the critical variables.

(B2) (Interpretation) Form $u_{nm} := NF[v'_{nm}[\Omega^{WC}]]$.

(B3) (Unfolding) Determine the set $K(v'_{nm})$ of critical variables and unfold them in v'_{nm}, i.e. form $v_{nm+1} := v'_{nm}[e_{nm}]$ where

$e_{nm}(x) := e(x)$ if $x \in K(v'_{nm})$ and $e_{nm}(x) := x$ otherwise. Afterwards apply step (B) to the new representation $\bar{r}_{nm+1} := (e, v_{nm+1})$ and to $\bar{1}$-1.

(C) (General unfolding) Form $t_{n+1} := v_{nk_n}[e]$ and apply step (A) to $r_{n+1} := (e, t_{n+1})$.

The termination criteria are analogous to those for data-driven evaluation:

(B1a) If v'_{nm} does not contain variables then $v'^{WC^\infty}_{nm} = r^{WC^\infty}_n$ and the process may stop.

(B2a) If u_{nm} is maximal in the respective carrier set then $u_{nm} = r^{WC^\infty}_n$ and the process may stop.

Theorem 5 (1) For the u_{nm} evolving during demand-driven evaluation of $r_n = (e, t_n)$ and the object $r^{WC^\infty}_n$ represented by r_n in WC^∞ one has $r^{WC^\infty}_n = \bigsqcup \{u_{mp} : m \geq n, p=1, \ldots, k_m\}$.

(2) If the carrier set of the sort of t_n is flat and none of the termination criteria is ever satisfied then $r^{WC}_n = \bot$.

Example: We enrich Σ_1 to Σ_2 by a sort bool and by the operations

true, false : → bool,
not : bool → bool,
iszero : nat → bool,
if . then . else . fi : bool × nat × nat → nat,
incr : sequ → sequ,
sel : nat × sequ → nat .

If we take true, false, 0, succ, (), add and the $\bot$'s as constructors, the following equations are safe:

not($\bot$) = $\bot$,
not(true) = false,
not(false) = true,

iszero($\bot$) = $\bot$,
iszero(0) = true,
iszero(succ(n)) = false,

if $\bot$ then m else n fi = $\bot$,
if true then m else n fi = m,
if false then m else n fi = n,

incr($\bot$) = $\bot$,
incr(()) = (),
incr(add(n, x)) =
  add(succ(n), incr(x)),

sel(n, $\bot$) = $\bot$,
sel(n, ()) = $\bot$,
sel(n, add(m, x)) =
  if iszero(n) then m else sel(pred(n), x)) fi.

The solution of the system

e : sequ nats = add(0, incr(nats))

is the infinite sequence add(0, add(1, add(2, ...))).

Still the data- as well as the demand-driven evaluation of a representation (e, sel(succ^k(0), nats)) terminate with the value succ^k(0). □

7. AN EQUATIONAL SPECIFICATION OF A FUNCTIONAL PROGRAMMING LANGUAGE

This section contains our main example for the techniques described. We shall give an equational specification of functionals of arbitrary order over sequences of natural numbers. For this purpose we first extend the set of sorts S_2 of signature Σ_2 by infinitely many new sorts for functionals to a set of sorts S_3 as follows:

(1) $S_2 \subseteq S_3$

(2) If $s_1, \ldots, s_n, s \in S_3$ then funct $(s_1, \ldots, s_n)$ $s \in S_3$

(3) S_3 is the least set (wrt. inclusion) satisfying (1) and (2).

The carrier of sort funct $(s_1, \ldots, s_n)$ s in the algebra to be constructed will contain denotations of function(al)s with argument sorts $s_1, \ldots, s_n$ and result sort s. Some members of S_3 are

funct (nat) nat, funct (nat, funct (bool, sequ) sequ) nat,
funct (funct (nat) sequ, funct (sequ, sequ) bool) funct (nat) nat .

The operation symbols for our algebra are the following:

$\underline{\text{if}} \cdot \underline{\text{then}} \cdot \underline{\text{else}} \cdot \underline{\text{fi}}_s : \underline{\text{bool}} \times s \times s \times s \to$ for all $s \in S_3$

$\text{apply}^s_{s_1 \ldots s_n} : \underline{\text{funct}}\ (s_1, \ldots, s_n)\ s \times s_1 \times \ldots \times s_n \to s$ for all $s_i, s \in S_3$

$f' : \to \underline{\text{funct}}\ (s_1, \ldots, s_n)\ s$ for all operation symbols $f : s_1 \times \ldots \times s_n \to s$ $(n > 0)$

$\pi^j_{s_1 \ldots s_n} : \to \underline{\text{funct}}\ (s_1, \ldots, s_n)\ s_j$ for all $s_1, \ldots, s_n \in S_3$ and $1 \leq j \leq n$

$\text{const}^s_{s_1 \ldots s_n} : s \to \underline{\text{funct}}\ (s_1, \ldots, s_n)\ s$ for all $s_1, \ldots, s_n, s \in S$

$\text{comp}^{s_1 \ldots s_n, s}_{u_1 \ldots u_m} : \underline{\text{funct}}\ (s_1, \ldots, s_n)\ s \times \underline{\text{funct}}\ (u_1, \ldots, u_m)\ s_1 \times \ldots$
$\times \underline{\text{funct}}\ (u_1, \ldots, u_m)\ s_n \to \underline{\text{funct}}\ (u_1, \ldots, u_m)\ s$
for all $u_1, \ldots, u_m, s_1, \ldots, s_n, s \in S_3$

$\text{cond}^u_s : \underline{\text{funct}}\ (s)\ \underline{\text{bool}} \times \underline{\text{funct}}\ (s)\ u \times \underline{\text{funct}}\ (s)\ u \to \underline{\text{funct}}\ (s)\ u$
for all $s, u \in S_3$

If we take all new operations beside $\underline{\text{if}} \cdot \underline{\text{then}} \cdot \underline{\text{else}} \cdot \underline{\text{fi}}$ and apply as constructors, the following equations are operative:

$\underline{\text{if}}\ \bot\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y\ \underline{\text{fi}}_s = \bot,$

$\underline{\text{if}}\ \text{true}\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y\ \underline{\text{fi}}_s = x,$

$\underline{\text{if}}\ \text{false}\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y\ \underline{\text{fi}}_s = y,$

$\text{apply}^s_{s_1 \ldots s_n}\ (\bot_{\underline{\text{funct}}\ (s_1, \ldots, s_n)\ s}, x_1, \ldots, x_n) = \bot_s,$

$\text{apply}^s_{s_1 \ldots s_n}\ (f', x_1, \ldots, x_n) = f(x_1, \ldots, x_n),$

$\text{apply}^{s_j}_{s_1 \ldots s_n}\ (\pi^j_{s_1 \ldots s_n}, x_1, \ldots, x_n) = x_j,$

$\text{apply}^s_{s_1 \ldots s_n}\ (\text{const}^s_{s_1 \ldots s_n}\ (z), x_1, \ldots, x_n) = z,$

$\text{apply}^s_{u_1 \ldots u_m}\ (\text{comp}^{s_1 \ldots s_n, s}_{u_1 \ldots u_m}\ (g, h_1, \ldots, h_n), x_1, \ldots, x_m) =$
$\text{apply}^s_{s_1 \ldots s_n}\ (g, \text{apply}^{s_1}_{u_1 \ldots u_m}\ (h_1, x_1, \ldots, x_m), \ldots,$
$\text{apply}^{s_n}_{u_1 \ldots u_m}\ (h_n, x_1, \ldots, x_m)\ )$

$\text{apply}^u_s\ (\text{cond}^u_s\ (p, g, h), x) =$

$\underline{\text{if}}\ \text{apply}^u_s\ (p, x)\ \underline{\text{then}}\ \text{apply}^u_s\ (g, x)$

$\underline{\text{else}}\ \text{apply}^u_s\ (h, x)\ \underline{\text{fi}}_u .$

Note, that only the first arguments of the $\underline{\text{if}} \cdot \underline{\text{then}} \cdot \underline{\text{else}} \cdot \underline{\text{fi}}_s$ and the $\text{apply}^s_{s_1 \ldots s_n}$ are critical; in the case of the apply's this is the function argument.

In the sequel we drop the indices of the operation symbols.

- apply denotes the operation of applying a function denotation to its arguments
- $\underline{\text{if}} \cdot \underline{\text{then}} \cdot \underline{\text{else}} \cdot \underline{\text{fi}}$ denotes the usual conditional
- f' is a syntactic construct for denoting a function f as a functional constant of the next-higher order
- π^j is a syntactic construct for denoting the j-th projection (or selection)
- const is a syntactic construct for denoting constant functions
- comp is a syntactic construct for denoting function composition
- cond is a syntactic construct for denoting the conditional.

Thus the constructors of our signature correspond to the syntactic constructs found in most functional (applicative) programming languages.

Let us illustrate this with an example: The while-combinator of /Backus 78/ can be denoted by the recursive definition

$\underline{\text{funct}}\ (\underline{\text{funct}}\ (s)\ \underline{\text{bool}}, \underline{\text{funct}}\ (s)\ s)\ \underline{\text{funct}}\ (s)\ s$ while =
comp(cond', π^1, comp(comp', while, π^2), const(id))

where the identity function id can again be realized by a projection (we have not done this in order to avoid distinguishing projections with different indices).

Let F = apply(while, comp(not', iszero'), pred'), or, in Backus's notation F = (while not $\circ$ iszero pred).

A busy evaluation of the term

apply(F, succ(0)),

or

F : succ(0),

first unfolds while once (since no simplifications are possible) giving

```
apply(apply(comp(cond', π¹,
                 comp(comp', while π²),
                 const(id),
                 comp(not', iszero'), pred'),
      succ(0)                              ).
```

This simplifies to

```
apply( cond(comp(not', iszero'),
            comp(F , pred'),
            id),
       succ(0)  )
```

and further to

```
apply(F , 0) .
```

Now again while is unfolded and this time the simplification yields 0 so that the busy evaluation stops.

