ISBN 82-553-0347-2

Mathematics
No 7 - June 1978

FINITE ALGORITHMIC PROCEDURES AND COMPUTATION THEORIES

by

J.Moldestad, V.Stoltenberg-Hansen &
J.V. Tucker

PREPRINT SERIES - Matematisk institutt, Universitetet i Oslo

FINITE ALGORITHMIC PROCEDURES AND COMPUTATION THEORIES

by

J.Moldestad, V.Stoltenberg-Hansen &
J.V. Tucker

This article analyses the relationships existing between some natural classes of machine-theoretic computable functions on a relational system A and between them and natural criteria for these classes to take on the large scale structure of the recursive functions on the natural numbers, ω. It is written in association with our [11] with which the reader is henceforth assumed acquainted, in particular there is to be found an extensive introduction to both papers.

The four kinds of function on A considered are those functions definable by a <u>finite algorithmic procedure</u>, a <u>fap</u>, by a <u>fap with a stack</u>, a <u>fapS</u> - these were defined in the first section of [11] - by a <u>fap with counting</u>, a <u>fapC</u>, and by a <u>fap with both counting and stacking</u>, a <u>fapCS</u> - these are defined in section two here. The classes of functions over A including all numbers of arguments are denoted FAP(A), FAPS(A), FAPC(A) and FAPCS(A) respectively.

The essential abstract global features of the recursive functions on ω such as the existence of codings and of universal computable functions, are invested in the axiomatic concept of a computation theory, the subject of section one. The principal question addressed here is What are the basic classes of machine computable functions on a relational system A, with a finite number of operations and relations, which take on the structure of a computation theory? The obvious numerical coding of programmes distinguishes the class FAPC(A) so we prepare our algebras by adjoining arithmetic to them in section three. In section four, the investigation reveals the algebraic foundation of these forms of computing and concludes with the answer that adding arithmetic is not enough:

Theorem FAPCS(A) is the class of functions computable in the minimal computation theory over A with code set ω.

In section five the uniqueness of the operations of stacking and counting is established by examples. And in section six we examine the situation where one wants to compute with the constant functions over the structure: here we invent a new coding and encounter the necessity of adjoining pairing functions to our algebras but analogous theorems are proved.

One of us - Tucker - wishes to acknowledge the indispensible support of a fellowship from the European Programme of the Royal Society, London.

1. Computation Theories

Throughout we are concerned with a relational structure of the form $A = (A ; \sigma_1,\ldots,\sigma_l ; S_1,\ldots,S_s)$ wherein the operations and relations need not be total; the set of all n-ary partial functions on A is denoted $P(A^n,A)$ with $P(A) = \bigcup_{n\in\omega} P(A^n,A)$, exactly the notation of [11] in fact. A^* is the set of all finite sequences of elements of A.

The central analytical idea in the paper is that of the computation theory which axiomatises the experience of the theory of the partial recursive functions on ω.

$\Theta \subset P(A)$ is said to be a computation theory over A with code set $C \subset A$ and its elements said to be Θ-computable functions iff associated to Θ is a surjection $\alpha : C \to \Theta$, called a coding and abbreviated by $\alpha(e) = \{e\}$ for $e \in C$, and a length of computation function $|\,| : C \times A^* \to On$, partially defined, $|e;\underline{a}|\downarrow \Longleftrightarrow \{e\}(\underline{a})\downarrow$, for which all the following properties hold.

I. C is acceptable as a code set in that it contains (an isomorphic copy of) ω and Θ contains (functions which correspond to) successor, predessor and zero on ω.

II. Θ contains these generating functions:

(i) for each n and $1 \leq i \leq n$ the projection functions $U_i^n(a_1,\ldots,a_n) = a_i$ with Θ-uniform codes $p_1(n,i)$;

(ii) each operation σ of A;

(iii) for each relation S of A the definition-by-cases function defined

$$DC_S(\underline{a},x,y) = x \text{ if } S(\underline{a})$$
$$= y \text{ if } \neg S(\underline{a}).$$

III. Θ is uniformly closed under

(i) <u>the composition of functions</u>: if f and g are n+1 and n-ary Θ-computable functions with codes $\hat{f},\hat{g}$ respectively then their composition defined $C(f,g)(\underline{a}) \simeq f(g(\underline{a}),\underline{a})$ is Θ-computable with Θ-uniform code $p_2(n,\hat{f},\hat{g})$.

(ii) <u>the permuting of arguments</u>: let ${}^j\underline{a} = (a_j,a_1,\ldots,a_{j-1},a_{j+1},\ldots,a_n)$ when $\underline{a} = (a_1,\ldots,a_n)$. If f is an n-ary Θ-computable function with code $\hat{f}$ then, for each $1 \leq j \leq n$, the function defined ${}^jf(\underline{a}) \simeq f({}^j\underline{a})$ is Θ-computable with Θ-uniform code $p_3(n,j,\hat{f})$.

(iii) <u>the addition of arguments</u>: if f is an n-ary Θ-computable function with code $\hat{f}$ then, for any m, the (n+m)-ary function g defined $g(\underline{a},\underline{b}) \simeq f(\underline{a})$ is Θ-computable with Θ-uniform code $p_4(n,m,\hat{f})$.

IV. Θ contains universal functions U_n such that for $e \in C, \underline{a} \in A^n$

$$U_n(e,\underline{a}) \simeq \{e\}(\underline{a})$$

with Θ-uniform codes $p_5(n)$.

V. Θ enjoys this iteration property: for each n,m there is a Θ-computable map S^n_m, with Θ-uniform code $p_6(n,m)$, such that for $e \in C, \underline{a} \in C^n, \underline{b} \in A^m$

$$\{S^n_m(e,\underline{a})\}(\underline{b}) \simeq \{e\}(\underline{a},\underline{b}).$$

And finally it is required of the length function to respect the efficiency of the functions mentioned in axioms III, IV and V.

VI. (i) Composition: $|(p_2(n,\hat{f},\hat{g});\underline{a})| > \max\{|(\hat{f};g(\underline{a}),\underline{a})|, |(\hat{g};\underline{a})|\}$.

(ii) Permutation: $|(p_3(n,j,\hat{f});\underline{a})| > |(\hat{f};{}^j\underline{a})|$.

(iii) Addition: $|(p_4(n,m,\hat{f});\underline{a})| > |(\hat{f};\underline{a})|$.

(iv) Universality: $|(p_5(n);e,\underline{a})| > |(e;\underline{a})|$.

(v) Iteration: $|(S^n_m(e,\underline{a});\underline{b})| > |(e;\underline{a},\underline{b})|$.

Notice that axiom I ensures a copy of the partial recursive functions on ω is contained within every computation theory.

There are a number of such axiomatisations, this definition is essentially that in [5] and is in our opinion the most successful. Its evolution is rather involved: it originates in the work of Y.N. Moschovakis [13,14,15] and was first taken up by Fenstad in [4]. Its subsequent development as a method of analysis and generalisation in Recursive Function Theory sets down roots in the theory of recursion in higher types, as in Moldestad's [10], and in degree theory on the ordinals, as in Stoltenberg-Hansens's [17]. For this paper familiarity with Moschovakis' [15] is invaluable but for a comprehensive introduction the reader should consult Fenstad's book [7] with which this article is consistent and from which we take the following ideas and facts without proofs.

