JOURNAL OF COMPUTER AND SYSTEM SCIENCES 27, 268–290 (1983)

Correctness of Recursive Parallel Nondeterministic Flow Programs

J. A. GOGUEN AND J. MESEGUER

SRI International, 333 Ravenswood Ave, Menlo Park, California 94025

Received November 15, 1982; revised February 7, 1983

THIS PAPER IS DEDICATED TO THE MEMORY OF CAL ELGOT

This paper presents a method for proving the partial correctness of programs with the following features: strongly typed expressions with call-by-value semantics for variables; iteration; recursive procedures with call-by-name semantics; nondeterminism; parallel assignment; and good old fashioned go-to's. An operational semantics is given to a program by viewing it as a program scheme together with an appropriate interpretation in a given model. Program schemes are viewed as diagrams in an algebraic theory, and the given models are relational algebras of this theory. A simple programming language, REPNOD, that embodies exactly the features that are discussed theoretically is defined, and several simple REPNOD programs, as well as a sample correctness proof, are given. This approach seems to provide a particularly simple framework for many problems in concurrent programming.

1. INTRODUCTION

This paper presents a surprisingly simple algebraic semantics for recursive nondeterministic strongly typed programs with parallel (sometimes called "simultaneous") assignments and go-to's. The main result is a method for proving the correctness of such programs that generalizes the so-called Floyd–Naur method in the reformulation of [5, 13]. In this approach, a nondeterministic (nonrecursive) program is represented by a graph whose nodes are labelled with sets, and whose edges are labelled with relations, such that if $e: n \to n'$ is an edge from n to n', and if the labels of e, n, n' are f, S, S', respectively, then f is a function $\mathscr{P}(S) \to \mathscr{P}(S')$, where $\mathscr{P}(A)$ is the set of all subsets of A, that is, the powerset of A; thus f is a relation between S and S'. The nodes correspond to "states of control" of the program, while an element of the label of a node is a memory state of the computation. The edges correspond to transitions of control state, and are labelled by relations which describe the resulting changes of state. Our approach therefore resembles the way that Elgot [10] collected monadic flow diagrams into algebraic theories.

* Research supported in part Office of Naval Research contracts N00014-80-0296 and N00014-82-C-0333, and National Science Foundation Grant MCS8201380.

0022-0000/83 $3.00

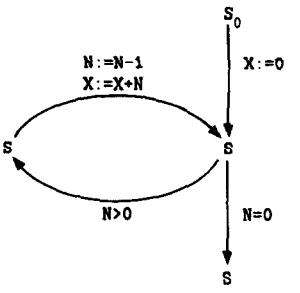

FIG. 1. A summation program.

Let us illustrate this with a simple example (see Fig. 1), a program to compute $X = \sum_{i=1}^{N} i$ given $N \geqslant 0$. In this diagram, nodes are labelled with S and S_0, where $S = [\{X, N\} \to \omega]$ is the set of possible states of knowing integer values for X, N, where $[A \to B]$ denotes the set of all (total) functions from A to B, ω denotes the set of all nonnegative integers, and $S_0 = [\{N\} \to \omega]$. A typical element of S might be the pair $\{\langle X, 0\rangle, \langle N, 3\rangle\}$, using the usual representation of a function as a set of ordered pairs.[1] We take account of nondeterminism by considering elements of $\mathscr{P}(S)$, that is, sets of possible states of memory, rather than just elements of S. (Since the program above is not really nondeterminstic, this feature is not necessary for its treatment.) Any relation $f \subseteq S \times S'$ defines a function $\mathscr{P}(f): \mathscr{P}(S) \to \mathscr{P}(S')$ as follows: for $R \subseteq S$, $\mathscr{P}(f)(R) = \bigcup \{f(s) \mid s \in R\}$, where $f(s)$ is the set of elements in S' which are related under f to $s \in S$. The label "$N > 0$" denotes $\mathscr{P}(f)$, where $f: S \to S$ is the partial identity function defined for those $s \in S$ with $s(N) > 0$, and then with value $f(s) = s$. It has the effect of a conditional branch on N positive. Similarly, "$N = 0$" denotes $\mathscr{P}(f)$ with $f: S \to S$ the partial identity function defined iff $s(N) = 0$, having the effect of a conditional branch on zero. Further, the label "$X := 0$" denotes $\mathscr{P}(f)$ with $f: S_0 \to S$ the function sending s to s' with $s'(X) = 0$ and $s'(N) = s(N)$. Finally "$N := N - 1$, $X := X + N$" denotes a parallel assignment $\mathscr{P}(f)$, where $f: S \to S$ sends s to s' defined by $s'(N) = s(N) - 1$ and $s'(X) = s(X) + s(N)$; the values of X and N are changed in parallel, rather than sequentially. Assuming that "$n - 1$" is a partial function that is defined iff $n > 0$, this function f is defined iff $s(N) > 0$; thus the earlier edge "$N > 0$" is actually unnecessary. (We use this observation later.)

An important ingredient of our approach is to pass from such programs to corresponding program schemes, which do not indicate what the states are, but only what variables are involved, and do not indicate what the tests and operations are, but only their names and how many arguments they want. These names stand for functions that are to be given by an interpretation. In the scheme, only a name remains, such as "pos(N)" or "dec(N)". A simplified scheme for the above program is given in Fig. 2. We have simplified the scheme by labelling nodes with sets of

[1] That is, a subset f of $A \times B$ such that: (1) $\langle a, b\rangle$, $\langle a, b'\rangle \in f$ implies $b = b'$; and (2) for each $a \in A$, there is some $b \in B$ such that $\langle a, b\rangle \in f$. A *partial* function drops requirements (2).

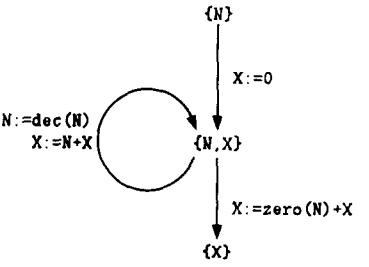

FIGURE 2

variables. rather than sets of states. (Also notice that we have dropped an irrelevant variable N at the exit node.) If e is an edge from node n to node n' and if n and n' are labelled S and S', respectively, then e is labelled with an S'-tuple of terms in S variables, each component written in the form "$X'_i := t_i(X_1,..., X_n)$," where $S' = \{X'_1,..., X'_m\}$ and $S = \{X_1,..., X_m\}$ are the sets of variables, and where each t_i is a *term* in primitive operations.

Looking carefully at the labels of nodes and edges, and the assumptions made about them leads to a generalization of the notion of algebraic theory, in the sense originally formulated by Lawvere [20], with modifications suggested by Goguen *et al.* [17] and John Reynolds. Moreover, an interpretation is exactly an algebra for a theory, or more precisely, a relational algebra as in [9].

The major remaining feature is recursion. Our basic approach is to treat each call of a procedure P as a command to substitute the definition of P, which is given by a flow diagram. Thus, a program with n procedures consists of n flow diagrams, each labelled with the name of the procedure it defines, and each able to use all other procedures within itself; the first one is the "main" one, at which execution starts. Fig. 3 shows a recursive version of the simple program that we have been discussing, and also a recursive program for factorial.

It is worth emphasizing that both iteration (that is, loops within a connected component graph) and recursion can occur in the same program in our formalism. Figure 4 is an iterative program that uses the recursive FACT program given above.

Now some notational details: First, we will no longer bother to put set brackets in node labels. Second, the graphs are all *protected*, meaning that the *entrance* node has no edges to it, and the *exit* node has no edges from it. Third, unlabelled edges represent projection functions or, in particular, identity functions (e.g., $X, N \to X, N$); and unmentioned identifiers (on an edge) are assumed to be unchanged (e.g., $X := X$). Fourth, we permit "$p(X_1,..., X_n)$" to stand for "$X_1 := p_1(X_1,..., X_n)$, $X_2 := p_2(X_1,..., X_n)$;..., $X_n := p_n(X_1,..., X_n)$", where $p = (p_1,..., p_n)$. This will look less peculiar when p is a partial identity function (e.g., "$X_1 = X_2$", or "zero(N)"). Finally, we write the basic operator names in lower case, and the procedures names in upper case; e.g., "zero" and "FACT."

An interesting, though possibly confusing, point is that both call-by-value and call-by-name semantics are used: variables, X, N, F, etc., are called by their value at the

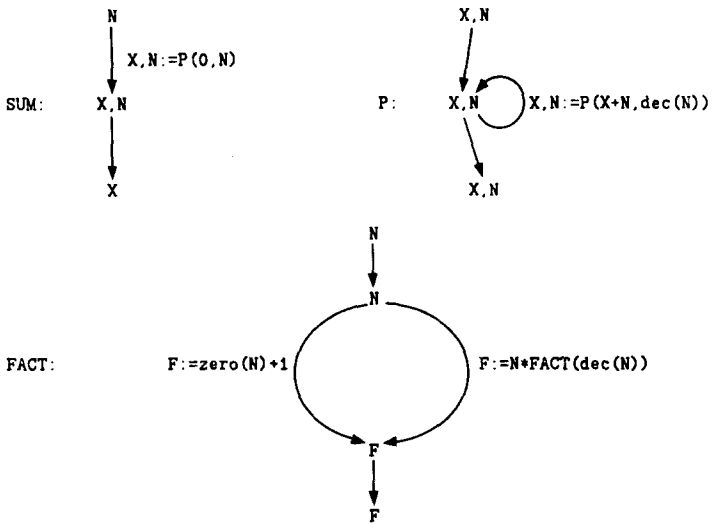

FIGURE 3

previous node; while procedures, SUM, P, FACT, etc., are called by name, using a copy rule. This will become clearer when we give a detailed operational semantics in Section 5.