Since busy evaluation unfolds all recursively defined variables, it corresponds to the full computation rule of /Vuillemin 74/. In lazy evaluation, only those variables which are critical for apply are unfolded. Because we represent a nested function application

```
f(f(x, y), f(u, v))
```

as

```
apply(f, apply(f, x, y), apply(f, u, v))
```

and only the first argument of apply is critical, lazy evaluation of functionals here corresponds to a restricted and more economical form of the parallel outermost rule (not all outermost occurrences of recursively defined functions are expanded but only those occurring within some critical argument of an extension).

8. CONCLUSION

The method of algebraic specification, when extended to inductive continuous algebras, is a convenient tool for defining and describing many interesting domains with nonstrict operations and with limit points. This answers to a number of remarks in /Cartwright, Donahue 82/ on the "restrictiveness" of the algebraic specification technique.

The main difference between the approach of /Cartwright, Donahue 82/ and that of the present paper seems to be the following: We specify the domains axiomátically rather than explicitly; syntactic criteria (in this paper the restriction to pure equations; more general forms are considered in /Möller 82/) then guarantee the existence of suitable continuous models. In /Cartwright, Donahue 82/ the reverse approach is taken: first the domains are given explicitly; afterwards their characteristic properties (corresponding to our axioms) are proved.

Our main aim, however, was to show how a precise operational definition of lazy and busy evaluation can be derived from their mathematical definition in terms of continuous algebras; derivations using axioms of a certain restricted form (operative axioms) lend themselves as a reduction calculus for the evaluation of terms over the respective algebras. This method, too, works uniformly for recursively defined data and functions.

Further research should concern techniques for implementing non-operative algebraic specifications by operative ones; in this way a first specification can be free of operational details which should only be introduced at a later stage in the development process.

REFERENCES

/ADJ 77/
J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright: Initial algebra semantics and continuous algebras. JACM 24, 68-95 (1977)

/Backus 78/
J. Backus: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM 21, 613-641 (1978)

/Bauer 79/
F.L. Bauer: Detailization and lazy evaluation, infinite objects and pointer representation. In: F.L. Bauer, M. Broy (eds.): Program construction. LNCS 69. Berlin: Springer 1979, 406-420

/Bauer, Wössner 82/
F.L. Bauer, H. Wössner: Algorithmic language and program development. Berlin-Heidelberg-New York: Springer 1982

/Bloom 76/
S.L. Bloom: Varieties of ordered algebras. JCSS 13, 200-212 (1976)

/Broy 80/
M. Broy: Transformation parallel ablaufender Programme. Fakultät für Mathematik der TU München, Dissertation, 1980. Institut für Informatik der TU München, TUM-I8001, 1980

/Cartwright, Donahue 82/
R. Cartwright, J. Donahue: The semantics of lazy (and industrious) evaluation. Conf. Record of the 1982 ACM Symposium on LISP and functional programming, 253-264

/Courcelle, Nivat 76/
B. Courcelle, M. Nivat: Algebraic families of interpretations. 17th Annual IEEE Symposium on Foundations of Computer Science, 1976, 137-146

/Friedman, Wise 76/
D.P. Friedman, D.S. Wise: CONS should not evaluate its arguments. In: S. Michaelson, R. Milner (eds.): Automata, languages and programming. Edinburgh: Edinburgh University Press 1976, 257-285

/Guttag 75/
J.V. Guttag: The specification and application to programming of abstract data types. PH.D. Thesis, University of Toronto, Department of Computer Science, Rep. CSRG-59, 1975

/Henderson, Morris 76/
P. Henderson, J.H. Morris: A lazy evaluator. Proc 3rd ACM Symposium on Principles of Programming Languages, 1976, 95-103

/Manna 74/
Z. Manna: Mathematical theory of computation. New York: McGraw-Hill 1974

/Möller 82/
B. Möller: Unendliche Objekte und Geflechte. Fakultät für Mathematik und Informatik der TU München, Dissertation, TUM-I8213, 1982

/Nivat 75/
M. Nivat: On the interpretation of recursive polyadic program schemes. Istituto Nazionale di Alta Matematica, Symposia Mathematica XV. London: Academic Press 1975, 255-281

/Treleaven et al. 82/
P.C. Treleaven, D.R. Brownbridge, R.P. Hopkins: Data-driven and demand-driven computer architecture. Computing Surveys 14, 93-143 (1982)

/Vuillemin 74/
J. Vuillemin: Correct and optimal implementations of recursion in a simple programming language. JCSS 9, 332-354 (1974)

/Vuillemin 75/
J. Vuillemin: Syntaxe, sémantique et axiomatique d'une langage de programmation simple. Basel: Birkhäuser 1975

SPECIFICATION-ORIENTED SEMANTICS FOR COMMUNICATING PROCESSES

E.-R. Olderog and C.A.R. Hoare

Programming Research Group, Oxford University
Institut für Informatik, Universität Kiel*)

Contents:

*) Authors' present address: Oxford University Computing Laboratory, Programming Research Group, 45 Banbury Road, Oxford OX2 6PE, United Kingdom

1. Introduction

For concurrent programs - even when restricted to a particular style like Communicating Processes - a variety of semantical models have been proposed (e.g.[Mi 80, HBR 81, BZ 82]). Each of these different models can be viewed as describing certain aspects of a complex behaviour of programs. It seems therefore desirable to bring some "order" into these semantical models so that one will finally be able to recommend each model for the purposes and applications for which it is best suited.

More specifically we pursue the following aims:

(1) The semantics of concurrent programs should lead to a simple correctness criterion, and simple proofs of correctness.

(2) Simple methods should be developed for generating sound semantical models for different purposes and applications.

(3) Existing semantical models should be related to each other in a clear system of classification.

In this paper we concentrate on an application to Communicating Processes and present concrete steps towards aim (1) - (3). In different settings such steps can also be found in recent work by [BZ 82, Br 82, Mi 82, NH 82] . We now outline the steps taken in our paper.

Ad (1): A program P is called correct w.r.t. a given specification S, abbreviated by P sat S, if every observation we can make about the behaviour of P is allowed by S. Varying the definition of an observation, we can express either partial or total correctness of P in this way.

In Sections 2-4 we start from a set M of observations and define the class of specifications S as a certain family of subsets of M. A specification-oriented semantics assigns denotationally to every program P a specification $[\![P]\!]$ such that P sat S holds iff $[\![P]\!] \subseteq S$ is true. This approach deals very simply with nondeterminism, which can be expressed just as the set union of sets of observations. Technically we require that a specification space over an observation space shall be a complete partial order. Designing a specification-oriented semantics for a programming language $L(\Sigma)$ means then mapping every language constructor in $L(\Sigma)$ onto a continuous operator on specifications. This enables us to treat recursion in the traditional way and leads to simple proof rules based on fixed point induction.

Ad (2): The simplest way of defining a language constructor is by pointwise application of a relationship between observations. Let g be such a relation. We define

$$\mathcal{O}_g(P) = \{ y \mid \exists x: x \in P \ \& \ x \ g \ y \}$$

Unfortunately, this operator is not necessary continuous. By viewing specification spaces as examples of information systems in the sense of [Sc 82] we arrive at a definition $\mathcal{C}_g$ from g, which is guaranteed to be continuous. Unfortunately, the result of this construction is rather obscure. Our main result is that under certain assumptions about g, we can show

$$\mathcal{C}_g = \mathcal{O}_g \quad \text{or} \quad \mathcal{C}_g = \mathcal{O}_g \cup \mathcal{O}_g^\infty$$

where $\mathcal{O}_g^\infty$ is defined explicitly in terms of g, and represents the possibility of divergence. The details are given in Sections 5-6.

Ad (3): In Sections 7-10 we apply the methods developed so far to generate and relate three semantical models for (sublanguages of) the language $L(\Sigma)$ of Communicating Processes. $L(\Sigma)$ describes networks of processes which work in parallel and communicate with each other in a synchronised way. To construct these networks, $L(\Sigma)$ has operators $a \longrightarrow$ (first communication), $\|$ (parallel composition with synchronisation), or (internal nondeterminism), $\square$ (external nondeterminism), $\backslash b$ (communication hiding), $\mu \xi.P$ (recursion).

The semantical models differ in the structure of their observation space and this influences the number of representable operators and the notion of correctness. The simplest model is the Counter Model $\mathcal{C}$ which can only deal with acyclic or tree-like networks of processes [Ho 82] . Arbitrary networks require the Trace Model $\mathcal{T}$ which can still not model $\square$ [Ho 80] .

In $\mathcal{C}$ and $\mathcal{T}$ only safety properties (partial correctness) can be described by P sat S. Dealing with the full language $L(\Sigma)$ and with liveness properties (total correctness) calls for the more sophisticated Readiness Model $\mathcal{R}$ [HBR 81, HH 82] .

In these models special attention is given to the hiding operator $\backslash b$ which localises communications on internal channels of a network. This is the most complicated operator, simply because the possibility of hiding infinitely many communications has to be considered. We show that in all three models the hiding operator can be represented as $Cg = \mathcal{O}g \cup \mathcal{C}^\infty g$. The remaining operators are simply of the form $Cg = \mathcal{O}g$.

Finally, in Section 11 we indicate further directions of research. Proofs of our results will appear in the full version of this paper.

2. Algebraic Preliminaries

In this section we explain the general format of our programming language and denotational semantics. The notation is similar to that of [GTWW 77] for continuous, many-sorted Σ-algebras.

