MATH. SCAND. 46 (1980), 77–94

FINITE ALGORITHMIC PROCEDURES AND COMPUTATION THEORIES

J. MOLDESTAD, V. STOLTENBERG-HANSEN and 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.

The four kinds of functions on A considered are those functions definable by a *finite algorithmic procedure*, a *fap*, by a *fap with a stack*, a *fapS* — these were defined in the first section of [11] — by a *fap with counting*, a *fapC*, and by a *fap with both counting and stacking*, a *fapCS* — 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, 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 on A computable in the minimal computation theory over A with code set* ω.

In section four the uniqueness of the operations of stacking and counting is established by examples. And in section five we examine the situation where

* J. V. Tucker wishes to acknowledge the indispensible support of a fellowship from the European Programme of The Royal Society, London.

Received November 21, 1978; in revised form October 10, 1979.

one wants to compute with the constant functions over the structures: here we invent a new coding and encounter the necessity of adjoining pairing functions to our algebras but analogous theorems are proved.

We gratefully acknowledge the hospitality of the Matematisk institutt, Universitetet i Oslo, where these investigations were undertaken.

1. Computation theories.

Throughout we are concerned with a relational structure of the form $A = (A; \sigma_1, \ldots, \sigma_p; R_1, \ldots, R_q)$ wherein the operations and relations are finitary; 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, the ordinals, partially defined, $|e; a|\downarrow \Leftrightarrow \{e\}(a)\downarrow$, 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 the (functions which correspond to) successor, predecessor and zero on ω.

II. θ contains these generating functions:
(i) for each n and $1 \leqq i \leqq n$ the projection function $U_i^n(a_1, \ldots, a_n) = a_i$ with θ-uniform codes $p_1(n, i)$;
(ii) each operation σ of A;
(iii) for each relation R of A the definition-by-cases function defined

$$DC_R(a, x, y) = x \quad \text{if} \quad R(a)$$
$$= y \quad \text{if} \quad \neg R(a).$$

III. θ is uniformly closed under
(i) *the composition of functions*: if f and g are $n+1$ and n-ary θ-computable functions with codes $\hat{f}, \hat{g}$ respectively then their composition defined $C(f, g)(a) = f(g(a), a)$ is θ-computable with θ-uniform code $p_2(n, \hat{f}, \hat{g})$.
(ii) *the permuting of arguments*: let ${}^j a = (a_j, a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n)$ when $a = (a_1, \ldots, a_n)$. If f is an n-ary θ-computable function with code $\hat{f}$ then, for each $1 \leqq j \leqq n$, the function defined ${}^j f(a) = f({}^j a)$ is θ-computable with θ-uniform code $p_3(n, j, \hat{f})$.
(iii) *the addition of arguments*: if f is an n-ary θ-computable function with

code $\hat{f}$ then, for any m, the $(n+m)$-ary function g defined $g(a,b)=f(a)$ is θ-computable with θ-uniform code $p_4(n,m,\hat{f})$.

IV. θ contains *universal functions* U_n such that for $e \in C$, $a \in A^n$

$$U_n(e,a) = \{e\}(a)$$

with θ-uniform codes $p_5(n)$.

V. θ enjoys this *iteration property*: for each n, m there is a θ-computable map, S_m^n, with θ-uniform code $p_6(n,m)$, such that for $e \in C$, $a \in C^n$, $b \in A^m$

$$\{S_m^n(e,a)\}(b) = \{e\}(a,b) .$$

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: $|(p_2(n,\hat{f},\hat{g});\, a)| > \max\{|(\hat{f};\, g(a),a)|, |(\hat{g};\, a)|\}$.
(ii) Permutation: $|(p_3(n,j,\hat{f});\, a)| > |(\hat{f};\, {}^j a)|$.
(iii) Addition: $|(p_4(n,m,\hat{f});\, a)| > |(\hat{f};\, a)|$.
(iv) Universality: $|(p_5(n);\, e,a)| > |(e;\, a)|$.
(v) Iteration: $|(S_m^n(e,a);\, b)| > |(e;\, a,b)|$.

