MATH. SCAND. 46 (1980), 62–76

FINITE ALGORITHMIC PROCEDURES AND INDUCTIVE DEFINABILITY

J. MOLDESTAD, V. STOLTENBERG-HANSEN and J. V. TUCKER*

The purpose of this article, along with its sequel [13], is to analyse various notions of computable functions over a relational structure A based upon machine-theoretic ideas. It is expected that the results obtained will be pertinent to considerations of strategies for perfecting a general recursion/computability theory in such an abstract setting, and that the rôles of arithmetic and pairing (though not that of search operators) together with the precise relationship between computing commitments will be clarified; references in mind are Feferman [2] and Fenstad [5, 6].

But it is not to these concerns of *theoria* that these articles are committed exclusively, rather the aim is to create a theory of computing in algebraic systems which appeals to an algebraist's turn of mind and which can be used in algebraic investigations, where questions of definability, constructiveness and complexity are involved; see Tucker's introductory paper [19].

Associated with a structure A is the family of A-register machines which can perform the basic operations, decide the basic relations of A, and perform some simple combinatorial operations. Each such machine defines a partial function on A in the usual fashion; the class of all such functions is denoted FAP (A). Extensions of this class are obtained by refining the capabilities of the computing device: allowing certain enlargements of the machine's storage facilities, or by allowing subcomputations on the natural numbers ω, or by arranging both. The classes of functions thus obtained are denoted respectively by FAPS (A), FAPC (A) and FAPCS (A).

H. Friedman was the first to consider FAP (A) and FAPC (A) in his fundamental paper [7] whereas FAPS (A) and FAPCS (A) are our own inventions though they have been identified and studied in variant forms by Constable and Gries [1] in the former case and Shepherdson [18] in the latter.

In this paper the classes FAP (A) and FAPS (A) are characterised in terms of the *inductively definable*, or *inductive*, functions over A, in the sense of Platek's

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

Received March 23, 1979; in revised form October 10, 1979.

[17]. In section one, the machine-theoretic notions are properly defined whilst the *inductive* functions on A, IND (A), and the *directly inductive* functions on A, DIND (A), are discussed in section two. In sections three and four we prove

THEOREM 1. DIND $(A) =$ FAP (A).

THEOREM 2. IND $(A) =$ FAPS (A).

In the companion paper [13] we examine machine computable functions from the point of view of the axiomatic analysis of recursion theory, that of Moschovakis [14, 15, 16] and, in particular, Fenstad [3, 4, 6]. There the central classes prove to be FAPC (A) and FAPCS (A) for these, it is shown, characterize minimal computing strengths on A necessary to generate workable computation theories (in the large).

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

1. Finite algorithmic procedures.

First, the few ideas from the theory of universal algebras used in this paper and in its companion are to be found in, for example, Mal'cev's book [10]. The relational structures considered are of the form $A = (A; \sigma_1, \ldots, \sigma_p, R_1, \ldots, R_q)$ where the operations and relations are finitary and need not be total.

If X, Y are non-empty sets, then by $P(X, Y)$ we denote the set of all partial functions $X \rightarrow Y$; the domain of definition of $f \in P(X, Y)$ is written dom (f).

An essential reference on faps is Friedman's article [7]. Let us take as understood the concept of an A-register machine with n registers M_A^n. Programmes for such machines are written in the following language.

Constant is H for *halt*. Variables are $r_0, r_1, r_2, \ldots$ for *algebra registers*. Function symbols and relation symbols are those used in the signature of the relational structure A.

A *finite algorithmic procedure* P is an ordered finite list of instructions $(I_1, \ldots, I_k)$ where instructions are of two kinds.

The *operational instructions* which manipulate elements of A are

$r_\mu := \sigma(r_{\lambda_1}, \ldots, r_{\lambda_n})$ meaning "apply the n-ary operation σ to the contents of registers $r_{\lambda_1}, \ldots, r_{\lambda_n}$ and replace the content of register r_μ by this value".

$r_\mu := r_\lambda$ meaning "replace the content of register r_μ with that of r_λ. If r_λ is empty, then r_μ is made empty".

H meaning "stop".

The *conditional instructions* which determine the order of implementing instructions are

if $R(r_{\lambda_1},\ldots,r_{\lambda_n})$ **then** i **else** j meaning "if the n-ary relation R is true of the contents of $r_{\lambda_1},\ldots,r_{\lambda_n}$, then the next instruction is I_i, otherwise it is I_j".

goto i meaning "the next instruction is I_i".

By convention, a fap P always involves an initial segment of the register variables $r_0, r_1,\ldots,r_m$ where the first few registers $r_1,\ldots,r_n$ are reserved as *input registers* and r_0 as *output register*. Thus, a given fap P with $m+1$ registers together with an appropriate machine M_A^{m+1} defines a partial function $A^n \to A$ for $n \leqq m$ in the obvious way: load the argument $a \in A^n$ into the input registers $r_1,\ldots,r_n$, the remaining registers being empty, and start at its first instruction. The instructions of P are executed in the order in which they are given, except where a conditional instruction directs otherwise. If the machine halts and the output register r_0 is not empty, then the value of the function $P(a)$ is defined to be the element in r_0, else no value of the function on a is defined.

A convention we adopt is that instructions involving operations or relations which are undefined force the computation to "hang" in that no further instructions are processed and the computation is undefined.

