SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0012 $01.00/0

ON THE COMPUTATIONAL COMPLEXITY OF PROGRAM SCHEME EQUIVALENCE*

H. B. HUNT III†, R. L. CONSTABLE‡ AND S. SAHNI¶

Abstract. The computational complexity of several decidable problems about program schemes, recursion schemes, and simple programming languages is considered. The strong equivalence, weak equivalence, containment, halting, and divergence problems for the single variable program schemes and the linear monadic recursion schemes are shown to be *NP*-complete. The equivalence problem for the Loop 1 programming language is also shown to be *NP*-complete. Sufficient conditions for a program scheme problem to be *NP*-hard are presented. The strong equivalence problem for a subset of the single variable program schemes, the strongly free schemes, is shown to be decidable deterministically in polynomial time.

Key words. computational complexity, *P*, *NP*, *NP*-complete, program scheme, recursion scheme, equivalence, containment, halting, divergence, and isomorphism

Introduction. Early work with program schemes was motivated by a quest for program optimization techniques [9], [10], [13]. Ideally one would find a class of schemes rich enough to model many interesting programs but simple enough to have decidable problems such as equivalence, halting, or divergence. No attempt was made, however, to assess the computational complexity of such decidable problems. Here, we show that a variety of such decidable problems for the single variable program schemes, the linear monadic recursion schemes, and several simple programming languages are *NP*-complete.

The remainder of this paper is divided into four sections. Section 1 contains definitions and basic properties of p-reducibility, program schemes, and recursion schemes. In § 2 the strong equivalence, weak equivalence, containment, halting, and divergence problems for the single variable program schemes and the linear monadic recursion schemes are shown to be *NP*-complete. We also present general sufficient conditions for a problem on the single variable program schemes to be *NP*-hard. In § 3 we consider subclasses of the single variable program schemes for which strong equivalence is decidable deterministically in polynomial time. Finally in § 4, we briefly consider the complexity of the equivalence problem for several classes of simple programming languages including the Loop 1 languages in [14].

1. Definitions. We present definitions and basic properties of p-reducibility, program schemes, and monadic recursion schemes needed in §§ 2, 3, and 4. The definitions of strings, alphabets, context-free grammars, and derivations used here are from [7]. We denote the empty word by λ.

DEFINITION 1.1. $P(NP)$ is the class of all languages over $\{0, 1\}$ accepted by some deterministic (nondeterministic) polynomially time-bounded Turing machine.

DEFINITION 1.2. Let Σ and Δ be finite alphabets. Let $\mathscr{F}(\Sigma, \Delta)$ denote the set of all functions from Σ^* into Δ^* computable by some deterministic polynomially time-bounded Turing machine. Let L_1 and L_2 be subsets of Σ^* and Δ^*, respectively. We say

* Received by the editors February 14, 1975, and in final revised form May 2, 1979. This research was supported in part by the National Science Foundation under grants DCR-74-14701, GJ 35570, and DCR-75-22505.

† Department of Electrical Engineering and Computer Science, Columbia University, New York, New York 10027.

‡ Department of Computer Science, Upson Hall, Cornell University, Ithaca, New York 14853.

¶ Department of Computer, Information and Control Sciences, University of Minnesota, Minneapolis, Minnesota 55455.

that L_1 is *p-reducible* to L_2, written $L_1 \leqq_{\text{ptime}} L_2$, if there exists a function f in $\mathcal{F}(\Sigma, \Delta)$ such that, for all $x \in \Sigma^*$, $x \in L_1$, if and only if, $f(x) \in L_2$.

DEFINITION 1.3. A language L_0 is said to be *NP-hard* if, for all L in *NP*, $L \leqq_{\text{ptime}} L_0$. A language L_0 is said to be *NP-complete* if it is *NP*-hard and is accepted by some nondeterministic polynomially time-bounded Turing machine.[1]

DEFINITION 1.4. A Boolean form f is a *D_3-Boolean form* if f is the disjunction of clauses $C_1, \cdots, C_p$ such that each clause C_i is the conjunction of at most three literals. A Boolean form f is a *C_3-Boolean form* if f is the conjunction of clauses $C_1, \cdots, C_p$ such that each clause C_i is the disjunction of at most three literals.

PROPOSITION 1.5[3]. *The sets $\mathcal{T}_1 = \{f \mid f$ is a nontautological D_3-Boolean form$\}$ and $\mathcal{T}_2 = \{f \mid f$ is a satisfiable C_3-Boolean form$\}$ are NP-complete.*

PROPOSITION 1.6[3]. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages. If $\mathcal{L}_1$ is NP-hard and $\mathcal{L}_1$ is p-reducible to $\mathcal{L}_2$, then $\mathcal{L}_2$ is NP-hard.*

DEFINITION 1.7. Let D be a set. A *predicate on D* is a function from D into {True, False}.

We assume that the reader is familiar with the basic properties and results concerning program schemes, monadic recursion schemes, and interpretations as presented in [1], [4], [10].

Program schemes are defined as follows. Let $\mathcal{L}$, $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{P}$ be mutually disjoint sets of labels, variable symbols, function symbols, and predicate symbols, respectively. A *program scheme* S is a finite nonempty sequence of

(1) *assignment statements* of the form $k \,.\, y \leftarrow f(x_1, \cdots, x_n)$, where k in $\mathcal{L}$ is a label, f in $\mathcal{F}$ is an n-ary function symbol, and $x_1, \cdots, x_n$, y in $\mathcal{V}$ are variable symbols;

(2) *conditional statements* of the form k. *If* $P_i(x_1, \cdots, x_n)$ *then* k_1 *else* k_2, where k, k_1, and k_2 are labels, P_i in $\mathcal{P}$ is an n-ary predicate symbol, and $x_1, \cdots, x_n$ in $\mathcal{V}$ are variable symbols; and

(3) *halt statements* of the form k. *Halt*, where k is a label.

We sometimes allow *loop statements* of the form k. *Loop* as abbreviations for the statement.

$$k \,.\, \textbf{If } P_i(x_1, \cdots, x_n) \textbf{ then } k \textbf{ else } k.$$

We frequently assume that the first element of S is its initial statement and the last element of S is either a loop or halt statement.

The meaning of a program scheme S is defined in terms of interpretations. Formally, an *interpretation* I of S consists of

(1) a nonempty set D, called the *domain* of I;

(2) an assignment of an element of D to each variable symbol in $\mathcal{V}$;

(3) an assignment of a function $f^I : D^n \to D$ to every n-ary function symbol f in $\mathcal{F}$; and

(4) an assignment of a predicate $P_i^I : D^n \to D$ to every n-ary predicate symbol P_i in $\mathcal{P}$.

The definition of a *computation of* a program scheme *S under an interpretation I* can be found in [10]. The *value of S under I*, denoted by $\text{val}_I(S)$, is the final value of the distinguished output variable of S if the computation of S under I halts; and is undefined otherwise.

A *monadic recursion scheme S* is a finite list of definitional equations

$$F_1x := \textbf{If } P_1x \textbf{ then } \alpha_1 x \textbf{ else } \beta_1 x,$$

$$F_nx := \textbf{If } P_nx \textbf{ then } \alpha_n x \textbf{ else } \beta_n x$$

[1] Definition 1.3 extends the concept of *NP*-completeness to languages over arbitrary finite alphabets.

where $F_1, \cdots, F_n$ are *defined function symbols*; $P_1, \cdots, P_n$ are (not necessarily distinct) *predicate symbols*; and $\alpha_1, \beta_1, \cdots, \alpha_n, \beta_n$ are (possibly empty) strings of defined and basis symbols. A monadic recursion scheme S is said to be *linear* if at most one defined function symbol occurs in each of the strings $\alpha_1, \beta_1, \cdots, \alpha_n, \beta_n$.

The semantics of a monadic recursion scheme S is also defined in terms of interpretations. Formally, an *interpretation* I of a monadic recursion scheme S consists of

(1) a nonempty set D, called the *domain* of I;
(2) an assignment of a function $f^I : D \to D$ to every basis function symbol f is S;
(3) an assignment of a predicate $P_i^I : D \to \{\text{True, False}\}$ to every predicate symbol P_i in S; and
(4) an assignment of an element x^I of D to x.

An interpretation I of a monadic recursion scheme S, with set of basis function symbols $\mathcal{F}$, is said to be *free* if

(i) the domain D of I equals $[\mathcal{F}]^* \cdot \{x\}$; and
(ii) for all f in $\mathcal{F}$ and strings wx in $[\mathcal{F}]^* \cdot \{x\}$, $f^I(wx)$ equals the string fwx.

For any interpretation I, $(f_1 \cdots f_n x)^I = (f_1)^I(\cdots(f_n)^I(x^I)\cdots)$.

