JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 119–160 (1973)

Program Schemes, Recursion Schemes, and Formal Languages

STEPHEN J. GARLAND* AND DAVID C. LUCKHAM**

Computer Science Department, University of California, Los Angeles, California

Received September 3, 1971; Revised May 26, 1972

This paper presents general methods for studying the problems of translatability between classes of schemes and equivalence of schemes in a given class. There are four methods: applying the theory of formal languages, programming, measuring the complexity of a computation, and "cutting and pasting." These methods are used to answer several questions of translatability and equivalence for classes of program schemes, program schemes augmented with counters, and recursively defined schemes. In particular, it is shown that (i) the quasirational recursion schemes are translatable into strongly equivalent program schemes, (ii) monadic recursion schemes are translatable into strongly equivalent program schemes with two counters, (iii) there is a monadic recursion scheme not strongly equivalent to any program scheme with one counter.

INTRODUCTION

The study of program schemes has at least three principle goals in view. First, to present a precise formal model of the notion of a computer program in a way that is entirely independent of features or workings of any (real or abstract) computing machinery. The schemes under consideration embody just those features or constructs of programming languages that appear to be "essential"—whether or not they are in some sense essential should be an outcome of the study. A second is to develop a basic theory of program optimization, again independently of any particular machine or programming language. And a third objective is that of finding general methods (e.g., systems of rules of inference, or transformations on programs), for checking or verifying a given program against its specifications. These goals are, of course, not independent. Results from a study of optimization almost certainly have application to the correctness problem and conversely.

An essential theme to all three of these goals is the question of translating programs having one set of properties into programs having different properties. A translatability theorem—one which says that any scheme in one class can be translated into an

* Mathematics Department, Dartmouth College, Hanover, NH.
** Computer Science Department, Stanford University, Stanford, California.

equivalent scheme in another class—results in our knowing how to replace certain features of programs by others. This may provide normal forms, it may help optimization, and it may reduce a correctness problem to one we already know how to do. On the other hand, a nontranslatability theorem—saying in effect that certain features cannot be replaced by others—provides us with some insight into that intuitive but rather elusive notion, the "power of expression" of a programming language.

In this paper we study translations between schemes and related equivalence problems, i.e., problems of determining whether two schemes in a given class are equivalent. In particular, we are concerned with the question of developing methods for attacking these problems. It seems to us that some "general methods" ought to exist or, at least, a general form ought to be given to the existing "tricks." Essentially, each section of this paper contains some of the applications we have been able to find of a particular method of attack. There are four methods: applying the theory of formal languages, programming, measuring the complexity of a computation, and "cutting and pasting" (roughly speaking, the application of a method due to Rabin and Scott [7] to bound the search necessary to determine if two schemes will behave differently).

Although a few of the results are true for some general notion of scheme, we have nearly always had in mind, for the sake of illustration, just three specific classes of schemes: program schemes, program schemes augmented with counters, and recursively defined schemes. The greater part of the study is restricted to monadic schemes (i.e., schemes containing only functions and predicates of a single argument), and in the case of recursively defined schemes, we are concerned almost exclusively with the single variable monadic recursion schemes introduced by de Bakker and Scott in [1].

Section 1 contains definitions, terminology, and a diagram summarizing most of the translatability and nontranslatability results of later sections. Section 2 deals with the applications of formal languages. Two kinds of languages are associated with a given monadic scheme and several nontranslatability results are then shown to follow immediately from standard theorems in the theory of formal languages. In addition, certain natural classes of recursion schemes (e.g., the linear recursion schemes) can be given precise definitions by reference to an associated language. Also, certain results in the theory of formal languages suggest analogous results about schemes; e.g., the Chomsky normal form theorem for context-free languages leads to the fact that any monadic recursion scheme is equivalent to one in which the terms of the defining equations contain at most two defined function letters.

Section 3 presents results obtainable by programming techniques, and these include the main translatability theorems. We study two definitions of translatability. The most natural one seems to be to say that scheme P is a translation of scheme Q if the two schemes produce the same output under any interpretation (i.e., P is strongly equivalent to Q). This places no restriction on the order in which computations

are carried out, but requires only that the outputs, if any, be the same. It is shown that the quasirational recursion schemes (the smallest class of single variable monadic recursion schemes containing the linear schemes and closed under functional substitution) are translatable into monadic program schemes. Also, all the single-variable monadic recursion schemes are translatable into program schemes augmented with two counters, a result which does not hold for binary recursion schemes. Programming techniques are also used to show that the class of value languages of program schemes is exactly the class of all recursively enumerable languages. This not only settles a small amount of controversy, but it also implies that the methods of Section 2 do not have as many applications as might be hoped. A weaker notion of translation is studied which says that P is a translation of Q if Q is an "inessential" extension of P. It turns out that any single-variable monadic recursion scheme is translatable in this weaker sense into a program scheme.

Section 4 deals with complexity of computations. An example is given of a single variable monadic recursion scheme which is not strongly equivalent to any program scheme nor any program scheme augmented with one counter. Essentially this improves the elegant example of a binary recursion scheme given in Paterson and Hewitt [6]. It also lends some plausibility to the guess that the quasirational schemes may be the largest class of recursion schemes translatable into program schemes.[1]

Section 5 presents applications of the "cutting and pasting" method. It is shown that the equivalence problem is solvable for the class of linear recursion schemes (which is a partial answer to a question raised by de Bakker and Scott in [1]) and for the class of Ianov schemes with constant functions.

During the course of the paper we raise several questions which have not yet yielded to our methods. We would be interested to know whether solutions can be obtained by our techniques or whether other general methods are required. For example, a finer analysis of the complexity of computations seems to be required to show that there is a monadic program scheme with a single counter which is not equivalent to any monadic program scheme without a counter.[1] On the other hand, it seems that the methods of Section 5 ought to yield a positive solution to the equivalence problem for more general classes of recursion schemes.

[1] D. A. Plaisted (Flowchart Schemata with Counters, *in* "Proceedings of Fourth Annual ACM Symposium on Theory of Computing," pp. 44–51) has shown recently in answer to our question that, contrary to our expectations, any monadic program scheme augmented by a single counter is equivalent to a monadic program scheme without a counter. It follows that the nonquasirational recursion scheme in Fig. 1 of Section 1 is equivalent to a program scheme, therby disproving the conjecture that the quasirational schemes are the largest class of recursion schemes translatable into program schemes.

1. Definitions

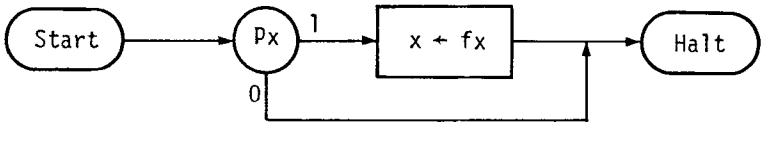
With a few indicated exceptions, we follow the definitions and notation of Hopcroft and Ullman [2] with regard to strings, languages, and automata. The zero-length (or empty) string is denoted by λ.

Schemes are abstract models of computer programs. Given a set $\mathscr{V}$ of variables (usually denoted by $x, y, z, \ldots$), a set $\mathscr{F}$ of *basis* functions (usually denoted by $f, g, \ldots$) which may be used to assign new values to the variables, and a set $\mathscr{P}$ of predicates (usually denoted by $P, Q, \ldots$), a scheme specifies the order in which computations involving the basis functions are to be performed in terms of the values of the predicates. We shall be interested primarily in two classes of schemes, namely, program schemes and recursion schemes.

A program scheme is a finite list of instructions of the form (a) assign variable x the value $fx_1, \ldots, x_n$, where $f \in \mathscr{F}$ is an n-ary function and $x_1, \ldots, x_n \in \mathscr{V}$, and proceed to the next instruction, (b) execute instruction i next if P is true of $x_1, \ldots, x_n$, where $P \in \mathscr{P}$ is an n-ary predicate and $x_1, \ldots, x_n \in \mathscr{V}$, otherwise execute instruction j next, or (c) halt. Details concerning program schemes may be found in Luckham, Park, and Paterson [4]. We remind the reader here that program schemes have natural representations as flowcharts; e.g., the scheme

1. execute 2 next if Px, otherwise execute 3,
2. assign x the value fx,
3. halt,

can be represented by the flowchart



We will represent a sequence of successive assignments by a single assignment of a composition of the functions involved. Thus arbitrary terms composed of basis function letters may appear in a single assignment box in diagrams.

We shall not give a general definition of the class of recursion schemes since we shall be concerned primarily with a subclass which possesses a simple representation. Call a scheme *monadic* if it involves only monadic (i.e., unary) functions and predicates. Then a *monadic recursion scheme* is a finite list of definitional equations

$$\begin{aligned} F_1x &:= \text{if } P_1x \text{ then } \alpha_1x \text{ else } \beta_1x \\ &\vdots \\ F_nx &:= \text{if } P_nx \text{ then } \alpha_nx \text{ else } \beta_nx, \end{aligned}$$

where $F_1,...,F_n$ are new *defined* function symbols, $P_1,...,P_n$ are (not necessarily distinct) predicates, and $\alpha_1, \beta_1,..., \alpha_n, \beta_n$ are (possibly empty) strings of defined and basis function symbols. For example,

$$Fx := \text{if } Px \text{ then } fx \text{ else } FFfx \tag{1.1}$$

is a monadic recursion scheme.

Schemes compute relative to *interpretations* which fix the meanings of the functions, predicates, and variables. An interpretation I consists of a nonempty set dom_I, called the *domain* of I, and assigns to each $f \in \mathscr{F}$ a function f_I over dom_I, to each $P \in \mathscr{P}$ the characteristic function of a relation P_I over dom_I, and to each $x \in \mathscr{V}$ an initial value $x_I \in \text{dom}_I$. An interpretation I of monadic schemes is *free* if $\text{dom}_I = \mathscr{F}^* \cdot \mathscr{V}$ (the set of all strings of basis function symbols followed by a variable), $f_I(a) = fa$ for all $f \in \mathscr{F}$ and $a \in \text{dom}_I$, and $x_I = x$ for all $x \in \mathscr{V}$. For any I, $(f_1 \cdots f_n x)_I = (f_1)_I(\cdots (f_n)_I(x_I))$.

The definition of a *computation* of a program scheme S under an interpretation I is straightforward and can be found in Luckham *et al.* [4]. The *value* $\text{val}_I(S)$ of S under I is the final value of a distinguished output variable, usually denoted by x, if the computation of S under I halts, and is undefined otherwise. (The other variables of S are sometimes referred to as *program* or *auxiliary* variables.)

The notion of a computation of a monadic recursion scheme can be defined concisely if we borrow some notation from the theory of formal grammars. With each such scheme

$$E: F_i x := \text{if } P_i x \text{ then } \alpha_i x \text{ else } \beta_i x \qquad (1 \leqslant i \leqslant n)$$

we associate a context-free grammar G with terminal symbols being the basis function symbols in $\mathscr{F}$, nonterminal symbols $F_1,...,F_n$, and productions $F_i \to \alpha_i$ and $F_i \to \beta_i$ for $1 \leqslant i \leqslant n$. Now let I be an interpretation. A computation of E under I corresponds to the unique rightmost derivation in the grammar G which is *legal* in the following sense: an atomic derivation $\gamma F_i w \Rightarrow_G \gamma\delta w$, where $\gamma, \delta \in (\mathscr{F} \cup \{F_1,...,F_n\})^*$ and $w \in \mathscr{F}^* \cdot \mathscr{V}$ is legal if $\delta = \alpha_i$ and $(P_i)_I(w_I) = 1$ or if $\delta = \beta_i$ and $(P_i)_I(w_I) = 0$. If $F_1 x \Rightarrow_G{}^* w$ by a legal rightmost derivation under I for some terminal string $w \in \mathscr{F}^* \cdot \mathscr{V}$, then $\text{val}_I(E) = w_I$; otherwise $\text{val}_I(E)$ is undefined.

Alternatively, we can visualize the computation of a monadic recursion scheme E as being carried out by a program scheme S with a single variable and a pushdown stack which can contain a finite string of symbols. At the start, the stack contains the single function letter F_1, and at any step in the computation the stack contains a string of basis and defined function letters yet to be applied to the current value of the variable. The scheme S is programmed to remove and examine the top letter on its stack. If this letter is F_i, then S performs the test P_i on the current value of its variable; if the value is 1, it places α_i on the stack (rightmost symbol on top);

if the value is 0, it places β_i on the stack. If the letter is a basis function f, S applies f to the current value of its variable. The computation halts when the stack becomes empty.

As can be seen above, the defined function F_1 plays a primary role in the recursion scheme E, while $F_2, \ldots, F_n$ act as auxiliary functions. At times, when we wish to emphasize or distinguish which defined function F_i plays this distinguished role, we refer to E as $E(F_i)$.

Various classes of schemes can be obtained as subclasses of the classes of program and recursion schemes. *Single variable* schemes, as the name suggests, involve but a single variable; all monadic recursion schemes are single-variable schemes. Monadic recursion schemes may be classified further by their associated grammars: linear recursion schemes have linear context-free grammars (i.e., the strings in the right-hand side of productions contain at most one nonterminal symbol), right-linear schemes have right-linear grammars, etc.

We shall be interested in studying the relative power of various classes of schemes. Two schemes R and S are (*strongly*) *equivalent*, written $R \equiv S$, if and only if for any interpretation I, $\mathrm{val}_I(R) \simeq \mathrm{val}_I(S)$ ($\simeq$ means that either both values are defined and equal, or both are undefined). A class of schemes $\mathscr{R}$ is *translatable* into a class of schemes $\mathscr{S}$, (notation: $\mathscr{R} \to \mathscr{S}$) if, for every $R \in \mathscr{R}$, there is a strongly equivalent $S \in \mathscr{S}$. It is well known, for example, that single-variable program schemes are translatable into monadic right-linear recursion schemes by assigning to each instruction in the program scheme a defined function F_i with defining equation

(a) $F_i x := F_{i+1} f x$ if instruction i is "$x \leftarrow fx$" (this apparently illegal equation is merely an abbreviation for $F_i x :=$ if Px then $F_{i+1} f x$ else $F_{i+1} f x$),

(b) $F_i x :=$ if Px then $F_j x$ else $F_k x$ if instruction i is "execute j if Px is true, otherwise execute k," or

(c) $F_i x := x$ if instruction i is "halt."

Two classes $\mathscr{R}$ and $\mathscr{S}$ of schemes are *intertranslatable* (notation: $\mathscr{R} \rightleftarrows \mathscr{S}$) if $\mathscr{R} \to \mathscr{S}$ and $\mathscr{S} \to \mathscr{R}$.

Proofs of equivalence and translatability are facilitated by the following lemma.