$f \in P(A^n, A)$ is *fap-computable* iff there exists a fap P and a machine M such that for each $a \in A^n$, $f(a) \simeq P(a)$.

Append to the syntax for faps the variable s for *stack register*, the new constants $1, 2, \ldots$ for *stack markers* and $\varnothing$ for the empty stack. The new operational instructions are

$s := (i, r_0,\ldots,r_m)$ meaning "place a copy of the contents of the registers $r_0,\ldots,r_m$ as an $(m+1)$-tuple at the *top* of the stack register together with the marker i".

restore $(r_0, r_1,\ldots,r_{j-1}, r_{j+1},\ldots,r_m)$ meaning "remove the last, or *topmost*, entry placed in the stack and replace the contents of the registers $r_0,\ldots,r_{j-1}, r_{j+1},\ldots,r_m$ by the corresponding compontents of the $(m+1)$-tuple".

if $s = \varnothing$ **then** i **else** j is a new conditional instruction and it takes its natural meaning.

In writing fapS's it is convenient to regulate how these new instructions appear in the basic faps through devising *stacking blocks* of instructions. A stacking block is a sequence of consequent instructions of the following form

```
⌈ s := (i; r_0,...,r_m)
| I_1
| ⋮
| I_t
| goto k
| *: r_j := r_0
⌊ restore (r_0, r_1,...,r_{j-1}, r_{j+1},...,r_m)
```

The marker i is unique to the block in any programme in which that block appears. The $I_1, \ldots, I_t$ are ordinary fap operational instructions referred to as *(re-)loading instructions.* The instruction I_k has a special rôle in the operation of the block, it is called the *return instruction* of the block and it must be either an ordinary fap instruction outside all the blocks in the programme or it is the *first* instruction of any block in the programme. The instruction (informally) prefixed by an asterisk is called the *exit instruction.*

The halt instruction also takes on the form of a block

```
┌ I_i      if s = ∅ then i+1 else i+2
│ I_{i+1}  H
│ I_{i+2}  goto (exit instruction of the block whose marker is
└          topmost in the stack).
```

This halting block we abbreviate: **if** $s=\varnothing$ **then H else** *.

The new instructions involving the stack may only occur in a stacking block or a halting block.

Another convention we operate is that from an ordinary fap conditional instruction one may not enter a block except by way of its first instruction.

So a *finite algorithmic procedure with stacking* is defined to be a programme of instructions satisfying the conditions and conventions described. A given fapS P, together with a machine M, defines a partial function over A in the obvious way and $f \in P(A^n, A)$ is said to be *fapS-computable* iff there exists an appropriate fapS P and machine M to compute it.

In working with programmes we shall often corrupt the formal language and instruction forms with informal descriptions where this simplifies our exposition.

2. Inductive definability.

Among a number of versions of inductive definability we choose that in Platek's thesis [17] where Kleene's [8, 9] recursion on higher types over ω was generalised to the hereditarily consistent functionals over an arbitrary set with some primitive structure; the only published account of Platek's work appears in Moldestad's [12].

The inductively definable functions on A are created from the system's operations and relations — presented in the form of definition-by-cases functions — by means of composition and taking fixed-points of certain specially constructed monotone functionals. Given Kleene's revision of recursion, this class IND (A) is a natural candidate for that of the recursive functions on A. We propose to give an entirely syntactic definition of IND (A) but will first consider it in a rather algebraic way.

In working with partial functions on A it is convenient to replace A by A_u being A with the symbol u for undefined adjoined; operations and relations of A take their obvious definitions on A_u: the value of a function on an argument involving u being u. We omit the subscript whenever there is no opportunity for confusion.

Consider simultaneously partial functions of all arguments over A, $P(A) = \bigcup_{n\in\omega} P(A^n, A)$, in which we specify a basic family of functions and on which we shall ultimately define two generating processes. The initial functions are these

i. For each n, the projection functions $U_i^n(a_1,\ldots,a_n)=a_i$, $1\leqq i\leqq n$, from $P(A^n, A)$.
ii. If σ is an n-ary operation of A then σ from $P(A^n, A)$.
iii. If R is an n-ary relation of A then

$$\begin{aligned} DC_R(a_1,\ldots a_n, x, y) &= x \quad \text{if} \quad R(a_1,\ldots,a_n) \\ &= y \quad \text{if} \urcorner R(a_1,\ldots,a_n) \end{aligned}$$

from $P(A^{n+2}, A)$.

iv. u_n the nowhere defined function from $P(A^n, A)$.

The operations on $P(A)$ are *general compositions*

$$C^{m,n}\colon\ P(A^m, A)\times P(A^n, A)^m \to P(A^n, A)$$

defined

$$C^{m,n}(f, g_1,\ldots,g_m)(a) = f(g_1(a),\ldots,g_m(a))$$

and more complicated operations involving fixed-points which we now begin to describe.