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-Hansen's [18]. 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 $\varphi\colon P(A^{n_1},A)\times\cdots\times P(A^{n_k},A)\times A^m \times A^n \to A$ is θ-*effective* over A iff there exists a θ-code $\hat{\varphi}$ such that for any appropriate $e_1,\ldots,e_k$,

$$\varphi(\{e_1\},\ldots,\{e_k\},b,a) = \{\hat{\varphi}\}(e_1,\ldots,e_k,b,a)$$

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

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, $\varphi(\underline{f},b)(a)=\varphi(\underline{f},b,a)$, in section

three. In connection with Theorem 2.1 of [11] we shall assume this delicate form of the

1.1. FIRST RECURSION THEOREM. *If φ is θ-effective and monotonic as $\Phi(\underline{f}, b)$, and if the $\underline{f}$ are θ-computable, then the least fixed point $\varphi(\underline{f}, b)^*$ 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 *subcomputation theory* of Φ iff $\theta \subset \Phi$ and there exists a Φ-computable map $p: \omega \times C \to C$ such that for each $e \in C$, $a \in A^n\{e\}(a) = \{p(n, e)\}(a)$ and, of course, $|(e; a)|_\theta \leqq |(p(n, e), a)|_\Phi$.

θ is said to be a *minimal computation theory over A with code set C* 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].

An *A-register machine with counting* (or *A-register machine with counting and stacking*) sees a finite number of counting registers, designed to hold natural numbers, added to an A-register machine (or A-register machine with stack) with the capacity to implement the basic operations of ω. Programmes for such machines are written in the following language. Variables are $r_0, r_1, \ldots$ for *algebra registers* and $c_0, c_1, \ldots$ for *counting registers*. s denotes the stack register. Function and relation symbols are those in the signature of the relational structure A. In addition there are function symbols for successor $(+1)$, and predecessor $(\dot{-}1)$, and the constant zero function on the natural numbers.

A *finite algorithmic procedure with counting*, or *fapC*, is an ordered, finite list of instructions $(I_1, \ldots, I_k)$ each instruction being either a fap instruction or one of these *counting instructions*:

$c_\mu := c_\lambda + 1$ meaning "add one to the contents of c_λ and place that value in c_μ".

$c_\mu := c_\lambda \dot{-} 1$ meaning "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".

$c_\mu := 0$ meaning "replace the contents of c_μ by 0".

if $c_\mu = c_\lambda$ **then** i **else** j meaning "if registers c_μ and c_λ contain the same number

then the next instruction is I_i otherwise if they contain distinct numbers it is I_j".

Similarly, a *finite algorithmic procedure with counting and stacking*, or *fapCS*, is defined to be a programme which interleaves fapS and these counting instructions with the single new convention that no counting instruction may appear in any stacking block.

Let $f \in P(A^n \times \omega^m, A)$ or $f \in P(A^n \times \omega^m, \omega)$. f is said to be *fapC-computable* (*fapCS-computable*) if there is a fapC (fapCS) together with an associated machine which using the following conventions computes f: Input registers are $r_1, \ldots, r_n$, $c_1, \ldots, c_m$ and output register is r_0 if $\text{im}(f) \subseteqq A$ and c_0 if $\text{im}(f) \subseteqq \omega$. 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 *potentially infinite* number of algebra registers. The following notions are due to Shepherdson [17] to whom reference must be made for their formal definition.

A *finite algorithmic procedure with index registers*, or *fapir*, is the following modification of a fapC. Algebra registers are to be indexed by counting registers, thus r_{c_λ} denotes the algebra register with subscript the content of c_λ. Instructions involving counting registers remain unchanged, but instructions involving algebra registers are modified as the following examples suggest. Let σ be a k-ary operation of A and R a k-ary relation of A.

$r_\mu := \sigma(r_{c_{\lambda_1}}, \ldots, r_{c_{\lambda_k}})$ meaning "replace the contents of r_μ by the result of applying σ to the contents of the registers indexed by the contents of counting registers $c_{\lambda_1}, \ldots, c_{\lambda_k}$".

if $R(r_{c_{\lambda_1}}, \ldots, r_{c_{\lambda_k}})$ **then** i **else** j which takes its obvious meaning.

The class of *fapir-computable* functions on A is defined in the usual fashion and denoted FAPIR (A). In section three it is deduced that FAPCS (A) is FAPIR (A).

Note that a fapir (as a syntactical object) is finite. The *countable algorithmic procedure*, or *cap*, is an extension of fap allowing possibly infinitely many instructions, the list of instructions being enumerated by a recursive function. CAP (A) denotes the set of all *cap-computable* functions on A.