A signature is a structure $\Sigma = (T,V,F)$ where $(t \in)T$ is a set of (ground) types, $(\xi \in)V$ is a set of variables each one with a certain ground type t associated, and a set $(f \in)F$ of operator symbols each one of a certain (derived) type $t_1, \ldots, t_n \longrightarrow t$. The set of variables of type t is denoted by $V_t \subseteq V$ and the set of operator symbols of type $t_1, \ldots, t_n \longrightarrow t$ by $F_{t_n, \ldots, t_n \longrightarrow t} \subseteq F$ where F_t abbreviates $F_{\longrightarrow t}$.

Every signature Σ determines a programming language $(P,Q \in) L(\Sigma)$, namely the set of all recursive terms or programs over Σ each of which with a certain type t. The set $L(\Sigma)_t \subseteq L(\Sigma)$ of programs with type t is defined by the following BNF-syntax.

$$P ::= \xi \mid f(P_1, \ldots, P_n) \mid \mu\xi.P$$

where $\xi \in V_t$, $f \in F_{t1, \ldots, tn \longrightarrow t}$, and $P_i \in L(\Sigma)_{ti}$. The construct $\mu\xi.-$ [Ba 80] defines a binding occurrence of ξ and gives rise to the usual notions of free and bound variables in programs. A program P without free variables is called <u>closed</u>.

Let $(\mathcal{D}, \sqsubseteq)$ be a partial order. A subset $X \subseteq \mathcal{D}$ is <u>directed</u> if every finite subset of X has an upper bound in X. $(\mathcal{D}, \sqsubseteq)$ is a <u>cpo</u> (<u>complete partial order</u>) if it has a least element $\bot$ and every directed subset X of $\mathcal{D}$ has a lub (least upper bound) $\bigsqcup X$ in $\mathcal{D}$. If $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are cpos, then so is $\mathcal{D}_1 \times \ldots \times \mathcal{D}_n$, with the componentwise ordering.

A function or operator $f: \mathcal{D} \longrightarrow \mathcal{E}$ from one cpo $\mathcal{D}$ into another cpo $\mathcal{E}$ is ($\sqsubseteq$ -) <u>continuous</u> if it preserves lubs of directed subsets, i.e. if $f(UX) = Uf(X)$ holds for every directed $X \subseteq \mathcal{D}$. (In particular f is then <u>monotonic</u>, i.e. preserves $\sqsubseteq$.) And f is called <u>strict</u> if it preserves the least element. We remark that an operator $f: \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \longrightarrow \mathcal{E}$ is continuous iff it is continuous in all its arguments (see e.g. [Ba 80]).

Every continuous operator $f: \mathcal{D} \longrightarrow \mathcal{D}$ has a least fixed point μf in $\mathcal{D}$, namely $\mu f = \bigsqcup \{ f^n(\bot) \mid n \geq 0 \}$ where $f^0(d) = d$ and $f^{n+1}(d) = f(f^n(d))$.

A (<u>continuous</u>) Σ-<u>algebra</u> $\mathcal{D}$ [GTWW 77] consists of a family $(\mathcal{D}_t \mid t \in T)$ of cpos together with a family $(\llbracket f \rrbracket_{\mathcal{D}} \mid f \in F)$ of continuous operators with

$$\llbracket f \rrbracket_{\mathcal{D}} : \mathcal{D}_{t1} \times \ldots \times \mathcal{D}_{tn} \longrightarrow \mathcal{D}_t$$

for every $f \in F_{t1, \ldots, tn \longrightarrow t}$. Every continuous Σ-algebra induces a straightforward <u>denotational semantics</u> of $L(\Sigma)$, also denoted by $\llbracket . \rrbracket_{\mathcal{D}}$. Let $(p \in)$ Env be the set of <u>environments</u>, i.e. mappings $p: V \longrightarrow \bigcup_t \mathcal{D}_t$ which respect types: $p(\xi) \in \mathcal{D}_t$ for $\xi \in V_t$. Then

$$\llbracket . \rrbracket_{\mathcal{D}} : L(\Sigma) \longrightarrow \text{Env} \longrightarrow \bigcup_t \mathcal{D}_t$$

is given by

(i) $\llbracket \xi \rrbracket_{\mathcal{D}}(p) = p(\xi)$

(ii) $\llbracket f(P_1, \ldots, P_n) \rrbracket_{\mathcal{D}}(p) = \llbracket f \rrbracket_{\mathcal{D}}(\llbracket P_1 \rrbracket_{\mathcal{D}}(p), \ldots, \llbracket P_n \rrbracket_{\mathcal{D}}(p))$

(iii) $\llbracket \mu\xi.P \rrbracket_{\mathcal{D}}(p) = \mu(\lambda d. \llbracket P \rrbracket_{\mathcal{D}}(p[d/\xi]))$

where the appropriate type constraints are assumed. As usual $p[d/\xi]$ denotes the redefinition of p at argument ξ by d. Note that $\llbracket . \rrbracket_{\mathcal{D}}$ respects types: $\llbracket P \rrbracket_{\mathcal{D}}(p) \in \mathcal{D}_t$ for $P \in L(\Sigma)_t$. For closed programs P we write $\llbracket P \rrbracket_{\mathcal{D}}$ instead of $\llbracket P \rrbracket_{\mathcal{D}}(p)$. $\mathcal{D}$ is also called a <u>model</u> for $L(\Sigma)$. When talking about a fixed model $\mathcal{D}$ we usually drop the subscript $\mathcal{D}$ at $\llbracket . \rrbracket_{\mathcal{D}}$.

For signatures Σ^1 and Σ^2 we write $\Sigma^1 \subseteq \Sigma^2$ if Σ^1 results from Σ^2 by removing ground types, variables, and operator symbols. For a Σ^2-algebra $\mathcal{D}$ and $\Sigma^1 \subseteq \Sigma^2$ the Σ^1-<u>reduct</u> $\mathcal{D} \restriction \Sigma^1$ is a Σ^1-algebra consisting of the families $(\mathcal{D}_t \mid t \in T^1)$ and $(\llbracket f \rrbracket_{\mathcal{D}} \mid f \in F^1)$.

Let $\mathcal{D}, \mathcal{E}$ be Σ-algebras with $\Sigma = (T,V,F)$ and $\Phi = (\Phi_t \mid t \in T)$ be a family of operators $\Phi_t: \mathcal{D}_t \longrightarrow \mathcal{E}_t$. Then Φ is a <u>homomorphism</u> from $\mathcal{D}$ to $\mathcal{E}$ if

$$\Phi_t(\llbracket f \rrbracket_{\mathcal{D}}(d_1, \ldots, d_n)) = \llbracket f \rrbracket_{\mathcal{E}}(\Phi_{t1}(d_1), \ldots, \Phi_{tn}(dn))$$

for all $f \in F_{t1, \ldots, tn \longrightarrow t}$ and $(d_1, \ldots, d_n) \in \mathcal{D}_{t1} \times \ldots \times \mathcal{D}_{tn}$.

Φ is called <u>continuous</u> (resp. <u>strict</u>) if every Φ_t is.

<u>Lemma 1</u>. Let $\mathcal{D}, \mathcal{E}$ be as above and $\Phi = (\Phi_t \mid t \in T)$ be a strict and continuous homomorphism from $\mathcal{D}$ to $\mathcal{E}$. Then $\Phi_t(\llbracket P \rrbracket_{\mathcal{D}}) = \llbracket P \rrbracket_{\mathcal{E}}$ holds for every closed program $P \in L(\Sigma)_t$.

3. Observations and Specifications

We now formalise the concepts of observation and specification mentioned in the introduction. We are interested in observations we can make about the behaviour of a program P during its operation. This intuition leads us to postulate a certain relation $\longrightarrow$ between observations which is intended to reflect that P usually produces the observations in a <u>step-by-step</u> manner. Thus $x \longrightarrow y$ means that observation y may be made immediately <u>after</u> x, without any intervening observation. This relation will be crucial later on in Section 6. First we need an auxiliary notion. A relation $g \subseteq M \times N$ between sets M and N is <u>domain finite</u> if for every $y \in N$ the set $\{x \mid x\, g\, y\}$ is finite.

<u>Definition 1</u>. An <u>observation space</u> is a structure $(M, \longrightarrow)$ where M is a set of so-called <u>observations</u> and $\longrightarrow$ is a relation $\longrightarrow \subseteq M \times M$ such that $\stackrel{+}{\longrightarrow}$ (the transitive closure of $\longrightarrow$) is non-reflexive and domain finite. This definition reflects the fact that at all times a program has made only a finite number of steps, and only a finite number of different observations can be made.

Equivalently, we could require that for every $x \in M$ the <u>set</u> of all descending chains $x_n \longrightarrow \ldots \longrightarrow x_1 = x$ is finite. Thus $(M, \stackrel{+}{\longrightarrow})$ is a simple well-founded order where every $x \in M$ has at most finitely many predecessors $y \longrightarrow x$. Note that $\longrightarrow$ may be empty. Some notation:

$$\text{Min} = \{ x \in M \mid \neg \exists y \in M: y \longrightarrow x \}$$

$$\text{pred}(x) = \{ y \in M \mid y \longrightarrow x \}$$

Note that pred(x) = $\emptyset$ iff x $\in$ Min. By a grounded chain of length $n \geq 0$ for x we mean a chain $x_0 \longrightarrow \ldots \longrightarrow x_n = x$ with $x_0 \in$ Min. The level of x is defined by level(x) = min $\{ n \in \mathbb{N}_o \mid \exists$ grounded chain of length n for x $\}$.

Example 1. Consider the set (s,t $\in$) A^* of words or traces over some alphabet (a,b $\in$) A with ε denoting the empty trace. Define the relation $\longrightarrow_A \subseteq A^* \times A^*$ as follows:

$$s \longrightarrow_A t \quad \text{iff} \quad \exists\, a \in A: \; s \cdot a = t.$$

Then $(A^*, \longrightarrow_A)$ is an observation space. And $s \xrightarrow{*}_A t$ (reflexive, transitive closure) holds iff s is a prefix of t ($s \leq t$ for short). Note that here every trace s has exactly one grounded chain $\varepsilon \longrightarrow_A \ldots \longrightarrow_A s$.