A functional of the form $\phi:P(A^{n_1},A)\times\ldots\times P(A^{n_k},A)\times A^m\times A^n\to A$ is <u>Θ-effective</u> over A iff there exists a Θ-code $\hat{\phi}$ such that for any appropriate $e_1,\ldots,e_k$,

$$\phi(\{e_1\},\ldots,\{e_k\},\underline{b},\underline{a}) \simeq \{\hat{\phi}\}(e_1,\ldots,e_k,\underline{b},\underline{a})$$

and its action is consistent with length of computation: there always exist $g_i \leq \{e_i\}$, $1\leq i\leq k$, such that $\phi(g_1,\ldots,g_k,\underline{b},\underline{a}) \simeq \phi(\{e_1\},\ldots,\{e_k\},\underline{b},\underline{a})$ and $|(\hat{\phi};e_1,\ldots,e_k,\underline{b},\underline{a})| > \max\{z_1,\ldots,z_k\}$ where $z_i = \sup\{|(e_i;\underline{b},\underline{x})|:g_i(x)\downarrow$.

Such a functional ϕ arises as a functional $P(A^n,A)\to P(A^n,A)$ with k function parameters and m algebra parameters, $\phi(\underline{f},\underline{b})(\underline{a}) = \phi(\underline{f},\underline{b},\underline{a})$, in section four. In connection with theorem 2.1 (and 2.2) of [11] we shall assume this delicate form of the

<u>1.1 First Recursion Theorem</u>

If ϕ is Θ-effective and monotonic as $\phi(\underline{f},\underline{b})$, and if the $\underline{f}$ are Θ-computable, then the least fixed-point $\phi(\underline{f},\underline{b})^*$ is Θ-computable. Moreover the fixed-point operator is a Θ-effective functional.

Let Θ and Φ be computation theories over A with code set C. Then Θ is said to be a <u>subcomputation theory</u> of Φ iff $\Theta\subset\Phi$ and there exists a Φ-computable map $p:\omega\times C\to C$ such that for each $e\in C,\underline{a}\in A^n$ $\{e\}(\underline{a}) = \{p(n,e)\}(\underline{a})$ and, of course, $|(e;\underline{a})|_\Theta \leq |(p(n,e),\underline{a})|_\Phi$.

Θ is said to be a <u>minimal computation theory over A with code set C</u> iff whenever Φ is a computation theory over A with code set C then Θ is a subcomputation theory of Φ.

2. Finite Algorithmic Procedures with Arithmetic

The notions of an A-register machine and an A-register machine with a stack for a relational structure were explained in [11]. Here we consider machines with the new capacity of performing recursive operations on the natural numbers, the idea, along with that of the A-register machine, of H. Friedman [8].

Programmes for such machines are written in the following language. Variables are $r_o, r_1, \ldots$ for algebra registers and $c_o, c_1, \ldots$ for counting registers which are to contain natural numbers. s denotes the stack register. Function and relation symbols are those used for the species of the relational structure A. In addition there are function symbols for successor (+1) and predecessor (-1) on the natural numbers.

A programme is an ordered finite list of instructions $(I_1, \ldots, I_k)$ each instruction being an operational instruction, a conditional instruction or a halting instruction. For completeness we list the permissible instructions and give their intended meaning along with numerical codes, whenever relevant, containing the characteristic parameters of the instruction.

The operational instructions are:

Code	Instruction	Interpretation
$<0,0,\mu,\lambda>$	$r_\mu := r_\lambda$	Replace the contents of r_μ with that of r_λ.
$<0,i,\mu,<\lambda_1,\ldots,\lambda_{n_i}>>$	$r_\mu := \sigma_i(r_{\lambda_1}, \ldots r_{\lambda_{n_i}})$	Apply the n_i-ary operation σ_i to the contents of $r_{\lambda_1}, \ldots, r_{\lambda_{n_i}}$ and place the value in r_μ.

Code	Instruction	Interpretation
<2,i>	$s:=(i;r_o,\ldots,r_m)$	Place the contents of $r_o,\ldots,r_m$ as an m+1 tuple along with the marker i topmost in the stack register.
<2.j>	restore $(r_o,\ldots,r_{j-1},r_{j+1},\ldots,r_n)$	Replace the contents of $r_o,\ldots,r_{j-1},r_{j+1},\ldots r_m$ by those of the topmost m+1 tuple in the stack register after which the m+1 tuple in the stack is deleted.
	$c_\mu:=c_\lambda+1$	Add one to the contents of c_λ and place that value in c_μ.
	$c_\mu:=c_\lambda-1$	If c_λ contains 0 place 0 in c_μ. Else subtract one from the contents of c_λ and place that value in c_μ.

The conditional instructions determine the order of executing instructions. They are:

$<3,0,\mu,\lambda,l,l'>$	if $r_\mu=r_\lambda$ then l else l'	If registers r_μ and r_λ contain the same elements then the next instruction is I_l else it is $I_{l'}$.
$<3,i,<\lambda_1,\ldots,\lambda_{m_i}>,l,l'>$	if $S_i(r_{\lambda_1},\ldots r_{\lambda_{m_i}})$ then l else l'	If the m_i-ary relation is true of the contents of $r_{\lambda_1},\ldots,r_{\lambda_{m_i}}$ then the next instruction is I_l else $I_{l'}$.

if $c_\mu = c_\lambda$ then l else l' — If registers c_μ and c_λ contain the same number then the next instruction is I_l else it is $I_{l'}$.

Conventions for sensible programmes and their application to machines were written down in [11], recall that stacking instructions may only appear in blocks as follows:

$$\left[\begin{array}{l} s := (i, r_o, \ldots, r_m) \\ I_1 \\ \cdot \\ \cdot \\ \cdot \\ I_l \\ \text{goto } i \rightarrow \\ * : r_j := r_o \\ \text{restore } (r_o, \ldots, r_{j-1}, r_{j+1}, \ldots, r_m) \end{array}\right.$$

Note that only (and all) algebra registers are stored in the stack, <u>not</u> counting registers. Furthermore $I_1, \ldots, I_l$ are operational instructions involving only algebra registers.

Finally there is the halting instruction H or, in case stacking operations are used, halting block: if $s = \emptyset$ then H else *. We give them code <4>.

A programme referring only to algebra registers is called a <u>fap</u>, one which also refers to counting registers is called a <u>fapC</u> If in addition stacking operations are used we obtain a fapS and fapCS, respectively.

$f \in P(A^n, A)$ is <u>fap-computable</u> (<u>fapS-computable</u>) if there is a fap (fapS) together with an associated machine which computes f using r_o as output register and $r_1, \ldots, r_n$ as input registers.

Each function in FAPS(A) (and hence in FAP(A)) is indexed by a number in a natural way. Suppose $f \in P(A^n, A)$ is computed by a fapS $(I_1, \ldots, I_k)$ then an index for f is $\langle n, \ulcorner I_1 \urcorner, \ldots, \ulcorner I_k \urcorner \rangle$ where $\ulcorner I_i \urcorner$ is the code assigned above to instruction I_i.

Any coding of these programmes which allows a recursive decompostion into programme parameters and codes for instructions, and from these calculation of the numerical parameters characterising the instructions listed previously, may be called a <u>standard coding</u> of the programmes. When formalised such a coding can be shown to be unique up to recursive equivalence in the Mal'cev-Ershov theory of computable numberings, see Ershov [2,3].

Let $f \in P(\omega^n \times A^m, A)$ or $f \in P(\omega^n \times A^m, \omega)$. f is said to be <u>fapC-computable</u> (<u>fapCS-computable</u>) if there is a fapC (fapCS) together with an associated machine which using the following conventions computes f : Input registers are $c_1, \ldots, c_n, r_1, \ldots, r_m$ and output register is r_o if $\mathrm{im}(f) \subseteq A$ and c_o if $\mathrm{im}(f) \subseteq \omega$. We make the assumption that initially all counting registers except the input registers contain 0. Of course, all the recursive functions on ω are fapC-computable.