(1.2) LEMMA. *For any schemes R and S: $R \equiv S$ if and only if $\mathrm{val}_I(R) \simeq \mathrm{val}_I(S)$ for all free interpretations I.*

Proof. Necessity is obvious. For sufficiency, suppose $R \not\equiv S$. Then for some interpretation I, $\mathrm{val}_I(R) \not\simeq \mathrm{val}_I(S)$. Define a free interpretation I' by setting $P_{I'}(a) = P_I(a_I)$ for all $P \in \mathscr{P}$ and $a \in \mathscr{F}^* \cdot \mathscr{V}$. Then $\mathrm{val}_I(R) \simeq (\mathrm{val}_{I'}(R))_I$ and $\mathrm{val}_I(S) \simeq (\mathrm{val}_{I'}(S))_I$ since I and I' are "isomorphic" interpretations. (A formal proof of this fact would require a formal definition of the notion of a scheme—an abstraction we choose

to avoid in this paper. The reader may supply proofs for program and recursion schemes.) Therefore $\mathrm{val}_{I'}(R) \not\simeq \mathrm{val}_{I'}(S)$.

The power of a class of schemes is sometimes increased if the schemes can be augmented by the addition of new basis functions or predicates with fixed or restricted interpretations. For example, program schemes could be augmented by an identity function [i.e., a basis function f such that for any interpretation I and any $a \in \mathrm{dom}_I$, $f_I(a) = a$], though as we shall see in Section 3, all uses of such a function can be eliminated. Schemes can also be augmented by constant functions (i.e., basis functions f such that f_I is constant for any I) or by counters. We sometimes refer to constants as *resets*. A *counter* in a program scheme is a variable which is not assigned a value in the usual way, nor can it be used in assignments of values to other variables; rather, there are two new basis functions *inc* (increment) and *dec* (decrement) which may be used to change its value, and a new predicate *zero*. An interpretation I is suitable for schemes with counters if dom_I includes all nonnegative integers, $\mathrm{inc}_I(n) = n + 1$, $\mathrm{dec}_I(n + 1) = n$, $\mathrm{dec}_I(0) = 0$, $\mathrm{zero}_I(n + 1) = 0$, and $\mathrm{zero}_I(0) = 1$. The program scheme with counter in Fig. 1 is strongly equivalent to the recursion scheme in Example 1.1.

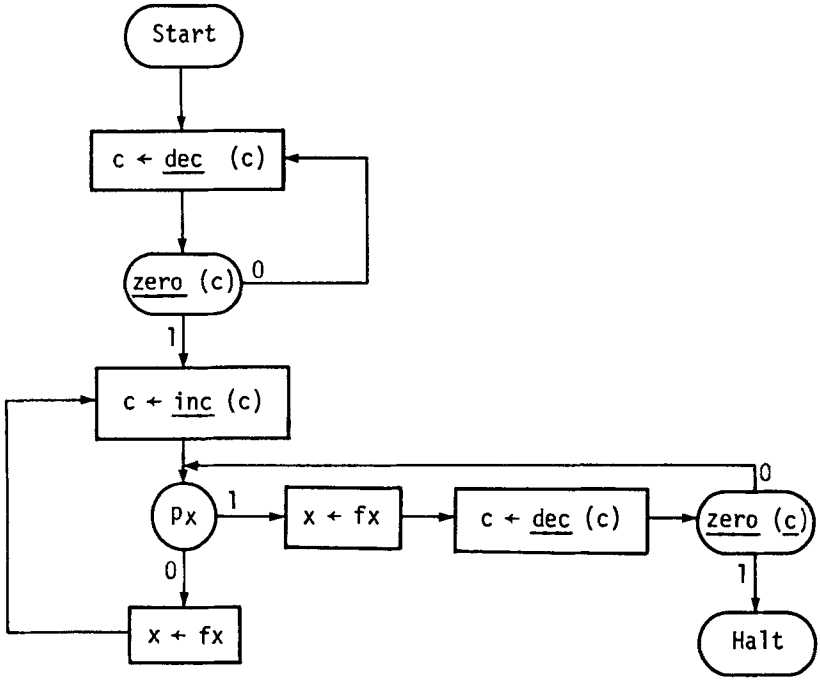


FIG. 1. Program scheme with counter equivalent to $Fx := \textit{if } Px \textit{ then } fx \textit{ else } FFfx$.

Our primary concern in this paper is to illustrate several techniques of general applicability in the study of translatability of schemes. Unless otherwise stated, all schemes considered are unaugmented monadic schemes; we shall not write the modifier "monadic" henceforth, except for emphasis. Figure 2 summarizes most of the translatability and nontranslatability results established in Sections 2, 3, and 4; numbers next to arrows refer to the appropriate theorems; D next to an arrow indicates the result is true by definition.

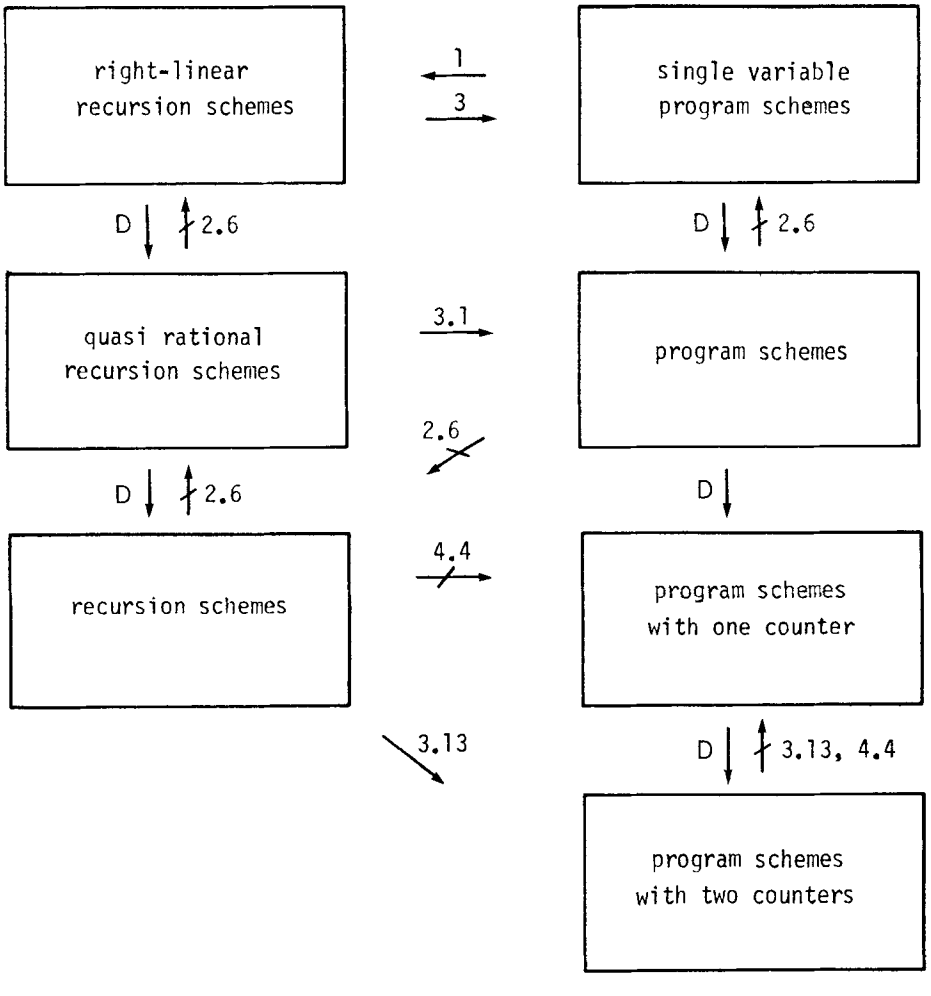


FIG. 2. Summary of translatability results for monadic schemes.

2. Value Languages

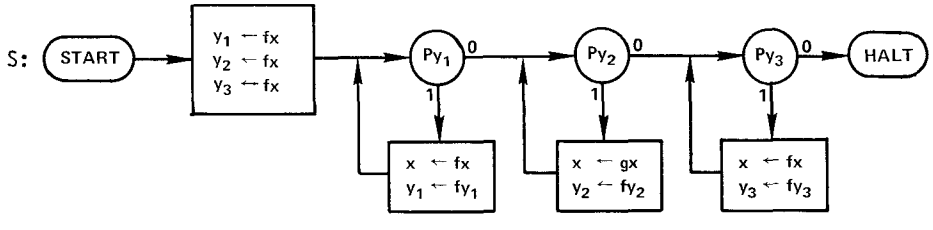
Many decidability and translatability results about classes of monadic schemes can be obtained as immediate consequences of known results in automata theory or mathematical linguistics. We shall use the simple device of assigning to each scheme in a given class a certain formal language, called its value language; then we apply appropriate facts about the classes of value languages so obtained. While some of the results in this section are well known, we still present proofs of them here to illustrate both the power and ease of application of the method of value languages.

Specifically, for any monadic scheme S involving function letters in some set $\mathscr{F}$ and output variable x, the *value language* $L(S)$ of S is the sublanguage of $\mathscr{F}^*$ consisting of all strings a such that $\mathrm{val}_I(S) = ax$ for some free interpretation I. For example, the set $\{a\tilde{a}: a \in \{f, g\}^*\}$ of even-length palindromes is the value language of the linear recursion scheme

$$E: \quad \begin{aligned} F_1x &:= \text{if } Px \text{ then } x \text{ else } F_2x \\ F_2x &:= \text{if } Qx \text{ then } fF_1fx \text{ else } gF_1gx, \end{aligned} \tag{2.1}$$

while $\{f^n g^n f^n: n \geqslant 0\}$ is the value language of the program scheme

(2.2)



A particular application of the technique of value languages will be to show that the recursion scheme E is not strongly equivalent to any single-variable program scheme and that the program scheme S is not strongly equivalent to any recursion scheme. To this end, for any class $\mathscr{S}$ of schemes, we consider the class

$$L(\mathscr{S}) = \{L(S): S \in \mathscr{S}\}$$

of value languages of schemes in $\mathscr{S}$ and prove a theorem which, despite its triviality, has a surprising number of consequences.

(2.3) Theorem. *For any schemes S and T: if $S \equiv T$, then $L(S) = L(T)$.*

Proof. By symmetry it suffices to show that $L(S) \subseteq L(T)$. Suppose $a \in L(S)$. Then $ax = \mathrm{val}_I(S)$ for some free interpretation I. Since $S \equiv T$, $\mathrm{val}_I(S) = \mathrm{val}_I(T)$, and hence $a \in L(T)$.

(2.4) COROLLARY. *For any classes $\mathscr{S}$ and $\mathscr{T}$ of schemes:*

(a) *if $\mathscr{S} \to \mathscr{T}$, then $L(\mathscr{S}) \subseteq L(\mathscr{T})$; and*

(b) *if $\mathscr{S} \rightleftarrows \mathscr{T}$, then $L(\mathscr{S}) = L(\mathscr{T})$.*

In order to apply the corollary to the schemes in (2.1) and (2.2), we first evaluate $L(\mathscr{S})$ for several interesting classes $\mathscr{S}$ of schemes.

(2.5) THEOREM. *Let $\mathscr{S}$ be the class of all* (a) *single-variable program schemes,* (b) *program schemes,* (c) *right-linear recursion schemes,* (d) *linear recursion schemes, or* (e) *recursion schemes which involve function letters in some set $\mathscr{F}$; then $L(\mathscr{S})$ is the class of all* (a) *regular,* (b) *recursively enumerable,* (c) *regular,* (d) *linear context-free, or* (e) *context-free sublanguages of $\mathscr{F}^*$.*

Proof. A direct proof of (a) is left to the reader since (a) follows from (c) and the intertranslatability of single-variable program schemes and right-linear recursion schemes to be established in Section 3. Similarly, a proof of (b) is deferred to Section 3. We prove (e) here and observe that an inspection of the proof also establishes (c) and (d).

We show first that any recursion scheme E has a context-free value language. Suppose that E is the recursion scheme involving function letters in $\mathscr{F}$ and predicate letters in $\mathscr{P}$ given by the system of equations

$$E(F_1)\colon F_i x := \text{if } P_i x \text{ then } \alpha_i x \text{ else } \beta_i x \qquad (1 \leqslant i \leqslant n),$$

where for each i, P_i is a predicate in $\mathscr{P}$ and α_i, β_i are terms in $(\mathscr{F} \cup \{F_1, \ldots, F_n\})^*$. In Section 1, a context-free grammar was associated with E in order to define the notion of a computation of E. Unfortunately, the language generated by this grammar is in general larger than the value language of E since not all derivations in the grammar correspond to computations of E. We remedy this defect by constructing a slightly more complicated grammar G as follows: Let H be the set of all functions from $\mathscr{P}$ into $\{0, 1\}$. For each i and each h in H, define a term $\gamma_{i,h}$ in $(\mathscr{F} \cup \{F_1, \ldots, F_n\})^*$ as follows: let

$$\gamma_{i,h,1} = \begin{cases} \alpha_i & \text{if } h(P_i) = 1 \\ \beta_i & \text{if } h(P_i) = 0, \end{cases}$$

$$\gamma_{i,h,j+1} = \begin{cases} \gamma_{i,h,j} & \text{if } \gamma_{i,h,j} \neq \gamma F_k \text{ for any } \gamma, k \\ \gamma\alpha_k & \text{if } \gamma_{i,h,j} = \gamma F_k \text{ and } h(P_k) = 1 \\ \gamma\beta_k & \text{if } \gamma_{i,h,j} = \gamma F_k \text{ and } h(P_k) = 0, \end{cases}$$

and let $\gamma_{i,h}$ be $\gamma_{i,h,n}$. Note that if $\gamma_{i,h}$ equals γF_k for some γ and k, then for any free interpretation I, if $P_I(x) = h(P)$ for all $P \in \mathscr{P}$, then $\mathrm{val}_I(E(F_i))$ is undefined since E goes into an infinite loop. Now let G be the context-free grammar with terminal symbols in $\mathscr{F}$, nonterminal symbols $F_1, \ldots, F_n$, start symbol F_1, and productions $F_i \to \gamma_{i,h}$ for all i and h such that $\gamma_{i,h} \neq \gamma F_k$ for any γ and k. Then for any string a in $\mathscr{F}^*$, $F_1 \Rightarrow_G^* a$ if and only if there is a free interpretation I such that $ax = \mathrm{val}_I(E)$ (cf. Section 1). Hence $L(E)$ is the context-free language generated by G.

