JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 543–578 (1973)

Program Schemata as Automata. I

J. D. RUTLEDGE

Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received July 21, 1971

1. INTRODUCTION

The notion of program schema, otherwise known as abstract program, has received substantial attention recently, and seems to be a useful abstraction of the control structure of programs. As introduced by Ianov [3] a program schema is a very automatonlike object, which operates on a tapelike 'evaluation sequence' to produce an 'application sequence' or output; schemata are characterized by the function from evaluation sequences to application sequences which they realize, and equivalence is defined in terms of these functions (S-equivalence). A different equivalence is defined in terms of all possible interpretations; two schemata which under every interpretation applicable to both, produce programs which compute the same (partial) function on data are equivalent in this sense (I-equivalence). For Ianov schemata, the two equivalence relations are the same, and the equivalence problem in the second sense can be solved by observing that schemata can in fact be reduced to finite automata, with S-equivalence corresponding exactly to finite automaton equivalence [10]. All of this depends on the picture of a schema operating at each step on all of memory, 'one and indivisible,' so that operations occur in linear sequence, and any change in that sequence means a change in the function computed by some interpretation. As soon as we allow the existence of two distinct segments of memory, such that a given operator may change one but not the other, the situation changes drastically. Park and Luckham [5, 6] show that the questions of I-equivalence and termination are unsolvable for these schemata in general, although Paterson [9] and Manna [7, 8] have pointed out various subclasses of schemata for which these problems are solvable. No generalization of S-equivalence has been given for partitioned-memory or multiple register schemata. Since, to be of any real interest, such an equivalence should include pairs of schemata in which a pair of noninteracting operations occur in different orders, it is immediate that the generalized evaluation sequence cannot be linearly ordered, nor, equally, can the generalized application sequence or output. While the quite restricted and conventional schemata of [6] serve nicely for counter examples to prove

their primarily negative results, it appears that a more natural context for a study of equivalence of multiregister schemata is one in which parallelism is permitted, and events occur subject to less than total sequence constraints. This paper develops a notion of schema and computation which generalizes Ianov's automatonlike S-equivalence in a multiregister context.

The formalism is based on a notion of labeled graph-cum-terminals (c-graphs), which is defined and developed in the third section. Program computation is then treated formally, introducing an 'output' which records the operations performed in their essential, not actual, (partial) sequence. The formal abstraction of program to schema poses the problem of partially sequenced evaluations, and this is investigated in the context of finite automata generalized to operate on labeled c-graphs in place of linear tapes. Decision methods are developed here, and applied to two schema forms which define two different notions of S-equivalence, bracketing I-equivalence. In a preview of part II, it is pointed out that conventional program and schemata relate naturally to linear-sequence computation, but not to the partial-sequence computation natural to the multiregister situation, and a more appropriate form of program and schema is promised.

Much of the material of this paper is contained in [11], although largely in a different formalism and from a somewhat different point of view.

2. Programs: Informally

To begin, we consider a familiar fairly general class of programs, which we will call control sequenced (c.s.) programs, and which we define in terms of their flowcharts. A c.s. program is specified by (1) a domain Q of objects on which it operates, (2) a set A of operators over Q, each with a given number of arguments, (3) a finite set of 'location names' M, which might represent storage locations or devices, (4) a set F of 'branch functions,' which may be thought of as choosing for each member (or pair, triple, etc., of members) of Q, a successor direction, say left or right, or more generally an integer, naming the branch to be followed, and (5) a 'flowchart' showing how the other components are to be selected and combined in a computation. This 'flowchart' is not quite a labeled directed graph in the usual sense, because (1) it has distinguished nodes, one (or more) for entry and for exit, and (2) certain of the nodes have several leaving edges which are distinguished. Entering edges, on the other hand, are not distinguished. We will shortly formalize this notion in terms of the class of c-graphs, but for the moment the informal notion will serve. Each (nonterminal) node of the flowchart is labeled with an operator, an appropriate number of argument locations from M, a result location from M,[1] and a branch function, taken as having the same

[1] The restriction to a single result location is purely to simplify notation. It would be natural to allow many result locations, but the extension is easily made, and the notational saving seems worthwhile for this exposition.

arguments as the operator. The operator or the branch function or both may, of course, be trivial. There is also a specification of input and output locations, which may be taken as ordered subsets of M labeling the entry and exit terminals. This notion of program is a very familiar one, restricted notably in the requirement of a finite set of named locations. Since these may contain such things as arrays of unbounded dimension, the total storage capacity is not restricted, only the number of names through which it is accessed.

Equally familiar is the process by which such a program defines a (partial) function from the pairs, entry and n-tuple on Q, to the pairs, exit and m-tuple on Q, where the input set for the entry has n members and the output set has m. This process produces a sequence of operators, and a sequence of mappings k: $M \rightarrow Q$, the storage contents or assignments; the first 'operator' is the entry node name, the first assignment maps each input location onto the corresponding value, and the final members of the sequence correspondingly give the exit and output values. All this is perfectly conventional. Note quite so conventional, but hardly new, is the observation that these sequences contain a good bit more order than is really required. In particular, our sequence of assignments seems to say that each quantity existing at step i of the computation follows every quantity at step $i - 1$. This is true in the physical time of a real computation, but in terms of dependency or logical sequence, it certainly is not true; each quantity logically succeeds only those quantities which contributed to its formation. If we represent the 'storage history' in this form, showing only logical sequence, we find that the operator sequence, too, can be altered without changing the essentials of the computation—in fact we can simply associate operators with their outputs in the storage history to represent the entire computation. At this point it is clear that many different programs, inequivalent under Ianov equivalence, can represent not only the same function from input to output, but the same function from input to computation. This latter sort of similarity is associated with schema equivalence, and it is in this context that we shall study it.

3. c-Graphs

We must obviously become much more precise than the preceding, rather vague dicussion, which is intended only to give a very general overview and motativation. The question arises, what sort of formalism is appropriate to the subject. The notation of set theory is certainly powerful enough, but becomes excessively complex (see [11]); category theory has been used for similar work [2], but in some respects even the notion of morphism is too restricted for our purposes. The principal properties we will require are (a) the capability of representing the kind of partially-ordered 'computation' described briefly above, where multiple predecessors are distinguished and (b) the capability of representing, in the same notation, the structure of generalized

'flowcharts' in which both predecessors and successors may be distinguished. We will want to carry out analogs of the standard automata theory arguments, including cut-and-splice, on 'computations' and 'flowcharts,' with as much facility as we have ordinarily in working with sequences. After considerable experimentation with other approaches, including those mentioned above, it seems worthwhile to introduce an apparently new class of mathematical objects, the *c-graphs*, with four fundamental and several derived operations on them, and to develop certain of the elementary algebraic properties of the resulting system which are required in the remainder of the paper. Since the theorems of this section are used repeatedly and the details are sometimes critical, we indicate some detail of the proofs. The reader may, if he likes, treat this section as an appendix, referring to it when he wishes to verify some property or identify an operation; it is recommended that at least the basic structures and operations be understood at a first reading.

We will initially define the class of c-graphs as a set-theoretic object (actually, something like an algebraic species) and define the four fundamental operations: double composition, deletion, dot deletion, and 'E' in similar terms; the remaining operations are defined in terms of these, together with several classes of special c-graphs which function as units and zeros. The final part of the section points out how an association of operator symbols with nodes may be naturally extended to an association of functions with graphs, such that graph-composition corresponds to general function composition. This will be used to associate values with the computations defined later, and may also be used to give a less computational notion of the function (or relation) to be associated with a program, although we will not pursue this line here (see [2]).

Definitions

We define a c-graph G as follows: Let

N be a set of distinct objects, the nodes

$\alpha \subset (N \cup \{N\}) \times J$

$\omega \subset (N \cup \{N\}) \times J$, where J is an index set, for example the integers. These are entry and exit terminals, respectively,

$R \subseteq \omega \times \alpha$ be the set of edges; its members are ordered pairs of the form

$\langle\langle n_1 , j_1\rangle, \langle n_2 , j_2\rangle\rangle$, where $\langle n_1 , j_1\rangle$ is the j_1 exit of node n_1, and $\langle n_2 , j_2\rangle$ is the j_2 entry of node n_2.

Then $G = \langle N; \alpha, \omega, R\rangle$.

EXAMPLE 1.[2]

[2] We use the notation $[n] = \{1, 2,..., n\}$ for n a nonnegative integer.

$N = \{a, b, c\}$

$$\alpha(a) = [2] \qquad \omega(a) = [1]$$
$$\alpha(b) = [2] \qquad \omega(b) = [2]$$
$$\alpha(c) = [3] \qquad \omega(c) = [2]$$
$$\alpha(N) = [2] \qquad \omega(N) = [3]$$

$$R = N_1, a_1 \qquad a_1, c_1 \qquad c_1, a_1$$
$$N_2, c_2 \qquad b_1, c_3 \qquad c_1, c_2$$
$$N_2, a_2 \qquad b_1, b_2 \qquad c_2, N_1$$
$$N_3, b_1 \qquad b_2, N_2$$

This can be simply drawn as in Fig. 1. We have just a directed graph, with the

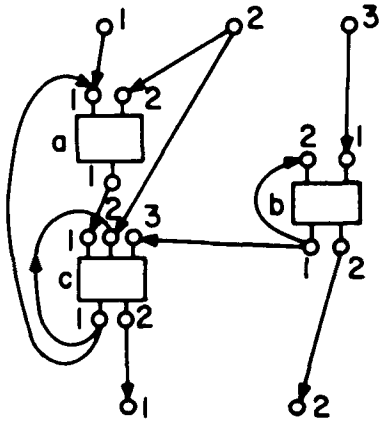


FIG. 1. G_1.

addition of entry and exit terminals on each node, and also entry and exit terminals for the graph as a whole. As a notational convenience we will use the notation $\bar{\alpha}(a) = \alpha \cap (\{a\} \times J)$ for $a \in N \cup \{N\}$, and similarly for ω. The set of entry terminals of the graph, then is just $\bar{\omega}(N)$.

We have here a slight notational anomaly, since from the inside of the graph (our present point of view) the graph entry terminals are the initial points of edges, hence exits (of something preceding); in the context of a 'higher level' graph $\mathbf{G} = \langle \mathbf{N}; \boldsymbol{\alpha}, \boldsymbol{\omega}, \mathbf{r} \rangle$ in which G is a single node ($G \in \mathbf{N}$), this set is $\bar{\boldsymbol{\alpha}}(G)$, while $\bar{\alpha}(N) = \bar{\omega}(G)$. If one keeps this difference in viewpoint in mind, the apparent anomaly disappears.

We can map a c-graph into an ordinary directed graph over the set

$$N \cup (\alpha \times \{i\}) \cup (\omega \times \{\sigma\});$$

that is just the set of all nodes and terminals. The edges are the edges of G with the addition of an edge from each node to each of its exit terminals and an edge from each entry terminal to its node. Call this the *derived digraph* of G; several properties of directed graphs carry over directly.

A c-graph will be called (*connected*) (*acyclic*) if its derived digraph is (connected) (acyclic).

A *path* in a c-graph is just a path in its derived digraph.

A node a is *accessible* from a node b iff a path beginning at a and ending at b exists in the derived digraph.

Continuing the analogy with standard graphs, we define a *labeled c-graph* as the structure $\langle G, l\rangle$, where G is a c-graph $\langle N; \alpha, \omega, R\rangle$, L is an arbitrary set of labels and

$$l: N \cup \alpha(N) \times \{i\} \cup \omega(N) \times \{\sigma\} \to L.$$

Often the set of labels assigned to the graph terminals is disjoint from that assigned to the nodes or empty; this will be indicated where necessary.

The following special graphs will be useful.

DEFINITION. For $\phi : S \to T$, where $S, T \subseteq J$, U_ϕ is given by:

$$\langle \varnothing; \{\varnothing\} \times T, \{\varnothing\} \times S, \{\langle\langle \varnothing, j\rangle\langle \varnothing, \phi(j)\rangle\rangle \mid j \in S\}\rangle.$$

For ϕ the identity map $S \to S$, define $1_S = U_\phi$

EXAMPLE 2. $S = \{1, 2, 3\}$, $T = \{x, y, z\}$, $\phi = \{\langle 1, x\rangle, \langle 2, y\rangle\langle 3, z\rangle\}$, U_ϕ is shown in Fig. 2.