The computations of a monadic recursion scheme can be defined in terms of context-free grammars as follows. To each scheme

$$S.\ F_i x := \textbf{If } P_i x \textbf{ then } \alpha_i x \textbf{ else } \beta_i x \qquad (1 \leqq i \leqq n),$$

we associate a context-free grammar G_S with terminal alphabet equal to $\mathcal{F}$, nonterminal alphabet F equal to $\{F_1, \cdots, F_n\}$, and set of productions equal to $\{F_i \to \alpha_1, F_i \to \beta_i | 1 \leqq i \leqq n\}$. Let I be an interpretation. Following [4] we say that a rightmost derivation of G_S is *legal* for I if, for every step in the deviation of the form $\gamma F_i w \underset{G_S}{\Longrightarrow} \gamma\delta w$, where $\gamma, \delta \in (\mathcal{F} \cup F)^*$ and $w \in \mathcal{F}^*$, $\delta = \alpha_i$ if $P_i^I(w^I) = \text{True}$ and $\delta = \beta_i$ if $P_i^I(w^I) = \text{False}$. The computation of S under I corresponds to the unique legal derivation for I. If $F_1 \underset{G_S}{\Longrightarrow}^* w$ for $w \in \mathcal{F}^*$ by the legal derivation for I, then $\text{val}_I(S) = w^I(x^I)$; otherwise, $\text{val}_I(S)$ is undefined.

Finally we assume that there is a finite alphabet Σ such that each scheme or program S is presented as a string σ_S over Σ. We say that the length of the string σ_S is the *size* of S.

DEFINITION 1.8. Let S and S' be program or monadic recursion schemes. We say that

(1) S *halts* if, for all interpretations I of S, the computation of S under I halts;
(2) S *diverges* if, for all interpretations I of S, the computation of S under I does not halt;
(3) S and S' are *strongly equivalent* if, for all interpretations I, either both of $\text{val}_I(S)$ and $\text{val}_I(S')$ are undefined, or both of $\text{val}_I(S)$ and $\text{val}_I(S')$ are defined and are equal;
(4) S and S' are *weakly equivalent* if, for all interpretations I for which both of $\text{val}_I(S)$ and $\text{val}_I(S')$ are defined, $\text{val}_I(S)$ equals $\text{val}_I(S')$ and
(5) S *contains* S' if, for all interpretations I for which $\text{val}_I(S')$ is defined, $\text{val}_I(S)$ is defined and equals $\tilde{\text{val}}_I(S')$.

Let S and S' be program schemes. We say that

(6) S is *isomorphic to* S' if, for all interpretations I, the sequences of the instructions executed by the computations of S and S' under I are the same.

Definition 1.9[10]. Let ρ be any binary relation on the program schemes or on the monadic recursion schemes such that, for all schemes S and S',

(1) if S and S' are strongly equivalent, then $S\rho S'$; and

(2) if $S\rho S'$, then S and S' are weakly equivalent.

Then, the relation ρ is said to be a *reasonable relation.*

2. Program and recursion schemes. A variety of decidable problems on the single variable program schemes (abbreviated svp schemes) and on the linear monadic recursion schemes (abbreviated lmr schemes) are shown to be NP-complete. These problems include strong equivalence, weak equivalence, containment, halting, and divergence. This is accomplished in two steps. First, we show that these problems are NP-hard for the svp schemes. Second, we show that these problems are in NP for the lmr schemes.

DEFINITION 2.1. A *switching scheme S* is a monadic, loop-free, svp scheme such that each of its statements is either a conditional or a halt statement.

Our first proposition relates the tautology problem for D_3-Boolean forms to the problem of deciding, for a switching scheme S with halt statement labeled B, if the statement labeled by B is executed during some computations of S. All our NP-hard lower bounds follow from it.

PROPOSITION 2.2. *There exists a deterministic polynomially time bounded Turing machine M_0 such that M_0, given a D_3-Boolean form f as input, outputs a switching scheme S_f with exactly two halt statements labeled* **A** *and* **B** *such that the statement labelled* **B** *is executed during some computation of S_f, if and only if, f is not a tautology.*

Proof. We illustrate how M_0 constructs S_f from f by an example. Suppose f equals $x_1\bar{x}_2x_4 \vee x_2\bar{x}_3x_4 \vee x_1\bar{x}_4\bar{x}_5$. Then, S_f is the following:

1. **If** $P_1(x)$ **then** 2 **else** 4
2. **If** $P_2(x)$ **then** 4 **else** 3
3. **If** $P_4(x)$ **then A else** 4
4. **If** $P_2(x)$ **then** 5 **else** 7
5. **If** $P_3(x)$ **then** 7 **else** 6
6. **If** $P_4(x)$ **then A else** 7
7. **If** $P_1(x)$ **then** 8 **else B**
8. **If** $P_4(x)$ **then B else** 9
9. **If** $P_5(x)$ **then B else A**

A. Halt.

B. Halt.

We denote the set $\{S_f | f$ is a D_3-Boolean form; and the Turing machine M_0 of Proposition 2.2, given input f, outputs $S_f\}$ by $\mathscr{C}$.

DEFINITION 2.3. Let S be an svp scheme with exactly two halt statements labeled **A** and **B**. Let $\mathscr{A}$ and $\mathscr{B}$ be svp schemes. The program scheme $[S, \mathscr{A}, \mathscr{B}]$ is the program scheme that results from S by replacing the statement labeled **A** in S by $\mathscr{A}$ and by replacing the statement labeled **B** in S by $\mathscr{B}$, with a suitable renumbering of the statements in $\mathscr{A}$ and $\mathscr{B}$ as necessary.

For example, let S, $\mathscr{A}$, and $\mathscr{B}$ be the following:

S: 1. **If** $P_1(x)$ **then** 2 **else** 3	$\mathscr{A}$: 1. $x \leftarrow f(x)$
2. $x \leftarrow x$	2. **Halt**.
A. Halt.	
3. $x \leftarrow x$	$\mathscr{B}$: 1. $x \leftarrow g(x)$
B. Halt.	2. **Halt**

Then, $[S, \mathcal{A}, \mathcal{B}]$ is the following:

1. **If** $P_1(x)$ **then** 2 **else** 3
2. $x \leftarrow x$
A. $x \leftarrow f(x)$
4. **Halt.**
3. $x \leftarrow x$
B. $x \leftarrow g(x)$
5. **Halt.**

The next theorem gives general sufficient conditions for a predicate on the svp schemes to be NP-hard.

THEOREM 2.4. *Let Π be any predicate on the svp schemes for which there exist svp schemes $\mathcal{A}$ and $\mathcal{B}$ such that, for all schemes S in $\mathcal{C}$, $\Pi([S, \mathcal{A}, \mathcal{B}])$ equals False if and only if the statement of S labelled* **B** *is executed during some computation of S. Then, the set $\{S|S$ is an svp scheme and $\Pi(S)$ equals False$\}$ is NP-hard. Moreover, if $\mathcal{A}$ and $\mathcal{B}$ are loop-free, then the set $\{S|S$ is a loop-free svp scheme and $\Pi(S)$ equals False$\}$ is also NP-hard.*

Proof. By Propositions 1.5 and 1.6, it suffices to show that the set $\mathcal{T}_1$ of nontautological D_3-Boolean forms is p-reducible to the set $\{S|S$ is an svp scheme and $\Pi(S)$ equals False$\}$. Let f be a D_3-Boolean form. Let S_f be the corresponding element of $\mathcal{C}$. Then, $\Pi([S_f, \mathcal{A}, \mathcal{B}])$ equals False, if and only if, the statement in S_f labeled **B** is executed during some computation of S_f. By Proposition 2.2 this is true, if and only if, f is not a tautology. Since $[S_f, \mathcal{A}, \mathcal{B}]$ is constructible from f by a deterministic polynomially time-bounded Turing machine, the theorem follows. QED

The next two corollaries yield some applications of Theorem 2.4. Henceforth, we denote the svp scheme 1. Halt. by $\mathcal{I}$.

COROLLARY 2.5. *Let Π be any of the following predicates on the svp schemes:*

(i) *S diverges;*
(ii) *S halts;*
(iii) *S is strongly equivalent to $\mathcal{I}$;*
(iv) *S contains $\mathcal{I}$;*
(v) *$\mathcal{I}$ contains S;*
(vi) *S is weakly equivalent to $\mathcal{I}$; and*
(vii) *for all reasonable relations ρ on the svp schemes, $S\rho\mathcal{I}$.*

Then, the set $\{S|S$ is an svp scheme and $\Pi(S)$ equals False$\}$ is NP-hard.

Proof. Each of the predicates in (i) through (vii) satisfies the conditions of Theorem 2.4, where the corresponding schemes $\mathcal{A}$ and $\mathcal{B}$ are as follows:

(i)	$\mathcal{A}$ is 1. Loop.	$\mathcal{B}$ is 1. Halt.
(ii)	$\mathcal{A}$ is 1. Halt.	$\mathcal{B}$ is 1. Loop.
(iii) through (viii)	$\mathcal{A}$ is 1. Halt.	$\mathcal{B}$ is 1. $x \leftarrow f(x)$ 2. Halt.