It will be shown that fapC is too weak a notion to obtain a computation theory over A, the problem being that a universal function may need arbitrarily many algebra registers. One is thus naturally led to considering machines allowing a <u>potentially infinite</u> number of algebra registers. The following notions are due to Shepherdson [16].

A finite algorithmic procedure with index registers or fapir is the following modification of fapC : Instructions involving counting registers remain unchanged. Algebra registers are indexed by counting registers. Thus r_{c_λ} denotes the algebra register with subscript the content of c_λ. Instructions involving algebra registers are modified as the following samples suggest where σ is an operation of A and S a relation of A :

$$r_c := \sigma(r_{c_{\lambda_1}}, \ldots, r_{c_{\lambda_n}})$$

$$\text{if } S(r_{c_{\lambda_1}}, \ldots, r_{c_{\lambda_m}}) \text{ then } 1 \text{ else } 1' .$$

The class of fapir-computable functions on A is defined in the usual fashion and denoted FAPIR (A). In section four it is deduced that FAPCS(A) is FAPIR(A). Incidentally, our general class FAPS(A) is that computed by the P_R schemes of Constable & Gries [1], see [12].

Note that a fapir (as a syntactical object) is finite. Our final machine-theoretic notion, the countable algorithmic prodecure, or cap, is an extension of fap allowing possibly infinitely many instructions, the list of instructions being enumerated by a recursive function.

Finally some Algebra. The set $T[X_1,\ldots,X_n]$ of terms in the indeterminates $X_1,\ldots,X_n$ is inductively defined solely by the clauses (i) $X_1,\ldots,X_n$ are terms, (ii) if $t_1,\ldots,t_m$ are terms, and σ is an m-ary operation symbol then $\sigma(t_1,\ldots,t_m)$ is a term.

$T[X_1,\ldots,X_n]$ is assumed numerically coded uniformly in n by a standard coordinatisation $\gamma_*^n : \Omega \subset \omega \to T[X_1,\ldots,X_n]$ in the sense that γ_*^n is a surjection - henceforth abbreviated $\gamma_*^n(i) = [i]$ - Ω is recursive, and there are recursive functions which tell if a code labels an indeterminate and, if it does, which or, if it does not, indicates the leading operational symbol and calculates codes for the subterms. Such a coding is unique up to recursive equivalence in the theory of computable algebras due to Mal'cev [9].

Each term $t(X_1,\ldots,X_n)$ defines a function $A^n \to A$ by substitution of algebra elements for indeterminates. Define $E_n : \Omega \times A^n \to A$ by $E_n(i,\underline{a}) = [i](\underline{a})$.

3. The Structure A_ω.

Our main objective is to find given an algebra A a machine theoretic characterisation of the minimal computation theory over A allowing recursive (sub-)computations on the natural numbers. We adjoin ω to A, to obtain the structure A_ω, in order to use it as a code set for the computation theory. The content of theorem 3.1 is that the extended structure A_ω is the natural one to consider in this setting.

Let $A = (A;\underline{\sigma},\underline{S})$ be a relational structure. Then set $A_\omega = (A\cup\omega;\underline{\sigma},\underline{S},s,p,0)$ where s, p, 0 are the successor, predecessor and constant zero functions on ω, respectively, and are trivially defined on A. s and p will be written as +1 and -1 as usual.

3.1. Theorem

(i) $f \in FAP(A_\omega)$ iff f is fapC-computable.

(ii) $f \in FAPS(A_\omega)$ iff f is fapCS-computable.

Proof: The proof of (i) is included in the proof of (ii). For simplicity we assume $f \in P(A^n, A)$, the modifications needed for the general cases being obvious.

Let P be a programme in the language of fapS over A defining f. We construct a programme P' in the language of fapCS over A simulating P in such a way that P' defines f. According to our conventions for P, r_o is the output register, $r_1, \ldots, r_n$ are input registers and the remaining registers $r_{n+1}, \ldots, r_m$ are working registers. The programme P' uses algebra registers $s_o, \ldots, s_m$ and counting registers $c_o, \ldots, c_m, c_{m+1}, \ldots, c_{m+k}$ where k is sufficiently large to perform all needed arithmetic operations using $c_{m+1}, \ldots, c_{m+k}$. Each instruction in P is simulated by a block of instructions in P'. Each step in the execution of P corresponds to a stage in the execution of P', viz. the execution of the associated block. If r_j at a particular step contains an element of A or is empty then $r_j = s_j$ and $c_j = 0$. If on the other hand r_j contains an element of ω then $s_j = \emptyset$ and $c_j = r_j + 1$.

Here are samples of translations of instructions in P (on the left) into blocks of instruction in P' (on the right):

$r_\mu := \sigma_i(r_{\lambda_1}, \ldots, r_{\lambda_{n_i}})$,		$s_\mu := \sigma_i(s_{\lambda_1}, \ldots, s_{\lambda_{n_i}})$ $c_\mu := 0$
$r_\mu := 0$		$s_\mu := \emptyset$ $c_\mu := 1$
$r_\mu := r_\lambda + 1$		$s_\mu := \emptyset$ if $c_\lambda = 0$ then l_1 else l_2
	l_1.	$c_\mu := 0$ goto l_3
	l_2.	$c_\mu := c_\lambda + 1$
	l_3.	. . . . .

The only difficulty in the reduction involves the stacking operations: In P all registers are stored while in P' only the algebra registers are stored. c_{m+1} plays the role of a stack for registers $c_o,\dots,c_m$ using a recursive pairing scheme on ω. Given a list of operational instructions over A we perform the translation indicated above. From that we extract all instructions involving counting registers not changing their order. This list we call the obtained arithmetical instructions. The list of the remaining instructions are the algebraic instructions. With this in mind we make the following translation of a stacking block:

$s := (i; r_o,\dots,r_m)$	$c_{m+1} := \langle\langle c_o,\dots,c_m\rangle, c_{m+1}\rangle$
Operational instructions	Arithmetical instructions
goto $i \rightarrow$	$s := (i; s_o,\dots,s_m)$
$*: r_j := r_o$	Algebraic instructions
restore	goto $i \rightarrow$
$(r_o,\dots,r_{j-1},r_{j+1},\dots,r_m)$	$*: s_j := s_o$
	restore $(s_o,\dots,s_{j-1},s_{j+1},\dots,s_n)$
	$c_j := c_o$
	Restore $c_o,\dots,c_{j-1},c_{j+1},\dots,c_m$
	from c_{m+1}

Note that the stacking block in P' follows the established conventions. For stacking blocks in P it is convenient to consider stages rather than steps. The first stage ranges from the entry of a block to the exit via the "goto $i \rightarrow$ " statement and the second from the reentry to the end of the block. It should be apparent that the above block for P' properly simulates the stacking block for P.

By induction on the steps (stages) in the execution of P and P' it is easily proven that P' simulates P as intended and hence that P' calculates f.

For the converse assume f is fapCS-computable by a programme P using algebra registers $s_o,\dots,s_m$ and counting registers $c_o,\dots,c_k$. We construct a programme P' in the language of fapS over A_ω simulating P. P' uses registers $r_o,\dots,r_m,v_o,\dots,v_k,$ $t_o,\dots,t_3,w_o,\dots,w_p$ where p is sufficiently large to perform the required arithmetic operations. $s_o,\dots,s_m$ correspond to $r_o,\dots,r_m$ and $c_o,\dots,c_k$ to $v_o,\dots,v_k$. Initial instructions in P' make $v_i = 0$ for $i=0,\dots,k$. The translation of instructions in P to instructions in P' is straightforward when not within the scope of a stacking block, just replace the registers used in P by the corresponding registers in P'.