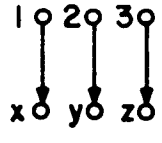


FIG. 2. U_φ for $\varphi = \langle 1, x\rangle, \langle 2, y\rangle, \langle 3, z\rangle$.

DEFINITION. For $S, T \subseteq J$, $Z_{S,T} = \langle \varnothing; \{\varnothing\} \times T, \{\varnothing\} \times S, \varnothing\rangle$.

EXAMPLE 3. $S = \{2, 3\}$, $T = \{x\}$, $Z_{S,T}$ is shown in Fig. 3.

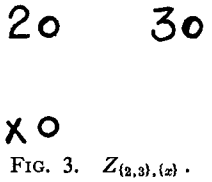


FIG. 3. $Z_{\{2,3\},\{x\}}$.

Since c-graphs combine aspects of relations and of sets, in dealing with them we need operations corresponding to both set union and difference and to relational composition. It turns out to be possible to define these in terms of two rather setlike fundamental operations, deletion and composition, with the aid of 'renaming' unit c-graphs, which are used to prevent undesired index-coincidences.

For graphs $G_i = \langle N_i ; \alpha_i , \omega_i , R_i \rangle$, let $a_i = \omega_i(N_i)$, $w_i = \alpha_i(N_i)$, that is the set of graph entry and exit terminal indices, respectively. When notationally convenient, we will also use $a(G_i) = \omega_i(N_i)$, $w(G_i) = \alpha_i(N_i)$. We can think of a as the entry terminal relation of the 'higher level' graph referred to above.

We define *double composition* of two graphs G_1, G_2, $G_i = \langle N_i ; \alpha_i , \omega_i , R_i \rangle$, as

$$
\begin{aligned}
&G = G_1 \overset{\circ}{\circ} G_2 , \text{ where} \\
&N = N_1 \cup N_2 \\
&\alpha = \alpha_1 \mid N_1 \cup \alpha_2 \mid N_2 \cup \{N\} \times ((w_1 - a_2) \cup (w_2 - a_1)) \\
&\omega = \omega_1 \mid N_1 \cup \omega_2 \mid N_2 \cup \{N\} \times ((a_1 - w_2) \cup (a_2 - w_1)) \\
&R = \overline{w}_2 \mid R_1' \mid \bar{a}_2 \cup \overline{w}_1 \mid R_2' \mid \bar{a}_1 \cup \overline{w}_2 \mid R_1' \mid w_1 \circ a_2 \mid R_2' \mid \bar{a}_1 \cup \overline{w}_1 \mid R_2' \mid w_2 \circ a_1 \mid R_1' \mid \bar{a}_2 \\
&\qquad \cup \overline{w}_2 \mid R_1' \mid w_1 \circ a_2 \mid R_2' \mid w_2 \circ a_1 \mid R_1' \mid \bar{a}_2 \cup \overline{w}_1 \mid R_2' \mid w_2 \circ a_1 \mid R_1' \mid w_1 \circ a_2 \mid R_2' \mid \bar{a}_1 \cup \cdots
\end{aligned}
$$

where $\overline{w}_2 \mid R_1' \mid \bar{a}_2 = R_1' \cap (\overline{\{N\} \times w_2} \times \overline{\{N\} \times a_2})$, and where R_i' is obtained from R_i by replacing N_i by N in all occurrences.

For an example, see Fig. 4. This definition being completely symmetric in 1 and 2, the operation commutes; however, it is associative only in special cases.

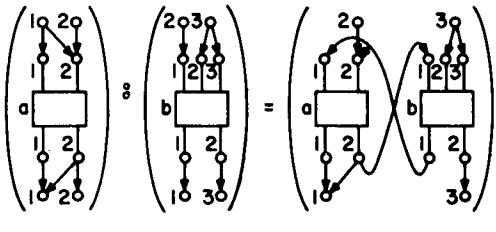


FIG. 4. Double composition.

In the case $a_1 \cap w_2 = \varnothing$, we get simple composition, $G_1 \circ G_2 = G_1 \overset{\circ}{\circ} G_2$. In the general case, while we can define *simple composition* in terms of double by use of renamings, it is more convenient to give the result directly:

$$
\begin{aligned}
&G = G_1 \circ G_2 , \text{ where} \\
&N = N_1 \cup N_2 \\
&\alpha = \alpha_1 \mid N_1 \cup \alpha_2 \mid N_2 \cup \{N\} \times (w_2 \cup (w_1 - a_2)) \\
&\omega = \omega_1 \mid N_1 \cup \omega_2 \mid N_2 \cup \{N\} \times (a_1 \cup (a_2 - w_1)) \\
&R = R_1' \mid \bar{a}_2 \cup \overline{w}_1 \mid R_2' \cup R_1' \mid w_1 \circ a_2 \mid R_2' .
\end{aligned}
$$

See example, Fig. 5.

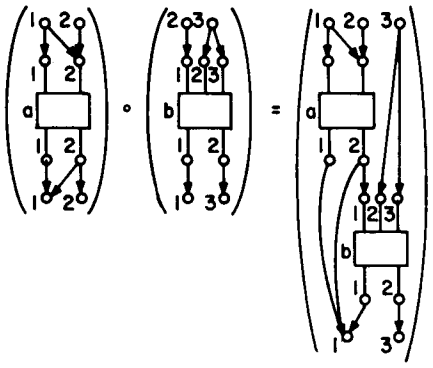


FIG. 5. Simple composition.

For a c-graph G_1 and set N_2, the *N_2-deletion of G_1*, $G = G_1 - N_2$, is given by

$$N = N_1 - N_2,$$
$$\alpha = \alpha_1 \mid N_1 \cup \{N\} \times [(R_1(\omega_1 \mid N))(N_1) \cup (((\omega_1 \mid N_1) \times (\alpha_1 \mid N_2)) \cap R_1)],$$
$$\omega = \omega_1 \mid N_1 \cup \{N\} \times [(R_1^{-1}(\alpha_1 \mid N))(N_1) \cup (((\omega_1 \mid N_2) \times (\alpha_1 \mid N_1)) \cap R_1)],$$
$$R = (R_1' \cup \phi_\alpha(R_1) \cup \phi_\omega(R_1)) \cap (\omega \times \alpha),$$

where $\phi_\alpha\langle x, y\rangle = \langle x, \langle N, \langle x, y\rangle\rangle\rangle$ and $\phi_\omega\langle x, y\rangle = \langle\langle N, \langle x, y\rangle\rangle, y\rangle$. Examples of deletion are given in Fig. 6. Intuitively, one can think of this as an operation which excises the node set $N_1 \cap N_2$ from N_1, replacing each edge which is cut in the process by a terminal carrying the name of the edge which was cut. This has the pleasant property that if $N = N_1 \cup N_2$, $N_1 \cap N_2 = \varnothing$ and graph terminal sets are appropri-

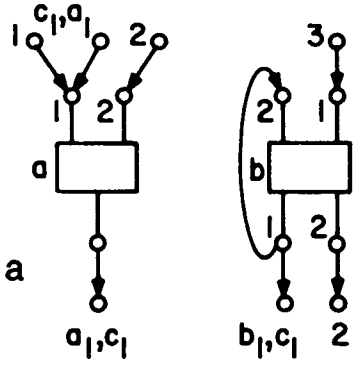


FIG. 6a. $G_1 - \{c\}$.

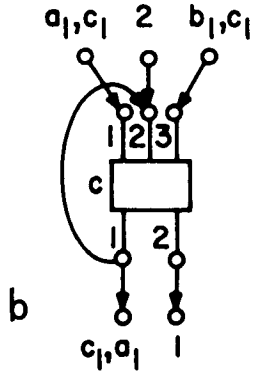


FIG. 6b. $G_1 - \{a, b\}$.

ately disjoint, then $(G - N_1) \circ\!\!\circ (G - N_2) = G$. As before, by a bit of attention to avoiding terminal conflicts, we can define another interesting operation, that of *substitution*. For c-graphs G_1, G_2, and set N_3, $G = \int_{G_2}^{N_3} G_1$, read G is the result of substituting G_2 at N_3 in G_1, $G = U_\phi \circ\!\!\circ ((U_\phi \circ\!\!\circ G_1) - N_3) \circ\!\!\circ G_2)$, where S is disjoint from and equicardinal with $a_1 \cup w_1$ and disjoint from $a_2 \cup w_2$, and ϕ: $S \overset{1\text{-}1}{\longleftrightarrow} (a_1 \cup w_1)$. Provided the graph terminals of G_1 do not have the rather odd names of the 'fresh cut' terminals of $G_1 - N_3$, this can be simplified to $(G_1 - N_3) \circ\!\!\circ G_2$. See example Fig. 7.

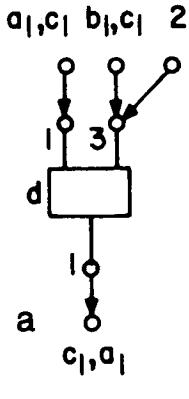


FIG. 7a. G_2.

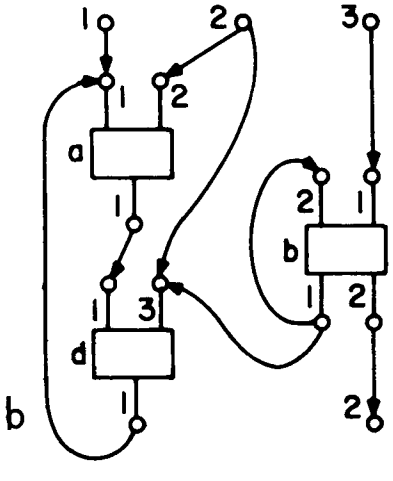


FIG. 7b. $\int_{G_2}^{\{c\}} G_1$.

Where convenient, we will use the name of a graph in place of the name of its node set with these operators, e.g., $G_1 - G_2 \equiv G_1 - N_2$.

We need also a distinct deletion operation, called *dot deletion*, making the cut at the node terminals rather than on the edges connecting the terminals to the rest of the graph. Let $G = G_1 \dot{-} n$, where

$$N = N_1 - \{n\},$$
$$\alpha = \alpha_1 \mid N \cup \{N\} \times [w_1 \cup \alpha_1(n)],$$
$$\omega = \omega_1 \mid N \cup \{N\} \times [a_1 \cup \omega_1(n)],$$
$$R = \zeta(R) \cap (\omega \times \alpha),$$

and where ζ replaces all occurrences of either N_1 or n by N. Based on this deletion, we define *dot substitution* by

$$\cdot \int_{G_2}^{n} G_1 = U_{\phi_1^{-1}} \circ (((U_{\phi_1} \circ G_1 \circ U_{\phi_2}) \dot{-} n) \circ\!\!\circ G_2) \circ U_{\phi_2^{-1}}.$$

where S, $T \subset J$ are disjoint from a_1, w_1, a_2, w_2, $\alpha_1(n)$, $\omega_1(n)$, and $\phi_2 : w_1 \to S$, $\phi_1 : T \to a_1$. Note that while in substitution and deletion all components are of uniform type, the dot-operations are mixed-type operations. See example, Fig. 8.

The property of deletion mentioned following its definition raises a question about the notion of subgraph. On the one hand, one would like to retain the term for an algebraic subsystem, i.e., G_1 *subgraph* of G iff $N_1 \subseteq N$, $\alpha_1 \subseteq \alpha$, $\omega_1 \subseteq \omega$, and $R_1 \subseteq R$. On the other hand, we cannot compose such subgraphs to give the original graph, so the notion of applying the term to the deletion of a graph is attractive. This is one of

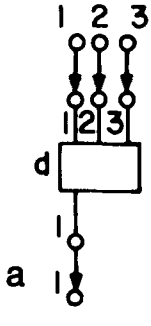


FIG. 8a. G_2.