Q.E.D.

COROLLARY 2.6. *Let ρ be any of the following binary relations on the svp schemes: for all svp schemes S and S', $S\rho S'$, if and only if,*

(i) *S is isomorphic to S';*
(ii) *S is strongly equivalent to S';*
(iii) *S contains S';*
(iv) *S is weakly equivalent to S'; and*
(v) *for all reasonable relations ρ_0 on the svp schemes, $S\rho_0 S'$.*

Then, the set $\{(S, S')|S$ and S' are svp schemes and $\sim(S\rho S')\}$ is NP-hard. Moreover, the set $\{(S, S')|(S$ and S' are loop-free svp schemes and $\sim(S\rho S')\}$ is also NP-hard.

Proof. The conclusions of this corollary, for the relations of (ii) through (v), follow easily from Theorem 2.4 and Corollary 2.5. Therefore, we only prove that the conclusions of this corollary hold for isomorphism. As in the proof of Theorem 2.4, we show that the set $\mathcal{T}_1$ of nontautological D_3-Boolean forms is p-reducible to the set $\{(S, S')|S$ and S' are loop-free svp schemes and S is not isomorphic to $S'\}$.

Let f be a D_3-Boolean form. Let S_f be the corresponding element of $\mathscr{C}$. Then, letting $\mathscr{B}_0$ denote the scheme

1. $x \leftarrow g(x)$
2. **Halt**.

the schemes $[S_f, \mathscr{I}, \mathscr{B}_0]$ and $[S_f, \mathscr{I}, \mathscr{I}]$ are isomorphic, if and only if, the statement in S_f labeled **B** is not executed during some computation of S_f. By Proposition 2.2 this is true, if and only if, f is a tautology. Since the schemes $[S_f, \mathscr{I}, \mathscr{B}_0]$ and $[S_f, \mathscr{I}, \mathscr{I}]$ are loop-free and are constructible from f by a deterministic polynomially time-bounded Turing machine, the corollary follows. Q.E.D.

The importance of Theorem 2.4 and Corollaries 2.5 and 2.6 lies in the weakness of the hypotheses needed to show that any predicate satisfying their conditions is NP-hard. Since no looping except possibly loop statements and only monadic functions and predicates are required, their conclusions hold for many other classes of program schemes, e.g. the monadic program schemes with nonintersecting loops, the liberal schemes, and the progressive schemes, see [10], [12]. In the remainder of this section, we show that similar results hold for the lmr schemes and that several of these NP-hard problems are NP-complete.

The effective translation of monadic svp schemes into strongly equivalent lmr schemes in [4] can easily be seen to be executable by a deterministic polynomially time-bounded Turing machine. Thus letting $\mathscr{I}'$ denote the lmr scheme

$$F_1x := \textbf{If } P_1x \textbf{ then } x \textbf{ else } x,$$

one immediate implication of Corollaries 2.5 and 2.6 is the following.

COROLLARY 2.7 (1). *Let Π be any of the following predicates on the lmr schemes:*

(i) *S halts*;

(ii) *S diverges*;

(iii) *S is strongly equivalent to $\mathscr{I}'$*;

(iv) *S contains $\mathscr{I}'$*;

(v) *$\mathscr{I}'$ contains S*;

(vi) *S is weakly equivalent to $\mathscr{I}'$*; *and*

(vii) *for all reasonable relations ρ on the lmr schemes, $S\rho\mathscr{I}'$. Then, the set $\{S|S$ is an lmr scheme and $\Pi(S)$ equals False$\}$ is NP-hard.*

(2) *Let ρ be any of the following binary relations on the lmr schemes: for all lmr schemes S and S', $S\rho S'$, if and only if,*

(viii) *S is strongly equivalent to S'*;

(ix) *S contains S'*;

(x) *S is weakly equivalent to S'*; *and*

(xi) *for all reasonable relations ρ_0 on the lmr schemes, $S\rho_0 S'$. Then, the set $\{(S, S')|S$ and S' are lmr schemes and $\sim(S\rho S')\}$ is NP-hard.*

The next two propositions will be used to derive upper bounds on the computational complexity of halting, divergence, strong equivalence, weak equivalence, and containment for the lmr and svp schemes. The first proposition is new. The second closely follows results in [4].

PROPOSITION 2.8. *Let R be an lmr scheme with n defining equations. Then, R diverges for some interpretation if and only if there exists a free interpretation I of R for which the computation of R under I takes at least $2n+1$ steps.*

Proof. The "only if" part is obvious. We show the "if" part. Suppose the computation of R under I takes at least $2n+1$ steps. Then some defining equation, say

$$F_i x := \textbf{If } P_i x \textbf{ then } \alpha_i x \textbf{ else } \beta_i x,$$

must be applied at least three times during it. Hence, the computation of R under I must contain at least two applications of this equation for which the predicate P_i^I takes the same value. Thus letting G_S be the context-free grammar associated with S, there exist strings b_1, b_2, c_1, and c_2 of basis function symbols such that

$$F_1 \underset{G_S}{\overset{*}{\Longrightarrow}} b_1 F_i c_1,$$

$$F_i \underset{G_S}{\overset{+}{\Longrightarrow}} b_2 F_i c_2,$$

$$P_i^I(c_1 x) = P_i^I(c_2 c_1 x)$$

for the legal derivation for I.

If c_2 equals λ, then the computation of R under I diverges. Otherwise, let I_0 be the free interpretation of R defined by; for all predicate symbols P_i in R;

$$P_i^{I_0}(wx) = \begin{cases} P_i^I(\alpha c_1 x), & \text{if } w = \alpha c_2^k c_1 \text{ and } \alpha \text{ is a suffix of } c_2; \\ P_i^I(wx), & \text{otherwise.} \end{cases}$$

Then, the computation of R under I_0 diverges.

PROPOSITION 2.9. *Let R and S be two lmr schemes, with set of basis function symbols $\mathscr{F}$ and set of defined function symbols F such that*

(i) *both of R and S have at most n defining equations;*

(ii) *the length of each string α_i and β_i in a defining equation of R or S is less than m; and*

(iii) *each string α_i and β_i in a defining equation of R or S is an element of $\mathscr{F}^* \cdot F \cdot (\mathscr{F} \cup \{\lambda\}) \cup \phi \cup \{\lambda\}$.*

Then, (1) if there exists an interpretation I' under which R and S differ but for which both of $\mathrm{val}_{I'}(R)$ and $\mathrm{val}_{I'}(S)$ are defined, then there is a free interpretation I, under which R and S differ and for which both of $\mathrm{val}_I(R)$ and $\mathrm{val}_I(S)$ are defined, such that the minimum of the lengths of $\mathrm{val}_I(R)$ and $\mathrm{val}_I(S)$ is less than $3n^3m$. Similarly, (2) if there exists an interpretation I' for which $\mathrm{val}_{I'}(R)$ is defined and $\mathrm{val}_{I'}(S)$ is not, then there is a free interpretation I, for which $\mathrm{val}_I(R)$ is defined and $\mathrm{val}_I(S)$ is not, such that the length of $\mathrm{val}_I(R)$ is also less than $3n^3m$.

The proofs of (1) and (2) appear on pages 154–157 in [4].

THEOREM 2.10. *The following sets are NP-complete:*

(i) $S_1 = \{S \mid S$ *is an lmr scheme; and S does not halt*$\}$;

(ii) $S_2 = \{S \mid S$ *is an lmr scheme; and S does not diverge*$\}$;

(iii) $S_3 = \{(S, S') \mid S$ *and S' are lmr schemes; and S and S' are not strongly equivalent*$\}$;

(iv) $S_4 = \{(S, S') \mid S$ *and S' are lmr schemes; and S and S' are not weakly equivalent*$\}$;

and

(v) $S_5 = \{(S, S') \mid S$ *and S' are lmr schemes; and S does not contain S'*$\}$.

Proof. By Corollary 2.7 each of these sets is NP-hard. We illustrate how Propositions 2.8 and 2.9 can be used to show that these sets are in NP. We only sketch the proofs for S_1 and S_4. The proofs for S_2, S_3, and S_5 are similar and are left to the reader.

(i) Let M be the nondeterministic Turing machine that operates as follows:

Step 1. M, given input S, checks if S is an lmr scheme. If not, M halts without accepting.

Step 2. M guesses a rightmost derivation Π of the context-free grammar G_S associated with S of the form $F_1 \underset{G_S}{\Longrightarrow} b_{i_1}F_{i_1}c_{i_1} \underset{G_S}{\Longrightarrow} \cdots \underset{G_S}{\Longrightarrow} b_{i_k}F_{i_k}c_{i_k}$, where, letting n_0 be the number of the defining equations of S, $k = 2n_0+1$; $F_1, F_{i_1}, \cdots, F_{i_k}$ are defined functions symbols of S; and $b_{i_1}, c_{i_1}, \cdots, b_{i_k}, c_{i_k}$ are strings of basis function symbols of S.

