MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LAB

Artificial Intelligence
Technical Memo No. 1
(Memo No. 211)

August 1967
Issued November 1970

EQUIVALENCE PROBLEMS IN A MODEL OF COMPUTATION

Michael Stewart Paterson

Work reported herein was reprinted by the Artificial Intelligence Laboratory, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0002.

Reproduction of this document, in whole or in part, is permitted for any purpose of the United States Government.

EQUIVALENCE PROBLEMS IN A MODEL OF COMPUTATION.

by

Michael Stewart Paterson.

August 1967. Trinity College.

A dissertation submitted for the degree of Doctor of Philosophy of the University of Cambridge.

PREFACE.

I am most grateful to my supervisor Dr. D.M. Park for his constant encouragement and assistance.

I should like to thank the Science Research Council and Trinity College, Cambridge, for grants to support this research. In this second typing I have been fortunate to have the typing services of the secretaries at the Artificial Intelligence Laboratory.

November, 1970

CONTENTS.

INTRODUCTION	1
PART I. DEFINITIONS AND ELEMENTARY RESULTS.	7
§1 Notation and prerequisites.	7
§2 'Program schemata' and some basic results.	12
2.1 Definitions.	13
2.2 Justification of definitions.	20
2.3 Syntactic characterizations.	23
2.4 'Reasonable' relations and rule-books.	31
§3 Some simple equivalence problems.	36
3.1 Loop-free schemata.	37
3.2 Schemata which always converge.	42
PART II. UNSOLVABLE PROBLEMS.	50
§4 Two-tape and two-headed automata.	51
4.1 Two-tape automata.	52
4.2 Two-headed automata.	56
4.3 Recursive tapes.	62
4.4 Translation to binary automata.	68
§5 Unsolvability of problems of program schemata.	71
5.1 Simulation and first results.	72
5.2 Decision problems for equivalences.	80
5.3 'Adequate' rule-books.	88

CONTENTS continued.

PART III. SOLVABLE PROBLEMS 95

§6 Schemata with non-intersecting loops and monadic function symbols. 98

6.1 Preliminaries. 99

6.2 P-representation. 103

6.3 Decision procedure. 111

§7 Free, liberal and progressive schemata. 118

7.1 Freeing liberal schemata. 118

7.2 Progressive schemata. 129

7.3 Full schemata. 131

7.4 Decision procedure for progressive schemata. 138

7.5 Conservative schemata. 145

SUMMARY. 147

References. 151

INTRODUCTION.

A central problem in the mathematical theory of computers and computation is to find a suitable framework for expressing the execution of a computer program by a computer. Within the framework we want to be able to provide answers to such questions as:

(1) Does a certain program perform a certain task?

(2) Are two programs equivalent, i.e., do they perform the same task?

(3) Under what conditions, if at all, will a program fail to halt?

(4) How can a given program be simplified, in some sense, or made more efficient?

These kinds of questions are customarily answered by experienced intuition, for simple programs, supplemented by trial and, often, error for more complicated ones. We should like to replace such methods by a formalizable procedure, capable of being carried out by a computer program. Unfortunately, under most definitions, all the above questions are provably unsolvable. The proof might proceed by saying that if we could answer any of these questions, we could certainly establish of a given program whether or not it ever halts. Then, if our programming language is sufficiently powerful to carry out

real computations, we should be able to reduce the halting problem for Turing machines, or some such unsolvable decision problem, to the halting problem for certain programs in our language.

In default of a complete procedure, it may, nevertheless, be worthwhile to look for various techniques by which to prove, say, that two programs are equivalent. J. McCarthy [9,10] introduces a formalism using conditional expressions, in which new functions are produced from old functions by a recursive definition process. A computer program in his formalism, given in the form of a flow diagram, in which the flow of calculation is controlled by conditional expressions, can be converted to a series of function definitions. Various techniques are presented whereby such recursive functions, and hence programs, may be proved equivalent.

Cooper [1] generalizes certain of these equivalence proofs to programs involving undefined functions, satisfying certain conditions and relations. These techniques seem to be applicable only to programs of a relatively simple nature and furthermore, for any actual programming language and computer, the intrinsic relations between the operations expressible in the language are often complicated. Even apparently simple relations may have to be qualified by exceptions and adjustments corresponding

to the idiosyncrasies of the language and computer. As a fairly trivial example, algebraically:

$$(x+1)(x-1) = x^2-1$$

but the corresponding results of the two 'routines':

```
Y := X-1
X := X+1          and     X := X × X
X := X × Y                X := X-1
```

as calculated by a computer will rarely be the same, and if X initially has a value close to unity the first expression is likely to give the more precise result. We cannot in general allow two such routines to be treated as equivalent.

Cooper stresses the need to be able to prove 'shallow results about large programs', by which he means results which depend only on rudimentary properties of the language and the simple relations between the basic functions; for example, we might consider the purely graph-theoretic transformations which can be made on the flow-diagram of a program.

Iu. Ianov [6], whose work is reported also by Rutledge [15], accomplishes this by considering the 'logical schemes' representing the sequential and control properties of programs which remain when almost all the

information about the nature of the basic operations is disregarded. He is able to obtain a complete decision procedure for the equivalence of schemes, but sacrifices a great deal of the essential structure of programs, leaving, in effect, little more than finite automata.

The work described here is an investigation into a model of computation which is, in a sense, an extension of Ianov's model. We regard our 'computer' as having storage divided up into a finite number of discrete 'locations', with each of which is associated a value from some fixed, usually infinite, domain. A typical computation statement or instruction:

$$L_2 := F(L_2, L_3)$$

has the effect of assigning a new value to the location L_2, which depends in some fixed but unspecific way on the previous values of L_2 and L_3 and the symbol F. The only other kind of statement performs some test, similarly unspecified, on the value of a location (without changing this value), and decides, on the basis of this test, which statement is to be executed next.

We are free to regard a statement as a 'machine-code' order for some computer, as an assignment or conditional jump statement in some autocode or ALGOL-like language, as a sub-routine of a program (in

which case a location may be a whole block of storage or some more general data structure), or even as some step in a process quite unconnected with digital computation. We shall usually adopt the terminology and motivation of the second of these.

A program, made up from such statements, is called a 'program schema', because as yet the function and test symbols which appear have no corresponding semantic content, that is to say, no arithmetical or logical functions are associated with them. An 'interpretation' of the language is an assignment of appropriate functions to these symbols and of initial values to the locations. An interpreted program schema is then, in effect, a computer program which could be executed on some idealized computer. We can define two program schemata to be 'equivalent' if, under all interpretations, the two programs produce identical results. Of course certain interpretations will be quite unsuitable in a computational context, and we can introduce various restrictions on the kind of interpretation, considering perhaps recursive interpretations, or interpretations over finite domains of values.

We should point out at this stage some of the shortcomings of our formalism, which arise from the

wholesale removal of explicitly defined functions and our consequent inability to express the self-modification of computer programs. We have no way of representing the recursive use of sub-routines or the indirect addressing of storage registers, and we must treat any vector or matrix as a single location rather than as an array of distinct elements. An extension of our language to include some such features would certainly be desirable, but this is beyound the scope of this dissertation.

The model of computation which program schemata provide, allows us to study the characteristic properties of, at least, some kind of computational process without the interference, and dominance, of the arithmetical and logical properties of the basic operations involved. We envisage a practical procedure for the automatic simplification of computer programs, which would consist of abstracting the corresponding schemata, applying to these some schemata-simplifying techniques which would be quite independent of the computer or language concerned, and finally translating back into the original language. In finding simplification algorithms for schemata, however, we have very soon to face the problems of deciding when two schemata are equivalent.

PART I. DEFINITIONS AND ELEMENTARY RESULTS.

§1. NOTATION AND PREREQUISITES.

We shall assume the familiarity of the reader with some of the elementary results of mathematical logic relating to computability and solvability, which appear, for example, in Davis [2] or Hermes [4]. We assume known a formal definition of algorithm or effective procedure. A decision problem enquires as to the truth or falsity of each of a whole class of statements, and such a problem is (recursively) unsolvable if there is no algorithm which supplies all the answers. For solvable decision problems, we will refrain from giving formal algorithms for the solutions and rely on intuition to verify that the informal procedures we describe could be replaced by such. We say that a decision problem is partially solvable if there is an effective enumeration of the true statements in the class, or equivalently, if there is an algorithm which, when presented with a statement of the class which is true, comes to the correct decision, but otherwise fails to terminate. We often wish to use results from the theory of Turing machines. The notation we adopt is close to that of Davis [2]. We give here our main definitions and state the theorems we shall require.

Definitions.

A Turing machine M, is a (2m x 4)-matrix, (m > 0), of the form:

$$\begin{pmatrix} q_1 & 0 & b_1 & q_1' \\ q_1 & 1 & b_2 & q_2' \\ q_2 & 0 & b_3 & q_3' \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ q_{m-1} & 1 & b_{2m-2} & q_{2m-2}' \\ q_m & 0 & b_{2m-1} & q_{2m-1}' \\ q_m & 1 & b_{2m} & q_{2m}' \end{pmatrix}$$

where $q_1,\ldots,q_m$ are distinct integers and:

$$q_i' \in \{q_1,\ldots,q_m\} \text{ for } i = 1,\ldots,2m.$$

The q_i are states and q_1 is the initial state of M. Each b_i is one of { 0,1,L,R,H }. We shall sometimes denote 0,1 by a_0,a_1 respectively.

A configuration α, ('instantaneous description' in [2]) is a string of symbols of the form:

$$Pq_ia_jQ\varepsilon$$

where P,Q are (possibly empty) strings of 0's and 1's, j = 0 or 1, and ε is a special symbol.

α is an initial configuration if $i = 1$, and α is a terminal configuration if ' $q_i\ a_j\ H\ q$ ' is a row of M, for some q.

The consecutive configuration to α is:

(i)	$OPq_ia_jQO\varepsilon$	if α is terminal (especial note !)
(ii)	$OPq_ra_kQO\varepsilon$	if ' $q_i\ a_j\ a_k\ q_r$ ' is a row of M,
(iii)	$OPa_jq_rQO\varepsilon$	if ' $q_i\ a_j\ R\ q_r$ ' is a row of M,
(iv)	$ORq_ra_ka_jQO\varepsilon$	if ' $q_i\ a_j\ L\ q_r$ ' is a row of M and $P = Ra_k$,
(iv)'	$q_rOa_jQO\varepsilon$	if ' $q_i\ a_j\ L\ q_r$ ' is a row of M and P is null.

A computation C, is a sequence of consecutive configurations starting from an initial configuration. The 0th configuration of C is the initial configuration and the $n+1^{th}$ is the consecutive configuration to the nth, for $n \geq 0$.

C halts if there is a terminal configuration in C, and diverges otherwise.

C halts on 0 if C halts and there is a terminal configuration of the form:

$$Pq_iOQ\varepsilon$$

and similarly for ' C halts on 1 '.

Consider the computation C corresponding to machine M and initial configuration:

$$11....1q_1 0\varepsilon$$
$$\longleftarrow n \longrightarrow$$

which may be described as ' starting M on n '. We shall say that:

$$C[M,n] = 0 \text{ if C halts on 0,}$$
$$= 1 \text{ if C halts on 1,}$$

and C[M,n] diverges otherwise.

The symbol 0 will be thought of as a 'blank' symbol, and the configuration:

$$q_1 0\varepsilon$$

will be described as ' starting on a blank tape '.

We now state, without proof, the few theorems which we shall require. Each is readily derivable from well-known results which appear as main theorems in any standard work on the theory of Turing machines. (Davis [2], Hermes [4].)

Theorem A. The decision problem of whether C[M,0] diverges for arbitrary M, is recursively unsolvable, and is not indeed even partially solvable.

[Hermes, p. 144]

Let $M_1, M_2 \ldots$ be an effective enumeration of the set of all Turing machines.

Theorem B. There is a Turing machine U, with the property that, for all n:

either $C[U,n] = C[M_n,n]$

or $C[U,n]$ and $C[M_n,n]$ both diverge.

[From Hermes, §30]

We will call such a U a universal Turing machine.

Theorem C. There is a Turing machine U, such that it is recursively unsolvable whether or not, C[U,n] diverges for arbitrary n.

[From Theorem B and Hermes, p.144]

Theorem D. Given any recursive function f, there is a Turing machine M, such that, for all $n \geq 0$,

$C[M,n] = 1$ if $f(n) = 0$

$= 0$ of $f(n) \neq 0$

[Hermes, Section 16]

§ 2. 'PROGRAM SCHEMATA' AND SOME BASIC RESULTS.

§ 2.0 In this section, we give the definitions of 'program schemata' and 'interpretation', of '(strong) equivalence' and several other relations between schemata, and justify our choice of some of these. We also consider alternative formulations of these relations which do not involve the concept of an 'interpretation'. We introduce the idea of a 'reasonable' relation between schemata, which is any relation intermediate to strong and weak equivalence, and finally, we formalize as a 'rule-book', the kind of algorithm which we would expect to be useful in simplifying schemata. These two concepts are due to D.M. Park.

§2.1 Definitions.

The formal language we will use contains the following symbols:

(i) integers,

(ii) $F_1^1, F_2^1, F_3^1, \ldots, F_1^2, F_2^2, \ldots$ (operator or function symbols),

(iii) $T_1, T_2, \ldots, T_k$ (transfer or test symbols),

(iv) $L_1, L_2, \ldots, L_m$ (location symbols),

(v) := (the assignment symbol),

(vi) brackets, commas, and other auxiliary symbols.

The number of symbols in the language is finite, but we shall not usually be concerned with the precise number.

The statements (or instructions) of the language are of two types:

(1) Operator instructions, (or computation instructions.)

a. $L_j := F_u^t(L_{k_1}, \ldots, L_{k_t})$

and (2) Transfer instructions, (or tests.)

a. $T_u(L_j)$ b,c

where $a,b,c,j,k_1,\ldots,k_n,t,u$ are integers. In both

cases, a is the prefix or address of the instruction. In (1), L_j is the assignment location and $L_{k_1}, \ldots, L_{k_t}$ are the retrieval locations. In (2), b and c are the left- and right- transfer addresses respectively.

A program schema is a finite sequence of instructions together with three integers, the initial address which is the prefix of some instruction, and the left- and right- terminal addresses which are not prefixes and are distinct, such that:

(i) the prefix of each instruction is its position in the sequence,

(ii) each transfer address is either the prefix of some instruction or else a terminal address,

(iii) either the last instruction in the sequence is a transfer instruction or else its prefix is one less than a terminal address.

We define a sequence π, through a schema as a (finite or infinite) sequence of integers satisfying the following conditions (where $\pi(i)$ denotes the ith element of π):

(1) $\pi(1)$ is the initial address of the schema,

(2) if $\pi(i)$ is the prefix of an operator instruction then:

$$\pi(i+1) = \pi(i) + 1$$

(3) if $\pi(i)$ is the prefix of a transfer instruction with transfer addresses b,c, then :

$\pi(i+1)$ is either b or c

(4) if $\pi(i)$ is a terminal address, it is the last element of π.

The value of a finite sequence through a schema is defined to be 0 or 1 according as the final element of it is the left- or right-terminal address. The value of an infinite sequence is undefined.

If the operator symbols and transfer symbols of a schema are interpreted as standing for particular functions and characteristic functions over a suitable domain, then the schema can be regarded as a computer program which could be executed by some idealized computer. A computation starts at the initial instruction of the schema with a specified initial value from the domain, associated with each location symbol. An operator instruction assigns a new value to the assignment location and the succeeding instruction is executed next. A transfer instruction leaves the values of all the locations unchanged but applies its characteristic function to the current value of its

location and, according to whether the resulting value is 0 or 1, the instruction with the left- or right-transfer address respectively, as prefix is executed next.

More formally, an interpretation I, is a mapping from the location, operator and transfer symbols into a set D, and the set of functions and characteristic functions on D, such that:

(i) to each L_i is assigned some element $I(L_i) \in D$,

(ii) to each F_i^n is assigned some n-adic function $I(F_i^n) : D^n \to D$,

(iii) to each T_u is assigned some characteristic function $I(T_u) : D \to \{0,1\}$.

The execution sequence π_I, and computation sequence $A_I(\pi_I)$, corresponding to a given interpretation I, are defined as follows, where $A_I(\pi_I)(i)(j)$ is the jth element of this:

(1) $\pi_I(1)$ is the initial address,

(2) $A_I(\pi_I)\ (1) = \langle I(L_1),\ldots,I(L_m)\rangle$

(3) if $\pi_I(i)$ is the prefix of the instruction:

$$L_v := F_u^t(L_{k_1},\ldots,L_{k_t})$$

then:

$$\pi_I(i+1) = \pi_I(i) + 1$$

$$A_I(\pi_I)(i+1)(j) = A_I(\pi_I)(i)(j) \quad \text{if } j \neq v, \text{ and}$$

$$A_I(\pi_I)(i+1)(v) = I(F_u^t)[A_I(\pi_I)(i)(k_1), \ldots, A_I(\pi_I)(i)(k_t)]$$

(4) if $\pi_I(i)$ is the prefix of the test:

$$T_u(L_v) \quad b,c$$

then:

$$\pi_I(i+1) = b \text{ if } I(T_u)[A_I(\pi_I)(i)(v)] = 0$$

$$= c \text{ if } \quad " \quad " \quad = 1$$

and

$$A_I(\pi_I)(i+1) = A_I(\pi_I)(i).$$

For any schema P and interpretation I, we define $\underline{val(P_I)}$ as the value of the corresponding execution sequence π_I if this is defined. In this case we shall say that P converges under I, and succeeds or fails when $val(P_I) = 0$ or 1 respectively. Otherwise $val(P_I)$ is undefined, and P diverges under I.

Definition. Two program schemata P,Q are (strongly) equivalent, $P \cong Q$, if, for all interpretations I,

either $val(P_I) = val(Q_I)$

or both P and Q diverge under I.

This is a very stringent relation, but we shall also study various weaker relations, which are all obtained by restricting the range of the quantification in the above definition. For convenience, the more important of these relations are defined here. It will be clear that the domain D of any interpretation may be taken to be countable and, without loss of generality, we may take for D either the set of natural numbers or the set of finite strings over a finite alphabet. A recursive interpretation over D is one for which all the functions assigned to symbols of the schemata are recursive functions over D. A finite interpretation is one which can be defined over a domain of finite cardinal.

Definitions. Two program schemata P,Q are recursively (respectively finitely) equivalent, $P \equiv_r Q$, (respectively $P \equiv_f Q$) if, for all recursive (respectively finite) interpretations I,

either $val(P_I) = val(Q_I)$

or both P and Q diverge under I.

Two program schemata P,Q are weakly equivalent, $P \simeq Q$, if, for all interpretations I,

$val(P_I) = val(Q_I)$ when both are defined.

Note: $\approx$ is not an equivalence relation, since any two schemata are both weakly equivalent to a schema which diverges under all interpretations.

The apparently weaker relations such as weak finite equivalence, ($P \approx_f Q$), obtained by restricting the last definition to finite interpretations, turn out to be the same as weak equivalence. For example,

Lemma. $P \approx_f Q \Rightarrow P \approx Q$

Proof. Suppose there is some interpretation I, for which both P_I and Q_I converge, but P_I succeeds and Q_I fails, then there is clearly some finite interpretation I', for which $P_{I'}$ succeeds and $Q_{I'}$ fails. __/

Thus we have the three implications expressed by:

$$P \equiv Q \Rightarrow P \equiv_r Q \Rightarrow P \equiv_f Q \Rightarrow P \approx Q$$

and we will show later [Theorem 5.2] that none of these implications can be reversed.

§ 2.2 Justification of definitions.

At this point it seems appropriate to explain our choice of definitions for equivalence etc. We have chosen to ignore the final values associated with the locations and to record only the final address. We regard a schema more as a (partially defined) predicate than as a function or transformation. We can show, however, that our definitions are not really any less general, by demonstrating how other definitions and formalisms could be 'simulated' by our schemata.

For example, suppose we select some subset $R = \{ L_{r_1}, \ldots, L_{r_x} \}$ of the locations, and call two schemata R-equivalent, if, under all interpretations, the schemata either both diverge, or both converge with matching final values associated with each location of R. [This definition, with R being the set of all the locations is used by Luckham and Park (7)]. Given any test-symbol T, and new function symbols $F_{r_1}, \ldots, F_{r_x}$, we may append at any terminal address of such a schema P, the sequence of instructions: (viz. p.28 for notation),

$$
\begin{array}{l}
L_{r_1} := F_{r_1}(L_{r_1}) \\
\quad T(L_{r_1})\ e_0,\ ^{\cdot}+1 \\
\quad \vdots \\
L_{r_x} := F_{r_x}(L_{r_x}) \\
\quad T(L_{r_x})\ e_0,\ e_1
\end{array}
$$

to produce the schema P' say. It is clear that, for all P,Q:

P R-equivalent to Q<=> P' ≡ Q'

and that similar results hold for recursive, finite and weak equivalences.

Instead of allowing a schema to halt, we may have it produce 'output' during the course of its computation. An 'output instruction' could be of the form:

OUTPUT := L_j

which leaves L_j unchanged, but records its current value as the next in a sequence, or 'output tape'. Two schemata would be <u>output-equivalent</u> if, under all interpretations, the corresponding output tapes they produce are identical. For the simulation, we introduce a new location N and a new function symbol H, and replace each output instruction of the above form by:

N := H(L_j,N)

T(N) e_0, ˙+1

Again, two schemata are output-equivalent if and only if their transformations are equivalent.

Similarly, suppose we allow 'input instructions' in our formal language, such as:

$$L_j := \text{INPUT}$$

with the effect that L_j takes on the next value from a fixed sequence of values, or 'input tape'. This may be simulated by introducing a new location N' and a new function symbol F', and replacing each instruction of the above form by:

$$L_j := F'(N')$$

$$N' := F'(N')$$

so that the input tape is provided by:

$$f(x),\ \ f(f(x)),\ \ f(f(f(x))),\ \ldots$$

where $f = I(F')$ and $x = I(N')$. By other constructions, many other, apparently new, features (such as statements of the form: '$L_i := L_j$') may be simulated.

Our definitions have their advantages in the ease of presentation of results and proofs, and in their simplicity. We note particularly that only two kinds of instruction are required. (For some purposes we even reduce this number!) The chief disadvantage is that, given two equivalent schemata, it is not usually an equivalence-preserving operation to replace one by the other as a sub-schema of a larger schema. However where we are concerned more with decision problems

than simplification algorithms, this consideration is of less importance.

§2.3 Syntactic characterizations.

Now we establish an alternative, more syntactic, definition of equivalence, which is due to D. Luckham, who proved essentially this result in [BBN Memorandum, October 1962 (unpublished)]. To this end we define, for any schema P and any sequence π through P, the free computation sequence, $A(\pi)$, which is a sequence of m-tuples of strings of symbols, where m is the number of locations used in the schema. $A(\pi)$ is defined inductively by:

(1) $A(\pi)(1) = \langle L_1,\ldots,L_m \rangle$

(2) if $\pi(i)$ is the prefix of the instruction:

$$L_v := F_u^t(L_{w_1},\ldots,L_{w_t})$$

then:

$$A(\pi)(i+1)(j) = A(\pi)(i)(j) \text{ for } j \neq v,$$

and:

$$A(\pi)(i+1)(v) = F_u^t X_{w_1} X_{w_2} \ldots X_{w_t}$$

where X_{w_s} is the string $A(\pi)(i)(w_s)$.