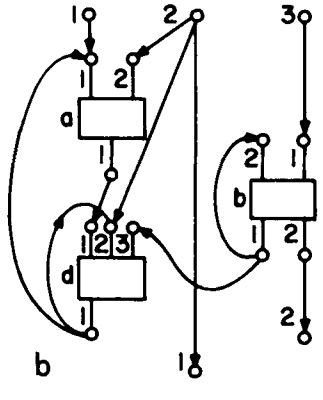


FIG. 8b. $\cdot \int_{G_2}^{c} G_1$.

the points where the setlike and relationlike properties of these objects lead in different directions. We will retain the term subgraph in its standard meaning, as above, but often when a portion of a graph is needed it is a deletion, not a subgraph which is required.

A case in point is *intersection.* We define $G_3 = G_1 \cap G_2$ for the case G_1, G_2 both subgraphs of G, as $G_1 - (N_1 - N_2)$. In case G_1 and G_2 are not identical on their common node set, the operation is (for the present) undefined. The natural intersection, analogous to the natural subset relation, leaves no way of recombining fragments so produced to rebuild the original graph, and so is unsatisfactory.

One additional operation, which we will call parallel composition, we define as:

$$G = G_1 * G_2 = U_{\phi_2^{-1}} \circ (G_1 \circ U_{\phi_1}) \circ (U_{\phi_2} \circ G_2) \circ U_{\phi_1^{-1}},$$

where $\phi_1 : w_1 \overset{1-1}{\to} S$, $\phi_2 : T \overset{1-1}{\to} a_2$, and S, T are disjoint from a_1, w_1, a_2, w_2 and from each other. This operation has the effect of merging like-named entry terminals of G_1 and G_2, and similarly for exit terminals, e.g., $w = w_1 \cup w_2$, $a = a_1 \cup a_2$. This is clearly associative and commutative. See example, Fig. 9. We will occasionally use

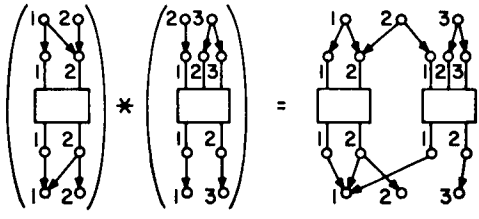


FIG. 9. Parallel composition.

the notation $\sum_{i\in S}^{*} E_i$ to indicate the parallel composition of all members of a set of graphs.

The reader will have noted that we have a strong suggestion of a hierarchical structure, in that a c-graph has the same form as a single node together with its terminals. In these terms, composition as defined above preserves level in the hierarchy, so the composition of two nodes-cum-terminals will be a c-graph having those nodes in its node set, but without interconnections ($R = \varnothing$). To compose primitive nodes we need a 'jump' operator, analogous to the $\{\ \}$ of set theory.

DEFINITION. Let $\langle n, a, w, v\rangle = N$ be a node with entry terminals a, exit terminals w and label v. The *elementary graph* of N is

$$E(N) = \langle\langle\{n\}; \{n\} \times a \cup \{\{n\}\} \times w, \{n\} \times w \cup \{\{n\}\} \times a,$$
$$\{\langle\langle x, y\rangle\langle z, y\rangle\rangle \mid \langle x, y\rangle \in \omega \text{ and } \langle z, y\rangle \in \alpha \text{ and } x \neq z\}\rangle, \{\langle n, v\rangle\}\rangle.$$

This is just the node n with like-indexed graph entry (exit) terminals. See Fig. 10.

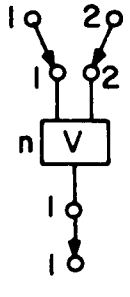


FIG. 10. $E(n, [2], [1], v)$.

It is worth noting that with the operation of double composition alone, we can construct an arbitrary c-graph starting from a set of elementary graphs which includes at least the graphs U_ϕ for $|\phi| = 1$, since double composition with appropriate

members of U_ϕ can be used both for renaming terminals and for making arbitrary connections.

By $G^1 \cong^\phi G_2$, we will mean the usual algebraic *isomorphism*, but with J fixed; i.e., $\phi: N_1 \xrightarrow{1-1} N_2$, $\phi: J \to J$ by identity, and ϕ commutes with α, ω, and R.

Algebraic Properties

Under (simple) composition, these graphs form an interesting algebraic system. If we restrict the operation $G_1 \circ G_2$ to graphs satisfying $\alpha_1(N_1) = \omega_2(N_2)$, we obtain a category over the subsets of J, with 'unit' elements 1_s. In this connection, see [2]. Note also that the empty graph is an identity for $\overset{\circ}{\circ}$ and $\circ$. Without the above restriction, we may lose associativity, but the full system is still of interest, and will prove to be quite useful.

While the algebraic properties of general c-graphs are weak, the following lemmas indicate the conditions for associativity and commutativity under simple composition.

LEMMA 1. $w_1 \cap a_2 \cap (a_3 - w_2) = \varnothing = a_3 \cap w_2 \cap (w_1 - a_2) \Rightarrow G_{1(23)} = G_1 \circ (G_2 \circ G_3) = (G_1 \circ G_2) \circ G_3 = G_{(12)3}$.

Proof. From the definition of $\circ$ and associativity of union, $N_{1(23)} = N_{(12)3}$. By straightforward Boolean manipulation on the definitions of α and ω, we find necessary and sufficient conditions for $\alpha_{(12)3} = \alpha_{1(23)}$ and $\omega_{(12)3} = \omega_{1(23)}$ to be, respectively,

$$w_2(w_1 - a_2) \subseteq w_3 \cup (w_2 - a_3) \cup (w_1 - a_3), \tag{1}$$

$$a_2(a_3 - w_2) \subseteq a_1 \cup (a_2 - w_1) \cup (a_3 - w_1). \tag{2}$$

To help keep some intuitive hold on the problem, consider the diagram of Fig. 11.

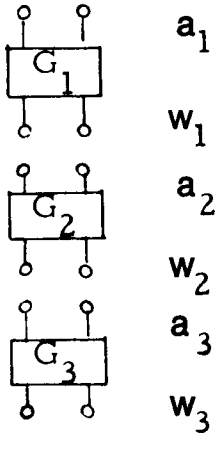


FIGURE 11

Terminals which match across the 1–2 interface disappear in $G_1 \circ G_2$, while those which do not, 'float' up or down to become part of a_{12} or w_{12}. If a terminal say $\langle N_1, j\rangle$ floating from w_1 has the same index as a member of w_2, thus is in w_{12}, it will be identified with it, and R_{12} will include a term $\langle x, \langle N_{12}, j\rangle\rangle$ derived from the term $\langle x, \langle N_1, j\rangle\rangle$ of R_1. If now $j \in a_3$, this terminal will be absorbed there in the construction of $G_{12} \circ G_3$, and unless j is also present in w_3, no j will be present in $w_{(12)3}$. On the other hand, if $j \in a_3 \cap w_2$, the j-terminal in w_1 will be unmatched in the formation of $G_{1(23)}$, and will thus appear in $w_{1(23)}$. Clearly the presence of j in w_3, while it does provide a j in $w_{(12)3}$, cannot provide the required term $\langle x, \langle N_{(12)3}, j\rangle\rangle$. If we drop the term w_3 in (1) above, we obtain

$$w_2 w_1 \bar{a}_2 \subseteq w_2 \bar{a}_3 \cup w_1 \bar{a}_3,$$

or

$$(w_1 w_2 \bar{a}_2)\overline{(w_2 \bar{a}_3 \cup w_1 \bar{a}_3)} = \varnothing$$

$$w_1 w_2 \bar{a}_2 a_3 = \varnothing,$$

which is just one of the required conditions; the other is obtained mutatis mutandis. This establishes the necessity of the condition in the sense that for any a_i, w_i not satisfying it nonassociative triples of graphs may be constructed, as well as its sufficiency for $\alpha_{(12)3} = \alpha_{1(23)}$ and $\omega_{(12)3} = \omega_{1(23)}$.

To see $R_{(12)3} = R_{1(23)}$, suppose that $\langle x, y\rangle \in R_{(12)3}$, $\langle x, y\rangle \notin R_{1(23)}$. By the above $x \in \omega_{1(23)}$ and $y \in \alpha_{1(23)}$; by the construction of $R_{(12)3}$ and $R_{1(23)}$,

$$\langle x, y\rangle \notin R_i \mid (N_i \times J) \times (N_i \times J), \qquad i = 1, 2, 3,$$

else it would necessarily appear in $R_{1(23)}$. Thus we must have one of the cases

$x \in a_1$	$\omega_1 \mid$		$a_2 - w_1$	$\omega_2 \mid$		$a_3 - w_2$	$\omega_3 \mid$	
$y \in$	$\alpha_1 \mid$	$w_1 - a_2$		$\alpha_2 \mid$	$w_2 - a_3$		$\alpha_3 \mid$	w_3

where $\alpha_i \mid$ stands for $\alpha_i \mid N_i$, the terms t in a_i and w_i stand for $N_{123} \times t$ and each x may be paired with any y below or to the right of it. Note that we have used our hypotheses in constructing this table. We will detail two typical cases.

First, let $x \in \omega_1 \mid$, $y \in N_{123} \times (w_2 - a_3)$. Since this is internal to G_{12}, except that the terminal y has 'floated' to $w_{(12)3}$, there must have been a term $\langle x, y\rangle \in R_{12}$, thus terms $\langle x, \langle N_1, z\rangle\rangle$ and $\langle\langle N_2, z\rangle, y\rangle$ in R_1 and R_2 respectively, or else $\pi_2 y \in w_1 - a_2$. In the first case R_{23} must contain $\langle\langle N_{23}, z\rangle, y\rangle$, so $R_{1(23)}$ contains $\langle x, y\rangle$. In the second case, since $\pi_2 y$ is in $\bar{a}_2 \bar{a}_3$, it will 'float' past R_{23} and yield the term $\langle x, y\rangle \in R_{1(23)}$ directly. Q.E.D.

For an extreme case, let $x \in a_1$, $y \in w_3$. This can only arise from terms $\langle x, z\rangle \in R_{12}$

and $\langle z, y\rangle \in R_3$, or else $\langle x, z\rangle \in R_{12}$ and $\pi_2 z = \pi_2 y$. $\langle x, z\rangle$, in turn, must have come either from $\langle x, v\rangle \in R_1$ and $\langle v, z\rangle$ in R_2 or else $\langle x, z\rangle \in R_1$ and $z \in w_1 - a_2$. In the first case, $\langle v, y\rangle$ must appear in R_{23}, thus $\langle x, y\rangle$ in $R_{1(23)}$; in the second, the condition gives us $z \notin a_3 w_2$, but $z \in a_3$ by the case hypothesis, so $z \in a_3 - w_2$ and thus $z \in a_{23}$ and $\langle z, y\rangle \in R_{23}$, hence $\langle x, y\rangle \in R_{1(23)}$. For the final subcase $\langle x, z\rangle \in R_{12}$, $\pi_2 z = \pi_2 y$, $z \in (w_2 - a_3)\, w_3$, essentially the same argument serves. The remaining 22 cases are left to the reader.

COROLLARY . $w_1 = a_2$ and $w_2 = a_3 \Rightarrow G_1 \circ G_2 \circ G_3$ *is associative.*

COROLLARY. $w_2 = a_2 \Rightarrow G_1 \circ G_2 \circ G_3$ *is associative.*

c-graphs do not of course commute in general. On the other hand, many pairs of *c*-graphs do commute; any pair of identities, for example, or any pair with disjoint sets of entry and exit terminal indices. To obtain a better criterion for commutativity, we observe the following.

LEMMA 2. $G \circ 1_s = 1_s \circ G \Leftrightarrow s \cap (a - w) = s \cap (w - a) = \varnothing$.