Step 3. M verifies that Π is legal for some free interpretation I of S. If so, M accepts S. Otherwise, M halts without accepting.

By Proposition 2.8, M accepts S_1. Moreover, M is polynomially time-bounded. This follows since

(1) the lengths of each of the sentential forms $F_1, b_{i_1}F_{i_1}c_{i_1}, \cdots, b_{i_k}F_{i_k}c_{i_k}$ in Π is less than $(2n_0+1)\cdot(m+1)+1$, where m is an upper bound on the lengths of the strings α_i, β_i in the defining equations of S; and

(2) Π is legal for some interpretation I, if and only if, for each pair $(b_{i_j}F_{i_j}c_{i_j}, b_{i_{j'}}F_{i_{j'}}c_{i_{j'}})$ of sentential forms in Π, if

$$b_{i_j}F_{i_j}c_{i_j} \underset{G_S}{\Longrightarrow} b_{i_j}\delta c_{i_j},$$

$$b_{i_{j'}}F_{i_{j'}}c_{i_{j'}} \underset{G_S}{\Longrightarrow} b_{i_{j'}}\delta' c_{i_{j'}},$$

$$c_{i_j} = c_{i_{j'}},$$

then $\delta = \delta'$.

Clearly conditions (1) and (2) can be checked deterministically in time bounded by a polynomial in the size of S.

(iv) Let M be the nondeterministic Turing machine that operates as follows:

Step 1. M, given input (S, S') checks if S and S' are lmr schemes. If not, M halts without accepting.

Step 2. M converts S and S' into strongly equivalent lmr schemes S_1 and S_1', respectively, that satisfy the conditions of Proposition 2.9.

Step 3. M guesses a rightmost derivation Π of G_{S_1}

$$F_1 \underset{G_{S_1}}{\Longrightarrow} \cdots \underset{G_{S_1}}{\Longrightarrow} b_{i_k}F_{i_k}c_{i_k} \underset{G_{S_1}}{\Longrightarrow} b_{i_{k+1}}c_{i_{k+1}}$$

and a rightmost derivation Π' of $G_{S_1'}$

$$F_1' \underset{G_{S_1'}}{\Longrightarrow} \cdots \underset{G_{S_1'}}{\Longrightarrow} b_{i_l}'F_{i_l}'c_{i_l}' \underset{G_{S_1'}}{\Longrightarrow} b_{i_{l+1}}'c_{i_{l+1}}',$$

where $k+1$ and $l+1$ are less than $3n^3m^3$, and $b_{i_{k+1}}c_{i_{k+1}} \neq b_{i_{l+1}}'c_{i_{l+1}}'$.

[Here, n equals the maximum of the number of defining equations in S and S'; and m equals the maximum of the lengths of the strings α_i and β_i in any of the defining equations of S_1 and S_1'.]

Step 4. M verifies that both of Π and Π' are legal for some interpretation. If so, M accepts (S, S'). Otherwise, M halts without accepting.

By Proposition 2.9 M accepts S_4. Moreover, M is polynomially time-bounded.

This follows by reasoning analogous to that in the proof of (i) and is left to the reader. Q.E.D.

COROLLARY 2.11. *The following sets are NP-complete*:

(i) $\{S|S$ *is an svp scheme; and* S *does not halt*$\}$;

(ii) $\{S|S$ *is an svp scheme; and* S *does not diverge*$\}$;

(iii) $\{(S, S')|S$ *and* S' *are svp schemes; and they are not strongly equivalent*$\}$;

(iv) $\{(S, S')|S$ *and* S' *are svp schemes; and they are not weakly equivalent*$\}$;

(v) $\{(S, S')|S$ *and* S' *are svp schemes; and* S *does not contain* $S'\}$; *and*

(vi) $\{(S, S')|S$ *and* S' *are loop-free svp schemes; and* S *and* S' *are not strongly equivalent*$\}$.

3. A deterministic polynomial time decidable equivalence problem.

3.1. Strongly free schemes. In § 2 we saw that the strong equivalence problem for the svp schemes is *NP*-complete and thus is likely to be computationally intractable. Here, we inquire if any interesting subclasses of the svp schemes have provably deterministic polynomial time decidable strong equivalence problems. We note that the proof above that strong equivalence for the svp schemes is *NP*-hard involves sieves of predicates of the type appearing in Figure 3.1 where (a) some predicates, such as P_1, P_2, and P_3 in Figure 3.1, test the same value twice; and (b) the sieve is a directed acyclic

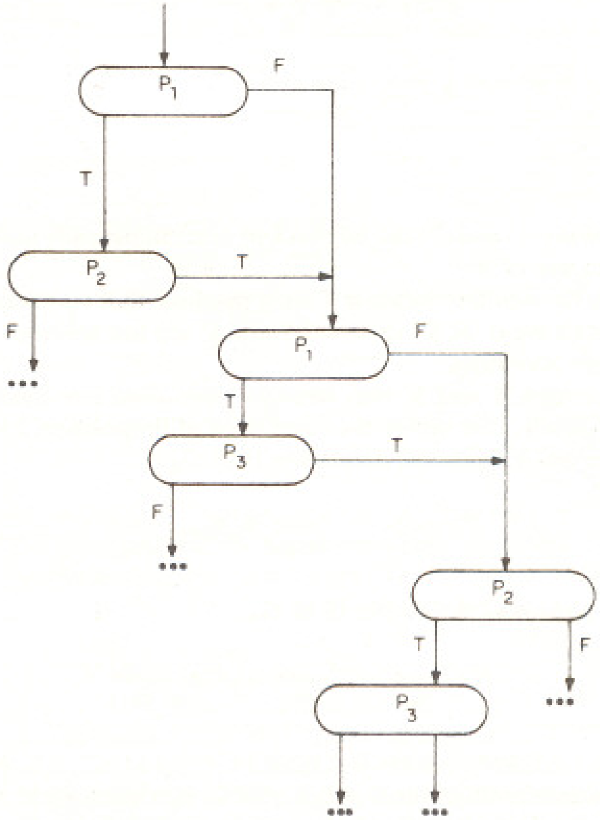


FIG. 3.1

graph but not a tree. Schemes with predicates satisfying (a) have the property that not all paths are executable and thus are unlikely to correspond to well-written computer programs. This suggests that we consider svp schemes which have no such predicates.

Using the terminology of [4], [10], svp schemes with no predicates satisfying (a) or, equivalently, svp schemes in which all paths are executable are said to be *free*. Thus, we are led to the question—

Q1: "Do free svp schemes have a deterministic polynomial time decidable strong equivalence problem?"

Only a partial answer to question Q1 is presented here. We show that the class of svp schemes in which no two predicates test the same value in a Herbrand interpretation, called the strongly free schemes, has a deterministic polynomial time decidable strong equivalence problem.

In a strongly free svp scheme there is a function application between any two predicates. These schemes behave like deterministic finite automata; and our technique for showing that their strong equivalence problem is decidable deterministically in polynomial time is to consider them as deterministic finite automata (as described below). There is one nontrivial difficulty, however. A strongly free scheme S may have redundant predicates, i.e. predicates whose left and right branches are equivalent. To obtain a deterministic polynomial time strong equivalence test, we must find a deterministic polynomial time redundancy test. This is accomplished by modifying the usual state minimization algorithm for deterministic finite automata.

Before presenting the results of this section, we need some notation. Recall that the value of a scheme S under interpretation H is denoted by val (S, H). We denote the value of a scheme S under interpretation H starting with statement L_0 by val (S, H, L_0). With every svp scheme S, we associate the three languages $L(S)$, $L^c(S)$, and $L^{c\#}(S)$ defined as follows.

DEFINITION 3.2. The *value language* of an svp scheme S, denoted by $L(S)$, is the set $\{\text{val}(S, H) | H$ is a Herbrand interpretation for which S halts$\}$.

Value languages were extensively used in [4].

DEFINITION 3.3. Let S be an svp scheme. Let H be a Herbrand interpretation. The *computation string* of S under H, denoted by Comp (S, H) is the (possibly infinite) string

$$\cdots \alpha_{m+1} P^{\pm}_{i_m} \cdots P^{\pm}_{i_2} \alpha_2 P^{\pm}_{i_1} \alpha_1$$

such that each α_i is a (possibly empty) string of function symbols of S, $P^{\pm}_{i_j}$ is either $P^{+}_{i_j}$ or $P^{-}_{i_j}$ where P_{i_j} is a predicate symbol of S, and

$$P^{\pm}_{i_j} \text{ is } P^{+}_{i_j}, \text{ if and only if, } (P_{i_j})^H(\alpha_j \cdots \alpha_1) = \text{True}.$$

The *computation language* of S, denoted by $L^c(S)$, is the set $\{\text{Comp}(S, H) | H$ is a Herbrand interpretation$\}$. The *terminating computation language* of S, denoted by $L^{c\#}(S)$, is set $\{\text{Comp}(S, H) | H$ is a Herbrand interpretation for which S halts$\}$.

