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 \textit{NP}-complete. The equivalence problem for the Loop 1 programming language is also shown to be \textit{NP}-complete. Sufficient conditions for a program scheme problem to be \textit{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, \textit{P}, \textit{NP}, \textit{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 \textit{NP}-complete.

The remainder of this paper is divided into four sections. Section 1 contains definitions and basic properties of \textit{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 \textit{NP}-complete. We also present general sufficient conditions for a problem on the single variable program schemes to be \textit{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 \textit{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 \(\lambda\).

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

\textbf{Definition 1.2.} Let \(\Sigma\) and \(\Delta\) be finite alphabets. Let \(\mathcal{F}(\Sigma, \Delta)\) denote the set of all functions from \(\Sigma^*\) into \(\Delta^*\) computable by some deterministic polynomially time-bounded Turing machine. Let \(L_1\) and \(L_2\) be subsets of \(\Sigma^*\) and \(\Delta^*\), 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.

\(^\dagger\) 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 \( L_1 \leq_{p \text{-reduce}} L_2 \), 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 \( L \in \text{NP} \), \( L \leq_{p \text{ - reduce}} L_0 \). 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 \( C_1, \ldots, C_r \) 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 \( C_1, \ldots, C_r \) such that each clause \( C_i \) is the disjunction of at most three literals.

**Proposition 1.5.** The sets \( \mathcal{T}_1 = \{ f \mid f \text{ is a nonautological } D_3 \text{-Boolean form} \} \) and \( \mathcal{T}_2 = \{ f \mid f \text{ is a satisfiable } C_3 \text{-Boolean form} \} \) are NP-complete.

**Proposition 1.6.** 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 \( \mathcal{L}, \mathcal{Y}, \mathcal{F}, \) 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 \( k. \; y := f(x_1, \ldots, x_n) \), where \( k \) in \( \mathcal{L} \) is a label, \( f \) in \( \mathcal{F} \) is an \( n \)-ary function symbol, and \( x_1, \ldots, x_n, y \) in \( \mathcal{Y} \) are variable symbols;
2. **conditional statements** of the form \( \text{if } P(x_1, \ldots, x_n) \text{ then } k_1 \text{ else } k_2 \), where \( k_1, k_2 \), and \( P \) are labels, \( P \) in \( \mathcal{P} \) is an \( n \)-ary predicate symbol, and \( x_1, \ldots, x_n \) in \( \mathcal{Y} \) are variable symbols; and
3. **halt statements** of the form \( k \). \text{Halt} \), where \( k \) is a label.

We sometimes allow **loop statements** of the form \( k. \text{Loop as } \), as abbreviations for the statement

\[
k. \text{ if } P(x_1, \ldots, x_n) \text{ then } k \text{ else } k.
\]

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 \( \mathcal{Y} \);
3. an assignment of a function \( f^n : D^n \rightarrow D \) to every \( n \)-ary function symbol \( f \) in \( \mathcal{F} \); and
4. an assignment of a predicate \( P^n : D^n \rightarrow \text{True or False} \) to every \( n \)-ary predicate symbol \( P \) 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

\[
F_{\alpha}x := \text{if } P_\alpha x \text{ then } \alpha x \text{ else } \beta x,
\]

\[
F_{\beta}x := \text{if } P_\beta x \text{ then } \alpha x \text{ else } \beta x.
\]

\(^1\) Definition 1.3 extends the concept of NP-completeness to languages over arbitrary finite alphabets.
where \( F_1, \ldots, F_n \) are defined function symbols; \( P_1, \ldots, P_m \) are (not necessarily distinct) predicate symbols; and \( \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m \) 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 \( \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m \).

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 \( f^i : D \rightarrow D \) to every basis function symbol \( f \) is \( S \);
3. an assignment of a predicate \( P_i : D \rightarrow \{\text{True}, \text{False}\} \) 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 \( [\mathcal{F}] \times \{x\} \); and
ii. for all \( f \) in \( \mathcal{F} \) and strings \( wx \) in \( [\mathcal{F}] \times \{x\} \), \( f^i(wx) \) equals the string \( f^i x \).