We conclude this foreshadowing of the paper to follow by giving the above programs in "REPNOD" (for "Recursive Parallel Nondeterministic"), a simple programming language for nonnegative integers which embodies the features discussed in this paper; see Apendix B for further details of REPNOD.

proc SUM(N)
  source (N) **assign** (X, N) := P(0, N) **target** (X, N)
  exit (X) **corp**

proc P(X, N)
  source a(X, N) **assign** (X, N) := P(X + N, dec(N)) **target** b
  source a(X, N) **assign** (X, N) := (X, zero (N))
  exit b(X, N) **corp**

proc FACT(N)
  source a(N) **assign** F := inc(zero(N)) **target** b
  source a(N) **assign** F := N*FACT(dec(N))
  exit b(F) **corp**

In outline, a REPNOD program is a sequence of procedures, where a procedure has both argument and return variable lists, and a body consisting of a sequence of statements, each statement of the form **source** *l*1 **assign** *a*1,..., *an* **target** *l*2, where *l*1

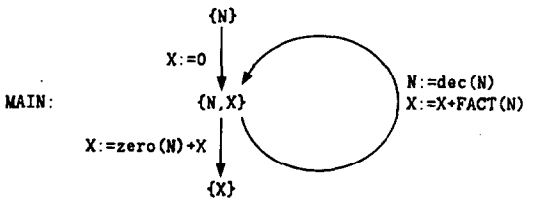

FIGURE 4

and *l2* are labels (corresponding to node names) and *a*1,..., *an* are assignments (to be executed in parallel). REPNOD keywords are in boldface.

Our main result is a criterion for partial correctness of this kind of program: if we "guess" (or "know," or have specifications for) the behavior R_j of each procedure P_j, and if when we substitute R_j for P_j in each flow diagram D_i, we in fact obtain the behavior R_i for D_i, then our guesses are partially correct, i.e., $R \subseteq B$, where B is the actual behavior and R is the tupling of the R_j's. This kind of result was conjectured in [14].

Something that we found very pleasing about writing this paper, was that the algebraic and programming concepts corresponded so precisely, with simplicity and power on each side; it was almost as if they were designed for each other, even though they were developed quite separately. Moreover, it seemed to the authors that the modifications of each side from the historical development required to bring about this agreement actually resulted in improvements! We hope the reader enjoys the result as much as we enjoyed the process.

There is a good deal of work related to the present research. Of course, most closely related of all is a conference proceedings preliminary version of this paper [15]. In a sense, Cal Elgot [10] started the whole thing with his idea of viewing program schemes as morphisms in an algebraic theory. Moreover, [11] is an independent discovery of how to use algebraic theories to represent the mathematical properties of the assignment statement. Reference [11] differs from the present paper in its concentration on an elegant treatment of just the assignment statement; Elgot [11] demonstrates that the difficulties with assignment cited by Backus [2] in calling it the "von Neumann bottleneck of programming languages" need not include giving a proper mathematical theory for the semantics of assignement.

Another related paper is [8], which constructs an algebraic theory whose morphisms are recursive monadic (flow diagram) program schemes. This use of algebraic theories is quite different from that of the present paper (our morphisms are parallel assignments rather than programs); but the subject of their research is similar, and it would be interesting to see a similar construction to theirs carried out for our programs.

The note [6] is perhaps the closest in content to the present paper, in that it treats assignments, go-to's, procedures, and of course expressions, in an integrated matter. It differs as follows: In a certain sense, Burstall's approach is dual to ours, in that it considers continuations from each point in a program instead of assertions; this idea

is one of Burstall's most interesting contributions. Reference [6] is also elegant in its mathematical simplicity; although inspired by [8], it entirely avoids the use of algebraic theories, using instead matrices of relations and fixed point methods. Moreover, program schemes are not treated, only programs; and there is no formal syntax. Whereas we suggest a two-step method, in which assertions are first given for procedures only, thus reducing the problem to a standard correctness problem for ordinary flow diagram programs, [6] shows how to accomplish the whole thing in one step. The note is sketchy about the treatment of procedures, and does not actually discuss many sortednes or parallel assignment. Both [6] and [8] apply to nondeterministic programs.

Gallier [12] also contains results similar to ours, but does not treat call-by-name for procedures (the actual of a call can only be variables, so only the special case of call-by-reference is actually permitted); only one type of data is allowed; algebraic theories are not used; assertions are expressed in a formal logic; and correctness is formulated in a second order logic in the style of [22–24]. Gallier [12] makes effective use of unfoldments, defined by colimits, rather than by an adjunction as in [13].

There appear to be serious problems with expressing assertions and correctness in any particular formal logical language. First of all, the notation will be awkward, difficult to read and inflexible. For example, in order to treat a new data type, such as matrices, one must introduce new functions, predicates, and axioms into the language, and in general, second order axioms will be required. It is much simpler to use simple set-theoretic constructions for new types, and to make free use of the flexibility of set theory. Essentially this approach is taken by [25] with the notion of a "state vector."

Thus, we do not agree with the Gallier [12] assessment of Goguen and Meseguer [15] that "a detailed formulation of the inductive assertion method for their model is not provided and the issue of completeness is not considered," since we intentionally avoided the straightjacket of a particular assertion language; in such an approach, it is pleasantly unnecessary to consider completeness. For the sets and relations that are needed obviously exist, and are describable in simple set theory. (See [14] for more polemic in the same vein.) Both [6] and [8] also wisely avoid logical formulas in favor of a set-theoretic framework.

The algebraic approach to data types provides another attractive alternative to the predicate logic formulations, and is compatible with the present paper [7, 17, 19, 26]. A set-theoretic approach is also taken by Blikle [4] using tools such as nets, quasinets, and Mazurkiewicz algorithms. However, later work of Blikle also uses an algebraic approach to data types.

2. Notation and Preliminaries

The powerset of a set X is denoted $\mathscr{P}(X)$ and the Cartesian product of a family $(A_i)_{i\in I}$ of sets is denoted $\prod_{i\in I} A_i$. The set of functions from B to A is denoted A^B or $[B \to A]$, and the trivial bijection $A^B \approx \prod_{b\in B} A$ is often used implicitly. $\#A$ denotes

the cardinal of A, and ω denotes the set of natural numbers. Given a function $f: A \to B$ and a set C, let f^C (resp. C^f) be "left (resp. right) composition by f", $f^C: A^C \to B^C$ by $g \mapsto f \circ g$ (resp. $C^f: C^B \to C^A$ by $h \mapsto h \circ f$). Functional notation, as $f: A \to B$, is also used for partial functions and for relations. However, a relation $f: A \to B$ is sometimes viewed as a function $f: A \to \mathscr{P}(B)$; note that we tend to denote relations by capitals, R, F, etc. The *tupling* $(f_1,...,f_n)$ of functions (or relations) $f_1: A \to B_1,...,f_n: A \to B_n$ is the function [or relation] $(f_1,...,f_n)$: $A \to B_1 \times \cdots \times B_n$ defined by: $a \mapsto (f_1 a,...,f_n a)$ (or $(f_1,...,f_n)(a; b_1,..., b_n)$ defined iff each $f_1(a, b_1),...,f_n(a, b_n)$ is defined).

Familiarity with the concepts of category, subcategory, functor, natural transformation, and isomorphism of categories is assumed (e.g., [1, 17, 18, 21]). We indicate categories with boldface capital letters, e.g. **C**, and we denote the object class of **C** by $|\mathbf{C}|$, while $\mathbf{C}(A, B)$ is its set of morphisms (arrows, or maps) from A to B. Given morphisms $f: A \to B$, $g: B \to C$, their composition is denoted $g \circ f$ or $gf: A \to C$. The identity map for an object A is denoted $1_A: A \to A$. **SET**, **PFN**, and **REL** denote the categories with objects sets, and morphisms functions, partial functions and relations respectively. Given $B_1,..., B_n \in |\mathbf{C}|$, and object $B \in \mathbf{C}$ together with morphisms $\pi_i: B \to B_i$ for $1 \leqslant i \leqslant n$ (called *projections*) is a *direct product* of $B_1,..., B_n$ iff given an object A and morphisms $f_i: A \to B$, $1 \leqslant i \leqslant n$, there exists a unique morphism $(f_1,...,f_n): A \to B$, such that $\pi_i \circ (f_1,...,f_n) = f_i$ for $1 \leqslant i \leqslant n$; we denote the product object by $B_1 \times \cdots \times B_n$. Define the direct product $\prod_{i \in I} A_i$ of an arbitrary family of objects $(A_i)_{i \in I}$ similarly. Note that Cartesian product is a direct product in **SET** but *not* in **PFN** or **REL**.