Conversely, let G be a context-free grammar with terminal symbols in $\mathscr{F}$, nonterminal symbols $F_1, \ldots, F_n$, initial nonterminal F_1, and productions $F_i \to \alpha_{ij}$, where $i = 1, \ldots, n$ and $j = 1, \ldots, p(i)$ for some function p. Let E be the recursion scheme with function letters in $\mathscr{F}$ and predicate letters P_{ij} for $1 \leqslant i \leqslant n$ and $1 \leqslant j < p(i)$ given by the equations

$$F_j^i x := \text{if } P_{ij}x \text{ then } \alpha_{ij}x \text{ else } F_{j+1}^i x \qquad [1 \leqslant j < p(i), 1 \leqslant i \leqslant n]$$

$$F_{p(i)}^i x := \alpha_{ip(i)}x,$$

where F_1^i is identified with F_i. For any string a derivable by G, consider a rightmost derivation

$$F_1 \underset{G}{\Rightarrow} \beta_1 F_{f(1)} a_1 \underset{G}{\Rightarrow} \beta_2 F_{f(2)} a_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a$$

of a, and define a free interpretation I by setting $(P_{ij})_I(a_k x) = 1$ if and only if the production $F_i \to \alpha_{ij}$ was applied in the derivation $\beta_k F_{f(k)} a_k \Rightarrow_G \beta_{k+1} F_{f(k+1)} a_{k+1}$ (where $\beta_0 F_{f(0)} a_0 = F_1$). Such a definition is possible provided that $F_{f(k)} a_k \neq F_{f(k')} a_{k'}$ for any $k \neq k'$, which is true if $F_i \not\Rightarrow_G^* \beta F_i$ for any β and i; this can always be arranged, for example, by putting G in Greibach normal form. By definition, $\mathrm{val}_I(E) = ax$ so $a \in L(E)$. On the other hand, if I is any free interpretation, $\mathrm{val}_I(E)$ is obviously derivable by G. Hence G generates the value language of E.

Finally, we note that in the above two constructions, if either E or G is linear or right linear to begin with, then the constructed G or E is linear or right linear also. Hence (c) and (d) follow.

As indicated before, a direct consequence of Theorem 2.5 and Corollary 2.4 is the following nontranslatability result.

(2.6) COROLLARY. (a) (de Bakker–Scott). *There is a recursion scheme which is not strongly equivalent to any single-variable program scheme.*

(b) *There is a program scheme which is not strongly equivalent to any recursion scheme.*

(c) *There is a recursion scheme which is not strongly equivalent to any linear recursion scheme.*

(d) *There is a linear recursion scheme which is not strongly equivalent to any right-linear recursion scheme.*

Proof. The linear recursion scheme E in Example 2.1 is not strongly equivalent to any single-variable program scheme nor to any right-linear recursion scheme since $L(E)$ is not regular. The program scheme S in Example 2.2 is not strongly equivalent to any recursion scheme since $L(S)$ is not context free. Finally, there exist recursion schemes whose value languages are not linear context free (cf. Example 4.2).

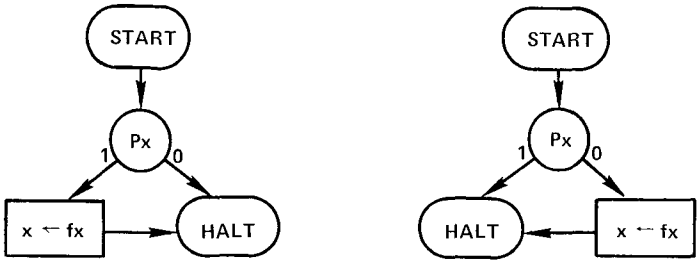
Conclusion (a) will be strengthened in Section 4 by another technique. The example used for (b) also shows that allowing extra variables in recursion schemes increases the power of such schemes; for example, the simple "binary" recursion scheme

$$F_1x := F_2(x, x)$$
$$F_2(x, y) := \text{if } Px \text{ then } fF_2(fx, y) \text{ else } F_3y$$
$$F_3x := \text{if } Py \text{ then } gF_3fy \text{ else } y$$

has value language $\{f^ng^nf^n: n \geqslant 0\}$ which is not context free, so that the scheme is not strongly equivalent to any monadic recursion scheme.

In addition to these consequences concerning translatability, the technique of value languages also has consequences concerning decidability of the existence of halting interpretations for schemes in a given class. Since there is a halting interpretation for a scheme S if and only if $L(S)$ is nonempty, there are effective procedures for deciding the existence of a halting interpretation for single variable program schemes (cf. Ianov [3]) or recursion schemes, but not for arbitrary program schemes (cf. Luckham, Park, and Paterson [4]) since the emptiness problem is solvable for context-free languages but not for recursively enumerable languages (cf. Hopcroft and Ullman [2, p. 230]) and since the correspondence between schemes and languages given by Theorem 2.5 is effective.

While Corollary 2.4 gives a necessary condition for the translatability of schemes, it does not give a sufficient condition since, as will be shown in Section 4, there is a recursion scheme which is not strongly equivalent to any program scheme even though its value language is certainly recursively enumerable since it is context free. This insufficiency is due to the failure of the converse of Theorem 2.3; schemes with the same value languages are not necessarily strongly equivalent as the following example shows:



The above two schemes both have the value language $\{\lambda, f\}$, though they are not strongly equivalent since they arrive at their values in opposite ways.

For a certain class of schemes—namely, single variable schemes with no constant functions—a converse to Theorem 2.3 can be obtained by considering a modification of the technique of value languages which takes account of the course as well as the value of a computation.

Let S be a scheme with predicate letters $P_1, \ldots, P_k$ and function letters in $\mathscr{F}$. For any free interpretation I under which S halts, the *interpreted value* $\mathrm{val}_I^{\#}(S)$ of S under I is the unique string $f_n p_{n-1} \cdots f_1 p_0 x$ in $(\mathscr{F} \cup \{0, 1\})^*$ such that $\mathrm{val}_I(S) = f_n \cdots f_1 x$ and for any $i < n$, p_i is a string $p_{i1} \cdots p_{ik}$ of k zeroes and ones such that for any j, $p_{ij} = (P_j)_I(f_i \cdots f_1 x)$. The *interpreted value language* $L^{\#}(S)$ is the set of all interpreted values $\mathrm{val}_I^{\#}(S)$ of S under free interpretations I. An interpreted value $f_n p_{n-1} \cdots f_1 p_0 x$ is *compatible* with a free interpretation I if and only if for any $i < n$ and $1 \leqslant j \leqslant k$, $p_{ij} = (P_j)_I(f_i \cdots f_1 x)$.

The important property of a single-variable scheme S with no constant functions is that for any free interpretation I there is at most one interpreted value string a in $L^{\#}(S)$ compatible with I, and if such a string exists, then it must equal $\mathrm{val}_I^{\#}(S)$. The reason for this is that for any free interpretation I and any $a \in L^{\#}(S)$ compatible with I, the only tests made by S in its computation under I are on substrings of a, and hence the computation of S under I must be the same as that of S under I', where $a = \mathrm{val}_{I'}(S)$. Using this property, we derive the following partial converse to Theorem 2.3.

(2.7) THEOREM. *For any single-variable schemes S and T with no constant functions, $S \equiv T$ if and only if $L^{\#}(S) = L^{\#}(T)$.*

Proof. Suppose $S \equiv T$ and I is a free interpretation. Then $\mathrm{val}_I(S) \simeq \mathrm{val}_I(T)$ and hence $\mathrm{val}_I^{\#}(S) \simeq \mathrm{val}_I^{\#}(T)$ by the definition of $\mathrm{val}_I^{\#}$. Consequently, $L^{\#}(S) = L^{\#}(T)$. Conversely, suppose that $L^{\#}(S) = L^{\#}(T)$, and let I be a free interpretation. It suffices to show that if $\mathrm{val}_I^{\#}(S)$ is defined, then so is $\mathrm{val}_I^{\#}(T)$ and $\mathrm{val}_I^{\#}(S) = \mathrm{val}_I^{\#}(T)$. Since $L^{\#}(S) = L^{\#}(T)$, $\mathrm{val}_I^{\#}(S)$ is in $L^{\#}(T)$. Since it is obviously compatible with I, it must therefore equal $\mathrm{val}_I^{\#}(T)$ by the remarks preceding the theorem. Hence, the theorem follows.

Theorem 2.7 has several applications. Letting $L^{\#}(\mathscr{S}) = \{L^{\#}(S): S \in \mathscr{S}\}$ for any class $\mathscr{S}$ of schemes, we obtain the following analogs of results 2.4 and 2.5.

(2.8) COROLLARY. *For any classes $\mathscr{S}$ and $\mathscr{T}$ of single-variable schemes with no constant functions:*

(a) *$\mathscr{S} \to \mathscr{T}$ if and only if $L^{\#}(\mathscr{S}) \subseteq L^{\#}(\mathscr{T})$; and*

(b) *$\mathscr{S} \rightleftarrows \mathscr{T}$ if and only if $L^{\#}(\mathscr{S}) = L^{\#}(\mathscr{T})$.*

(2.9) THEOREM. *Let $\mathscr{S}$ be the class of all* (a) *single-variable program or right-linear recursion schemes,* (b) *linear recursion schemes, or* (c) *recursion schemes which have no constant functions; then $L^*(\mathscr{S})$ contains only* (a) *regular,* (b) *deterministic linear context-free, or* (c) *deterministic context-free languages.*

Proof. The reader may check that the inclusion of the testing behavior of a scheme in its interpreted value removes the nondeterminism from the grammars defined in the proof of Theorem 2.3. For example, the interpreted value language of a recursion scheme

$$E\colon F_i x := \text{if } Px \text{ then } \alpha_i x \text{ else } \beta_i x$$

with one predicate is accepted by the deterministic pushdown automaton diagrammed in Fig. 3, where $\rightarrow^{aA/\alpha}$ means that when a is the next input symbol and A is the

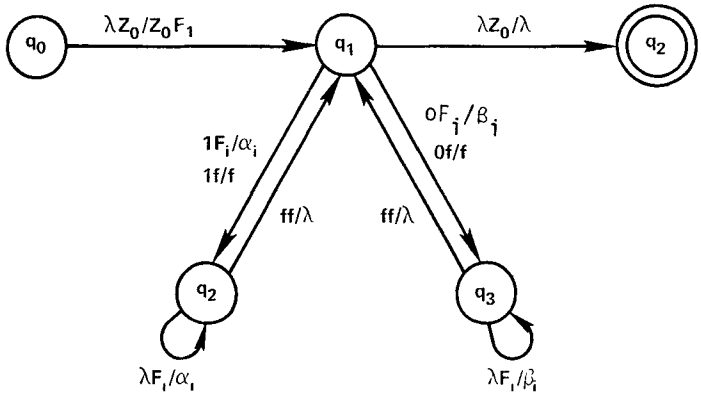


FIGURE 3

symbol on top of the pushdown store, the indicated state transition occurs with A being replaced by α on the stack (if $a = \lambda$, the input tape is not moved).

Corollary 2.8 is not particularly useful unless one strengthens Theorem 2.9 to characterize completely the languages $L^*(\mathscr{S})$ mentioned there. However, Theorem 2.9 without any strengthening at all allows us to reduce some decision problems regarding schemes to decision problems regarding languages.

(2.10) THEOREM. (a) (Ianov). *There is an effective procedure for determining whether two single-variable program schemes are strongly equivalent.*

(b) *The equivalence problem for (linear) recursion schemes is reducible to the equivalence problem for deterministic (linear) context-free languages.*

Proof. Since there is an effective procedure for deciding whether two regular

languages are equal (cf. Hopcroft and Ullman [2]), part (a) follows from 2.8 and 2.9. Part (b) also follows from 2.8 and 2.9.

It would be interesting to know if the converse of part (b) were also true, since then some unresolved decision problems in formal languages (e.g., the equivalence problem for deterministic linear context-free languages) could be reduced to known decision problems for schemes (cf. Section 5).

3. Translations of Schemes

In this section we study the problem of translating recursion schemes into program schemes. In particular, we are able to show (Theorem 3.5) that any quasirational recursion scheme is translatable into a strongly equivalent program scheme. Under somewhat less stringent conditions, which we call "weak translatability," any recursion scheme is weakly translatable into a program scheme; that is, for any recursion scheme R there is a program scheme S having the same value language as R, and such that whenever S halts, R also halts with the same output [i.e., $L(R) = L(S)$ and R is an extension of S]. These translation results depend on a few simple programming techniques. We use the same techniques to show that any recursively enumerable language is the value language of a monadic program scheme. Finally, we show that similar techniques can be used to prove that any one-variable monadic recursion scheme is translatable into a program scheme with two counters, a result *not* true for binary recursion schemes (cf. Paterson and Hewitt [6]).

Formal proofs of these results obscure the simple ideas involved, so the proofs given here tend to be in the spirit of "proofs by construction" accompanied by informal arguments showing that the constructions "work."

Essentially, the following example contains the programming techniques to be used.

Let E be the palindrome recursion scheme of Example 2.1. We show that E is strongly equivalent to the program scheme S of Fig. 4. The program variable u "holds" the value being computed. The block A of S simulates the computation of E under an interpretation I until u is assigned a value a such that $P_I(a) = 1$. Suppose that $a = (f_n \cdots f_1 x)_I$. Then $P_I((f_i \cdots f_1 x)_I) = 0$ if $1 \leqslant i < n$, and

$$f_i = \begin{cases} f & \text{if } Q_I((f_{i-1} \cdots f_1 x)_I) = 1 \\ g & \text{otherwise.} \end{cases}$$

Since $\mathrm{val}_I(E) = (f_1 \cdots f_n f_n \cdots f_1 x)_I$, blocks B and C of S must apply $f_n, \ldots, f_1$ in that order to the value in u. To see how this is accomplished, consider the situation when u has been assigned a value $(f_{n-i+1} \cdots f_n f_n \cdots f_1 x)_I$ for some $i < n$ and f_{n-i} must be applied to u next. This is done in block B, which "expects" the value of v on entry to be $(f_i \cdots f_1 x)_I$ so that after $n - i$ cycles through the loop in B, v has value a