Let X, Y be non-empty sets. If $f, g \in P(X, Y)$ then f is a *subfunction* of g, $f\leqq g$, iff dom $(f)\subset$ dom (g) and for each $x\in$ dom (f), $f(x)=g(x)$. A map $\psi\colon P(X, Y) \to P(X, Y)$ is *monotonic* iff for each $f, g\in P(X, Y)$, if $f\leqq g$ then $\psi(f)\leqq\psi(g)$. And ψ is *continuous* iff for each $f\in P(X, Y)$ and any $y_1,\ldots,y_n\in X$ there exist $x_1,\ldots,x_m\in X$ such that if $g(x_i)=f(x_i)$, $1\leqq i\leqq m$, then $\psi(g)(y_i)=\psi(f)(y_i)$ for $1\leqq i \leqq n$. The set of all continuous and monotonic maps $P(X, Y)\to P(X, Y)$ we denote $CM(P(X, Y), P(X, Y))$; it is closed under composition.

2.1. LEAST FIXED-POINT THEOREM. *A continuous monotonic function $\psi\colon P(X, Y)\to P(X, Y)$ has a unique least fixed-point ψ^* and, moreover, $\psi^* = \mathrm{lub}_{n\in\omega}\psi^n(u)$.*

PROOF. Define the countable sequence $f^0=u$, $f^{n+1}=\psi(f^n)$. It is easy to see that for each n, $f^n \leqq f^{n+1}$. By induction on n: it is true for $n=0$ as u is a subfunction of any function. If $f^n \leqq f^{n+1}$, then $\psi(f^n) \leqq \psi(f^{n+1})$ as ψ is monotonic, but this is the relation $f^{n+1} \leqq f^{n+2}$. So set $f=\bigcup_{n\in\omega} f^n$, the least upper bound of the $\psi^n(u)$.

We claim that $\psi(f)=f$ and that if $\psi(g)=g$, then $f \leqq g$. $f \leqq \psi(f)$ follows from monotonicity: for any n, $f^{n-1} \leqq f$ thus $\psi(f^{n-1}) \leqq \psi(f)$ and $f^n \leqq \psi(f)$. So $f \leqq \psi(f)$. $\psi(f) \leqq f$ requires continuity: let $x \in \mathrm{dom}(f)$, by continuity of ψ there exist $x_1,\ldots,x_m \in X$ such that $g(x_i)=f(x_i)$, $1 \leqq i \leqq m$, entails $\psi(g)(x)=\psi(f)(x)$. Choose $g=f\restriction\{x_1,\ldots,x_m\}$. Now $g \leqq f^n$ for some n and $\psi(g) \leqq \psi(f^n)=f^{n+1} \leqq f$. Finally, to show that if $\psi(g)=g$, then for each n, $f^n \leqq g$ we use induction. The statement is obviously true for $n=0$. If $f^n \leqq g$ then $\psi(f^n) \leqq \psi(g)$ which is $f^{n+1} \leqq g$.

Thus for each n there is a *fixed-point operator*

$$FP^n\colon\ CM(P(A^n,A),P(A^n,A)) \to P(A^n,A)$$

defined $FP^n(\psi)=\psi^*$ inductively and constructively in this proof. For the existence and inductive character of a least fixed-point the hypothesis of continuity is immaterial, it is included because constructivity is required; a useful reference for fixed-points is Manna and Shamir's [11]. In these fixed-point operators is the essence of recursion and to complete the definition of IND (A) we have only to explain the construction of appropriate continuous, monotonic functionals from given partial functions.

For the syntactic definition naturally we work with respect to the signature of A. The terms required are defined inductively and solely by the following clauses:

(i) the algebra element indeterminates $X=\{x_1,x_2,\ldots\}$ are terms of type 0;
(ii) $\underline{u}$ is a term of type 0;
(iii) for each n and $1 \leqq i \leqq n$, the n-ary functional symbol $\underline{U_i^n}$ is a term of type $1.n$;
(iv) for each n-ary operation σ the function symbol $\underline{\sigma}$ is a term of type $1.n$;
(v) for each n-ary relation R the function symbol $\underline{DC_R}$ is a term of type $1.n+2$;
(vi) for each n, the n-ary partial function indeterminates $P^n=\{p_1^n,p_2^n,\ldots\}$ are terms of type $1.n$;
(vii) if t is a term of tvpe 0 then $FP[\lambda p_i^n,y_1,\ldots,y_n.t]$ is a term of type $1.n$; here $y_1,\ldots,y_n$ are algebra indeterminates which along with p_i^n are closed in the whole term;
(viii) if T is a term of type $1.n$ and $t_1,\ldots,t_n$ are terms of type 0 then $T(t_1,\ldots,t_n)$ is a term of type 0.

Let T_0 be the set of terms of type 0 and T_1 the set of terms of type 1 so called *algebra terms* and *function terms* respectively.

It is intuitively clear how this syntax is used to define the A-recursive functions: a partial function $f\colon A^n \to A$ will be *inductively definable* iff there is an algebra term $t(y_1,\ldots,y_n)$ with $y_1,\ldots,y_n$ its only free variables such that for all $a_1,\ldots,a_n \in A$, $f(a_1,\ldots,a_n) \simeq t(a_1,\ldots,a_n)$, the "value" of the term t at $a_1,\ldots,a_n$. The valuation functions which formally define the semantics of these terms are constructed in the usual way by induction on the complexity of the terms. A technical point worthy of note is that the semantics of an FP term requires the semantics of its principal subterm to be a continuous monotonical functional so that Theorem 2.1 is applicable.