A *graph* G is a set G of edges together with a set $|G|$ of *nodes*, and two functions $\partial_0, \partial_1: G \to |G|$ called *source* and *target*. A *graph morphism* from $G = (G, |G|, \partial_0, \partial_1)$ to $H = (H, |H|, \partial_0, \partial_1)$ is a pair of functions $F: G \to H$, $|F|: |G| \to |H|$ which "preserve source and target," i.e., $|F| \partial_i = \partial_i F$ for $i = 0, 1$. Graphs and graph morphisms form a category **GPH**. A graph is *finite* iff both G and $|G|$ are finite. A *path* p in a graph G is a string $e_0 e_1 \cdots e_{n-1}$ of edges in G such that $\partial_1 e_{i-1} = \partial_0 e_i$ for $0 < i < n$; we say that n is the *length* of p. For $n > 0$, we say $p = e_0 \cdots e_{n-1}$ is a path *from source* $\partial_0 p = \partial_0 e_0$ *to target* $\partial_1 p = \partial_1 e_{n-1}$. For any $v \in |G|$, the empty string λ is the length zero path from v to v. Given a graph G, define the *path category* of G, $\mathbf{Pa}_G$, by: $|\mathbf{Pa}_G| = |G|$; $\mathbf{Pa}_G(u, v) = \{(u, p, v) \mid p$ is a path from u to $v\}$ for any $u, v \in |G|$; composition is path (i.e., string) concatenation, $(v, p', w) \circ (u, p, v) = (u, pp', w)$; and $1_u = (u, \lambda, u)$ for $u \in |G|$. For example, if G is the graph of Fig. 5, then we have, for instance, $\mathbf{Pa}_G(u, u) = \{(u, \lambda, u)\}$; $\mathbf{Pa}_G(v, w) = \{(v, (bcd)^n fg^m, w) \mid n, m \in \omega\} \cup \{(v, (bcd)^n bceg^m, w) \mid n, m \in \omega\}$; and $\mathbf{Pa}_G(u, z) = \{(u, a(bcd)^n fg^m h, z) \mid n, m \in w\} \cup \{(u, a(bcd)^n bceg^m h, z) \mid n, m \in w\}$, where if p is a path from a node y to itself, then p^n is λ if $n = 0$, and is $p^{n-1}p$ otherwise.

Any small category **C** (i.e., $|\mathbf{C}|$ a set) defines a graph $U\mathbf{C}$ with nodes the objects and edges the morphisms of **C**. Similarly, any functor $F: \mathbf{C} \to \mathbf{D}$ between two small categories induces a graph morphism $UF: U\mathbf{C} \to U\mathbf{D}$; hence we have a functor U: **CAT** $\to$ **GPH**. There is also a graph morphism $\eta_G: G \to U\mathbf{Pa}_G$ which is the identity on nodes, and maps an edge e in G to the path $(\partial_0 e, e, \partial_1 e)$. $\mathbf{Pa}_G$ has the following

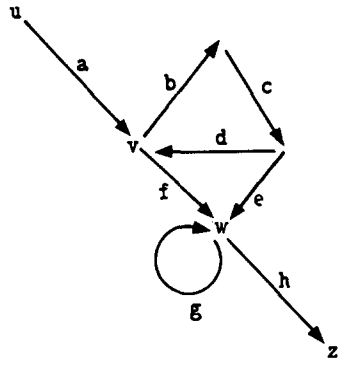

FIG. 5. A graph G.

"universal property": given a small category $\mathbf{C}$ and a graph morphism $F: G \to U\mathbf{C}$, there exists a unique functor $F^{\#}: \mathbf{Pa}_G \to \mathbf{C}$ that "extends F", i.e., such that $UF^{\#} \circ \eta_G = F$. This universal property intuitively says that to define $F^{\#}$ it is anough to say where each edge goes, i.e., to give F. This is because any path is a composition of basic edges and, since the functor $F^{\#}$ preserves composition, the path must go to the composition of the images of its edges. A formal proof is by induction on the length of paths. The reader may want to consider examples, such as G the graph of Fig. 5 and $\mathbf{C}$ the category of sets, or perhaps $\mathbf{C} = \mathbf{Pa}_{G'}$ for G' another graph.

An abstract algebra is a set together with a collection of operations on it. For instance, a group G is a set together with the following operations: a multiplication $\bullet: G^2 \to G$, an inverse map $(\ )^{-1}: G \to G$ and a neutral element $e \in G$; in addition we require the usual group axioms to hold. We say that $\bullet$ has arity 2, that $(\ )^{-1}$ has arity 1 and we view e as a zero-ary operation, i.e., as a map $G^0 = 1 \to G$ whose image is the element e. The operations $\bullet$, $(\ )^{-1}$ and e, together with their arities, define a class of algebras to which all groups belong, the "Σ-algebras" (for $\Sigma = \{\bullet, (\ )^{-1}, e\}$). More formally now, a *ranked alphabet* is a set Σ, together with a map arity: $\Sigma \to \omega$ that assigns a natural number to each $\sigma \in \Sigma$; let Σ_n denote all σ in Σ of arity n. A *Σ-algebra* is a set A together with an operation $A\sigma: A^n \to A$ for each $\sigma \in \Sigma_n$. A *Σ-homomorphism* from Σ-algebra A to Σ-algebra B is a map $f: A \to B$ such that for any $\sigma \in \Sigma_n$ we have $f \circ A\sigma = B\sigma \circ f^n$, where $f^n(a_1,\ldots,a_n) = (fa_1,\ldots,fa_n)$. Σ-algebras and Σ-homomorphisms form a category $\mathbf{ALG}_\Sigma$ under function composition, and we have a functor $U: \mathbf{ALG}_\Sigma \to \mathbf{SET}$ that sends $f: (A, (A\sigma)_{\sigma\in\Sigma}) \to (B, (B\sigma)_{\sigma\in\Sigma})$ to $f: A \to B$. Given a set X, the *free Σ-algebra* on X is the smallest set $T_\Sigma(X)$ on words over the alphabet $\Sigma \cup X \cup \{(,)\}$ such that:

(1) $X \subseteq T_\Sigma(X)$;

(2) $\Sigma_0 \subseteq T_\Sigma(X)$; and

(3) if $\sigma \in \Sigma_n$ for $n > 0$ and if $w_1,\ldots, w_n \in T_\Sigma(X)$, then $\sigma(w_1 \cdots w_n) \in T_\Sigma(X)$.

$T_\Sigma(X)$ is a Σ-algebra with constants from Σ_0, and for arity$(n) \geqslant 0$, operations defined by $T_\Sigma(X)\,\sigma$: $(w_1,\ldots, w_n) \mapsto \sigma(w_1 \cdots w_n)$. $T_\Sigma(X)$ has the following "universal property":

Given a Σ-algebra A, and a map $f: X \to A$, there exists a unique homomorphism $f^{\#}: T_\Sigma(X) \to A$ that "extends f" (i.e., $Uf^{\#} \circ \eta_X = f$, were η_X is the inclusion map of X in $T_\Sigma(X)$). The proof is well known and goes by induction on the "depth" of the words in $T_\Sigma(X)$, where depth means maximum number of nested parenthesis pairs. The result says that for defining a homomorphism $f^{\#}: T_\Sigma(X) \to A$, it is enough to say where the generators (i.e., the elements of X) of $T_\Sigma(X)$ go, i.e., to give a map $f: X \to A$. The rest of the map $f^{\#}$ is then uniquely determined by the fact that it is a Σ-homomorphism. Assume for instance σ_0, σ_1, $\sigma_2 \in \Sigma$ of arities 0, 2, 2, respectively, with $x, x' \in X$, and f as above. Then the word $w = \sigma_2(\sigma_1(x', \sigma_0)\, x)$ (of depth 2) has to go to $f^{\#}(w) = A\sigma_2(f^{\#}(\sigma(x', \sigma_0)), fx) = A\sigma_2(A\sigma_1(fx', A\sigma_0), fx)$, where the last term is a well-defined element of the algebra A. Note that in both steps we have used that $f^{\#}$ is a Σ-homomorphism that extends f.

A *partial Σ-algebra* (resp. a *relational Σ-algebra*) is obtained by allowing the operations $A\sigma: A^n \to A$ to be partial functions (resp. relations). A Σ-homomorphism is then a partial map (resp. relation) $f: A \to B$ such that $f \circ A\sigma = B\sigma \circ fn$, where $f^n((a_1,..., a_n), (b_1,..., b_n))$ is defined iff $f(a_1, b_1),..., f(a_n, b_n)$ are all defined. We then have categories $\mathbf{PALG}_\Sigma$ and $\mathbf{RALG}_\Sigma$ of partial (resp. relational) Σ-algebras and homomorphisms, and "forgetful" functors $U: \mathbf{PALG}_\Sigma \to \mathbf{PFN}$ and $\mathbf{RALG}_\Sigma \to \mathbf{REL}$. Partial algebras are in particular relational algebras, and we shall often so view them.

An example of a partial algebra is the set ω of natural numbers with the following operations:

(i) *binary*: $+$, $-$, $*$, standing for ordinary addition, subtraction, and multiplication, but with $n - m$ undefined for $m > n$;

(ii) *unary*: pos: $\omega \to \omega$, defined by $\text{pos}(n) = n$ if $n > 0$ and pos(0) undefined; and

(iii) *constant*: $0 \in \omega$.

For an example of a relational algebra, consider a nondeterministic automaton over a given input alphabet Σ. Each $\sigma \in \Sigma$ gives rise to a unary relational operation, namely, if S is the state set, $S\sigma: S \to S$ is the transition relation corresponding to the input σ. One can also have a constant corresponding to the initial state(s).