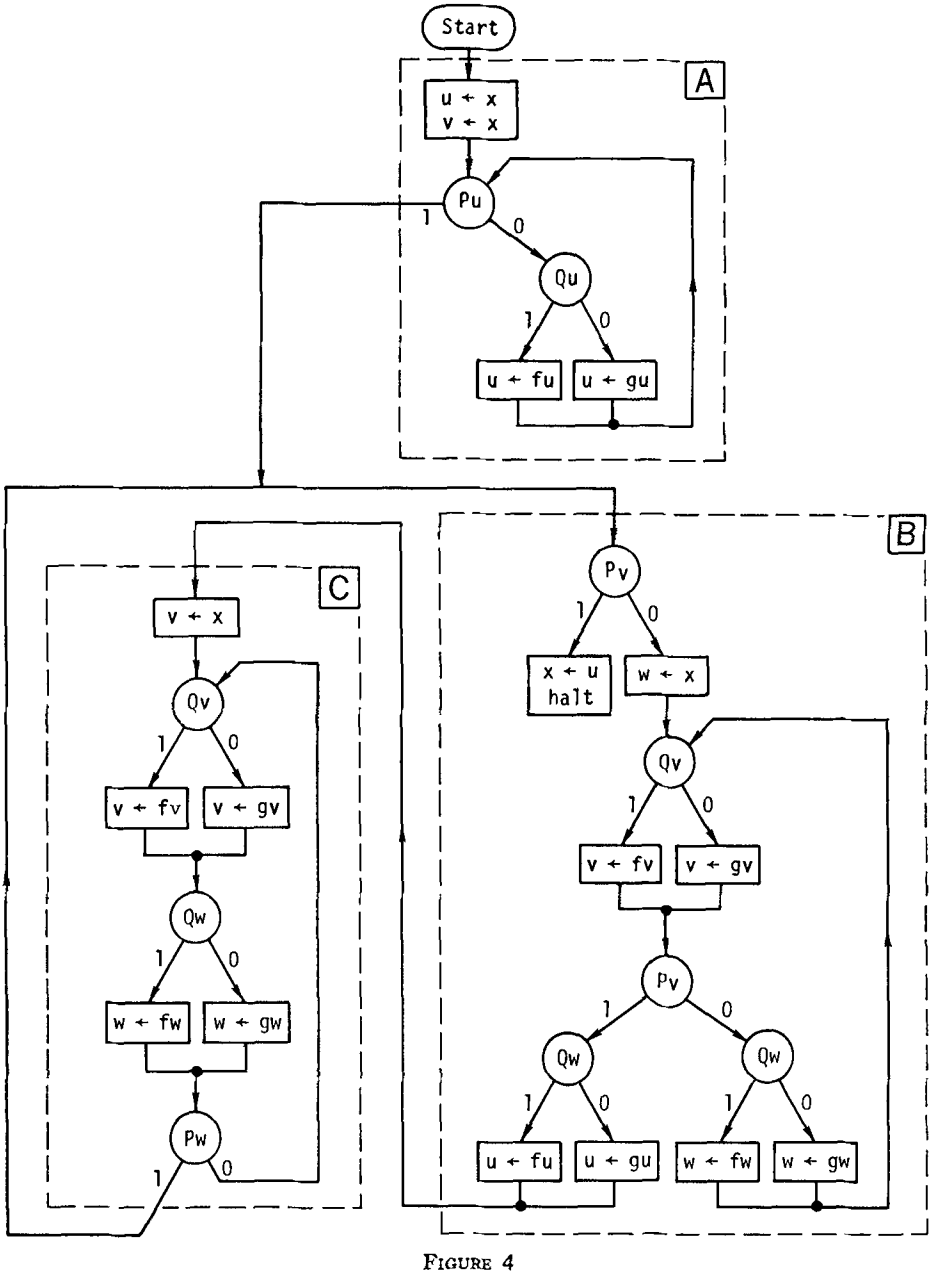

Start
A
u ← x
v ← x
Pu
1
0
Qu
1
0
u ← fu
u ← gu
B
Pv
1
0
x ← u
halt
w ← x
Qv
1
0
v ← fv
v ← gv
Pv
1
0
Qw
Qw
1
0
1
0
u ← fu
u ← gu
w ← fw
w ← gw
C
v ← x
Qv
1
0
v ← fv
v ← gv
Qw
1
0
w ← fw
w ← gw
Pw
1
0

FIGURE 4

and the test Pv exits from the loop. The value of w is then $(f_{n-i-1} \cdots f_1 x)_I$ [note that the exit occurs before the $(n - i)$-th assignment to w]. Hence, f_{n-i} is applied to u as desired. Block C now uses w to reset v to the next value expected by block B. It takes $i + 1$ cycles through the loop in C before w has value a, so that on exit from C, v has value $(f_{i+1} \cdots f_1 x)_I$, as desired. Finally, when B is entered with a as the value of v, the computation halts with u and x having the value $(f_1 \cdots f_n f_n \cdots f_1 x)_I$.

The translation of a sizable subclass of the class of recursion schemes into strongly equivalent program schemes can be achieved by constructions very similar to the one above for the palindrome scheme. The idea, as above, is to use extra program variables to "count" how many steps in the computation of a recursion scheme have already been fully simulated.

We consider first the case of linear recursion schemes, representing such a scheme E with n equations as follows:

$$E(F_1): F_i x := \text{if } P_i x \text{ then } \alpha_i F_{l(i)} \beta_i x \text{ else } \gamma_i F_{r(i)} \delta_i x, \qquad 1 \leqslant i \leqslant n,$$

where P_i is a predicate letter and $\alpha_i, \beta_i, \gamma_i, \delta_i$ are strings of basis function letters. The indexing functions, l (left) and r (right) map $\{1, 2, \ldots, n\}$ into $\{0, 1, 2, \ldots, n\}$ with the convention that F_0 is the empty string, and if $l(i) = 0$ then α_i is empty, while if $r(i) = 0$ then γ_i is empty. A computation of E will terminate whenever it reaches a term "containing" F_0, so we call these terms the *end points* of E.

The computations of linear schemes are easily described. Suppose an end point is reached after m steps in the computation of E under I. Let f be an indexing function that tells us which equation was executed at the i-th step for $1 \leqslant i \leqslant m$. Clearly, $f(1) = 1$, and f can be defined inductively in terms of l and r. Then there are strings, $a = a_m \cdots a_1$ and $b = b_1 \cdots b_{m-1}$ such that $\mathrm{val}_I(E) = (bax)_I$, where for any i,

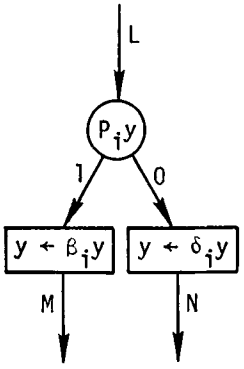
$$a_i = \begin{cases} \beta_{f(i)} & \text{if } (P_{f(i)})_I (a_{i-1} \cdots a_1 x)_I = 1 \\ \delta_{f(i)} & \text{otherwise,} \end{cases}$$

and

$$b_i = \begin{cases} \alpha_{f(i)} & \text{if } (P_{f(i)})_I (a_{i-1} \cdots a_1 x)_I = 1, \\ \gamma_{f(i)} & \text{otherwise.} \end{cases}$$

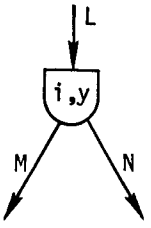
Any linear recursion scheme E can be translated into a strongly equivalent program scheme $S(E, x)$. S has program variables u, v, w in addition to the input–output variable x, and is composed of three blocks. Both the variables and the blocks serve exactly the same function as in the palindrome construction. The blocks are connected

sets of elementary (labelled) subschemes obtained from the equations of E; corresponding to the i-th equation, there are subschemes of the form



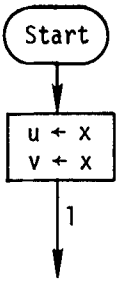
where y is any of the variables u, v, w, and L, M, N are labels; if either β_i or δ_i is the empty string, the corresponding assignment is omitted from the subscheme. The labels are merely a notational convenience to help describe how subschemes are connected together and are not part of the schemes. When a set of such schemes is connected, each exit from a scheme points to a scheme that has the same label at its entry.

Let us represent the above subschemes corresponding to the i-th equation by

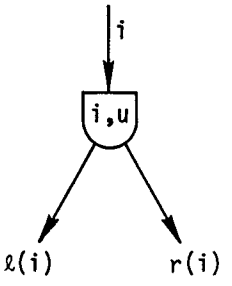


The first block of S (block A) is constructed by connecting together the following set of schemes (the connections are determined uniquely by the labels):

(a)



(b) for $1 \leqslant i \leqslant n$



The exit from block A has the label 0. The block is shown in Fig. 5; the loop indicates that connections (for labels $i > 0$) are made within the block.

It should be clear that a computation under an interpretation I as above reaches the exit labeled $i = 0$ exactly when u has value $(ax)_I$. Blocks B and C must now

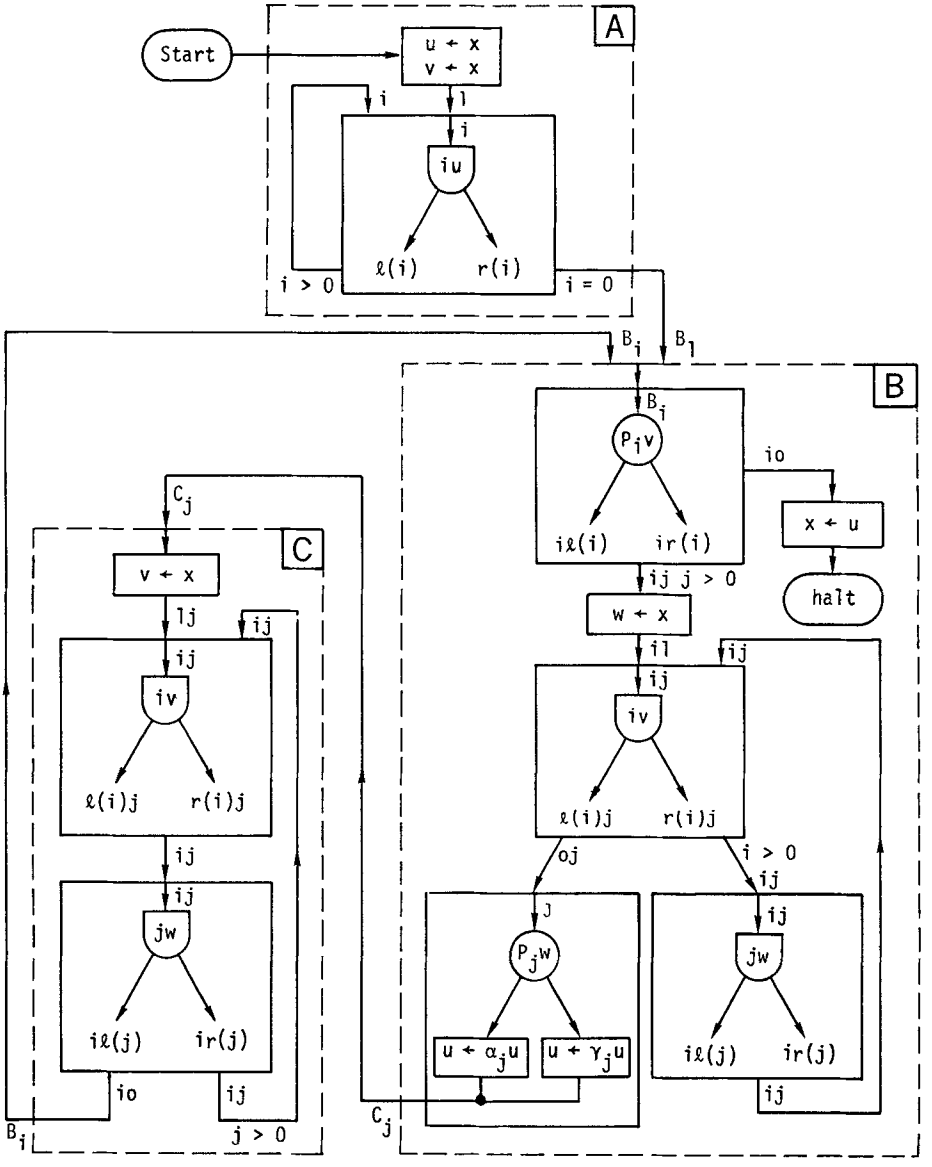


FIGURE 5

set the value of u to $(bax)_I$. This is accomplished as for the palindrome scheme, though now blocks B and C are more complicated since they must keep track not only of the state of the computation using v and w, but also the next equations to be applied to v and w; the doubly indexed labels in blocks B and C take care of this.

Within block B (see Fig. 5) an exit from a subscheme computing on v is connected to a subscheme computing on w, and vice versa, with the exception that exits labeled "oj" from subschemes computing on v break the "loop" in B and go to subschemes computing on u. B has n entry points, $B_1, \ldots, B_n$. If the i-th equation of E has an end point, then entry B_i performs the test $P_i v$ and the corresponding exit leads to halt.

Suppose that u has been assigned the value $(b_{m-k+1} \cdots b_{m-1} a_m a_{m-1} \cdots a_1 x)_I$, so that b_{m-k} must be applied to u next. Block B "expects" to be entered at the label B_i, where i is the next equation to apply to the expected value $(a_k \cdots a_1 x)_I$ of v. As before, after $m - k$ cycles through the "loop" in B, v has value $(ax)_I$ and w has value $(a_{m-k-1} \cdots a_1 x)_I$, so that b_{m-k} is applied to u by equation j. Block C (Fig. 5) now uses w to reset v to the next value expected by block B. It takes $k + 1$ cycles through the "loop" in C before w has value $(ax)_I$, so that on exit from C, v has value $(a_{k+1} \cdots a_1 x)_I$, as desired. Finally, when B is entered with $(a_m \cdots a_1 x)_I$ as the value of v, the computation halts after assigning x the value $(bax)_I$.

It should now be clear that E is strongly equivalent to S. Thus we have the following theorem.

(3.1) THEOREM. *Any linear recursion scheme is translatable into a strongly equivalent program scheme. There is an algorithm for doing the translation.*

Observe that if E is a set of right linear equations, so that for each i, α_i and γ_i are empty, then blocks B and C of S and variables u, v, w may be omitted. Thus the right-linear recursion schemes are translatable into strongly equivalent single-variable program schemes. The converse follows from the standard method of translating program schemes into multivariable recursion schemes.

The above translation has a further useful property.

(3.2) *Remark.* For any I, a value is computed during the computation of E under I if and only if it is computed during the computation of $S(E, x)$ under I.

We would like to be able to characterize the class of all monadic recursion schemes that can be translated into program schemes.[2] Certainly, one would suspect that Theorem 3.1 holds for simple compositions of linear schemes.

(3.3) DEFINITION (functional substitution). Let E_1 and E_2 be recursion schemes having no defined function symbols in common. Suppose that the initial defined

[2] For binary recursion schemes, the class is not recursively enumerable.

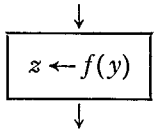
functions are F_1 and F_2, respectively. Let E_1' be obtained by replacing all occurrences of a basis function f in E_1 by F_2. Then the recursion scheme with equations $E_1' \cup E_2$ and initial function symbol F_1 is obtained by *functional substitution* of E_2 for f in E_1.

Notation. $E_1(f; E_2)$ denotes the scheme obtained by functional substitution of E_2 for f in E_1.

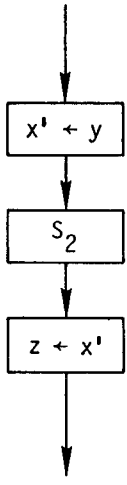
(3.4) THEOREM. *The class of those recursion schemes translatable into program schemes in a manner satisfying Remark* 3.2 *is closed under functional substitution.*

Proof. Suppose that E_1 and E_2 are translatable into program schemes S_1 and S_2, respectively. We may assume that the two program schemes have no common variables, the basis function f occurs in E_1 but not E_2, and (3.2) is true for both pairs of schemes.

Let $E' = E_1(f; E_2)$. We construct a new program scheme, S', by replacing all computations involving f in S_1 by copies of S_2. For example, the assignment,



is replaced by



where x' is the input variable of S_2 and the halt instructions in S_2 are replaced by "go to exit." The identity assignments, which ensure that S_2 receives the correct input and assigns its output correctly, can be eliminated; that is, there is an equivalent

program scheme satisfying (3.2) which does not contain any assignments between pairs of program variables (Theorem 3.7).