Proof. From the definition of composition, using subscripts 12 and 21 for the two orders of composition, we have as necessary and sufficient conditions:

$$
\begin{aligned}
w_{12} &= s \cup (w - s) = w_{21} = w \cup (s - a),\\
a_{12} &= a \cup (s - w) = a_{21} = s \cup (a - s),\\
R_{12} &= R \cup \operatorname{diag}((s - w) \times (s - w)),\\
R_{21} &= R \cup \operatorname{diag}((s - a) \times (s - a)).
\end{aligned}
$$

By Boolean manipulation again,

$$
\begin{aligned}
& s \cup w\bar{s} = w \cup s\bar{a},\\
& (s \cup w) \cap (s \cup \bar{s}) \overset{?}{=} (w \cup s) \cap (w \cup \bar{a}),\\
& \text{iff } w \cup s \subseteq w \cup \bar{a},\\
& \text{iff } (w \cup s) \cap \overline{(w \cup \bar{a})} = \varnothing,\\
& \cdot (w \cup s)\, \bar{w}a = \varnothing\\
& \cdot s(a - w) = \varnothing. \qquad \text{Q.E.D.}
\end{aligned}
$$

Similarly, $a_{12} = a_{21}$ is equivalent to the other condition, and $(s - a) = (s - w)$ is equivalent to the conjunction of both conditions.

COROLLARY. $a = w \Rightarrow G \circ 1_s = 1_s \circ G$.

LEMMA 3. $a_1 \cap w_2 = a_2 \cap w_1 = \varnothing \Rightarrow G_1 \circ G_2 = G_2 \circ G_1$.

Proof. From the definition, $a_{12} = a_{21} = a_1 \cup a_2$, $w_{12} = w_{21} = w_1 \cup w_2$, and $R_{12} = R_{21} = R_1' \cup R_2'$ since ψ in either order is empty. Of course, this permits $a_1 \cap a_2$ and $w_1 \cap w_2$ to be nonempty.

Having noted that nontrivial graphs may commute, albeit in a somewhat trivial situation, we look at a decomposition which we will find useful. We define an *identity-free* graph to be one in which no graph entry (exit) terminal is R-related just to the similarly indexed exit (entry) terminal.

PROPOSITION. *Every c-graph has a unique decomposition of form* $G = 1_S \circ F \circ 1_T$, *where* F *is identity-free.*

Proof. Let S be the subset of a for which the above condition is violated, $S = \{x \mid \langle N, x\rangle \in \omega \wedge R(\langle N, x\rangle) = \{\langle N, x\rangle\}\}$, and T be the corresponding subset of w; F is what remains when these terminals and their connections are removed. Note that by construction $S \cap a_F = T \cap w_F = \varnothing$, so the conditions for associativity are satisfied.

We can now look at commutativity of two graphs in terms of commutativity of their identity-free parts with each other and with the various identities. For example, if the identity-free parts satisfy $a_{F_i} = w_{F_i}$, then G_1, G_2 commute iff F_1, F_2 do.

LEMMA 4. $1_T \circ G \circ 1_S$ *is always associative.*

Proof. The possible failure of associativity is an exit terminal of 1_T which 'floats' past $G \circ 1_S$, but is present in S, so is caught in $(1_T \circ G) \circ 1_S$. But this is simply connected to the exit terminal via R_{1_S}, so associativity is maintained. The same argument applies to entry terminals of 1_S floating to 1_T.

Evaluation

If we view the label of a node n as an operator symbol, it is natural to associate with the labeled node the function which maps an $|\alpha(n)|$-tuple of terms associated with the entry terminals into a term associated with each of the exit terminals. If the entry terminals have associated sets of terms, the natural extension of this function produces a set of terms for each exit terminal. For $\langle x, y\rangle \in R$, we require that the set of terms associated with y contain that associated with x; if we now associate with the graph entry terminals their respective indices as variables, a minimum set of terms satisfying the above conditions is well defined for each terminal of the graph. These sets will be finite for acyclic graphs, but may be infinite in general [2]. In particular, sets of terms are assigned to the exit terminals, so the graph may be viewed as defining a function from its set of entry terminals as variables to the sets of pairs ⟨exit terminal, set of terms⟩. We will call this the *evaluation function* of the graph, and denote it by

$V(G)$. In the case of an acyclic graph with R^{-1} a function, the sets associated with terminals are always singletons, and the graph defines a set of terms indexed by exit terminals, in its entry terminal indices as variables and node labels as operator symbols. If the node labels are operators, and the graph entry terminals are indexed by values in the appropriate domain, then the terms associated with the exit terminals are also values, so we can think of such a graph as representing an indexed set of functions on the set of $|a(G)|$-tuples over the domain.

Let $w_1 = a_2 = [k]$. Then

$$V(G_1 \circ G_2) = (V(G_2))(V_1(G_1), V_2(G_1),..., V_k(G_1))$$

where $V_i(G_1)$ is the set of terms at exit i of G_1 under V. This composition works equally smoothly in the general case of simple composition, but conventional functional notation does not allow the corresponding proposition to be easily stated there.

We note in passing that simple composition preserves the properties of acyclicity and R^{-1} functional, so the composition of two functional graphs is functional.

An operation analogous to substitution for variables is pre-composition with a unit graph $U_{\phi^{-1}}$, $\phi: a(G) \to X$. This corresponds to the substitution, according to ϕ, of a member of X for each graph entry terminal index. Using the notation $\int_b^x f$ for the result of substituting b for a free variable x in a formula f, we have $V(U_\phi \circ G) = \int_{\phi(j_1)\phi(j_2)...}^{j_1 j_2 \cdots} V(G)$, substituting for all members of $a(G)$.

The two subsystems of our system of c-graphs which are obtained by: (1) restricting composition to matching graphs, i.e., $G_1 \circ G_2$ defined iff $w_1 = a_2$, and 2) by the restriction $a = w$, are clearly both of special interest, especially the first, which fits neatly into the framework of category theory. We can probably fit most of the system into that framework by allowing rather free composition with units to achieve matching following Lawvere [4]. This, as well as many other aspects of the system, remains open for investigation. For our present purposes it seems preferable to use the more general system, preserving the inherent property of nonlinear ordering of computation which we are studying.

4. Formal Programs and Schemata

Returning now to our initial idea of program, we extract the notion of an interpreted operator set, comprising the first four components, and a basic control structure, consisting of the remaining one. The 'flowchart' is now seen as a labeled c-graph $\langle N; \alpha, \omega, r\rangle$ subject to the conditions that (1) for all $a \in N$, $|\alpha(a)| = 1$, and (2) r is a function on ω. The label function has the form $l: N \to A \times F \times M^* \times M$, subject to the restrictions: if $l(n) = (a, f, m_1, ..., m_k, m)$, then

$$i(f) = i(a) = k, \qquad |\omega(n)| = \sigma(f),$$

where $i(a)$ is the number of inputs required by a, and $\sigma(f)$ is the number of values taken by f.

$(N, j) \in \omega \Rightarrow l\langle(N, j), i\rangle = I_j \in M^*$, the input locations for the j-th entry

$(N, j) \in \alpha \Rightarrow l\langle(N, j), \sigma\rangle = \theta_j \in M^*$, the output locations for the j-th exit.

Program Computation

For a computation of such a program, the conventional definition uses a sequence of assignments (functions $k: M \to Q$) and instructions (node labels), or something equivalent. However, we wish to record only the essential sequence, so in addition to these elements, which will appear in the induction, we will construct an object, called the *output*, which will be labeled c-graph D, constructed as a composition of elementary graphs.

An operator symbol a with k arguments corresponds naturally to a node $\mathbf{a} = \langle\lambda, [k], [1], a\rangle$, where λ is a free node name. An occurrence of this operator in a program has specified argument and result locations, given as part of the label of the node, $l(n) = a, f, m_1, \ldots, m_k, m$; the corresponding elementary graph would first appear to be $A'(n) = U_{\phi_1} \circ E(\mathbf{a}) \circ U_{\phi_2}$, where $\phi_1(m_i) = i$, $1 \leqslant i \leqslant k$, and $\phi_2 : [1] \to \{m\}$. This is shown in Fig. 12.

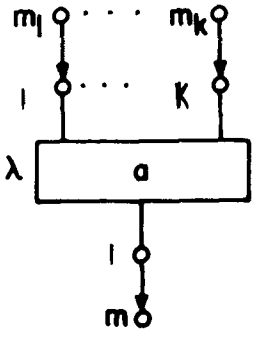


FIG. 12. $U_{\phi_1} \circ E(\mathbf{a}) \circ U_{\phi_2}$.

As a first approximation, we might reflect the i-th node execution, say of node n, by forming the composition $D_i \circ A'(n)$, where D_i is the partial output resulting from the first $(i - 1)$ node executions. However the action corresponding to the execution of a node in a c-s flowchart is somewhat more complex than this. First, if $m \notin \{m_1, \ldots, m_k\}$ and we form $D_{i+1} = D_i \circ A'(n)$ where $m \in (D_i)$, the result will have an exit terminal m connected both to $\langle\lambda, m\rangle$ and to its previous connection in D_i; the execution, however, destroys the previous content of m, breaking the latter connection. On the other hand, this D_{i+1} will have no terminals for $m_1, , \hat{m}, , m_k$, although these are unaffected

by the execution. The appropriate graph to represent the effect of the execution is $A(n) = Z_{m,\varnothing} \circ A'(n) \circ 1_{\{m_1,\ldots,m_k\}}$, which has the form of Figure 12, with the addition of a disconnected entry terminal m in case $m \notin \{m_1,\ldots, m_k\}$, and of exit terminals for $m_1,\ldots, \hat{m},\ldots, m_k$, connected directly to the appropriate entry terminals.

When f is a constant function, i.e., $|\omega(n)| = 1$, this suffices. Otherwise the effects of the execution are much more widespread. Not only is the new content of m a logical successor of the old contents of $m_1,\ldots, m_k$ but, at this level of analysis at least, so is every other quantity which exists after the execution. The effect is that the content of every active location $m' \neq m$ is replaced by $U_{k+1}^{k+1}((m_1),\ldots, (m_k), (m'))$, where we write (m_i) for 'content of m_i', and U_{ji} is the standard i-ary selector function which has as value its j-th argument. Thus the scope of the execution of a node is not determined by the node and its label alone, but also by the current state of the computation. All of these operations occur in effect simultaneously, so we construct their representation as follows, where D is the c-graph representing the previous computation, and $M^\dagger = w(D) - \{m\}$ if $|\omega(n)| > 1$, $M^\dagger = \varnothing$ otherwise

$$D^\dagger(n, D) = \sum_{m' \in M^\dagger}^{*} E(U_{k+1}^{k+1}(m_1,\ldots, m_k, m') \circ A(n).$$

Note that $D^\dagger(n, D)$ has the property that R^{-1} is functional, so the evaluation function, $V(D^\dagger)$, gives a unique term at each exit.

Given these correspondences, we can now define the computation of a c–s program $C = \langle\langle N; \alpha, \omega, r\rangle, l\rangle$ from entry j on a set of arguments $x_1,\ldots, x_{k_j}$, to produce an output D.

Initial:

$$D_0 = 1_{\pi_2 I_j}, \qquad \phi(m_p) = x_p \qquad \text{for} \quad m_p = (I_j)_p,$$

$$n_1 = (r(\langle N, j\rangle))_1.$$

Step: if $n_i \in N$:

$$D_i = D_{i-1} \circ D^\dagger(n_i, D_{i-1}),$$

$$n_{i+1} = (r(f((V(U_\phi \circ D_{i-1}))_{\langle m_1,\ldots,m_k\rangle})))_1.$$

where: $m_1,\ldots, m_k = (l(n_i))_3$, $f = (l(n_i))_2$

if $n_{i+1} = N$, $(r(f((V(U_\phi \circ D_{i-1}))_{\langle m_1,\ldots,m_k\rangle})))_2 = j'$, let ξ: $m_p \to \langle j', m_p\rangle$ for $m_p \in \theta_{j'}$, then $D = D_i \circ Z_{w(D_i)-\theta_{j'},\varnothing} \circ U_\xi$.