(3) if $\pi(i)$ is a transfer instruction then:

$$A(\pi)(i+1) = A(\pi)(i) .$$

For any sequence π and any transfer symbol T_u, we define two sets of strings $\mathbb{L}_u(\pi)$ and $\mathbb{R}_u(\pi)$ as follows:

Consider the set of all i such that $\pi(i)$ is the prefix of an instruction of the form,

$$T_u(L_v) \quad b,c$$

and $b \neq c$, then,

$$A(\pi)(i)(v) \in \mathbb{L}_u(\pi) \quad \text{if } \pi(i+1) = b,$$
$$\in \mathbb{R}_u(\pi) \quad \text{if } \pi(i+1) = c .$$

Definition. A set of sequences $\pi^{(P)}, \pi^{(Q)}, \pi^{(R)}, \ldots$ through schemata P,Q,R,... are consistent if, for all transfer symbols T_u, the sets:

$$\mathbb{L}_u(\pi^{(P)}) \cup \mathbb{L}_u(\pi^{(Q)}) \cup \ldots .$$

and $$\mathbb{R}_u(\pi^{(P)}) \cup \mathbb{R}_u(\pi^{(Q)}) \cup \ldots .$$

are disjoint.

Theorem 2.1 For any two schemata, $P \equiv Q$ if and only if, for all pairs of sequences $\pi^{(P)}$ and $\pi^{(Q)}$ through P and Q respectively, if $\pi^{(P)}$ and $\pi^{(Q)}$ are consistent then the values of $\pi^{(P)}$ and $\pi^{(Q)}$ are equal or are

both undefined.

Proof. Suppose that $P \equiv Q$ and that $\pi^{(P)}$ and $\pi^{(Q)}$ are consistent. Let I^+ be the interpretation over the domain D^+ of strings of function and location symbols, defined by:

(i) $I^+(L_i) = L_i$ for each L_i,

(ii) $I^+(F_u^t)(x_1,\ldots,x_t) = F_u^t x_1 \ldots x_t$ (i.e. the string obtained by concatenating the symbol F_u^t and the strings $x_1,\ldots,x_t$) for each F_u^t,

(iii) $I^+(T_u)(x) = 0$ if $x \in \mathbb{L}_u(\pi^{(P)}) \cup \mathbf{L}_u(\pi^{(Q)})$

$= 1$ otherwise.

Clearly $\pi_{I^+}{}^{(P)} = \pi^{(P)}$ and $\pi_{I^+}{}^{(Q)} = \pi^{(Q)}$, where $\pi_{I^+}{}^{(R)}$ is the execution sequence in R corresponding to I^+, for R = P,Q. Therefore the values of $\pi^{(P)}$ and $\pi^{(Q)}$ are equal or are both undefined.

Conversely, suppose that $P \not\equiv Q$ and $val(P_I) \neq val(Q_I)$ for some I say. Any interpretation I, defines a natural mapping I^* from the symbol strings which occur in free computation sequences into the domain of I, by:

(i) $I^*(L_i) = I(L_i)$

(ii) $I^*(F_u^t E_1 \ldots E_t) = I(F_u^t)[\, I^*(E_1),\ldots,\, I^*(E_t)\,]$

where $E_1,\ldots,E_t$ are strings.

Clearly, for any execution sequence π_I:

$$I^*(A(\pi_I)\ (i)\ (j)\ ) = A_I(\pi_I)\ (i)\ (j) \text{ for all } i, j.$$

So for each transfer symbol T_u, and all strings E,

$$E \in L_u(\pi_I) \Rightarrow I(T_u)[\ I^*(E)\ ] = 0$$

and similarly for R and 1. Since this holds for both $\pi_I^{(P)}$ and $\pi_I^{(Q)}$, they must be consistent, but we know that they do not have the same value. __/

Corollary. A sequence is consistent if and only if it is the execution sequence of some interpretation.

An interpretation which has the form of I^+ in the above proof is called a free interpretation.

It is natural now to attempt to find 'syntactic' definitions for the other relations defined above. To obtain analoguous results to Theorem 2.1, we must impose some further condition apart from consistency, on the pairs of sequences $\pi^{(P)}$, $\pi^{(Q)}$ considered. For weak equivalence the way is obvious and we state without proof:

Theorem 2.2 $P \simeq Q$ if and only if, for all pairs of finite sequences $\pi^{(P)}$ and $\pi^{(Q)}$ which are consistent, the values of $\pi^{(P)}$ and $\pi^{(Q)}$ are equal.

Definitions. A sequence π through a schema is ultimately periodic if there exist integers $h \geq 0$, $k > 0$, such that, for all $n \geq h$, $\pi(n) = \pi(n+k)$ if these are defined.

Note: any finite sequence is ultimately periodic.

A sequence through a schema P is recursive, if π, regarded as a function from the integers into the set of addresses of P is recursive.

Theorem 2.3 For any schema P and interpretation I,

(i) if I is finite then $\pi_I^{(P)}$ is ultimately periodic,

(ii) if I is recursive then $\pi_I^{(P)}$ is recursive.

Proof. (i) Denote the $(m+1)$-vector,

$$< \pi_I^{(P)}(i), A_I(\pi_I^{(P)})(i) >$$

by $\alpha(i)$. Since the number of addresses of P and the domain of I are both finite, there are only a finite number of distinct values for $\alpha(i)$. Furthermore it is easy to verify that if $\alpha(i) = \alpha(j)$, then $\alpha(i+n) = \alpha(j+n)$ for all $n \geq 0$, so that α is ultimately periodic.

(ii) The definition of π_I is clearly recursive in the functions of I. Hence if these are recursive, so is π_I. ___/

Unfortunately, and perhaps surprisingly, the implications in Theorem 2.3 do not hold in reverse, and the problems of finding syntactic characterizations of the execution sequences of finite and recursive interpretations remain open. An example of a recursive execution sequence which cannot correspond to any recursive interpretation will be produced in §5.1, but a simple example for the other case can be given here.

Schema S_1.

$M_2 := F(M_1)$

a. $L_1 := H(M_1,M_1)$

$L_2 := H(M_1,M_2)$

$M_1 := F(M_1)$

$M_2 := F(M_2)$

$M_2 := F(M_2)$

$T(L_1)$ e_1, a+1

$T(L_2)$ a, e_1

Informal notation.

In the interests of clarity, we employ certain conventions and contractions when we give examples of schemata. Unless otherwise specified, the first instruction is the initial instruction, and the terminal addresses are always denoted by e_0 and e_1 respectively. Prefixes are in general omitted except where transfers are involved,

in which case the prefix is usually denoted by a small letter. The expression ' '+1 ' wherever it occurs as a transfer address denotes the prefix of the following instruction in the schema. The effect of a transfer instruction of the form:

$$T_u(L_i) \quad a, a$$

is independent of u and i and will be called an unconditional transfer. Where it is convenient to avoid the arbitrary choice of u and i we introduce the statement:

goto a

as an abbreviation for such a transfer instruction. Minor points to note are that subscripts and superscripts are often dropped, where this cannot lead to any confusion, and a variety of different capital letters are used as symbols.

As we examine the schema S_1, we observe that there is only one infinite sequence through it. Let I be some finite interpretation with this as its execution sequence and we write:

$$f^0a \qquad \text{for } I(M_1)$$

and $$f^{n+1}a \qquad \text{for } I(F)[f^na],\ n \geq 0.$$

At successive points in the computation sequence which

correspond to the address a of the schema, the successive values associated with the locations M_1 and M_2 are:

$$\begin{array}{ll} f^0a & f^1a \\ f^1a & f^3a \\ f^2a & f^5a \\ f^3a & f^7a \\ \cdot & \cdot \\ \cdot & \cdot \end{array}$$

Since I is finite, not all these values are distinct, so suppose:

$$f^{s+r}a = f^s a \quad \text{for some } r,s > 0$$

Therefore:

$$f^s a = f^{s+r}a = f^{s+2r}a = \ldots = f^{2s+t+1}a$$

for some $t \geq 0$. Hence, putting $u = s+t$:

$$f^u a = f^{2u+1}a$$

When locations L_1 and L_2 are tested for the $(u+1)^{th}$ time, their associated values are:

$$b_1 = I(H)[f^u a, f^u a] \text{ and } b_2 = I(H)[f^u a, f^{2u+1}a]$$

To continue for another loop requires the conditions:

$$I(T)[\, b_1] = 1 \quad \text{and} \quad I(T)[\, b_2\,] = 0$$

but since $b_1 = b_2$, we have a contradiction. Hence S_1 has the affirmed property.

§2.4 'Reasonable' relations and rule-books.

The relation of (strong) equivalence is a natural one and the strongest among the sort of relations we are considering. Weak 'equivalence' is rather less natural, but it seems to be the weakest relation of interest, since it is difficult to reject, on any perfectly general grounds, any interpretation for which both the relevant schemata converge, or more precisely, to reject any consistent pair of finite execution sequences. Finite and recursive equivalence are again more naturally motivated, because we consider, for the former, just those interpretations which are theoretically realizable by a finite automaton (such as a digital computer with a limited amount of storage), and for the latter, just those which could be realized by a Turing machine (or a digital computer with unrestricted storage space). Many other relations, which fall between strong and weak equivalence, could be of interest, and it is convenient to be able to prove, at one stroke, theorems about all such relations.

Definition. A relation $\sim$ on program schemata is reasonable, if, for all P,Q:

(i) if $P \sim Q$ then $P \approx Q$

(ii) if $P \equiv Q$ then $P \sim Q$

Weak equivalence is an intransitive reasonable relation. The relation $\prec$, defined by:

$$P \prec Q \text{ if } \quad val(P_I) = val(Q_I) \text{ whenever } val(P_I) \text{ is defined}$$

is an example of an asymmetric reasonable relation. Finite and recursive equivalences are, of course, reasonable equivalence relations.

We may expect any practical algorithm for improving the efficiencey of computer programs to include a set of rules which may be applied locally, to some part of the program, to produce some improvement or rearrangement. We will give a few illustrations of such rules applied to schemata. In each case it is easy to see that the transformation results in an equivalent schema.

(I). Permutation of computation instructions.

$$\begin{array}{ll} L_1 := F(L_2) & \longleftrightarrow \quad L_3 := G(L_2) \\ L_3 := G(L_2) & \qquad\quad L_1 := F(L_2) \end{array}$$

(II). Deletion of 'vacuous' instructions.

$$\begin{array}{ll} L_1 := F(L_2) & \\ L_1 := G(L_2) & \longrightarrow \quad L_1 := G(L_2) \end{array}$$

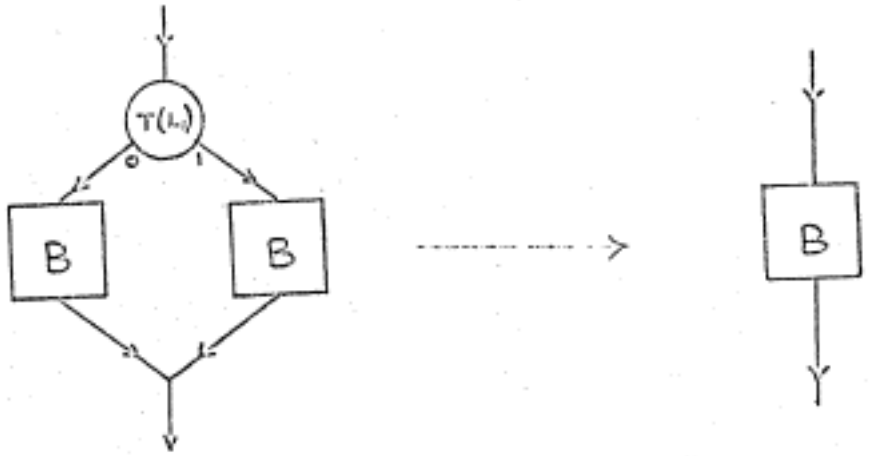
(III). Substitution and deletion.

$L_2 := F(L_1)$ ⟶ $L_3 := F(L_1)$

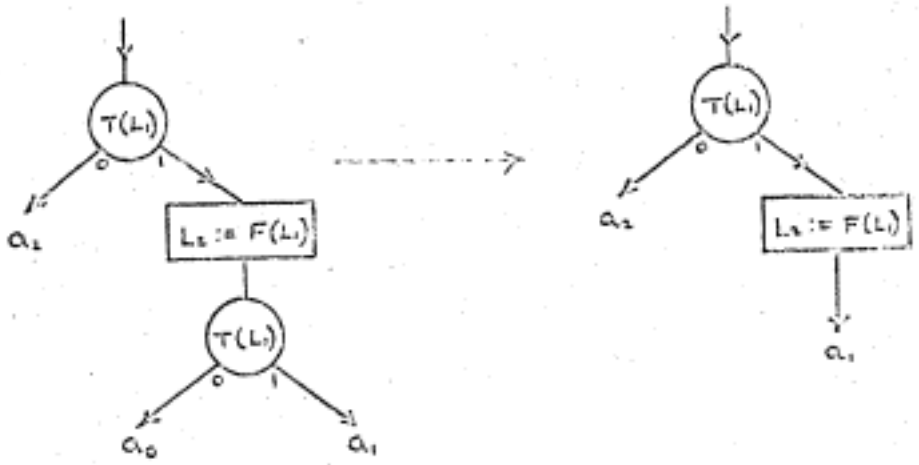
$L_3 := F(L_1)$ $\qquad$ $L_2 := G(L_3)$

$L_2 := G(L_2)$

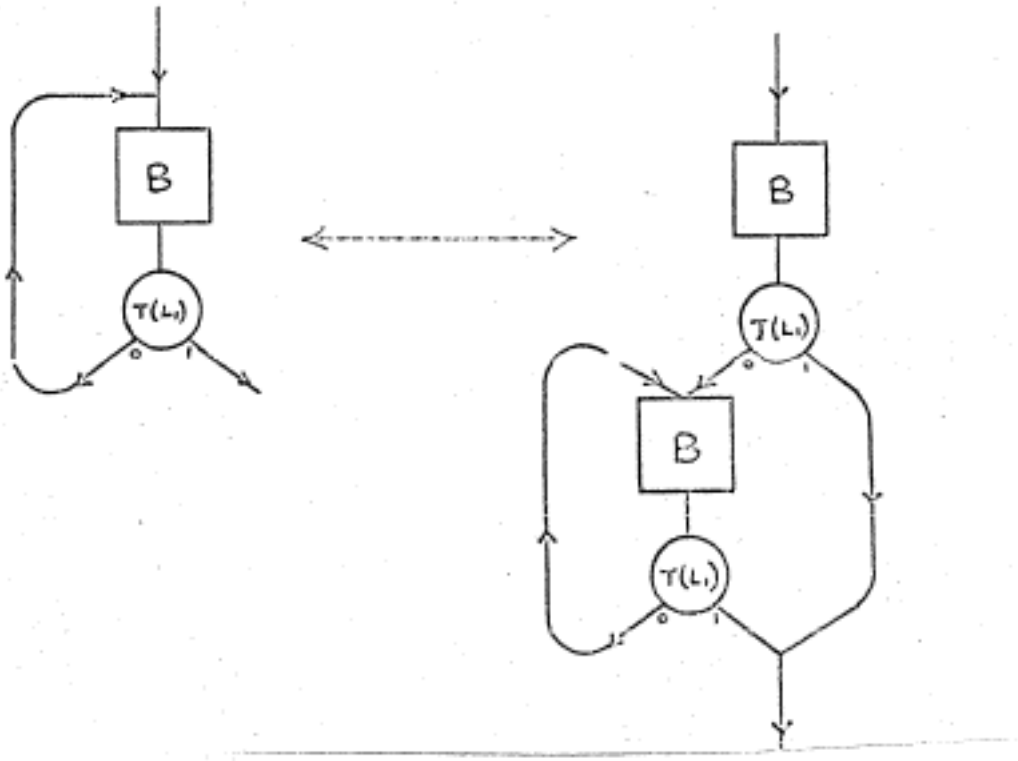
(IV). Removal of redundant tests.



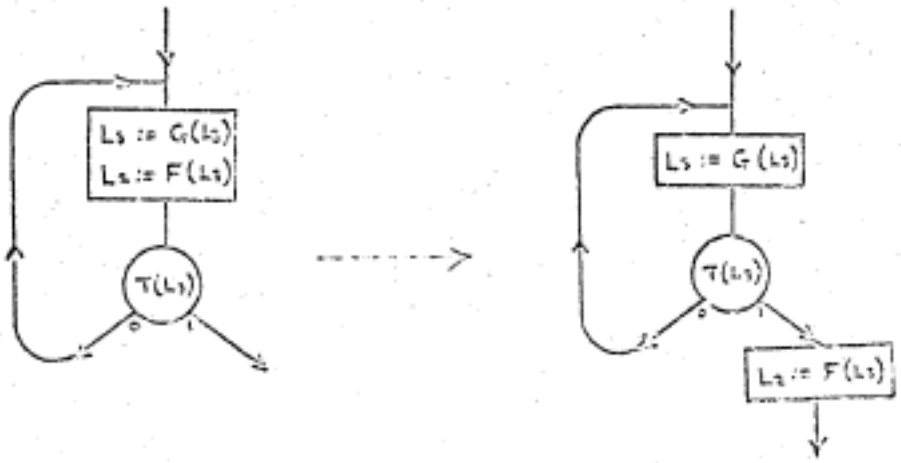
(v). Removal of repeated tests.



(VI). 'Unwinding' loops (or the reverse.)



(VII). 'Tightening' of loops.



As we will see in retrospect, it is inappropriate to consider such transformations in more detail at present. If $\sim$ is any relation between schemata, a $\sim$-rule is a recursive procedure, which, when applied to a schema P, produces if anything a schema Q such that $P \sim Q$. We are only interested in $\sim$-rules when $\sim$ is a reasonable and transitive relation, for then we can apply a sequence of such rules and produce at each stage, a schema which $\sim$-related to the original schema. The transformations given above are all simple Ξ-rules, but we can have very much more complicated rules such as the (possibly non-terminating) construction of E(P) described in §3.2. A rule-book for $\sim$ is a finite set of $\sim$-rules; and we will want to see to what extent a rule-book can serve to produce a 'simplest' schema $\sim$-related to a given schema. Firstly though we show how a 'direct' decision procedure and simplification algorithm can be obtained for a special class of schemata.

§3. SOME SIMPLE EQUIVALENCE PROBLEMS.

§3.0 Marill [8] has considered some of the problems of simplification and storage minimization for schemata with input and output statements but with no transfer instructions. We now give a complete decision procedure for the equivalence of schemata of a more general type, schemata without loops, and show that this can be extended to cover any schemata which always halt. We observe that, for such schemata, strong and weak equivalence, and hence all reasonable relations, are the same, so the decision problem for any reasonable relation is solvable. It will be evident that a 'canonical form' for schemata without loops can be obtained and we do not pursue this further. The section ends with an unsolved problem presented by a certain simple schema.

3.1 Loop-free schemata.

A loop-free schema is one for which no sequence can contain the same address twice. For these schemata, the equivalence problem is, in a sense, finite, and fairly easily solvable, so we will give only a sketch of the solution together with an illustrative example.

Since all sequences through a loop-free schema P, are bounded by the size of P, there are only a finite number which go from the initial instruction to the left-terminal address e_0. For any such sequence π, we determine the finite sets $L_u(\pi)$ and $R_u(\pi)$, for all test symbols T_u. Any π then, corresponds to a conjunction of conditions, each of which may be expressed in the form:

$$T_u(E_j) = \varepsilon$$

where E_j is a string of symbols and ε is 0 or 1. The condition (on the interpretation) that P_I succeeds is a disjunction of such conjunctions, with one conjunction corresponding to each successful sequence. If we write just:

$$T_u(E_j) \quad \text{for} \quad T_u(E_j) = 0$$

and

$$\neg T_u(E_j) \quad \text{for} \quad T_u(E_j) = 1$$

then the disjunction, let us call it prop(P), appears

as a sentence of the propositional calculus with the '$T_u(E_j)$' as atomic formulae.

Since all the functions are independent, any assignment of truth-values to the atomic formulae is the assignment corresponding to some interpretation, and P succeeds under this interpretation if and only if the value of prop(P), after substituting in these truth-values, is 'true'. Given two such sentences, prop(P) and prop(Q);

$$\text{prop(P)} \iff \text{prop(Q)}$$

is a theorem of the propositional calculus, if and only if they take the same truth-value for all such assignments. Hence:

Theorem 3.1 The equivalence problem for loop-free schemata is solvable.

For an example of the application of the solution described, we consider the schema S_2, given below, whose flow-diagram is given in a self-explanatory form in fig. 1.