Each instruction I can be naturally encoded by a natural number $\ulcorner I \urcorner$ thus inducing a numerical coding of programmes $(I_1, \ldots, I_k)$ as numbers $\langle \ulcorner I_1 \urcorner, \ldots, \ulcorner I_k \urcorner \rangle$ where $\langle , \ldots , \rangle$ is some recursive pairing function. Any coding of these programmes which allows a recursive decomposition into programme parameters and codes for instructions, and from these calculation of the numerical parameters characterising the instructions listed previously may be called a *standard coding* 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 Mal'cev [9] and Ershov [2, 3].

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

$T[X_1,\ldots,X_n]$ is assumed numerically coded uniformly in n by a *standard coordinatisation* $\gamma^n_*\colon \Omega_n \subset \omega \to T[X_1,\ldots,X_n]$ in the sense that γ^n_* is a surjection — henceforth abbreviated $\gamma^n_*(i)=[i]-\Omega_n$ 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 $E_n\colon \Omega_n \times A^n \to A$ by $E_n(i,a) = [i](a)$.

3. The minimal computation theory.

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 a computation theory over A.

Let $A=(A;\sigma_1,\ldots,\sigma_p,R_1,\ldots,R_q)$ be a relational structure. Then set

$$A_\omega = (A \cup \omega;\sigma_1,\ldots,\sigma_p,R_1,\ldots,R_q,+1,\dot{-}1,0,=)$$

where $+1$, $\dot{-}1$, 0 are the successor, predecessor and constant zero function on ω, respectively, $=$ is equality on ω and all are trivially defined on A. That the fapC and fapCS computable functions over A can be investigated over the extended structure A_ω is guaranteed by the following:

3.1. THEOREM. *Suppose* $f \in P(A^n \times \omega^m, A)$ *or* $f \in P(A^n \times \omega^m, \omega)$. *Then*

(i) $f \in \mathrm{FAP}(A_\omega)$ *iff* f *is* fapC*-computable.*
(ii) $f \in \mathrm{FAPS}(A_\omega)$ *iff* f *is* fapCS*-computable.*

The proof of Theorem 3.1 is long and tedious and is ommitted.

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

3.2. THEOREM. $\mathrm{FAP}(A_\omega)$ *is a computation theory iff* E_n *is* fapC*-computable, uniformly in* n.

PROOF. Assume FAP (A_ω) is a computation theory. The evaluation of a given term is FAP (A_ω)-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 $f\colon \omega \to C$ such that if i is a code for a term then $f(i)$ is a FAP (A_ω)-index for the function evaluating the term. Thus $E_n(i,a) = \{f(i)\}(a) = U_n(f(i),a)$ which is uniformly fapC-computable by our assumption on FAP (A_ω).

The easy verifications that FAP (A_ω) 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 $U_n(e,a) = \{e\}(a)$. r_0 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 $c, t, v_1,\ldots,v_p$, here p is the maximum arity of a relation of 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. The e_t denotes $\ulcorner I_i \urcorner$ where $\ulcorner I_i \urcorner$ is a code for the ith instruction of programme e, if register t contains $i, 1 \leqq i \leqq$ number of instructions in programme e. Suppose programme e refers to the first $m+1$ registers, $m \geqq n$. Then c will contain an $m+1$-tuple of codes for terms $\langle c_0, c_1,\ldots,c_m\rangle$ 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 μth component of c by the λth component of c, and $c_\mu := \ulcorner \sigma(c_{\lambda_1},\ldots,c_{\lambda_k}) \urcorner$ stands for instructions calculating a code for the term $\sigma(t_{\lambda_1},\ldots,t_{\lambda_k})$ and placing it in the μth component of c if $c_{\lambda_j} = \ulcorner t_{\lambda_j} \urcorner$ for $j = 1,\ldots,k$. 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 $\langle \ulcorner u \urcorner, \ulcorner x_1 \urcorner,\ldots, \ulcorner x_n \urcorner, \ulcorner u \urcorner,\ldots, \ulcorner u \urcorner\rangle$, 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 programme is entered once for each step simulated.