The simulation of a stacking block is problematic since only algebra registers are stored in P whereas all registers are stored in P'. Thus P' may loose information in the simulated counting registers when making a restore. The problem is resolved by P' performing each subcomputation twice, first obtaining the algebraic element and then obtaining the contents of the counting registers. Below we give the translation of a stacking block and the halting block.

$s := (i; s_o, \ldots, s_m)$

Algebraic operations

goto $i \rightarrow$

$* : \; s_j := s_o$

restore $(s_o, \ldots, s_{j-1}, s_{j+1}, \ldots s_m)$

if $s = \emptyset$ then H else $*$

$s := (i; r_o, \ldots)$

$t_o := 0$

$t_1 := r_j$

Operations involving $r_o, \ldots, r_m$

goto $i \rightarrow$

$* : \; r_j := r_o$

restore $(r_o, \ldots, r_{j-1}, r_{j+1}, \ldots)$

$s := (i'; r_o, \ldots)$

$t_o := 1$

$t_2 := r_j$

$r_j := t_1$

Operations involving $r_o, \ldots, r_m$ as above

goto $i \rightarrow$

$* \; : t_3 := r_o$

restore $(r_o, \ldots, t_2, w_o, \ldots, w_p)$

Restore $v_o, \ldots, v_k$ from t_3

if $t_1 = 0$ then l_2 else l_1

l_1. $\quad r_o := \langle v_o, \ldots, v_k \rangle$

l_2. $\quad$ if $s = \emptyset$ then H else $*$.

We leave to the reader the non-trivial exercise of proving that P' does in fact simulate P.

Q.E.D.

4. The Minimal Computation Theory

Recall from section 2 that $E_n : \Omega \times A^n \to A$ is the term evaluation function.

4.1. Proposition. $FAP(A_\omega)$ is a computation theory iff E_n is uniformly fapC-computable.

Proof: Assume $FAP(A_\omega)$ is a computation theory. The evaluation of a given term is $FAP(A_\omega)$-computable using projection functions, the basic operations, composition and permutation of arguments. In fact it is easily seen that there is a fapC-computable function $f : \omega \to C$ such that if i is a code for a term then $f(i)$ is a $FAP(A_\omega)$-index for the function evaluating the term. Thus $E_n(i,\underline{a}) \simeq \{f(i)\}(\underline{a}) \simeq U_n(f(i),\underline{a})$ which is uniformly fapC-computable by our assumption on $FAP(A_\omega)$.

The easy verifications that $FAP(A_\omega)$ in its coding, and using step counting as length function, satisfies all conditions of being a computation theory are left to the reader, except that of the existence of universal functions. The problem with the universal function, in the absence of a computable pairing scheme, is that a machine with a fixed number of registers may not be able to simulate a machine with a very large number of registers. This problem is avoided by letting the simulating machine manipulate codes for terms instead of actually performing the simulated operations, the point being that codes for terms are natural numbers for which pairing is available. Only when simulating a conditional instruction, and immediately before a halt instruction, is there a need to evaluate terms and it is for this we use the computability of E_n.

We shall give (macro) instructions for a programme which together with an associated machine computes $U_n(e,\underline{a}) \simeq \{e\}(\underline{a})$. r_o will, according to our usual conventions, serve as output register and $r_1,\ldots,r_{n+1}$ as input registers. The contents of the input registers will remain unchanged throughout a computation. As working registers we use $c,t,v_1,\ldots,v_p$, p being the maximum arity of a relation on A, and sufficiently many other registers to perform term evaluation and all recursive operations on ω. Suppose e is a (valid) index for a programme. Then e_t denotes $\ulcorner I_i \urcorner$ where $\ulcorner I_i \urcorner$ is a code for the i:th instruction of programme e, if register t contains i, $1 \leq i \leq$ number of instructions in programme e. Suppose programme e refers to the first $m+1$ registers, $m \geq n$. Then c will contain an $m+1$-tuple of codes for terms $<c_o,c_1,\ldots,c_m>$ simulating the contents of the registers used by a machine associated to the programme e, m is obtained recursively from e. $c_\mu := c_\lambda$ stands for instructions replacing the μ:th component of c by the λ:th component of c, and $c_\mu := \ulcorner \sigma_i(c_{\lambda_1},\ldots,c_{\lambda_{n_i}}) \urcorner$ stands for instructions calculating a code for the term $\sigma_i(t_{\lambda_1},\ldots,t_{\lambda_{n_i}})$ and placing it in the μ:th component of c if $c_{\lambda_j} = \ulcorner t_{\lambda_j} \urcorner$ for $j=1,\ldots,n_i$. Finally $r_\mu := TE(c_\lambda)$ denotes a sequence of instructions which evaluates the term coded by c_λ using $r_2,\ldots,r_{n+1}$ as input registers and places the result in r_μ.

Initially the programme determines whether or not e is a valid index. If not, undefined is simulated. If e is a valid index, t is set to 1, the number of registers which are to be simulated is determined and c is set to $<\ulcorner u \urcorner,\ulcorner x_1 \urcorner,\ldots,\ulcorner x_n \urcorner,\ulcorner u \urcorner,\ldots,\ulcorner u \urcorner>$, where $\ulcorner u \urcorner$ is a code for the undefined or empty term. The remaining part of the programme consists of a main programme MP and finitely many subroutines. The main program is entered once for each step simulated.

<u>MP</u> if $e_t = \ulcorner r_\mu := r_\lambda \urcorner$ then goto $OP(:=)$

if $e_t = \ulcorner r_\mu := \sigma_i(r_{\lambda_1},\ldots,r_{\lambda_{n_i}}) \urcorner$ then goto $OP(\sigma_i)$

if $e_t = \ulcorner$if $r_\mu = r_\lambda$ then 1 else 1'$\urcorner$ then goto REL(=)

if $e_t = \ulcorner$if $s_i(r_{\lambda_1},\ldots,r_{\lambda_{m_i}})$ then 1 else 1'$\urcorner$ then

goto $REL(S_i)$

$r_o := TE(c_o)$

H

<u>OP(:=)</u> $c_\mu := c_\lambda$

$t := t+1$

goto MP

<u>$OP(\sigma_i)$</u> $c_\mu := \ulcorner \sigma_i(c_{\lambda_1},\ldots,c_{\lambda_{n_i}}) \urcorner$

$t := t+1$

goto MP

<u>REL (=)</u> $v_1 := TE(c_\mu)$

$v_2 := TE(c_\lambda)$

if $v_1 = v_2$ then $t := 1$ else $t := 1'$

goto MP

<u>$REL(S_i)$</u> $v_1 := TE(c_{\lambda_1})$

.
.
.

$v_{m_i} := TE(c_{\lambda_{m_i}})$

if $S_i(v_1,\ldots,v_{m_i})$ then $t := 1$ else $t := 1'$

goto MP

It is an easy matter to prove by induction on the simulated step that the programme above with an associated machine calculates $U_n(e,\underline{a}) \simeq \{e\}(\underline{a})$. Furthermore an index for the above programme is obtained uniformly from n since by assumption an index for TE is obtained uniformly from n. And the length condition on computations is satisfied.

Q.E.D.

<u>4.2. Proposition</u>. E_n is uniformly fapCS-computable.