Schema S_2.

a_1. $L_1 := F(L_1, L_2)$
$L_3 := G(L_2)$
$T(L_1)$ a_3, a_2 E_1

a_2. $T(L_3)$ a_4, a_5 E_2

a_3. $L_3 := F(L_1, L_3)$
$T(L_3)$ a_4, e_1 E_3

a_4. $L_2 := G(L_2)$
$T(L_2)$ a_7, e_1 E_2

a_5. $L_3 := F(L_1, L_3)$
$T(L_3)$ e_1, a_6 E_3

a_6. $L_2 := G(L_2)$

a_7. $L_3 := F(L_1, L_2)$
$T(L_3)$ e_0, e_1 E_3

The sequences from a_1 to e_0 are specified by:

P_1 :- a_1, a_3, a_4, a_7, e_0

P_2 :- a_1, a_2, a_4, a_7, e_0

P_3 :- $a_1, a_2, a_5, a_6, a_7, e_0$

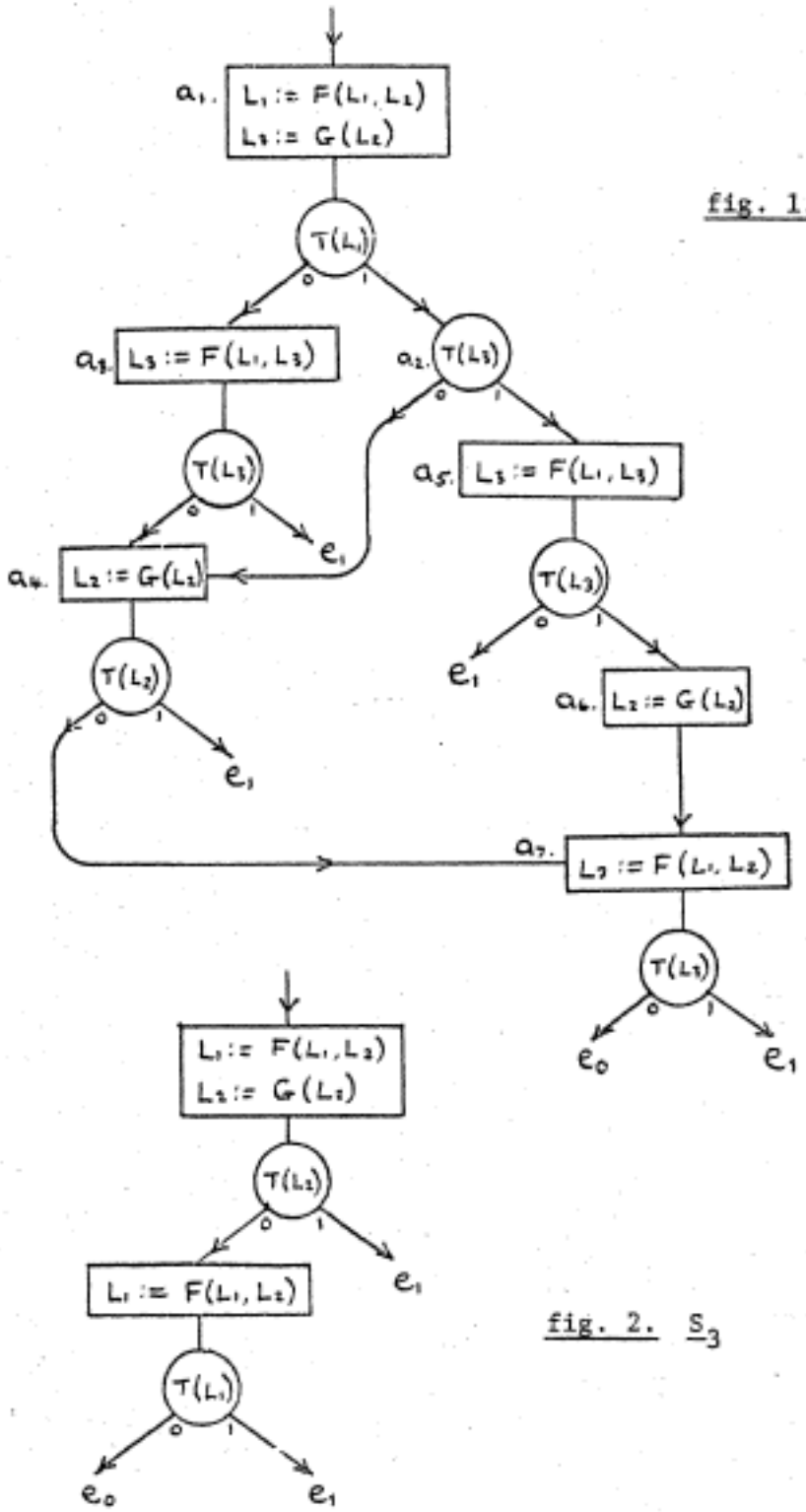


fig. 1. S_2

fig. 2. S_3

Let $E_1 = FL_1L_2$, $E_2 = GL_2$, and $E_3 = FFL_1L_2GL_2$.

prop(S_2) is then (the notes to the right of the schema may bo of assistance):

$$T(E_1) \wedge T(E_3) \wedge T(E_2) \wedge T(E_3)$$

$$\vee \quad \neg T(E_1) \wedge T(E_2) \wedge T(E_2) \wedge T(E_3)$$

$$\vee \quad \neg T(E_1) \wedge \neg T(E_2) \wedge \neg T(E_3) \wedge T(E_3)$$

which simplifies to:

$$T(E_2) \wedge T(E_3)$$

S_2 is therefore equivalent to the schema S_3 given as:

$L_1 := F(L_1,L_2)$

$L_2 := G(L_2)$

$T(L_2)$ $\quad$ $+1$, e_1

$L_1 := F(L_1,L_2)$

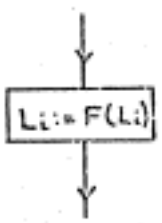
$T(L_1)$ $\quad$ e_0, e_1

The flow-diagram of S_3 is shown in fig. 2.

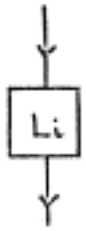
§3.2 Schemata which always converge.

Let $\mathbf{C}$ be the class of schemata which converge under all interpretations. We shall give an effective method of constructing, from any schema in $\mathbf{C}$, a finite loop-free schema which is equivalent to it. This solves the equivalence problems for $\mathbf{C}$, since we solved the general problem for loop-free schemata in §3.1

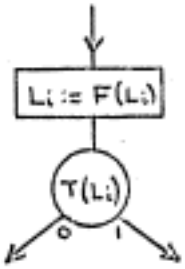
Any schema may be regarded as a directed graph, its flow-diagram, in a natural way. To each computation instruction there corresponds a node of the flow-diagram with one branch leaving it, and to each transfer instruction, a node with two branches leaving. Given any flow-diagram P, we define the lift of P, L(P), which is a tree with a copy of the initial node of P as the root, and for every node N_0 in P with successor(s) N_1 (and N_2), every copy of N_0 in L(P) has copies of N_1 (and N_2) as successor(s) in the same way. Unless P is loop-free, L(P) is infinite. We illustrate the construction of L(P) by an example, (figures 3 and 4 below). Part of the (infinite) lift of a schema S_4 is shown and dotted lines indicate where parts are omitted. In these figures, and in the remaining figures of this section, we abbreviate:



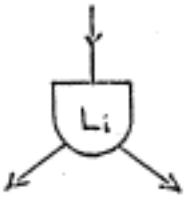
by:



and:



by:



We can 'prune' away parts of the tree L(P), replacing conditional transfers by unconditional transfers wherever certain branches are inaccessible from the root under any interpretation, leaving an equivalent flow-diagram. In fig. 4. such branches are crossed by double lines, as for instance the first time that location L_1 is tested, the value associated with it is one that has previously been tested in L_2 and found to have test-value 0, so that the crossed branch can never be selected. The sub-tree of L(P) remaining when all such excisions are made, we call the execution tree of P E(P). For the schema S_4 the execution tree happens to be finite and is presented in fig. 5.

fig. 3. Flow-diagram of P.

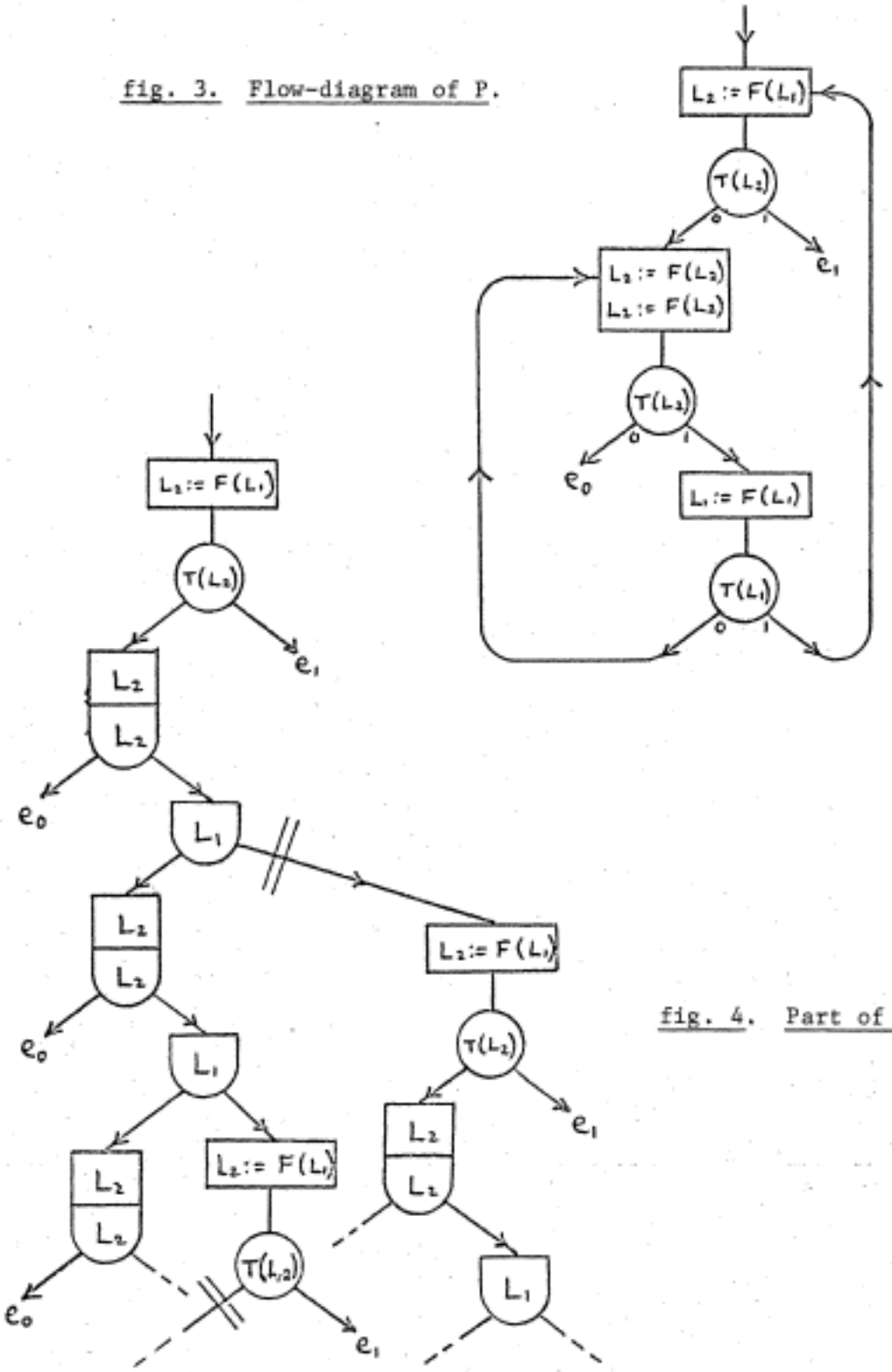


fig. 4. Part of L(P).

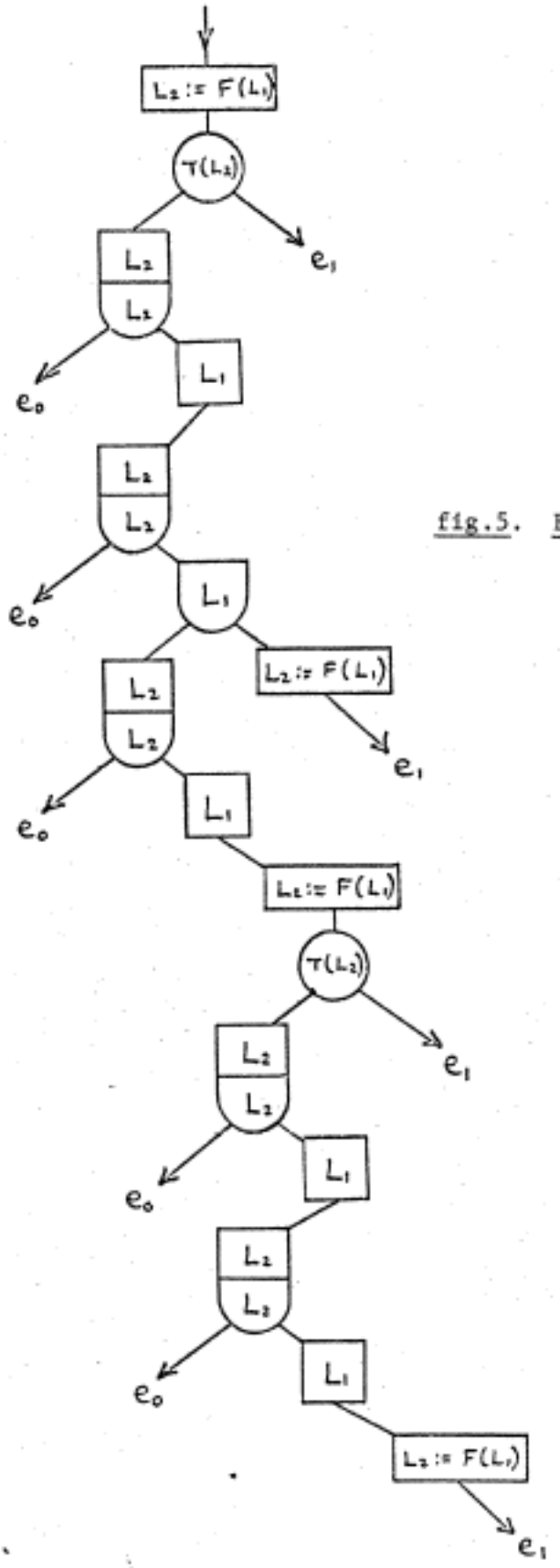

L2 := F(L1)
T(L2)
e1
L2
L2
L1
e0
L2
L2
L1
e0
L2 := F(L1)
L2
L2
e1
L1
e0
L2 := F(L1)
T(L2)
e1
L2
L2
L1
e0
L2
L2
L1
e0
L2 := F(L1)
e1

fig.5. E(P).

The construction of E(P) for a general schema P, is described now in more detail. Given the lift of P, we mark the initial node. The construction is then carried out in steps until, maybe, no further step is possible. At any step, we select any unmarked node N, whose predecessor is marked. If N is a computation node or a terminal node, we immediately mark it and the step is complete. If N is a test node, we examine the (unique, finite) path from the initial node to N and determine what expression (of the free computation sequence) is being tested at N. We further determine whether that expression has been calculated and tested by the same test function at any previous node on that path. If not, N is marked; if so, then by the constuction it will have been tested just once and we can anticipate the result this time. We replace N by a marked 'empty' node and delete the inaccessible succeeding node and its following sub-tree. The step is then complete.

Suppose the construction does not terminate, then the execution tree contains infinitely many nodes which are accessible from the initial node, b_0 say. One or other of the successors of b_0, say b_1, must also have an infinite number of nodes accessible from it. Similarly some b_2, a successor of b_1, and some b_3, a successor of

b_2, and so on, each have an infinite number of nodes accessible from them. Therefore there is an infinite path $b_0, b_1, b_2, b_3, \ldots$ in the execution tree. (This is an application of the well-known Infinity Lemma of graph theory.) By the construction, any path in this tree is consistent and so, by the corollary to Theorem 2.1, is an execution sequence of P. Therefore, if P belongs to $\mathbb{C}$, the construction of E(P) must always terminate. In this case, there is an effective, finite, procedure which produces the execution tree corresponding to P, and it is then easy to produce from this, a schema in conventional form, by ignoring empty nodes and introducing unconditional transfers where necessary. E(P) is equivalent to P and loop-free, and we have a decision procedure for the equivalence of such schemata, (Theorem 3.1). Hence:

Theorem 3.2 The equivalence problem for schemata which converge under all interpretations is solvable.

We conclude this section by demonstrating that, even in apparently simple cases, it is not always a trivial matter to determine whether or not a given schema is in the class $\mathbb{C}$. The following schema always converges, but has an execution sequence of length 147. (see fig. 6)

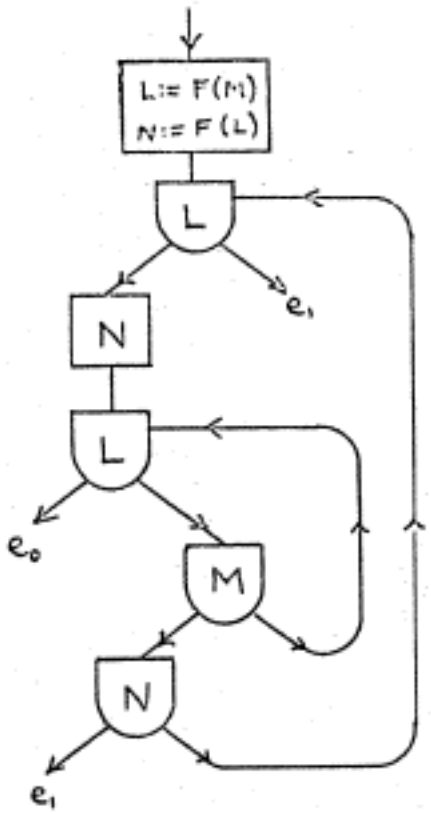


fig. 6.

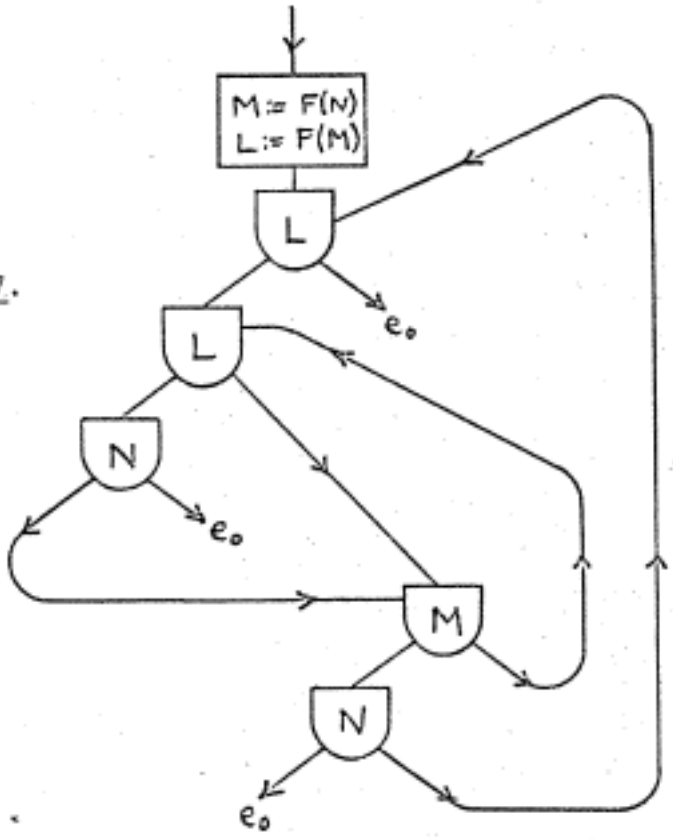


fig. 7.

The schema in fig. 7. seems likely 'on general grounds' to be in **C**, but execution sequences of lengths greater than 600 have been found. I have been unable to settle this matter, which is left as an exercise for the reader.

PART II. UNSOLVABLE PROBLEMS.

Luckham and Park [7] first proved the unsolvability of the equivalence problem for program schemata and pointed out that this precluded the possibility of there being any 'completely adequate' simplifying transformation. Their proof was rather less direct than that presented here and involved the use of many locations. R. Floyd [unpublished communication] also has recently produced an elegant 'indirect' proof of the above result.

We prove somewhat stronger results about other reasonable relations and for schemata satisfying various conditions. Our unsolvability results are proved for schemata with only two locations and in Part III we shall see that this number is the best possible.

In the next section we apparently digress to consider a variety of finite automata.

§4. TWO-TAPE AND TWO-HEADED AUTOMATA

§4.0 Our unsolvability proofs for schemata depend on a class of schemata which can be regarded as 'simulating' two-headed finite automata. (The reader will have already had a taste of this simulation in following the execution of the schemata given in §3.) Although we could preceed independently of such analogues, it seems more natural and gives us greater versatility of exposition to derive our main theorems initially in this guise. Besides, some of our results take on an extra interest in their 'machine' form.

Rosenberg [14] has independently obtained several interesting results on multi-headed automata, including what is essentially our Theorem 4.2. The questions considered by Elgot and Rutledge [3] are related but do not seem to be directly applicable in the context of schemata.

In this section we establish the basic theorems on the correspondence with Turing machines and prepare for the similation of two-headed automata by program schemata. Theorem 4.4 could perhaps be endowed with some philosophical significance.

§4.1 Two-tape automata

Rabin and Scott in [13] investigated two-tape, one-way finite automata. Such a two-tape automaton M, is a finite automaton equipped with two scanning-heads each of which reads its own input tape, one symbol at a time. With each internal state of M (except the terminal state or states) is associated an integer specifying the tape from which the next symbol is to be read, and a transition function giving, for each tape-symbol read, the next state of M to be entered. One state of M is designated as the initial state. The reader is referred to [13] for definitions given in more detail.

Our treatment differs in that we dispense with special 'end-markers' on the tapes and consider the behaviour of automata presented with pairs of infinite tapes.

For any automaton M and terminal state s, we define $T_s(M)$ as the set of pairs of (finite) sequences of input symbols, which take M from the initial state to the state s. Often the automaton M, will have just two terminal states, denoted by a and r, then, it is convenient to say that M accepts those pairs of tapes with initial segments in $T_a(M)$, and rejects those with initial segments in $T_r(M)$. M will diverge on the set $T_D(M)$ of those pairs of tapes for which M never enters a terminal state.

We shall present particular automata by specifying all or part of the transition function either in a table or illustrated in a flow-diagram. In the latter the initial state is indicated by an arrow which does not originate at any of the nodes, and the number in parentheses at a node indicates which tape is to be read for the next symbol. For example:

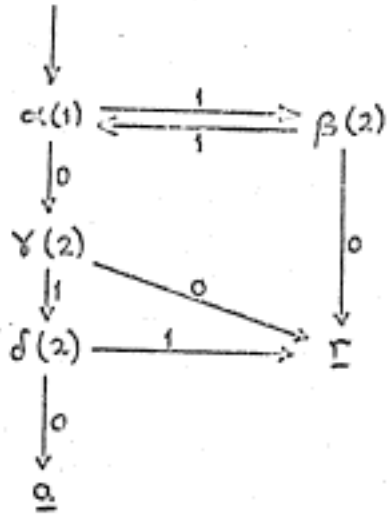


fig. 1. Flow-diagram of M_1.

It may be verified that:

$$T_{\underline{a}}(M_1) = \{ < 1^m0, 1^{m+1}0 > \mid m \geq 0 \}$$

$$T_{\underline{r}}(M_1) = \{ < 1^{m+1}, 1^m0 >, <1^m0, 1^m0 >, < 1^m0, 1^{m+2} > \mid m \geq 0 \}$$

$$T_D(M_1) = \{ < 1^\infty , 1^\infty > \}$$

We now describe the construction of an important class of two-tape automata.

Lemma 1. Given any Turing machine U, there is a two-tape automaton M_U, with terminal states n, a and r, such that, if c is a configuration of U, and c' is the succeeding configuration (see §1),

$$< c, c' > \varepsilon\ T_{\underline{n}}(M_U) \text{ if } c \text{ is not terminal,}$$

$$< c, c' > \varepsilon\ T_{\underline{a}}(M_U) \text{ if } c \text{ is terminal,}$$

and if the second sequence does not start with c', the pair is rejected.