Specifications are certain sets of observations which reflect the structure $\longrightarrow$ on observations according to

Definition 2. Let $(M, \longrightarrow)$ be an observation space. A subset $X \subseteq M$ is generable w.r.t. $\longrightarrow$ if $\forall x \in X \setminus \text{Min} \; \exists y \in X: y \longrightarrow x$. If $\longrightarrow$ is empty, every subset $X \subseteq M$ is generable.

We say that an observation x satisfies a specification S or S allows x if $x \in S$. Specifications are ordered in a Smyth-like manner [Sm 78]:

$$S_1 \sqsubseteq S_2 \quad \text{iff} \quad S_1 \supseteq S_2 .$$

$S_1 \sqsubseteq S_2$ means that S_2 is stronger or more deterministic than S_2 resp. S_1 is weaker than S_2. M is the weakest specification allowing every observation.

Definition 3. A specification space over an observation space $(M, \longrightarrow)$ is a subset $\mathfrak{M} \subseteq \mathcal{P}(M)$ of so-called specifications such that

(S1) $M \in \mathfrak{M}$.

(S2) $\emptyset \notin \mathfrak{M}$.

(S3) Every $S \in \mathfrak{M}$ is generable w.r.t. $\longrightarrow$.

(S4) $(\mathfrak{M}, \supseteq)$ is a cpo.

$\mathfrak{M}$ is called a simple specification space if Min is finite and $\mathfrak{M}$ consists of all non-empty, generable subsets of M.

Example 2. Take $(A^*, \longrightarrow_A)$ of Example 1. A subset $X \subseteq A^*$ is prefix-closed if $t \in X$ and $s \leq t$ always imply $s \in X$. Let $\mathcal{T}_A$ be the set of all prefix-closed subsets $X \subseteq A^*$ with $\varepsilon \in X$. Then $\mathcal{T}_A$ is a simple specification space over $(A^*, \longrightarrow_A)$.

For observation spaces $(M, \longrightarrow_M)$ and $(N, \longrightarrow_N)$ the product is given by $(M \times N, \longrightarrow_{M\times N})$ where $\longrightarrow_{M\times N} \subseteq (M \times N) \times (M \times N)$ is defined as follows: $(x_1, y_1) \longrightarrow_{M\times N} (x_2, y_2)$ if either $x_1 \longrightarrow_M x_2$ and $y_1 = y_2$ or $x_1 = x_2$ and $y_1 \longrightarrow_N y_2$.

Definition 4. For specification spaces $\mathfrak{M}$ over $(M, \longrightarrow_M)$ and $\mathfrak{N}$ over $(N, \longrightarrow_N)$ the product specification space is $\mathfrak{M} \times \mathfrak{N}$ over $(M \times N, \longrightarrow_{M\times N})$.

Due to (S2) in Definition 3 the natural order $\supseteq$ on $\mathfrak{M} \times \mathfrak{N}$ coincides with the componentwise orders $\supseteq$ on $\mathfrak{M}$ and $\mathfrak{N}$. Thus $(\mathfrak{M} \times \mathfrak{N}, \supseteq)$ is indeed a cpo (cf. Section 2).

4. Specification-Oriented Semantics

We now bring together the concepts described in the previous two sections.

Definition 5. A Σ-specification algebra $\mathfrak{D}$ is a special continuous Σ-algebra consisting of a family $(\mathfrak{D}_t \mid t \in T)$ of specification spaces together with a family $(\llbracket f \rrbracket_{\mathfrak{D}} \mid f \in F)$ of $\supseteq$-continuous operators on specifications. By a specification-oriented semantics of the programming language $L(\Sigma)$ we mean a semantics $\llbracket \cdot \rrbracket_{\mathfrak{D}}$ of $L(\Sigma)$ induced by a specification algebra $\mathfrak{D}$.

So a specification-oriented semantics assigns a specification $\llbracket P \rrbracket_{\mathfrak{D}}$ as meaning to every program $P \in L(\Sigma)$. Correctness of a program $P \in L(\Sigma)_t$ w.r.t. a specification $S \in \mathfrak{D}_t$ is expressed by correctness "formulas" P sat S which are interpreted as follows:

$$\models_{\mathfrak{D}} P \text{ sat } S \quad \text{iff} \quad \llbracket P \rrbracket_{\mathfrak{D}} \subseteq S.$$

Informally P sat S holds if every observation we can make about P is allowed by S. In Sections 8-10 we shall see that both safety and liveness properties of communicating processes can be expressed within this general framework.

5. Information Systems

A very important property of a specification semantics is that all operators must be continuous. This ensures that recursion works in the expected fashion, and that proofs by simple induction are valid.

To support this task we develop general theorems for constructing $\supseteq$-continuous operators $C: \mathfrak{M} \longrightarrow \mathfrak{N}$ from one specification space $\mathfrak{M}$ to another one $\mathfrak{N}$. We do so by applying methods of Scott's theory of domains as presented in [Sc 82]. Crucial in this theory is the notion of finite element in a domain. Once these have been identified, continuous operators can be constructed in a standard way. To help to identify the

finite elements Scott has set up the concept of an information system.

First we explain how to view every simple specification space $\mathcal{M}$ as an example of an information system $\mathcal{F}$ such that the elements of $\mathcal{F}$ are just the specifications in $\mathcal{M}$. Well almost, because we have to face the problem that Scott's approach automatically leads to $\subseteq$-continuity whereas we are interested in $\supseteq$-continuity according to the Smyth-like order. But this difficulty is easy to overcome by constructing $\mathcal{F}$ so that its elements are exactly the complements of specifications in $\mathcal{M}$. So $\mathcal{F}$ is actually dealing with counter-observations to specifications.

An information system is a structure $\mathcal{F} = (\mathfrak{D}, \mathrm{Con}, \vdash)$ where is a set of data objects or propositions, Con is a set of finite subsets of $\mathfrak{D}$ called consistent sets of data objects, and $\vdash$ is a binary relation between Con and $\mathfrak{D}$ called the entailment relation. According to [Sc 82] the components $\mathfrak{D}$, Con, and $\vdash$ of $\mathcal{F}$ have to satisfy certain simple axioms.

For a given simple specification space $\mathcal{M}$ over $(M, \longrightarrow)$ we define $\mathcal{F}_m = (\mathfrak{D}_m, \mathrm{Con}_m, \vdash_m)$ as follows:

(i) $\mathfrak{D}_m = M$ (counterobservations)

(ii) Con_m consists of all finite subsets $X \subseteq M$ such that not $\mathrm{Min} \subseteq X$. (consistent sets of counterobservations)

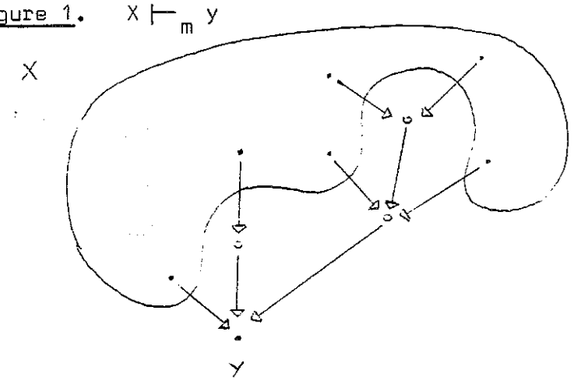
(iii) The entailment relation $\vdash_m$ is defined recursively (and more generally) as a relation $\vdash_m \subseteq \mathcal{P}(M) \times \mathcal{P}(M)$:

$X \vdash_m Y$ iff $\forall y \in Y:\ y \in X \vee (y \notin \mathrm{Min} \wedge X \vdash_m \mathrm{pred}(y))$.

Instead of $X \vdash_m \{y\}$ we write $X \vdash_m y$.

Informally $X \vdash_m y$ means that y can be excluded as a possible observation provided that every member of X has been excluded. This will be so if y is in X or if all predecessors of y have already been excluded on the ground of exclusion of X. Equivalently one could say that every grounded chain for y will eventually lead to some counterobservation in X (see Figure 1).

Figure 1. $X \vdash_m y$



Every information system $\mathcal{F}$ determines a set $\langle \mathcal{F} \rangle$ of elements [Sc 82]. For $\mathcal{F}_m$ the set $\langle \mathcal{F}_m \rangle$ consists of all proper subsets $X \subsetneq M$ with

$$\forall x \in M \setminus \mathrm{Min}:\ \mathrm{pred}(x) \subseteq X \longrightarrow x \in X.$$

Note: "proper subsets" is guaranteed by Min being finite (cf. Definition 3).

The set $\langle \mathcal{F}_m \rangle_{\mathrm{fin}}$ of finite elements of $\mathcal{F}_m$ consists by definition of all closures $\overline{X} = \{x \in M \mid X \vdash_m x\}$ of finite sets $X \in \mathrm{Con}_m$. But remember that the elements of $\mathcal{F}_m$ are sets of counterobservations. To get the corresponding sets of observations we take complements. Indeed

$$\mathcal{M} = \{M \setminus X \mid X \in \langle \mathcal{F}_m \rangle\}$$

holds. Thus $\mathcal{F}_m$ exactly determines the simple specification space $\mathcal{M}$. The complements of the finite elements $\overline{X}$ of $\mathcal{F}_m$ we call finitary specifications. Let $(F, G \in)\ \mathcal{M}_{\mathrm{fin}} \subseteq \mathcal{M}$ be the set of these.