<u>Proof</u>: In view of 3.1, of course, we prove E_n is fapS-computable over A_ω; by theorem 2 of [11] this is equivalent to showing it is inductively definable over A_ω. Now E_n is informally recursively defined in our coding by

$$
\begin{aligned}
E_n(i,\underline{a}) &= a_j && \text{if } i \text{ codes the indeterminate } X_j; \\
&= \sigma_j(E_n(i_1,\underline{a}),\ldots,E_n(i_k,\underline{a})) && \text{if } [i] = \sigma_j([i_1],\ldots,[i_k]); \\
&= u && \text{if } i \text{ does not code a term,} \\
& && \text{or codes the empty term.}
\end{aligned}
$$

Thus E_n is defined by the induction term

$$FP[\lambda p,z,y_1,\ldots,y_n.t(p,z,y_1,\ldots,y_n)](x_o,x_1,\ldots,x_n)$$

with the evaluation $x_o = i$ and $x_j = a_j$, $1 \leq j \leq n$, and t is the algebra term informally described by

$$
\begin{aligned}
t(p,z,y_1,\ldots,y_n) &= y_j && \text{if } ind(z,j); \\
&= \underline{\sigma}_j(p(z_1,y_1,\ldots,y_n),\ldots,p(z_k,y_1,\ldots,y_n)) && \text{if } op(z,j); \\
&= \underline{u} && \text{if } empcode\ (z); \\
&= \underline{u} && \text{if } \neg TCode\ (z).
\end{aligned}
$$

where the relations ind, op, empcode, TCode are terms taking their obvious meaning and where z_j is the term for the appropriate recursive function which calculates i_j from i, for $1 \leq j \leq k$;

a rather complicated definition-by-cases construction over A and ω . The uniformity required is that of a recursive function $p : \omega \to C$ which computes the fapCS-code p(n) for E_n: this follows from the constructiveness of proposition 4.1 of [11] expessed in terms of a gödel numbering of the induction terms, a point more carefully discussed in theorem 4.4 later.

Q.E.D.

4.3. Theorem. FAPS(A_ω) is a computation theory.

Proof: 4.2 expresses the key property that term evaluation is uniformly fapCS-computable. It therefore suffices to append the proof of 4.1 by adding blocks to simulate store and restore instructions and the halting block. For this we add a working register w initialised to < > which is to simulate the stack by "stacking" codes for terms. In the main programme we delete the last two instructions and add the following conditional clauses.

if $e_t = \ulcorner s := (i;r_o,\ldots,r_m)\urcorner$ then goto STORE

if $e_t = \ulcorner$ restore $(r_o,\ldots,r_{j-1},r_{j+1},\ldots,r_m)\urcorner$ then goto RESTORE

if $e_t = \ulcorner$ if s=∅ then H else *$\urcorner$ then goto HALT

In the customary notation for pairing and unpairing on ω we add the following subroutines.

STORE $w := \langle\langle i,c\rangle,w\rangle$

$t := t+1$

goto MP

RESTORE $v_1 := (w)_o$

$w := (w)_1$

$v_2 := c_j$

$c := (v_1)_1$

$c_j := v_2$

goto MP

a rather complicated definition-by-cases construction over A and ω. The uniformity required is that of a recursive function $p:\omega\to C$ which computes the fapCS-code $p(n)$ for E_n: this follows from the constructiveness of proposition 4.1 of [11] expessed in terms of a gödel numbering of the induction terms, a point more carefully discussed in theorem 4.4 later.

Q.E.D.

4.3. Theorem. FAPS(A_ω) is a computation theory.

Proof: 4.2 expresses the key property that term evaluation is uniformly fapCS-computable. It therefore suffices to append the proof of 4.1 by adding blocks to simulate store and restore instructions and the halting block. For this we add a working register w initialised to $<>$ which is to simulate the stack by "stacking" codes for terms. In the main programme we delete the last two instructions and add the following conditional clauses.

if $e_t = \ulcorner s := (i; r_o, \ldots, r_m)\urcorner$ then goto STORE
if $e_t = \ulcorner$ restore $(r_o, \ldots, r_{j-1}, r_{j+1}, \ldots, r_m)\urcorner$ then goto RESTORE
if $e_t = \ulcorner$ if $s=\emptyset$ then H else $*\urcorner$ then goto HALT

In the customary notation for pairing and unpairing on ω we add the following subroutines.

STORE $w := \langle\langle i,c\rangle,w\rangle$
$t := t+1$
goto MP

RESTORE $v_1 := (w)_o$
$w := (w)_1$
$v_2 := c_j$
$c := (v_1)_1$
$c_j := v_2$
goto MP

<u>HALT</u> if $w = < >$ then H1 else H2

H1. $r_o := TE(c_o)$

H

H2. $t := *$ in block i where $(w)_o = <i,c>$

goto MP

Q.E.D.

<u>4.4. Theorem</u>. $FAPS(A_\omega)$ is the minimal computation theory.

<u>Proof</u>: By theorem 2 in [11] $FAPS(A_\omega) = Ind(A_\omega)$. Moreover there is a recursive function g such that if e is a code for a fapS then $g(e)$ is a gödel number for the term which is equivalent to the fapS. If t is an algebra term with free function variables among $p_1,\dots,p_k$, free algebra variables among $x_1,\dots,x_l$ then let ϕ_t be the following functional: $\phi_t(f_1,\dots,f_k,a_1,\dots,a_l) \simeq$ the value of t when $f_1,\dots,f_k,a_1,\dots,a_l$ are substituted for $p_1,\dots,p_k,x_1,\dots,x_l$. By lemma 2.2 in [11] ϕ_t is monotonic. Let Θ be a computation theory on A_ω. We will define a Θ-computable function h such that if e is a gödel number for a term t then $h(e)$ is a Θ-index for ϕ_t. This will prove the theorem for the length condition follows from the fact that the length function in $FAPS(A_\omega)$ is there computable.

Let t be a term. Then t is of the form $\underline{u}$, x, $\underline{c}$, $\underline{\sigma}(t_1,\dots,t_n)$, $\underline{DC}_S(t_1,\dots,t_n,t_{n+1},t_{n+2})$, $p(t_1,\dots,t_n)$ or $FP[\lambda p,x_1,\dots,x_n.t_o](t_1,\dots,t_n)$.

i) $t = \underline{\sigma}(t_1,\dots,t_n)$. Let ϕ_i be the functionals associated to t_i, $i=1,\dots,n$. $\phi_t(f_1,\dots,f_k,a_1,\dots,a_l) \simeq \sigma(\phi_1(f_1,..f_k,a_1,..,a_l),\dots,\phi_n(f_1,\dots,f_k,a_1,\dots,a_l))$. By several applications of composition and the iteration property a Θ-index for ϕ_t can be found uniformly from Θ-indices for $\phi_1,\dots,\phi_n$.

ii) $t = FP[\lambda p, x_1, \ldots, x_n . t_o](t_1, \ldots, t_k)$. It suffices to find a Θ-index for the functional Ψ defined by $FP[\lambda p\ x_1, \ldots, x_n . t_o]$. as a Θ-index for ϕ_t can then be constructed as in i). Let ϕ be the functional defined by t_o. ϕ is effective by the induction hypothesis. It follows from the First Recursion Theorem that Ψ is effective.

Q.E.D.

4.5. Proposition. $FAPS(A_\omega) = FAPIR(A_\omega) = CAP(A_\omega)$.

Proof: First we sketch a proof of $FAPS(A_\omega) \subseteq CAP(A_\omega)$. Given a fapS P we need construct a cap P' simulating P. The only problematic point is to simulate store and restore instructions and halting blocks. To the usual simulation and instructions for the ω-recursive operations needed append infinitely many store and restore blocks, each block using storing registers not used elsewhere in the programme. Index the store and restore blocks by (a register) q. The store part of a block will simply consist of instructions storing the marker i and registers $r_o, \ldots, r_m$ into distinct registers used only by that block and the restore part will restore the registers into $r_o, \ldots, r_m$ except for r_j, the j being indicated to the block in some way. q will contain a number indicating the depth of the simulated stack and is used to find the correct store and restore block. The simulation of a halting block will, of course, use q to determine what action to take.