We also make the convention that the valuation of an algebraic term is undefined if the valuation of any of its subterms is undefined *except* for the basic type 0 terms x_i and $\underline{u}$ given in (i) and (ii). Thus, for $a \in A$ and σ an operation of A not defined on u if t is $\underline{U}_1^2(x_1,x_2)$ and t' is $\underline{U}_1^2(x_1,\underline{\sigma}(x_2))$ then $t(a,u)=a$ while $t'(a,u)$ is undefined.

The function defined by the term $FP[\lambda p, y_1,\ldots,y_n.t]$ is nothing other than the function given by the "recursion" $f(a_1,\ldots,a_n)=t(f,a_1,\ldots,a_n)$ as may be seen in the following simple example. Let $G=(G;\ \cdot,{}^{-1},1,=)$ be a group. Consider the function $f\colon G \to G$ defined $f(a)=1$ if a is of finite order in G and is undefined otherwise. Define inductively

$$\begin{aligned} g(a,b) &= 1 && \text{if } a=1\\ &= g(a\cdot b,b) && \text{if } a\neq 1 \end{aligned}$$

so that $f(a)=g(a,a)$. Then $t(x)=FP[\lambda p,y_1,y_2.\underline{DC}_=(y_1,1,1,p(y_1\cdot y_2,y_2))](x,x)$ is an algebra term such that for every $a \in G$ the valuation function V_a defined by $x \mapsto a$ is such that $f(a)=V_a(t(x))=$"$t(a)$".

The directly inductive functions are obtained from a subset of T_0. Let t be an algebra term with p a function variable free in t. Then p is said to *occur in a conditional place* in t iff there is a subterm $\underline{DC}_R(t_1,\ldots,t_n,t_{n+1},t_{n+2})$ such that p occurs in one (or more) of the t_i, $1\leq i\leq n$. And p is said to *occur in the scope of a function term* in t iff p occurs in one of $t_1,\ldots,t_n$ and $T(t_1,\ldots,t_n)$ is a subterm of t where T is any of (iii), (iv), (vi) and (vii).

An algebra term t is said to be *direct* iff for each subterm t_0 all function variables which are free in t_0 do not occur in a conditional place in t_0 nor in the scope of a function term in t_0. Thus, the term given in the example is direct.

A partial function $f\colon A^n \to A$ is *directly inductively definable* iff it is inductively definable by a direct term.

3. Programmes from inductions.

The relationship between induction terms and (informal) recursions exemplified in the previous section suggest a relationship between fixed-points in induction terms and the loop phenomena characteristic of programmes. It is this latter relationship we analyse and exploit in our proofs.

In proving the theorems we work over the structure A_u where the undefined element u in a computation over A_u corresponds to an empty register in a computation over A. The reader should recall the conventions decided upon with regard to undefined machine instructions and the rôle of $\underline{u}$ in the valuation of induction terms.

3.1. Theorem. IND $(A) \subseteq$ FAPS (A).

Proof. The following more general result is proved. If t is an algebraic term with free function variables $p_1,\ldots,p_e$ then there is a fapS P in the language of the signature of A extended by $p_1,\ldots,p_e$ (i.e. allowing operational instructions $r_j := p_i(\vec{r})$ where $\vec{r}$ is a list of registers of appropriate length) such that t and P define the same partial function for any interpretation of $p_1,\ldots,p_e$ on A_u.

The proof is by induction on the complexity of the term t which has one of the following forms:

(i) x_i
(ii) $\underline{u}$
(iii) $\underline{U}_i^k(t_1,\ldots,t_k)$
(iv) $\underline{\sigma}(t_1,\ldots,t_k)$
(v) $\underline{DC}_R(t_1,\ldots,t_k,t_{k+1},t_{k+2})$
(vi) $p_i(t_1,\ldots,t_k)$
(vii) $FP[\lambda p, y_1,\ldots,y_k.t_0](t_1,\ldots,t_k)$,

where each t_i is an algebraic term.

All cases except (vii) are straight-forward. Thus, (iv) is essentially the standard composition of programmes argument. By the induction hypothesis there are fapS's $P_1,\ldots,P_k$ which compute $t_1,\ldots,t_k$. These programmes are composed to a programme P computing $\sigma(t_1,\ldots,t_k)$ in a natural way: P consists of P_1 followed by P_2 and so on, but saves the result of the computation of each P_i and ends by computing $\sigma(t_1,\ldots,t_k)$ using the operational instruction for σ on the saved values. The technical details are left to the reader. Note that a similar procedure is used for (iii), that is all of $t_1,\ldots,t_k$ are computed, not only t_i. This ensures that the function defined by the term in (iii) is defined iff the function defined by the constructed fapS is defined.

Now consider case (vii). By the composition argument of case (iv) we may assume without loss of generality that t has the simpler form $FP[\lambda p, y_1,\ldots,y_k.t_0](z_1,\ldots,z_k)$. By the induction hypothesis there is a fapS P_0 such that P_0 and t_0 define the same partial function for any given interpretation of $p, p_1, \ldots, p_e$. Let the free algebraic variables in t_0 be $x_1, \ldots, x_n$, $y_1, \ldots, y_k$. We may assume $r_1, \ldots, r_n$, $r_{n+1}, \ldots, r_{n+k}$ to be the input registers for P_0 and that their contents are unaltered by P_0. P is obtained from P_0 by replacing each instruction $r_j := p(\vec{r})$ with a stacking block

```
┌ $s := (i; r_0, \ldots, r_m)$
│ $r_{n+1}, \ldots, r_{n+k} := \vec{r}$
│ **goto** $i \rightarrow$
│ $* \; r_j := r_0$
└ **restore** $(r_0, \ldots, r_{j-1}, r_{j+1}, \ldots, r_m)$ ,
```

where i is a marker unique to the block and $i \rightarrow$ denotes the first instruction of P.