Proof. We outline the operation of M_U. Given a configuration under Head 1, it checks that the tape under Head 2 is identical, except for an extra 0 symbol at the left end of the (Turing machine) tape, until some state symbol of U is encountered on either tape. M_U then verifies that the difference between the configurations is compatible with the rules of U, and finally checks that the remainders of the configurations are identical up until the final terminating symbol ε, except again for an additional 0 on the second tape. The transition table for M_U is given in fig. 2. Where a

fig.2 Transition table for a 2-tape automaton M_U, to check successive configurations of a Turing machine U.

state	head	symbol	new state	
$[a_j, -]$	2	a_j	$[-, -]$	
		q_i	$[a_j,q_i]$	
$[-, -]$	1	a_j	$[a_j, -]$	
		q_i	$[q_i, -]$	
		ε	$[\varepsilon, -]$	
$[a_j,q_i]$	2	a_j	$[x,q_ix]$	
$[x,q_rx]$	1	q_i	$[xq_i,q_rx]$	
$[q_i, -]$	1	a_j	$[q_ia_j,-]$	
$[q_ia_j,-]$	2	a_j	$[q_ia_j,a_j]$	
		q_r	$[q_ia_j,q_r]$	
$[xq_i,q_rx]$	1	a_j	$[a_j, -]$	if ' q_i a_j L q_r ' is in U.
$[q_ia_j,a_j]$	2	q_r	$[-, -]$	if ' q_i a_j R q_r ' is in U.
$[q_ia_j,q_r]$	2	a_k	$[-, -]$	if ' q_i a_j a_k q_r ' is in U.
		a_j	$[-, -]$*	if ' q_i a_j H q ' is in U, i = r.
$[-, -]$*	1	a_j	$[a_j, -]$*	
		ε	$[\varepsilon, -]$*	
$[a_j, -]$*	2	a_j	$[-, -]$*	
$[\varepsilon, -]$*	2	a_0	$[\varepsilon, 0]$*	
$[\varepsilon, 0]$*	2	ε	a	
$[\varepsilon, -]$	2	a_0	$[\varepsilon, 0]$	
$[\varepsilon, 0]$	2	ε	n	

Notes.

1. The initial state is $[a_0, -]$ (where $a_0 = 0$).
2. Transitions not given are transitions to r.
3. The quantification 'for all i,j,r ' is supposed for each entry here.

transition is not given, a transition to the terminal state r is to be assumed. The names of the states are supposed to have mnemonic value, and the proof that M_U has the required properties is by inspection. __/

§4.2 Two-headed automata.

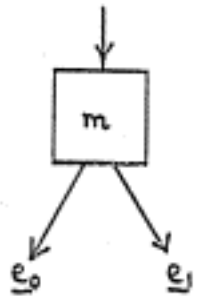
A two-headed (one-tape) automaton is identical with a two-tape automaton except that the two scanning heads are regarded as reading independently the same input tape. Alternatively, we can regard it as a two-tape automaton which is always presented with identical pairs of tapes. For any two-tape automaton M with terminal states a and r, the corresponding two-headed automaton M' is said to accept (respectively reject, diverge on) those tapes t such that:

$$< t_1, t_2 > \varepsilon\ T_a(M) \quad \text{(respectively } T_r(M),\ T_D(M)\ \text{)}$$

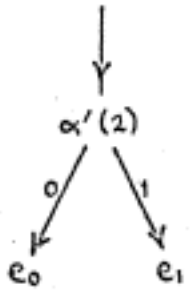
for some initial segments t_1, t_2 of t.

Given a number of two-tape (similarly two-headed) automata $M_1, M_2, \ldots$, we can construct a new automaton M, by replacing terminal states of the M_i, where they occur

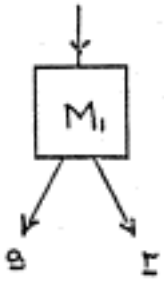
in transition tables, by either new terminal states or the initial states of some M_i. The initial state of one of the M_i is designated as the initial state of M. For example, if:



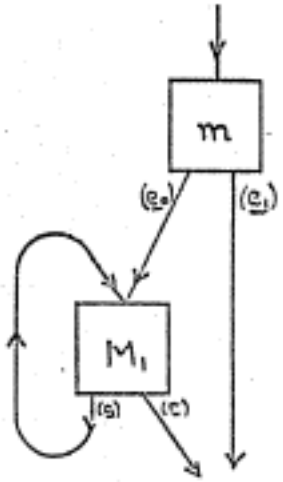
denotes



and



is the automaton of fig. 1., then:



denotes

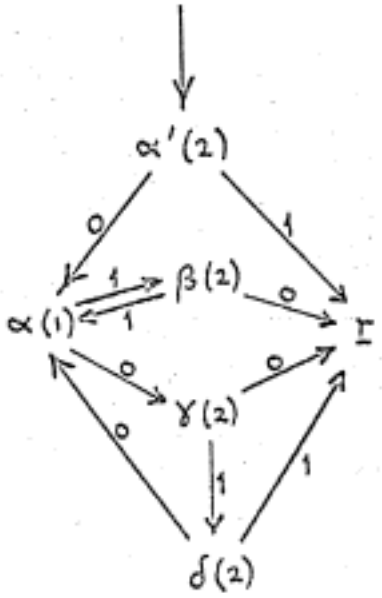


fig. 3. F

This automaton F, could of course be simplified by taking δ as the initial state and deleting the state α'. F, regarded as a two-headed automaton, can only diverge on the tape:

01011011101111011111101111110.....

The behaviour of F on such a tape is that, whilst Head 1 is scanning each sequence of 1's, Head 2 is scanning the succeeding sequence and, unless this contains precisely one more 1, the state r is reached.

Theorem 4.1 There is a two-headed automaton F, which diverges on some tape but rejects any ultimately periodic tape.

A consideration of the modus operandi of F serves as an introduction to Lemma 2. Given any Turing machine U, and initial configuration c (= C(0)), let C(1), C(2), C(3),.... be the sequence of successive configurations.

Lemma 2. There is a two-headed automaton $M_{U,c}$ which accepts just those tapes with an initial segment of the form:

C(0) C(1) C(2) C(3) .... C(n) C(n+1)

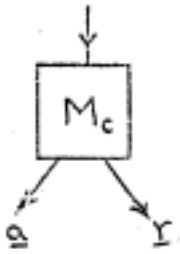
where C(n) is a terminal configuration of U (and the first in the sequence). If machine U does not terminate

when started from configuration c, then $M_{U,c}$ cannot accept any tape, but diverges on the tape:

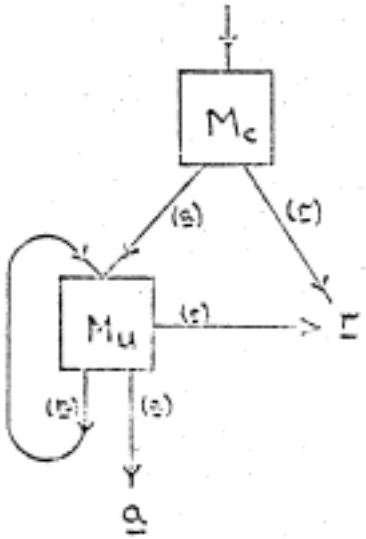
C(0) C(1) C(2) C(3) ....

Otherwise $M_{U,c}$ never diverges.

Proof. It is a simple matter to construct, for any configuration c, an automaton which reads only from Head 2, and accepts only the sequence c, rejecting anything else. We denote this by:



We combine this with the automaton M_U described above, to produce $M_{U,c}$ in the following way:



Because of the operation of M_c, any tape which does not commence with C(0) is rejected. Head 1 then scans this C(0) while Head 2 reads the next part of the tape. The properties of M_U ensure that if this next part is not C(1), the tape is rejected. After this stage, Head 1 is at the start of C(1) and Head 2 is again ready to scan the next part of the tape. While Head 1 is scanning any configuration, Head 2 scans the subsequent part to check that it represents the succeeding configuration. These operations continue until a check fails or until a terminal configuration is read by Head 1. ___/

Therrem 4.2 For any Turing machine U, there is an effective construction of a two-headed automaton A_U with the following properties:

(i) if U halts when started on a blank tape, there is no tape on which A_U can diverge but it can accept some tape,

(ii) If U diverges when started on a blank tape, A_U can diverge but cannot accept any tape,

(iii) A_U cannot diverge on any ultimately periodic tape.

Proof. If $c' = q_1 0\varepsilon$, then A_U is the automaton $M_{U,c'}$, whose construction is clearly effective from U, and

which, by Lemma 2, satisfies (i) and (ii). Property (iii) is satisfied because A_U can only diverge on a tape which represents a sequence of successive configurations. On such tapes however, the end-of-configuration symbol ε, occurs only at steadily increasing intervals, and so no such tape can be ultimately periodic. __/

With the help of Theorem A, we immediately derive:

Theorem 4.3 For two-headed automata A, the properties:

(i) A accepts no tape, [$T_{\underline{a}}(A) = \emptyset$],

(ii) A diverges on some tape, [$T_D(A) \neq \emptyset$],

are not partially solvable, even when restricted to automata which converge on all ultimately periodic tapes.

We remark that the negations of these predicates are both partially solvable. This is obvious for:

$$T_{\underline{a}} \neq \emptyset$$

For the other, the method of proof is very similar to that used for the corresponding result about program schemata. There is a procedure which generates, for any A, a tree whose nodes are states of A, such that any sequence of states describing an initial portion of the behavior of A on some tape, corresponds to some

path in the tree starting from the root, and vice versa. There are at most finitely many outgoing branches at each node, so that, if A never diverges and all the paths in this tree are finite, the infinity lemma assures us that there are only a finite number of nodes in the tree. In this case the generating procedure terminates, and so provides, in general, a partial decision procedure for the property:

$$T_D(A) = \emptyset.$$

§4.3 Recursive tapes.

The next four lemmata prepare the way for an even stronger theorem, in which we replace 'ultimately periodic' above by 'recursive'. A tape is recursive if the sequence of symbols is given by a recursive function from the positive integers to some finite alphabet.

Let $\underline{n}$ denote a sequence of n 1's followed by a 0.

Lemma 3. There is a two-tape automaton N which, for arbitrary $n \geq 0$, when presented with a sequence:

$$\underline{n}\ \underline{n-1}\ \ldots\ \underline{1}\ \underline{0}$$

under Head 1, will accept only the sequence:

$\underline{n+2}\ \underline{n+1}\ \ldots\ \underline{3}\ \underline{2}\ \underline{1}\ \underline{0}$

under Head 2, rejecting all others.

Proof. N is illustrated. Any transition not shown is to r.

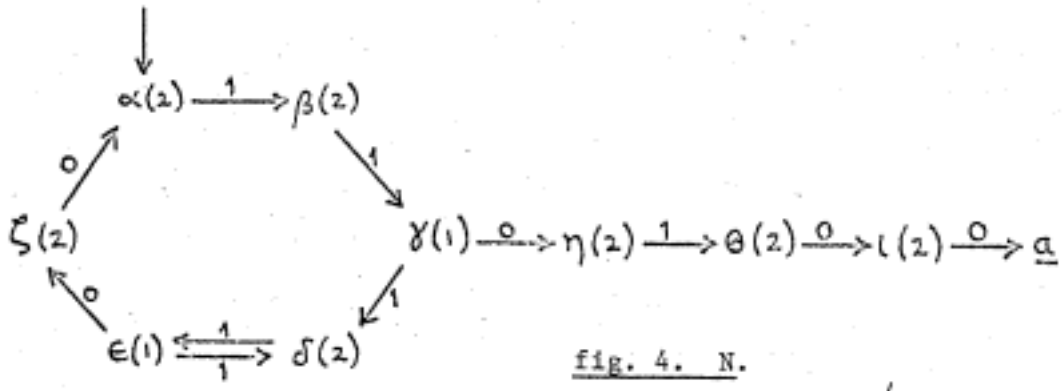


fig. 4. N.

__/

Let U be some fixed, universal Turing machine, (Theorem B,§1). We denote by ' C_{n0} ', the initial configuration of U started on n, and by ' C_{nm} ', the configuration resulting after m steps of the computation.

Lemma 4. There is a two-tape automaton C_0, which, given the sequence n under Head 1, can accept either of the sequences:

$$a\ C_{n0}\ 0\ b \qquad \text{or} \qquad a\ C_{n0}\ 1\ b$$

where a,b are new symbols, and rejects any other.

Proof. The construction of C_0 is easy. __/

Lemma 5. There is a two-tape automaton M_U^*, which, given the sequence:

$$C_{nm}\delta \quad \text{for arbitrary } n,m \geq 0, \text{ and } \delta = 0 \text{ or } 1$$

under Head 1, can accept the sequence:

$$C_{n(m=1)}\delta$$

under Head 2, just when C_{nm} is non-terminal, or when C_{nm} represents a halt on the symbol δ (= 0,1), and rejects anything else.

Proof. It is a simple matter to modify the automaton M_U, of Lemma 1, so that the terminal state $\underline{a}$, corresponding to a terminal configuration, is replaced by two states, $\underline{a}_0$ and $\underline{a}_1$, which are reached according to whether the configuration under Head 1 represents a halt on a 0 or a 1 respectively. If this new automaton is M_U', then M_U^* is given by:

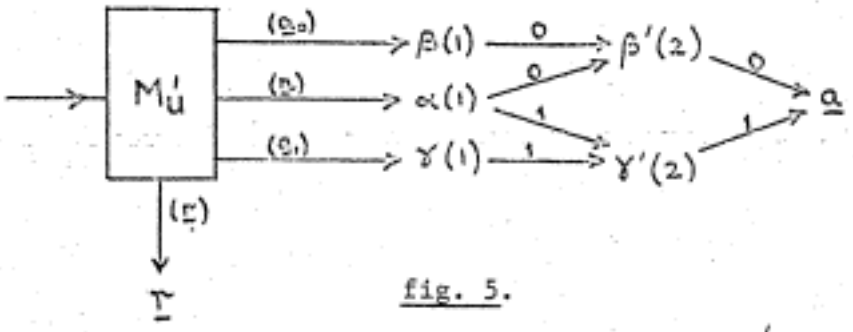


fig. 5.

/

Lemma 6. If U is a universal Turing machine, as described in Theorem B, there is no recursive function f, such that:

$$C[U,n] = 0 \Rightarrow f(n) = 0$$

and $$C[U,n] = 1 \Rightarrow f(n) = 1$$

Proof. Suppose such an f is recursive, then, by Theorem D, there is a machine M, and say $M = M_m$, the mth in the enumeration, such that:

$$C[M_m,n] = 1 \text{ if } f(n) = 0$$

and $$C[M_m,n] = 0 \text{ if } f(n) \neq 0$$

Then:

$$f(m) = 0 \Rightarrow C[M_m,m] = 1$$
$$\Rightarrow C[U,m] = 1$$
$$\Rightarrow f(m) = 1$$

and: $$f(m) \neq 0 \Rightarrow C[M_m,m] = 0$$
$$\Rightarrow C[U,m] = 0$$
$$\Rightarrow f(m) = 0$$

These contradictions prove the lemma. ___/

This result is of course well-known, but proved here for convenience.

Consider now the two-headed automaton R, whose flow-diagram is given below (with transitions to r suppressed).

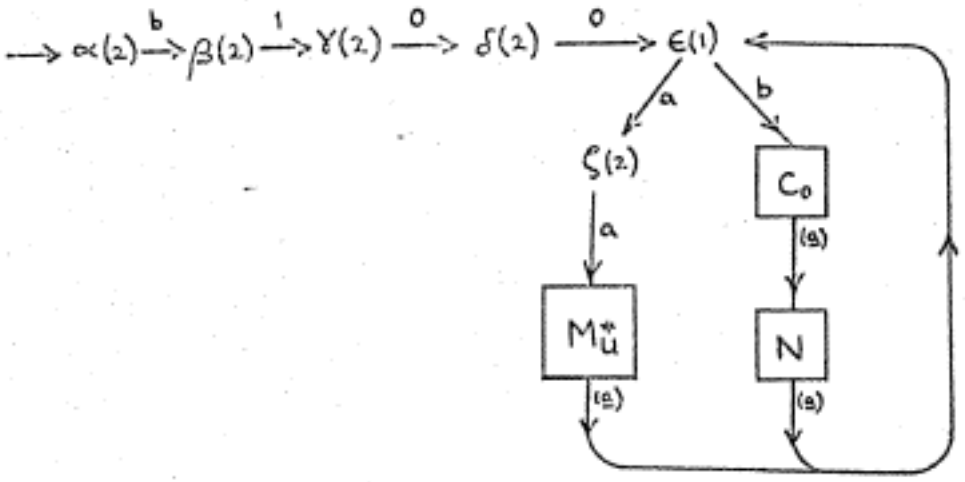


fig.6. Flow-diagram of R.

R rejects any tape not of the form:

$$b\ \underline{1}\ \underline{0}\ a\ C_{10}\ \delta_1\ b\ \underline{2}\ \underline{1}\ \underline{0}\ a\ C_{11}\ \cdots\cdot$$

which is more conveniently displayed as:

$$b\ \underline{1}\ \underline{0}$$

$$a\ C_{10}\ \delta_1\ b\ \underline{2}\ \underline{1}\ \underline{0}$$

$$a\ C_{11}\ \delta_1\ a\ C_{20}\ \delta_2\ b\ \underline{3}\ \underline{2}\ \underline{1}\ \underline{0}$$

$$a\ C_{12}\ \delta_1\ a\ C_{21}\ \delta_2\ a\ C_{30}\ \delta_3\ b\ \underline{4}\ \underline{3}\ \underline{2}\ \underline{1}\ \underline{0}$$

$$a\ C_{13}\ \delta_1\ a\ C_{22}\ \delta_2\ a\ C_{31}\ \delta_3\ a\ C_{40}\ \delta_4\ b\ \underline{5}\ \underline{4}\ \underline{3}\ \underline{2}\ \underline{1}\ \underline{0}$$

$$a\ C_{14}\ \delta_1\ a\ C_{23}\ \delta_2\ a\ C_{32}\ \delta_3\ a\ C_{41}\ \delta_4\ a\ C_{50}\ \delta_5\ b\ \underline{6}\ \underline{5}\ .\ .$$

where $\delta_i = 0$ or 1 for $i = 1,2,\ldots$

Furthermore, M^*_U is designed to ensure that R can only diverge on such a tape if the sequence:

$$\delta_1,\delta_2,\delta_3,\ldots$$

is such that the function f, defined by:

$$f(n) = \delta_n \quad \text{for } n = 1,2,3,\ldots$$

satisfies the conditions described in Lemma 6. Suppose that R diverges on some tape defined by a recursive function, g say. The sequence of δ's is recursive in g, to be precise:

$$f(n) = \delta_n = g(2/3(n^3+6n^2+8n))$$

Therefore f is recursive, which contradicts Lemma 6.

Hence:

Theorem 4.4 There is a two-headed automaton R, which rejects any recursive tape, but can diverge on some (non-recursive) tape.

For any Turing machine M, we can modify R so that tapes it does not reject have the form of the triangular representation above, except that at the beginnings of successive rows there appear the successive configurations corresponding to M started on blank tape. This new automaton checks that the tape has this form, but accepts the tape if a terminal configuration of M is ever reached. Again, it cannot diverge on any recursive tape, so we have:

Theorem 4.5 For any Turing machine U, there is an effective construction of a two-headed automaton R_U, such that:

(i) if U halts from a blank tape, there is no tape on which R_U can diverge, but it can accept some tape,

(ii) if U diverges from blank tape, R_U cannot accept any tape, but can diverge,

(iii) R_U cannot diverge on any recursive tape.

§4.4 Translation to binary automata.

The theorems of this section have been proved for automata with large (but finite) alphabets, but it is more convenient to set up a correspondence between program schemata and binary automata, that is automata over an alphabet {0,1}.

Theorem 4.6 Theorems 4.1 to 4.5 are valid even if the automata are restricted to being binary.

Proof. Let us rewrite the alphabet, {a,b,ε,1,0,...}, we have been using as:

$$S = \{s_0, s_1, s_2, \ldots, s_h\}$$

and let p be the map from sequences on S to sequences on {0,1}, which is obtained by replacing:

s_0 by 0

s_1 by 10

s_2 by 110

.
.
.
.
etc.

For any automaton A over S, let p'(A) be the binary automaton obtained by introducing, for each state q, the new states $q^1, q^2, \ldots, q^h$, and by replacing each line of the table giving the transition function of A:

state	Head	next state s_0 s_1 s_2 . . . s_h
q	i	r_0 r_1 r_2 . . . r_h

say, by the new lines:

state	Head	next state 0	1
q	i	r_0	q^1
q^1	i	r_1	q^2
.	.	.	.
.	.	.	.
q^{h-1}	i	r_{h-1}	q^h
q^h	i	r_h	$\underline{r}$

where we recall that $\underline{r}$ is the reject state. It is easy

to verify that the automaton A will accept, reject, or diverge on, any tape t, precisely as the automaton p'(A) accepts, rejects, or diverges on, the tape p(t), respectively. Also, for any binary tape t', not of the form p(t) for any t,

(i) if some p'(A) accepts t', then it also accepts some tape p(t),

(ii) no p'(A) diverges on t'.

For any tape t over S,

(1) t is ultimately periodic if and only if p(t) is ultimately periodic,

(2) t is recursive if and only if p(t) is recursive.

We can therefore 'translate' each of the theorems in this section.___/

§5. UNSOLVABILITY OF PROBLEMS OF PROGRAM SCHEMATA.

§5.0 The first two lemmata show that the two-headed automata of the last section can be 'simulated' by schemata, and provide an effective 'translation' procedure. We prove that recursive equivalence is properly weaker than strong equivalence and also that recursive interpretations are not characterized by their recursive execution sequences. Theorems 5.5 and 5.6 answer all the solvability questions of the kind we are considering. Finally we strengthen one of these results to show that, under certain very acceptable hypotheses, there can be no ' adequate' rule-book of simplifying transformations.

§5.1 Simulation and first results.

We have only to show that any two-headed binary automaton can be 'simulated', in a suitable sense, by some program schema, to be able to apply the theorems of §4. Suppose that our formal language contains at least one test-symbol T, and at least one function symbol. We can assume, without loss of generality, that there is some monadic function symbol F, since otherwise we could take any function symbol and use it as a monadic function by always using it with all of its arguments identical. We require that the language contains at least two location symbols, L_1 and L_2. For any interpretation I, of the language, let:

$$x_0 = I(F)[\, I(L_1)]$$

and

$$x_{n+1} = I(F)[\, x_n\,] \quad \text{for all } n \geq 0$$

We define the tape of I, ε_I, as the sequence:

$$\varepsilon_1, \varepsilon_2, \varepsilon_3, \ldots \quad \text{where } \varepsilon_n = I(T)[\, x_n\,] \quad \text{for } n \geq 1$$

For any (infinite) binary tape s, there is clearly an interpretation I, such that $s = \varepsilon_I$.

Lemma 1. (i) A tape s, is ultimately periodic if and only if there is a finite interpretation I such that $s = \epsilon_I$.

(ii) A tape s, is recursive if and only if there is a recursive interpretation I, such that $s = \epsilon_I$.

Proof. The sufficiency is immediate in each case. As for the necessities, in (i), suppose that:

$$s = \delta_1, \delta_2, \ldots$$

and for some $h \geq 0$, $k > 0$, and for all $n \geq 0$,

$$\delta_{h+k+n} = \delta_{h+n}$$

Let I be an interpretation over the domain of non-negative integers less than h+k such that:

$$I(F)[\, I(L_1)\,] = 0$$

$$I(F)[m] = m+1 \quad \text{if } m < h+k-1$$

$$= h \quad \text{if } m = h+k-1$$

and $I(T)[m] = \delta_m$ for all m in the domain.