MP **if** $e_t = \ulcorner r_\mu := r_\lambda \urcorner$ **then goto** $OP(:=)$
if $e_t = \ulcorner r_\mu := \sigma(r_{\lambda_1}, \ldots, r_{\lambda_k}) \urcorner$ **then goto** $OP(\sigma)$
if $e_t = \ulcorner$**if** $R(r_{\lambda_1}, \ldots, r_{\lambda_k})$ **then** i **else** $j \urcorner$ **then goto** $REL(R)$
$r_0 := TE(c_0)$
H

$OP(:=)$ $c_\mu := c_\lambda$
$t := t+1$
goto MP

$OP(\sigma)$ $c_\mu := \ulcorner \sigma(c_{\lambda_1}, \ldots, c_{\lambda_k}) \urcorner$
$t := t+1$
goto MP

$REL(R)$ $v_1 := TE(c_{\lambda_1})$
$\vdots$
$v_k := TE(c_{\lambda_k})$
if $R(v_1, \ldots, v_k)$ **then** $t := i$ **else** $t := j$
goto MP.

It is an easy matter to prove by induction on the simulated step that the programme above with an associated machine calculates $U_n(e, a) = \{e\}(a)$. 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.

3.3. THEOREM. *E_n is* fapCS-*computable, uniformly in n.*

PROOF. 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

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

Thus E_n is defined by the induction term

$$FP[\lambda p, z, y_1, \ldots, y_n . t(p, z, y_1, \ldots, y_n)](x_0, x_1, \ldots, x_n)$$

with the evaluation $x_0 = i$ and $x_j = a_j$, $1 \leqq j \leqq n$, and t is the algebra term informally described by

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

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 $1 \leqq j \leqq k$; a rather complicated definition-by-cases construction over A and ω. The uniformity required is that of a recursive function $p\colon \omega \to C$ which computes the fapCS-code $p(n)$ for E_n: this follows from the constructiveness of proposition 3.1 of [11] expressed in terms of gödel numbering of the induction terms, a point more carefully discussed in Theorem 3.5 later.

3.4. THEOREM. FAPS (A_ω) *is a computation theory.*

PROOF. Theorem 3.3 expresses the key property that term evaluation is uniformly fapCS-computable. It therefore suffices to append the proof of Theorem 3.2 by adding blocks to simulate store and restore instructions and the halting block. For this we add a working register w initialised to $\langle\,\rangle$ which is to simulate the stack by "stacking" codes for terms. In the main programme we delete the last two imstructions and add the following conditional clauses.

if e_t = ⌜$s := (i; r_0, \ldots, r_m)$⌝ **then goto** STORE
if e_t = ⌜**restore** $(r_0, \ldots, r_{j-1}, r_{j+1}, \ldots, r_m)$⌝ **then goto** RESTORE
if e_t = ⌜**if** $s = \varnothing$ **then** H **else** *⌝ **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 $v_1 := (w)_0$
$w := (w)_1$
$v_2 := c_j$
$c := (v_1)_1$
$c_j := v_2$
goto MP

HALT **if** $w = \langle\,\rangle$ **then** $H1$ **else** $H2$

$H1$. $r_0 := TE(c_0)$
H

$H2$. $t :=$ * in block i where $(w)_0 = \langle i, c\rangle$
goto MP.

Notice that, by adding an independent counter to MP, the step counting function $\text{Step}(e,a)=|e;a|$ becomes fapCS-computable over A.

3.5. THEOREM. $\text{FAPS}(A_\omega)$ *is the minimal computation theory over* A_ω.

PROOF. By Theorem 2 in [11], $\text{FAPS}(A_\omega)=\text{IND}(A_\omega)$. 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 $p_1,\ldots,p_n$, free algebra variables among $x_1,\ldots,x_l$ then let φ_t be the following functional: $\varphi_t(f_1,\ldots,f_n,a_1,\ldots,a_l)\cong$ the value of t when $f_1,\ldots,f_n,a_1,\ldots,a_l$ are substituted for $p_1,\ldots,p_n,x_1,\ldots,x_l$. By [11, section 2] φ_t is monotonic. Let θ be a computation theory over 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 $\text{FAPS}(A_\omega)$ is there computable.

Let t be a term. Then t is of the form $\underline{u}$, x, c, $\underline{\sigma}(t_1,\ldots,t_k)$, $\underline{DC}_R(t_1,\ldots,t_k,t_{k+1},t_{k+2})$, $p(t_1,\ldots,t_k)$ or $FP[\lambda p,x_1,\ldots,x_k.t_0](t_1,\ldots,t_k)$.