Observe that, although D is expressed as a linear composition of node execution graphs, the dependency relations expressed in the final graph are just the essential ones. As one would expect, two nonbranch node execution graphs $D^\dagger(n_i)$, $D^\dagger(n_j)$ commute iff neither result location is an argument or result location of the other node. The

proof of this is straightforward, using the c-graph commutativity lemmas for the components of $D^{\dagger}(n_i)$. Since for every $D^{\dagger}$, $a = w$, no problems of associativity arise.

The output is a structure related to a set of terms sharing arguments and subexpressions, with the evaluation being just the function (set) defined. The property 'R^{-1} is functional' assures that values are well-defined provided all the requisite arguments are supplied. At the first reference to each memory location not in the input set I_j, the corresponding $D^{\dagger}$ has an unmatched entry terminal, which 'floats' to the top, as an additional entry terminal for D_i and eventually D. If the reference is as result, this is a disconnected terminal, which would match and delete a like-named exit terminal of any precomposed graph, corresponding to the destruction of content of that location (the computation is not transparent to it). If the first reference is as an argument, the terminal is similarly supplied, though the value now becomes undefined, and the computation may fail to be completed, due to the inability to select a next node at some point. If it does complete successfully then the output represents the appropriate set of terms, with the entry terminal for the omitted argument in place. This structure, due to the restriction $|\omega(n)| = 1$, is isomorphic to a labeled directed ordered acyclic graph, for which we see [1], where the process of unfolding such a graph to the more usual tree-form representation of a set of terms is detailed.

Schemata

Since we wish to focus on the control structure of programs, ignoring the actual argument domains and operations, we now make the usual abstraction to schemata. A schema is obtained from a program by taking the operators as operator symbols and the branch functions as functions on values of predicate symbols; all else remains the same. We will also find it useful to allow for retaining a certain amoung of information about the interaction between operators and predicates in a shift relation, and the fact that certain operators are identities on Q. We will package the components which roughly correspond to the instruction set of the underlying machine together in the operator set Θ with the components:

A, a doubly graded finite set of operator symbols;

P, a graded finite set of predicate symbols;

F, a finite set of functions $2^P \to J$;

$S \subseteq A$, the identity or selector operators;

$G \subseteq A \times (2^P)^* \to (2^P)^*$, the shift relation.

A *c.s. schema* is now specified as

$$C = \langle\langle C, I\rangle, \Theta, M\rangle,$$

where

M is a finite set (of memory names),
Θ is an operator set,
$\langle C, l\rangle$ is a labeled c-graph,
$C = \langle N; \alpha, \omega, r\rangle$,
$l: N \to A \times F \times M^* \times M$,

subject to:

if $l(n) = \langle a, f, m_1, \ldots, m_k, m\rangle$, then
$i(f) = i(a) = k$,
$\sigma(f) = \omega(n)$,
$\langle N, j\rangle \in \omega \Rightarrow l\langle(N, j), i\rangle = I_j \in M^*$, input names for the j-th entry,
$\langle N, j\rangle \in \alpha \Rightarrow l\langle(N, j), \sigma\rangle = \theta_j \in M^*$, output names for the j-th exit,
r is a function on ω,
$|\alpha(n)| = 1$ for all $n \in N$.

To reverse the abstraction, we apply an *interpretation* $I = \langle Q, h, p\rangle$ where

$$h: A \to Q^{(Q^{i(a)})} \qquad \text{and} \qquad p: P \to 2^{Q^{i(p)}}.$$

The c.s.-program which we started with is just an interpreted c.s. schema. The usual procedure in studying schemata with $|M| > 1$ is to identify the schema behavior with the class of behaviors of its interpreted programs; two schemata are *equivalent* if they define the same mapping from interpretations to function $Q^j \to Q$, a schema is *terminating* if all its interpretations uniformly terminate, it is *empty* if there is no interpretation which ever terminates, and so on. We will call these respectively *I-equivalence*, *I-termination*, *I-emptyness*. However, in the case $|M| = 1$, it is relatively easy to define a notion of schema computation which does not refer to interpretations, and which gives an independent notion of equivalence, etc. In this notion, the schema is viewed as an object very like a finite automaton, which is supplied with a semi-infinite sequence of 'evaluations' specifying successive values of the predicates, which it uses to make branch decisions in stepping through its operations, adding the operator symbol to an output string at each step, until (if ever) it reaches an exit and halts. A schema thus defines a partial function from the set of evaluation sequences to the set of words on the operator set; this function can be taken as characterizing the schema, and schema-equivalence, schema-termination and schema-emptiness defined. These turn out, pleasantly, to be identical to the corresponding interpretation properties,

thereby giving decision procedures directly from the corresponding decision procedures for finite automata. This is presented at length in [10].

It is natural to take a similar approach to schemata with multiple registers. Looking at the definition of program computation given above, we see that it will do quite nicely as a definition of schema computation, except for the fact that U_ϕ is undefined, there being no $\langle x_1, \ldots, x_k \rangle$, so $V(D_i)$ is now a set of terms in A and M, while f is a function on predicate values (which are not defined for terms). In fact we have nothing corresponding to the input tape of an automaton, and no source of information for branch decisions. If we were willing to accept a linear sequence of operations as characterizing a given computation, we could use a linear sequence of evaluations, each giving the values of the predicates on all registers, but this would simply reduce to the one-register case, and nothing but complexity would be added. What is needed, clearly, is an 'input tape' or evaluation 'sequence' which has the same sort of partial order as the output, and each node of which is labeled with the evaluation of a newly produced quantity. Each execution, then, obtains an appropriate successor node to the nodes which supplied its argument quantities, and uses the evaluation given in that node as the evaluation of its result. So far, this is a rough idea; we must formalize and refine it. In particular, since the order type of our 'tapes' is no longer the integers, 'next place' is no longer well defined; we must establish some way of deciding which successor of a node or set of nodes is to be chosen at a given point in computation. As usual, it is useful to abstract as far as possible, attempting to extract a fundamental, more or less mathematical structure, and see what possibilities arise there, unprejudiced by the peculiarities of the particular instance in which we initially wish to apply it.

Since we are concerned with the interaction of schema and 'tape,' which we will henceforth call *evaluation net* or simply *net*, we can ignore many internal details of the schema, and even (for the present) the output. The program is coupled to its data, and thus the schema to its net, through the memory locations; data, or the evaluations which represent data, are seen only by a call for the content of a location. We can view the schema as a sequential control with 'heads' M; at each step of the computation, it performs, depending on the current inputs from the 'heads,' a choice of next internal state and a motion of a single head to a different net location or evaluation. This, of course, is a description of a multihead, perhaps multitape finite automaton; at least some of the investigations of multitape automata have been motivated in just this way, with each memory location corresponding to a tape, on which the head advances each time a new quantity is stored there. As long as each memory location is used independently, and the content of one is never transferred to another, this mirrors a schema computation well enough, although most dependencies are not represented. Unfortunately, that is a rare and relatively uninteresting mode of operation. We need at least the elementary operation 'place head m_j on the same square with head m_i'. To model programs with a single unary operator, this is sufficient and it is essentially this model that is used in [5, 6, 9] to prove the undecidability of I-emptiness for two-

register program schemata. The 'move m_j to m_i' operator is used only once in the initialization, and the remainder of the operation is that of a conventional one-tape, two-head finite automaton. For schemata with n-ary operator symbols, $n > 1$, the next-node function on the net requires something more, since a node may have a number of immediate predecessors and successors. A typical operation is "place head m on the common immediate successor of the current positions of $m_1, \ldots, m_k$", or "... of $\langle m_1, \ldots, m_k \rangle$". To make this deterministic, one can restrict the tapes to allow only a single unique successor to an ordered set of nodes. This is clearly too severe for our purposes, since a program may easily want to compute both the sum and difference of a given pair of quantities, for example. This suggests that the tape nodes should be labeled with operator symbols, in addition to evaluations. This is still not enough for deterministic computation on arbitrary tapes, but here the restriction to 'at most one a-labeled successor to any ordered set of nodes' is a natural one, at least in the application, since the sum of two quantities, for example, is always the same, no matter how many times is it computed, and therefore a single evaluation on the tape to represent it is appropriate. Alternatively, we might give up determinacy and use the class of computations of an automaton on a tape. Both of these approaches will be used, among others, but it is time to become a bit more formal.

Linear Net Automata

DEFINITION. An *evaluation net* over an alphabet Σ is a labeled c-graph $\langle\langle \Gamma; \alpha_e, \omega_e, R_e \rangle, l_e \rangle$ where $l_e : \Gamma \to \Sigma$. We will denote the set of acyclic evaluation nets over Σ by $\Sigma^\dagger$.

Let the set of *Σ-neighborhoods* of set μ, $\eta(\mu, s, \Sigma)$, be the set of c-graphs on $\mu \cup s$, which are such that every node is in μ or is an immediate successor or predecessor of a member of μ, and with label set Σ. A *linear*[3] *net automaton* $\mathcal{A}$ is specified by:

N, set of internal states;

M, set of heads;

Σ, input alphabet;

$f: N \times \Sigma^M \to N$, next state function;

$I \subseteq N$, initial states;

$F \subseteq N$, final states.

This much is the internal structure, and is perfectly conventional. Tape action is represented by $\sigma: N \times \eta(M, s, \Sigma) \to (M \times (M \cup s) \times G(\Sigma))^*$, where $G(\Sigma)$ is the set of c-graphs labeled by Σ.

For net E_i and neighborhood $\nu \in \eta(M, s, \Sigma)$, we say ν *matches* E_i iff there is an

[3] 'linear' because the set of configurations occurring in a computation is linearly ordered by 'next configuration'. Part II will introduce net automata which do not have this property.

isomorphism $\phi_i : \nu \xrightarrow{\text{into}} E_i$ which is identity on M. If there is more than one such isomorphism on dom σ the operation is nondeterministic, of which more later.

The *computation* of an automaton $\mathscr{A}$ from initial state $n_0 \in I$ on an evaluation net E is defined by the iteration:

$$E_0 = E,$$
$$n_i = f(n_{i-1}, l_{E_i}(M)).$$

The operation which produces the next net is the following chain of mappings:

$$n, E_i \xrightarrow{1,1\times\phi_i^{-1}} n, E_i, \nu \xrightarrow{1,1,\sigma} n, E_i, \langle\langle m_{i_1}, s_{i_1}, G_{i_1}\rangle,\ldots,\langle m_{i_k}, s_{i_k}, G_{i_k}\rangle\rangle$$
$$\xrightarrow{1,1,\phi_i} n, E_i, \langle\langle m_{i_1}, \gamma_{i_1}, G_{i_1}\rangle,\ldots,\langle m_{i_k}, \gamma_{i_k}, G_{i_k}\rangle\rangle \xrightarrow{1,\nu} n, E_{i+1},$$

where ν is defined by:

$$E_{i+1} = \cdot \int_{G_{i_k}}^{\gamma_{i_k}} \cdot \int_{Z_{a_{i_k}, w_{i_k}}}^{m_{i_k}} \cdot \int \cdots \cdot \int_{G_{i_1}}^{\gamma_{i_1}} \cdot \int_{Z_{a_{i_1}, w_{i_1}}}^{m_{i_1}} E_i$$

with

$$a_{i_1} = \alpha_i(m_{i_1}), \qquad w_{i_1} = \omega_i(m_{i_1}), \text{ etc.}$$

This looks rather formidable, and it does allow a great deal of flexibility. Intuitively, the net operation which corresponds to tape action consists of the following.

1. For each internal state n, the domain of σ contains one or more 'patterns', including nodes representing heads. Identify (if such exists) a neighborhood of E_i which 'matches' (by isomorphism ϕ_i) one of these patterns.
2. For the first triple in the value of σ for this pattern, the first and second components identify a 'head' m_i and a node, respectively; via ϕ_i, these correspond to a similarly named 'head' node and a node γ_i in E_i.
3. Delete the 'head' node by substituting a Z-graph for it, and substitute the third component of the value of σ for the node γ_i.

Repeat steps 2 and 3 for each subsequent triple in the value of σ, if any. This is just a generalized "read, move head and write" operation. Its specialization to several familiar cases is given in the following examples.