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

\[
S.F_{\alpha} := \text{If } P_{\alpha} \text{ then } \alpha \text{ else } \beta \quad (1 \leq i \leq n),
\]

we associate a context-free grammar \( G_S \) with terminal alphabet equal to \( \mathcal{F} \), nonterminal alphabet \( \mathcal{F} \) equal to \( \{F_1, \ldots, F_n\} \), and set of productions equal to \( \{F_1 \rightarrow \alpha_1, F_i \rightarrow \beta_i, 1 \leq i \leq n\} \). 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 derivation of the form \( \gamma F w \Rightarrow \delta w \), where \( \gamma, \delta \in (\mathcal{F} \cup F)^* \) and \( w \in F^* \), \( \delta \in \alpha_i \) if \( P_i(w) = \text{True} \) and \( \delta = \beta_i \) if \( P_i(w) = \text{False} \). The computation of \( S \) under \( I \) corresponds to the unique legal derivation for \( I \). If \( F_i \Rightarrow^* w \)

for \( w \in F^* \) by the legal derivation for \( I \), then \( \text{val}_I(S) = w(x^i) \); otherwise, \( \text{val}_I(S) \) is undefined.

Finally we assume that there is a finite alphabet \( \Sigma \) such that each scheme or program \( S \) is presented as a string \( \sigma_S \) over \( \Sigma \). We say that the length of the string \( \sigma_S \) 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 \( \text{val}_I(S') \).

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

Then, the relation \( \rho \) 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 lm 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 lm 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 labeled \( B \) 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 \( x_1 \overline{x}_3 x_4 \lor \overline{x}_2 x_3 x_4 \lor x_1 x_5 x_6 \). 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 \( P_3(x) \) then \( B \) else \( A \)
A. Halt
B. Halt

We denote the set \( \{ S_f | f \text{ is a } D_3 \text{-Boolean form} \} \); and the Turing machine \( M_0 \) of Proposition 2.2, given input \( f \), outputs \( S_f \) by \( \mathcal{O} \).

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

For example, let \( S, \mathcal{A}, \) and \( \mathcal{B} \) be the following:

\( S: \ 1. \ \text{If } P_1(x) \ \text{then } 2 \ \text{else } 3 \ \ \ \mathcal{A}: \ 1. \ x \leftarrow f(x) \)
2. \( x \leftarrow x \)
A. Halt
3. \( x \leftarrow x \)
B. Halt
\( \mathcal{B}: \ 1. \ x \leftarrow g(x) \)
2. Halt
Then, $[S, A, B]$ is the following:

1. \( \text{if } P_1(x) \text{ then } y \text{ else } z \) 
2. \( x \leftarrow x \)  
3. \( x \leftarrow f(x) \)  
4. \( \text{Halt} \)  
5. \( x \leftarrow g(x) \)  
6. \( \text{Halt} \)  

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

**Theorem 2.4.** Let \( \Pi \) 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 \( \mathcal{C} \), \( \Pi([S, A, B]) \) equals False if and only if the statement of \( S \) labelled \( B \) is executed during some computation of \( S \). Then, the set \( \{S \mid \Pi(S) \text{ is an svp scheme and } \Pi(S) \text{ equals False} \} \) is NP-hard. Moreover, if \( \mathcal{A} \) and \( \mathcal{B} \) are loop-free, then the set \( \{S \mid \Pi(S) \text{ is a loop-free svp scheme and } \Pi(S) \text{ equals False} \} \) is also NP-hard.

**Proof.** By Propositions 1.5 and 1.6, it suffices to show that the set \( \mathcal{F} \) of non-Boolean forms is \( \Pi \)-reducible to the set \( \{S \mid \Pi(S) \text{ equals False} \} \). Let \( f \) be a \( D_2 \)-Boolean form. Let \( S_f \) be the corresponding element of \( \mathcal{C} \). Then, \( \Pi([S_f, A, B]) \) 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 \( \{S_f, A, B\} \) 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 \( \mathcal{F} \).