The proof of the following lemma about terminating computation languages is left to the reader.

LEMMA 3.4. *For svp schemes S_1 and S_2, if $L^{c\#}(S_1) = L^{c\#}(S_2)$, then*

(i) *S_1 and S_2 halt for the same Herbrand interpretations*; *and*

(ii) *for all Herbrand interpretations for which both S_1 and S_2 halt*, Comp $(S_1, H) =$ Comp (S_2, H).

Recall that an svp scheme S is said to be free if *no* predicate is tested twice with the same argument values under any Herbrand interpretation. This implies that there must be a function application between any two separate occurrences of the same predicate test. We define a similar but stronger notion of freedom.

DEFINITION 3.5. An svp scheme S is said to be *strongly free* if and only if no two predicates test the same value in any Herbrand interpretation.

For strongly free svp schemes, there must be a function application between any two predicate tests.

DEFINITION 3.6. The occurrence of a predicate P in statement L_1 in a scheme S, say

$$L_1.\ \textbf{If}\ P(x)\ \textbf{then}\ L_l\ \textbf{else}\ L$$

is said to be *superfluous*, if and only if, val $(S, H, L_l) =$ val (S, H, L_r) for all Herbrand interpretations H. More generally, two statements L_1 and L_2 in schemes S_1 and S_2, respectively, are said to be *equivalent*, if and only if, val $(S_1, H, L_1) =$ val (S_2, H, L_1) for all Herbrand interpretations H. Finally, a scheme S is said to be *reduced*, if and only if, it contains no superfluous predicate occurrences.

Our first theorem shows how reduced strongly free svp schemes can be characterized by their terminating computation languages. It will be used to show how reduced strongly free svp schemes behave like deterministic finite automata.

THEOREM 3.7. *If S_1 and S_2 are reduced strongly free svp schemes, then $S_1 \equiv S_2$ if and only if $L^{c\#}(S_1) = L^{c\#}(S_2)$.*

Proof. (1) ($\Rightarrow$) Let $S_1 \equiv S_2$. We show that $L^{c\#}(S_1) = L^{c\#}(S_2)$ by proof by contradiction. Suppose $L^{c\#}(S_1) \neq L^{c\#}(S_2)$. Let $x = x_n \cdots x_1$ be a string in one of $L^{c\#}(S_1)$ and $L^{c\#}(S_2)$ but not both, say $x \in L^{c\#}(S_1) - L^{c\#}(S_2)$. Let x^1 be the subsequence of x obtained by deleting all predicate tests. Then, there exists a string $y = y_m \cdots y_1$ in $L^{c\#}(S_2)$ such that

(a) letting y^1 be the subsequence of y obtained by deleting all predicate tests, we have $y^1 = x^1$; and

(b) no other string in $L^{c\#}(S_2)$ satisfies (a) and agrees with x on a longer final segment.

Such a string y exists since $x^1 =$ val (S_1, H) for some Herbrand interpretation for which S_1 halts and $S_1 \equiv S_2$ by assumption.

Let k $(1 \leqq k \leqq \min(n, m))$ be the least positive integer such that $x_k \cdots x_1 \neq y_k \cdots y_1$. Let α be the string that results from $x_{k-1} \cdots x_1$ by deleting all predicate tests (α can be the empty string). Since $x^1 = y^1$ and S_1 and S_2 are strong free svp schemes, both of x_k and y_k must be predicate tests. Suppose the test in x_k is P_i and the test in y_k is P_j. By assumption $P_i \neq P_j$. Let the corresponding statements in S_1 and S_2 be

$$L_{i_0} \cdot \textbf{If}\ P_i(x)\ \textbf{then}\ L_1\ \textbf{else}\ L_2 \quad \text{and}$$

$$L_{k_0} \cdot \textbf{If}\ P_j(x)\ \textbf{then}\ L_1^1\ \textbf{else}\ L_2^1,$$

respectively. Then L_1 cannot be equivalent to both of L_1^1 and L_2^1, otherwise the occurrence of P_i in statement L_{i_0} is superfluous. So suppose that L_1 and L_1^1 are not equivalent. Then there is a Herbrand interpretation H_0 such that

$$\text{val}\,(S_1, H_0, L_1) \neq \text{val}\,(S_2, H_0, L_1^1).$$

Since S_1 and S_2 are free, we can also choose a Herbrand interpretation H_1 such that

$$\text{Comp}\,(S_1, H_1) = \cdots x_k x_{k-1} \cdots x_1,$$

$$\text{Comp}\,(S_2, H_1) = \cdots y_k y_{k-1} \cdots y_1.$$

Let H_2 be any Herbrand interpretation satisfying the following:

(A) For all proper suffixes α' of α and for all predicate symbols P_l

$$(P_l)^{H_2}(\alpha' x) = (P_l)^{H_1}(\alpha' x);$$

(B) $(P_i)^{H_2}(\alpha x) = \text{True}$;

(C) $(P_j)^{H_2}(\alpha x) = \text{True}$; and

(D) For all strings $\alpha' = w'\alpha$ such that α is a proper suffix of α' and for all predicate symbols P_i,

$$(P_i)^{H_2}(\alpha' x) = (P_i)^{H_0}(w' x).$$

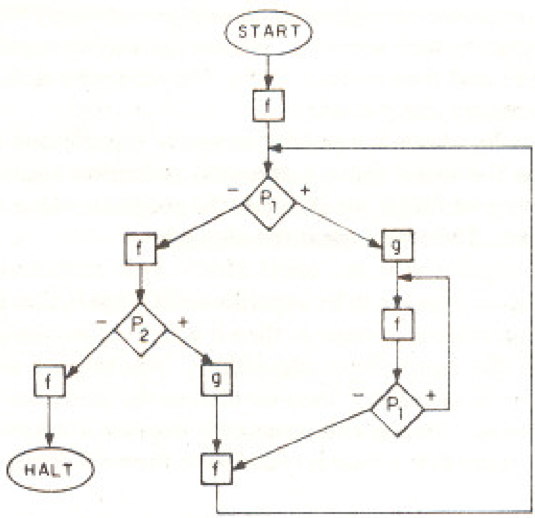
Clearly such Herbrand interpretations exists. For each such Herbrand interpretation H_2,

$$\text{val}\,(S_1, H_2) \neq \text{val}\,(S_2, H_2).$$

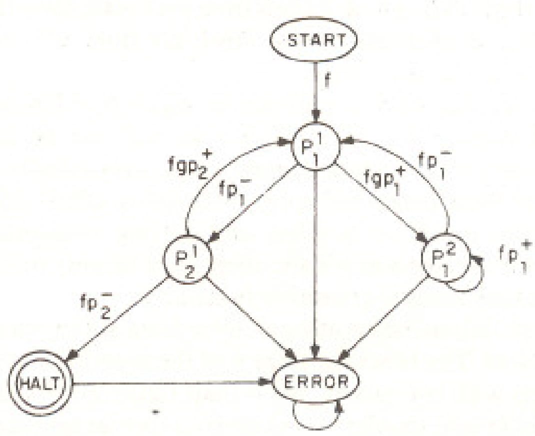
contradicting the assumption that S_1 and S_2 are strongly equivalent. (2) ($\Leftarrow$). This follows immediately from Lemma 3.4.

A reduced strongly free svp scheme S can be viewed as a deterministic finite automaton $A(S)$ accepting $L^{c\#}(S)$. To see how this works, consider the reduced strongly free svp scheme S_0 and its associated deterministic finite automaton $A(S_0)$ shown in Fig. 3.8. The alphabet of $A(S_0)$ is

$$\Sigma = \{f, fp_1^-, fgp_1^+, fp_1^+, fp_2^-, fgp_2^+\};$$



S_0



A(S_0)

FIG. 3.8

and the state set is

$$K = \{\text{start}, P_1^1, P_2^1, P_1^2, \text{halt}, \text{error}\},$$

where P_i^j is the jth occurrence of predicate P_i (in some arbitrary ordering of occurrences.) Finally in the state diagram of $A(S_0)$, we intend that all unlabeled edges be implicitly labeled by those elements of Σ not occurring as labels on outgoing edges.

Clearly the automaton $A(S_0)$ accepts the language $L^{c*}(S_0)$. Thus rephrasing Theorem 3.7, for reduced strongly free svp schemes S_1 and S_2, $S_1 \equiv S_2$, if and only if, the associated deterministic finite automata $A(S_1)$ and $A(S_2)$ are equivalent. Noting that, for a strongly free svp scheme S_1, $A(S_1)$ is constructable from S_1 deterministically in polynomial time and that the equivalence problem for deterministic finite automata is decidable deterministically in polynomial time [6], we have the following.

THEOREM 3.9. *The strong equivalence problem for reduced strongly free svp schemes is decidable deterministically in polynomial time.*

3.2. Reducing strongly free Ianov schemes in deterministic polynomial time. In order to extend the equivalence algorithm to arbitrary strongly free Ianov schemes, we give a method of reducing such schemes. We can not simply regard these schemes S as finite automata $A(S)$ and then reduce $A(S)$. The difficulty is illustrated by a simple example which the reader can provide.