For any interpretation I of E' and S', let I' be an extension of I such that

$$f_{I'}(v) = \begin{cases} \text{val}_I(E_2)(v) & \text{if the computation of } E_2 \text{ under } I \text{ with input} \\ & \text{value } v \text{ halts,} \\ \Omega & \text{otherwise,} \end{cases}$$

where $\Omega \notin \text{dom}_I$ and $\text{dom}_{I'} = \text{dom}_I \cup \{\Omega\}$.

First, consider the relationship between the computations of S' and E' under I and the computations of S_1 and E_1 under I'. The two pairs of computations are identical except at points where a value of $f_{I'}$ is computed in S_1 or E_1. If the values of $f_{I'}$ are not Ω, these points correspond to complete finite computations of S_2 or E_2 yielding the same values. Such subcomputations do not disturb the values of variables of S_1 since the schemes have separate variables. If a value of $f_{I'}$ is Ω, that point corresponds to an infinite computation of S_2 or E_2 so that the computations of S' and E' under I diverge.

Now, our assumption of (3.2) implies that under I', S_1 and E_1 compute the same values of $f_{I'}$. If all of these subcomputations halt, then $\text{val}_I(S') \simeq \text{val}_{I'}(S_1) \simeq \text{val}_{I'}(E_1) \simeq \text{val}_I(E')$. If some subcomputation does not halt, then both S' and E' diverge. Thus, $E' \equiv S'$. Furthermore, it is easily seen that (3.2) holds between these schemes as well.

The translation algorithm for linear schemes can now be extended by the replacement algorithm given in the proof of (3.4). Whenever we know how to build a given scheme from some finite set of linear schemes by functional substitutions, we can construct a program scheme equivalent to it.

Theorem 3.4 implies that certain natural classes of recursion schemes are translatable into program schemes. Such a class is the *metalinear* schemes: if $E_1(F_1),\ldots, E_n(F_n)$ are linear schemes with disjoint sets of defined functions, then the scheme $E(F)$ with equations, $\{Fx := F_1F_2 \cdots F_nx\} \cup \bigcup_{i=1}^{n} E_i$, is metalinear. A more comprehensive class is the *quasirational* schemes (i.e., those with associated quasirational grammars, Nivat [5]), which is precisely the smallest class containing the linear schemes and closed under functional substitution.

(3.5) THEOREM. *The quasirational recursion schemes are translatable into strongly equivalent program schemes.*

Are some nonquasirational recursion schemes translatable into program schemes? In Section 4 we show that not all are, but that there is a nonquasirational scheme translatable into a program scheme with a single counter. This leads us to ask the following question.

(3.6) QUESTION. *Is the class of those monadic recursion schemes that are translatable into program schemes with one counter so that* (3.2) *is satisfied closed under functional substitution?*

We conjecture that the answer is *no*. If correct, this would immediately imply that the class of program schemes with one counter is more powerful than the class of program schemes, which we believe to be true that are unable to show.[3]

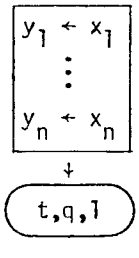
(3.7) THEOREM. *For any program scheme S with identity assignments (i.e., instructions $x \leftarrow y$) there is a program scheme S' whose only identity assignments have unassigned input variables on the right-hand side such that $S \equiv S'$ and property* (3.2) *is satisfied.*

Proof. Suppose S has variables $x_1, \ldots, x_n$ and functions $f_1, \ldots, f_m$. We introduce new variables $y_1, \ldots, y_n$ and $z_1, \ldots, z_n$ to be used in S' as follows: the computation of S on the x's will be carried out by S' on the y's so that the x's can be used as unassigned input variables, while the z's will provide "backup" to the y's in order to eliminate identity assignments of noninput values.

Specifically, at any time in the computation of S, with each variable x_i is associated either an input value or a function f_k and a variable x_j such that the present value of x_i is f_k of some previous value of x_j. There are only a finite number of such associations, so they can be coded into S'. If the correct previous value of x_j is kept in z_l say, an identity assignment $x_h \leftarrow x_i$ in S can be replaced in S' by $y_h \leftarrow f_k z_l$.

Let T be the collection of all mappings t: $\{1,\ldots, n\} \to \{0,\ldots, m\}$ and Q be the collection of all mappings q: $\{1,\ldots, n\} \to \{1,\ldots, n\}$. We index instructions in S' by $T \times Q \times I$, where I is the index set of the instructions of S.

S' starts with



where $t(i) = 0$, $q(i) = i$ for all i.

The remainder of S' is obtained by connecting the instructions indicated in Fig. 6.

Simple programming also enables us to characterize the class of value languages of program schemes. Although there has been some speculation as to the extent of this class, it turns out to be exactly all the recursively enumerable languages. This

[3] The answer to (3.6) is *yes* (see footnote 1).

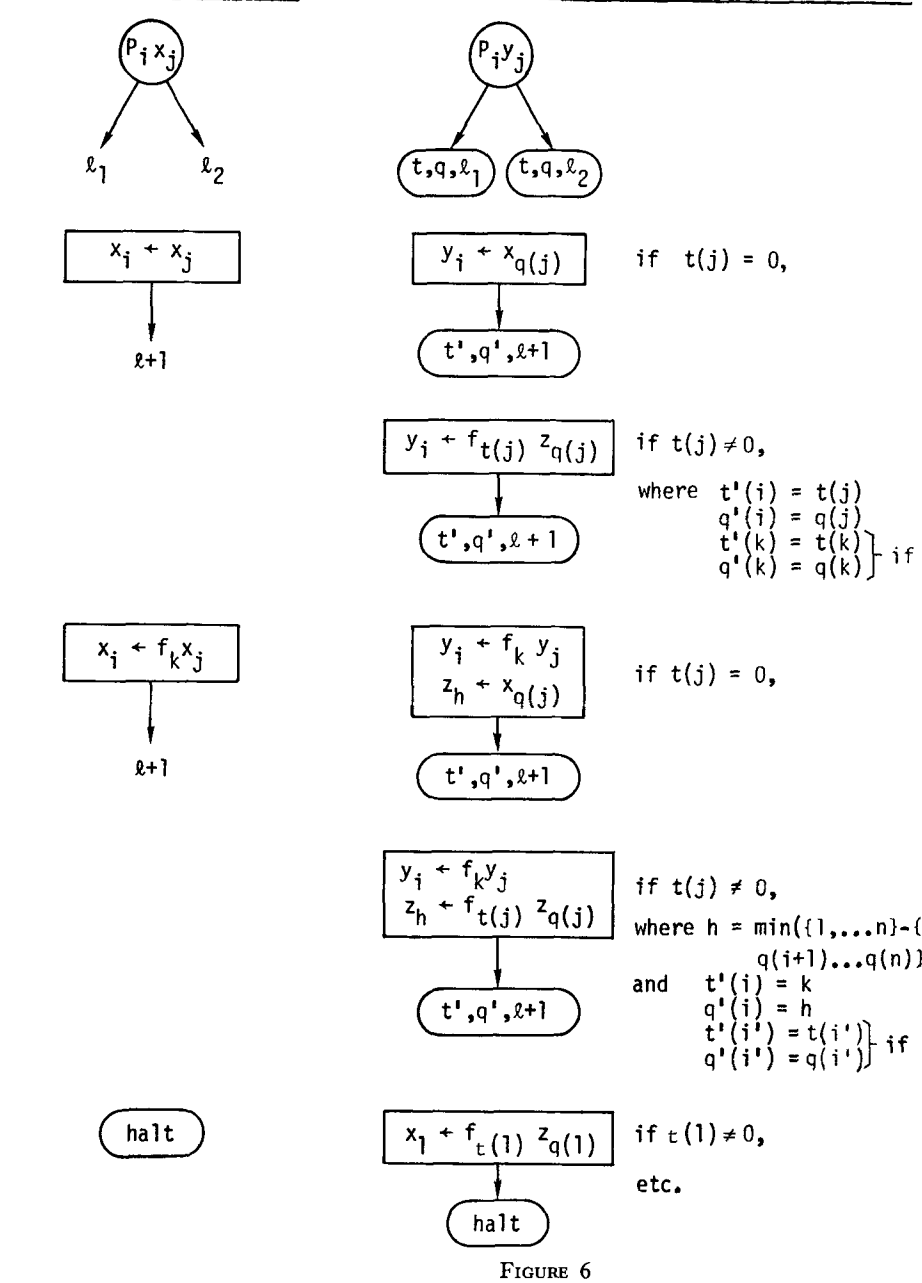

if instruction ℓ in S is
instruction < t,q,ℓ > in S' is
$P_i x_j$
ℓ_1
ℓ_2
$P_i y_j$
t,q,ℓ_1
t,q,ℓ_2
$x_i \leftarrow x_j$
$\ell+1$
$y_i \leftarrow x_{q(j)}$
t',q',ℓ+1
if t(j) = 0,
$y_i \leftarrow f_{t(j)}\ z_{q(j)}$
t',q',ℓ + 1
if t(j) ≠ 0,
where t'(i) = t(j)
q'(i) = q(j)
t'(k) = t(k)
q'(k) = q(k)
if k ≠ i.
$x_i \leftarrow f_k x_j$
$\ell+1$
$y_i \leftarrow f_k\ y_j$
$z_h \leftarrow x_{q(j)}$
t',q',ℓ+1
if t(j) = 0,
$y_i \leftarrow f_k y_j$
$z_h \leftarrow f_{t(j)}\ z_{q(j)}$
t',q',ℓ+1
if t(j) ≠ 0,
where h = min({1,...n}-{q(1)...q(i-1)
q(i+1)...q(n)})
and t'(i) = k
q'(i) = h
t'(i') = t(i')
q'(i') = q(i')
if i' ≠ i .
halt
$x_1 \leftarrow f_{t(1)}\ z_{q(1)}$
halt
if t(1) ≠ 0,
etc.

FIGURE 6

can be regarded as "unfortunate" since it limits the possible exploitation of value languages to prove nontranslatability results (cf. Section 2).

The fact that we can construct a program scheme having a given r.e. language as its value language is a direct consequence of being able to program universal computing machines into the behavior of program schemes. To demonstrate this, we find it easiest to use the register machines of Shepherdson and Sturgis [8].

We recall that a register machine M has a finite number of registers $x_1, \ldots, x_r$ and a finite list of instructions of the forms (i) $x_i := x_i + 1$, (ii) $x_i := x_i \dot{-} 1$ (where $x \dot{-} 1 = x - 1$ if $x > 0$ and $0 \dot{-} 1 = 0$), and (iii) transfer to instruction l if $x_i \neq 0$. The function computed by M is that function f such that for any n, $f(n)$ is the final value of x_1 if M halts given input n in x_1 and 0 in $x_2, \ldots, x_r$, while $f(n)$ is undefined if M does not halt.

(3.8) THEOREM. *The class of recursively enumerable languages is exactly the class of value languages of monadic program schemes.*

Proof. For any program scheme T, $L(T)$ is r.e. since the set of halting computations of T under free interpretations is r.e. It is therefore necessary to prove that for a set $\mathscr{F}$ of (function) symbols, if X is an r.e. subset of $\mathscr{F}^*$, then $X = L(T)$ for some T. Now, there is a register machine M such that for any string a, $a \in X$ if and only if the Godel number[4] of a is in the range of the function computed by M. We show first that the register machine M can be "simulated" by a program scheme S. If M has registers $x_1, \ldots, x_r$, S will have variables $x_1, \ldots, x_r, u, v, w$, a single function letter f, and a predicate letter P. S is built up from elementary schemes corresponding to the instructions of M.

The way in which S simulates M is based on the following idea. For any interpretation I and any element a in dom_I, we say that a "codes" an integer n if n is the least integer such that $P_I(f_I^n a) = 1$. At any point in the computation of S under I, the *content* of a register x is the integer coded by the value of that register. Note that a variable can have an arbitrary content under an interpretation I if, for any $a \in \mathrm{dom}_I$, the sequence $P_I(a), P_I(f_I a), P_I(f_I^2 a), \ldots$ contains arbitrarily long sequences of zeroes separated by ones, i.e., if

$$\forall a \in \mathrm{dom}_I \, \forall n \, \exists m [P_I(f_I^{m+n} a) = 1 \quad \text{and} \quad (\forall i < n) \, P_I(f_I^{m+i} a) = 0].$$

We call such interpretations *good interpretations*; for example, the free interpretation such that $P_I fx, P_I f_x^2, \ldots$ is the sequence 1010010001... is good. Notice that if the content of a register x with value a is n, then one can "add one" to the content of x under a good interpretation I by an assignment $x \leftarrow f^m x$, where $P_I(f_I^{m+n+1} a) = 1$ and $P_I(f_I^{m+i} a) = 0$ for all $i \leqslant n$.

[4] We assume some standard computable Godel numbering of strings of symbols.