A *poset* is a set P together with a partial order, denoted $\subseteq$. The least upper bound or *supremum* of a family $(a_i)_{i \in I}$ is denoted $\bigcup_{i \in I} a_i$ if it exists. A poset is *complete* if every chain has a supremum; in particular the supremum of the empty chain is the minimum element, denoted $\perp$. $\mathcal{P}(A)$ is a complete poset partially ordered by inclusion. If P and Q are complete posets, $P \times Q$ is also a complete poset with the component-wise ordering. A map $f: P \to Q$ between two posets is *continiuous* iff for each nonempty chain $(a_i)_{i \in I}$ with supremum in P, the image has a supremum, and $f(\bigcup_{i \in I} a_i) = \bigcup_{i \in I} f(a_i)$. A theorem of Tarski guarantees that there is a minimal fixpoint for any continuous map $f: P \to P$ of a complete poset into itself, namely, the element $\bigcup_{m \in \omega} f^m \perp$.

3. The Algebraic Theory of Parallel Assignment

First, we view expressions as Σ-terms with variables. The basic idea of our construction is then to view a parallel assignment of a tuple of (values of) expressions to a corresponding tupleof variables as an *arrow* in a category; these arrows can be composed by substitution, and this category therefore becomes a kind of algebraic theory. A second step views interpretations, i.e., algebras, as functors from that category to the category of sets. This approach allows us to strip away many routine details and concentrate attention on key points; it also gives a precise and compact notation. The original idea of viewing an algebraic theory as a category whose arrows are tuples of terms is due to Lawvere [20], but we have adapted it to provide a neat description of programming concepts; a similar adaptation is due to Elgot [11].

Our exposition avoids explicit recursive definitions both at the syntactic level, where well-formed expression and substitution of well-formed expression are defined by free algebras and homomorphisms, and at the semantic level, where the meaning of derived expressions is obtained by functoriality.

Definition 1. Given a ranked alphabet Σ and a fixed denumerable set $\mathscr{X}$ whose elements we call "variables," the *algebraic theory of parallel assignment* over Σ is the category $\mathscr{P}_\Sigma(\mathscr{X})$ defined as follows:

(1) *objects*: finite subsets X, Y, Z,... of $\mathscr{X}$;

(2) *morphisms*: $\mathscr{P}_\Sigma(\mathscr{X})(X, Y) = (T_\Sigma(X))^Y$; thus an arrow $\alpha: X \to Y$ is a map $\alpha: Y \to T_\Sigma(X)$;

(3) the *composition* $\beta \circ \alpha$, for $\alpha: X \to Y$ and $\beta: Y \to Z$, is given in Fig. 6, where $\alpha^\#$ is the unique homomorphism guaranteed for α by the universal property of $T_\Sigma(Y)$; and

(4) finally, the identities of $\mathscr{P}_\Sigma(\mathscr{X})$, $1_X: X \to X$, are the inclusion maps $\eta_X: X \to T_\Sigma(X)$.

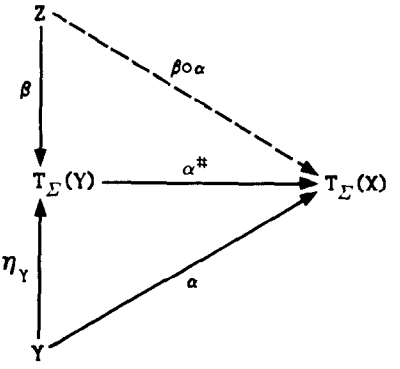

Fig. 6. Composition of parallel assignments.

Variables in a set X will be denoted x, y, z, x_1, x_2, y_1, y_2, etc. When reasoning with generic finite subsets X, Y of $\mathscr{X}$ we sometimes find it useful to write $X = \{x_1,..., x_n\}$ and $Y = \{y_1,..., y_m\}$, in the understanding that neither are the variables ordered nor are the subsets necessarily disjoint. Words in $T_\Sigma(X)$ are denoted u, v, w, w_1, w_2, etc. We can think of a morphism $\alpha: X \to Y$ in $\mathscr{P}_\Sigma(\mathscr{X})$ as "Y-indexed list of words," and hence we sometimes use *list* (or tuple) notation $(w_1,..., w_m): X \to Y$ to denote the map sending y_j in Y to w_j in $T_\Sigma(X)$ for $1 \leqslant j \leqslant m$. A variant of this is the *assignment notation*,

$$(y_1 := w_1, y_2 := w_2,..., y_m := w_m): X \to Y,$$

which makes explicit the variables in the target; in pictures, we may write these assignements in parallel over an arrow, e.g.,

$$\begin{array}{c} y_1 := w_1 \\ y_2 := w_2 \\ \vdots \\ y_m := w_m \\ X \longrightarrow Y. \end{array}$$

Arrows in $\mathscr{P}_\Sigma(\mathscr{X})$ correspond to *parallel assignments* in programming, and composition in $\mathscr{P}_\Sigma(\mathscr{X})$ is *word substitution*, as illustrated by the following:

EXAMPLE. Let $\Sigma = \{+, \bullet, -, 0, 1\}$ with $+$, $\bullet$ and $-$ binary, and 0 and 1 constants. Let $X = \{x, y\}$, $Y = \{x, z, v\}$, $Z = \{x, z\}$. Then the composition of the parallel assignments

$$\begin{array}{ll} x := \bullet(+(xx)\, y), & \\ z := -(x1), & x := +(+(zv)\, x), \\ v := +(y_1), & z := \bullet(xv), \\ X \longrightarrow Y, & Y \longrightarrow Z, \end{array}$$

is given by

$$\begin{array}{l} x := +(+(-(x1) + (y1)) \bullet (+(xx)\, y)), \\ z := \bullet(\bullet(+(xx)\, y) + (y1)), \\ X \longrightarrow Z \end{array}$$

An algebraic theory compactly describes the action, not only of the basic operations $\sigma \in \Sigma$ of a given Σ-algebra A, but also its "derived operations" obtained by composition and tupling. Indeed we can associate to each Σ-algebra A, a functor $\underline{A}: \mathscr{P}_\Sigma(X) \to \mathbf{SET}$ which maps a parallel assignment $(y_1 := w_1,..., y_m := w_m): X \to Y$ to its interpretation as a function $(y_1 := w_1,..., y := w_m): A^X \to A^Y$, mapping an

X-indexed list of elements of A to the Y-indexed list of "results after executing the assignments in parallel in the Σ-algebra A."

THEOREM 2. *The category* $\mathbf{ALG}_\Sigma$ *of* Σ*-algebras and homomorphisms is isomorphic to the category* $\mathbf{ALG}_{\mathscr{P}_\Sigma(\mathscr{X})}$ *having as its* objects *functors* $\underline{A}: \mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{SET}$ *such that there is a set* A *such that*

(1) *for each object* X *of* $\mathscr{P}_\Sigma(\mathscr{X})$, $\underline{A}(X) = A^X$; *and*

(2) *if* X *is* $\{x_1,..., x_n\}$, *then for* $1 \leqslant j \leqslant n$, $\underline{A}(x_j)$: $A^X \to A^{\{x_j\}}$ *is the "*j*th projection" mapping each list* $a: X \to A$ *to the list* $x_j \circ a$, *i.e., mapping* $(a_1,..., a_n)$ *to* a_j; *morphisms are then the natural transformations between such functors.*

The technicalities of a proof are given in Appendix A. At this point, however, let us remark that conditions (1) and (2) guarantee that the functor $\underline{A}$ does exactly what we want it to do. The key reasons are that any parallel assignment can be obtained by composition and tupling from variables and the basic assignments $x := \sigma(x'_1,..., x'_n)$ for $\sigma \in \Sigma$, which are essentially the corresponding operations $\underline{A}\sigma$, and that $\underline{A}$ preserves not only composition but also tupling (by condition (2)). The correspondence between homomorphisms $f: A \to B$ and natural transformations $\underline{f}: \underline{A} \to \underline{B}$ is less obvious, but can be seen for the basic assignments using condition (2).

We want relational algebras also to fit in this framework. The trick [9] is to consider a relational algebra to be an ordinary algebras on its powerset. This viewpoint is used, for example, in the usual proof that nondeterminism adds nothing to regular languages, by passing from a nondeterministic automaton $\Sigma \times S \to S$ to the corresponding deterministic one $\Sigma \times \mathscr{P}(S) \to \mathscr{P}(S)$. In order not to lose information in this passage, we must characterize the additional properties (generally referred to as "additivity") of the new ordinary operations and homomorphisms that correspond to the relational ones:

THEOREM 3. *The category* $\mathbf{RALG}_\Sigma$ *of relational* Σ*-algebras and homomorphisms is isomorphic to the subcategory of* $\mathbf{ALG}_\Sigma$ *with* objects *the* Σ*-algebras with carrier a powerset* $\mathscr{P}(A)$, *such that for each* $n \geqslant 0$, $\sigma \in \Sigma$ *with* $arity(\sigma) = n$, *and* $(A_1,..., A_n) \in \mathscr{P}(A)^n$ *we have*

(i) $\mathscr{P}(A)\,\sigma(A_1,..., A_n) = \bigcup\{\mathscr{P}(A)\,\sigma(\{a_1\},..., \{a_n\}) \mid (a_1,..., a_n) \in A_1 \times \cdots \times A_n\}$

and having as morphisms *the* Σ*-homomorphisms* $f: \mathscr{P}(A) \to \mathscr{P}(B)$ *such that for each* $A' \in \mathscr{P}(A)$

(ii) $fA' = \bigcup_{a \in A'} f\{a\}$.

The proof is given in Appendix A.