6. Continuous Operators

Let $\mathcal{M}$ and $\mathcal{N}$ be simple specification spaces over $(M, \longrightarrow_M)$ resp. $(N, \longrightarrow_N)$. We derive now general theorems for constructing $\supseteq$-continuous operators $C_g: \mathcal{M} \longrightarrow \mathcal{N}$ working on specifications by starting from certain relations $g \subseteq M \times N$ which describe the desired effect of C_g "pointwise" for single observations.

Definition 6. A relation $g \subseteq M \times N$ is called generable if for every non-empty, generable set $X \subseteq M$ also $g(X)$ is non-empty and generable.

Theorem 1. Let $g \subseteq M \times N$ be generable and $C_g: \mathcal{M} \longrightarrow \mathcal{N}$ defined by

$$C_g(S) = \bigcap \{G \in \mathcal{N}_{\mathrm{fin}} \mid \exists F \in \mathcal{M}_{\mathrm{fin}}:\ S \subseteq F \wedge g(F) \subseteq G\}.$$

Then C_g is well-defined, i.e. for every non-empty, generable $S \in \mathcal{M}$ the image $C_g(S)$ is a non-empty, generable set in $\mathcal{N}$, and C_g is $\supseteq$-continuous.

Proof. Consider the corresponding information systems $\mathcal{F}_m = (\mathfrak{D}_m, \mathrm{Con}_m, \vdash_m)$ and $\mathcal{F}_n = (\mathfrak{D}_n, \mathrm{Con}_n, \vdash_n)$. We define a relation $fg \subseteq \mathrm{Con}_m \times \mathrm{Con}_n$ by

$$X\ fg\ Y \quad \text{iff} \quad \overline{X} \supseteq g^{-1}(\overline{Y})$$

where $\overline{X}, \overline{Y}$ denote the closures of X, Y under $\vdash_m$ resp. $\vdash_n$. Since g is generable, fg is an approximable mapping [Sc 82]. Hence the operator $C_{fg}: \langle \mathcal{F}_m \rangle \longrightarrow \langle \mathcal{F}_n \rangle$ working on the elements of $\mathcal{F}_m$ defined by

$$C_{fg}(Z) = \{ Y \in Con_n \mid \exists X \in Con_m : X \subseteq Z \wedge X\, fg\, Y \}$$

is well-defined and $\subseteq$-continuous. Taking complements we see that for $S \in \mathfrak{M}$

$$Cg(S) = N \setminus C_{fg}(M \setminus S)$$

holds which proves the claim about Cg. □

Theorem 1 is a general continuity result, but it is <u>too abstract</u> for our purposes. When applying the operator Cg to a specification S we are not interested in how exactly S is approximated by finitary specifications F and we don't want to follow the tedious construction of Cg(S) described in Theorem 1. We would rather like to apply the relation g directly to S. In the rest of this section we investigate this idea and derive <u>explicit representations</u> of Cg(S) in terms of g. The advantage of Theorem 1 is that it relieves us of the obligation of proving continuity of these representations directly.

First we compare Cg with the standard operator $\mathcal{C}_g : \mathfrak{M} \longrightarrow \mathcal{P}(N)$ induced by g, namely

$$\mathcal{C}_g(S) = \{ y \in N \mid \exists x \in S : \; x\, g\, y \} = g(S).$$

Note: If g is generable then $\mathcal{C}_g(S) \subseteq C_g(S)$ for every $S \in \mathfrak{M}$.

<u>Theorem 2</u>. If g is generable and domain finite, then $Cg(S) = \mathcal{C}g(S)$ for every $S \in \mathfrak{M}$.

If the grounded orders $\longrightarrow_M$ and $\longrightarrow_N$ are empty, every $g \subseteq M \times N$ is generable. Then Theorem 2 reduces to the well-known fact that domain finiteness of g implies $\geq$-continuity of $\mathcal{C}_g$ as an operator on arbitrary subsets of M. As we shall see in Sections 8-10, most operators for Communicating Processes are induced by domain finite relations g. But the crucial hiding-operator is <u>not</u>.

<u>Example 3</u>. Take the simple specification space $\mathcal{T}_A$ over $(A^*, \longrightarrow_A)$ of Example 2. For observations (traces) $s, t \in A^*$ we define

$$s\, g\, t \quad \text{iff} \quad s\backslash b = t$$

where $s\backslash b$ is obtained from s by deleting (or <u>hiding</u>) all occurrences of b in s. Then g is generable, but not domain finite. And $\mathcal{C}_g : \mathcal{T}_A \longrightarrow \mathcal{P}(A^*)$ is not $\geq$-continuous as soon as $|A| \geqslant 2$ holds. Take $S_n = \{b^n\} \cdot A^*$ for $n \geqslant 0$. Then $S_0 \geq \ldots \geq S_n \geq \ldots$ is a descending chain (hence directed). But

$$\bigcap_n \mathcal{C}_g(S_n) = (A \setminus \{b\})^* \neq \{\varepsilon\} = \mathcal{C}_g(\{b\}^*) = \mathcal{C}_g(\bigcap_n S_n).$$

We remark that $C_g(\bigcap_n S_n) = (A \setminus \{b\})^*$.

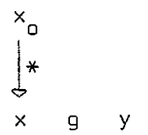
In the rest of this section we study this problem in considerable generality. Our solution depends critically on the relation $\longrightarrow$ among observations and the concept of generability. First we introduce a new operator $\mathcal{C}_g^\infty : \mathfrak{M} \longrightarrow \mathcal{P}(N)$ by

$$\mathcal{C}_g^\infty(S) = \{ y' \mid \exists y \xrightarrow{*} y' \;\; \overset{\infty}{\exists} x \in S : \; x\, g\, y \wedge y' \in g(M) \}$$

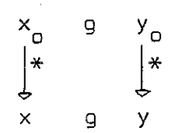
where $\overset{\infty}{\exists}$ means "there exist infinitely many".

<u>Definition 7</u>. A relation $g \subseteq M \times N$ is called <u>level finite</u> if for every $y \in N$ and $l \in \mathbb{N}_o$ there are only finitely many $x \in g^{-1}(y)$ with level $(x) = l$.

<u>Definition 8</u>. A relation $g \subseteq M \times N$ is called <u>downward consistent</u> if whenever



holds there exists some y_o with

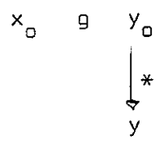


<u>Lemma 2</u>. Let $g \subseteq M \times N$ be generable, level finite and downward consistent. Then

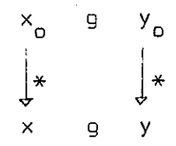
$$C_g(S) \subseteq \mathcal{C}_g(S) \cup \mathcal{C}_g^\infty(S) \subseteq g(M)$$

holds for every $S \in \mathfrak{M}$.

<u>Definition 9</u>. A relation $g \subseteq M \times N$ is called <u>upward consistent</u> if whenever



with $y \in g(M)$ holds then there exists some x with



And g is called <u>consistent</u> if g is both upward and downward consistent.

Lemma 3. Let $g \subseteq M \times N$ be level finite and upward consistent. Then

$$\mathcal{O}_g(S) \cup \mathcal{O}_g^{\infty}(S) \subseteq C_g(S)$$

holds for every $S \in \mathfrak{M}$.

Combining Lemma 2 and 3 yields our main result.

Theorem 3. Let $g \subseteq M \times N$ be generable, level finite and consistent. Then

$$C_g(S) = \mathcal{O}_g(S) \cup \mathcal{O}_g^{\infty}(S)$$

holds for every $S \in \mathfrak{M}$.

If the grounded orders $\longrightarrow$ are empty, Theorem 3 reduces to Theorem 2. So far we considered only simple specification spaces. When dealing with non-simple ones, Theorems 1-3 yield of course $\supseteq$-continuous operators $C_g : \mathfrak{M} \longrightarrow \mathcal{P}(N)$. But it remains to be shown that indeed $C_g(S) \in \bigcap_{\mathcal{R} \subseteq \mathcal{P}(N)} \mathcal{R}$ holds for every $S \in \mathfrak{M}$. An example of non-simple specification spaces will be studied in Section 10. Dealing with operators C_g of several arguments is easy: we just take the product of the argument specification spaces.

7. Communicating Processes

A process can engage in certain observable communications, determined by its alphabet. We are interested in describing networks of such processes which work in parallel and communicate with each other in a synchronised way. Communicating Processes is a language $L(\Sigma)$ which describes how such networks can be constructed.

Formally, we start from a set (a,b $\in$) Comm of communications. Usually Comm is structured as Comm = Cha x $\mathcal{M}$ where Cha is a set of channel names and $\mathcal{M}$ is a set of messages. But for simplicity we shall not exploit this structure here. By alphabets we mean finite subsets $A,B \subseteq$ Comm. The signature $\Sigma = (T,V,F)$ for Communicating Processes is given as follows:

(i) T is the set of alphabets $A \subseteq$ Comm.

(ii) V is an arbitrary set of variables.

(iii) $\underline{stop}_A, \underline{chaos}_A \in F_A$

$a \longrightarrow \quad \in F_A \longrightarrow A$ provided $a \in A$

$\underline{or}, \Box \quad \in F_{A,A \longrightarrow A \cup A}$

$\| \quad \in F_{A,B \longrightarrow A \cup B}$

$\setminus b \quad \in F_{A \longrightarrow A \setminus \{b\}}$ provided $b \in A$

To introduce some notational conventions let us restate $L(\Sigma)_A$:

$$P ::= \xi \mid \underline{stop}_A \mid \underline{chaos}_A \mid a \longrightarrow P \mid$$
$$P \ \underline{or}\ Q \mid P \Box Q \mid P \| Q \mid P \setminus b \mid \mu\xi.P$$

Some informal explanation: $\underline{stop}_A$ denotes a process which engages in no communication at all. It terminates at the very beginning. $\underline{chaos}_A$ is wholly arbitrary and can exhibit every possible behaviour (within the alphabet A). The process $a \longrightarrow P$ first engages in a and then behaves like P. The intuition behind the remaining operator symbols has been mentioned in the introduction.