Fix $a \in A_u^n$ and interpretations of $p_1, \ldots, p_e$ and let f be the k-ary partial function defined by $FP[\lambda p, y_1, \ldots, y_k . t_0]$ with these parameters. As in the proof of Theorem 2.1, $f = \bigcup_{i \in \omega} f^i$, where $f^0 = u_k$ and f^{i+1} is the function defined by t_0 with f^i substituted for p, i.e. for each $b \in A_u^k$, $f^{i+1}(b) \cong t_0(f^i)(b)$. The k-ary function g computed by P with these fixed parameters is similarly stratified. For each $b \in A_u^k$ and $c \in A$ let $g^i(b) \cong c$ iff P with input a, b stops with output c and at no stage of the computation were there i vectors in the stack with markers from new blocks, i.e. blocks in P but not in P_0. Of course $g^i(b)$ is undefined if for no $c \in A$, $g^i(b) \cong c$. Thus, $g^0 = u_k$, $g^i \leqq g^{i+1}$ and $g = \bigcup_{i \in \omega} g^i$.

Let $P_0(h)$ denote the k-ary partial function computed by P_0 when p is interpreted as h.

CLAIM: *For each $i \in \omega$ and $b \in A_u^k$, $g^{i+1}(b) \cong P_0(g^i)(b)$.*

Assuming the claim we can complete the proof by showing that $f^i = g^i$ for each i and hence $f = g$. First $f^0 = u_k = g^0$. So assume $f^i = g^i$ and let $b \in A_u^k$. Then

$$f^{i+1}(b) \cong t_0(f^i)(b) \cong t_0(g^i)(b) \cong P_0(g^i)(b) \cong g^{i+1}(b)$$

by definition, the two induction hypotheses and the claim.

To prove the claim fix i and b and consider the concurrent computations $P_0(g^i)(b)$ and $P(b)$ step by step. By definition P_0 and P coincide up to the first p-assignment $r_j := p(\vec{r})$ of P_0 where P sees its first new stacking block B. If no p-assignment is met, then the claim is trivially true. So assume $r_j := p(\vec{r})$ is the first p-assignment met and let τ be the k-tuple of the contents of $\vec{r}$. Assume $g^i(\tau)$ is defined. In P the block B determines a subcomputation $g(\tau) = g^i(\tau)$. The subcomputation has come to an end when control is returned to B with the stack containing only the initial vector with markers from new blocks. And since $g^i(\tau)$ was defined there were at no stage more than i such vectors in the stack. Exiting block B after the subcomputation, the initial vector is removed, the value of $g^i(\tau)$ is in r_j and the contents of the remaining registers are restored to what they were prior to the subcomputation. Thus P_0 and P will again coincide until the next p-assignment of P_0, so that if every required subcomputation $g^i(\tau)$ is defined, then the claim is true. Assume, then, some required subcomputation $g^i(\tau)$ is undefined, so $P_0(g^i)(b)$ is undefined. Then, either $g(\tau)$ is undefined in which case the subcomputation performed by P never converges, i.e. $P(b) \simeq g(b)$ is undefined, or else at least i vectors with new markers were simultaneously in the stack due to the computation $g^i(\tau)$ and hence at least $i+1$ such vectors were simultaneously in the stack during the computation $P(b)$. In either case $g^{i+1}(b)$ is undefined.

3.2. THEOREM. DIND $(A) \subseteq$ FAP (A).

PROOF. Again we prove a more general result. If t is a *direct* algebraic term with free function variables $p_1,\dots,p_e$ then there is a fap P possibly with operational instructions $r_0 := p_i(\vec{r})$, each of which is immediately followed by a halt, such that t and P define the same partial function for any interpretation of $p_1,\dots,p_e$.

The proof is by induction on the complexity of the term t. Consider case (vi), t is $p_i(t_1,\dots,t_k)$. Since t is a direct term, $t_1,\dots,t_k$ contain no free function variables. Thus, the fap programmes $P_1,\dots,P_k$ which compute $t_1,\dots,t_k$ by the induction hypothesis, include no operational instructions involving these free function variables. Let P be obtained from $P_1,\dots,P_k$ using the standard composition of programmes technique, the last instructions being $r_0 := p_i(\vec{r}), \boldsymbol{H}$ where $\vec{r}$ names the saved registers. Note that this is the only case where operational instructions involving free function variables are introduced. Only in case (vii) are they removed.