Clearly $s = \epsilon_I$. For (ii), suppose that s is defined by the recursive function f, then s is the tape of an interpretation I, whose domain is the non-negative integers, for which:

$$I(F)[I(L_1)] = 0 \quad \text{and } I(F)[m] = m+1$$

$$I(T)[m] = f(m). \qquad \text{___/}$$

Given a two-headed binary automaton A, we show how to construct a program schema P(A), such that the behaviour of A started on any tape s, will be paralleled by the execution of P(A) under any interpretation I, for which $s = \varepsilon_I$. At parallel stages in the operations of A and P(A), as A reads the nth symbol δ_n, of s with Head i, P(A) computes the value x_n, as defined above, assigns it to L_i and applies to it the test function I(T), thus obtaining the value δ_n. In describing the schema P(A), we shall prefix certain of the instructions with symbols $q_0, q_1, \ldots$, corresponding to states of A, and then use these symbols as transfer addresses in instructions.

The first instructions of P(A) are:

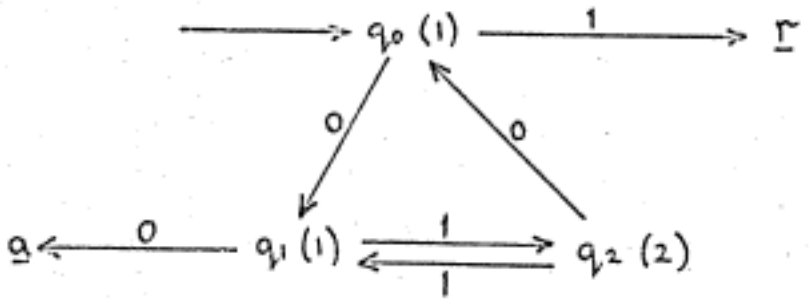
$$L_2 := F(L_1)$$
$$L_1 := F(L_1)$$

after the execution of which under I, each location holds the value x_0. The next instruction of P(A) is that which is prefixed with the initial state of A. For each state q of A, which reads with Head i and has transitions to states q', q", according to whether the symbol read is 0,1 respectively, we have the two instructions of P(A):

a. $L_i := F(L_i)$
$T(L_i)$ q',q"

The left- and right-terminal addresses are a and r respectively, which are replaced by e_0 and e_1 as usual. This concludes the description of P(A).

To illustrate this construction, suppose A' is the 5-state automaton shown below, with initial state q_0.



Automaton A'.

The schema P(A') is the following:

$L_2 := F(L_1)$

$L_1 := F(L_1)$

q_0. $L_1 := F(L_1)$

$T(L_1)$ q_1, e_1

q_1. $L_1 := F(L_1)$

$T(L_1)$ e_0, q_2

q_2. $L_2 := F(L_2)$

$T(L_2)$ q_0, q_1

Note: P(A') can only diverge under an interpretation whose tape is of the form:

0101101110111101111101111110....

It succeeds under an interpretation which breaks this sequence with an unexpected 0, and fails under interpretations which give an extra 1.

Lemma 2. For any binary two-headed automaton A, and binary tape $s = \epsilon_I$, A accepts, rejects, diverges on, s according as P(A) succeeds, fails, diverges under interpretation I.

Proof. Immediate, from the construction of P(A). __/

Using Lemmata 1 and 2 and Theorem 4.6, we can 'translate' Theorems 4.1 and 4.2 to obtain:

Theorem 5.1 There are program schemata Z^F and Z^R such that:

(I) Z^F diverges under some interpretation but fails under any finite interpretation.

(II) Z^R diverges under some interpretation but fails under any recursive interpretation.

With the aid of these results we can settle several

questions which were mentioned in §2. We promised to show:

Theorem 5.2 The sequence of relations:

$$= \ , \ \equiv_f \ , \ \equiv_r \ , \ \equiv$$

is one of strictly increasing strength.

Definitions. Let:

E_0 denote the schema ' goto e_0 '

E_1 " " " ' goto e_1 '

D " " " ' a. goto a '

E_0 always succeeds, E_1 always fails, and D always diverges.

Proof of Theorem 5.2 We note:

(a) $E_1 = D$ but $E_1 \not\equiv_f D$

(b) $E_1 \equiv_f Z^F$ but $E_1 \not\equiv_r Z^F$

(c) $E_1 \equiv_r Z^R$ but $E_1 \not\equiv Z^R$ __/

We can modify Z^R to produce a counter-example to the tempting supposition that any recursive execution sequence through a schema is the sequence corresponding to some recursive interpretation. In terms of automata, we remove

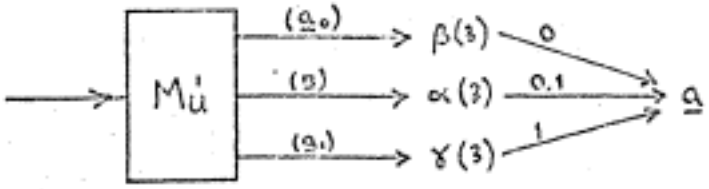
the sequence of δ's from the main tape, and put it instead on a second tape. The automaton still demands that this sequence predict on which symbol, if any, each Turing machine halts, and so can only diverge if the second tape is non-recursive. However the automaton is designed so that it does not take account of a symbol on this tape until after the corresponding Turing machine has halted. In this way, the sequence of states when the automaton diverges remains recursive. In terms of schemata, we introduce two new locations M_1 and M_2, and it is the 'tape of I' corresponding to the location M_1 which is the δ-sequence. M_2 is used as a 'scanning head', while M_1 stays constant. The instructions of Z^R concerned with 'reading the δ-sequence' are deleted, and in their place:

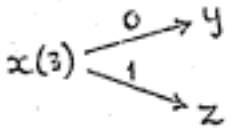
(i) whenever 'the symbol b is read by Head 1' (in automata terms) the instruction:

$$M_2 := F(M_1)$$

is executed, which corresponds to re-setting the scanning-head at the beginning of the tape, and then the original course of instructions is resumed,

(ii) in similarly hybrid terms, the schema/automaton M_U^*, whose defining diagram was fig.6 (§4.3) is replaced by:



where  translates to ' $M_2 := F(M_2)$ $T(M_2)$ y, z '

Thus, for any interpretation under which the new automaton diverges, the value of each δ_n has no effect on the execution sequence until the simulation of the nth Turing machine computation has halted, and so the (unique) divergent execution sequence is just a 'diagonal' simulation of U started on every $\underline{n}$ in turn, and is clearly recursive. There are interpretations I, with this divergent execution sequence but since the tape of I corresponding to location M_1 has to be non-recursive, by Lemma 1 any such I is non-recursive.

Recalling the results of §2.3, we have:

Theorem 5.3

(I) Any finite interpretation gives rise to an ultimately periodic execution sequence, but there are ultimately periodic execution sequences which do not

correspond to any finite interpretation.

(II) Any recursive interpretation gives rise to a recursive execution sequence, but there are recursive execution sequences which do not correspond to any recursive interpretation.

§5.2 Decision problems for equivalences.

Theorems 4.2 and 4.5 translated into program schemata terms, yield:

Theorem 5.4 There is an effective construction which produces, for any Turing machine U, program schemata P_U and P_U^r with the properties:

(i) If U halts from a blank initial tape, there is no interpretation under which either P_U or P_U^r can diverge, but each can succeed under some interpretation.

(ii) If U diverges from a blank tape, neither can succeed under any interpretation, but each can diverge under some interpretation.

(iii) P_U^r converges under all recursive interpretations, but, in case (ii) above, P_U can diverge under a recursive interpretation.

We shall use the symbols P,Q as variables to range over the class of program schemata. In a mild abuse of language, we shall write, for example:

' P $\not\equiv_f$ Q is partially solvable '

to mean:

' There is a recursive procedure which, for all pairs of schemata P,Q, when applied to P and Q, terminates if and only if P $\not\equiv_f$ Q .'

We write, for example:

' P $\{\overset{\equiv r}{=}\}$ Q '

as an abbreviation for:

' P $\sim$ Q for all relations $\sim$, satisfying,

(i) $\equiv r \Rightarrow \sim$

(ii) $\sim \Rightarrow =$ '

Thus $\{\overset{\equiv}{=}\}$ stands for all reasonable relations.

Lemma 3. (i) P $\not\equiv_f$ Q is partially solvable.

(ii) P $\not=$ Q is partially solvable.

Proof. For any finite interpretation I, and schema P, since each location can only take on a finite number of

different values, the computation sequence $A_I(\pi_I^{(P)})$ must either terminate or repeat itself. So it is determinable in a finite number of operations whether $val(P_I)$ is 0,1 or undefined. We showed in §2.1 that $\simeq$ is the same as $\simeq_f$. The set of finite interpretations is effectively enumerable, and the partial procedure for both $\not\simeq_f$ and $\not\simeq_f$ consists in investigating the values of P_I and Q_I for each I in turn, terminating only when a suitable disparity is found. __/

Lemma 4. (i) $P \simeq D$ is true for all P.

(ii) $P \{\overset{\not\simeq f}{\not\simeq}\} D$ is partially solvable.

Proof of (ii). If $P \not\simeq D$, then P converges under some interpretation, and therefore also under some finite interpretation. The result follows from Lemma 3. __/

Lemma 5. $P \{\overset{\equiv}{\equiv_f}\} D$ is not partially solvable.

Proof. Let P_U' be the schema P_U of Theorem 5.4 modified by introducing a 'dynamic stop' :

$$e_1.\quad \text{goto}\ \ e_1$$

at the old right-terminal address, so that a failure of

P_U is a divergence of P_U'. Then $P_U' \equiv D$ if and only if U diverges from a blank tape. The lemma follows from Theorem A, and the proof of Lemma 4. __/

Lemma 6. (i) $P \equiv E_1$ is partially solvable.

(ii) $P \{\overset{\equiv r}{=}\} E_1$ is not partially solvable, (even when restricted to schemata which converge under all recursive interpretations.)

Proof. (i) We showed in §3.2 that the equivalence problem for schemata which converge under all interpretations is solvable, and if $P \equiv E_1$, it is certainly such a schema, so we already have an appropriate partial procedure.

(ii) If U halts from a blank tape, $P_U{}^r \neq E_1$. If U diverges from a blank tape, $P_U{}^r \equiv r\ E_1$. The result follows from Theorem A. __/

Lemma 7. $P \{\overset{\not\equiv r}{\neq}\} E_1$ is not partially solvable.

Proof. Let P_U'' be the schema P_U, modified by inserting an 'unconditional transfer' to e_1:

e_0. goto e_1

as a new instruction, with prefix the old left-terminal address of P_U, i.e. by 'counting success as failure'. Then:

(i) if U halts from a blank tape, $P_U'' \equiv E_1$,

(ii) if U diverges from a blank tape, $P_U'' \not\equiv_r E_1$,

since P_U'' can diverge under some recursive interpretation. Theorem A again gives the result. __/

Although the last two lemmata show that, for any reasonable relation $\sim$, if it is stronger than or weaker than recursive equivalence, $P \sim E_1$ is recursively unsolvable, there remains to prove:

Lemma 8. $P \ (\overset{\equiv}{\sim}) \ E_1$ is recursively unsolvable, even when restricted to schemata which converge under all recursive interpretations.

Proof. (due to D. M. Park) Let U be a universal Turing machine. Using the techniques of this section and §4, we can construct for each m, a schema P_m which converges under every recursive interpretation and

(i) if $C[U,m] = 0$, P_m can succeed,

(ii) if $C[U,m] = 1$, P_m always fails.

Suppose there is a reasonable relation $\sim$, such that ' $P \sim E_1$ ' is solvable, then there is a recursive function

f such that:

$$C[U,m] = 0 \quad \Rightarrow \quad P_m \text{ can succeed}$$
$$\Rightarrow \quad P_m \neq E_1$$
$$\Rightarrow \quad P_m \not\simeq E_1$$
$$\Rightarrow \quad f(m) = 0$$

and:

$$C[U,m] = 1 \quad \Rightarrow \quad P_m \equiv E_1$$
$$\Rightarrow \quad P_m \sim E_1$$
$$\Rightarrow \quad f(m) = 1$$

This contradicts Lemma 6, §4.3. /

Lemma 9. $P \{\overset{\equiv}{\sim}\} Q$ is not partially solvable, even if P,Q are restricted to schemata which converge under all recursive interpretations.

Proof. Let $P_U^{r''}$ be the schema obtained from P_U^{r} by 'counting success as failure' just as in the proof of Lemma 7. Then:

(i) if U halts from a blank tape, $P_U^{r} \neq P_U^{r''}$,

(ii) if U diverges, $P_U^{r} \equiv P_U^{r''}$,

and both these schemata converge under all recursive interpretations. /

This result differs sharply from the main result of §3.2, where we showed that any reasonable relation has a solvable decision problem when restricted to schemata which converge under all interpretations. We summarize the results of this section, (abbreviating 'partially solvable' to 'p.s.').

Theorem 5.5

(A) $P\ \{\underset{\equiv f}{\equiv}\}\ D$ is not p.s.; $P\ \{\underset{\not\equiv}{\not\equiv f}\}\ D$ is p.s.;

$P \simeq D$ is true; $P \not\simeq D$ is false.

(B) $P \equiv E_1$ is p.s.; $P\ \{\underset{\not\equiv}{\not\equiv r}\}\ E_1$ is not p.s.;

$P\ \{\underset{=}{\equiv r}\}^*\ E_1$ is not p.s.; $P \not\equiv f\ E_1$ is p.s.;

$P \not\equiv E_1$ is p.s.

(B') $P\ \{\underset{=}{\equiv}\}^*\ E_1$ is recursively unsolvable.

[Of course the same results hold with E_0 in place of E_1.]

(C) $P\ \{\underset{=}{\equiv\equiv}\}^*\ Q$ is not p.s.; $P\ \{\underset{\not\equiv}{\not\equiv r}\}\ Q$ is not p.s.;

$P \not\equiv f\ Q$ is p.s.;

$P \not\equiv Q$ is p.s.

An asterisk signifies that the result is true even if

the relations are restricted to schemata which converge under all recursive interpretations.

Related questions are settled by:

Theorem 5.6

(A) Can P diverge under any

(i) interpretation,

(ii) recursive interpretation,

(iii) finite interpretation ?

(B) Does P converge under every

(i) interpretation,

(ii) recursive interpretation,

(iii) finite interpretation?

(A) (i) is not p.s.	(B) (i) is p.s.
(ii) is not p.s.	(ii) is not p.s.
(iii) is p.s.	(iii) is not p.s.

Proof. The results for (A) (i) and (ii) come from the proof of Lemma 7, and (iii) holds because there is an effective method of determining if a given schema diverges under a given finite interpretation, and because the set of finite interpretations is countable.

The result for (B) (i) follows at once from

there being an effective procedure which produces, from any always-convergent schema, an equivalent schema without loops. As for (B)(ii) and (iii), consider the schema $P_U^{r'}$, which is P_U^{r} with a 'dynamic stop' inserted at the old left-terminal address, so:

$$e_o. \quad \text{goto} \quad e_0$$

If U diverges from a blank tape, $P_U^{r'}$ cannot diverge under any recursive interpretation, but if U halts, then there is a finite interpretation under which the simulation of U terminates and $P_U^{r'}$ diverges in the 'dynamic stop'. __/

§5.3 'Adequate' rule-books.

To prepare for our chief result on 'rule-books' for simplification, we must prove a result as broad as that of Lemma 9, for the case when Q is some particularly simple, fixed, schema, Z say. None of E_0, E_1 or D is a possible choice for Z. Of course if we are demanding that Z be simple, we cannot hope to prove:

$$P \{\overset{\equiv}{=}\}^{*} Z \qquad \ldots . (X)$$

(i.e. with the asterisk) to be **not partially solvable**, because if Z does not converge under all recursive interpretations then (X) is always false, and if Z does then it must surely always converge and (X) is partially solvable.

Let Z be the following schema:

$$\begin{array}{ll} a. & L_1 := F(L_1) \\ & T(L_1)\ \ e_1,\ \ a \end{array}$$

which, for any I, fails unless the tape of I consists entirely of 1's. For any Turing machine U, we can construct a schema Z_U which starts:

$$\begin{array}{ll} & L_2 := F(L_1) \\ & L_1 := F(L_1) \\ & T(L_2)\ \ e_1,\ \ ^{\cdot}+1 \\ a. & L_2 := F(L_2) \\ & T(L_2)\ \ \ ^{\cdot}+1,\ a \\ & \vdots \end{array}$$

and succeeds precisely under those interpretations with a tape of the form:

$$\underline{n}\ C(0)\ \underline{n-1}\ C(1)\ \underline{n-2}\ \ldots\ \underline{n-r}\ C(r) \quad \text{for some } n > 0$$

where C(m) denotes the mth configuration of U started on a blank tape, C(r-1) is terminal and $r \leq n$. If the computation of U has not halted after (n-1) steps the schema fails. The only interpretations for which Z_U can diverge are those with:

$$I^*(F^m L_1) = 1 \quad \text{for all } m$$

which corresponds to $n = \infty$ above. Provided n is finite, a 'time-limit' will have been set and Z_U must converge at least by the time this is exceeded. If U halts from a blank tape, then there is an interpretation which provides a sufficiently large 'time-limit' n, for Z_U to succeed, but for which Z still fails, so:

$$Z_U \not\equiv Z$$

Alternatively, if U does not halt:

$$Z_U \equiv Z$$

since any finite 'time-limit' will be exceeded. Thus we have proved:

Theorem 5.7 There is a very simple 2-instruction schema Z, such that, for any reasonable relation $\sim$;

$$P \sim Z \text{ is not partially solvable.}$$

In §2.4 we discussed 'rule-books' for transitive reasonable relations, as the possible basis of a practical simplification algorithm. Let $\sim$ by some fixed transitive reasonable relation.

Definition. A rule-book $\mathbb{B}$ for $\sim$ is adequate (with respect to some concept of 'simplicity'), if, for all P, if $P \sim Q$ and Q is 'simpler' than P, then Q is derivable from P by a sequence of rules of $\mathbb{B}$.

(H): Hypothesis (about 'simplicity' and $\sim$).

If Z is the 2-instruction schema described above, then $P \sim Z$ is partially solvable for all P which are at least as 'simple' as Z.

Under this hypothesis, which is trivially true if, say, only finitely many schemata are at least as 'simple' as Z, we have:

Theorem 5.8 There is no adequate rule-book for $\sim$.

Proof. Suppose $\mathbb{B}$ is an adequate rule-book for $\sim$, then, for any P, if Z is simpler than P, Z is derivable from P by rules of $\mathbb{B}$ and so $P \sim Z$ is established by a partial procedure which merely enumerates sequences of such rules, and if Z is not simpler, $P \sim Z$ is established

by the partial procedure of the hypothesis (H). This contradicts Theorem 5.7 and the theorem follows. __/

We describe briefly how we can obtain a similar result to Theorem 5.7 with P restricted to schemata which converge under all finite (or recursive) interpretations. We recall the schema Z^F of Theorem 5.1, which could only diverge for an interpretation with a tape of the form:

$$\underline{0}\ 1\ \underline{2}\ 3\ \underline{4}\ 5\ \underline{6}\ \ldots$$

By using Z^F in place of Z as the initial portion of Z_U, we obtain a schema Z_U^F with the following properties:

(i) Z_U^F converges under all finite interpretation,

(ii) if U diverges, $Z_U^F \equiv Z^F$,

(iii) if U halts, Z_U^F can succeed under an interpretation whose tape starts:

$$\underline{0}\ 1\ \underline{2}\ \ldots\ \underline{n}\ C(0)\ \underline{n-1}\ C(1)\ \ldots\ \underline{n-r}\ C(r)$$

for some $n \geq r > 0$, and terminal C(r-1), so:

$$Z_U^F \neq Z^F$$

A similar construction produces Z_U^R from U and R. Thus we have:

Theorem 5.9 There are schemata Z^F, Z^R, such that, for all reasonable relations $\sim$;

$$P \sim Z^F \quad (\ P \sim Z^R \text{ respectively})$$

is not partially solvable, even if P is restricted to schemata which converge under all finite (respectively recursive) interpretations.

There is a 7-instruction schema which may be taken as Z^F, so the hypothesis corresponding to (H) is, in this case, still very acceptable, but in the case of Z^R, the restriction seems rather stronger. Of course, under these hypotheses, we obtain results corresponding to Theorem 5.8.

Just one more feature of a construction in this section is worth pointing out. We define a trivial loop as one with no exits, e.g.:

a. goto a

and a unitary loop as one with just one exit, e.g.:

a. L :=F(L)

T(L) a, b

In §3.2 we found a decision procedure for schemata which always converge, and we could extend this to schemata which either converge or enter a trivial loop.

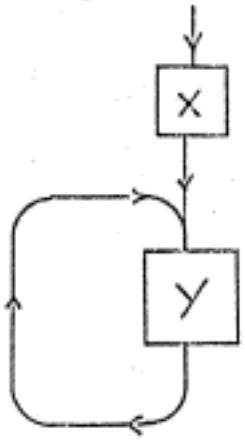
We might expect that in any class of schemata with unsolvable decision problems, some schemata should diverge in complicated ways, however even for the class of schemata which converge or else diverge in some specified unitary loop, the equivalence problem is unsolvable. Furthermore the unitary loop may be of the very simple form of the example in the definition. We merely remark that the schemata Z_U of this section have this property. (This result can be strengthened slightly to show the unsolvability for schemata which diverge in such a unitary loop or fail.)

PART III. SOLVABLE PROBLEMS.

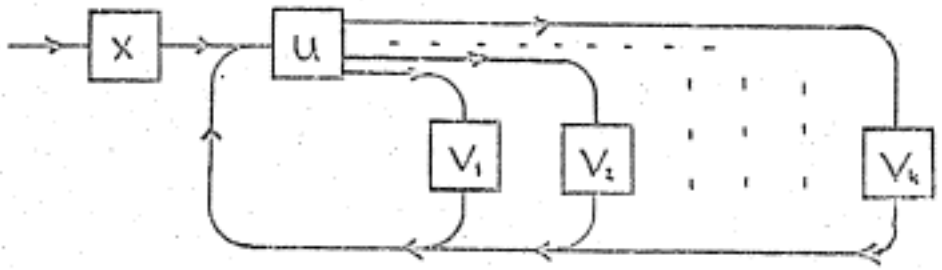
Since we are motivated by the desire to use program schemata as a tool in the simplification and 'proving' of computer programs, the results of the last section can only be regarded as negative. The remainder of this dissertation is devoted to the more positive results we have obtained, which hold out some hope of providing a basis for practical techniques. From now on we restrict ourselves to the decision problem for (strong) equivalence. In some cases the corresponding results for other relations are readily obtained, but in others there are further difficulties. We describe results obtained in two main directions.