Besides the "full" language $L(\Sigma)$ of Communicating Processes we consider two sublanguages $L(\Sigma^1)$, $L(\Sigma^2)$ of $L(\Sigma)$ with $\Sigma^1 \subseteq \Sigma^2 \subseteq \Sigma$.

- $\Sigma^2 = (T,V,F^2)$ is obtained from Σ by removing $\Box$ from F.
- $\Sigma^1 = (T,V,F^1)$ is obtained from Σ^2 by imposing stricter alphabet constraints on $\|$:

$$\| \in F^1_{A,B \longrightarrow A \cup B} \quad \text{only if} \quad |A \cap B| \leq 1$$

By varying the structure of observations, we will be able to give different semantics for $L(\Sigma^1)$, $L(\Sigma^2)$ or $L(\Sigma)$; each will be slightly simpler than its successor.

8. The Counter Model $\mathcal{C}$

We start with the simplest language $L(\Sigma^1)$. We postulate that the only thing we can observe about a process P is how many times each communication in its alphabet has occurred up to any given moment [Ho 82]. Formally, we define for a given alphabet A the set of A-observations by

$$(h \in) \quad \mathcal{H}_A = A \longrightarrow \mathbb{N}_0$$

i.e. for each communication $a \in A$ there is a separate counter. $\mathcal{H}_A$ induces an observation space $(\mathcal{H}_A, \longrightarrow_A)$ with

$$h \longrightarrow_A h' \quad \text{iff} \quad \exists a \in A: \ h' = h\,[h(a) + 1/a]\,.$$

Then $h \overset{*}{\longrightarrow}_A h'$ means $h(a) \leq h'(a)$ for every $a \in A$ ($h \leq h'$ for short). Let $ZERO_A$ denote the constant mapping h with $h(a) = 0$ for $a \in A$.

Let the set $\mathcal{C}_A$ of A-specifications consist of all generable w.r.t. $\longrightarrow_A$ subsets $S \subseteq \mathcal{H}_A$ with $ZERO_A \in S$. Then $\mathcal{C}_A$ is a simple specification space over $(\mathcal{H}_A, \longrightarrow_A)$. We can now introduce the Counter Model $\mathcal{C}$ as a Σ^1-specification algebra consisting of the families $(\mathcal{C}_A \mid A \text{ alphabet})$ and $(\llbracket f \rrbracket_{\mathcal{C}} \mid f \in F^1)$ with $\llbracket f \rrbracket$ (we drop subscript $\mathcal{C}$) as follows:

(i) $[\![\underline{stop}_A]\!] = \{ZERO_A\}$

(ii) $[\![\underline{chaos}_A]\!] = \mathcal{H}_A$

(iii) $[\![a \longrightarrow P]\!] = Cg([\![P]\!])$ where we use the notation of Theorem 1 with $g \subseteq \mathcal{H}_A \times \mathcal{H}_A$ as follows:

$$h\ g\ h' \text{ iff } h' = ZERO_A \text{ or } h' = h[h(a) + 1/a].$$

Since g is generable and domain finite, Theorem 2 implies $Cg = \mathcal{O}g$ yielding as explicit definition

$$[\![a \longrightarrow P]\!] = \{ZERO_A\} \cup \{h[h(a) + 1/a] \mid h \in [\![P]\!]\}$$

which is continuous and well-defined by Theorem 1.

(iv) Let P and Q have alphabets A resp. B with $|A \cap B| \leq 1$.

$[\![P \parallel Q]\!] = Cg([\![P]\!], [\![Q]\!])$ where g relates the product $H_A \times H_B$ with $H_{A \cup B}$ by

$(h_1, h_2)\ g\ h$ iff $h = h_1 \cup h_2$ and $\forall a \in A \cap B: h_1(a) = h_2(a)$,

This formalises the intuition that P and Q work independently except for their common communications. Clearly g is domain finite, but generability depends critically on the alphabet restriction $|A \cap B| \leq 1$ imposed in Σ^1. Again Theorem 2 yields $Cg = \mathcal{O}g$. Thus

$$[\![P \parallel Q]\!] = \left\{ h_1 \cup h_2 \;\middle|\; \begin{array}{l} h_1 \in [\![P]\!] \wedge\ h_2 \in [\![Q]\!] \\ \wedge \forall a \in A \cap B: h_1(a) = h_2(a) \end{array} \right\}.$$

This is actually the join-operator $[\![P]\!]$ join $[\![Q]\!]$ known from relational data bases. The phenomenon that join does not maintain generability in general is due to the fact that we cannot observe the relative timing between different communications in the Counter Model $\mathcal{C}$. A similar problem, known as merge anomaly, can arise in loosely coupled nondeterministic dataflow networks [BA 81, Br 82].

(v) $[\![P \text{ or } Q]\!] = [\![P]\!] \cup [\![Q]\!]$.

(vi) $[\![P \setminus b]\!] = Cg([\![P]\!])$ where $g \subseteq \mathcal{H}_A \times \mathcal{H}_{A \setminus \{b\}}$ is given by

(+) $h\ g\ h'$ iff $h' = h \restriction A \setminus \{b\}$ iff $\exists n \in \mathbb{N}_0: h = h' \cup \{(a,n)\}$.

Intuitively, g hides all communications b in h. Now g is not domain finite any more, but at least it is level finite. Also g is generable and consistent. Thus Theorem 3 yields $Cg = Og \cup Og^\infty$ leading to the following explicit definition

$$(++)\ [\![P \setminus b]\!] = \{h \mid \exists n \in \mathbb{N}_0: h \cup \{(a,n)\} \in [\![P]\!]\} \cup \{h' \mid \exists h \leq h'\ \overset{\infty}{\exists} n \in \mathbb{N}_0: h \cup \{(a,n)\} \in [\![P]\!]\}.$$

Here it is an advantage to have Theorem 3 available because it is not easy to prove continuity of (++) directly. Moreover, Section 6 tells us that (++) is the natural continuous operator induced by the intuitive hiding relation (+).

The general theory also explains the design decisions taken in the Counter Model $\mathcal{C}$ - once the underlying concept of observation was fixed:

- Generability of specifications is needed to ensure continuity of the hiding operator.
- The restriction $|A \cap B| \leq 1$ imposed on parallel composition is due to its simple definition as join-operator and the need to preserve generability. If we picture processes $P_1 \parallel \ldots \parallel P_n$ working in parallel as networks with $P_1, \ldots, P_n$ as nodes and common communications between P_i and P_j as edges, the restriction means that we can deal only with acyclic or tree-like networks (cf. comment following Corollary 1 in the next section).
- Non-emptiness of specifications helped to deal with operators of two arguments according to Section 3.

9. The Trace Model

To deal with the language $L(\Sigma^2)$, allowing cyclic networks of processes, we must be able to observe more about the behaviour of processes. The set of A-observations is now given by the trace set

$$(s, t \in)\ A^*$$

i.e. we can additionally observe the relative order of communications [Ho 80]. A^* induces an observation space $(A^*, \longrightarrow_A)$ and a simple specification space $\mathcal{T}_A$ over $(A^*, \longrightarrow_A)$ as defined in Examples 1 and 2.

The Trace Model $\mathcal{T}$ is the Σ^2-specification algebra consisting of the families $(\mathcal{T}_A \mid A \text{ alphabet})$ and $([\![f]\!]_{\mathcal{T}} \mid f \in F^2)$. We state only the explicit definitions of $\parallel$ and $\setminus b$:

$$[\![P \parallel Q]\!] = \{s \in (A \cup B)^* \mid s \restriction A \in [\![P]\!] \wedge\ s \restriction B \in [\![Q]\!]\}$$

$$[\![P \setminus b]\!] = \{s \setminus b \mid s \in [\![P]\!]\} \cup \{(s \setminus b) \cdot t \mid \forall n \geq 0: s \cdot b^n \in [\![P]\!] \wedge t \in (A \setminus \{b\})^*\}$$

where $s \restriction A$ results from s by removing all communications outside A and $s \setminus b = s \restriction (A \setminus \{b\})$. As with the Counter Model these explicit definitions can be derived from appropriate relations g on traces.

Let us now relate the two models $\mathcal{T}$ and $\mathcal{C}$. For traces s let $a \# s$ denote the number of occurrences of a in s. Then the relation $g \subseteq A^* \times \mathcal{H}_A$ with

$$s\ g\ h \text{ iff } \forall a \in A: h(a) = a \# s$$

describes the natural conversion from traces into counters. Since g

is generable and domain finite, the operator $\Phi_A : \mathcal{T}_A \longrightarrow \mathcal{C}_A$ defined by $\Phi_A(S) = \sigma_g(S)$ is well-defined and continuous by Theorems 1 and 2. Since Φ_A is not injective, we lose information by going from $\mathcal{T}$ to $\mathcal{C}$, but we can state

Proposition 1. The family $\Phi = (\Phi_A \mid A \text{ alphabet})$ is a homomorphism from the reduct $\mathcal{T}\restriction\Sigma^1$ to $\mathcal{C}$.

Corollary 1. For every closed program $P \in L(\Sigma)_A$ we have $\Phi_A([\![P]\!]_{\mathcal{T}}) = [\![P]\!]_{\mathcal{C}}$.

If we assume the channel structure Comm = Cha x $\mathcal{M}$ of communications, an interesting combination of the two models $\mathcal{C}$ and $\mathcal{T}$ is possible, namely when we postulate that the relative order between communications can be observed if and only if they are sent along the same channel. Such a combined model $\mathcal{C}-\mathcal{T}$ is able to describe networks or processes acyclicly connected via channels. Applications of this kind of networks are buffers and protocols [CH 81].