(i) $t=\sigma(t_1,\ldots,t_k)$. Let φ_i be the functionals associated to t_i, $i=1,\ldots,k$. $\varphi_t(f_1,\ldots,f_n,a_1,\ldots,a_l)\cong\sigma(\varphi_1(f_1,\ldots,f_n,a_1,\ldots,a_l),\ldots,\varphi_k(f_1,\ldots,f_n,a_1,\ldots,a_l))$. By several applications of composition and the iteration property a θ-index for φ_t can be found uniformly from θ-indices for $\varphi_1,\ldots,\varphi_k$.

(ii) $t=FP[\lambda p,x_1,\ldots,x_k.t_0](t_1,\ldots,t_k)$. It suffices to find a θ-index for the functional Ψ defined by $FP[\lambda p,x_1,\ldots,x_k.t_0]$ as a θ-index for φ_t can then be constructed as in (i). Let φ be the functional defined by t_0, φ is effective by the induction hypothesis. It follows from the First Recursion Theorem that Ψ is effective.

3.6. THEOREM. $\text{FAPS}(A_\omega)=\text{FAPIR}(A_\omega)=\text{CAP}(A_\omega)$.

PROOF. First we sketch a proof of $\text{FAPS}(A_\omega)\subseteqq\text{CAP}(A_\omega)$. Given a fapS P we need construct a cap P' simulating P. The only problematical point is to simulate store and restore instructions and halting blocks. To the usual similation 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 $r_0,\ldots,r_m$ into distinct registers used only by that block and the restore part will restore the registers into $r_0,\ldots,r_m$ 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 $\mathrm{CAP}(A_\omega) \subseteqq \mathrm{FAPIR}(A_\omega)$ is given in Shepherdson [16]. Thus it remains to prove $\mathrm{FAPIR}(A_\omega) \subseteqq \mathrm{FAPS}(A_\omega)$. The ideas of the proof are based upon those of 3.2: 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 $c_0, \ldots, c_k$ and suppose P is to calculate an n-ary function. We construct a fapCS programme P' simulating P. P' will use algebra registers $r_0, \ldots, r_n, v_1, \ldots, v_p$ and counting registers $c_0, \ldots, c_k$ 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 Theorem 3.2 and will be initialised with $\langle \ulcorner u \urcorner, \ulcorner x_1 \urcorner, \ldots, \ulcorner x_n \urcorner \rangle$. TE denotes instructions for term evaluation just as in 3.2.

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 Theorem 3.2 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.

$c_\mu := c_\lambda + 1$	$c_\mu := c_\lambda + 1$
$r_{c_\mu} := \sigma(r_{c_{\lambda_1}}, \ldots, r_{c_{\lambda_k}})$	$d_{c_\mu} := \ulcorner \sigma(d_{c_{\lambda_1}}, \ldots, d_{c_{\lambda_k}}) \urcorner$
if $R(r_{c_{\lambda_1}}, \ldots, r_{c_{\lambda_k}})$ **then** i **else** j	$v_1 := TE(d_{c_{\lambda_1}})$
	$\vdots$
	$v_k := TE(d_{c_{\lambda_k}})$
	if $R(v_1, \ldots, v_k)$ **then** (block) i **else** (block) j
H	$r_0 := TE(d_0)$
	H.

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

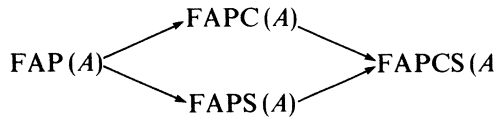
The proof of Theorem 3.6 actually shows that for an arbitrary relational structure A, $\mathrm{FAPCS}(A) = \mathrm{FAPIR}(A) = \mathrm{CAP}(A)$.

3.7. COROLLARY. *If* E_n *is* fapC-*computable for each* n *then* $\mathrm{FAPC}(A) = \mathrm{FAPIR}(A)$.

PROOF. Note that the constructed fapCS P' simulating the fapir P in the proof of 3.6 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.