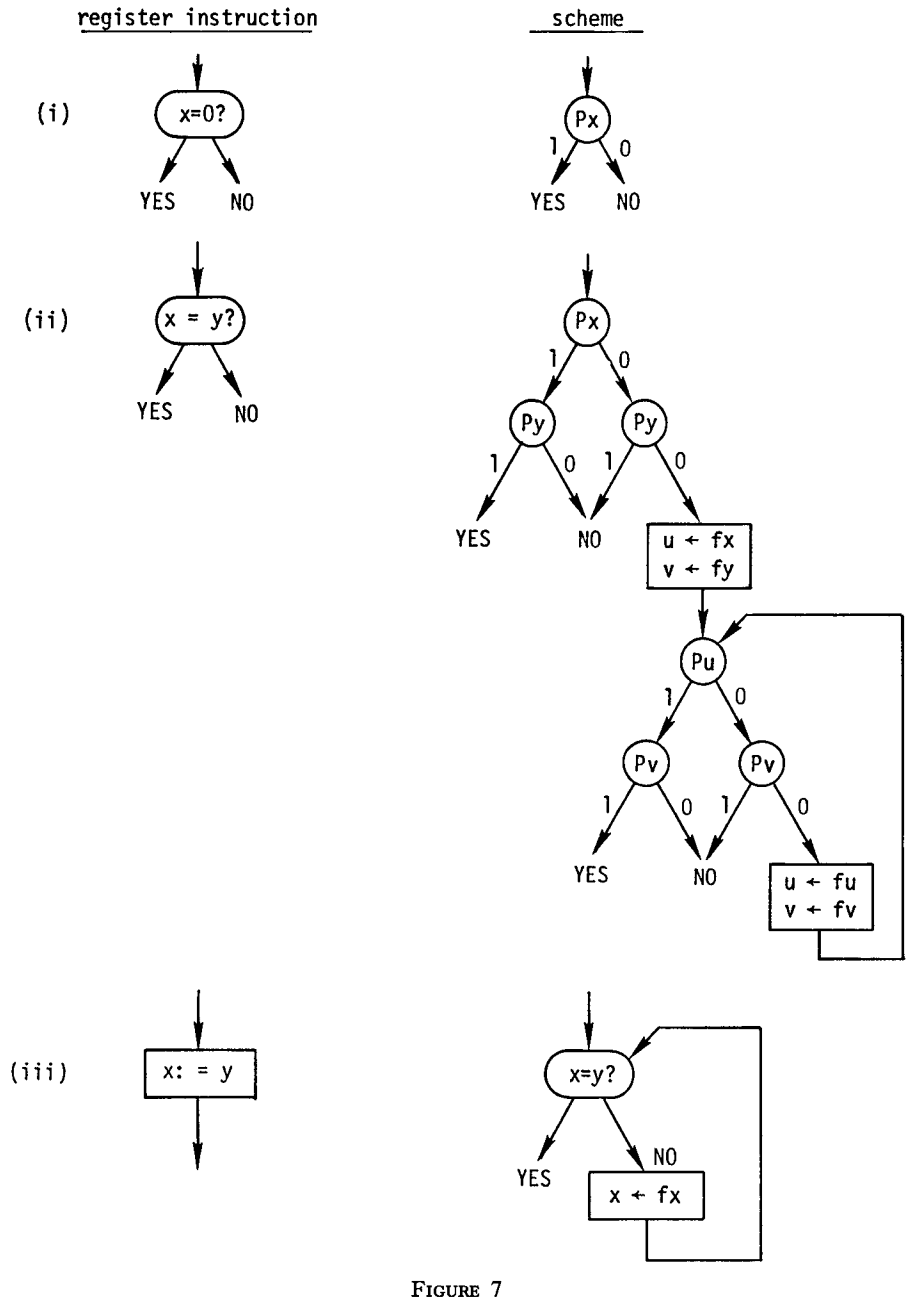

register instruction
scheme
(i)
x=0?
YES
NO
Px
1
0
YES
NO
(ii)
x = y?
YES
NO
Px
Py
Py
Py
Py
u ← fx
v ← fy
Pu
Pv
Pv
Pv
Pv
u ← fu
v ← fv
(iii)
x: = y
x=y?
YES
NO
x ← fx

FIGURE 7

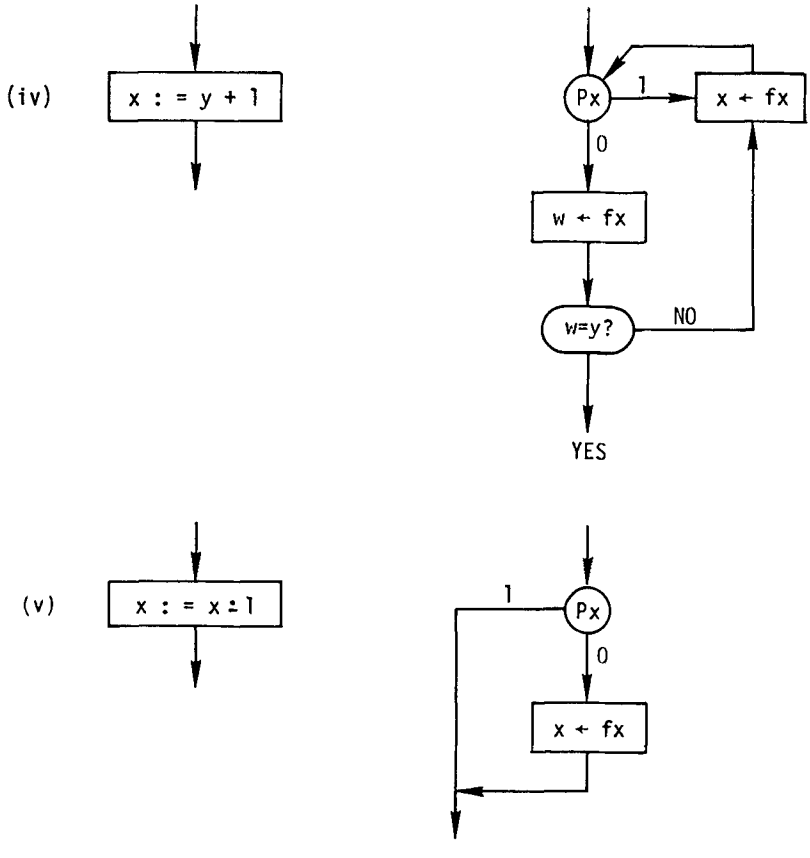


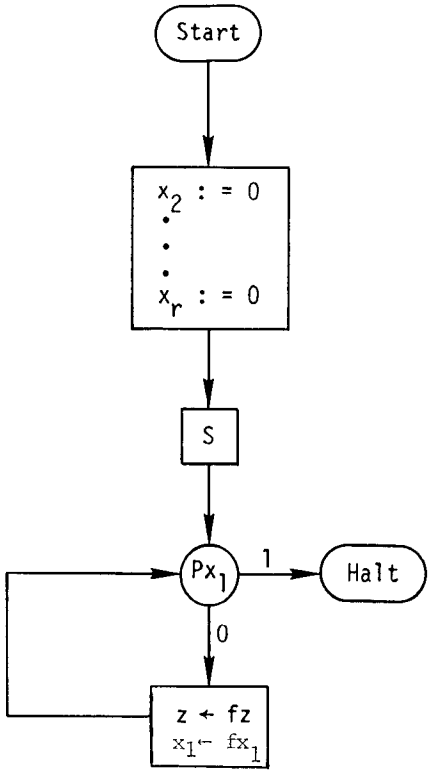
FIGURE 7 (continued)

For each of the basic instructions in M, there is an elementary scheme that under any interpretation will either perform the corresponding operation (in the above sense) on the contents of its variable or will go into a loop trying to do the required manipulation. The elementary schemes are given in Fig. 7 (x, y are distinct registers). This assumption that x and y are distinct registers is inessential since slightly more complicated schemes will simulate, e.g., $x := x + 1$.

As usual, S is constructed from M by replacing each register instruction by the corresponding scheme and connecting the schemes exactly as the instructions of M are connected. Under any interpretation and any input, S either loops or transforms the contents of its variables in the same way that M transforms the contents of its registers. Under any good interpretation, S simulates M.

Given S, we now construct a program scheme T with $L(T) = X$. To do this we must decode the output from S. This clearly can be done since the translation

from a Godel number to the string it represents can be made by a register machine. In the simplest case when $X = \{f^n : n \in \text{range}(M)\}$, $X = L(T)$ where T is the program scheme with output variable z,



If X is an r.e. language over a larger alphabet, a more complicated decoding at the end will give X as the value language of a program scheme.

(3.9) *Remark.* Theorem 3.8 can be strengthened by restricting the class of program schemes needed to obtain all r.e. languages as value languages. Any r.e. language is in fact the value language of a program scheme with resets (i.e., constant functions) in which all variables and resets are independent. That is, the scheme contains variables $x_1, \ldots, x_k$, distinct constant functions $a_1, \ldots, a_k$, and monadic function f, and all of its assignments are of the forms, $x_i := fx_i$ or $x_i := a_i$. We leave the proof to the reader.

The ability to simulate register machines by program schemes allows us to prove further translation results directly.

First, we relax slightly the stipulation of strong equivalence required for translatability.

(3.10) DEFINITION. S is a *weak translation* of R if

(i) $L(R) = L(S)$, and

(ii) for any interpretation I, if S halts under I then R halts under I with the same output.

Thus R is an extension of S, but in an inessential way; anything R does, S will also do on some interpretation.

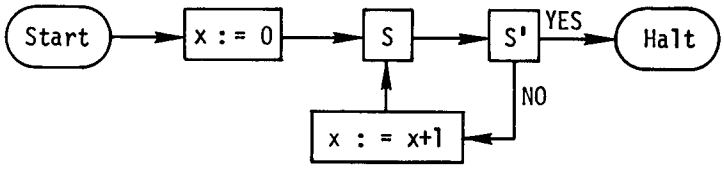
(3.11) DEFINITION. Let $\mathscr{R}$ and $\mathscr{S}$ be classes of schemes. $\mathscr{R}$ is *weakly translatable* into $\mathscr{S}$ (*notation*: $\mathscr{R} \to {}_w\mathscr{S}$) if

$$(\forall P \in \mathscr{R})(\exists Q \in \mathscr{S})[Q \text{ is a weak translation of } P].$$

(3.12) THEOREM. *The class of monadic recursion schemes is weakly translatable into the class of monadic program schemes.*

Proof. We merely sketch the proof since the details are similar to those of previous proofs.

Suppose R is a one-variable monadic recursion scheme. Then $L^{\#}(R)$ is r.e. Hence, following the proof of Theorem 3.8 there is a program scheme S such that the contents of the outputs of S are the Godel numbers of the strings, $w_0, w_1, w_2, \ldots$ of $L^{\#}(R)$. Using S, a program scheme Q can be constructed such that for any interpretation I, Q_I generates and tests each w_i for compatibility with I, and stops when one is found. In more detail, the output from S, an encoding of w_i say, is input to another scheme S'. Recall that w_i is of the form $f_m p_m \cdots f_2 p_2 f_1 p_1$, where f_j is a basis function of R and p_j is a binary sequence of length n. The computation of S' under I decodes w_i and checks that it is compatible with I. To do this, S' checks each p_j in w_i by performing successive tests, for $1 \leqslant k \leqslant n$, of the form $P_k f_j \cdots f_1 x = p_{jk}?$, where p_{jk} denotes the k-th element of p_j, and P_k is the k-th predicate in R. Note that S' can be constructed so that it always halts under a good interpretation. If all of the tests for every p_j in w_i have a "yes" answer, then $\mathrm{val}_I(Q) = (f_m \cdots f_1 x)_I$; if a "no" answer occurs, Q then returns to S to generate w_{i+1}. Q may be represented as follows:



If Q halts under I, then R also halts under I and $\mathrm{val}_I(Q) = \mathrm{val}_I(R)$ by the remark preceding Theorem 2.7. Conversely, if R halts under a good interpretation I, then Q halts under I. Since any halting computation of R is carried out under some good interpretation, we have that $L(Q) = L(R)$. Therefore Q weakly translates R.

Note that the weak translations in the preceding theorem can be converted into strong translations if one augments the class of program schemes by a predicate P and a function f with a fixed good interpretation. Alternately, one can dispense with P and f entirely and augment program schemes by some finite number of counters: since a register machine R *is* a program scheme with a finite number of counters, one can use counters to simulate R directly in the preceding constructions. How many counters are required in order to translate an arbitrary monadic recursion scheme into a program scheme? The following theorem shows that two are enough, while the results of Section 4 show that one does not suffice.

(3.13) THEOREM. *Every monadic recursion scheme is strongly equivalent to a monadic program scheme with two counters.*

Proof. As described in Section 1, the computation of a monadic recursion scheme can be visualized as operating with a stack of functions yet to be applied to the current value of the variable. The theorem follows easily from this representation and the standard coding of a stack of symbols by two counters (cf. [2, Lemma 6.2, Theorem 6.4]).

It should be noted that the theorem fails for binary recursion schemes (cf. [6]) since the stack "required" for the computation of such a scheme must hold the values being computed and not merely a list of operations to be performed; whereas the latter stack can be simulated by two counters, the former cannot.

4. LENGTHS OF COMPUTATIONS

The technique of value languages enabled us to establish nontranslatability results in Section 2 by showing that a given scheme S was not strongly equivalent to any member of a given class $\mathscr{S}$ of schemes since the schemes in $\mathscr{S}$ were not "rich" enough to compute all the possible values of S. This technique fails to establish nontranslatability when the schemes in $\mathscr{S}$ are "rich" enough to compute all the possible values of S, but not "rich" enough to compute the right values under the right interpretations. Such is the case with respect to recursion schemes and program schemes for, as we shall see below, the weak translations of recursion schemes into program schemes established in Section 3 cannot be improved to give strong-equivalence-preserving translations.

The intuition behind the nontranslatability of recursion schemes into program schemes is that the computations of recursion schemes can be much more complex than those of program schemes. This intuition does not prove anything, however, since the notion of strong equivalence concerns only the values of schemes and not how those values are obtained. Still, the intuitive notion that computations of certain recursion schemes "take too long" for those schemes to be translatable into program schemes can be formalized to show that there is a monadic recursion scheme not strongly equivalent to any program scheme, or even to any program scheme augmented by a counter.

How can the length of a computation be measured and used to establish the nontranslatability of a scheme S into any member of a class $\mathscr{S}$ of schemes? Since translatability is concerned with the values of schemes under given interpretations, the solution is to record the length of the computation of S under suitable interpretations I in $\mathrm{val}_I(S)$. Then it can be shown that for any scheme T in $\mathscr{S}$, $\mathrm{val}_I(S) \neq \mathrm{val}_I(T)$ for interpretations I under which S has computations that are "too long" for T to keep up with.

More precisely now, let $\mathscr{F}$ be a set of function letters and $\mathscr{P}$ a set of predicate letters. For any $n > 0$, an interpretation I for $\mathscr{F}$ and $\mathscr{P}$ is a *counting interpretation of rank n* if and only if for some set A of n symbols and some symbol b,

(a) $\mathrm{dom}_I = \{ab^m\colon a \in A \,\&\, m \geqslant 0\}$,

(b) for any $f \in \mathscr{F}$ there is a function f' from A into itself and a constant $k \leqslant 1$ such that for any $a \in A$ and any m, $f_I(ab^m) = f'(a)b^{m+k}$,

(c) for any $P \in \mathscr{P}$, any $a \in A$, and any m, m', $P_I(ab^m) = P_I(ab^{m'})$, and

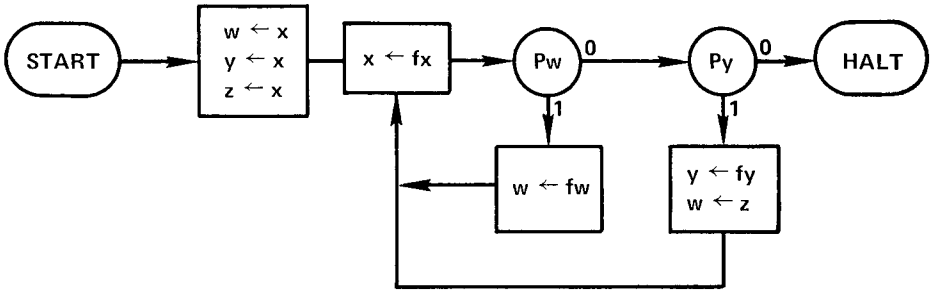
(d) for any $x \in \mathscr{V}$, $x_I = a_1$.

In other words, in a counting interpretation of rank n, the value of a function or predicate depends only upon the first symbol of the string to which it is applied, so that at most n objects in the domain can be distinguished; yet the length of a computation can be recorded in the length of the output.