Combining Theorems 2 and 3, we see that to each relational Σ-algebra A corresponds bijectively a functor $\underline{\mathscr{P}(A)}$: $\mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{SET}$. Moreover, a parallel assignment $\alpha: X \to Y$ corresponds to an (additive) map $\underline{\mathscr{P}(A)}(\alpha)$: $\mathscr{P}(A)^X \to \mathscr{P}(A)^Y$. From $\mathscr{P}(A)(\alpha)$ we can now obtain $\underline{A}(\alpha)$, the relation computed by the parallel

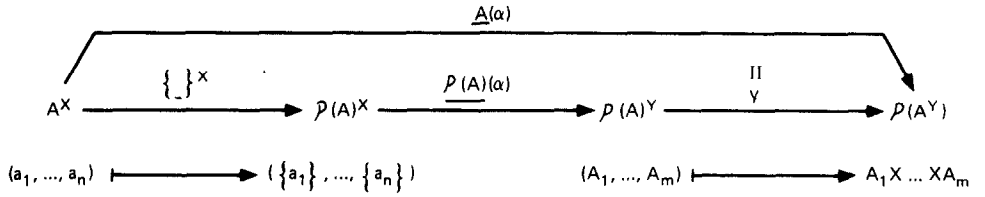

FIGURE 7

assignment α in the relational Σ-algebra A, defined to be the composition in Fig. 7, where $\{\ \}$ sends $(a_1,...,a_n)$ to $(\{a_1\},...,\{a_n\})$ and $\prod_Y$ sends $(A_1,...,A_m)$ to $A_1 \times \cdots \times A_m$.

We hope that context will make clear whether $\underline{A}(\alpha)$ denotes such a relation or a map arising from a $\mathscr{P}_\Sigma(\mathscr{X})$-algebra, $\underline{A}: \mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{SET}$. Contrary to what might be expected, for a given relational Σ-algebra A, the assignment to each $\alpha: X \to Y$ in $\mathscr{P}_\Sigma(\mathscr{X})$ of its corresponding relation $\underline{A}(\alpha)$ *does not* give a functor $\underline{A}: \mathscr{P}_\Sigma(X) \to \mathbf{REL}$, but rather a graph morphism between the corresponding underlying graphs $\underline{A}: \mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{REL}$ (we omit the U for convenience; incidentally, readers who are suspicious about "big graphs" (such as **REL**) may take refuge in some Grothendieck universe). If the relational algebra A is actually a partial algebra, then $\underline{A}(\alpha)$ is a partial function, and we get a graph morphism $\underline{A}: \mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{PFN}$.

4. Flow Diagram Programs with Parallel Assignments

DEFINITION 4. A *flow diagram program schema* (FDPS) with parallel assignment and with operations Σ, is a graph morphism $S: G \to \mathscr{P}_\Sigma(\mathscr{X})$, where $G = (G, v_0, v_1)$ is a *fully protected reachable finite graph*, i.e., a finite graph with given nodes v_0 and v_1 such that no edges go out of v_1 (the *output node*) or into v_0 (the *input node*); and every node of G lies on some path from v_0 to v_1.

EXAMPLE. Let $\Sigma = \{+, -, \bullet, \text{zero}, \text{pos}, 0, 1\}$ with $+$, $-$, $\bullet$ binary, zero and pos unary, and 0, 1, constants. Let $S: G \to \mathscr{P}_\Sigma(\mathscr{X})$ be the graph morphism with source the graph on the left and target of the graph on the right of Fig. 8. The input and output nodes of G have been circled. It is often convenient to think of S in terms of its image, rather than its underlying graph.

DEFINITION 5. A *flow diagram program* (FDP) with parallel assignment and with operations Σ is a pair (S, A) with S a flow diagram program scheme and A a relational Σ-algebra.

EXAMPLE. Let S be as in Fig. 8 with $A = \omega$ and with $+$, $-$, $\bullet$ the ordinary addition, subtraction and multiplication; with pos(n) defined and equal to n iff n is positive; and with zero(n) defined and equal to 0 iff n is 0. With these operations and

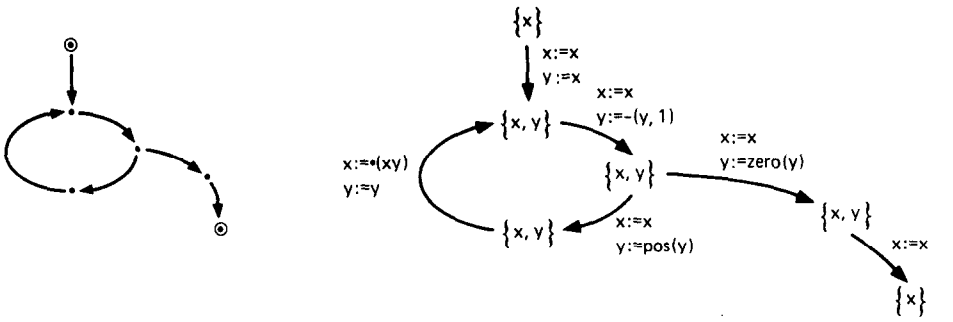

FIG. 8. Source and target for a flow diagram program scheme.

the consants 0, 1, ω is a relational (actually a partial) Σ-algebra. (See also the example of a Σ-algebra discussed in Section 2.)

Remark. A flow diagram program (S, A) determines a graph morphism to **REL**, namely, the composition

$$G \xrightarrow{S} \mathscr{P}_{\Sigma}(\mathscr{X}) \xrightarrow{A} \mathbf{REL}.$$

This shows the connection with the more general definition of a flow diagram program as graph morphism $P: G \to \mathbf{REL}$ (or $P: G \to \mathbf{PFN}$) in [13], which also suggests

DEFINITION 6. Given a flow diagram program (S, A), its *behavior* is the relation $B(S, A): A^{S(v_0)} \to A^{S(v_1)}$ defined to be the union of relations $\bigcup\{(\underline{A} \circ S^{\#}(f) \mid f: v_0 \to v_1$ in $\mathbf{Pa}_G\}$, where $(\underline{A} \circ S)^{\#}$ is the unique functor $\mathbf{Pa}_G \to \mathbf{REL}$ extending $\underline{A} \circ S: G \to \mathbf{REL}$ guaranteed by the universal property of $\mathbf{Pa}_G$.

EXERCISE. Check that the behavior $B(S, A)$ of the program in the example above is the factorial function $n \mapsto n!$.

5. THE BEHAVIOR AND CORRECTNESS OF RECURSIVE PARALLEL PROGRAMS

DEFINITION 7. Given a ranked alphabet Σ of "operation symbols" and another finite alphabet $\Pi = \{P_1, \ldots, P_n\}$ of "procedure symbols," the *algebraic theory of parallel assignment with operations Σ and procedures Π*, denoted $\mathscr{P}_{\Sigma,\Pi}(\mathscr{X})$, is the algebraic theory $\mathscr{P}_{\Sigma \cup \Pi}(\mathscr{X})$.

DEFINITION 8. *A procedure schema* is a graph morphism $G \to \mathscr{P}_{\Sigma,\Pi}(\mathscr{X})$ from a finite, fully protected and reachable G. *A recursive flow diagram program schema* (RFDPS) with operations Σ and procedures Π is a collection $S = (S_1, \ldots, S_n)$ of procedure schemas, one for each procedure symbol in $\Pi = \{P_1, \ldots, P_n\}$, such that if P_j

has arity m_j, then $\#S_j(v_0^j) = m_j$ and $\#S_j(v_1^j) = 1$ (these are the image nodes of the input and output nodes of G_j under $S_j: G_j \to \mathscr{P}_{\Sigma,\Pi}(\mathscr{X})$).

EXAMPLE. Let Σ as in (4.2) and let $\Pi = \{\text{FACT}\}$, FACT of arity 1. Then Fig. 9 is an RFDPS on Σ and Π.

Remarks. We shall write the label "P_j:" to the left of the scheme S_j in diagrams like that of Fig. 9. Although we only needed one procedure symbol, of arity 1, to calculate factorial, there is no general bound on the number that may be needed. This suggests letting $\Pi' = \{P, Q, R, ..., P_1, ..., P_n, ...\}$ be such that the set $\Pi'_n \subseteq \Pi'$ of procedure symbols of arity n is denumerable for each n, and then defining $\mathscr{P}_{\Sigma,\Pi'}(\mathscr{X}) = \mathscr{P}_{\Sigma\cup\Pi'}(\mathscr{X})$. Any finite ranked Π can be seen as contained in Π', so that $\mathscr{P}_{\Sigma,\Pi}(\mathscr{X})$ is then a subtheory of $\mathscr{P}_{\Sigma,\Pi'}(\mathscr{X})$. Then *a recursive flow diagram program scheme* with operations in Σ is an RFDPS on Σ and Π, in the sense of Definition 8, for some finite $\Pi \subseteq \Pi'$.

DEFINITION 9. A *recursive flow diagram program* (RFDP) on Σ and Π is a pair (S, A) with S an RFDPS on Σ and Π, and A a relational Σ-algebra.

For example, the pair (S, A) with S as in the above example and $A = \omega$ as in Section 4 is a recursive flow diagram program.