The proof of $CAP(A_\omega) \subseteq FAPIR(A_\omega)$ is given in Shepherdson [16]. Thus it remains to prove $FAPIR(A_\omega) \subseteq FAPS(A_\omega)$. The ideas of the proof are based upon those of 4.1: when simulating a fapir, codes for terms are manipulated and term evaluation is invoked when necessary. Suppose P is a fapir programme using counting registers

$c_o,\dots,c_k$ and suppose P is to calculate an n-ary function. We construct a fapCS programme P' simulating P. P' will use algebra registers $r_o,\dots,r_n,v_1,\dots,v_p$ and counting registers $c_o,\dots,c_k$ and d. In addition P' will use sufficiently (but finitely) many other registers to be able to perform the required operations. d will play the same role as c in 4.1 and will be initialized with $<\ulcorner u\urcorner,\ulcorner x_1\urcorner,\dots,\ulcorner x_n\urcorner>$. TE denotes instructions for term evaluation just as in 4.1.

Each instruction in P is simulated by a block of instructions in P'. Below we give samples of how instructions in P (on the left) are translated to blocks of instructions in P' (on the right). Given 4.1 the notation for "instructions" in P' should be self-explanatory noting that the tuple in d will be extended whenever necessary by inserting $\ulcorner u\urcorner$ in the new components.

$c_\mu := c_\lambda+1$	$c_\mu := c_\lambda+1$
$r_{c_\mu} := \sigma_i(r_{c_{\lambda_1}},\dots,r_{c_{\lambda_{n_i}}})$	$d_{c_\mu} := \ulcorner\sigma_i(d_{c_{\lambda_1}},\dots,d_{c_{\lambda_{n_i}}})\urcorner$
if $S_i(r_{c_{\lambda_1}},\dots,r_{c_{\lambda_{m_i}}})$ then l else l'	$v_1 := TE(d_{c_{\lambda_1}})$ . . . $v_{m_i} := TE(d_{c_{\lambda_{m_i}}})$ if $S_i(v_1,\dots,v_{m_i})$ then (block) l else (block) l'
H	$r_o := TE(d_o)$ H

An easy induction argument shows that P' and P compute the same n-ary function.

Q.E.D.

The proof of 4.5 actually shows that for an arbitrary relational structure A, FAPCS(A) = FAPIR(A) = CAP(A).

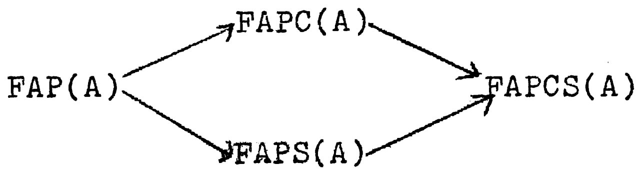
4.6. Corollary. If E_n is fapC-computable for each n then FAPC(A) = FAPIR(A).

Proof: Note that the constructed fapCS P' simulating the fapir P in the proof of 4.5 contains stacking instructions only in the routines evaluating terms. If term evaluation can in fact be performed using fapC instructions then P' is a fapC programme.

Q.E.D.

5. Examples

Obviously, the four types of function discussed in these papers are related thus



and, in connection with proposition 4.1, we have declared the customary situation in Algebra to be this

$$\mathrm{FAP}(A) \longrightarrow \mathrm{FAPS}(A) \longrightarrow \mathrm{FAPC}(A) = \mathrm{FAPCS}(A)$$

The question arises, Are these inclusions strict ?

In his original article [8], p.376, Friedman showed that FAP(A) and FAPC(A) were distinct; the relational structure he constructed is now superseded by the general analysis of [18] where examples of groups and fields A are given for which $\mathrm{FAP}(A) \subsetneq \mathrm{FAPC}(A)$.

However, we begin by using Friedman's structure A_F to separate FAPS(A) and FAPC(A), in this we are indebted to our colleague, D.Normann, for his observations reported in [6].

A_F has domain ω and a single unary operation σ defined as follows. First we define a partition C of ω by $C_1 = \{0\}$, $C_2 = \{1,2\}$, $C_3 = \{3,4,5\}$ and, in general, C_n consists of the first n numbers not in $C_1 \cup \cdots \cup C_{n-1}$. The action of σ is to permute these disjoint cycles so $\sigma \upharpoonright C_n = \{a_1, \cdots, a_n\}$ maps $a_i \to a_{i+1}$, if $i < n$, and $a_n \to a_1$; here are formulae for C and for σ.

The first number in the n-th cycle is $\frac{1}{2}n(n-1)$ and the last is $\frac{1}{2}n^2$, and the number a lies in cycle numbered
$|a| = \max\{z: \frac{1}{2}z(z-1) \leq a\}$. so

$$\begin{aligned} \sigma(a) &= a+1 && \text{if } a \neq \tfrac{1}{2}|a|^2, \\ &= \tfrac{1}{2}|a|(|a|-1) && \text{otherwise.} \end{aligned}$$

Clearly, σ is a recursive function on ω. $A_F = (\omega, \sigma)$.

<u>5.1 Theorem</u> $FAPS(A_F) \subsetneq FAPC(A_F) = FAPCS(A_F)$

<u>Proof:</u> It is straight forward to verify that term evaluation is fapC-computable and so it is enough for us to define a function $g: A_F \to A_F$ which is fapC-computable but not fapS-computable.

<u>5.2 Lemma</u> The domain of a fapS-computable function on A_F is a recursive subset of ω.

First, observe that a fapC-computable function on A_F is recursive as a function on ω because σ is recursive on ω. Secondly, we take a theorem from [18], <u>if A is a locally finite algebraic system, then the halting problem for fapS's is fapCS-decidable</u>. Thus $FAPS(A_F)$ has fapC-decidable halting problem and, in particular, the relation

$$H(e,a) \iff \{e\}(a)\downarrow$$

is recursive on ω, hence 5.2.

So let $S \subset \omega$ which is r.e. but not recursive and define $g: A_F \to A_F$ by

$$g(a) = a \quad \text{if} \quad |a| \in S$$
$$= u \quad \text{if} \quad |a| \notin S$$

the domain of which is r.e. and not recursive: by 5.2 g cannot be fapS-computable on A_F, but it is fapC-computable by this programme: let P be a fapC with domain S say with input register n_1; we need to calculate $|\ |: A_F \to \omega$ by a fapC. Notice $\sigma^{|a|}(a) = a$:

1. $r_1 := a$
2. $c := 1$
3. $r_2 := \sigma(r_1)$
4. if $r_1 = r_2$ then 8 else 5

5. c : = c+1

6. r_2: = $\sigma(r_2)$

7. goto 4

8. n_1: = c

Instructions of P with H
replaced by r_0: = r_1,H.

Q.E.D.

From the point of view of computing it is necessary to establish the incomparability of the storing facility of the stack and that of counting which, of course, no ordinary algebraic structure will exemplify; we have these examples.