4. Examples.

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



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 [19] where examples of groups and fields A are given for which FAP$(A) \subsetneqq$ FAPC(A). However, we begin by using Friedman's structures 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, the relation of equality on 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 $C_1 \cup \ldots \cup C_{n-1}$. The action of σ is to permute these disjoint cycles so $\sigma | C_n = \{a_1, \ldots, a_n\}$ 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 nth cycle is $\frac{1}{2}n(n-1)$ and the last is $\frac{1}{2}(n-1)(n+2)$, and the number a lies in cycle numbered $|a| = \max\{z : \frac{1}{2}z(z-1) \leqq a\}$. So

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

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

4.1. THEOREM. FAPS$(A_F) \subsetneqq$ FAPC(A_F) = FAPCS(A_F).

PROOF. It is straight forward to verify that term evaluation is fapC-computable and so it is enough for us to define a function $g\colon A_F \to A_F$ which is fapC-computable but not fapS-computable.

4.2. LEMMA. *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 [19], *if A is a locally finite algebraic system, then the halting problem for* fapS*'s is*

fapCS-*decidable*. Thus FAPS(A_F) has fapC-decidable halting problem and, in particular, the relation

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

is recursive on ω, hence 4.2.

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

$$\begin{aligned} g(a) &= a \quad \text{if } |a| \in R \\ &= \uparrow \quad \text{if } |a| \notin R \end{aligned}$$

the domain of which is r.e. and not recursive: by 4.2. g cannot be fapS-computable on A_F, but it is fapC-computable since $|.|: A_F \to \omega$ is fapC-computable over A_F (as $\sigma^{|a|}(a) = a!$).

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, in fact, *no ordinary* algebraic structure will exemplify; we have these examples.

4.3. THEOREM. *There is a relational system A where*

$$\text{FAPC}(A) = \text{FAP}(A) \subsetneqq \text{FAPS}(A) = FAPCS(A)\,.$$

PROOF. Let ω_1 and ω_2 be copies of the natural numbers and set $N = \omega_1 \dot{\cup} \omega_2$, the system has the form $A = (N; S, P, 0, \sigma_1, \sigma_2, \sigma_3, =, R)$ where $0 \in \omega_1$ and

$$\begin{aligned} S(a) &= a+1 \quad \text{if } a \in \omega_1, \qquad P(a) = a-1 \quad \text{if } a \in \omega_1\,, \\ &= 0 \quad \text{if } a \in \omega_2, \qquad\quad = 0 \quad \text{if } a \in \omega_2 \end{aligned}$$

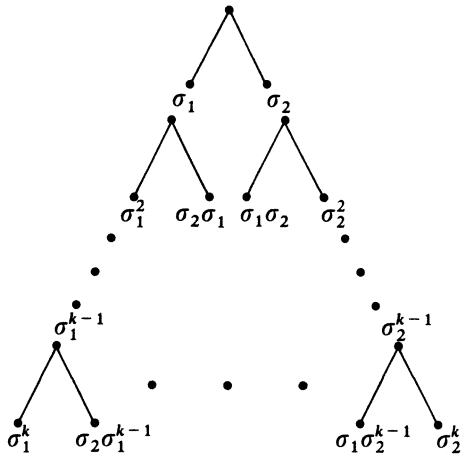
where σ_1, σ_2 are unary operations, σ_3 is binary and R is a unary relation. We shall show how to define these operations so that the function with term

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

is not fap-computable over A—it is fapS-computable by [11, 3.1] of course; these operations will be trivial on ω_1, and defined in an irregular way on ω_2 by means of 4.1. This establishes 4.3 as FAPC(A)=FAP(A) is the observation that counting is possible in FAP(A) by using fap instructions on $(\omega_1; S, P, 0)$.

Give ω_2 the partition $C_1, C_2, \ldots$ of 4.1. For each $k \in \omega$ choose $n = n(k)$ sufficiently large ($> 2^{k+1} + 2^k$) and fix the kth element a_k of C_n. Define $a_k \notin R$, 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 $\neg R$. 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 $\leqq k$, for which one must calculate $pq(a_k)$ in computing $t(a_k)$; call it the kth tree:



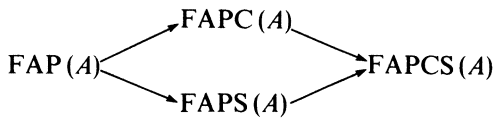
σ_1, σ_2 are defined so that for each k, $q_1(a_k) \neq q_2(a_k)$ for q_1, q_2 different polynomials in the tree (for this $n(k) \geqq 2^{k+1}$) and $\sigma_1(a) = \sigma_2(a) = 0$ when $a \neq q(a_k)$ for q in the kth tree. R is defined by taking for each k, $R \cap C_{n(k)}$ to consist of the values of the polynomials in the kth row on a_k and no other elements; with this R, $tq(a_k) = q(a_k)$ when q is in the kth 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 jth polynomial in the ith row of the mth three. 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.