A characteristic feature of the schemata with unsolvable decision problems that we have been considering, is that most expressions which are calculated are recalculated later on in an execution sequence, a feature which would be unusual and undesirable in an actual computer program. We shall investigate various restrictions which can be imposed on schemata to inhibit this repetitive behaviour. Firstly though, we shall examine schemata whose flow-diagrams have a particularly simple structure.

A typical form for the 'simulation' schemata we have been using is the following:



where we neglect transfers to terminal addresses, and where X 'checks an initial configuration' and Y 'checks that each configuration follows from the previous one'. Furthermore, by simulating a universal Turing machine we can obtain a class of schemata with unsolvable decision problems by keeping Y fixed and varying X. We observe that X is always loop-free, so we do not require arbitrarily complicated loop-structures in an unsolvable class of schemata. However, taking the structure of Y into account, the typical flow-diagram can be put in the form:



and we see that the loops in the schemata are 'nested' to a considerable degree. The schemata we investigate in the next section are those with no 'nested' loops. So far only a decision procedure for such schemata with only monadic function symbols, has been found. I conjecture that there is also one for the general case, but there are obstacles to extending the present procedure.

§6. SCHEMATA WITH NON-INTERSECTING LOOPS AND MONADIC FUNCTION SYMBOLS.

§6.0 We express the execution of a finite sequence of computation instructions as the multiplying of the vector giving the values of the locations by a certain matrix, and show that any path, whether finite or infinite, in a schema with non-intersecting loops can be completely specified by a vector of integers with a bounded number of components. The value of a location after the execution of such a path proves to be expressible by a 'linear word function' and a formula of the additive theory of the natural numbers with the components of the path-vector as free variables. Hence we find a formula which expresses the equivalence of two given schemata, and as the truth of formulae of this theory is decidable [5], the equivalence problem is recursively solvable.

§6.1 Preliminaries.

We define $\mathbb{S}$ as the class of schemata satisfying the restrictions:

(I) No computation instruction is in more than one loop.

(II) The programming language contains only function symbols which take one argument.

We shall assume that there is just a single test symbol T, but, in distinction from the above two conditions, this assumption involves no real loss of generality. For suppose we had the test function T', then we merely introduce the new location L' and new function symbol F', and replace each:

T'(L) b, c

by:

L' := F'(L)
T(L') b, c

This is an equivalence-preserving transformation and one which also preserves conditions (I) and (II) above.

Let P be a schema in $\mathbb{S}$. We can assume that there is some computation instruction in each loop, because there are obvious ways of producing a schema equivalent to P and having this property if P does not have it,

and we select one such as a base-point in each loop. Let their addresses be $b_1, b_2, \ldots, b_q$, which are distinct by condition (I).

A path is a sequence through a schema which does not necessarily start at the initial address nor reach a terminal address.

Lemma 1. If there is a path from b_i to b_j $(i \neq j)$ then there is not also a path from b_j to b_i.

Proof. Obvious from condition (I). /

For any finite path p in P, we define, for each $i = 1, \ldots, q$, the ith loop-coefficient, γ_i, to be one less than the number of occurrences of b_i in p if positive, and zero otherwise. Lemma 1 assures us that the occurrences of different base-points do not interlace each other. We now obtain the reduced path, p', by cutting out, for each positive loop-coefficient, that part of p which follows the corresponding base-point up to and including its final occurrence. The reduced path cannot contain any loops more than once, and so the number of possible reduced paths in any schema is finite. Let us call them $p_1, \ldots, p_v$ for P. Any finite path path p, in P can be specified completely by the reduced path $p' = p_j$ say, and the loop-coefficients $\gamma_1, \ldots \gamma_q$.

Suppose the function symbols $F_1,\ldots F_p$ and location symbols $L_1,\ldots,L_m$ are in our language. Let Z be the free semigroup with generators F_i, i=1,..p, and L_j, j=1,..,m and zero and identity elements (0, 1) adjoined in the natural way. We use concatenation to denote the semigroup operation.

Definitions. An (m-)transform is an (m x m)-matrix over Z with exactly one non-zero element in each row. Multiplication of transforms is denoted by concatenation and is defined by:

$$(AB)_{rs} = A_{rt}B_{ts} \quad \text{where t is the unique integer such that } A_{rt} \neq 0$$

Multiplication is associative and the product of two transforms is a transform. It is convenient to abbreviate the instruction:

$$L_i := F_k(L_j)$$

by:

$$T_{ijk}$$

$M(T_{ijk})$, the transform corresponding to the instruction T_{ijk}, is defined as A where:

$$A_{ij} = F_k$$
$$A_{rr} = 1 \text{ if } r \neq i$$
$$A_{rs} = 0 \text{ otherwise.}$$

$M(T_{ijk})$ has the form:

$$
\begin{array}{c}
 \\
 \\
 \\
 \\
i^{th}\rightarrow \\
 \\
 \\

A linear word function (lwf) is a word function of the form:

$$W(m_1,\ldots,m_k) = U_1^{m_1} U_2^{m_2} \ldots U_k^{m_k}$$

where $U_i \in Z$ for $i=1,..,k$, and m_i is a non-negative integer.

$$U_i^{m_i} = \underbrace{U_i U_i \ldots U_i}_{m_i \text{ times}} \quad \text{if } m_i > 0,$$

$$= 1 \quad \text{if } m_i = 0$$

§6.2 P-representations.

Let $\mathbb{P}$ be the first order theory of the natural numbers (non-negative integers) with addition as the only operation. Hilbert and Bernays ([5], vol.1, pp 359-366) have given a decision procedure for the truth of formulae in this theory. (Presburger [12] proved the elementary theory of the integers to be decidable somewhat earlier.) We shall call a formula of $\mathbb{P}$ a Presburger formula or P-formula. A P-formula $\phi(t_1,..,t_r;n_1,..,n_s)$, with the free variables as indicated, will be called a P-function if, for all natural numbers $n_1,..,n_s$, there exist unique natural numbers $t_1,..,t_r$ such that $\phi(t_1,..,t_r;n_1,..,n_s)$ is true. This condition can be succintly expressed as:

$$(\underline{n})(E!\underline{t})[\ \phi(\underline{t};\underline{n})\ ]$$

where the quantifier ' E! ', which states unique existence, is readily definable in terms of the equality relation and E, the standard existential quantifier. We shall often find it convenient to express metatheoretical statements in the formalism of the predicate calculus.

A word function $G(n_1,..,n_s)$ is <u>P-representable</u> if there is a <u>lwf</u> W, and a P-function ϕ, such that:

$$(\underline{t})(\underline{n})[\ \phi(\underline{t};\underline{n}) \implies G(\underline{n}) = W(\underline{t})\ ]$$

<u>Lemma 3</u>. If $G_1(\underline{m})$, $G_2(\underline{n})$ are P-representable then so are:

(i) $G_3(\underline{m},\underline{n}) = G_1(\underline{m})G_2(\underline{n})$

(ii) $G_4(\underline{m},\underline{n},r) = G_1(\underline{m})$ if $\delta(r)$

$= G_2(\underline{n})$ if $\neg\delta(r)$

where δ is a P-formula

<u>Proof</u>. Suppose:

$$(\underline{s},\underline{m})[\ \phi_1(\underline{s};\underline{m}) \implies G_1(\underline{m}) = W_1(\underline{s})\ ]$$

and $$(\underline{t},\underline{n})[\ \phi_2(\underline{t};\underline{n}) \implies G_2(\underline{n}) = W_2(\underline{t})\ ]$$

then:

(1) $$(\underline{s},\underline{t},\underline{m},\underline{n})[\ \phi_1(\underline{s};\underline{m}) \wedge \phi_2(\underline{t};\underline{n}) \ .\Longrightarrow.\ G_3(\underline{m},\underline{n}) = W_1(\underline{s})W_2(\underline{t})\ ]$$

$$(ii)\quad (\underline{s},\underline{t},\underline{m},\underline{n},r)[\ \delta(r) \wedge \phi_1(\underline{s};\underline{m}) \wedge \underline{t}=\underline{0}\ \ .\vee.$$

$$\neg\delta(r) \wedge \phi_2(\underline{t};\underline{n}) \wedge \underline{s}=\underline{0}\ \ :\Rightarrow:$$

$$G_4(\underline{m},\underline{n},r) = W_1(\underline{s})W_2(\underline{t})\ \ ]$$

—/

Lemma 4. If A is an (m-) transform, there exist $h,k > 0$ with $h+k \leq m^m$, such that:

$$A^{h+k} = A^h D \quad \text{where D is a diagonal matrix.}$$

Proof. We can find B,D_1 where B is a transform with elements either 0 or 1, and D_1 is a diagonal matrix, such that:

$$A = D_1 B$$

Similarly there is a diagonal matrix D_2 such that:

$$BD_1 = D_2 B$$

and we define D_{n+1} for each $n \geq 1$, by equations:

$$BD_n = D_{n+1} B$$

and denote $D_1 D_2 \ldots D_r$ by D'_r . For each r, B^r is a transform with only 1's as its non-zero elements. There are at most m^m such transforms, so not all of $I,B,B^2,..,B^{(m^m)}$ are distinct. So $B^{h+k} = B^h$ say, for some h,k with $h+k \leq m^m$. Then:

$$A^{h+k} = (D_1 B)^{h+k}$$

$$= (D_1 B)^h (D_1 B)^k$$

$$= D_h' B^h D_k' B^k$$

$$= D_h' E B^{h+k} \qquad \text{where } B^h D_k' = E B^h$$

$$= D_h' E B^h$$

$$= D_h' B^h D_k'$$

$$= A^h D_k'$$

But D_k' is a diagonal matrix. ___/

Let α be a segment, and let $G_{\alpha,i}(n)$ be the word-function whose value for each n is the expression associated with L_i after n iterations of α, starting from the initial expressions $\underline{L} = L_1,\ldots,L_m$.

Lemma 5. $G_{\alpha,i}(n)$ is P-representable.

Proof. By Lemma 4, there is a diagonal matrix $D = \left[d_1,\ldots,d_m\right]$, such that:

$$(M(\mathbf{a}))^{h+k} = (M(\alpha))^h D \quad \text{for some } h,k \text{ with } h+k \leq m^m.$$

Therefore we can find words in Z, say $v_1,\ldots,v_{h-1},w_0,\ldots,w_{k-1}$, and integers $j_1,\ldots,j_{h-1},i_0,\ldots i_{k-1}$ of the set $\{i,\ldots,m\}$ so that:

$$G_{\alpha,i}(n) = \begin{cases} L_i & \text{if } n = 0 \\ v_1 L_{j_1} & \text{if } n = 1 \\ \vdots & \vdots \\ v_{h-1} L_{j_{h-1}} & \text{if } n = h-1 \\ w_0 d_{i_0}^{r} L_{i_0} & \text{if } n = k.r + h \text{ for some } r \geq 0 \\ w_1 d_{i_1}^{r} L_{i_1} & \text{if } n = k.r + h + 1 \text{ " " } \\ \vdots & \vdots \\ w_{k-1} d_{i_{k-1}}^{r} L_{i_{k-1}} & \text{if } n = k.r + h + (k-1) \text{ " } \end{cases}$$

For fixed k,h,s, we express ' $n = k.r + h + s$ for some $r \geq 0$ ' as:

$$(Er)\left[n = \underbrace{r+r+\ldots+r}_{k \text{ times}} + h + s\right]$$

which is a P-formula. Using Lemma 3(ii) repeatedly, we can construct a P-function and a lwf which P-represent $G_{\alpha,i}(n)$. ___/

Given P-representations of $G_{\alpha,j}$ for j=1,...,m, and of $G_{\beta,i}$, we produce a P-representation for the word-function $G_{\alpha,\beta,i}(r,s)$ of two arguments which gives the value of L_i after the sequence ' $\alpha^r \beta^s$ ', by 'substituting' the lwf's of $G_{\alpha,j}$ for the appropriate L_j's in the lwf of $G_{\beta,i}$.

For a particular location L_i and reduced path p_u in a schema Q, the word-function $G_{u,i}(\gamma_1,\ldots,\gamma_q)$, which gives the value of location L_i after the execution of the path p, with reduced path p_u and loop-cooeficients $\gamma_1,\ldots,\gamma_q$, if such a p exists (and has the value 0 otherwise), is P-representable. This result is readily obtainable from Lemmata 3 and 5, and the last remark.

Now let $\underline{h} = (u,i,\gamma_1,\ldots,\gamma_q)$ be a (q+2)-vector, which is intended to specify both a path in Q and a location L_i. We define the word-function:

$$Q(\underline{h}) = G_{u,i}(\gamma_1,\ldots,\gamma_q) \quad \text{if this is defined,}$$

$$= 0 \qquad\qquad\qquad\qquad \text{otherwise.}$$

Obviously Q is P-representable too.

We shall be interested in both finite and infinite paths through schemata. In schemata of the class $\mathbb{S}$, any infinite path must ultimately remain in a single loop. We represent such a path by its finite initial segment which terminates at the first occurrence of the corresponding base-point. We adopt the convention that the loop-coefficient for this ultimate loop shall be 0. For any schema Q, let:

(i) $fin_0^Q(\underline{h})$ be the condition that $\underline{h}$ represents a path in Q from the initial address to the terminal address e_0,

(ii) $fin_1^Q(\underline{h})$ be the corresponding condition with e_1,

(iii) $inf^Q(\underline{h})$ be the condition that $\underline{h}$ represents an infinite sequence from the initial node of Q, under the convention given above,

(iv) $sub^Q(\underline{k},\underline{h})$ be the condition that $\underline{k}$ represents a path in Q which is an initial segment of the path represented by $\underline{h}$,

(v) $T_0^Q(\underline{k},\underline{h})$ be the condition that $sub^Q(\underline{k},\underline{h})$ and that the path $\underline{k}$ is finite and leads to a transfer instruction at which the location corresponding to $\underline{k}$ is tested, and which requires that the test-function takes the value 0 in order that the path $\underline{h}$ be continued, and

(vi) $T_1^Q(\underline{k},\underline{h})$ be the corresponding condition with 1 in place of 0.

We notice that '$fin_0^Q(\underline{h})$', '$fin_1^Q(\underline{h})$', and '$inf^Q(\underline{h})$' just take the value 'true' for a certain finite range of values of u ($=h_1$) under some corresponding conditions on the loop-coefficients. 'sub^Q' is a

straightforward predicate involving only a finite number of possibilities for h_1 and k_1, some equalities and one inequality between some loop-coefficients. 'T_0^Q' and 'T_1^Q' involve 'sub^Q', but are otherwise just enumerations of a finite number of possibilities. I claim that 'fin_0^Q', 'fin_1^Q', 'sub^Q', 'T_0^Q' and 'T_1^Q' are predicates definable in P, and there is an effective construction of them from a given schema Q. Detailed proofs of these claims would, however, be cumbersome, tedious to read and, surely, superfluous.

We now require a method of constructing a P-formula from two given lwf's, specifying the precise conditions on their arguments, that they take the same value. This is given by Theorem 6.1 below. First we need to prove:

Lemma 6. Let U,V be words in Z, and u = length(U), v = length(V). Suppose that $v \geq u$ and n is an integer such that $n > (v/u)+1$. If:

$$U^n X = VVY \quad \text{for some words } X,Y$$

then there is a word W and integers r,s (≥ 1) such that:

$$U = W^r \quad \text{and} \quad V = W^s.$$

Proof. Assuming that $U^r \neq V$ for any r, we can suppose that:

$$U^r z_1 z_2 \ldots z_t = V \text{ for some } r > 0, \text{ and some } t < u,$$

where $U = z_1 z_2 \ldots z_u$ and the z's are generators.

From the hypothesis, we have:

$$z_{t+1} z_{t+2} \ldots z_u z_1 \ldots z_t = z_1 z_2 \ldots z_u.$$

So:

$$z_i = z_{[i+t]} = z_{[i+2t]} = \ldots = z_{[i+mt]} \text{ for all } m \geq 0, \text{ where } [j] \equiv j \pmod{u} \text{ and } 0 < [j] \leq u, \text{ for } i \leq u.$$

Let $h = \text{hcf}(u,v) = \text{hcf}(u,t)$, then:

$$z_i = z_{i+mh} \text{ for } 1 \leq i \leq h \text{ and } m = 0,1,\ldots,(u/h) - 1.$$

Therefore if W denotes '$z_1 z_2 \ldots z_h$' :

$$U = W^r \text{ and } V = W^s \text{ where } rh = u \text{ and } sh = v . \quad /$$

§6.3 Decision Procedure

Theorem 6.1 If H and K are lwf's, there is a P-formula $\varphi_{H,K}$, which is effectively obtainable from H and K, such that:

for all $\underline{h},\underline{k}$, $H(\underline{h}) = K(\underline{k})$ if and only if $\varphi_{H,K}(\underline{h},\underline{k})$ is true.

Proof. Let $H(\underline{h}) = U_1^{h1}\ U_2^{h2}\ \ldots\ U_m^{hm}$, $K(\underline{k}) = V_1^{k1}\ V_2^{k2}\ \ldots\ V_n^{kn}$, The result is trivial if m=0 or n=0. Suppose that the result holds for all pairs with $m+n < t$, where $t \geq 1$. We show that we can construct a P-formula $\varphi_{H,K}$ for the case when $m+n = t$, then the result follows by induction.

We write 'U < V' for words U,V when U properly divides V on the left, i.e. when UW = V for some non-null W.

Case (i) $U_1 = V_1$

$\varphi_{H,K}(\underline{h},\underline{k})$ is equivalent to:

$$(Eh)\left[U_1^h U_2^{h_2} \ldots U_m^{h_m} = V_2^{k_2} \ldots V_n^{k_n} \wedge h_1 = h+k_1\right] .\ V\ .$$

$$(Ek)\left[U_2^{h_2} \ldots U_m^{h_m} = V_1^k V_2^{k_2} \ldots V_n^{k_n} \wedge k_1 = k+h_1\right]$$

and the induction is complete.

Case (ii) $U_1 \not< V_1$ and $V_1 \not< U_1$ and $U_1 \neq V_1$

$\varphi_{H,K}(\underline{h},\underline{k})$ is equivalent to:

$$\left[U_2^{h2} \ldots U_m^{hm} = V_1^{k1} \ldots V_n^{kn} \wedge h_1 = 0\right] .\ V\ .$$

$$\left[U_1^{h1} \ldots U_m^{hm} = V_2^{k2} \ldots V_n^{kn} \wedge k_1 = 0\right]$$

Case (iii) $U_1 < V_1$ and similarly for $V_1 < U_1$

Applying Lemma 6, we see that there are two

alternatives:

either (a) $U_1 = W^r$, $V_1 = W^s$ for some word W and integers r,s, when $\varphi_{H,K}(\underline{h},\underline{k})$ is equivalent to:

$$(Ei)\left[W^i U_2^{h_2} \ldots U_m^{h_m} = V_2^{k_2} \ldots V_n^{k_n} \wedge i + s.k_1 = r.h_1\right] \quad . V .$$

$$(Ej)\left[U_2^{h_2} \ldots U_m^{h_m} = W^j V_2^{k_2} \ldots V_n^{k_n} \wedge j + r.h_1 = s.k_1\right]$$

where '$r.h_1$' denotes '$\underbrace{h_1 + h_1 + \ldots + h_1}_{r \text{ times}}$',

or (b) there is no such W, and $h_1 \leq r+1$ or $k_1 < 2$, where $r = \left[\frac{\text{length } (V_1)}{\text{length } (U_1)}\right]$. In this case, $\varphi_{H,K}$ can be written as the disjunction of the r+4 possibilities corresponding to $h_1 = 0,1,\ldots,r+1$, or $k_1 = 0,1$. In each of these the initial factor of either H or K can be replaced by a fixed word, so we require a P-formula to represent an equality of the form:

$$WU_2^{h_2} \ldots U_m^{h_m} = V_1^{k_1} \ldots V_n^{k_n}$$ or similarly with W on the right hand side. Let $r = \left[\frac{\text{length } (W)}{\text{length } (V_1)}\right]$, then we distinguish two further subcases:

(b_1) if $W \neq V_1^{r+1}$, then $k_1 = 0,1,\ldots,$ or r.

Each of these possibilities gives rise to an equation of the form:

$$WU_2^{h2} \ldots U_m^{hm} = W'V_2^{k2} \ldots V_n^{kn}$$

and then either $W = W'W''$ and the equation is representable by:

$$\left[W''^{h}U_2^{h2} \ldots U_m^{hm} = V_2^{k2} \ldots V_n^{kn} \wedge h = 1 \right]$$

or else $W \not\supset W'$ and the equation is always false (and so representable by '0=1', say). In either case the induction is complete.

(b_2) If $WY = V_1^{r+1}$ and $V_1 = XY$ for some X and Y, then we have the possibilities of (b_1) or else $k_1 \geq r+1$. The latter case reduces to the equation:

$$U_2^{h2} \ldots U_m^{hm} = YV_1^{k} \ldots V_n^{kn} \quad \text{where } k+r+1 = k_1.$$

We now repeat the whole process on this equation, taken in the form:

$$U_2^{h2} \ldots U_m^{hm} = (YX)^{k}YV_2^{k2} \ldots V_n^{kn} .$$

If cases (i), (ii) or (iii)(a) occur, the induction is completed. In case (iii)(b), we obtain equations of the form:

$$U_2^{h2} \ldots U_m^{hm} = WYV_2^{k2} \ldots V_n^{kn}$$

or: $WU_3^{h3} \ldots U_m^{hm} = Y(XY)^{k}V_2^{k2} \ldots V_n^{kn}$

both of which can be dealt with as in case (iii)(b_1). ___/

For any schema Q and path/location vector for Q, $\underline{h}$, we defined $Q(\underline{h})$ to be the word-function giving the value of the location corresponding to $\underline{h}$ after the execution of the path $\underline{h}$, and we showed that $Q(\underline{h})$ was P-representable. So suppose β_Q is a P-function, Q* is a lwf, and:

$$(\underline{s},\underline{h})\,[\beta_Q(\underline{s};\underline{h}) \Rightarrow Q(\underline{h}) = Q^*(\underline{s})]$$

For two schemata Q,R, the condition $\underline{diff_{Q,R}(\underline{h},\underline{k})}$ on $\underline{h}$ and $\underline{k}$ that $Q(\underline{h}) \neq R(\underline{k})$ is the P-formula:

$$(\underline{s},\underline{t})\Big[\beta_Q(\underline{s};\underline{h}) \wedge \beta_R(\underline{t};\underline{k}) \quad .\Rightarrow. \neg\, \varphi_{Q^*,R^*}(\underline{s},\underline{t})\Big]$$

where φ_{Q^*,R^*} is the predicate of Theorem 6.1 for the lwf's Q* and R*.

Theorem 6.2 For any two schemata Q,R, there is a P-formula, $\gamma_{Q,R}$, free in the variables $\underline{h},\underline{k}$, such that:

$\gamma_{Q,R}(\underline{h},\underline{k})$ is true if and only if the paths corresponding to $\underline{h}$ and $\underline{k}$ in the schemata Q and R are consistent, and represent execution sequences.