5.3 Theorem There is a system A where

$$\mathrm{FAPC}(A) = \mathrm{FAP}(A) \subsetneq \mathrm{FAPS}(A) = \mathrm{FAPCS}(A).$$

Proof: Let ω_1 and ω_2 be copies of the natural numbers and set $N = \omega_1 \mathring{\cup} \omega_2$, the system has the form $A = (N; S,P,O,\sigma_1,\sigma_2,\sigma_3;R)$ where $O \in \omega_1$ and $S(a) = a+1$ if $a \in \omega_1$, $P(a) = a-1$ if $a \in \omega_1$, and

$= 0$ if $a \in \omega_2$, $= 0$ if $a \in \omega_2$

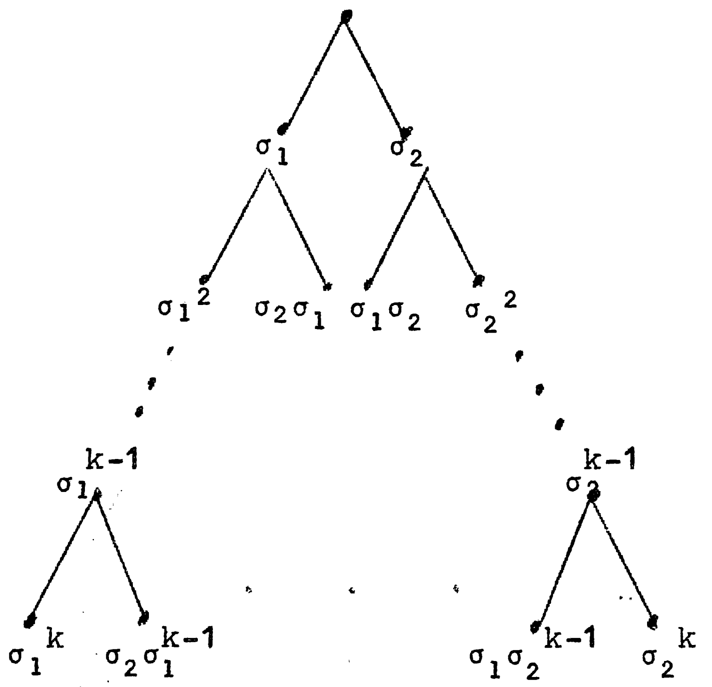
(where σ_1,σ_2 are unary operations, σ_3 is binary and R is a unary relation. We shall show how to define these operations so that the function with term

$$f(x) = FP[\lambda p,y.\ DC_S(y,y,\sigma_3(p\sigma_1(y),p\sigma_2(y)))](x) = t(x)$$

is not fap-computable over A, it is fapS-computable by 4.1 of [ ] of course; these operations will be trivial on ω_1, and defined in an irregular way on ω_2 by means of 5.1. This establishes 5.3 as FAPC(A) = FAP(A) is the observation that counting is possible in FAP(A) by using fap instructions on $(\omega_1;S,P,O)$.

Give ω_2 the partition $C_1,C_2,\cdots$ of 5.1. For each $k \in \omega$

choose $n = n(k)$ sufficiently large ($> 2^{k+1} + 2^k$) and fix the k-th element a_k of C_n, define $a_k \notin S$, thus to calculate $t(a_k)$ one has to calculate $p\sigma_1(a_k)$ and $p\sigma_2(a_k)$ whence $t(a_k) = \sigma_3(p\sigma_1(a_k), p\sigma_2(a_k))$. We now define $\sigma_1(a_k)$ and $\sigma_2(a_k)$ to be distinct elements of $C_n - \{a_k\}$ and, whatever the choice, define them to be in $\neg S$. Thus to continue to calculate $t(a_k)$ in computing $p\sigma_1(a_k)$, $p\sigma_2(a_k)$ one must first compute $p\sigma_1{}^2(a_k)$, $p\sigma_2\sigma_1(a_k)$ and $p\sigma_2{}^2(a_k)$, $p\sigma_1\sigma_2(a_k)$. This regression is continued into this tree of polynomials q, of degree $\leq k$, for which one must calculate $pq(a_k)$ in computing $t(a_k)$, call it the k-th tree:



σ_1, σ_2 are defined so that for each k, $q_1(a_k) \neq q_2(a_k)$ for q_1, q_2 different polynomials in the tree (for this $n(k) \geq 2^{k+1}$) and $\sigma_1(a) = \sigma_2(a) = 0$ when $a \neq q(a_k)$ for q in the k-th tree. S is defined by taking for each k, $S \cap C_{n(k)}$ to consist of the values of the polynomials in the k-th row on a_k and no other elements; with this S, $tq(a_k) = q(a_k)$ when q is in the k-th row. We have only to define σ_3. For each q not in the lowermost row assume $t\sigma_1 q(a_k)$, $t\sigma_2 q(a_k)$ to be defined and take $\sigma_3(t\sigma_1 q(a_k), t\sigma_2 q(a_k)) = tq(a_k)$ to be a new element in $C_{n(k)}$, not any value of operations so far defined (this requires the further 2^k elements); elsewhere σ_3 takes the value 0.

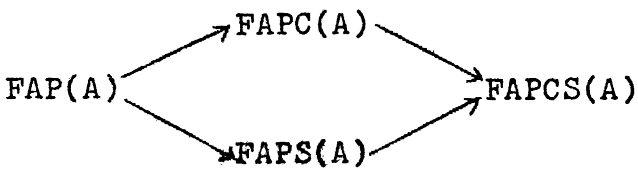
Assume f is fap-computable by programme P involving m-registers, we obtain a contradiction in showing that $f(a_m)$ requires at least m+1 registers to fap-compute. Let a_{ij} be the value of the j-th polynomial in the i-th row of the m-th tree. Consider the stage where $a_{01} = f(a_m)$ first appears in the registers of the machine M^m implementing P: by construction it arises from an instruction of the form $r_k := \sigma_3(r_i, r_j)$ with $a_{11} \in r_i$ and $a_{12} \in r_j$ - P involves at least two registers. Now consider the stage where the last of a_{11}, a_{12} first enters the machine, say it is a_{11}: prior to this the distinct elements a_{12} and a_{21}, a_{22} lie in the machine for $a_{11} = \sigma_3(a_{21}, a_{22})$ - P involves at least three registers. Considering the stage of which the latest of a_{12}, a_{21}, a_{22} first appears one can continue this regression until at least m+1 elements have been found necessary to have stored as may be easily verified.

Q.E.D.

5.4 Corollary Term evaluation E_1 is not fapC-computable over A.

Now combining 5.1 and 5.3 we can prove

5.5 Theorem There is a system A where the following inclusions are strict



Proof: Clearly it is sufficient to construct an A where FAPC(A) $\subsetneq$ FAPS(A) and FAPS(A) $\subsetneq$ FAPC(A). Let ω_1 and ω_2 be copies of the natural numbers and set $N = \omega_1 \dot{\cup} \omega_2$: such a structure is $A = (N;0,\sigma_0,\sigma_1,\sigma_2,\sigma_3;R)$ wherein σ_0 is the cycle translation function σ of 5.1 defined on ω_1, and trivially extended to ω_2 and $0,\sigma_1,\sigma_2,\sigma_3$ and R are the operations and relations defined on N in 5.3. Since $\sigma_1,\sigma_2,\sigma_3$ can be chosen recursive and A is locally finite the argument of 5.1 produces a function which is fapC-computable but not fapS-computable. And the argument of 5.3 applies directly to A to yield a function which is fapS-computable but not fapC-computable.

Q.E.D.

6. Computing with constants

To compute with the constant functions on the relational structure A is to use programmes which allow them as basic combinational operations. In this final section we reconsider the preoccupations of our two papers with the new requirement that the constant functions be computable; as we are interested in the ideas and results for comparison the details of our proofs are not included.