Consider case (vii). We may again assume without loss of generality that t is $FP[\lambda p, y_1,\dots, y_k . t_0](z_1,\dots, z_k)$. By the induction hypothesis there is a fap P_0 where each p-assignment is followed by a halt (this is what makes the stack superfluous!) such that P_0 and t_0 define the same partial function for any given interpretation of $p, p_1,\dots,p_e$. Obtain P from P_0 by replacing each p-assignment $r_0 := p(\vec{r}), \boldsymbol{H}$ by the block $r_{n+1},\dots,r_{n+k} := \vec{r}$, **goto** 1 where 1 is the first instruction of P. Here, as in the previous proof, $r_1,\dots,r_n$ contain algebra parameters. Stratify the k-ary function g computed by P by defining for each $b \in A_u^k$ and $c \in A$, $g^i(b) \simeq c$ iff P with input b halts with output c and the added blocks of assignments have been processed less than i times. Clearly, $g^0 = u_k$, $g^i \leqq g^{i+1}$ and $g = \bigcup_{i \in \omega} g^i$. And let f be the function defined by $FP[\lambda p, y_1,\dots,y_k . t_0]$ stratified in the usual way. We prove by induction on i that $f^i = g^i$ and hence $f = g$. Obviously, $f^0 = u_k = g^0$. Assume $f^i = g^i$ and let

$b \in A^k$. By definition and the two induction hypotheses $f^{i+1}(b) \simeq t_0(f^i)(b) \simeq P_0(f^i)(b) \simeq P_0(g^i)(b)$. Consider the concurrent computations $P_0(g^i)(b)$ and $P(b)$ step by step. If no p-assignment is met in P_0 then the computations will coincide and $P(g^i)(b) \simeq P(b) \simeq g^1(b) \simeq g^{i+1}(b)$. Assume on the other hand that $r_0 := p(\vec{r})$, $\boldsymbol{H}$ is met in P_0. Letting τ be the k-tuple which is the contents of $\vec{r}$, it is immediate from the definition of g^i that $g^{i+1}(b) \simeq g^i(\tau)$. But $P_0(g^i)(b) \simeq g^i(\tau)$. In either case $f^{i+1}(b) \simeq g^{i+1}(b)$.

4. Inductions from programmes.

4.1. THEOREM. FAP $(A) \subseteq$ DIND (A).

PROOF. First, a technical remark: for the sole purpose of simplifying the presentation of the argument, we here treat instructions **goto** i as instances of the conditional instructions **if** R **then** i **else** i.

Given a fap P we are to construct a direct term t such that P and t define the same partial function. This is achieved by constructing a tree describing all possible paths through P, each node of the tree representing a conditional or halt statement in P. To each node is assigned a direct term corresponding to a subcomputation of P represented by the node. The sought term t is that assigned to the top node and represents the computation of P.

Assume P uses registers $r_0, \ldots, r_m$ and let $J_1, \ldots, J_e$ be the conditional statements in P in order of appearance. To each node we assign a label and a sequence of operational terms or polynomials $t_0, \ldots, t_m$. The latter are obtained from variables $x_0, \ldots, x_m$ using the operations of A and they describe operations performed *between* conditionals or *between* a conditional and a halt. If a halt appears before J_1 the tree will consist of precisely one node labelled H. And the assigned $t_0, \ldots, t_m$ are polynomials giving the contents of $r_0, \ldots, r_m$ at the halt if the registers contained $x_0, \ldots, x_m$ at the start of the programme. If J_1 appears first, the top node will be labelled 1. The assigned polynomials will be the same. J_1 is a conditional of the form **if** $R(\vec{r})$ **then** i **else** j. It will give rise to two nodes below node 1. For the lefthand node, move downwards in P from instruction i until a halt or a conditional statement appears. If it is a halt, the new node is labelled H while if it is the conditional J_k the new node is labelled k. In either case the polynomials $t_0, \ldots, t_m$ assigned to the node will give the contents of $r_0, \ldots, r_m$ at the conditional or halt statement if their contents were $x_0, \ldots, x_m$ when starting to process instruction i. A similar assignment is made to the right-hand node starting from instruction j. No nodes are constructed below an H node. And two nodes are constructed below a k node with assignments as above *unless* there is a preceding node also labelled k in which case no new lower nodes are constructed. A continuation of the tree in this last case would consists of an infinite number of copies of the segment between the two k nodes. Note that our tree is finite.

We now assign a direct term to each node starting with the bottom nodes. For an H node with assigned polynomials $t_0,\ldots,t_m$ assign the direct term $\underline{U}_1^{m+1}(t_0,\ldots,t_m)$ (giving the content of the output register). And to a bottom node labelled k assign $p_k(t_0,\ldots,t_m)$ where p_k is an $(m+1)$-ary function variable and $t_0,\ldots,t_m$ the assigned polynomials. Now consider a node labelled k with assigned polynomials $t_0,\ldots,t_m$ which is not a bottom node. Let the corresponding conditional be **if** $R(\vec{r})$ **then** i **else** j. Let s_3 and s_4 be the direct terms assigned to the nodes immediately below. *Case* 1: No node below is labelled k. Obtain s_1 and s_2 from s_3 and s_4 respectively by replacing each x_i by t_i and let t be the list of polynomials corresponding to the list of registers $\vec{r}$. Assign the direct term $\underline{DC}_R(t,s_1,s_2)$. *Case* 2: There is a node below labelled k (this corresponds to a loop). Introduce variables $y_0^k,\ldots,y_m^k$ and obtain s_3' and s_4' from s_3 and s_4 respectively by replacing each x_i by y_i^k. And let y be the list of variables from $y_0^k,\ldots,y_m^k$ corresponding to $\vec{r}$. Assign the direct term

$$FP[\lambda p_k, y_0^k,\ldots,y_m^k.\underline{DC}_R(y,s_3',s_4')](t_0,\ldots,t_m)\,.$$

For each node N in our tree we define a programme P_N such that the direct term assigned to N and P_N define the same partial function under any given interpretation of the free function variables. To obtain P_N alter P as follows: Whenever a node above N and N or a node below N are labelled k then J_k is replaced by $r_0\!:=p_k(r_0,\ldots,r_m),\boldsymbol{H}$. Furthermore, processing of P_N starts with the operational instructions giving rise to the polynomials assigned to N. If N is the top node, then P_N and P are, of course, identical.