Proof. $\gamma_{Q,R}(\underline{h},\underline{k})$ is the formula:

$$\Big\{ (\underline{h}_0,\underline{h}_1,\underline{k}_0,\underline{k}_1)\Big[T_0^Q(\underline{h}_0,\underline{h}) \wedge T_1^Q(\underline{h}_1,\underline{h}) \ .\Rightarrow.\ \mathrm{diff}_{Q,Q}(\underline{h}_0,\underline{h}_1)$$

$$:\wedge: \ T_0^R(\underline{k}_0,\underline{k}) \wedge T_1^R(\underline{k}_1,\underline{k}) \ .\Rightarrow.\ \mathrm{diff}_{R,R}(\underline{k}_0,\underline{k}_1)$$

$$:\wedge: \ T_0^Q(\underline{h}_0,\underline{h}) \wedge T_1^R(\underline{k}_1,\underline{k}) \ .\Rightarrow.\ \mathrm{diff}_{Q,R}(\underline{h}_0,\underline{k}_1)$$

$$:\wedge: \ T^Q(\underline{h}_1,\underline{h}) \wedge T_0^R(\underline{k}_0,\underline{k}) \ .\Rightarrow.\ \mathrm{diff}_{Q,R}(\underline{h}_1,\underline{k}_0) \Big]$$

$$.:\wedge:. \ \mathrm{fin}_0{}^Q(\underline{h}) \vee \mathrm{fin}_1{}^Q(\underline{h}) \vee \mathrm{inf}^Q(\underline{h})$$

$$.:\wedge:. \ \mathrm{fin}_0{}^R(\underline{k}) \vee \mathrm{fin}_1{}^R(\underline{k}) \vee \mathrm{inf}^R(\underline{k}) \Big\} \quad \text{—/}$$

Theorem 6.3 Given schemata Q,R, in **S**, there is an effective construction of a formula Eq(Q,R) of **P** which is true if and only if Q are R are equivalent.

Proof. Eq(Q,R) is the formula:

$$(\underline{h},\underline{k})\Big[\gamma_{Q,R}(\underline{h},\underline{k}) :\Rightarrow: \ \mathrm{fin}_0{}^Q(\underline{h}) \wedge \mathrm{fin}_0{}^R(\underline{k})$$

$$.\vee. \ \mathrm{fin}_1{}^Q(\underline{h}) \wedge \mathrm{fin}_1{}^R(\underline{k})$$

$$.\vee. \ \mathrm{inf}^Q(\underline{h}) \wedge \mathrm{inf}^R(\underline{k}) \Big] \quad \text{—/}$$

The theory **P** is decidable [5], so we have proved:

Theorem 6.4 The equivalence problem for schemata in the class **S** is recursively solvable.

§7. Free, liberal and progressive schemata

§7.1 Freeing liberal schemata

Definitions. A schema P is free is every sequence through P is an execution sequence.

A schema P is liberal if, in every sequence through P, no expression (under a free interpretation) is computed more than once.

Theorem 7.1 Given any liberal schema P, there is an effective construction of an equivalent free schema P^{φ}. (This construction will be called 'freeing P'.)

Proof. We have only to ensure that no location P is tested more than once by the same test-function between successive assignments to it, for then, any sequence is necessarily consistent and therefore an execution sequence. The construction of P^{φ} is by stages. At each stage, for some u,j, we 'free the schema with respect to $T_u(L_j)$' , in the following way.

Three copies of P are made, P^N, P^0, P^1, say. The addresses corresponding to each 'a' in P will be denoted by a^N, a^0, a^1 respectively and the transfer addresses in each copy altered accordingly. Any test in P of the form:

$T_u(L_j)$ b,c

is replaced in P^N by:

$$T_u(L_j) \quad b^0, c^1$$

in P^0 by:

$$\text{go to } b^0$$

and in P^1 by:

$$\text{go to } c^1.$$

Any computation instruction in P of the form:

$$a. \quad L_j := .....$$

is replaced in both P^0 and P^1 by:

$$\text{go to } a^N.$$

The initial address of the new schema is a_0^N, where a_0 is the initial address of P. Since the part of the new schema corresponding to P^0 is only accessible when the expression in location L_j has already been tested by T_u and $T_u(L_j) = 0$, and similarly for P^1, it is clear that the new schema is equivalent to P. Furthermore, in the new schema L_j can be tested at most once by T_u between assignments.

When we have successfully freed P with respect to each pair of location and test symbol, the result is the new equivalent free schema P^{φ}.

The application of this construction to a simple schema is illustrated in flow-diagrammatic form in Figs. (i), (ii) and (iii) on the following pages. (Note that

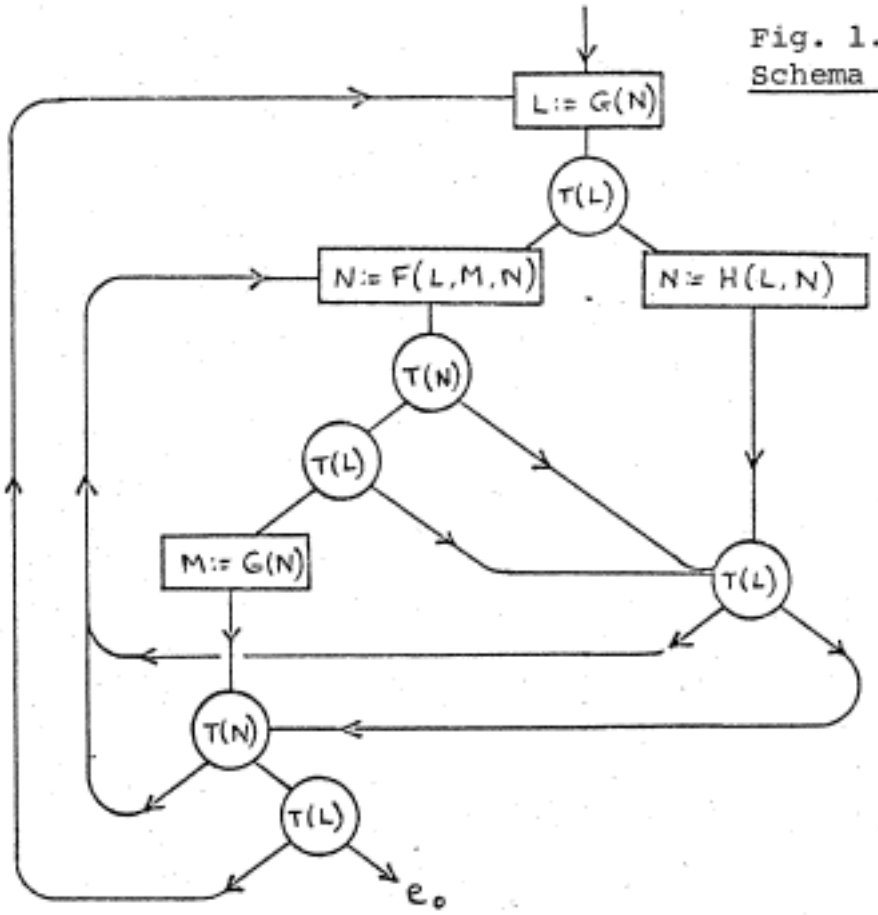


Fig. 1.
Schema P.

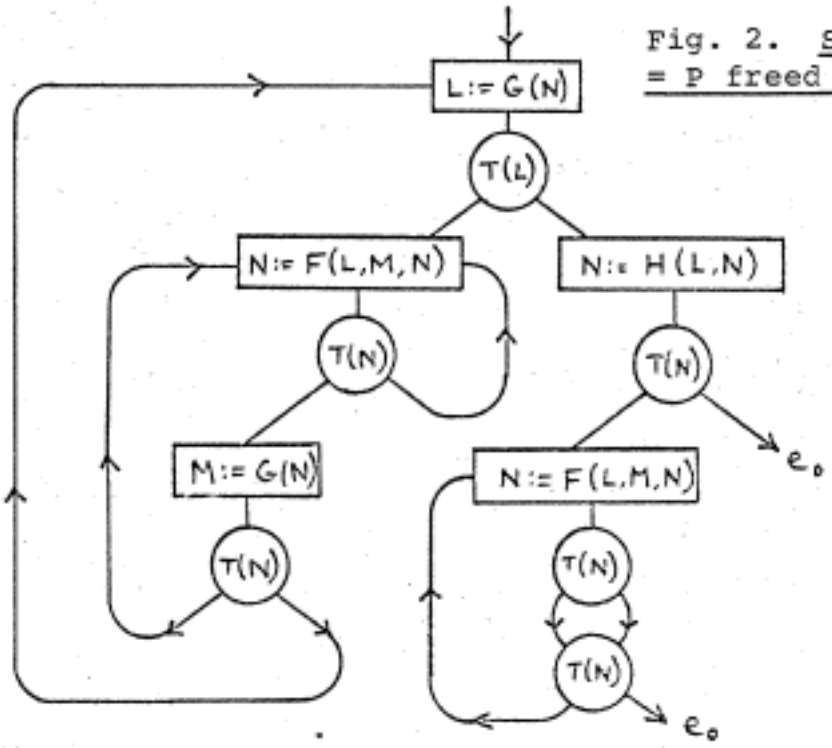


Fig. 2. Schema P'
= P freed w.r.t. T(L).

Fig. 3. Schema P"
= P' freed w.r.t. T(N)

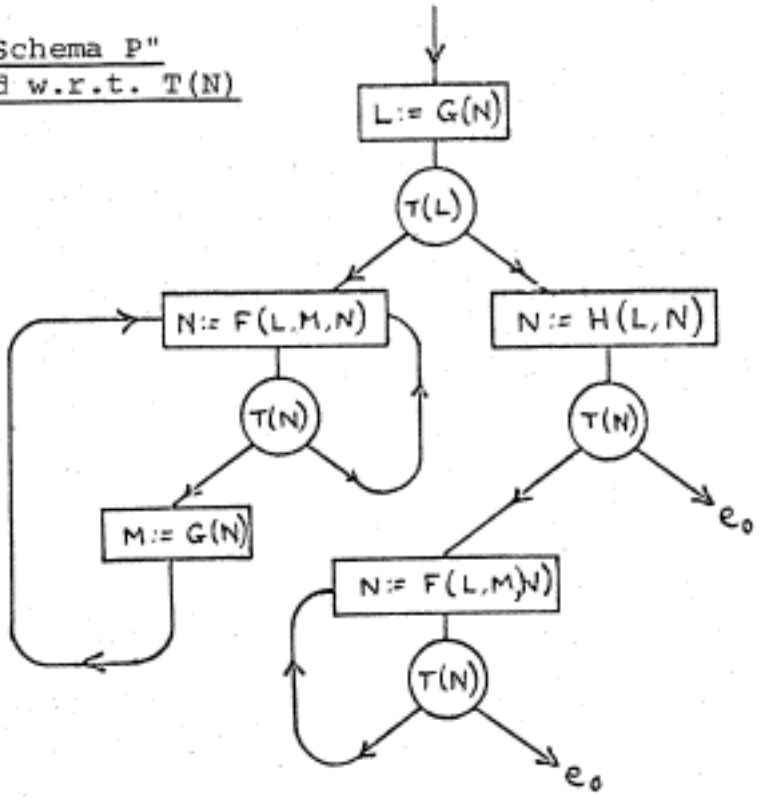


Fig. 4. Abbreviated form of P".

1. L := G(N) π,2
2. N := H(L,N) 3,e_0
3. N := F(L,M,N) 3,e_0

π. goto π

for the sake of clarity various 'inaccessible' parts of the schemata have been omitted.)

Without losing any generality, we may assume that in any schema, no location is tested before it has been assigned to, because given any schema P we can, if necessary, introduce a new monadic operator K, say, and precede the schema by the new instructions:

$L_1 := K(L_1)$

$L_2 := K(L_2)$

.
.
.

etc.

to produce the schema P'. Clearly,

$P \equiv Q$ if and only if $P' \equiv Q'$

and if N(P') is a canonical form for P', then a canonical form for P is produced by just deleting from N(P') all those instructions involving K.

Suppose we take any schema satisfying this condition, and, for each location L, insert after each computation instruction assigning to L the trivial sequence of test instructions,

$T_1(L)$ ·+1, ·+1

$T_2(L)$ ·+1, ·+1

.
.
.

$T_k(L)$ ·+1, ·+1

where $T_1,\ldots,T_k$ are all the test symbols. The resulting schema is equivalent to the original one and we note that, after applying the freeing construction described in Theorem 1, the new schema still has the property that each location is tested in turn by each test symbol after each assignment to it. Following this sequence of tests, the next instruction may be one of up to 2^k successors. For our purposes it is a convenient abbreviation to express the computation and succeeding tests in a single statement as, say:

$$L_j := F_s(L_{j_1},\ldots,L_{j_r}) \quad a_1,a_2,\ldots,a_{2^k}$$

so the next instruction has address a_n where

$$n = 1 + \sum_1^k 2^{r-1}.T_r(L_j)$$

We are therefore considering schemata whose statements are all either of this form or else are unconditional transfers. The latter, however, are clearly eliminable, provided that we allow just one new instruction of the form:

$$\pi.\ \text{goto}\ \pi$$

which provides an equivalent for any loop consisting entirely of unconditional transfers.

At this stage it is convenient to make some further simplifications. The initial address is marked.

All successors of marked addresses are marked and this step is repeated until no further addresses can be marked. The marked addresses are then precisely those which are accessible by a sequence (and so an execution sequence) through the schema. All other instructions can be deleted, without loss of equivalence. In a similar way, though proceeding in the reverse direction, we can find all addresses from which either of the terminal addresses is accessible. All other instructions can be deleted and transfers to them replaced by transfers to π. Finally, if all the successors of an instruction are e_0 (e_1 respectively) then the instruction can be deleted and transfers to it replaced by transfers directly to e_0 (e_1 respectively). In some trivial cases, sensible alternatives to these simplifications are required.

As a result of these considerations we state, without further proof:

Theorem 7.2 There is an effective method whereby, from any liberal schema, an equivalent liberal schema may be constructed with the following properties:

(i) it is free

(ii) it is expressible in the abbreviated form given above,

(iii) any (partial) sequence in it which has not reached π, may be extended to a terminating execution sequence,

(iv) any (partial) sequence in it which has not reached e_0 (e_1 respectively) may be extended to an execution sequence which never reaches e_0 (e_1 respectively).

It seems likely that the decision problem for the equivalence of free schemata is solvable, and certainly none of the techniques we have used so far to prove unsolvability is at all applicable. Theorem 7.1 shows that, for all practical purposes, any liberal schema can be considered to be free, but the reverse is not the case. For example, the free schema:

$L_2 := F(L_1)$

a. $L_1 := F(L_1)$

$L_1 := F(L_1)$

$L_2 := F(L_2)$

$T(L_1)$ $\dot{}+1$, a

$L_2 := G(L_2)$

$T(L_2)$ e_0, e_1

can be readily shown not to be equivalent to any liberal schema. The difference between the two classes is apparent too in the next two theorems.

Theorem 7.3 The property of being free is not recursively solvable.

In the proof of this Theorem we shall require a result of E.L. Post [11]. The 'correspondence problem' is the following:

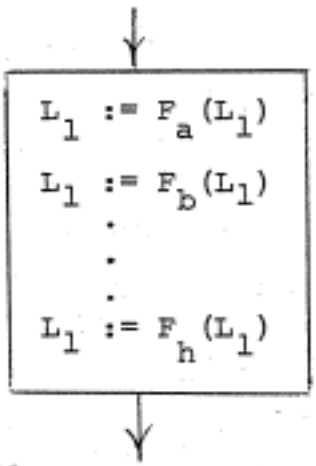
> Given a set of pairs of words $\{\langle U_i, V_i\rangle \mid i = 1,\ldots,n\}$ over a finite alphabet, to decide whether there exists a sequence of indices $i_1, i_2, \ldots i_k$ $(k \geq 1)$ where $1 \leq i_j \leq n$, such that:

$$U_{i_1} U_{i_2} \cdots U_{i_k} = V_{i_1} V_{i_2} \cdots V_{i_k}$$

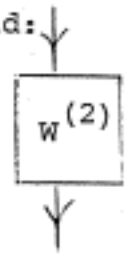
Post proved that the correspondence problem (for an alphabet with more than one letter) is recursively solvable.

Proof of Theorem 7.3. For any word $W = F_a F_b \ldots F_h$ over an alphabet $\{F_0, F_1, \ldots, F_m\}$, let:

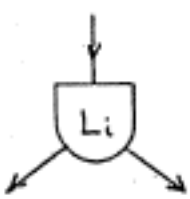
denote the sequence of instructions:



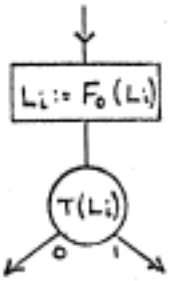
and:



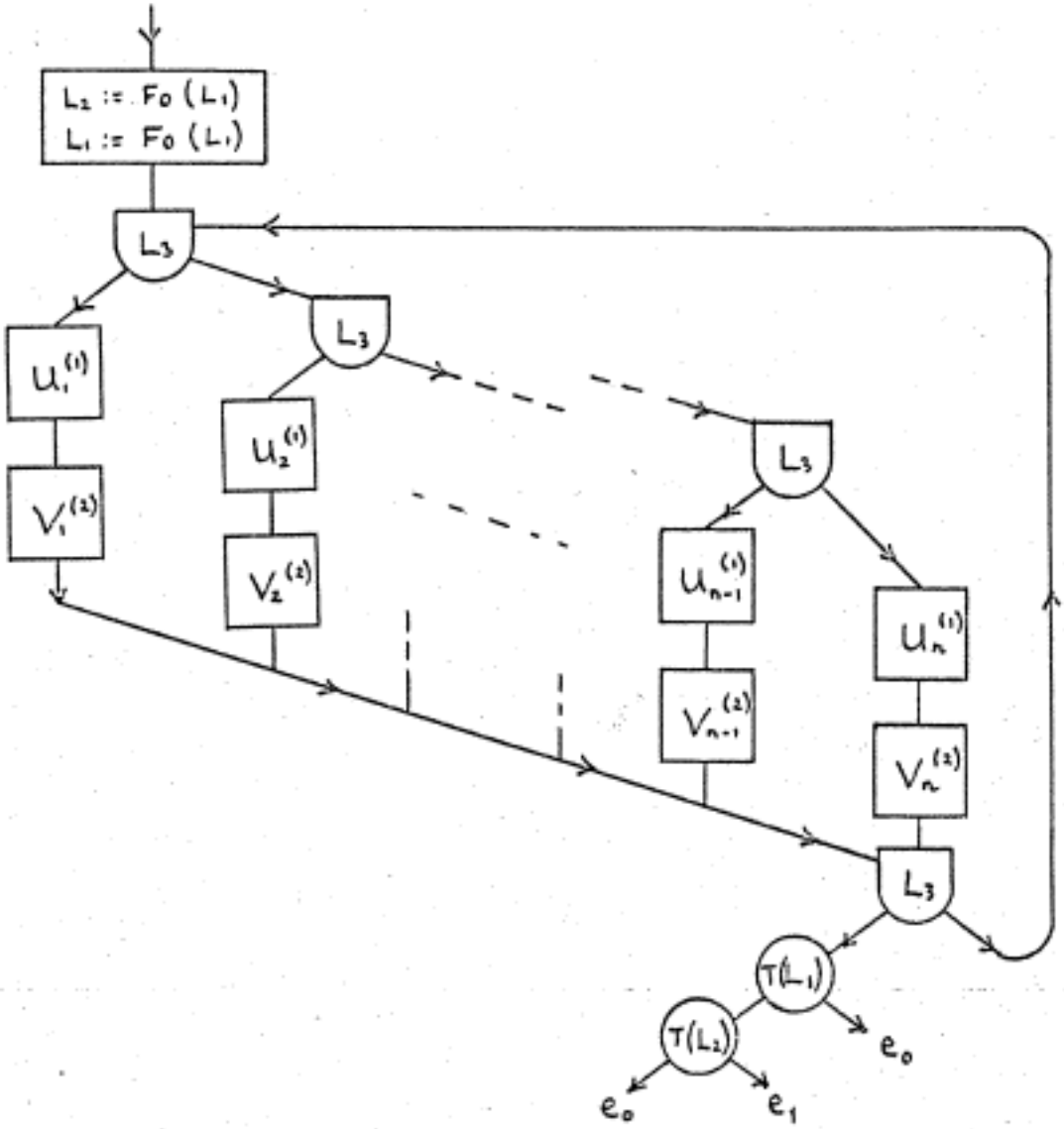
the corresponding sequence with L_2 in place of L_1.



denotes



Consider the schema:



This is evidently free if and only if the correspondence problem $\{ \langle U_1, V_1 \rangle, \ldots, \langle U_n, V_n \rangle \}$ has no solution. ___/

Theorem 7.4 The property of being liberal is recursively solvable.

Proof. For any sequence in which the same expression is computed twice, we consider the first such occurrence. We observe that, for this to happen, the two relevant computation instructions must be identical up to the choice of assignment location, and none of the retrieval locations is assigned to between them. Let us call this an immediate repetition. A schema is therefore illiberal (i.e. not liberal) if and only if there is an immediate repetition in some execution sequence. We can apply the construction of Theorem 7.2 to any schema, and produce an equivalent schema which is free if the original is liberal, but is illiberal if the original is illiberal (because corresponding sequences of computation instructions are the same). The decision procedure is merely to look for basic repetitions in any sequence through this schema. If there is such an immediate repetition, then there must be one where the number of intervening instructions is less than the total number in the schema, therefore the search can obviously be made finite. If no immediate repetition is found, then, if the schema is free it has been found in an execution sequence, and if not this can only be because the original schema was illiberal. ___/

§7.2 Progressive schemata

Definition. A schema P is progressive if, in any sequence through P, the assignment location of any computation instruction is taken as a retrieval location by the next computation instruction, if any.

The 'expression computed by an instruction' in an execution sequence will mean the string which is produced under the corresponding free interpretation.

Lemma 1.

(i) The lengths of successive expressions computed by a sequence through a progressive schema are strictly increasing.

(ii) Given any expression E computable by a sequence through a progressive schema, there is a unique integer n, and a unique sequence s, of expressions

$$s = \langle E_1, \ldots, E_n = E \rangle$$

such that any sequence of a progressive schema which computes E computes precisely the expressions of s in turn.

Proof.

(i) Each expression must contain the previous one as a proper subexpression.

(ii) E is uniquely decomposable into its immediate

subexpressions and either each is an initial symbol, i.e. an L_i, or else one of the subexpressions was the expression computed immediately before E and so is distinguished by being the longest, in view of (i). The result is then proved easily by induction on the length of a sequence which computes E. ___/

Corollary of Lemma 1(i). Any progressive schema is liberal.

Definition. Let depth (E) be the number n of Lemma 1 for suitable E (and undefined otherwise).