Let $R = (R_1, ..., R_n)$ be a family of relations, each $R_j: A^{mj} \to A$ with arity mj equal to that of P_j; this makes the relational Σ-algebra A into a $\Sigma \cup \Pi$-relational algebra, say A_R and we can then associate to each S_j in S an ordinary FDP with operations $\Sigma \cup \Pi$, namely, (S_j, A_R) and hence a behavior $B(S_j, A_R)$. We call the collection $B(S, A_R) = (B(S_1, A_R), ..., B(S_n, A_R))$ the *behavior of* (S, A) *relative to* $R = (R_1, ..., R_n)$. This defines a map $B(S, A\_): \mathscr{P}(A^{m1} \times A) \times \cdots \times \mathscr{P}(A^{mn} \times A) \to \mathscr{P}(A^{m1} \times A) \times \cdots \times \mathscr{P}(A^{mn} \times A)$ that sends $R = (R_1, ..., R_n)$ to $B(S, A_R) = (B(S_1, A_R), ..., B(S_n, A_R))$; note that we have identified A^{Xj} with A^{mj} for $Xj = S_j(v_0^j)$ with $\#Xj = mj$.

THEOREM 10. *$B(S, A\_)$ is continuous.*

Proof. For any chain $(R_i = (R_n^i))_{i \in I}$ with I totally ordered and $R_k^i \subseteq R_k^j$ if $1 \leqslant k \leqslant n$ and $i \leqslant j$, we are to prove that

$$\bigcup_{i \in I} B(S, A_{R_i}) = B(S, A_{\bigcup_{i \in I} R_i})$$

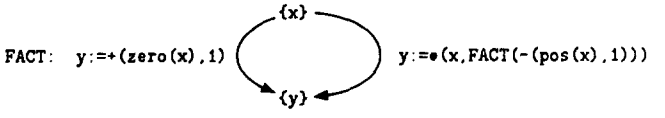

FIG. 9. A diagram for factorial.

or more explicitly (see Definition 9)), that

$$\bigcup_{i\in I}\left(\bigcup\{(\underline{A}_{R_i}\circ S_j)^{\#}(f)\mid f\colon v_0^j\to v_1^j \text{ in } \mathbf{Pa}_{Gj}\}\right)=\bigcup\{(\underline{A}_{\bigcup_i R_i}\circ S_j)^{\#}(f)\mid f\colon v_0^j\to v_1^j \text{ in } \mathrm{PA}_{Gj}\}.$$

Now, if we prove that $\bigcup_{i\in I}(\underline{A}_{R_i}\circ S_j)^{\#}(f)$ for each $f\colon v_0^j\to v_1^j$ in $\mathbf{Pa}_{Gj}$ for $1\leqslant j\leqslant n$, we are done. But this follows from the universal property of $\mathbf{Pa}_{Gj}$ if we prove that $\bigcup_{i\in I}(\underline{A}_{R_i}\circ S_j)^{\#}(e)=(\underline{A}_{\bigcup_i R_i}\circ S_j)^{\#}(e)$ for each edge e in Gj for $1\leqslant j\leqslant n$. For this, it suffices to prove

LEMMA 11. *For any $\alpha\colon X\to Y$ in $\mathscr{P}_{\Sigma,\Pi}(\mathscr{C})$ and any chain $(R_i)_{i\in I}$ as above,*

$$\bigcup_{i\in I}(A_{Ri})^{\#}(\alpha)=(A_{\bigcup_i Ri})^{\#}(\alpha).$$

Proof. A simple but tedious induction on the *depth* of (i.e., maximum number of nested parentheses occurring in) words $w\in T_{\Sigma\cup\Pi}(\mathscr{C})$ using the following facts:

(1) if $\alpha\colon X\to Y$, $\#Y=1$, and w in $\mathscr{P}_{\Sigma,\Pi}(\mathscr{C})$ is of the form $w=\sigma(w_1\cdots w_n)$ for $\sigma\in\Sigma$ (resp., of the form $w=P_j(v_1\cdots v_m)$), then $\underline{A}_R(w)=\underline{A}_R(w_1,\ldots,w_n)\circ A\sigma$ (resp., $\underline{A}_R(v_1,\ldots,v_m)\circ R_j$);

(2) given $(w_1,\ldots,w_m)\colon X\to Y$ in $\mathscr{P}_{\Sigma,\Pi}(\mathscr{C})$, then $\underline{A}_R(w_1,\ldots,w_m)=(\underline{A}_R(w_1),\ldots,\underline{A}_R(w_m))$ (recall the definition of tupling of relations in Section 2);

(3) composition of relations is continuous, i.e., if $(Q_i)_{i\in I}$ and $(R_i)_{i\in I}$ are chains of relations, then $(\bigcup_i Q_i)\circ(\bigcup_i R_i)=\bigcup_i(Q_i\circ R_i)$; and

(4) tupling of relations is continuous, i.e., if $(R_j^i)_{i\in I}$ are chains of relations for $1\leqslant j\leqslant n$, then $\bigcup_i(R_1^i,\ldots,R_n^i)=(\bigcup_i R_1^i,\ldots,\bigcup_i R_n^i)$.

Facts (1) and (2) follow from the definition of $\underline{A}_R$ using the functoriality of $\mathscr{P}(\underline{A}_R)\colon\mathscr{P}_{\Sigma,\Pi}(\mathscr{C})\to\mathbf{SET}$; (3) and (4) follow by a mutual inclusion argument. Because of facts (2) and (4) we only have to consider morphisms $\alpha\colon X\to Y$ with $\#Y=1$, i.e., words, and then use induction on the depth. This completes the proof of the lemma, and hence of the theorem. ∎

We are now ready to define the behavior of an RFDOP with "call by name" or "copy rule" semantics. Note that if we are only interested in the first "main" procedure, we can select just the first component of the behavior.

DEFINITION 12. The *behavior* $B(S,A)$ of an RFDP (S,A) is the least fixpoint of the continuous function $B(S,A\_)$; thus it is given by the formula

$$B(S,A)=\bigcup_{k\in\omega}B(S,A\_)^k(\varnothing,\ldots,\varnothing).$$

From this definition and the last theorem we get the following reduction of the correctness problem for RFDPs to the previously studied problem of correctness for ordinary FDPs [13, 14].

THEOREM 13. *Given an* RFDP (S, A) *on* Σ *and* Π *and a family* $R = (R_1,..., R_n)$ *of relations with* R_j *of the same arity as* $P_j \in \Pi$, *if* $B(S, A_R) \subseteq (R_1,..., R_n)$, *then* (S, A) *is partially correct with respect to* $R_1,..., R_n$ *in the sense that* $B(S, A) \subseteq (R_1,..., R_n)$.

Proof. Let us write $B(S, A_Q) = F(Q)$. Then $F(R) \subseteq R$ and $\perp \subseteq R$ imply by induction that $\perp \subseteq F^k(\perp) \subseteq F^k(R)$ for all $k \in \omega$. Thus $B(S, A) \subseteq R$. ∎

6. EXTENSIONS TO MANY SORTS AND COARITY

The results in this paper can be generalized in several natural directions. For expository simplicity we have restricted the body of the paper to a special situation. The two main generalizations that we have in mind are given in the following two paragraphs.

To generalize from one sort to many sorts, for example from *integer* to {*integer*, *real*, *boolean*, *array*}, with operations taking arguments and values in different sorts, we now consider *I-sorted relational algebras* $((A_i)_{i \in I}, (A\alpha)_{\alpha \in \Sigma})$ with relational operations $A\alpha$ of the form: $A\alpha: A_{i1} \times \cdots \times A_{in} \to A_i$ for $i1,..., in$, $i \in I$ and $n \geqslant 0$. The algebraic theory of parallel assignment $\mathscr{P}_\Sigma(\mathscr{X})$ generalizes to many sorts without serious difficulty; for the case of ordinary algebraic theories, see [3] and the simplification in [17]. In fact, all results in this paper stand exactly as written for the many-sorted case.

Coarity allows operations of the form $A\alpha: A^n \to A^m$ or in the many-sorted case of the form $A\alpha: A_{i1} \times \cdots \times A_{in} \to A_{j1} \times \cdots \times A_{jm}$. The only technical difficulty is that relational algebras cannot be treated as ordinary algebras on the powerset if there are operations with coarity. The reason lies in the difference between sets of tuples and tuples of sets: we would want to describe a relational operation $A\alpha: A^n \to A^m$ as an additive map $\mathscr{P}(A)\,\alpha: \mathscr{P}(A)^n \to \mathscr{P}(A)^m$; but this cannot be done in general. Although relations $A\alpha: A^n \to A^m$ are in bijective correspondence with maps $A_\alpha: A^n \to \mathscr{P}(A^m)$, there is no bijective correspondence between $\mathscr{P}(A^m)$ and $\mathscr{P}(A)^m$. The best we have is the map $\Pi: \mathscr{P}(A)^m \to \mathscr{P}(A^m)$ sending $(A_1,..., A_m)$ to $A_1 \times \cdots \times A_m$; it is inective for tuples $(A_1,..., A_n)$ with each A_i nonempty, but it is not surjective. Therefore we cannot describe a relational algebra as a functor $\mathscr{P}(A): \mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{SET}$. However, this causes no essential harm, because what we will actually use is the *graph morphism* $\underline{A}: \mathscr{P}_\Sigma(\mathscr{X}) \to \mathbf{REL}$ as defined at the end of Section 3; but it can also be defined directly by induction for either one or many sorts, with or without coarity. We now consider $\mathscr{P}_\Sigma(\mathscr{X})$ simply as a graph, with its arrows "tuples" of Σ-words. The meaning of a variable is a projection, and the meaning of a basic word $\alpha(X_1,..., X_n)$ is given by the corresponding relational operation $A\alpha$. The rest is then defined by induction on the depth and tupling of relations as in (1) and (2) of Lemma 11. All he results of the paper then go through for coarity (either one or many-sorted) with exactly the same arguments.