The proof that P_N and the direct term assigned to N define the same function is by induction on the nodes starting at the bottom nodes. All cases are immediate except the one involving an FP term. So suppose N is a node labelled k to which the direct term

$$FP[\lambda p_k, y_0^k,\ldots,y_m^k.\underline{DC}_R(y,s_3',s_4')](t_0,\ldots,t_m)\,,$$

call it t, has been assigned and let N_1 and N_2 be the nodes immediately below N. Let P_1 and P_2 be the programmes for N_1 and N_2 respectively which, by the induction hypothesis, define the same functions as s_3 and s_4. Let P_N' be obtained from P_N by having processing start at J_k and let g be the function computed by P_N'. Stratify g as follows: For $a\in A_u^{m+1}$ and $b\in A$, let $g^i(a)\cong b$ iff P_N' with input a stops with output b and the computation processed J_k at most i times. And let f be the function defined by

$$FP[\lambda p_k, y_0^k,\ldots,y_m^k.\underline{DC}_R(y,s_3',s_4')]$$

with its usual stratification. If $f=g$ then clearly t and P_N define the same function. We show by induction on i that $f^i = g^i$ and hence $f = g$. As usual, $f^0 = u_{m+1} = g^0$. Assume $f^i = g^i$ and let $a \in A_u^{m+1}$. Assume further that $R(a')$ where a' is the list obtained from a corresponding to $\vec{r}$, the argument being analogous if $\neg R(a')$. Then $f^{i+1}(a) \simeq s'_3(f^i)(a) \simeq P_1(f^i)(a) \simeq P_1(g^i)(a)$ by definition and the two induction hypotheses. Consider the computation $P'_N(a)$ step by step. Since $R(a')$ holds, this computation will after the first step coincide with that of $P_1(g^i)(a)$ until P again processes J_k. If this never happens, then $P_1(g^i)(a) \simeq g^1(a) \simeq g(a)$. Suppose it happens and let τ be the contents of the registers at that time. Clearly, $P_1(g^i)(a) \simeq g^i(\tau)$ and $g^{i+1}(a) \simeq g^i(\tau)$. In either case $P_1(g^i)(a) \simeq g^{i+1}(a)$ so $f^{i+1} = g^{i+1}$.

Finally, if a fap P computes $f \in P(A^n, A)$ and t is the direct term obtained from P as described above then f is defined by the direct term obtained from t by replacing each of the free variables $x_0, x_{n+1}, \ldots, x_m$ by $\underline{u}$.

4.2. THEOREM. FAPS $(A) \subseteq$ IND (A).

PROOF. For each fapS P with e stacking blocks and using registers $r_0, \ldots, r_m$ we are to construct a term t such that P and t define the same function in $P(A_u^{m+1}, A)$. The theorem then follows as in the proof of Theorem 4.1.

For $e = 0$ Theorem 4.1 provides us with a term. So suppose $e > 0$. We construct fap programmes $P_0, P_1, \ldots, P_e$ in the language of the signature of A extended by $p_1, \ldots, p_e$ where each p_i is an $(m+1)$-ary function symbol. P_0 is obtained from P by replacing the kth stacking block with $r_{j_k} := p_k(\tau_k)$ for $k = 1, \ldots, e$. τ_k is the list of operational terms given by the (re-loading) instructions in the kth block. P_k is the same as P_0 *except* that $k \rightarrow$ is the first instruction processed where $k \rightarrow$ is the return instruction of the kth block.

By Theorem 4.1, there are terms $t_0, t_1, \ldots, t_e$ defining the same functions in $P(A_u^{m+1}, A)$ as $P_0, P_1, \ldots, P_e$ for any given interpretation of $p_1, \ldots, p_e$. We replace the free function variables in t_0 by terms to obtain the sought term t as follows. Let t_i^0 be t_i for $i = 0, 1, \ldots, e$. Suppose the terms $t_0^i, t_{i+1}^i, \ldots, t_e^i$ are defined and $p_1, \ldots, p_i$ are not free in these. Let t_{i+1}^{i+1} be $FP[\lambda p_{i+1}, x_0, \ldots, x_m . t_{i+1}^i]$ and substitute t_{i+1}^{i+1} for p_{i+1} in $t_0^i, t_{i+2}^i, \ldots, t_e^i$ to obtain $t_0^{i+1}, t_{i+2}^{i+1}, \ldots, t_e^{i+1}$. Finally, let t be t_0^e.

We define the following programmes. Let P_k^0 be P_k for $k = 0, \ldots, e$. P_k^{i+1} is obtained from P_k^i by replacing $r_{j_{i+1}} := p_{i+1}(\tau_{i+1})$ with the $(i+1)$-th stacking block. Of course P_0^e is just P. Thus the following claim proves the theorem.