4.4. COROLLARY. *Term evaluation E_1 is not* fapC*-computable over A.*

Now combining 4.1 and 4.3 we can prove

4.5. THEOREM. *There is a structure A where the following inclusions are strict*



PROOF. Clearly it is sufficient to construct an A where $\mathrm{FAPC}(A) \not\subseteq \mathrm{FAPS}(A)$ and $\mathrm{FAPS}(A) \not\subseteq \mathrm{FAPC}(A)$. 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 4.1 defined on ω_1, and trivially extended to ω_2 and 0, σ_1, σ_2, σ_3 and $=$, R are the operations and relations defined on N in 4.3. Since $\sigma_1, \sigma_2, \sigma_3$ can be chosen recursive and A is locally finite the argument of 4.1 produces a function which is fapC-computable but not fapS-computable. And the argument of 4.3 applies directly to A to yield a function which is fapS-computable but not fapC-computable.

Actually, the customary situation in Algebra is

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

because term evaluation is fapC-computable for semigroups, groups, associative rings, Lie rings, lattices and so forth. This fact can be expressed as a general theorem about varieties of algebraic systems, see Tucker's [19].

5. 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_K*-computable* if there is a fap-computable $g \in P(A^{n+m}, A)$ and $b \in A^m$ such that for each $a \in A^n$, $f(a) \cong g(a, b)$. The class of all fap_K-computable functions on A is denoted $\mathrm{FAP}_K(A)$. Clearly $\mathrm{FAP}_K(A)$ contains every constant function on A. Corresponding to fapC, fapS and fapCS there are the classes $\mathrm{FAPC}_K(A)$, $\mathrm{FAPS}_K(A)$ and $\mathrm{FAPS}_K(A)$: The relationships

between the computing power of the considered classes determined in section four extend to our present setting.

The classes $\mathrm{IND}_K(A)$ and $\mathrm{DIND}_K(A)$ are defined in an analogous manner from $\mathrm{IND}(A)$ and $\mathrm{DIND}(A)$, i.e. using parameters. The main results from [11] lift directly as

5.1. THEOREM.
i) $\mathrm{FAP}_K(A) = \mathrm{DIND}_K(A)$
(ii) $\mathrm{FAPS}_K(A) = \mathrm{IND}_K(A)$.

In section three 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 *pairing scheme* 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 $L(M(a,b)) = b$, for all $a, b \in A$. (Observe that pairing schemes exist only on infinite structures.)

A_* is obtained from A by adjoining a pairing scheme (M, K, L) to A. A moderate aim is to find a machine-theoretic characterisation of the minimal computation theory over A_* containing all constant functions. We now assume equality on A is among the basic relations of A.

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

5.2. THEOREM.
(i) $\mathrm{FAP}(A_*) = \mathrm{FAPC}(A_*) = \mathrm{FAPS}(A_*) = \mathrm{FAPCS}(A_*)$.
(ii) $\mathrm{FAP}_K(A_*) = \mathrm{FAPC}_K(A_*) = \mathrm{FAPS}_K(A_*) = \mathrm{FAPCS}_K(A_*)$.

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

The transformation from A to A_*, necessary for Theorem 5.3, 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.

5.3. THEOREM. $\mathrm{FAP}_{\mathrm{K}}(A_*)$ *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 $\langle , \ldots , \rangle_{\underline{\omega}}$ on $\underline{\omega}$) as in section two. Suppose for each $a \in A^n, f(a) \cong g(a, b)$, where $g \in \mathrm{FAP}(A_*)$ is computable by a fap $(I_1, \ldots, I_k)$. Then we code f by $\langle \underline{n}, \langle \ulcorner I_1 \urcorner, \ldots, \ulcorner I_k \urcorner \rangle_{\omega}, b \rangle$.

It is easily seen that term evaluation is fap-computable over A_* where an index for a term carries along the parameter b using pairing. Now we can imitate the proof of 3.2 to show that $\mathrm{FAP}_{\mathrm{K}}(A_*)$ is a computation theory. The proof of minimality is similar to that of 3.5.