APPENDIX A: PROOFS OF RESULTS

Proof of Theorem 2. (Lawvere [20]). Though the functor $\underline{A}: \mathscr{P}_\Sigma(\mathscr{K}) \to \mathbf{SET}$ contains information about how all parallel assignments are interpreted, actually everything depends on how the assignements corresponding to basic operations are interpreted, i.e., those of the form $x := \alpha(x_1 \cdots x_n): X = \{x_1, ..., x_n\} \to \{x\} = Y$, for $\alpha \in \Sigma_n$ and $n \geqslant 0$. For each α, there are many such assignments as the sets of variables X and Y vary. However, they all must have essentially the same meaning because for $X' = \{x_1', ..., x_n'\}$ and $Y' = \{x'\}$, the following diagram commutes in $\mathscr{P}_\Sigma(\mathscr{K})$:

$$\begin{array}{ccc} X & \xrightarrow{x := \alpha(x_1 \cdots x_n)} & Y \\ {\scriptstyle x_1' := x_1 \atop \vdots \atop x_n' := x_n}\big\downarrow & & \big\downarrow{\scriptstyle x' := x} \\ X' & \xrightarrow[x' := \alpha(x_1' \cdots x_n')]{} & Y' \end{array}$$

and by functoriality, so does the image diagram under $\underline{A}$ in **SET**. Taking an explicit bijection of X with the natural numbers, i.e., numbering the variables as we have already done and using the bijection $[x_1, ..., x_n]\ A^X \to A^n: a \mapsto (a(x_1), ..., a(x_n))$ we have the desired correspondence with operations $A\alpha: A^n \to A$. The rest of the argument consists of realizing that any morphism in $\mathscr{P}_\Sigma(\mathscr{K})$ is obtained from the variables and the basic operations by composition and tupling, due to the inductive definition of Σ-words and to the definition of composition in $\mathscr{P}_\Sigma(X)$, which as we have already seen is word substitution. Now, $\underline{A}$ preserves composition (by functoriality) and also tupling (this is actually equivalent to the condition (2) which says that $\underline{A}$ preserves direct products). Hence the image of any arrow in $\mathscr{P}_\Sigma(\mathscr{K})$ is uniquely determined by those of the basic ones, which in turn correspond to the giving of a Σ-algebra structure. This proves the bijection on the objects.

For the bijection on morphisms, note for $f: A \to B$ a map, that condition (2) forces a natural transformation $\underline{f}(X) = f^X: A^X \to B^X$ for each $X \in \mathscr{P}_\Sigma(\mathscr{K})$. Specializing naturality to the basic arrows $\alpha(x_1, ..., x_n)$ we see that f is in fact a Σ-homomorphism between the Σ-algebras corresponding to $\underline{A}$ and $\underline{B}$; but any such homomorphism defines in fact a natural transformation from $\underline{A}$ to $\underline{B}$ (proof by induction on the depth of words, using the fact that both $\underline{A}$ and $\underline{B}$ preserve composition and tupling). ∎

Proof of Theorem 3. Given a relation $f: A \to B$, define $f^{\S}: A \to \mathscr{P}(B)$ as the map sending a to $\{b \in B \mid f(a, b)\}$; and given a map $f^{\S}: A \to \mathscr{P}(B)$, define $f^{\dagger}: \mathscr{P}(A) \to \mathscr{P}(B)$, with the property that $f^{\dagger}(A') = \bigcup_{a \in A'} f^{\dagger}(\{a\})$, by $f^{\dagger}(A') = \bigcup_{a \in A'} f^{\S}(a)$. This defines a bijective correspondence, as does passing from a relation $A\sigma: A^n \to A$ to the map $A\sigma^{\S}: A^n \to \mathscr{P}(A)$ as above, and also passing from a map $A\sigma^{\S}: A_n \to \mathscr{P}(A)$ to a map$A\sigma^{\ddagger}: \mathscr{P}(A)^n \to \mathscr{P}(A)$ with the property that

$$A\sigma^{\ddagger}(A_1, ..., A_n) = \bigcup \{A\sigma^{\ddagger}(\{a_1\}, ..., \{a_n\}) \mid (a_1, ..., a_n) \in A_1 \times \cdots \times A_n\}$$

by

$$A\sigma^{\ddagger}(A_1,...,A_n) = \bigcup \{A\sigma^{\dagger}(a_1,...,a_n) \mid (a_1,...,a_n) \in A_1 \times \cdots \times A_n\}. \quad \blacksquare$$

APPENDIX B: REPNOD

We give a syntax, semantics, and further examples for the programming language REPNOD embodying the features discussed in this paper; because our purpose is purely illustrative we do no try to be completely precise.

The following definition of REPNOD syntax uses these conventions: keywords are boldface; so are parentheses and commas when used as terminal symbols in the syntax definition, but not in program texts; ":=" is a special terminal symbol used for assignment; nonterminals are enclosed in ⟨ ⟩; ↵ is "carriage return," which begins a new line (it does not appear in program text, but its effect can be seen); X^* means zero or more instances of X; X^+ means one or more; $[X]$ indicates an optional occurrence of X; "|" means "or;" and "::=" is metasyntax for "is defined as." Now the abstract syntax:

(1) ⟨prog⟩ ::= ⟨proc⟩$^+$

(2) ⟨proc⟩ ::= **proc** ⟨p-name⟩⟨var-list⟩[**entry** ⟨label⟩] ↵
⟨assign⟩$^+$ **exit**[⟨label⟩]⟨var-list⟩ **corp** ↵

(3) ⟨assign⟩ ::= [**source**[⟨label⟩][⟨var-list⟩]] **assign** [↵]
⟨test⟩*⟨op⟩*[**target** ⟨label⟩ ↵]

(4) ⟨test⟩ ::= ⟨pred⟩⟨term-list⟩(↵ | ,)

(5) ⟨op⟩ ::= ⟨var-list⟩ := ⟨term-list⟩(↵ | ,)

(6) ⟨var-list⟩ ::= ([⟨var⟩(,⟨var⟩)*])

(7) ⟨term-list⟩ ::= ([⟨term⟩(,⟨term⟩)*])

⟨label⟩ and ⟨var⟩ will be lower and upper case roman characters, respectively, with primes or subscripts if needed; they denote node and variable names, respectively. ⟨p-name⟩ will be upper case roman strings, for procedure names, i.e., elements of Π. ⟨pred⟩ refers to a special subset, Δ say, of Σ, consisting of predicate symbols. ⟨term⟩ refers to well-formed (SUP)-terms, using prefix, infix, or whatever mixfix notation is convenient.

In rule (2), "[**entry**⟨label⟩]" optionally gives a name to the entry node of a procedure; the first ⟨var-list⟩ gives its parameters; the one or more ⟨assign⟩ constitute its body; "**exit**[⟨label⟩]⟨var-list⟩" (optionally names the exist node and) names the variables returned; "**corp**" terminates a procedure definition. Note that ⟨label⟩s are strictly **local** to a procedure. In rule (3), names are optionally given to the **source** and **target** nodes of an ⟨assign⟩ edge, with its parallel ⟨test⟩ and ⟨op⟩ as body; the first time that a source is named, the variables that occur at that node must be given in the ⟨var-list⟩; the default for a **source** is given by the **target** (or **entry**) node above it in

the program text; dually, the default for a **target** node is given by the **source** (or **exit**) node below it. Variables for **targets** are given at the corresponding **source**; we require that each **target** connect to a unique **source** (or **exit**) in order for a REPNOD program to be well formed. The variables used in the ⟨test⟩ and ⟨op⟩ of an ⟨assign⟩ must all be on its source node, and the variables assigned to, on its **target** node. If a target node variable x is not mentioned, default is the identity assignment $x := x$. The ⟨pred⟩s in ⟨test⟩ (rule (4)) must be in Δ and must be interpreted as partial identity functions.

Basically, the body of this paper gives the semantics of REPNOD. One way that our intention for REPNOD differs from this theory is in coarity, that is, in operators which return tuples of values. This requires an expanded notion of signature, with a function rank: $\Sigma \to \omega \times \omega$ whose first component gives arity and whose second gives coarity (which must be nonzero). It is simplest to let such a signature Σ define another, Σ^1, with trivial coarity, consisting of the untuplings of elements of Σ; thus if $\sigma \in \Sigma$ with $\text{rank}(\sigma) = (n, m)$, $m \geqslant 1$, let $\sigma_1, \ldots, \sigma_m \in \Sigma^1$ with each $\text{arity}(\sigma_i) = n$. Then an instance of σ is regarded as an abbreviation for the tupling $(\sigma_1, \ldots, \sigma_m)$, and we are on familiar ground. But (warning!) if nondeterminism is rampant, you may not get the results you expect; this has to do with the difference between sets of tuples and tuples of sets. However, as explained in Appendix A, things can be patched by giving suitable recursive definitions for tuple-valued relations in an interpretation. Note that we permit nontrivial coarity in the signature Π of procedure names.

A further development of REPNOD, which is quite straightforward, provides for many sorts. What we have so far has just one sort, **nat**, for natural numbers. By using many-sorted signatures and theories, as in [16], the present development is easily extended. An example later uses **list**, for list of integers, in addition to **nat**. The syntax definition is changed by replacing rule (6) by

(6′) ⟨var-list⟩ ::= ([⟨var⟩: ⟨sort⟩(,⟨var⟩: ⟨sort⟩)*])

so that a ⟨sort⟩ is always declared for a ⟨var⟩; of course, ⟨term⟩ will have to respect ⟨sort⟩ in order to be well-formed.