In order to decide whether a predicate test is superfluous we need to apply an algorithm similar to the usual finite automaton reduction technique. We search for nonredundancy. When we find it, we attached the predicate value P_i^+ or P_i^- to the edges leading from the state. Then we repeat the algorithm.

Informally the algorithm is the usual Moore type reduction algorithm on $A(S)$ except that if a predicate appears to be superfluous at stage n, that is, both branches lead to states which are equivalent at stage n, then it is treated as superfluous (the predicate label is not used in the equivalence algorithm). Whenever a suspected superfluous predicate turns out to be necessary, then we restore the predicate label and recompute the equivalence relation. This algorithm succeeds because if it is possible to reduce $A(S)$ and assume at every stage that a state is redundant, then it is really redundant (we prove this in Theorem 3.12).

Before we can describe the reduction algorithm we need a number of conventions. First, given scheme S and its associated automaton $A(S)$ we associate with each state the predicate P_i of S corresponding to it. Labels from each state have the form yP_i^+, yP_i^- for $x, y \in \{f_j\}^*$. To *remove a predicate from a label*, say from xP_i^+ or yP_i^-, means to replace these labels by x or y respectively.

In the reduction algorithm we will consider various sets of labels for the edges of the state diagram. At stage n of the algorithm we will use an alphabet denoted $\Sigma^n := \{a_1^n, \cdots, a_{p_n}^n\}$. For any state in the automaton $A(S)$ associated with a strongly free scheme S, at most two of these labels will apply (will lead to other than an error state). Call these letters 0_s (the predicate is false) and 1_s (the predicate is true). After predicates are removed from labels at a state, there may be only one label remaining. This gives rise to a nondeterministic transition function δ.

As in the Moore type minimization algorithm for finite automata (see [5, 6, 7]), we will group states into *blocks*. The blocks at stage n of the algorithm will be denoted B_i^n.

The algorithm starts with two blocks, $B_1^0 := \{\text{halt state}\}$, $B_2^0 := \{\text{all nonhalt states}\}$, and proceeds to split blocks into smaller blocks until no further splitting is possible. It is possible to split a block B_i^n as long as condition ** given below holds:

** $\exists a \in \Sigma^n \exists s_1, s_2 \in B_i^n$ such that
$\delta(s_1, a) \in B_j^n$ and $\delta(s_2, a) \notin B_j^n$; for δ the transition function of $A(S)$.

That is, there are two states in a block which we can recognize as distinct (inequivalent) by one of δ's transition on the label a.

The informal algorithm is this.

REDUCTION ALGORITHM 3.10. Start with Σ, $A(S)$. Form Σ^0 as the set of labels with predicates removed and $A^0(S)$ as the automaton with predicates removed from labels (but written on the states). Let B_1^0 contain the half state and B_2^0 all non-halt states. Let N be the *stage* number, initially $N=0$. Let δ_0 be the (nondeterministic) transition function arising from the δ of $A(S)$.

BEGIN REDUCTION ALGORITHM

initialize (set $N=0$, set up B_1^0, B_2^0).

while ** **do**

begin (1) compute the output behavior of each state under Σ^N (at stage N) using *each* possible transition of δ_N.

(2) locate the non-redundant states at stage N, i.e. $\delta_N(s, a) \in B_i^N$ and $\delta_N(s, b) \in B_j^N$, $i \neq j$ (possibly $a = b$).

(3) form a new set of labels, Σ^{N-1}, by restoring the predicates to the labels on the outgoing edges of non-redundant states located in step (2). The new automaton diagram is denoted $A^{N+1}(S)$.

(4) recompute the output behavior using Σ^{N+1}, δ_{N+1}.

(5) split blocks B_i^N to form blocks B^{N+1} by grouping only those states of B_i^N which have the same output behaviour as computed in (4).

end

Redundant states are those whose outgoing edges do not have predicates restored to their labels.

END REDUCTION ALGORITHM.

Given the reduced automaton, say $\hat{A}(S)$, we can construct from it a scheme $\hat{S}$ having no redundant predicates. We remove each redundant state, say

$$\text{L: } \textbf{if } P_i \textbf{ then } L_l \textbf{ else } L_r$$

and connect all incoming edges to L_l (that is, replace any **goto** L by **goto** L_l).

Combining this algorithm with the reduction algorithm, we have an algorithm for transforming strongly free Ianov schemes S to reduced strongly free Ianov schemes $\hat{S}$ (we prove this below). The application of Algorithm 3.10 is illustrated in Figs. 3.11a, 3.11b, and 3.11c.

Analysis of runtime. It is easy to see that the Reduction Algorithm is in the worst case bounded above by $O(|\Sigma| \cdot |K|^2)$. Consider the time for each step, the bounds are

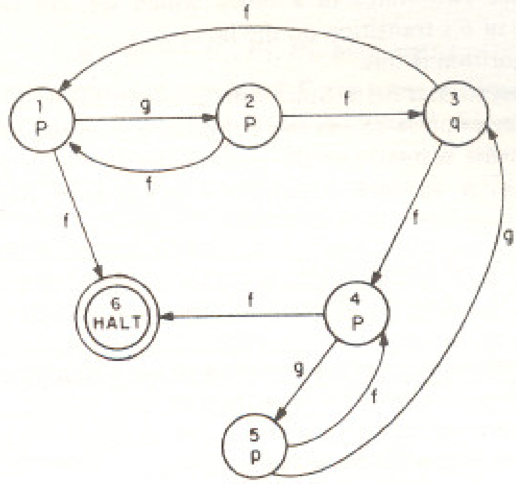
$$(1)\ \leqq |\Sigma| \cdot |K| \quad (2)\ \leqq |K| \quad (3)\ \leqq |\Sigma| \quad (4)\ \leqq |\Sigma| \cdot |K| \quad (5)\ \leqq |K|.$$

So the worst case occurs when at most one state is split off of a block on each iteration. Thus the worst case is

$$O(|K| \cdot [2 \cdot |\Sigma| \cdot |K| + 2 \cdot |K| + |\Sigma|]).$$

If we use a more efficient algorithm, such as Hopcroft [6] (also see Gries [5]), then the time is $O(|\Sigma| \cdot |K| \cdot \log(|K|))$.

In any case this is a polynomial time bounded algorithm in either $|K|$, $|\Sigma|$ or in $|S|$. We now summarize our knowledge in a theorem.



$A^0(S)$

B_1^0	B_2^0
{6}	{1, 2, 3, 4, 5}

The B_i^1 blocks are for $i = 1, 2, 3, 4, 5$:

FIG. 3.11A

B_1^1	B_2^1	B_3^1	B_4^1	B_5^1
{6}	{1}	{2, 3}	{4}	{5}

FIG. 3.11B

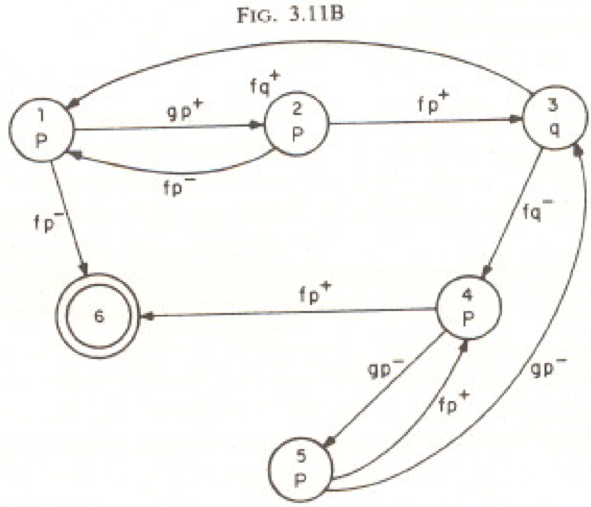


FIG. 3.11C

3.2.1. Correctness of the algorithms.

THEOREM 3.12. *There is an algorithm whose runtime is no more than a polynomial in $|S|$ which produces the reduced scheme $\hat{S}$ given S. That is,*

(i) *$S \equiv \hat{S}$ and*

(ii) *S contains no redundant predicates.*

Proof. The time analysis given above shows that the algorithm is polynomial in $|S|$. We need only show (i) and (ii). We first consider (i).