Lemma 2. Let t_1, t_2 be sequences through the progressive schema P, which compute E_1, E_2 respectively, and suppose that

$$\text{depth } (E_1) = \text{depth } (E_2)$$

but $E_1 \neq E_2$. Let t_1', t_2' be continuations of t_1, t_2 respectively, which compute E_1' and E_2', say. Then $E_1' \neq E_2'$.

Proof. Trivial from Lemma 1(ii). ___/

Suppose P,Q are free, liberal, progressive schemata with the properties described in Theorem 7.2.

Theorem 7.5 If $P \equiv Q$, then for all interpretations I, the execution sequences $\pi_I^{(P)}$ and $\pi_I^{(Q)}$ compute precisely the same sequence of expressions.

Proof. Suppose $\pi_I^{(P)}$ only computes n expressions in the sequence t_1, which then terminates or reaches the address π, while $\pi_I^{(Q)}$ computes an (n+1)th expression in the sequence t_2. By Theorem 7.2, there is a continuation t_2' of t_2 which behaves differently from t_1, and clearly t_1 and t_2' are consistent. So $P \not\equiv Q$.

Alternatively, suppose the (n+1)th expressions calculated by $\pi_I^{(P)}$ and $\pi_I^{(Q)}$ exist and are different, and denote the sequence thus far by t_1 and t_2 respectively. There is some continuation, t_1' say, of t_1 wnich terminates at $e(=e_0$ or $e_1)$. There is also some continuation t_2' of t_2 which never reaches e. Since t_1 and t_2 are consistent and, by Lemma 2, the sets of expressions computed in the continuation parts of t_1' and t_2' are disjoint, it follows that t_1' and t_2' are consistent, and so $P \not\equiv Q$. These contradictions prove the theorem. ___/

In a §7.4 we shall produce an algorithm to decide the equivalence of any progressive schemata, however it seems desirable to present first the easier decision procedure for progressive schemata of a more restricted class.

§7.3 Full schemata

Definition. A full (or fully progressive) schema is one

in which each computation instruction takes every location as a retrieval location.

Corollary. A full schema is progressive.

Lemma 3. For any two sequences in full schemata which compute the same sequence of expressions, the two corresponding sequences of computation instructions are identical except perhaps for the last instructions of the sequence, whose assignment locations may differ.

Proof. Assume that the result is true for sequences which compute just r expressions and further that for such sequences the two corresponding free computation sequences are identical except perhaps after the last instructions. This hypothesis is trivially true for $r = 1$. Suppose it to be true for $r < k$ $(k > 1)$, and suppose two sequences compute identical sequences of expressions of length k. The elements of each member of the free computation sequence are clearly distinct and each occurs as a subexpression of the next computed expression, so the penultimate computations must replace the same expressions. Hence the penultimate assignment locations are the same and therefore so are the next members of the free computation sequence. The last computation instructions must therefore agree to within choice of assignment location and so the hypothesis holds

for r = k. The result follows by induction. ___/

Now we define an equivalence relation $\sim$, on the statement addresses of schemata in the abbreviated form described in Theorem 7.2.

Definitions. If address a has the successor addresses $a_1,\ldots,a_{2^k}$, then we define:

$$\underline{succ_i(a)} = a_i \qquad \text{for } i = 1,\ldots,2^k$$

The instruction associated with the address a is denoted by inst(a).

Definition of $\sim$.

Clause (i).

$a \sim b$ if and only if inst(a) = inst(b), and for each $i = 1,\ldots,2^k$, $succ_i(a) \sim succ_i(b)$ (or $succ_i(a) = succ_i(b)$ if either one is e_0, e_1 or $*$.)

Clause (ii).

$\sim$ is the weakest relation satisfying Clause (i).

Lemma 4. The definition of $\sim$ is welldefined and effective.

Proof. For certain pairs a,b, Clause (i) gives $a \not\sim b$. This set of pairs can be effectively constructed by stages. At stage n ($n \geq 1$), we add just those pairs

whose inequivalence follows from Clause (i) and, if $n > 1$, the pairs added at the previous stage. When this construction terminates, we can clearly define all the remaining pairs to be equivalent, without obtaining a contradiction from Clause (i). The relation so defined also satisfied Clause (ii). ____/

Lemma 5. $\sim$ (and indeed any relation satisfying Clause (i)) has the 'substitution property', that is,

> 'Given any schema P, with addresses a,b where $a \sim b$, the schema P', obtained by replacing an occurrence of 'a' in a transfer address by 'b', is equivalent to P.'

Proof. Under any interpretation, the execution sequences in P,P' can be shown inductively to have corresponding addresses equivalent and so containing the same instructions. Hence the values of the two sequences are the same. ____/

Construction of the 'canonical form' $\hat{P}$.

Given any full schema P, we can effectively construct an equivalent schema satisfying the conditions of Theorem 7.2. Assuming now the abbreviated form, we replace the assignment location in any statement all of whose successor addresses are π, e_0 or e_1 by some fixed

location, L_1 say. This produces, of course, an equivalent schema. Now we select one from each $\sim$-equivalence class of addresses and replace each transfer address by the chosen address of its class. Repeated application of Lemma 5 shows that the resulting schema is equivalent to the original one. With the representative from the class containing the old initial address as the new initial address, all except the chosen representatives are inaccessible and may be deleted, leaving a set of addresses no two of which are $\sim$-equivalent. This schema with a suitable renaming of the addresses is $\hat{P}$. One method of naming which suggests itself is to label the initial statement with '1', and then proceed by using the lowest integer not yet used, to label the earliest successor of the lowest-labelled statement that is not yet labelled. By Theorem 7.2, each statement is accessible and so gets labelled eventually.

We require to prove that for any equivalence class of schemata the canonical form obtained as described is unique.

Lemma 6. If $P \equiv Q$, then corresponding statement addresses of any consistent pair of execution sequences through P and Q are $\sim$-equivalent.

Proof. Suppose $P \equiv Q$, and that $x = x_1, x_2, \ldots$ and $y = y_1, y_2, \ldots$ are such a pair of sequences and that

$$x_r \sim y_r \quad \text{for } r < n, \text{ but } x_n \not\sim y_n.$$

Then, by the proof of Lemma 4, there is a consistent pair x',y' of extensions of $x_1, \ldots, x_n$ and $y_1, \ldots y_n$ respectively, such that for some $m \geq n$, either $\text{inst}(x'_m) \neq \text{inst}(y'_m)$, or one of x'_m, y'_m is e_0, e_1 or π and $x'_m \neq y'_m$. From Theorem 7.5, Lemma 3, and the consideration that the exceptional case of Lemma 3 has been explicitly prevented, the first case is impossible. By Theorem 7.5, x' and y' reach one of e_0, e_1, π simultaneously and so the second case is a contradiction of $P \equiv Q$. ____/

Theorem 7.6. If P,Q have canonical forms $\hat{P}, \hat{Q}$ as described above then:

$P \equiv Q$ if and only if $\hat{P}$ and $\hat{Q}$ are identical.

Proof. Since a schema and its canonical form are equivalent:

$\hat{P}, \hat{Q}$ identical implies $P \equiv Q$.

For the reverse implication, if $P \equiv Q$, then by Lemma 6, the initial statements of P,Q, and hence of $\hat{P}, \hat{Q}$, are $\sim$-equivalent. By the definition of $\sim$, corresponding successors of equivalent addresses are equivalent, and,

by the construction of the canonical form, there is only one representative of each equivalence class in each schema. Therefore $\hat{P},\hat{Q}$ are the same to within, possibly, renaming of addresses, but both were named in a standard way which depended only on the structure, so $\hat{P},\hat{Q}$ are identical. ____/

Hence we have:

Theorem 7.7. The problem of deciding whether or not two given full schemata are equivalent is recursively solvable.

Corollary. The equivalence problem for schemata which use only one location is recursively solvable.

§7.4 Decision procedure for progressive schemata.

We can assume at once that any progressive schema is already in the abbreviated form guaranteed by Theorem 7.2. For any statement:

$$a. \quad L_j := F_u^t(L_{k_1},\ldots,L_{k_t}) \quad a_1,\ldots,a_{2k})$$

we define:

$$\underline{fn(a)} = F_u^t \ ,$$

the assignment index of a, $\underline{asg(a)} = j$,

the retrieval vector of a, $\underline{Ret(a)} = \langle k_1,\ldots,k_t \rangle$

and denote the ith component of Ret(a) by $\underline{ret_i(a)}$.

A location L_j is alive at address a if there is a sequence through the schema from a:

$$a = b_0,b_1,b_2,\ldots,b_n \quad (n \geq 0)$$

such that:

(i) $j \in Ret(b_n)$

but (ii) $j \neq asg(b_i)$ for any i, $0 \leq i < n$.

(No location is alive at a terminal address.) If there is such a sequence, then there is clearly one of length less than or equal to the number of statements in the schema, so there are effective and efficient algorithms for determining the set:

$$\underline{Live(a)} \underset{def}{=\!=} \left\{ j \,\middle|\, L_j \text{ alive at } a \right\}.$$

For any point $\pi(n)$ of a sequence π through any schema, we define the free configuration, $C(\pi,n)$, as the (m+1)-vector:

$$C(\pi,n) = \langle \pi(n), A(\pi)(n) \rangle$$

and the reduced configuration, $C'(\pi,n)$,

$$C'(\pi,n) = \langle \pi(n), A'(\pi)(n) \rangle$$

where $A'(\pi)(n)(j) = A(\pi)(n)(j)$ if L_j is alive at $\pi(n)$,

$= 0$ otherwise.

Also Range(π,n) is the set of all non-zero elements of $A'(\pi)(n)$, (which are necessarily all distinct).

It is evident that, for any interpretation and for any point of the corresponding execution sequence, the continuation and value of the sequence depend only on the reduced configuration at that point.

Lemma 7. $P \equiv Q$ implies that, for every interpretation I and integer n:

$$\text{Range}(\pi_I^{(P)},n) = \text{Range}(\pi_I^{(Q)},n) \quad .$$

Proof. Suppose π,π' are consistent sequences in P,Q respectively and some expression E is in Range (π,n) but not in Range (π',n). There is a sequence continuing from $\pi(n)$, in which E is taken as an argument in the computation of some expression E', say, but no continuation

from $\pi'(n)$ can ever use E as an argument and so can ever compute E'. Thus there is an interpretation for which the two corresponding execution sequences compute different sequences of expressions, and so, by Theorem 7.5, $P \not\equiv Q$. ____/

Let m be the (fixed) number of location symbols involved in the schemata under consideration, and let S_m be the group of all permutations on m elements. If $\varphi \in S_m$, we define for any vector of integers in the set $\{1,\ldots,m\}$:

$$\varphi(<k_1,\ldots,k_t>) = <\varphi(k_1),\ldots,\varphi(k_t)>$$

and for a set of such integers:

$$\varphi(T) = \{\varphi(j) \mid j \in T\}$$

Definition. Address a is φ-similar to address b, $a \stackrel{\bullet}{=} \varphi(b)$, when:

(I) if a,b are terminal or equal π (the 'dynamic stop'), then a = b;

(II) if a,b are prefixes of statements, then:

(i) fn(a) = fn(b)

(ii) Ret(a) = φ(Ret(b))

(iii) $\underline{\text{either}} \left[\text{asg(a)} = \varphi(\text{asg(b)}) \right]$

$\underline{\text{or}} \left[\text{asg(a)} \notin \bigcup_i \varphi(\text{Live}(\text{succ}_i(\text{b}))) \right.$

$\left. \text{and} \quad \varphi(\text{asg(b)}) \notin \bigcup_i \text{Live}(\text{succ}_i(\text{a})) \right]$

(iv) $\text{Live(a)} = \varphi(\text{Live(b)})$

$\underline{\varphi^{a,b}}$ is defined as $\varphi.(rs)$ where $r = \text{asg(a)}$ and $s = \varphi(\text{asg(b)})$ (i.e. the permutation φ followed by the transposition of r and s).

For any configuration $\alpha = \langle a, \underline{D} \rangle$, we define:

$$\varphi\left[\underline{D}\right] = \langle D_{\varphi^{-1}(1)}, \ldots, D_{\varphi^{-1}(m)} \rangle$$

If configuration β immediately follows α in an execution sequence corresponding to some interpretation I, we write:

$$\alpha \xrightarrow{I} \beta \qquad \text{or just} \qquad \alpha \longrightarrow \beta$$

Let $\alpha = \langle a, \underline{A} \rangle$, $\beta = \langle b, \underline{B} \rangle$, $\gamma = \langle c, \underline{C} \rangle$, $\delta = \langle d, \underline{D} \rangle$ denote general <u>reduced</u> configurations.

<u>Lemma 8</u>. Suppose $\underline{A} = \varphi\left[\underline{B}\right]$, the expressions computed at α and β are the same, and for all interpretations I,

$\text{Range}(\gamma) = \text{Range}(\delta)$ where $\alpha \xrightarrow{I} \gamma$ and $\beta \xrightarrow{I} \delta$

then:

(1) $\underline{C} = \varphi^{a,b}\left[\underline{D}\right]$ for such γ, δ

(2) $a \doteq \varphi(b)$

Proof. Part (1) is trivial. For part (2), the conditions (i) and (iv) of the definition of ' $a \doteq \varphi(b)$ ' are clearly satisfied, and since the non-zero components of any reduced configuration are always distinct, so also is condition (ii). Suppose that:

$$\mathrm{asg}(a) \neq \varphi(\mathrm{asg}(b)) \quad \text{and} \quad \varphi(\mathrm{asg}(b)) \in \mathrm{Live}(c)$$

for some $\gamma = \langle c, \underline{C} \rangle$, say, where $a \longrightarrow \gamma$. Then:

$$\begin{aligned} B_{\mathrm{asg}(b)} &= A_{\varphi(\mathrm{asg}(b))} \\ &= C_{\varphi(\mathrm{asg}(b))} \quad \text{since only } L_{\mathrm{asg}(a)} \text{ is assigned to.} \\ &= D_{(\varphi\, a, b)-1(\varphi(\mathrm{asg}(b)))} \\ &= D_{\varphi^{-1}(\mathrm{asg}(a))} \\ &= B_{\varphi^{-1}(\mathrm{asg}(a))} \end{aligned}$$

which is impossible. A similar argument is followed if $\mathrm{asg}(a) \in \varphi(\mathrm{Live}(d))$, and so condition (iii) is satisfied, and part (2) is proved. ____/

For any two sequences π, π', we defined $\underline{\mathrm{perm}(\pi, \pi', n)}$ by:

$$\mathrm{perm}(\pi, \pi', 0) = \langle\ = \text{identity permutation}$$

$$\mathrm{perm}(\pi, \pi', n+1) = \left[\mathrm{perm}(\pi, \pi', n)\right]^{\pi(n), \pi'(n)}$$

for $n \geq 0$.

Theorem 7.8. For any schemata P,Q: $P \equiv Q$ if and only if for all interpretations I and integers n:

$$\pi_I^{(P)}(n) \doteq \varphi(\pi_I^{(Q)}(n))$$

where $\varphi = \text{perm}(\pi_I^{(P)}, \pi_I^{(Q)}, n)$

Proof. The sufficiency condition follows from Part (1) of the definition of φ-similarity. The necessity is a consequence of Theorem 7.5, Lemma 7, and Lemma 8, and is proved by induction on n. _____/

A triple $\langle a,b,\varphi \rangle$ is valid if $a \doteq \varphi(b)$, and invalid otherwise. A triple $\langle c,d,\varphi' \rangle$ follows a triple $\langle a,b,\varphi \rangle$ if $\varphi' = \varphi^{a,b}$, and for some i, $c = \text{succ}_i(a)$ and $d = \text{succ}_i(b)$. A chain of triples is a sequence $x_1, x_2, x_3, \ldots$ of triples, where x_{n+1} follows x_n for all $n \geq 1$.

Theorem 7.8 can be alternatively expressed as:

' $P \equiv Q$ if and only if there is no chain starting from $\langle p_0, q_0, (\ ) \rangle$, where p_0, q_0 are the initial addresses of P,Q respectively, and containing an invalid triple. '

Since it is effectively determinable whether or not a given triple is valid, and there are only a finite number of triples involved, we have:

Theorem 7.9. It is recursively solvable, given any two progressive schemata P and Q, whether or not $P \equiv Q$.

§7.5 Conservative schemata

A conservative schema is one in all of whose computation instructions the assignment location appears as a retrieval location, e.g.:

$$L_2 := H(L_3, L_2, L_2)$$

Lemma. Any conservative schema is liberal.

Proof. The proof follows by induction when we note that:

(i) the expressions associated with the locations are all distinct,

(ii) whenever any new expression is computed, one of its subexpressions is lost by being 'overwritten', so the computation cannot be repeated. ____/

If we delete the first instruction:

$$L_2 := F(L_1)$$

from each of the simulation schemata of §5, the resulting schemata are conservative. There is an obvious isomorphism between conservative schemata with a single monadic function symbol F, say, and m locations and m-tape finite automata. Given such a schema, we prefix it with the new sequence of instructions:

$$L_1 := F(L_1)$$

$$\vdots$$

$$L_m := F(L_m)$$

and then we can construct the equivalent free schema, as in Theorem 7.2. To each abbreviated statement:

$$a_0.\ \ L_i := F(L_i) \qquad a_1,\ldots,a_{2k}$$

of this schema, we make correspond the state q_0 in an m-tape automaton with the transitions:

state	head	next state
		$s_1\ s_2\ \cdots\ s_{2k}$
q_0	i	$a_1\ a_2\ \cdots\ a_{2k}$

The trivial loop at π corresponds to a trivial loop in the automaton. The reverse correspondence is obvious.

Theorem 7.10. Two such schemata are equivalent if and only if their corresponding m-tape automata are equivalent (that is, accept and reject the same sets of tapes).

Proof. We just extend the correspondence by noting that an m-tape automaton accepts, rejects, diverges on, respectively, an m-tuple of tapes, just as the corresponding schema succeeds, fails or diverges, under the

interpretation which corresponds with the m-tuple in the obvious way. Any interpretation corresponds to an m-tuple of tapes. _____/

The equivalence problem for multitape finite automata is well known, but to date is unresolved.

SUMMARY

In §2 and §3 we introduced program schemata, gave some justification of the reasons behind our definitions, and began to discover the sort of problems we ought to investigate. If the formalism is to provide the basis of an adequate optimization procedure, it is preferable that the equivalence problems involved be at least partially solvable. In Part II we began to chart the unsolvable domains that would have to be avoided. Some of the general conclusions we can draw, as a result of this, are:

(1) By electing to restrict the range of possible interpretations in some general way, we can gain, at most, some partial solvability results.

(2) It avails us nothing in solvability to reduce the number of function and test symbols to the minimum which leaves a nontrivial language, and unsolvability is with us already in schemata with just two locations.

(3) We do not require arbitrarily complicated loop-structures in the flow-diagrams for unsolvability, but we can hope to find decision procedures under certain nontrivial constraints on the structure.

In Part III we started to investigate some of the possible solvable domains. In §6 for example, we found

a decision procedure for a class of schemata suggested by (3). While this is encouraging, we shall have to extend this result somewhat, before it can have very general application. A very noticeable feature of the schemata used to prove unsolvability results in Part II is the high degree of repetition of computation and testing of values. This suggested that some attention be given to 'free' schemata, which never test the same value twice, and to 'liberal' schemata, which never compute the same value twice. In §7.1 we derived some information about the relationship between these different classes, but we required the further restriction to 'progressive' schemata, before a decision procedure emerged (§7.4). The final short §7.5 on 'conservative' schemata merely serves to introduce a class of schemata, with a potentially solvable decision problem, and which is wide enough to encompass a usefully large variety of schemata. A solution to the equivalence problem for multitape automata would be a major step toward a decision procedure for these schemata, and we await this solution with interest.

The need now arises to develop techniques by which actual computer programs may best be expressed in the form of program schemata, and if possible program schemata in a solvable domain. This abstraction may be carried out at

several levels in turn. At the 'deepest' level, we may express single 'machine-orders', or even parts of such orders, as schema statements and consider simplifications of the program, a small portion at a time. For 'straight-through' sequences of instructions, existing compilers use various ad hoc techniques for, say, eliminating redundant orders or collecting together common subexpressions of expressions to be calculated, and it would be valuable to have a formalism in which to express these techniques, and to develop new ones in a 'machine-independent' form. At 'shallower' levels of simplification, we might use one schema statement to represent a whole subprogram or routine, operating perhaps on complex data structures. Abstraction to a schema would allow the convenient manipulation of these blocks of instructions and permit simplifications of a grosser kind.

These ideas have provided the motivation for the present research, but we do not discuss the practical applications in any detail in this dissertation. We have set down a theoretical basis of this model of computation, investigated some of its immediate ramifications and indicated some of the directions in which we hope to progress.

References

[1] COOPER, D. 'Mathematical Proofs about Computer Programs.' Machine Intelligence I. Ed: Collins and Michie. (Oliver and Boyd 1967) pp 17-28.

[2] DAVIS, M. Computability and Unsolvability. (McGraw-Hill 1958).

[3] ELGOT, C. and RUTLEDGE, J. 'RS-Machines with Almost Blank Tape.' Journal of the Association for Computing Machinery 11 3 (July 1964) pp 313-337.

[4] HERMES, H. Enumerability, Decidability, Computability. English translation. (Springer-Verlag 1965).

[5] HILBERT, D. and BERNAYS, P. Grundlagen der Mathematik. Reprinted from German edition. (Edwards Brothers Inc. 1944).

[6] IANOV, Iu. 'The Logical Schemes of Algorithms.' (Russian) Problems of Cybernetics I. (1958) pp 75-127. English translation (Pergamon Press Ltd. 1960) pp 82-140.

[7] LUCKHAM, D. and PARK, D. 'The Undecidability of the Equivalence Problem for Program Schemata.' Report No. 1141. Bolt Beranek and Newman Inc.

[8] MARILL, M. 'Computational Chains and the Simplification of Computer Programs.' IRE Transactions on Electronic Computers. EC-11, 2 (April 1962) pp 173-180.

[9] MC CARTHY, J. 'Towards a mathematical theory of computation.' Proceedings of IFIP Congress 1962, pp 21-28.

[10] MC CARTHY, J. 'A basis for a mathematical theory of computation,' Computer Programming and Formal Systems. Ed: Braffort and Hirschberg (North-Holland 1963) pp 33-70.

[11] POST, E. 'A variant of a recursively unsolvable problem,' Bulletin of the American Mathematical Society 52 (1946) pp 264-268.

[12] PRESBURGER, M. "Uber die Vollständigkeit eines gewissen Systems der Arithmetic ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt.' Spawozdanie z I Kongresu Matematykow Krajow Slowianskich, Warsaw 1930, pp 92-101.

[13] RABIN, M. and SCOTT, D. 'Finite Automata and Their Decision Problems.' IBM Journal of Research and Development 3, 2 (April 1959) pp 114-125.

[14] ROSENBERG, A. ' On Multi-Head Finite Automata.' IBM Journal of Research and Development. 10, 5 (September 1966) pp 388-394.

[15] RUTLEDGE, J. 'On Ianov's Program Schemata.' Journal of the Association for Computing Machinery. 11, 1 (January 1964) pp 1-9.