What is the notion of correctness induced by $\mathcal{T}$ (and $\mathcal{C}$)? For a program $P \in L(\Sigma^2)_A$ and a specification $S \in \mathcal{T}_A$

(*) $\models_{\mathcal{T}} P \text{ sat } S$ iff $[\![P]\!]_{\mathcal{T}} \subseteq S$.

Note that there is a particular program P which satisfies every specification $S \in \mathcal{T}_A$, namely $P = \text{stop}_A$. Such a program is called a miracle [Ho 82]. The existence of miracles shows that (*) expresses only safety properties [OL 80] of P in the sense that P does nothing that is forbidden by S (cf. counterobservations in Section 5). The situation has its analogue in the theory of partial correctness for sequential programs where the diverging program div satisfies every partial correctness formula $\{P\}\ \text{div}\ \{Q\}$. In the next section we study a refinement of the Trace Model which can deal also with total correctness or better liveness properties [OL 80].

10. The Readiness Model $\mathcal{R}$

We consider now the full language $L(\Sigma)$ of Communicating Processes. We postulate that not only the "past" of a process can be observed via traces, but also a part of the "future" via so-called ready sets indicating which communications can happen next [Ho 81, HH 82]. Thus the set of A-observations is now given by

$$((s,X),(t,Y) \in) \quad B_A = A^* \times \mathcal{P}(A).$$

The first component s of an A-observation (s,X) is a trace in A^* just as in the Trace Model $\mathcal{T}$, and the second component X is a subset of A, the ready set of (s,X). On B_A we define a relation $\longrightarrow_A$ as follows:

$$(s,X) \longrightarrow_A (t,Y) \text{ iff } \exists a \in A: \ s \cdot a = t.$$

Since A is finite, $(B_A, \longrightarrow_A)$ is an observation space where observations (ε, X) are minimal.

Let the set $\mathcal{R}_A$ of A-specifications consist of all non-empty subsets $S \subseteq B_A$ which are generable w.r.t. $\longrightarrow_A$ and extensible in the following sense:

$$(s,X) \in S \wedge a \in X \longrightarrow \exists Y: \ (s.a, Y) \in S.$$

Extensibility formalises the intuition that all communications in a ready set X can happen next. $\mathcal{R}_A$ is a non-simple specification space over $(B_A, \longrightarrow_A)$ without miracles. The existence of observations $(s,\emptyset)$ enables us to specify and prove of particular programs that they will not occur. This is why the correctness criterion P sat S deals now with both safety and liveness properties. For example, a specification

$$S = \{(a^n, \{a\}) \mid n \geq 0\}$$

forces a program P with P sat S to be live and to "send" an infinite stream of communications a.

The Readiness Model $\mathcal{R}$ is a Σ-specification algebra consisting of the families $(\mathcal{R}_A \mid A \text{ alphabet})$ and $([\![f]\!]_{\mathcal{R}} \mid f \in F)$ where we only state the explicit definitions of $\|$, $\Box$ and $\backslash b$:

$$\text{(i)} \quad [\![P \| Q]\!] = \left\{ (s, (X \cap Y) \cup (X \setminus B) \cup (Y \setminus A)) \;\middle|\; \begin{array}{l} \text{where} \\ s \in (A \cup B)^* \wedge (s\restriction A, X) \in [\![P]\!] \\ (s\restriction B, Y) \in [\![Q]\!] \end{array} \right\}$$

$$\text{(ii)} \quad [\![P \,\Box\, Q]\!] = \left\{ (\varepsilon, X \cup Y) \;\middle|\; (\varepsilon,X) \in [\![P]\!] \wedge (\varepsilon,Y) \in [\![Q]\!] \right\} \cup \left\{ (s,X) \;\middle|\; s \neq \varepsilon \wedge (s,X) \in [\![P]\!] \cup [\![Q]\!] \right\}$$

$$\text{(iii)} \quad [\![P \setminus b]\!] = \left\{ (s\setminus b, Y) \;\middle|\; (s,X) \in [\![P]\!] \wedge Y = X \setminus \{b\} \right\} \cup \left\{ ((s\setminus b)\cdot t, Y) \;\middle|\; \begin{array}{l} \forall n \geq 0 \ \exists X: (s \cdot b^n, X) \in [\![P]\!] \\ \wedge\ (t,Y) \in B_A \setminus \{b\} \end{array} \right\}$$

Ready sets enables us to model external nondeterminism: in $P \,\Box\, Q$ the environment can control whether this process behaves like P or like Q by choosing (in the first step only) either a communication in the ready set X of P or in Y of Q.

Again we use Theorems 1-3 for deriving these explicit operator definitions systematically from the corresponding relations on observations. Additionally, we must prove that each constructor preserves the extensibility property of its operands (cf. Section 6). This little extra work reflects the fact that extensibility has nothing to do with the concept of continuity.

Finally we relate the models $\mathcal{R}$ and $\mathcal{T}$. Note that the projection relation $g \subseteq B_A \times A^*$

$$(s,X)\ g\ t \quad \text{iff} \quad s = t$$

is generable and domain finite. Theorems 1 and 2 imply that $\Phi_A : \mathcal{R}_A \longrightarrow \mathcal{T}_A$ with

$$\Phi_A(S) = \sigma_g(S)$$

is well-defined and continuous. (We need not bother about extensibility here.) Thus

Proposition 2. The family $\Phi = (\Phi_A \mid A \text{ is an alphabet})$ is a homomorphism from the reduct $\mathcal{R}\restriction\Sigma^2$ to $\mathcal{T}$.

Corollary 2. For every closed program $P \in L(\Sigma)_A$ we have $\Phi_A([\![P]\!]_{\mathcal{R}}) = [\![P]\!]_{\mathcal{T}}$.

Note that for $P,Q \in L(\Sigma)$

$$\Phi_A([\![P \,\square\, Q]\!]_{\mathcal{R}}) = \Phi_A([\![P \text{ or } Q]\!]_{\mathcal{R}})$$

holds, i.e. in the Trace Model $\mathcal{T}$ external nondeterminism reduces to internal nondeterminism.

11. Conclusion

We are aiming at a classification of semantical models for communicating processes that will enable us to recommend certain models which are just detailed enough for particular applications. But before such an aim can be fully realised, more sophisticated models of processes should be studied.

For example, we have not considered the notion of state so far. This would allow to add assignment and explicit value passing between processes, thus combining sequential programs with communicating processes.

It is also important to ensure that the operators satisfy the usual algebraic laws, for example parallel composition should be associative.

The relationship between specification-oriented denotational semantics used here and the operational semantics used in [Mi 80, Mi 82, Pl 82] should also be studied. In particular, it is interesting to investigate how the correctness criterion P sat S can be derived systematically from the operational semantics. A significant step in this direction has already been made in [NH 82].

Finally, an explicit syntax for the specification language and proof systems for the relation P sat S should be developed. First proposals for such proof systems can be found in [CH 81, Ho 81].

Acknowledgement. The first author was supported by the DFG under grant no. La 426/3-1, and by the University of Kiel in granting him leave of absence.

References

[Ba 80] J.W. de Bakker, Mathematical Theory of Program Correctness (Prentice Hall, London, 1980).

[BZ 82] J.W. de Bakker, J.I. Zucker, Denotational semantics of concurrency, Proc. 14th ACM Symp. on Theory of Computing (1982) 153-158.

[BA 81] J.D. Brock, W.B. Ackermann, Scenarios: a model of non-determinate computations, in: J. Diaz, I. Ramos, Eds., Formalisation of Programming Concepts, LNCS 107 (Springer, Berlin-Heidelberg-New York, 1981) 252-267.

[Br 82] M. Broy, Fixed point theory for communication and concurrency, in: D. Bjørner, Ed., Formal Description of Programming Concepts II, Preliminary Proc. IFIP TC-2 Working Conference (North Holland, Amsterdam, 1982) 104-126.

[CH 81] Z. Chaochen, C.A.R. Hoare, Partial correctness of communicating processes, in: Proc. 2nd International Conference on Distributed Computing Systems, Paris (1981).

[GTWW 77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, Initial algebra semantics and continuous algebras, J. ACM 24 (1977) 68-95.

[HH 82] E.C.R. Hehner, C.A.R. Hoare, A more complete model of communicating processes (to appear in TCS) 1982.

[Ho 80] C.A.R. Hoare, A model for communicating sequential processes, in: R.M. McKeag, A.M. McNaghton, Eds., On the Construction of Programs (Cambridge University Press, 1980) 229-243.

[Ho 81] C.A.R. Hoare, A calculus of total correctness for communicating processes, SCP 1 (1981) 49-72.

[Ho 82] C.A.R. Hoare, Specifications, programs and implementations, Tech. Monograph PRG-29, Oxford Univ., Progr. Research Group, Oxford 1982.

[HBR 81] C.A.R. Hoare, S.D. Brookes, A.W. Roscoe, A theory of communicating sequential processes, Tech. Monograph PRG-16, Oxford Univ., Progr. Research Group, Oxford 1981.

[Mi 80] R. Milner, A calculus of communicating systems, LNCS 92 (Springer, Berlin-Heidelberg-New York, 1980).

[Mi 82] R. Milner, Four combinators for concurrency, in: Proc. ACM SIGACT - SIGOPS Symp. on Principles of Distributed Computations, Ottawa, 1982.

[NH 82] R. de Nicola, M.C.B. Hennessy, Testing equivalences for processes, Internal Report CSR-123-82, Univ. of Edinburgh, Computer Science Dept., 1982.

[OL 80] S. Owicki, L. Lamport, Proving liveness properties of concurrent programs, Tech. Report, Computer Science Lab., Stanford University, 1980.

[Pl 82] G.D. Plotkin, An operational semantics for CSP, in: D. Bjørner, Ed., Formal Description of Programming Concepts II, Preliminary Proc. IFIP TC-2 Working Conference (North Holland, Amsterdam, 1982) 185-208.