(1) Clearly if a predicate P_i remains in S then it is not redundant because the algorithm produces an interpretation under which the true and false branches from P_i are distinct. So we need only show that if a predicate occurrence is removed, say at state s as

$$s: \textbf{if } P_i \textbf{ then } L_i \textbf{ else } L_j,$$

then that occurrence is really redundant. To prove this, suppose some predicate occurrences were erroneously removed, say P_{i_1} at state $s_{i_1}, \cdots, P_{i_n}$ at states s_{i_n}. Then order these by the length of the interpretation under which the true and false branches are distinct. Suppose P_i is one with the least length interpretation. Then that interpretation cannot involve another predicate erroneously removed in an essential way. That is, either the two computations, the true one which is $x_n x_{n-1} \cdots x_1$ or the false one, $y_m y_{m-1} \cdots y_1$ either (a) do not contain any P_{i_j} (erroneously removed predicates) or (b) if such a P_{i_j} does occur, then the true branch from it to the halt state (x_n or y_m) is the same as the false branch, because otherwise this P_{i_j} would have a shorter interpretation showing it to be nonredundant than P_i does, contradicting the definition of P_i. Thus in either case (a) or (b), the computations $y_m \cdots y_1$ and $x_n \cdots x_1$ appear already in some $A^k(S)$. That is, neither computation requires the presence of an erroneously classified predicate. Therefore, P_i would be discovered to be nonredundant at some state k of the reduction algorithm.

(2) Finally, to show $S \equiv \hat{S}$ we notice that S and $\hat{S}$ are nearly isomorphic. For every state s of S there is a corresponding state $\hat{s}$ of $\hat{S}$ unless s is redundant. But if s is redundant, then we know that the edges in S which by-pass s do not change equivalence. The reader can prove this by carefully considering these "near isomorphisms" under any Herbrand interpretation H.

We now state a fact about finite automata.

Fact 3.13. *There is an $O(|\Sigma| \cdot n \log(n))$ time algorithm to decide the equivalence of finite automata S_1, S_2 over Σ where $n = \max(|K_1|, |K_2|)$, K_1, K_2 the state sets of S_1, S_2.*

Using this we have the theorem we need.

THEOREM 3.14. *There is a polynomial time bounded algorithm to decide the equivalence of strongly free Ianov schemes.*

3.2.2. Extension to predicate clusters. We now want to mention an extension of the reduction and equivalence algorithms from strongly free Ianov schemes to strongly free Ianov schemes with tree-like predicate clusters substituted for predicates. The idea is to replace any tree-like cluster of predicates by a single multi-exit predicate.

Let S be a Ianov scheme, then a *cluster of predicates* in S is a loop free subscheme of S containing no function applications and such that no edge can be extended without including a function application. A *tree-like cluster* is such that the cluster is a tree whose nodes are predicates.

Notice that in a free scheme no predicate can occur more than once on a path from root to leaf in a cluster, but predicates may indeed occur more than once.

We represent these clusters by multi-exit predicates and can make this assignment of multi-exit predicates to clusters uniform if we choose a specified ordering of predicates. For example, suppose we have P, Q, R, T. We then label all outputs in the order P, Q, R, T.

To decide equivalence of free Ianov schemes S1, S2 we convert the predicate clusters to multi-exit predicates and then convert the result to a finite automaton, $A(S)$, with labels on predicates given in a standard order. Even in the case of free Ianov schemes, the generation of multi-exit predicates may require exponential time.

If all the predicate clusters in a Ianov scheme are tree-like, then the multi-exit predicate has the same number of exits as there are leaves in the tree, thus it can be generated in polynomial time (in the number of edges) given the cluster.

We can use essentially the same type of reduction algorithm as before, but we must be careful to say exactly when a predicate in a cluster is redundant on the basis of information gathered about the multi-exit predicate in $A(S)$.

During the reduction algorithm, the edges leaving a multi-exit predicate can be grouped together into edge-groups, $E_i^N(s)$; that is a stage N there may be $i = 1, \cdots, m$ edge-groups associated with state s. We say that a predicate occurrence Q in a cluster C *is redundant with respect to the edge-groups* E_i^N if for all sequences of predicate tests z_1 such that $z_1 Q^+ y \in E_i^N$ there is a sequence z_2 compatible with z_1 (no predicate P appears as P^+ in z_1 and P^- in z_2 or vice versa) such that $z_2 Q^- y \in E_i^N$. That is, Q does not affect the decisions made by predicates tested after Q. For example, let the edge-groups be labeled A, B, C and consider the tree-like predicate cluster and the equivalent multi-exit predicate in Figs. 3.15a and 3.15b. The sequences in the edge-groups are

A	B	C
P^+Q^+	P^+Q^-	P^-Q^+
		P^-Q^-

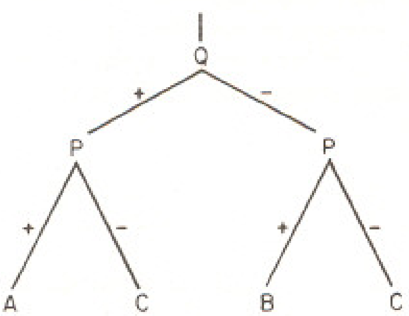


FIG. 3.15A

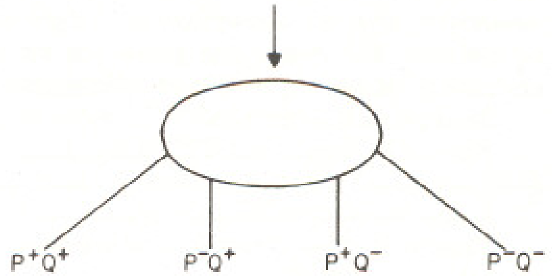


FIG. 3.15B

Thus, the predicate Q is redundant with respect to edge-group C. (Therefore in the reduction algorithm the labels P^-Q^+ and P^-Q^- are replaced by P^-.)

This example suggests how inefficient a predicate cluster might be. But we do not need to consider methods of finding the minimum cluster equivalent to a given cluster in order to obtain a polynomial equivalence algorithm. We only need a method of eliminating the redundant predicates from the labels on outgoing edges of multi-exit predicates.

This example is too simple to illustrate the difficulties in testing for redundant predicate occurrences. It is not sufficient to see whether xQ^+y and xQ^-y both appear in an edge group. For example, consider the tree-like predicate cluster in Figure 3.16. Then in edge-group B we have P^-Q^+y, $S^-R^+Q^-y$, R^-Q^-y. So Q is redundant for B because both S^-R^+ and R^- are compatible with P^-.

In order to mimic the reduction algorithm for strongly free schemes, we need a procedure to check for redundancy in predicate clusters given as assignment of edge-groups (this assignment comes from the main algorithm).

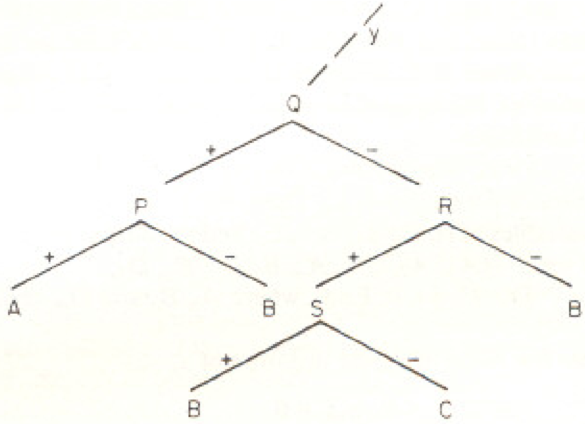


FIG. 3.16

3.2.3. Multi-exit nonredundancy procedure. Given predicate cluster C and edge-groups $E_1, \cdots, E_m$, to test whether a predicate occurrence Q in a label on an edge in E_i^N is redundant, do the following:

begin

(1) locate Q in the cluster (let y be the path to Q).
(2) list all prefixes of the form z where zQ^-y is in E_i^N.
(3) for each z in (2) check whether there is a prefix w
where (a) wQ^-y is in E_i^N
(b) w and z are compatible.
(4) if there is a w for each z, then Q is redundant, otherwise it is not and the predicate is output.

end

The reader can now check the validity of the following claims.

PROPOSITION 3.17. *A predicate occurrence Q in tree-like cluster C is nonredundant with respect to edge group E_i if the multi-exit redundancy procedure generates Q given C and E_i.*

It is also easy to check that this procedure runs in polynomial time in the number of predicates in the cluster.

PROPOSITION 3.18. *If tree-like predicate cluster C has n predicates, then the multi-exit redundancy procedure runs in at most n^2 steps.*

4. Simple programs. We conclude by showing that the equivalence problems for several very elementary programming languages are also NP-complete. First, we consider the Loop 1 languages in [12], [14].

DEFINITION 4.1. A *loop program* is a finite sequence of instructions of the five types: (1) Do x, (2) End, (3) $x \leftarrow 0$, (4) $x \leftarrow y$, and (5) $x \leftarrow x+1$.

A subset of the variables used in a loop program is designated as the *input variables* of the program. One variable is designated as the *output variable* of the program. Loop programs compute functions of their input variables. We say that two loop programs are *equivalent* if they compute the same function.

DEFINITION 4.2. For all $i = 0, 1, 2 \cdots$, L_i is the class of all loop programs in which the maximum level of nesting of Do statements equals i. The set *Inequiv* (L_i) is the set of all pairs of inequivalent L_i programs.