For any scheme S and any counting interpretation I under which S halts, define the length $\mathrm{len}_I(S)$ of S under I to be the unique m such that $\mathrm{val}_I(S) = ab^m$ for some $a \in A$; for any $n > 0$, define $\mathrm{len}(S, n)$ to be the maximum value of $\mathrm{len}_I(S)$ over all counting interpretations I of rank n [$\mathrm{len}(S, n)$ is always defined since there are only finitely many counting interpretations of rank n]. It should be obvious that a necessary condition for two schemes S and T to be strongly equivalent is that $\mathrm{len}(S, n) = \mathrm{len}(T, n)$ for all n [if it is not obvious, note that if $\mathrm{len}(S, n) > \mathrm{len}(T, n)$, then $\mathrm{len}_I(S) = \mathrm{len}(S, n) > \mathrm{len}(T, n) \geqslant \mathrm{len}_I(T)$ for some counting interpretation I of rank n, so that $\mathrm{val}_I(S) \neq \mathrm{val}_I(T)$].

Some examples may help clarify these definitions and their intended application. The program scheme S with flow chart

(4.1)



has value language $\{f^{n^2}: n > 0\}$, and hence is not strongly equivalent to any single variable program scheme since its value language is not regular (indeed, it is not strongly equivalent to any recursion scheme, since its value language is not context free, but that does not concern us here). An alternative proof of this result can be given as follows. For any $n > 0$, let I be a counting interpretation of rank n such that

$$\begin{aligned}
\mathrm{dom}_I &= \{a_i b^m : 1 \leqslant i \leqslant n \;\&\; m \geqslant 0\},\\
f_I(a_i b^m) &= \begin{cases} a_{i+1} b^{m+1} & \text{if } \; 1 \leqslant i < n \\ a_i b^{m+1} & \text{if } \; i = n, \end{cases}\\
P_I(a_i b^m) &= 1 \qquad \text{if and only if } \; 1 \leqslant i < n, \;\text{ and}\\
x_I &= a_1 .
\end{aligned}$$

Then $\mathrm{val}_I(S) = a_n b^{n^2}$, so $\mathrm{len}(S, n) \geqslant \mathrm{len}_I(S) \geqslant n^2$. On the other hand, as we shall show below, $\mathrm{len}(T, n)$ is bounded by some constant multiple kn of n for any single-variable program scheme T; thus, for any $n > k$ and any counting interpretation I of rank n as above, $\mathrm{len}_I(T) \leqslant kn < n^2 = \mathrm{len}_I(S)$, so that S is not strongly equivalent to any single-variable program scheme.

A more interesting example is the recursion scheme

(4.2) $$E: Fx := \text{if } Px \text{ then } fx \text{ else } gFFhx.$$

For any $n > 0$, let I be a counting interpretation of rank n such that

$$\begin{aligned}
\mathrm{dom}_I &= \{a_i b^m : 1 \leqslant i \leqslant n \;\&\; m \geqslant 0\},\\
f_I(a_i b^m) &= a_i b^{m+1},\\
g_I(a_i b^m) &= \begin{cases} a_{i-1} b^m & \text{if } \; i > 1 \\ a_i b^m & \text{if } \; i = 1, \end{cases}\\
h_I(a_i b^m) &= \begin{cases} a_{i+1} b^m & \text{if } \; i < n \\ a_i b^m & \text{if } \; i = n, \end{cases}\\
P_I(a_i b^m) &= 1 \qquad \text{if and only if } \; i = n, \;\text{ and } \; x_I = a_1 .
\end{aligned}$$

Then $\mathrm{val}_I(E) = a_1 b^{2^{n-1}}$ [since $F_I(a_{n-i}b^m) = a_{n-i}b^{m+2^i}$, as can be shown by induction on i], so that $\mathrm{len}(E, n) \geqslant \mathrm{len}_I(E) \geqslant 2^{n-1}$. As will be shown later, it follows that E is not strongly equivalent to any program scheme, or even to any program scheme with a counter, since any such scheme T has $\mathrm{len}(T, n)$ bounded by some constant power n^k of n, which is less than 2^{n-1} for sufficiently large n. The same argument also applies to the scheme

$$Fx := \text{if } Px \text{ then } x \text{ else } gFFhx.$$

The bounds on $\mathrm{len}(T, n)$ for various schemes T are established by the following theorem.

(4.3) THEOREM. *For any scheme T there is a constant $k > 0$ such that for any $n > 1$,*

(a) *if T is single-variable program scheme, then* $\mathrm{len}(T, n) \leqslant kn$,

(b) *if T is a program scheme, then* $\mathrm{len}(T, n) \leqslant n^k$,

(c) *if T is a program scheme with a single counter, then* $\mathrm{len}(T, n) \leqslant n^k$, *and*

(d) *if T is a recursion scheme, then* $\mathrm{len}(T, n) \leqslant k^n$.

Proof. For parts (a) and (b), suppose that T is a program scheme with i instructions and v variables, and that I is a counting interpretation of rank n under which T halts. A *state* in the computation of T under I is a $v + 1$-tuple$\langle j, a_{n_1}, \ldots, a_{n_v}\rangle$, where j is the number of the instruction being executed and for each l, the current value of the variable x_l is $a_{n_l}b^{m_l}$ for some integer m_l. Since T halts under I, no state can be repeated in the computation of T; hence there are at most $i \cdot n^v$ steps (i.e., instructions executed) in the computation of T (that being the number of distinct states), and so $\mathrm{len}_I(T) \leqslant i \cdot n^v$ since each instruction adds at most one b to the output. Since I is any counting interpretation of rank n, $\mathrm{len}(T, n) \leqslant i \cdot n^v$. Part (a) is now immediate, as is (b) by choosing k large enough so that $n^{k-v} \geqslant i$.

For (c), let i and v be as above, and let I be a counting interpretation of rank n under which T halts. We show first that the maximum value of the counter attained during the computation of T under I is at most $i \cdot n^v$. Suppose not. Let the counter attain its maximum value $m > i \cdot n^v$ at some step s_m in the computation. Let s_0 be the last step before s_m when the counter was zero, and for each j between 0 and m, let s_j be the last step before s_m that the counter attains a value of j. Since $m > i \cdot n^v$, there must be two values j and j' of the counter such that $0 < j < j' \leqslant m$ and T is in the same state (as defined above, disregarding the value of the counter) at steps s_j and $s_{j'}$. But then T must go into an infinite loop, since after step s_j the counter never reaches zero and, for any number s of steps, at step $s_{j'} + s$, T must be in the same state as at step $s_j + s$ with the exception that the value of its counter is now $j' - j$ more than before. Thus, taking the value of the counter into account, T has at most $(i \cdot n^v)^2$ states, so that $\mathrm{len}_I(T) \leqslant (i \cdot n^v)^2$. Since I was an arbitrary counting

interpretation of rank n, $\text{len}(T, n) \leqslant i^2 n^{2v}$, and part (c) follows by taking k large enough so that $n^{k-2v} \geqslant i^2$.

For (d), let T be a recursion scheme with r recursion equations, q function letters, and terms of length at most l. Let I be a counting interpretation of rank n. Let us visualize a computation of T as operating with a stack, as described in Section 1.

How large can the stack grow during a computation? Let m be the maximum height of the stack, and for each $j \leqslant m$ let s_j be the last step in the computation that the stack went from a height less than j to one j or greater before reaching a height of m. Note that at step s_j, the height of the stack is less than $j + l$, and that between steps s_j and s_m the height of the stack never goes below j. Thus, if after each of two distinct steps s_j and $s_{j'}$ the last l symbols on the stack are identical and the current value of the variable begins with the same symbol a_i, then T will go into an infinite loop. Hence $m < (r + q)^l \cdot n \cdot l$.

Given this bound on the height of the stack, T has at most

$$n \cdot (r + q)^{(r+q)^l \cdot n \cdot l}$$

states under I, none of which are repeated in a halting computation. Thus for any sufficiently large k, $\text{len}_I(T) \leqslant k^n$.

(4.4) COROLLARY. *There is a recursion scheme which is not strongly equivalent to any program scheme or to any program scheme augmented by a single counter.*

Proof. The recursion scheme E in example (4.2) has $\text{len}(E, n) = 2^{n-1}$, so that the corollary follows from parts (b) and (c) of the theorem since for any k, $2^{n-1} > n^k$ for all sufficiently large n.

We note that neither the theorem nor the corollary can be extended to program schemes with two counters. In the first place, there are program schemes T with two counters such that $\text{len}(T, n)$ is not bounded by any recursive function of n, a fact which follows from the nonexistence of a recursive bound on the length of a computation of a universal two-register register machine. In the second place, as shown in Section 3, any recursion scheme is strongly equivalent to a program scheme with two counters. The chosen example 4.2 is almost optimal since any linear recursion scheme is strongly equivalent to a program scheme. However we do not know whether the recursion scheme

$$Fx := \text{if } Px \text{ then } fx \text{ else } FFfx,$$

which is strongly equivalent to a program scheme with a counter (cf. Section 1), is also strongly equivalent to a program scheme without a counter. We suspect that

it is not, though we have not been able to prove yet that there is any program scheme with a single counter which is not strongly equivalent to a program scheme without a counter.[5]

5. Decidable Cases of the Equivalence Problem

How powerful must a class of schemes be in order for it to be undecidable whether or not given pairs of schemes in the class are strongly equivalent? Simple classes of schemes (e.g., single-variable program schemes) have a decidable equivalence problem (Ianov [3]), while more complicated classes (e.g., program schemes) have an undecidable equivalence problem (Luckham *et al.* [4]); for other classes (e.g., recursion schemes), the status of the equivalence problem is unknown (cf. deBakker and Scott [1]). In this section we provide a partial answer to the problem raised by deBakker and Scott by showing that the equivalence problem for the class of linear recursion schemes is decidable. Furthermore, we investigate how large an extension of the class of single-variable program schemes possesses a decidable equivalence problem, showing in particular that the augmentation of such schemes by constant functions does not affect the decidability of the equivalence problem.

The techniques to be used can best be visualized as "cut and paste" techniques: we shall show that if two schemes in a given class produce different outputs under some interpretation I, but take a long time doing so, then it is possible to modify I slightly so that the two schemes skip over part of their previous computations (the "cutting and pasting") and still produce different outputs; a decision procedure is obtained since the equivalence of the two schemes in question does not depend upon their values under all interpretations, but only upon their values under a finite number of "small" interpretations. Historically, these techniques can be traced to their use by Rabin and Scott [7] to show the decidability of the equivalence problem for finite automata.

(5.1) Theorem. *There is an effective procedure for deciding whether two linear recursion schemes are strongly equivalent.*

Proof. By Lemma 1.2, it suffices to show that it is decidable whether, given any two linear recursion schemes R and S, there is a free interpretation under which they differ. For this it suffices to obtain an effective bound on the length of the shortest output from one scheme under a free interpretation under which the other scheme either diverges or produces a different output: given such a bound, one can check whether R is strongly equivalent to S by evaluating both schemes under the finite

[5] See footnote 1.

number of finite interpretations whose domains consist of all strings of function letters with length less than the bound.

Suppose that R and S have at most e recursion equations, and that each term in an equation has at most l symbols. Suppose also that I is a free interpretation under which R and S differ, and furthermore that the minimum n of the lengths of $\mathrm{val}_I(R)$ and $\mathrm{val}_I(S)$ (at least one of which is defined) is as small as possible. We shall show that $n < 3e^3l$. First we consider the case that both R and S halt under I.

By symmetry, we may assume that

$$\mathrm{val}_I(R) = w = f_n \cdots f_1 ,$$
$$\mathrm{val}_I(S) = w' = f'_{n'} \cdots f_1' ,$$

and that $n < n'$ or for some m, $f_i = f_i'$ for $0 < i < m$ but $f_m \neq f_m'$. Under the supposition that $n \geqslant 3e^3l$, we shall show that it is possible to define a new interpretation under which R and S produce different outputs which are shorter than w and w' and which consist of "parts" of w and w'.

Since R and S both halt under I, each can apply at most e equations to an input before changing it, i.e., in the computation of w or w' there is no sequence.

$$u_0 F_{i_0} v \Rightarrow u_1 F_{i_1} v \Rightarrow \cdots \Rightarrow u_e F_{i_e} v$$

in which e equations are evaluated without changing the argument to which the defined functions are applied. Since each term in an equation has at most l symbols, each time e equations are evaluated, at most el symbols can be added to the output. Hence, in order to produce outputs w and w' with lengths $\geqslant 3e^3l$, both R and S must change their input at least $3e^2$ times. Without loss of generality, we can assume that each time an input is changed, its length is increased by exactly one symbol; this assumption is possible since, for example, an equation $F := \alpha F' f_1 \cdots f_i$ can be replaced by the equations $F := F^i f_i$, $F^i := F^{i-1} f_{i-1}, \ldots, F^2 := \alpha F' f_1$, where $F^i, \ldots, F^2$ are new defined functions.

For each $i < 3e^2$, let E_i be the set of all pairs $\langle F_j^R, F_k^S \rangle$ of equations that R and S apply in the computation of w and w' to inputs of length i. Since there are at most e^2 pairs of equations, there must be particular equations F_j^R and F_k^S plus integers $i_1 < i_2 < i_3 < 3e^2$ such that $\langle F_j^R, F_k^S \rangle \in E_{i_1} \cap E_{i_2} \cap E_{i_3}$. In other words, the computations of w and w' must pass through the stages

$$\begin{aligned} F_1^R &\overset{*}{\Rightarrow} c_1 F_j^R a_1 \\ &\overset{*}{\Rightarrow} c_1 c_2 F_j^R a_2 a_1 \\ &\overset{*}{\Rightarrow} c_1 c_2 c_3 F_j^R a_3 a_2 a_1 \\ &\overset{*}{\Rightarrow} c_1 c_2 c_3 b a_3 a_2 a_1 = w \end{aligned} \qquad \begin{aligned} F_1^S &\overset{*}{\Rightarrow} c_1' F_k^S a_1' \\ &\overset{*}{\Rightarrow} c_1' c_2' F_k^S a_2' a_1' \\ &\overset{*}{\Rightarrow} c_1' c_2' c_3' F_k^S a_3' a_2' a_1' \\ &\overset{*}{\Rightarrow} c_1' c_2' c_3' b' a_3' a_2' a_1' = w' \end{aligned}$$

where

$$| a_1 | = | a_1' | = i_1 ,$$
$$| a_2 | = | a_2' | = i_2 - i_1 > 0,$$

and

$$| a_3 | = | a_3' | = i_3 - i_2 > 0.$$

The rest of the proof of this case is divided into subcases in order to show, contrary to our assumption, that w and w' are not the shortest differing outputs of R and S. In the first five subcases we suppose that w and w' differ first on the symbols f_m and f_m' and show how to produce shorter outputs which still differ; in the last subcase, we do the same supposing that $n < n'$.