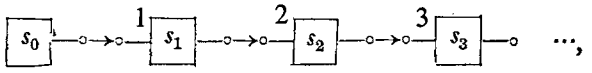
EXAMPLE 1. one-tape, one-head, one-way motion:

$$M = \{0\},$$

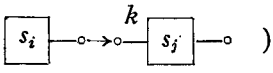
domain is

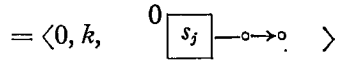
$$\langle\langle Z; Z \times \{1\}, Z \times \{1\}, \langle\langle n, 1\rangle, \langle n', 1\rangle\rangle \text{ for } n \geqslant 0\rangle, l: Z \to \Sigma\rangle,$$

where Z is the nonnegative integers. This can be drawn

$E_0 =$ 

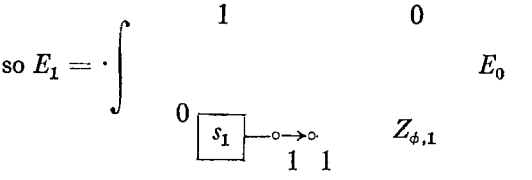
where terminal indices, all being equal to 1, have been omitted.

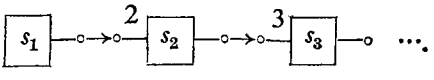
$\sigma(n,$ 

$) = \langle 0, k,$ 

$\rangle$

for all $n \in N$, $s_i, s_j \in \Sigma$, k a free variable over Z,

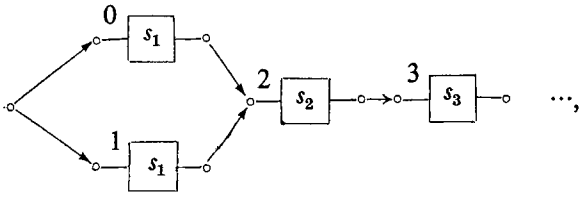
so $E_1 =$ 

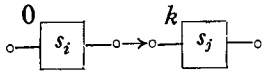
$=$ 

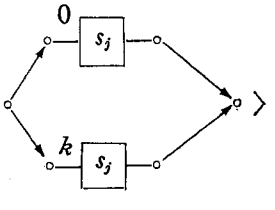
EXAMPLE 2. one-tape, one-head, two-way motion:

$$M = \{0\},$$

domain nets have the form

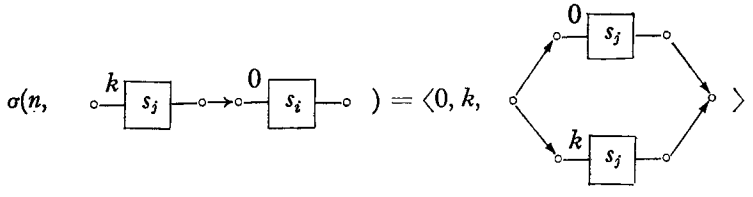


$\sigma(n,$ 

$) = \langle 0, k,$ 

$\rangle$

for n a 'read forward' state



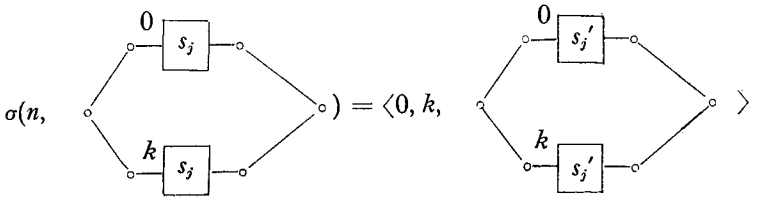
for n a 'read backward' state.

Multitape, one head per tape, is just a tupling of the above. For multiple heads per tape, the only question concerns the case of two or more heads on the same square. Due to the form of the neighborhoods, the heads do not see each other, and do not interact. In a 'destructive read' automaton, such as the first example above, multiple heads do of course interact.

EXAMPLE 3. Turing machine, one-head, one-tape

$$M = \{0\},$$

domain same as previous example.



for $\langle n, s_j \rangle \rightarrow$ print $s_{j'}$.

For 'move right' and 'move left', same as for the two-way automaton.

Tree automata do not fit within this model in a reasonable way, since the number of heads and of internal states required in a computation is unbounded.

We can easily add an output to make the automaton a transducer by adding

$$\begin{aligned} &\Sigma' && \text{the output alphabet,} \\ &g\colon N \rightarrow G(\Sigma) && \text{the output function,} \end{aligned}$$

and an action

$$D_{i+1} = D_i \circ (g(n_i))',$$

where $(g(n_i))'$ is an isomorph of $g(n_i)$ with node set disjoint from that of D_i .

A net automaton $\mathscr{a}$ now defines a partial function or relation $\mathscr{a}: \Sigma^{\dagger} \to N \times \Sigma'^{\dagger}$, and we can examine it in terms of the various specializations of this function, e.g., $\mathscr{I}(\mathscr{a}) = \mathscr{a}^{-1}(F \times \Sigma'^{\dagger})$, or the restriction of the function $\mathscr{a}$ to $\mathscr{I}(\mathscr{a})$, its action on nets which it accepts. Nearly all of the classical questions are unsolvable for the full generality of the model, but we can find appropriate restrictions on σ which will let many of the standard arguments go through in substantially more general situations. In order to state a basic lemma which underlies one of the standard arguments of finite automata theory, we must define several subgraphs of E and E_i. From the definition of neighborhood and of a match between a neighborhood and a graph, we can see that only nodes which are in the domain of the isomorphism ϕ_i can affect the action of $\mathscr{a}$ at step i. E_i can contain other successors and predecessors to members of M, and the nodes M may even have additional terminals and incident edges, without having any effect on the operation of $\mathscr{a}$, at least at step i. If we define a *σ-neighborhood* of M in E_i as the subgraph of E_i containing the domain of every M-preserving isomorphism $\phi: \eta \to E_i$, where $\eta \in \pi_2$ dom σ, we can say that the next-net operation of $\mathscr{a}$ on E_i depends only on this σ-neighborhood. Extending, we can define the following: $A_{\sigma}(E, M)$, the *M-neighborhood-accessible* subgraph of E, is $E - (\Gamma - b)$, where b is the set of nodes of E in σ-neighborhoods of nodes accessible from members of the set M.

$V(E, \mathscr{a}, i)$, the *subgraph of E visited by $\mathscr{a}$ up to step i*, is $E - (\Gamma - v_i)$, where v_i is the set of nodes selected for substitution (i.e., appearing as ϕ_i-image of a second component of a value of σ used in the i-th step) in steps $1,\ldots, i$ of the computation of α on E, together with all nodes of E in the neighborhoods used in their selection.

$W(E, \mathscr{a}, i)$ is the subgraph of E_i, $E_i - (\Gamma_i - \text{Ins}_i)$ where Ins_i is the set of nodes which have been inserted by substitution up to step i together with their σ-neighborhoods.

In contexts where the automaton $\mathscr{a}$ and net E are fixed, we can use the abbreviations

$$A_j^i \text{ for } A_{\sigma}(E_j^i, M), \qquad V_j^i \text{ for } V(E^i, \mathscr{a}, j), \qquad W_j^i \text{ for } W(E^i, \mathscr{a}, j), \text{ etc.}$$

The following lemma is a generalization of a familiar, and quite simple lemma of standard automata theory, which might be called the 'cut and paste' lemma, namely, the following.

Let A be an (R-S one-way, one-head finite) automaton with state set S and alphabet Σ, and let $r, t, u, v, w, y \in \Sigma^*$, $s_i \in S$. Consider A as a mapping: $\Sigma^* \to \Sigma^* \times S$. If $A(u) = t, s_1$, $A(uv) = tw, s_2$, and $A(r) = y, s_1$, then $A(rv) = yw, s_2$.

The central property, of course, is that the behavior of A on v depends only on its internal state on entering v. The corresponding lemma for two-head one tape automata requires that not only the internal state reached at the transition point, but also the segment of tape between the two heads be the same for both computations. The following lemma extends this idea to general linear net automata.

LEMMA 5. *For any linear net automaton a and nets E^1, E^2, if for some i, j,*

$$n_i^1 = n_j^2, \qquad A_i^1 \cap W_i^1 \overset{\phi_1}{\cong} A_j^2 \cap W_j^2 \qquad \text{and} \qquad w(D_i^1) \overset{\phi_2}{\cong} w(D_j^2),$$

then there exist ψ_1, ψ_2 such that for

$$E = \int_{U_{\psi_1} \circ (A_j^2 - W_j^2) \circ U_{\psi_2}}^{\Gamma^1 - V_i^1} E^1,$$

$$a(E) = \left\langle n^2, \int_{U_{\phi_2} \circ (D^2 - D_j^2)}^{D_{i-1}^1} D^1 \right\rangle$$

Proof. The isomorphism ϕ_1 as applied to the graph terminals of $A_i^1 \cap W_i^1$ must be understood as holding the terminal index set of E^1, E^2 fixed, but not the entire set of graph terminal indices. Consider a possible element of ϕ, on a 'fresh-cut' exit terminal, $\phi(\langle N_i^1, \langle\langle a, x\rangle, \langle b, y\rangle\rangle\rangle) = \langle N_j^2, \langle\langle c, x\rangle, \langle d, y\rangle\rangle\rangle$, where N is the node set of $A \cap W$, $a \in N_i^1$, $b \in E_i^1 - N_i^1$, $c \in N_j^2$, $d \in E_j^2 - N_j^2$, $x, y \in J$ and $\phi_1(a) = c$. This induces an extension of ϕ_1, $\phi_1' : b \to d$. Thus ϕ_1' maps the accessible nodes 'adjacent' to $A_i^2 \cap W_i^2$ 1 — 1 onto those 'adjacent' to $A_j^2 \cap W_j^2$. But these adjacent nodes are nodes of E^1 and E^2, respectively, and ψ_1 can be defined by:

$$\psi_1 : \langle V_i^1, \langle\langle a, x\rangle, \langle b, y\rangle\rangle\rangle \to \langle (E^2 - V_j^2), \langle\langle c, x\rangle, \langle d, y\rangle\rangle\rangle$$

for all a, b, c, d such that $\phi_1' : a \to c$, $b \to d$ and $b \in E^1 - V_i^1$, $d \in E^2 - V_j^2$; ψ_2 is defined correspondingly, from exit terminals of $E^2 - V_j^2$ to entry terminal of $E^1 - V_i^1$.

It suffices to show that $a : E$ reaches a configuration with internal state n_j^2 and M-neighborhood accessible subgraph isomorphic to A_j^2 (written 'accessible configuration $(n_j^2, A_j^2)'$), since from a configuration it must continue exactly as $a : E^2$. We are given that $a : E^1$ reaches accessible configuration (n_i^1, A_i^1) depending only on the subgraph V_i^1 of E^1; i.e., no nodes outside V_i^1 affect the computation, and the same configuration (n_i^1, A_i^1) would be reached for $a : V_i^1$, and equally for E. $E_i - W_i = E - V_i$ since this is just the undisturbed portion of E at step i, so E_i contains an isomorph of A_j^2,

$$\int_{A_i^1 \cap W_i^1}^{A_j^2 \cap W_j^2} A_j^2,$$

and, since M is included in this isomorph,

$$A(E_i, M) \cong A_j^2.$$

Since we are given $n_i^1 = n_j^2$, the lemma is proved. Q.E.D.

COROLLARY. *Under the hypotheses of the Lemma,*

$$E^2 \in \mathscr{I}(a) \leftrightarrow E \in \mathscr{I}(a).$$

From Lemma 5, the following theorem follows by a familiar argument:

THEOREM 1. *Let $\mathscr{a}$ be finite, and such that at most $\nu < \omega$ nonisomorphic graphs $A_i \cap W_i$ can occur in any one of its computations. Then the question, is $\mathscr{I}(\mathscr{a}) = \varnothing$, is decidable.*

Proof. If $\mathscr{a}(E) \in F \times \Sigma'^{\dagger}$, there is a computation of length k of $\mathscr{a}$ on E such that $n_k \in F$. If $k \geqslant \nu$, there must be i, j, $i < j$, to which Lemma 5 can be applied. If $k < \nu$, then the computation is one of a finite set which can be enumerated to recover an E' such that $\mathscr{a}(E') \in \mathscr{I}(\mathscr{a})$. If the set of all computations of length ν does not contain at least one which leads to a state in F, then no such E exists.