The application of a pairing hypothesis, as above, has many precedents, but remains suspect. For example, it is not difficult to find a structure A where

$$\mathrm{FAPCS}_{\mathrm{K}}(A) \subsetneqq \bigcap_{\text{all pairings}^*} \mathrm{FAP}_{\mathrm{K}}(A_*)$$

and where, indeed, *any* pairing implies computable search for A.

It seems to us that the natural class of functions making up a "computation theory" over A containing all constant functions is $\mathrm{FAPIR}^*(A)$: 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.

Readers of [11] and this paper may care to know that several other disparate methods of defining computability in the setting of an indefinite algebraic system have beeen identified in terms of the fap formalism. These include, most notably, *equational definability*; the *set recursion* of D. Normann [16]; the *program schemata* of Theoretical Computer Science, see Constable and Gries [1]. For a systematic treatment of this classification and a discussion of the resulting Church-Turing Thesis, which distinguishes the fapCS-computable functions as the definitive characterisation of those functions on an algebraic structure calculable by finite, deterministic algorithms, see [12].

REFERENCES

1. R. C. Constable and D. Gries, *On classes of ptrogram schemata*, SIAM J. Comput. 1 (1972), 66–118.
2. Y. L. Ershov, *Theorie der Numerierungen* I, Z. Math. Logik Grundlagen Math. 19 (1973), 289–388.
3. Y. L. Ershov, *Theorie der Numerierungen* II, Z. Math. Logik Grundlagen Math. 21 (1975), 473–584.

4. J. E. Fenstad, *On axiomatising recursion theory*, pp. 385–404 of J. E. Fenstad and P. G. Hinman (eds.) *Generalized 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 and K. Potthoff (eds.) *Logic conference, Kiel* 1974, Springer-Verlag, Berlin - Heidelberg - New York, 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 and G. E. Sacks (eds.), *Generalized recursion theory* II, North-Holland, Amsterdam, 1978.
7. J. E. Fenstad, *General recursion theory. An axiomatic approach*, Springer-Verlag, Berlin - Heidelberg - New York, 1980.
8. H. Friedman, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory*, pp. 316–389 of R. O. Gandy and 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 - Heidelberg - New York, 1977.
11. J. Moldestad, V. Stoltenberg-Hansen and J. V. Tucker, *Finite algorithmic procedures and inductive definability*, Math. Scand. 46 (1980), 62–76.
12. J. Moldestad and J. V. Tucker, *On the classification of computable functions in an abstract setting*, Mathematical Centre Report, Amsterdam, 1979.
13. Y. N. Moschovakis, *Abstract first-order computability* I, Trans. Amer. Math. Soc. 138 (1969), 427–464.
14. Y. N. Moschovakis, *Abstract first-order computability* II, Trans. Amer. Math. Soc. 138 (1969), 465–504.
15. Y. N. Moschovakis, *Axioms for computation theories—first draft*, pp. 119–225 of R. O. Gandy and C. M. E. Yates (eds.), *Logic colloquium* '69, North-Holland, Amsterdam, 1971.
16. D. Normann, *Set recursion*, pp. 303–320 of J. E. Fenstad, R. O. Gandy and G. E. Sacks (eds.), *Generalised recursion theory* II, North-Holland, Amsterdam, 1978.
17. J. C. Shepherdson, *Computation over abstract structures: serial and parallel procedures and Friedman's effective definitional schemes*, pp. 445–513 of H. E. Rose and J. C. Shepherdson (eds.), *Logic colloquium* '73, North-Holland, Amsterdam, 1975.
18. V. Stoltenberg-Hansen, *Finite injury arguments in infinite computation theories*, Ann. Math. Logic 16 (1979), 57–80.
19. J. V. Tucker, *Computing in algebraic systems*, Matematisk institutt, Universitetet i Oslo, Preprint Series, No. 12 (ISBN 82-553-0358-8), Oslo, 1978.

UNIVERSITETET I OSLO, MATEMATISK INSTITUTT
BLINDERN, OSLO 3
NORWAY

UPPSALA UNIVERSITET, MATEMATISKA INSTITUTIONEN
THUNBERGSVÄGEN 3, S-752 38 UPPSALA
SWEDEN

MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49, 1091 AL AMSTERDAM
THE NETHERLANDS