Now some examples. First a program which requires a different treatment of coarity to give the intended result, which is $S(N) = \{(A, B) \mid A + B = N\}$.

```
proc S(N)
    assign A := 0, B := N
    source (A, B) assign (A, B) := PAIRS(A, B)
    exit (A, B) corp
proc PAIRS(A, B) entry a
    source a target b
    source a assign(A, B) := PAIRS(inc(A), dec(B)) target b
    exit b(A, B) corp
```

Here is a more complex program, to sort a list A of numbers.

```
proc SORT(A: list)
    assign N := length(A)
    source (N: nat, A: list) assign A := SORTN(N, A)
    exit (A: list) corp
proc SORTN(N: nat, A: list) entry a
    assign (N, A, T) := EXCH(pos(N), A, 0, 1) target b
    source b(N: nat, A: list, T: nat) assign one(N) target c
    source b assign zero(T) target c
    source b assign pos(T), A := SORTN(pos(dec(N)), A) target c
    source a assign zero(N)
    exit c(A: list) corp
proc EXCH(N: nat, A: list, T: nat, J: nat) entry a
    assign J < N target b
    source b(N: nat, A: list, T: nat, J: nat) A[J + 1] < A[J]
          A[J] := A[J + 1], A[J + 1] := A[J], T := 1 target c
    source b assign A[J + 1]TA[j]
    source c(N: nat, A: list,T: nat, J: nat)
        assign(N, A, T) := EXCH(N, A, T, inc(J)) target c
    source a assign J = N
    exit d(N: nat, A: list, T: nat) corp
```

APPENDIX C: A SAMPLE PROOF

This appendix gives an example proof of the partial correctness of a nondeterministic program DIV for finding all the divisors of a number A smaller than the $(n_0 + 1)$th prime, denoted $p[n_0 + 1]$. Our interpretation consists of the set of positive natural numbers less than $p[n_0 + 1]$, with operations of multiplication (defined if the result is less than $p[n_0]$), integer division, successor (defined for all values except for $p[n_0 + 1] - 1$), and an array $p[1 \cdots n_0]$ of the first n_0 prime numbers. In addition there is a predicate $p[k] \mid N$, divisibility by $p[k]$, for each $1 \leqslant k \leqslant n_0$. The diagram for DIV is given in Fig. 10.

We "guess" the relation R such that $R(A)$ is the set of divisors of A, for each $1 \leqslant A \leqslant p[n_0 + 1] - 1$. The program is partially correct, because substituting DIV for R, the values we get along all possible paths are:

(i) $p[N]$ a divisor of A, or

(ii) $A/p[N]$, with $p[N]$ a divisor of A, or

(iii) $p[N] \bullet M$, with M a divisor of $A/p[N]$

which are all divisors of A. In addition, it is easy to see that R is a fixpoint: if $M = p[k_1]^{a_1} \cdots p[k_q]^{a} q$, with $k_1 < \cdots < k_q$, divides A and if k_0 is the fist prime that

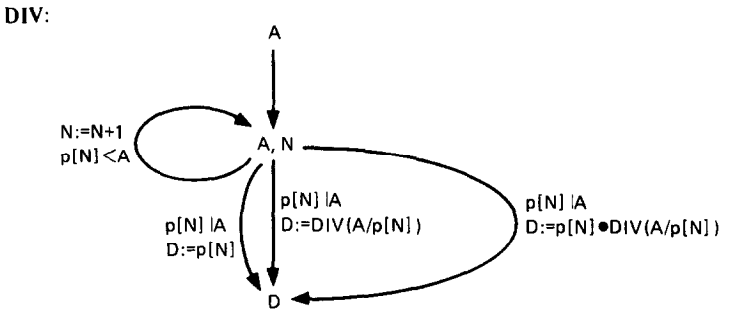

FIG. 10. Diagram for the divisors program.

divides A, then, assuming that $k_0 < k_1$, M divides $A/p[k_0]$ and is computed by going $k_0 - 1$ times along the loop and taking the edge in the middle to the output node. Similarly, it is computed by going the same number of times along the loop and taking the edge in the right to the output node, if $k_0 = k_1$.

ACKNOWLEDGMENTS

We wish to thank Rod Burstall for his encouragement and hospitality, and John Reynolds for the idea of using sets of variables as the objects of a theory. We also thank Jesse Wright, Saunders MacLane, Bill Lawvere, Jim Thatcher, and Eric Wagner for their inspiration. Andrezj Blikle deserves a special thank you for his helpful comments, and for presenting the original version of this paper at a conference. This paper is dedicated to the memory of Cal Elgot, who was a fundamental influence and source of inspiration and encouragement to the entire area of category theory applied to computation.

REFERENCES

1. M. A. ARBIB AND E. MANES, "Arrows, Structures, and Functors," Academic Press, New York, 1975.
2. J. BACKUS, Can programming be liberated from the von Neumann style? *Comm ACM* **21** (8) (1978), 613–641.
3. J. BENABOU, Structures Algébriques dans les Catégories. *Cahiers Topologie Géom. Différentielle* **10** (1968), 1–126.
4. A. BLIKLE, "An Analysis of Programs by Algebraic Means," *Banach Center Publications*, pp. 167–213, 1977.
5. R. M. BURSTALL, An algebraic description of programs with assertions, verification, and simulation, *In* "Proceedings of ACM Conference on Proving Assertions about Programs," pp. 7–14, 1972. ACM, New York, 1972.
6. R. M. BURSTALL, "A Note on Program Proof by a Continuation Method," Technical Report, Univ. of Edinburgh, Department of Artificial Intelligence, 1975.
7. R. M. BURSTALL AND J. A. GOGUEN, Putting theories together to make specifications, *in* "Proceedings, Fifth International Joint Conference on Artificial Intelligence" pp. 1045–1058, 1977.
8. R. M. BURSTALL AND J. THATCHER, The algebraic theory of recursive program schemes, *in* "Proceedings, Symposium on Category Theory Applied to Computation and Control," Lecture Notes in Computer Science, Vol. 25, pp. 126–131, Springer-Verlag, Berlin/New York, 1975.

9. S. EILENBERG AND J. B. WRIGHT, Automata in general algebras, *Inform. and Control* 11 (1967), 452–470.
10. C. ELGOT, Algebraic theories and program schemes, *in* "Symposium on Semantics of Algorithmic Languages" (E. Engeler, Ed.), Lecture Notes in Mathematics, Vol. 188, pp. 71–88, Springer-Verlag, Berlin/New York, 1971.
11. C. ELGOT, Assignement statements in the context of algebraic theories, *in* "Proceedings, Third IBM Japan Symposium," 1978.
12. J. GALLIER, Semantics and correctness of nondeterministic flowchart programs with recursive procedures, *in* "Proceedings, Fifth ICALP," 1978.
13. J. A. GOGUEN, On homomorphisms, correctness, termination, unfoldments, and equivalence of flow diagram programs, *J. Comput. System Sci.* **8** (974), 333–365.
14. J. A. GOGUEN, "Set-Theoretic Correctness Proofs," Technical Report, UCLA Computer Science Department., 1974.
15. J. GOGUEN AND J. MESEGUER, Correctness of recursive flow diagram programs, *in* "Proceedings, Mathematical Foundations of Computer Science," Lecture Notes in Computer Science, Springer-Verlag, Vol. 53, pp. 580–595, 1977.
16. J. A. GOGUEN, J. W. THATCHER, AND E. WAGNER, An initial algebra approach to the specification, correctness and implementation of abstract data types, *in* "Current Trends in Programming Methodology" (R. Yeh, Ed.), Prentice–Hall, Englewood Cliffs, N.J., 1978, Original version IBM T. J. Watson Research Center Technical Report RC 6487, October 1976.
17. J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER, AND J. B. WRIGHT, "An Introduction to Categories, Algebraic Theories, and Algebras." Technical Report, IBM T. J. Watson Research Center Research Report RC 5369, Yorktown Heights, New York, 1975.
18. R. GOLDBLATT, "The Categorial Analysis of Logic," North-Holland, Amsterdam, 1979.
19. J. V. GUTTAG, "The Specification and Application to Programming of Abstract Data Types," Ph.D. thesis, Report CSRG-59, Univ. of Toronto, 1975.
20. F. W. LAWVERE, "Functorial semantics of algebraic theories, *in* "proceedings, National Academy of Science," No. 50, 1963; Summary of Ph.D. thesis, Columbia University.
21. S. MACLANE, "Categories for the Working Mathematician," Springer-Verlag, Berlin/New York, 1971.
22. Z. MANNA, The correctness of programs, *J. Comput. System Sci.* 3 (1969), 119–127.
23. Z. MANNA, The correctness of nondeterministic programs, *Artificial Intelligence* 1 (1970), 1–26.
24. Z. MANNA, Mathematical theory of partial correctness, *in* "Symposium on Semantics of Algorithmic Languages" (E. Engeler, Ed.), Lecture Notes in Computer Science, Vol. 188, pp. 252–269, Springer-Verlag, Berlin/New York, 1971.
25. J. MCCARTHY, Towards a mathematical science of computation, *in* "Proceedings, 1962 IFIP Congress," pp. 21–28, North-Holland, Amsterdam, 1962.
26. S. ZILLES, "Abstract Specification of Data Types," Technical Report 119, Computation Structures Group, MIT, Cambridge, Mass., 1974.