For any class of net automata, this gives an approach to solving the emptiness problem by showing that the 'active sets', $A_i \cap W_i$, are finite, and therefore belong to only finitely many isomorphism classes. For the familiar classes of one way, one or many-tape finite automata, this is straightforward. It is of course impossible for two-head-per-tape automata, and also fails for two-way automata, although the emptiness problem is solvable for the one-head, one-tape case.

Schema as Automaton

Returning now to schemata, we can see easily how the graph and branch function of a c.s. schema are modeled by a linear net automaton. The output action can also be taken over directly, while the input net and next-net operation must be specified so as to provide the proper abstraction of the data generated and received as input in a program computation. We wish to represent only the essential sequence, so as with the output, the node representing a given quantity should require connection only with those quantities which entered into its formation. When an operation is performed which in an interpretation would produce a new quantity, the corresponding evaluation is to be found as a successor of the argument evaluations. We can in fact consider a c.s. schema $\mathscr{C}$ directly as a net automaton, its *n.d.r.* linear net automaton, with

$N = N \cup (\omega \mid \{N\}) \times \{i\} \cup (\alpha \mid \{N\}) \times \{\sigma\}$, states correspond to instruction nodes and entry-exit terminals;

$M = M$, heads correspond to storage locations;

$\Sigma = A \times$ evaluations, the operator set and the values for the predicate symbols;

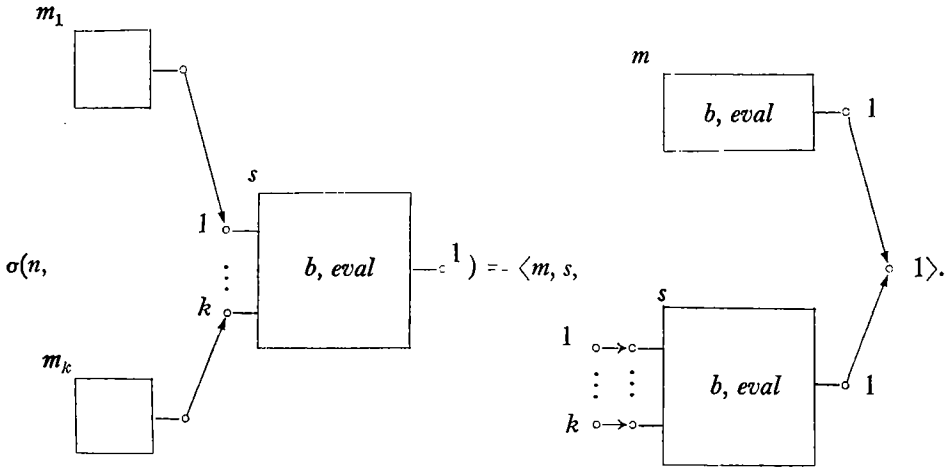
$f = f \circ r$, next instruction state is determined by choice of node exit and its r-successor;

$I = (\omega \mid \{N\}) \times \{i\}$, initial states are entry terminals;

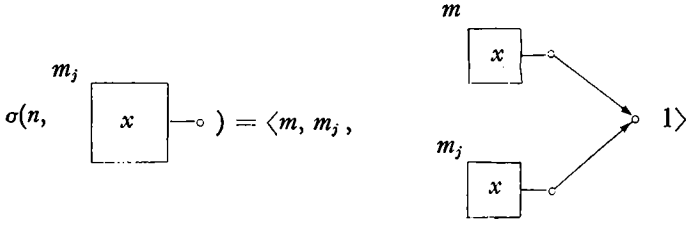
$F = (\alpha \mid \{N\}) \times \{\sigma\}$, final states are exit terminals;

σ has terms of the form below:

If $l(n)$ has form $\langle b, f, m_1, \ldots, m_k, m \rangle$, and $b \notin S$ (the identity operators) then σ includes a term



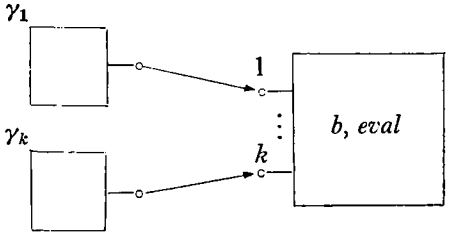
For $l_1(n) = b \in S$, say $b = U_j^k$, σ includes a term



where x, like *eval*, is a variable, this time over Σ. The subset $\mathscr{E}$ of $\Sigma^\dagger$ is defined by the following constraints:

$$E \in \mathscr{E} \Rightarrow E \quad \text{is acyclic.}$$

For any subset of Γ $\{\gamma_1, \ldots, \gamma_k\}$ and any operator symbol b there is at most one subgraph of E of the form



If $\mathscr{C}$ includes a nontrivial shift relation, this would enter in the definition of σ, in that the schema above would include evaluations labeling $m_1, \ldots, m_k$, and the *eval* in node s, instead of representing a free variable over evaluations, would be an evaluation which, with evaluations $1, \ldots, k$, satisfied the relation.

A step on the input, then, does the following: The unique node labeled with the operator of the current instruction, and immediately succeeding the ordered set $m_1, \ldots, m_k$ of 'head' nodes is selected for substitution; it is replaced by itself with 'head' node m attached in parallel with its exit terminal, hence to every entry terminal which immediately succeeds it, and with its content, particularly the evaluation, as label, and hence available for the next-state function of subsequent steps.

The hypotheses of Theorem 1 are not satisfied here, since we can easily place two 'head' nodes at the same point (i.e., give them the same successors) and then move one an indefinite distance, making the set $A_i \cap W_i$ arbitrarily large. Since a linear-sequenced net with unconstrained evaluations over $\Sigma = A \times \{0, 1\}$ is clearly within the set $\mathscr{E}$, we can easily mimic Paterson's proof of undecidability of termination for two-register schemata [9, 6], to show that for this class of automata the emptiness problem is unsolvable. On the other hand, we get immediately the result that emptiness is decidable for liberal schemata, since by definition of liberality no expression is computed twice, which in this context (due to the 'unique b-successor' constraint on $\mathscr{E}$) means that no node of E is ever selected twice for substitution, which means in turn (given the form of the neighborhoods of σ) that we are assured that for any $E \in \mathscr{I}(a)$, there is a subgraph E' of E such that for a operating on E', the node set of $A_i \cap W_i$ is always contained in M, and hence the theorem applies to a over the set of subgraphs E', which has an equivalent emptiness problem.

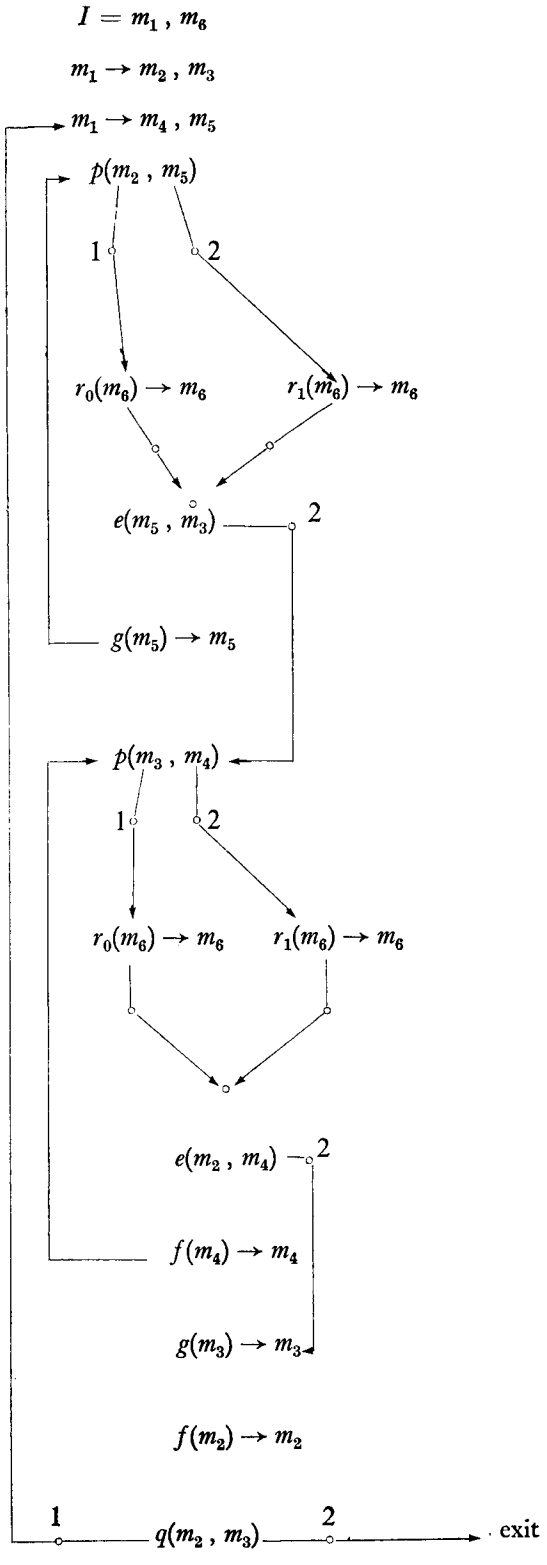
The preceding discussion gives a mode of schema operation which corresponds nicely to program computation for schemata with monadic predicates only. In fact, we can easily prove the following.

THEOREM 2. *For every c.s. schema $\mathscr{C}$, interpretation I and argument set $X = \langle x_1, \ldots, x_k \rangle$ such that $\mathscr{C}$ has only monadic predicate symbols, there is an evaluation net E such that output of $(I(\mathscr{C}))X = a(E)$, where a is the n.d.r. linear net automaton of $\mathscr{C}$, and conversely, for every E, a corresponding I and X exist such that the above equation holds.*

Proof. We take $\Sigma = A \times 2^p$; each node is thus labeled with all the information the schema uses about the corresponding quantity. It is easy to see how a computation of $I(\mathscr{C})$ can be made to produce a corresponding net, using operations only slightly modified from the next-net action of a, to produce a new node and evaluation when a reads one. For the converse, we use the familiar free interpretation, in which the domain is the set of terms in the operator symbols and a set of variables, and the function assigned to an n-ary operator symbol b maps an n-tuple of terms $\langle t_1, \ldots, t_n \rangle$ into the term $b(t_1, \ldots, t_n)$. Since every node of a net E read by a corresponds to a newly-produced term, the predicates can be defined over terms to correspond to the values given by the net E.

Unfortunately, when we try to extend this correspondence, we encounter the following case.

Let $\mathscr{C}$ be the schema indicated by:


$I = m_1 , m_6$
$m_1 \rightarrow m_2 , m_3$
$m_1 \rightarrow m_4 , m_5$
$p(m_2 , m_5)$
1
2
$r_0(m_6) \rightarrow m_6$
$r_1(m_6) \rightarrow m_6$
$e(m_5 , m_3)$
2
$g(m_5) \rightarrow m_5$
$p(m_3 , m_4)$
1
2
$r_0(m_6) \rightarrow m_6$
$r_1(m_6) \rightarrow m_6$
$e(m_2 , m_4)$
2
$f(m_4) \rightarrow m_4$
$g(m_3) \rightarrow m_3$
$f(m_2) \rightarrow m_2$
1
$q(m_2 , m_3)$
2
exit

For an interpretation in which e is equality, $f^n(x) = f^m(x) \Rightarrow n = m$ and similarly for g, and $r_0(r_1)$ adjoins 0(1) to an initially empty word, we can easily see that the amount of information required for branching (and recorded by r) increases as the square of the number of quantities produced, since $p(f^n(m_1), g^p(m_1))$ is used for all $p \leqslant n$, and conversely, for unbounded n. Therefore no finite alphabet Σ will suffice to supply this information at the rate of one letter of Σ per quantity produced in the computation. This difficulty relates just to the size of the 'active set' $A_i \cap W_i$ of nodes in E_i, the nodes which have entered the computation and may do so again. Any class of schemata with polyadic predicates for which this set in the related automata may be unbounded will contain members in which the necessary information content of the path of computation of an interpretation grows more rapidly than the number of new quantities computed. Conversely, if the active set is bounded, then the only information for the schema contained in a new quantity is the value of the predicates on it in combination with the other members of the active set, which is finite, and can be encoded in the evaluation, as will be detailed below.