[Sc 82] D.S. Scott, Domains for denotational semantics, in: M. Nielsen, E.M. Schmidt, Eds., Proc. 9th ICALP, LNCS 140 (Springer, Berlin-Heidelberg-New York, 1982) 577-613.

[Sm 78] M.B. Smyth, Power domains, J. CSS 16 (1978) 23-26.

Prof. Dr. Andrzej Salwicki
Institut für Informatik
und Praktische Mathematik
der Christian-Albrechts-Universität
2300 Kiel

" ON THE CONCATENATION RULE "

Abstract

The concatenation rule of type declarations is a natural extension of copy rule. However, the power and the usefulness of the former go beyond intuitive expectations. We would like to mention numerous applications of the rule in : implementation of data structures, problem oriented languages, operating systems etc. The difficulties of an efficient and correct implementation of the rule will be discussed. Finally we shall present a solution implemented in LOGLAN compiler. The concatenation rule should be studied further in order :
1 to learn about new applications, 2 to find other variants which might appear as more attractive.

SCHRIFTEN ZUR INFORMATIK UND ANGEWANDTEN MATHEMATIK

Herausgeber: K.Indermark, J.Merkwitz, W.Oberschelp, B.Schinzel, W. Thomas, RWTH Aachen

1972	1	V. PENNER	Entscheidbarkeit und Akzeptierbarkeit auf Push-Down-Store Maschinen
	2	K. STEFFENS	Injektive Auswahlfunktionen
1973	3	W. OBERSCHELP D. WILLE	Einführung in Mathematische Logik und Mengenlehre
	4	M. DEZA	On minimal number of terms in representation of natural numbers as a sum of Fibonacci numbers
	5	V. PENNER	Push-Down-Store-berechenbare Funktionen
	6	K.U. GUTSCHKE	Einige Abschätzungen zum Turán-Problem
	7	D. WILLE	Anzahlbestimmungen in einer Klasse von Relationen
	8	P. BÖHNE	Über die Partitionszahlen p(n,k)
	9	W. OBERSCHELP	Ein Differentiationskalkül zur Berechnung von Turmpolynomen
	1o	K. STEFFENS	Anzahluntersuchungen von Transversalen und injektiven Auswahlfunktionen an abzählbaren Familien mit lauter endlichen Mitgliedern
	11	D. WILLE	Selbstkomplementäre Graphen und Relationen
1974	12	R. MÖHRING	Reduzierte Berechnung der minimalen Kettendekomposition endlicher Ordnungen
	13	H.G. SPELDE	Ein Beitrag zur Kardinalzahlbestimmung der Isomorphieklassen endlicher Ordnungen und Primordnungen
	14	J. GIESE	Nichtnormale asynchrone Automaten und ihre Zustandszuordnungen
	15	J. BISKUP	Bi-immune Mengen und Aufzählungsoperatoren
	16	W. OBERSCHELP D. WILLE	Diskrete Strukturen
	17	W. OBERSCHELP	Ausgewählte Kapitel der Kombinatorik
	18	P. ALTMANN	Ein Vergleich zwischen partiell-rekursiven und LISP-ähnlichen Funktionen
1975	19	R. PARCHMANN M. SEDELLO	Minimalisation of the Heigth of Binary Trees Representing Arithmetic Expressions for Parallel Evaluation
1975	20	R. PARCHMANN	Syntaxgesteuerte Codierung zur Berechnung arithmetisc Ausdrücke auf einer Parallelmaschine unter Verwendung KNUTH'scher Attribute
	21	G. REIS	Eine untere Abschätzung für die Komplexität von Determinante und Permanente
	22	M.M. RICHTER C. SCHIPPANG	Rekursionstheorie
	23	M.M. RICHTER	Resolution, Paramodulation and Gentzen-Systems
	24	K. JUSTEN C. SCHIPPANG	Gentzen-Systems and some extensions of Maslow's inverse method
1976	25	W. OBERSCHELP G. REIS	Einführung in die Mathematische Komplexitätstheorie
	26	J. BISKUP W. OBERSCHELP	Informatik I
	27	J. GIESE	Informatik III
	28	P. ALTMANN J. GIESE V. PENNER U. STEUP	Vollständige Beschreibung der Programmiersprache BROTAL und eine theoretisch begründete Implementierung eines BROTAL → MIXAL - Compilers
	29	P. ALTMANN J. GIESE V. PENNER U. STEUP	Anhang zu Bericht Nr. 28 "Implementierung der Programmiersprache BROTAL": Programmliste eines BROTAL → MIXAL - Compilers und ein Übersetzungsbeispiel
	30	E. BÖRGER	A new general approach to the theory of the many-one equivalence of decision problems for algorithmic systems
	31	M.M. RICHTER	Mathematische Logik I
1977	32	J. BISKUP	A Note on the Time Measure of One-Tape Turing Machines
	33	V. PENNER	Formalisierung der Semantik von Programmiersprachen und Anwendungen
	34	H. BÜCKEN	Wortprobleme bei Reduktions- und Gleichungssystemen
	35	K. INDERMARK	Berechenbarkeit und Algorithmen
	36	G. SANDLÖBES	Perfekte Gruppen bis zur Ordnung 10^4
1978	37	J. BISKUP	On the complementation rule for multivalued dependencies in database relations
	38	P. ALTMANN	Implementierung von Standardprozeduren für BCPL auf einer TR440

1978	39	J. BISKUP W. OBERSCHELP	Informatik II
	40	H.NEFFKE W.SCHUMACHER	Mathematische Formulierung eines Lagerhaltungsmodells mit variablen Inspektionszeitpunkten und Vergleich dieses Modells mit dem Periodenmodell mittels Simulation
	41	W. REISIG	On a Class of Co-operating Sequential Processes
	42	W. DAMM	The IO- and OI-Hierarchies
	43	H. KLAEREN	Datenräume mit algebraischer Struktur
	44	E. FEHR	On Typed and Untyped λ-Schemes
	45	R. MÖHRING	Vorlesungen über Ordnungen und Netzplantheorie
	46	R. PARCHMANN	Grammatiken mit Attributschema und zweistufige Auswertung attributierter Grammatiken
	47	M.M.RICHTER S.KEMMERICH	BOOLE'SCHE ALGEBREN
	48	P.ALTMANN	Zu den theoretischen Grundlagen und der Implementierung eines Scannergenerators
	49	J.Tiuryn	Unique fixed points vs. least fixed points
1979	50	V. PENNER	Die Eingabesprache LDL für ein Compiler-erzeugendes System und ein umfassendes Beispiel
	51	J. GIESE	Der Einsatz von assoziativen Speichern in Steuerwerken zur Überlappung von Mikrobefehlen
	52	M.M.RICHTER	Geometrie und Logik; eine Vorlesung zur Einführung in die Theorie der Topoi
	53	L.RUDAK	Equational Definability of Iterative Theories
	54	J. TIURYN	Fixed points in the power set algebra of infinite trees
	55	J. TIURYN	Logic of Effective Definitions
	56	W. REISIG	Schemes for Nonsequential Processing Systems
1980	57	E. FEHR	Lambda-Calculus as Control Structures of Programming Languages
	58	H. KLAEREN	A Simple Class of Algorithmic Specifications for Abstract Software Modules
	59	W. REISIG	Deterministic Buffer Synchronization of Sequential Processes
	60	R.E. BURKARD	Lineare Optimierung
	61	H. NEFFKE	a) The Law of the Iterated Logarithm for Renewal Counting Processes. b) Das Gesetz des iterierten Logarithmus für kumulative Prozesse.
1980	62	H. Burwick	Anwendungsorientierte Programmiersprachen
	63	T. Dell K. Weihrauch	Berechenbarkeit auf CPO-s
	64	K. Indermark	On rational definitions in complete algebras without rank
	65	H. Klaeren H. Petzsch	The development of an interpreter by means of abstract algebraic software specifications
	66	H. Klaeren	On parameterized abstract software modules using inductively specified operations
1981	67	J. Biskup	Über Datenbankrelationen mit Nullwerten und Maybe-Tupeln
	68	H.Klaeren H.Petzsch	Algebraic software specification and compiler generation - A case study
	69	U.Goltz	Konzepte der Programmiersprache Ada
	7o	W. Oberschelp	Informatik III (Datenstrukturen) Ausarbeitung: Heiner Klocke
	71	K. Indermark H. Klaeren	Automatentheorie und formale Sprachen I
	72	E. Fehr	The Lambda-Semantics of LISP
	73	M.M. Richter B. Schlüsser D. Schwarz	Modelltheorie
	74	M.M. Richter E. Friesch	Universelle Algebra
1982	75	H. Klaeren	Einführung in die Abstrakte Software-Spezifikatic
	76	Ch. Kreitz	Zulässige CPO's: ein Entwurf für ein allgemeines Berechenbarkeitskonzept
	77	W. Damm	A sound and relatively* complete Hoare-logic
		B. Josko	for a language with higher type procedures
	78	H. Klaeren	A constructive method for abstract algebraic Software specification.
	79	K. Indermark	Reduction semantics for rational schemes
	80	B. Josko	A note on expressivity definitions in Hoare-logics
	81	G. Weiss	Stochastic Bounds on Distributions of Optimal Value Functions with Applications to Pert, Network Flow and Reliability
	82	P. Horster	Kryptologie

1982	83	H. Burwick	Einführung in die Datenbankprogrammierung anhand der Systeme IMS (IBM) und DMS/77 (GMI)
	84	K. Indermark	Complexity of infinite trees
1983	85	H. Burwick	Sprachbeschreibung der anwendungsorientierten Programmiersprache GOGOL
1983	86	V. Penner	Entwicklung und Verifikation eines Scanner-Generators mit dem GYPSY Verification Environment
1983	87	K. Indermark	Arbeitstagung "Theorie der Programmierung" 13.12.-16.12.82 in Altenahr