**Corollary 2.5.** Let \( \Pi \) be any of the following predicates on the svp schemes:

(i) \( S \) diverges;

(ii) \( S \) halts;

(iii) \( S \) is strongly equivalent to \( \mathcal{F} \);

(iv) \( S \) contains \( \mathcal{F} \);

(v) \( \mathcal{F} \) contains \( S \);

(vi) \( S \) is weakly equivalent to \( \mathcal{F} \); and

(vii) for all reasonable relations \( p \) on the svp schemes, \( Sp\).

Then, the set \( \{S \mid \Pi(S) \text{ is an svp scheme and } \Pi(S) \text{ equals False} \} \) 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 (vii) \( \mathcal{A} \) is 1. Halt, \( \mathcal{B} \) is 1. \( x \leftarrow f(x) \)  
2. Halt.

Q.E.D.

**Corollary 2.6.** Let \( p \) be any of the following binary relations on the svp schemes:

for all svp schemes \( S \) and \( S' \), \( SpS' \), 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 \( p \) on the svp schemes, \( SpS' \).

Then, the set \( \{(S, S') \mid S \text{ and } S' \text{ are svp schemes and } \sim(\text{SpS'}) \} \) is NP-hard. Moreover, the set \( \{(S, S') \mid S \text{ and } S' \text{ are loop-free svp schemes and } \sim(\text{SpS'}) \} \) 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 $\mathcal{F}_1$ of nontautological $D_1$-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 $D_1$-Boolean form. Let $S_f$ be the corresponding element of $\mathcal{E}$. Then, letting $\mathcal{R}_0$ denote the scheme

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

the schemes $[S_f, \mathcal{R}_0]$ and $[S_f, \mathcal{R}_0']$ are isomorphic, if and only if, the statement in $S_f$ labeled $\mathcal{B}$ is not executed during some computation of $S_f$. By Proposition 2.2 this is true, if and only if, $\mathcal{F}$ is a tautology. Since the schemes $[S_f, \mathcal{R}_0]$ and $[S_f, \mathcal{R}_0']$ 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 $\text{NP}$-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 $\text{lmr}$ schemes and that several of these $\text{NP}$-hard problems are $\text{NP}$-complete.

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

$$F_{1,x} := \begin{cases} \text{if } P_{1,x} \text{ then } x \text{ else } x, \end{cases}$$

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

**Corollary 2.7 (1).** Let $\Pi$ be any of the following predicates on the $\text{lmr}$ schemes:

(i) $S$ halts;
(ii) $S$ diverges;
(iii) $S$ is strongly equivalent to $\mathcal{F}'$;
(iv) $S$ contains $\mathcal{F}'$;
(v) $\mathcal{F}'$ contains $S$;
(vi) $S$ is weakly equivalent to $\mathcal{F}'$; and
(vii) for all reasonable relations $\rho$ on the $\text{lmr}$ schemes, $S \rho \mathcal{F}'$. Then, the set $\{S|S$ is an $\text{lmr}$ scheme and $\Pi(S)$ equals False is $\text{NP}$-hard.

(2) Let $\rho$ be any of the following binary relations on the $\text{lmr}$ schemes: for all $\text{lmr}$ 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 $\rho_0$ on the $\text{lmr}$ schemes, $S \rho_0 S'$. Then, the set $\{S, S'|S$ and $S'$ are $\text{lmr}$ schemes and $\neg(\mathcal{F}_{s \rho S'})\}$ is $\text{NP}$-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 $\text{lmr}$ and $\text{svp}$ schemes. The first proposition is new. The second closely follows results in [4].
**Proposition 2.8.** Let ℛ be an lnr 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 there defining equation, say

\[ F_{k} \vdash_{K} \text{If } P_{k} \text{ then } \alpha_{k} \text{ else } \beta_{k}, \]

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 \( P_{k} \) takes the same value. Thus letting \( G_{k} \) 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

\[ F_{1} \vdash_{G_{k}} b_{1}F_{c_{1}}, \]

\[ F_{2} \vdash_{G_{k}} b_{2}F_{c_{2}}, \]

\[ P_{i}^{1}(c_{1}, x) = P_{i}^{1}(c_{2}, x) \]

for the legal derivation for I.