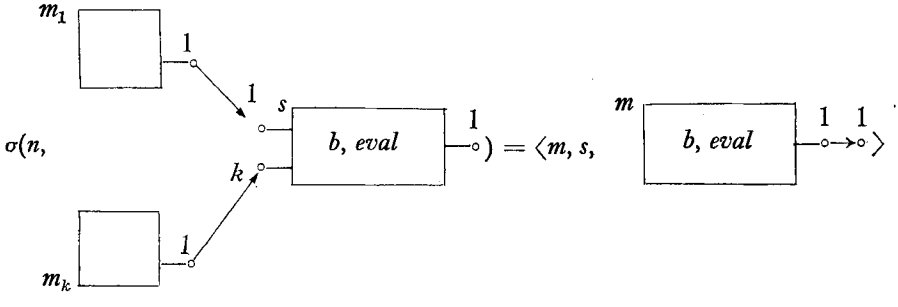
In the other direction, we can still say that for any net E there is an interpretation I and arguments $X = \langle x_1, ..., x_k \rangle$ such that output of $(I(\mathscr{C}))X = a(E)$.

The free interpretation still works, although it is perhaps clearer to bring the evaluations up into the interpretation, and let b operating on $t_1, ..., t_n$ produce $\langle b(t_1, ..., t_n), eval \rangle$, where *eval* is the evaluation in E corresponding to this term. The predicates of I are then just components of the function used in the schema for decoding evaluations. This yields the following:

THEOREM 3. $(\exists E)\, a_1(E) \neq a_2(E) \Rightarrow (\exists I, X)(I(\mathscr{C}_1))(X) \neq (I(\mathscr{C}_2))(X)$, *where* a_i *is the n.d.r. linear net automaton of* $\mathscr{C}_i$.

This situation is clearly unsatisfactory, since equivalent automata can be obtained from inequivalent schemata. In looking for a derived automaton which will avoid this difficulty, we would like the following (inconsistent) properties: (1) new information should be supplied from the net each time a term is recomputed, (2) the active set in E_i should be bounded, and (3) all evaluations supplied for a given term should be consistent. The following provides the first two:

The *d.r. linear net automaton* of c.s. schema $\mathscr{C}$ is identical to the n.d.r. linear net automaton of $\mathscr{C}$ except for the terms of σ corresponding to proper operations ($b \notin S$), which have the form:



That is, the 'head' node m completely replaces the selected node, which is missing from E_j for all $j > i$. Since any node not in M which is part of a matched neighborhood is deleted in the same step, we clearly have $W_i \subseteq M$, hence $A_i \cap W_i$ is bounded, and since a new node must be used for each recomputation of a term, new information is supplied; the third desideratum is not met. We must clearly expand the set $\mathscr{E}$ of nets, at least for illiberal schemata, by allowing multiple successors of a given set of nodes with the same operator label. We will call this expanded set $\mathscr{E}_{\text{d.r.}}$. This introduces a nondeterminism into the automaton computation, which we can accommodate by defining $a(E)$ as the class of all exit–output pairs resulting from possible computations of a on E. Alternatively we may think of the output of a applied to E as homomorphic to a tree, with a branch of unbounded degree corresponding to each multiple match; the computation proceeds independently on each branch, and a path in the tree corresponds to a possible computation of a on E.

LEMMA 6. *Let $\mathscr{C}$ be a* c.s. *schema and a its derived* d.r. *linear net automaton. Then for every I and X, there is an E in $\mathscr{E}_{\text{d.r.}}$ such that $(I(\mathscr{C}))(X) = a(E)$.*

Proof. We must first show that an encoding scheme exists to construct evaluations with sufficient information for polyadic predicates. Let each evaluation be an array with an index number $\leqslant |M|$, containing a row for each predicate symbol. The row for an n-adic predicate will have a place for each n-tuple of numbers $\leqslant |M|$ which includes the index of the evaluation. For example, in evaluation indexed k, the row for a binary predicate P has positions for $1k, 2k, \ldots, |M|k, k1, k2, \ldots, k|M|$. In the j, k place is recorded a 0 or 1 such that the mod 2 sum of that value with that in the j, k place for the same predicate in evaluation j is the truth value of P on the quantities corresponding to evaluations k and j. As long as not more than $|M|$ quantities are simultaneously present, evaluations can clearly be assigned with index the smallest unused number so that the value of a predicate on any set of these quantities can be determined from the evaluations corresponding to these quantities. For quantities $x_{i_1}, \ldots, x_{i_j}$ and x_{i_k}, ordered predicate set P and bound $|M|$ we define

$$\text{Eval}(|M|, P, \{x_{i_1}, \ldots, x_{i_j}\}, x_{i_k})$$

to be the evaluation for x_{i_k} so constructed, with entry 0 for each combination including an index $\leqslant |M|$ not present in $i_1, \ldots, i_j, i_k$. Referring now to the definition of computation of a c.s. program, we construct the E corresponding to any computation by adding the following to each step of the iteration.

Initial: $E_0 = \sum^*_{i \in I_j} E(\{m_i\}, \varnothing, \{m_i\}, \text{Eval}(|M|, P, x_{I_j}, x_i))$

Step: For $a_i \in S$, $E_{i+1} = E_i \circ U_\phi$ where ϕ: $(l_3(n_i))_k \to l_4(n_i)$, i.e., $m_k \to m$.

For

$$a_i \notin S, \qquad E_{i+1} = E_i \circ Z_{\{m\},\varnothing}$$

$$\circ\, E(\gamma_i, \{m_1, \ldots, m_k\}, \{m\}, \langle a_i, \mathrm{Eval}(|\, M \,|, P, l_{D_i}(\{N_i\} \times M), l_{D_i}\langle N, m\rangle)\rangle)$$

$$\circ\, 1_{\{m_1,\ldots,m_k\}}.$$

where $\gamma_i \neq \gamma_j$, $j < i$.

It is immediate on comparing this with the operation of the d.r. automaton derived from $\mathscr{C}$, that a set of choices of matches in the computation of a exists such that the two computations correspond at every step, and that evaluation nodes are deleted by the automaton just where they are created by the program, and finally that the predicate values derived from the evaluations by the automaton are just those which hold for the memory content at the corresponding point of the program computation.

As before, but now in the inverse direction, we have the following.

THEOREM 4. $(\exists I, X)(I(\mathscr{C}_1))(X) \neq (I(\mathscr{C}_2))(X) \Rightarrow (\exists E)\, a_1(E) \neq a_2(E)$ *when* a_i *is the* d.r. *automaton derived from* $\mathscr{C}_i$, $i = 1, 2$.

Thus, schema equivalence implies I-equivalence for d.r. automata, while the reverse is true for n.d.r. automata. Also, from the above remark that $W_i \subseteq M$ for all d.r. automaton computations, and the theorem, we have

COROLLARY. *S-emptiness is decidable for* d.r. *automata and their schemata.*

Some comments are in order on the relation of this result to previous work on schema I-termination and I-equivalence. Since the implication of Theorem 4 runs in one direction only, I-equivalent schemata may not be S-equivalent, and a schema whose d.r. automaton terminates for some E may not have any interpretation–argument pair for which it terminates. However, for liberal schemata, as remarked above, the d.r. and n.d.r. automata accept essentially the same nets, and I-termination is therefore the same as S-termination, and is hence decidable, as shown in [9]. A small extension, which should nevertheless be mentioned explicitly, arises from the shift relation. As in [10], this allows a finite amount of interpretation to be fixed—not all interpretations are to be considered, but only those which satisfy the shift relation. For example, any finite number of 'tags', 'flags', bounded counters, or computation by means of finite functions on finite amounts of storage may be specified as fixed for all interpretations of the schema. 'Liberal' and 'free' are naturally understood relative to this partial fixing: a schema is 'liberal' iff no result of a nonfixed operation is recomputed, and 'free' iff all paths allowed by the fixed portion of the interpretation may be traversed under some (allowed) interpretation.

The basic result on decidability of termination, Theorem 1, is not directly comparable with previous work on schema termination, even for 'pure' schemata without

shift relation; it is likely that it can be made to yield a somewhat wider class of decidable schemata then the liberal. This question has not been pursued exhaustively here, since our principal purpose is to establish a foundation for a fuller analysis of polyadic (multi-argument) computation to be carried out in Part II.

Preview of Part II

In introducing the evaluation function V of a labeled graph, we remarked that it could be applied to nonacyclic graphs, but we have not used this fact. Consider now $V(\mathscr{C})$, where $\mathscr{C}$ is a c.s. schema. Since $|\alpha(n)| = 1$ for all $n \in N$, the operators are all monadic, and the terms associated with the exits are linear strings of node labels, that is instructions, each augmented by a node exit terminal index. Each of these terms corresponds to a path through the graph from the entry terminal, the variable of the term, to the associated exit. If we interpret both schema and value, the operators are interpreted as functions which take memory content into memory content provided that the value of the branch function f on the argument content agrees with the node exit terminal index; otherwise, the value is undefined. Under this interpretation, at most one term in the entire value is defined for each initial complete memory content. We see thus that the schema value represents the set of all possible computations in the linear sequence sense; this is the proper and natural representation for unitary memory [2]. For multilocation memory, as we have seen, the essential sequence of a computation is much weaker; something like the program or schema output defined above represents it better. Is there a class of finite objects which relates to these computations as the c.s. schema and program relate to the linear sequenced computation? Part II proposes such a class of objects, the data sequenced schemata and programs, and investigates their properties.

References

1. M. A. Arbib and Y. Give'on, Algebra automata I: Parallel programming as a prolegomena to the categorical approach, *Information and Control* **12** (1968), 331–345.
2. C. C. Elgot, "The Common Algebraic Structure of Exit Automata and Machines" IBM Research RC 2744; Computing, Vol. 6, No. 3–4, January 1971.
3. Iu Ianov, The Logical Schemes of Algorithms, *in* "Problems of Cybernetics. I" (A. A. Lyapunov, Ed.) (1958) English Translation, Pergamon Press, 1960.
4. F. W. Lawvere, Functional semantics of algebraic theories, *Proc. Nat. Acad. Sci.* **50** (1963), 869–872.
5. D. Luckham and D. Park, "The Undecidability of the Equivalence Problem for Program Schemata," Report No. 1141, Bolt, Beranek, and Newman.
6. D. Luckham, D. Park, and M. S. Paterson, On formalized computer programs, *J. Comput. Sys. Sci.* **4** (1970), 220–249.
7. Zohar Manna, Properties of programs and the first-order predicate calculus, *J. Assoc. Comput. Mach.* **16** (1969), 244–255.

8. ZOHAR MANNA, The correctness of programs, *J. Comp. Sys. Sci.* **3** (1969), 119–127.
9. M. S. PATERSON, "Equivalence Problems in a Model of Computation," Thesis, Trinity College, University of Cambridge (August 1967).
10. J. D. RUTLEDGE, On Ianov's Program Schemata, *J. Assoc. Comput. Mach.* **11** (1964), 1–9.
11. J. D. RUTLEDGE, "Parallel Processes—Schemata and Transformations," IBM Research RC2912 (June 1970); Chapter in "Architecture and Design of Digital Computers" (G. G. Boulaye, Ed.), pp. 91–129, Dunod, Paris, 1971.
12. M. P. RABIN AND D. SCOTT, Finite automata and their decision problems, *IBM J. Res. Develop.* **3** (1959), 114–125.