PROPOSITION 4.3. Inequiv (L_1) *is NP-complete.*

Proof. To show that Inequiv (L_1) is NP-hard, it suffices to show that $\mathcal{T}_2$, the set of satisfiable C_3-Boolean forms, is p-reducible to it. We show how, for each C_3-Boolean form f, to efficiently construct an L_1 program Π_f such that, for all assignments of initial values to its input variables, the value of its output variable upon termination equals 0, if and only if, f is not satisfiable.

Let $f = c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i = c_{i1} \vee c_{i2} \vee c_{i3}$ and each c_{ij} is a literal. Let the propositional variables of f be $x_1, \cdots, x_n$. Then, the L_1 program Π_f is constructed as follows. The input variables of Π_f are $x_1, \cdots, x_n$; and the output variable of Π_f is z. The program Π_f has the form—$(A_1, A_2, \cdots, A_n, B_1, \cdots B_k, D_1, \cdots, D_k, z \leftarrow 0, z \leftarrow z+1,$ Do C_1', $z \leftarrow 0$, End, $\cdots$ Do, C_k $z \leftarrow 0$, End), where A_j, B_i, and D_m are program blocks defined as follows:

(1) A_j computes the value of $\bar{x}_j$, the negation of x_j

$$
\begin{aligned}
A_j \text{ is } & \bar{x}_j \leftarrow 0\\
& \bar{x}_j \leftarrow \bar{x}_j + 1\\
& \text{Do } x_j\\
& \bar{x}_j \leftarrow 0\\
& \text{End.}
\end{aligned}
$$

(2) B_i computes the value of the clause c_i, for any given values of $x_1, \cdots, x_n, \bar{x}_1, \cdots, \bar{x}_n$. We illustrate the construction of B_i by an example. Suppose c_i is $x_1 \vee \bar{x}_2 \vee \bar{x}_3$, then

$$
\begin{aligned}
B_i \text{ is } & c_1 \leftarrow 0\\
& \text{Do } x_1\\
& c_1 \leftarrow 0\\
& c_1 \leftarrow c_1 + 1\\
& \text{End}\\
& \text{Do } \bar{x}_2\\
& c_1 \leftarrow 0\\
& c_1 \leftarrow c_1 + 1\\
& \text{End}\\
& \text{Do } x_3\\
& c_1 \leftarrow 0\\
& c_1 \leftarrow c_1 + 1\\
& \text{End.}
\end{aligned}
$$

(3) D_m computes the value of $\bar{c}_m$, the negation of c_m.

$$
\begin{aligned}
D_m \text{ is } & \bar{c}_m \leftarrow 0\\
& \bar{c}_m \leftarrow \bar{c}_m + 1\\
& \text{Do } c_m\\
& \bar{c}_m \leftarrow 0\\
& \text{End.}
\end{aligned}
$$

We leave it to the reader to verify that the program Π_f outputs a 1 for some assignment of values to its input variables, if and only if, f is satisfiable. Otherwise, Π_f always outputs 0.

Finally, the fact that Inequiv (L_1) is in NP follows immediately from Theorems 4 and 7 in [14]. Q.E.D.

We note that the equivalence problem for L_0 programs can be solved deterministically in linear time, and that the equivalence problem for L_2 programs is undecidable [12].

DEFINITION 4.4. Let x and y be nonnegative integers. Then,

$$x \dot{-} y = \begin{cases} x - y, & \text{if } x \geqq y, \\ 0, & \text{otherwise}. \end{cases}$$

DEFINITION 4.5. $\mathscr{P}_1$ is the class of all programs consisting of a finite sequence of instructions of the form (1) $x_i \leftarrow x_j + x_k$, and (2) $x_i \leftarrow 1 \dot{-} x_j$, where x_j and x_k may be nonnegative integer constants. $\mathscr{P}_2$ is the class of all programs consisting of a finite sequence of instructions of the forms $x_i \leftarrow x_j \dot{-} x_k$, where x_j and x_k may be non-negative integer constants.

$\mathscr{P}_1$ and $\mathscr{P}_2$ programs compute functions in the obvious manner. We say that two $\mathscr{P}_1$ or $\mathscr{P}_2$ programs are *equivalent* if they compute the same function. *Inequiv* ($\mathscr{P}$) (*Inequiv* ($\mathscr{P}_2$)) is the set of all pairs of inequivalent $\mathscr{P}_1(\mathscr{P}_2)$ programs.

PROPOSITION 4.6. Inequiv ($\mathscr{P}_1$) *and* Inequiv ($\mathscr{P}_2$) *are NP-complete.*

Proof. (1) To show that Inequiv ($\mathscr{P}_1$) is *NP*-hard, it suffices to show that $\mathscr{T}_2$ is p-reducible to it. This is accomplished by simulating the construction in the proof of Proposition 4.3.

Let $f = c_1 \wedge c_2 \wedge \cdots \wedge c_k$, where $c_i = c_{i1} \vee c_{i2} \vee c_{i3}$ and each c_{ij} is a literal. Let the propositional variables of f be $x_1, \cdots, x_n$. The $\mathscr{P}_1$ program Π_f has n input variables $x_1, \cdots, x_n$, output variable $\mathscr{P}$, and has the form—$(A_1, \cdots, A_n, B_1, \cdots, B_k, D_1, \cdots, D_k, P \leftarrow 1 \dot{-} \bar{c}_1, \cdots, P \leftarrow 1 \dot{-} \bar{c}_k)$. Here, A_1, B_j and D_m are—

(a) A_i is $\bar{x}_i \leftarrow 1 \dot{-} x_i$,

(b) letting $c_j = x_1 \vee \bar{x}_2 \vee x_3$, B_j is $c_j \leftarrow x_1 + \bar{x}_2$, $c_j \leftarrow c_j + x_3$, and

(c) D_m is $\bar{c}_m \leftarrow 1 \dot{-} c_m$

To show that Inequiv ($\mathscr{P}_1$) is in *NP*, it suffices to note that there is a deterministic polynomially time-bounded Turing machine M such that M, given a $\mathscr{P}_1$ program Π as input, outputs an equivalent L_1 program Π'. Thus, since Inequiv (L_1) is in *NP*, so is Inequiv ($\mathscr{P}_1$).

M operates as follows. It replaces all instruction of the form $x_i \leftarrow 1 - x_j$ in Π by the program fragment

$$\begin{array}{l} x_i \leftarrow 0 \\ x_i \leftarrow x_i + 1 \\ \text{Loop } x_j \\ x_i \leftarrow 0 \\ \text{End} \end{array}$$

It replaces all instructions of the form $x_i \leftarrow x_j + x_k$ in Π by the L_1 program fragment $x_i \leftarrow x_j$

$$\begin{array}{l} \text{Loop } x_k \\ x_i \leftarrow x_i + 1 \\ \text{End.} \end{array}$$

(2) The proof that Inequiv ($\mathscr{P}_2$) is *NP*-complete is similar and will not be presented here. Details can be found in [2].

REFERENCES

[1] E. ASHCROFT, Z. MANNA AND A. PNEULI, *Decidable properties of monadic functional schemes*, J. Assoc. Comput. Mach., 20 (1973), pp. 489–499.

[2] R. L. CONSTABLE, H. B. HUNT III, AND S. SAHNI, *On the computational complexity of scheme equivalence*, TR-74-201, Department of Computer Science, Cornell University, Ithaca, NY, 1974.

[3] S. A. COOK, *The complexity of theorem proving procedures*, Proceedings Third Annual ACM Symposium on Theory of Computing, May 1971, pp. 151–158.

[4] S. J. GARLAND AND D. C. LUCKHAM, *Program schemes, recursion schemes, and formal languages*, J. Comput. System Sci. 7 (1973), pp. 119–160.

[5] D. GRIES, *Describing an algorithm by Hopcroft*, Acta Informat. 2 (1973), pp. 97–109.

[6] J. E. HOPCROFT, *An nlogn algorithm for minimizing states in a finite automaton*, Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, 1971, pp. 189–196.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[8] I. IANOV, *On logical algorithm schemata*, Cybernetics Problems I, 1958.

[9] D. M. KAPLAN, *Regular expressions and the equivalence of programs*, JCSS, 4 (1969), pp. 361–386.

[10] D. C. LUCKHAM, D. M. R. PARK, AND M. S. PATERSON, *On formalized computer programs*, Ibid., 4 (1969), pp. 220–249.

[11] Z. MANNA, *Program schemas*, Currents in the Theory of Computing, A. V. Aho, ed., Prentice-Hall, Englewood Cliffs, NJ, 1973, pp. 90–142.

[12] A. R. MEYER AND D. RITCHIE, *Computational complexity and program structure*, IBM research paper, May 15, 1967.

[13] M. PATERSON, *Equivalence problems in a model of computation*, MIT A.I. Laboratory Technical Memo. No. 1, Massachusetts Institute of Technology, Cambridge, Ma, 1970.

[14] D. TSICHRITZIS, *The equivalence problem of simple programs*, J. Assoc. Comput. Mach., 17 (1970), pp. 729–738.