$f \in P(A^n,A)$ is fap*-computable if there is a fap-computable $g \in P(A^{n+m},A)$ and $\underline{b} \in A^m$ such that for each $\underline{a} \in A^n$, $f(\underline{a}) \simeq g(\underline{a},\underline{b})$. The class of all fap*-computable functions on A is denoted $FAP^*(A)$. Clearly $FAP^*(A)$ contains every constant function on A. Corresponding to fapC, fapS and fapCS there are the classes $FAPC^*(A)$, $FAPS^*(A)$ and $FAPCS^*(A)$: The relationships between the computing power of the considered classes determined in section five extend to our present setting.

The classes $Ind^*(A)$ and $DInd^*(A)$ are defined in an analogous manner from Ind(A) and DInd(A), i.e. using parameters. The main results from [11] lift directly as

6.1. Theorem

(i) $FAP^*(A) = DInd^*(A)$

(ii) $FAPS^*(A) = Ind^*(A)$.

In section four we gave a machine-theoretic characterisation of the minimal computation theory over A or strictly speaking A_ω. In order to obtain a similar characterisation of the minimal computation theory containing all constant functions it seems necessary to assume a computable pairing scheme.

(M,K,L) is a <u>pairing scheme</u> on A if M is an injection $A \times A \to A$ and K and L are the inverse functions of M, i.e. $K(M(a,b)) = a$ and $L(M(a,b)) = b$. (Observe that pairing schemes exist only on infinite structures.)

A_* is obtained from A by adjoining a pairing scheme (M,K,L) to A. Thus if $A = (A;\underline{\sigma},\underline{S})$ then $A_* = (A;\underline{\sigma},M,K,L;\underline{S})$. Our moderate aim is to find a machine-theoretic characterization of the minimal computation theory over A_* containing all constant functions.

Assume there are at least two constants in $FAP(A_*)$ say 0 and 1. Define inductively $\underline{0} = M(1,0)$ and $\underline{n+1} = M(0,\underline{n})$. It is easily seen that the elements of $\underline{\omega} = \{\underline{0},\underline{1},\underline{2},\cdots\}$ are distinct and, furthermore, the successor and predecessor operations on $\underline{\omega}$ can be expressed respectively as $\underline{n} + 1 = M(0,\underline{n})$ and $\underline{n} - 1 = DC_{\underline{=}}(\underline{n},\underline{0},\underline{0},L(\underline{n}))$: it follows that all the recursive functions on $\underline{\omega}$ are in $FAP(A_*)$. Also it is easily verified that the storing operations invested in a stack can be performed by a fap over A_*. This proves

<u>6.2. Theorem.</u>

(i) $FAP(A_*) = FAPC(A_*) = FAPS(A_*) = FAPCS(A_*)$.

(ii) $FAP^*(A_*) = FAPC^*(A_*) = FAPS^*(A_*) = FAPCS^*(A_*)$.

Thus if there is a fap-computable pairing scheme on A then all classes coincide.

The transformation from A to A_*, necessary for theorem 6.3, is not very satisfactory for not only does the transformation obliviate the distinction between the various types of functions, but the computing power is directly dependent on the particular choice of pairing scheme. It seems to us that the natural class of funtions making up a "computation theory" over A containing all constant functions is

$FAPIR^*(A)$: not in the strict sense of section one for the code set for the "computation theory" would be $\omega \times A^*$ where A^* is the set of all finite sequences of A. However, this will not be pursued further here.

6.3. Theorem. $FAP^*(A_*)$ is the minimal computation theory over A_* containing all constant functions.

Proof: Code all fap instructions by elements of $\underline{\omega} \subseteq A_*$ (using computable pairing $< ,\ldots, >_{\underline{\omega}}$ on $\underline{\omega}$) as in section two. Suppose for each $\underline{a} \in A^n$, $f(\underline{a}) \simeq g(\underline{a},\underline{b})$, where $g \in FAP(A_*)$ is computable by a fap $P = (I_1,\ldots,I_k)$. Then we code f by $<\underline{n},<\ulcorner I_1 \urcorner,\ldots,\ulcorner I_k \urcorner>_{\underline{\omega}},\underline{b}>$.

It is easily seen that term evaluation is fap-computable over A_* where an index for a term carries along the parameter $\underline{b}$ using pairing. Now we can imitate the proof of 4.1 to show that $FAP^*(A_*)$ is a computation theory. The proof of minimality is similar to that of 4.4.

Q.E.D.

REFERENCES

[1] R.C. Constable & D. Gries — On classes of program schemata. SIAM Journal on Computing 1 (1972) pp.66-118

[2] Y.L. Ershov — Theorie der Numerierungen, I. Zeitschrift für Matematische Logik und Grundlagen der Mathematik 19 (1973) pp. 289-388

[3] Y.L. Ershov — Theorie der Numerierungen, II. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 21 (1975) pp. 473-584

[4] J.E. Fenstad — On axiomatising recursion theory pp. 385-404 of J.E. Fenstad & P.G. Hinman (eds.) Generalised recursion theory, North-Holland, Amsterdam, 1974

[5] J.E. Fenstad — Computation theories: an axiomatic approach to recursion on general structures pp.143-168 of G. Müller, A. Oberschelp, & K. Potthoff (eds.) Logic conference, Kiel 1974 Springer-Verlag, Heidelberg, 1975

[6] J.E. Fenstad — On the foundation of general recursion theory: computations versus inductive definability pp. 99-111 of J.E. Fenstad, R.O. Gandy, & G.E. Sacks Generalised recursion theory II, North-Holland, Amsterdam, 1978

[7] J.E. Fenstad — Recursion theory: an axiomatic approach Springer-Verlag, Berlin, to appear.

[8] H. Friedman — Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory pp. 316-389 of R.O. Gandy & C.M.E. Yates (eds.) Logic colloquium '69, North-Holland Amsterdam, 1971

[9] A.I. Mal'cev — Constructive algebras I pp.148-212 of A.I. Mal'cev The meta-mathematics of algebraic systems. Collected papers: 1936-1967. North-Holland, Amsterdam, 1971

[10] J. Moldestad — Computations in higher types Springer-Verlag, Berlin, 1977

[11] J. Moldestad, V. Stoltenberg-Hansen & J.V. Tucker — Finite algorithmic procedures and inductive definability Matematisk institutt, Universitetet i Oslo, Preprint Series, No. 6 (ISBN 82-553-0346-4), Oslo, 1978

[12] J. Moldestad & J.V. Tucker — On the classification of computable functions in an abstract setting In preparation.

[13] Y.N. Moschovakis — Abstract first-order computability,I. Transactions American Mathematical Society 138 (1969) pp. 427-464

[14] Y.N. Moschovakis — Abstract first-order computability,II Transactions American Mathematical Society 138 (1969) pp. 465-504

[15] Y.N. Moschovakis — Axioms for computation theories - first draft pp. 119-255 of R.O.Gandy & C.M.E. Yates (eds.) Logic colloquium' 69, North-Holland, Amsterdam, 1971

[16] J.C. Shepherdson — Computation over abstract structures: serial and parallel procedures and Friedman's effective definitional schemes pp. 445-513 of H.E. Rose & J.C. Shepherdson (eds.) Logic colloquium '73, North-Holland, Amsterdam, 1975

[17] V. Stoltenberg-Hansen — Finite injury arguments in infinite computation theories Matematisk institutt, Universitetet i Oslo, Preprint Series, No. 12 (ISBN 82-553-0313-8), Oslo, 1977

[18] J.V. Tucker — Computing in algebraic systems Matematisk institutt, Universitetet i Oslo, Preprint Series, Oslo, 1978