CLAIM. *P_j^i and t_j^i define the same partial functions, $j = 0, i, \ldots, e$, for any given interpretation of $p_{i+1}, \ldots, p_e$.*

The proof is by induction on i. By choice of terms and programmes the claim is true for $i=0$. Assuming it is true for i we first show P^{i+1}_{i+1} and t^{i+1}_{i+1} define the same function. The function g computed by P^{i+1}_{i+1} is stratified as follows: For $a \in A^{m+1}_u$ and $b \in A$, let $g^k(a) \cong b$ iff P^{i+1}_{i+1} with input a halts with output b and at no stage of the computation were there k vectors with marker $i+1$ in the stack. The function f defined by t^{i+1}_{i+1} is stratified as usual. By induction on k we show $f^k = g^k$ and hence $f = g$. Obviously, $f^0 = u_{m+1} = g^0$. Assume $f^k = g^k$ and let $a \in A^{m+1}_u$. Then

$$f^{k+1}(a) \cong t^i_{i+1}(f^k)(a) \cong p^i_{i+1}(f^k)(a) \cong p^i_{i+1}(g^k)(a)$$

by definition and the two induction hypotheses. And, by the argument for the claim in the proof of Theorem 3.1, $P^i_{i+1}(g^k)(a) \cong g^{k+1}(a)$.

It remains to prove P^{i+1}_j and t^{i+1}_j define the same functions for $j=0$, $i+2,\ldots,e$. Let h be the function defined by t^{i+1}_{i+1}. Then, for each $a \in A^{m+1}_u$, $t^{i+1}_j(a) \cong t^i_j(h)(a) \cong P^i_j(h)(a)$. Fix a and consider the concurrent computations $P^i_j(h)(a)$ and $P^{i+1}_j(a)$ step by step. They are identical until P^i_j meets $r := p_{i+1}(\tau)$ and P^{i+1}_j meets the $(i+1)$-th stacking block. The return instruction in this block is $(i+1) \rightarrow$ so that the subcomputation is computed by P^{i+1}_{i+1}, that is the subcomputation is $h(\tau)$. It follows that $P^i_j(h)(a) \cong P^{i+1}_j(a)$.

Note that the passages from term to programme, or programme to term, are all constructive in the four theorems.

REFERENCES

1. R. C. Constable and D. Gries, *On classes of program schemata*, SIAM. Comput. 1 (1972), 66–118.
2. S. Feferman, *Inductive schemata and recursively continuous functionals*, pp. 373–392 of R. O. Gandy and J. M. E. Hyland (eds.), *Logic colloquium* '76, North-Holland, Amsterdam, 1977.
3. J. E. Fenstad, *On axiomatizing recursion theory*, pp. 385–404 of J. E. Fenstad and P. G. Hinman (eds.), *Generalized recursion theory*, North-Holland, Amsterdam, 1974.
4. J. E. Fenstad, *Computation theories: an axiomatic approach to recursion on general structures*, pp. 143–168 of G. Müller, A. Oberschelfp and K. Potthoff (eds.), *Logic conference, Kiel* 1974, Springer-Verlag, Berlin - Heidelberg - New York, 1975.
5. 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: *Generalized recursion theory II*. North-Holland, Amsterdam, 1978.
6. J. E. Fenstad, *General recursion theory: an axiomatic approach*, Springer-Verlag, Berlin - Heidelberg - New York, 1980.
7. 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.
8. S. C. Kleene, *Recursive functionals and quantifiers of finite type I*, Trans. Amer. Math. Soc. 91 (1959), 1–52.

9. S. C. Kleene, *Recursive functionals and quantifiers of finite type II*, Trans. Amer. Math. Soc. 108 (1963), 106–142.
10. A. I. Mal'cev, *Algebraic systems*, Springer-Verlag, Berlin - Heidelberg - New York, 1973.
11. Z. Manna and A. Shamir, *The convergence of functions to fixed-points of recursive definitions*, Theor. Comput. Sci. 6 (1978), 109–141.
12. J. Moldestad, *Computations in higher types*, Springer-Verlag, Berlin - Heidelberg - New York, 1977.
13. J. Moldestad, V. Stoltenberg-Hansen and J. V. Tucker, *Finite algorithmic procedures and computation theories*, Math. Scand. 46 (1980), 77–94.
14. Y. N. Moschovakis, *Abstract first-order computability I*, Trans. Amer. Math. Soc. 138 (1969), 427–464.
15. Y. N. Moschovakis, *Abstract first-order computability II*, Trans. Amer. Math. Soc. 138 (1969), 465–504.
16. Y. N. Moschovakis, *Axioms for computations theories – first draft*, pp. 119–255 of R. O. Gandy and C. M. E. Yates (eds.), *Logic colloquium* '69, North-Holland, Amsterdam, 1971.
17. R. A. Platek, *Foundations of recursion theory*, Ph. D. Thesis, Stanford University, Stanford, 1966.
18. J. C. Shepherdson, *Computations 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.
19. J. V. Tucker, *Computing in algebraic systems*, Matematisk institutt, Universitetet i Oslo, Preprint Series, Oslo, 1978.

UNIVERSITETET I OSLO, MATEMATISK INSTITUTT
POSTBOKS 1053, BLINDERN, OSLO 3, NORWAY

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

MATHEMATISCH CENTRUM,
TWEEDE BOERHAAVESTRAAT 49, 1091 AL, AMSTERDAM, THE NETHERLANDS