Subcase 1. $f_m \in a_1 , f_m' \in a_1'$

Define a new free interpretation I_1 from I by setting, for any predicate P occurring in R or S,

$$P_{I_1}(x) = \begin{cases} P_I(ya_2a_1) & \text{if } x = ya_1 \\ P_I(ya_2'a_1') & \text{if } x = ya_1' \\ P_I(x) & \text{otherwise.} \end{cases}$$

Then, under the interpretation I_1,

$$R \text{ produces output } u_1 = c_1c_3ba_3a_1 ,$$

and

$$S \text{ produces output } u_1' = c_1'c_3'b'a_3'a_1'.$$

But $u_1 \neq u_1'$ since $a_1 \neq a_1'$, and $| u_1 | < | w |$, contradicting the choice of I.

Subcase 2. $f_m \in a_2 , f_m' \in a_2'$

Define a new free interpretation I_2 from I by

$$P_{I_2}(x) = \begin{cases} P_I(ya_3a_2a_1) & \text{if } x = ya_2a_1 \\ P_I(ya_3'a_2'a_1') & \text{if } x = ya_2'a_1' \\ P_I(x) & \text{otherwise.} \end{cases}$$

Then, under the interpretation I_2,

$$R \text{ produces output } u_2 = c_1c_2ba_2a_1 ,$$

and

$$S \text{ produces output } u_2' = c_1'c_2'b'a_2'a_1'.$$

Again $u_2 \neq u_2'$ since $a_2a_1 \neq a_2'a_1'$, and $|u_2| < |w|$, contradicting the choice of I.

Subcase 3. $f_m \in c_3ba_3$, $f_m' \in c_3'b'a_3'$

Since $a_1 = a_1'$ and $a_2 = a_2'$, the interpretation I_1 is well defined. As in Subcase 1, $u_1 \neq u_1'$ and $|u_1| < |w|$, contradicting the choice of I.

Subcase 4. $f_m \in c_3b$, $f_m' \in c_1'c_2'$ or $f_m \in c_1c_2$, $f_m' \in c_3'b'$

By symmetry, it suffices to consider the first alternative. Since $a_1 = a_1'$, $a_2 = a_2'$, and $a_3 = a_3'$, interpretations I_1 and I_2 are well defined, as is the interpretation I_3, where

$$P_{I_3}(x) = \begin{cases} P_I(ya_3a_2a_1) & \text{if } x = ya_1 \\ P_I(x) & \text{otherwise.} \end{cases}$$

Under interpretation I_3,

$$R \text{ produces output } u_3 = c_1ba_1$$
$$S \text{ produces output } u_3' = c_1'b'a_1'.$$

Since u_1, u_2, and u_3 are all shorter than w, the choice of I implies that $u_1 = u_1'$, $u_2 = u_2'$, and $u_3 = u_3'$. Since equal strings must have the same length, it follows that $|c_1b| = |c_1'b'|$, $|c_2| = |c_2'|$, and $|c_3| = |c_3'|$. Consequently, $n = n'$.

Consider now the other consequences of the equalities $u_i = u_i'$. Since $f_m \in c_3b$, f_m is the $(m - |a_2|)$-th symbol in u_1 and must equal the $(m - |a_2|)$-th symbol in u_1'; this symbol is $f'_{m+|c_2'|}$ since $f_m' \in c_1'c_2'$ and hence, $f'_{m+|c_2'|} \in c_1'$. Next, $f'_{m+|c_2'|}$ is the $(m - |a_3'a_2'| - |c_3'|)$-th symbol in u_3', while the matching symbol in u_3 is $f_{m-|c_3|}$ since $f'_{m-|c_3'|} \notin a_3'a_2'a_1'$, $|c_3'| = |c_3|$, and hence $f_{m-|c_3|} \in b$. Thus $f_m = f'_{m+|c_2'|} = f_{m-|c_3|}$. Finally, $f_{m-|c_3|}$ is the $(m - |a_3| - |c_3|)$-th symbol in u_2, while the matching symbol in u_2' is f_m' since $f_m' \in c_1'c_2'$. Thus $f_m = f_m'$, which is a contradiction.

Subcase 5. $f_m \in c_1c_2$, $f_m' \in c_1'c_2'$

Since $a_2a_1 = a_2'a_1'$ and $a_3 = a_3'$, the interpretation I_2 is well defined. As in Subcase 2, $u_2 \neq u_2'$ and $|u_2| < |w|$, contradicting the choice of I.

Subcase 6. $n < n'$ and $f_i = f_i'$ for $0 < i \leqslant n$

As in Subcase 4, $n = n'$, which is a contradiction.

Thus it is impossible for w and w' to be the shortest differing outputs of R and S if both have length $\geqslant 3e^3l$. This completes the proof of the case in which both $\mathrm{val}_I(R)$ and $\mathrm{val}_I(S)$ are defined.

Suppose now that $\mathrm{val}_I(S)$, say, is undefined. The proof is essentially the same as above, though with the following modifications. Suppose that

$$\mathrm{val}_I(R) = w = f_n \cdots f_1$$

and that $n \geqslant 3e^3l$. We show that for some I', $\mathrm{val}_{I'}(R)$ has length less than n, while $\mathrm{val}_{I'}(S)$ is still undefined. For each $i < 3e^2$, let E_i be the set of all pairs $\langle F_j^R, F_k^S \rangle$ of equations that R and S apply under I to arguments of length i or, if S does not apply any equations to arguments of length i, that S applies to arguments of maximum length. As above, the computations of R and S under I must pass through stages

$$\begin{aligned} F_1^R &\overset{*}{\Rightarrow} c_1F_j^Ra_1 & F_1^S &\overset{*}{\Rightarrow} c_1'F_k^Sa_1' \\ &\overset{*}{\Rightarrow} c_1c_2F_j^Ra_2a_1 & &\overset{*}{\Rightarrow} c_1'c_2'F_k^Sa_2'a_1' \\ &\overset{*}{\Rightarrow} c_1c_2c_3F_j^Ra_3a_2a_1 & &\overset{*}{\Rightarrow} c_1'c_2'c_3'F_k^Sa_3'a_2'a_1' \\ &\overset{*}{\Rightarrow} c_1c_2c_3ba_3a_2a_1 = w, \end{aligned}$$

where

$$\begin{aligned} |a_1| &= |a_1'| \text{ or } S \text{ never changes its input beyond } a_1', \\ 0 < |a_2| &= |a_2'| \text{ or } S \text{ never changes its input beyond } a_2'a_1', \\ 0 < |a_3| &= |a_3'| \text{ or } S \text{ never changes its input beyond } a_3'a_2'a_1'. \end{aligned}$$

The rest of the proof in this case is divided into three subcases analogous to the first three subcases above. In the first subcase, if $a_1 \neq a_1'$, then for the interpretation I_1 defined above, $|\mathrm{val}_{I'}(R)| < n$ and $\mathrm{val}_{I'}(S)$ is undefined. In the second subcase, if $a_1 = a_1'$ but $a_2 \neq a_2'$, then interpretation I_2 has the desired effect. Finally, if $a_2a_1 = a_2'a_1'$, then interpretation I_1 has the desired effect. Thus the proof is complete.

Having seen that a certain subclass of the class of recursion schemes has a decidable equivalence problem, we show the same for the class of single-variable program schemes with constant functions. The equivalence problem for single-variable program schemes without constant functions was shown to be decidable in Section 2. The equivalence problem for two variable program schemes is undecidable (cf. [4]) as is the problem for the class of program schemes with two independent variables and independent constants referred to in (3.9). Hence, the following theorem is optimal.

(5.2) THEOREM. *There is an effective procedure for deciding whether two single-variable program schemes with constant functions are strongly equivalent.*

Proof. Suppose that S is a single-variable program scheme, k of whose instructions are assignments involving constant functions, and let I be any interpretation. The *computation history* of S under I is the (possibly infinite) string $w = \cdots w_2c_2w_1c_1w_0$ of function letters applied by S to its input under I, where the symbols c_i are constant function letters and the w_i are strings of nonconstant function letters. Notice that if S executes the same assignment involving a constant function twice under I, then it must go into an infinite loop. Hence, the computation history w of S under I has one of two forms: (a) $w = w_nc_n \cdots w_1c_1w_0$, where $n \leqslant k$ and where w_n may be

an infinite string of nonconstant function letters if S does not halt under I, or (b) $w = \cdots (w_n c_n \cdots w_m c_m)(w_n c_n \cdots w_m c_m)\, w_{m-1} c_{m-1} \cdots w_1 c_1 w_0$, where $n \leqslant k$ and S loops under I.

We shall show that given any two single-variable program schemes R and S with constant functions, there is an effective bound such that if $R \not\equiv S$, then there is an interpretation under which R and S differ, and under which one of the schemes has a computation history with length less than the bound. As before, this suffices to prove the theorem since the equivalence of R and S can then be checked by evaluating them under a finite number of finite interpretations.

Suppose then that R and S are schemes with at most e instructions, of which at most k involve constant functions. Let I be a free interpretation under which R and S differ, and suppose, without loss of generality, that R and S have computation histories

$$w = \mathrm{val}_I(R) = w_n c_n \cdots w_1 c_1 w_0$$

and

$$w' = \mathrm{val}_I(S) = w'_{n'} c'_{n'} \cdots w_1' c_1' w_0$$

or

$$w' = \cdots (w'_{n'} c'_{n'} \cdots w'_{n''} c'_{n''})(w'_{n'} c'_{n'} \cdots w'_{n''} c'_{n''})\, w'_{n''-1} c'_{n''-1} \cdots w_1' c_1' w_0',$$

where w' may be infinite, $|\, w \,| \leqslant |\, w' \,|$, and $n, n' \leqslant k$. We show that if $|\, w \,| > k + (k+1)2^{2ke}$, then there is a free interpretation I' under which R and S differ, and under which R has a computation history of length less than $|\, w \,|$.

Suppose $|\, w \,| > k + (k+1)2^{2ke}$. Then for some $m \leqslant n$, $|\, w_m \,| > 2^{2ke}$. We shall treat the case $m > 0$, the other case being similar and simpler. Suppose $w_m = f_{|w_m|} \cdots f_1$. For each $i < |\, w_m \,|$, let A_i be the set of all triples $\langle j, r, l \rangle$ such that

(a) $j = 0$, $r \leqslant n$, $c_r = c_m$, $w_r = u f_i \cdots f_1$ for some string u, and instruction l of R was applied to $f_i \cdots f_1 c_r w_{r-1} \cdots c_1 w_0$, or

(b) $j = 1$, $r \leqslant n'$, $c_r' = c_m$, $w_r' = u f_i \cdots f_1$ for some string u, and instruction l of S was applied to $f_i \cdots f_1 c_r' w'_{r-1} \cdots c_1' w_0'$.

Since there are at most 2^{2ke} different such sets A_i, there must be integers $i_1 < i_2 \leqslant |\, w_m \,|$ such that $A_{i_1} = A_{i_2}$. Now for some strings a_1 of length i_1, a_2 of length $i_2 - i_1$ and a_3, $w_m = a_3 a_2 a_1$. Call an instance c_r (or c_r') of a constant function crucial if $c_r = c_m$ ($c_r' = c_m$) and $w_r = u a_1$ ($w_r' = u a_1$) for some string u. Then by the choice of i_1 and i_2, for any crucial c_r (or c_r'), $w_r = v a_2 a_1$ and some instruction of R is applied to both $a_2 a_1 c_r w_{r-1} \cdots c_1 w_0$ and $a_1 c_r w_{r-1} \cdots c_1 w_0$ (ditto for w_r').

Now define a new free interpretation I' such that for any predicate P,

$$P_{I'}(v) = \begin{cases} P_I(ua_2a_1c_m) & \text{if } v = ua_1c_m \\ P_I(v) & \text{otherwise.} \end{cases}$$

For any r, let

$$\hat{w}_r = \begin{cases} ua_1 & \text{if } c_r \text{ is crucial and } w_r = ua_2a_1 \\ w_r & \text{otherwise,} \end{cases}$$

and define $\hat{w}_r'$ similarly. Then the computation histories $\hat{w}$ and $\hat{w}'$ of R and S under I' are those strings obtained from their histories under I by replacing w_r by $\hat{w}_r$ and w_r' by $\hat{w}_r'$ throughout. Since $|\hat{w}_m| < |w_m|$, it follows that $|\hat{w}| < |w|$, so that it remains to be shown that $\mathrm{val}_{I'}(R) \not\simeq \mathrm{val}_I(S)$.

If S diverges under I, then S diverges under I' and hence $\mathrm{val}_{I'}(R) \not\simeq \mathrm{val}_{I'}(S)$. If S converges under I, then $w_nc_n \neq w_{n'}'c_{n'}'$ since $\mathrm{val}_I(R) \neq \mathrm{val}_I(S)$. There are three cases:

(1) Neither c_n nor $c_{n'}'$ is crucial. Then $\mathrm{val}_{I'}(R) = \hat{w}_nc_n = w_nc_n \neq w_{n'}'c_{n'}' = \hat{w}_{n'}'c_{n'}' = \mathrm{val}_{I'}(S)$.

(2) Exactly one of c_n and $c_{n'}'$ is crucial. Then, say, $\hat{w}_nc_n = ua_1c_m$ for some u, but $\hat{w}_{n'}'c_{n'}' \neq va_1c_m$ for any v; hence, $\mathrm{val}_{I'}(R) \neq \mathrm{val}_{I'}(S)$.

(3) Both c_n and $c_{n'}'$ are crucial. Then $w_n = ua_2a_1$, $w_{n'}' = va_2a_1$, and $u \neq v$, so that $\hat{w}_n = ua_1 \neq va_1 = \hat{w}_{n'}'$ and hence, $\mathrm{val}_{I'}(R) \neq \mathrm{val}_{I'}(S)$.
Thus, in any case, $\mathrm{val}_{I'}(R) \neq \mathrm{val}_{I'}(S)$, and the proof is complete.

We are in the process of using the combined techniques of Theorems 5.1 and 5.2 to show that the equivalence problem for linear recursion schemes with constant functions is decidable.

REFERENCES

1. J. W. DEBAKKER AND DANA SCOTT, "A Theory of Programs," August 1969, unpublished report.
2. J. E. HOPCROFT AND J. D. ULLMAN, "Formal Languages and their Relation to Automata," Addison-Wesley, Reading, MA, 1969.
3. IU. I. IANOV, The logical schemes of algorithms, *Problemy Kibernet.* 1 (1960), 82–140.
4. D. C. LUCKHAM, D. M. R. PARK, AND M. S. PATERSON, On formalized computer programs, *J. Comput. System Sci.* 4 (1970), 220–249.
5. M. NIVAT, "Transductions des Languages de Chomsky," Chapter 6, doctoral dissertation, Grenoble University, 1967.

6. M. S. PATERSON AND C. E. HEWITT, "Comparative Schematology," pp. 119–128, record of the Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970, published by the ACM Dec. 1970.
7. M. RABIN AND D. SCOTT, Finite automata and their decision problems, *IBM J. Res. Develop.* **3** (1959), 114–125.
8. J. C. SHEPHERDSON AND H. E. STURGIS, Computability of recursive functions, *J. Assoc. Comput. Mach.* **10** (1963), 217–255.