If \( c_{2} \) equals \( \lambda \), then the computation of ℛ under I diverges. Otherwise, let \( I_{0} \) be the free interpretation of ℛ defined by: for all predicate symbols \( P_{k} \) in ℛ,

\[ P_{i}^{1}(w, x) = \begin{cases} P_{i}^{1}(w, x), & \text{if } w = \alpha c_{2} \text{ and } \alpha \text{ is a suffix of } c_{2}; \\ P_{i}^{1}(w, x), & \text{otherwise.} \end{cases} \]

Then, the computation of ℛ under \( I_{0} \) diverges.

**Proposition 2.9.** Let ℛ and ™ be two lnr schemes, with set of basis function symbols \( \mathcal{F} \) and set of defined function symbols \( \mathcal{F} \) such that

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

(ii) the length of each string \( \alpha_{i} \) and \( \beta_{i} \) in a defining equation of ℛ or ™ is less than \( m \); and

(iii) each string \( \alpha_{i} \) and \( \beta_{i} \) in a defining equation of ℛ or ™ is an element of

\[ \mathcal{F}^{*} \cdot \mathcal{F} \cdot (\mathcal{F} \cup \{\lambda\}) \cup \emptyset \cup \{\lambda\}. \]

Then, (1) if there exists an interpretation \( I \) under which ℛ and ™ differ but for which both of \( \text{val}_{1}(ℛ) \) and \( \text{val}_{1}(™) \) are defined, then there is a free interpretation \( I_{0} \), under which ℛ and ™ differ and for which both of \( \text{val}_{1}(ℛ) \) and \( \text{val}_{1}(™) \) are defined, such that the minimum of the lengths of \( \text{val}_{1}(ℛ) \) and \( \text{val}_{1}(™) \) is less than \( 3n^{3}m \). Similarly, (2) if there exists an interpretation \( I \) for which \( \text{val}_{1}(ℛ) \) is defined and \( \text{val}_{1}(™) \) is not, then there is a free interpretation \( I_{0} \), for which \( \text{val}_{1}(ℛ) \) is defined and \( \text{val}_{1}(™) \) is not, such that the length of \( \text{val}_{1}(ℛ) \) is also less than \( 3n^{3}m \).

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

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

(i) \( S_{1} = \{ ⟨ ℛ, ™ \rangle | ℛ \text{ is an lnr scheme; and } ™ \text{ does not halt} \} \);

(ii) \( S_{2} = \{ ⟨ ℛ, ™ \rangle | ℛ \text{ is an lnr scheme; and } ℛ \text{ does not diverge} \} \);

(iii) \( S_{3} = \{ ⟨ ℛ, ™ \rangle | ℛ \text{ and } ™ \text{ are lnr schemes; and } ℛ \text{ and } ™ \text{ are not strongly equivalent} \} \);

(iv) \( S_{4} = \{ ⟨ ℛ, ™ \rangle | ℛ \text{ and } ™ \text{ are lnr schemes; and ℛ and ™ are not weakly equivalent} \} \);

and

(v) \( S_{5} = \{ ⟨ ℛ, ™ \rangle | ℛ \text{ and } ™ \text{ are lnr schemes; and } ℛ \text{ does not contain } ™ \} \).

**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 \( \Pi \) of the context-free grammar \( G_S \) associated with \( S \) of the form
\[
F_1 \rightarrow b_1 F_{i_1} c_{i_1} \rightarrow \cdots \rightarrow b_n F_{i_n} c_{i_n},
\]
where, letting \( n_0 \) be the number of the defining equations of \( S \), \( k = 2n_0 + 1; \)
\( F_1, F_{i_1}, \ldots, F_{i_n} \) are defined functions symbols of \( S \); and \( b_{i_1}, c_{i_1}, \ldots, b_{i_n}, c_{i_n} \)
are strings of basic function symbols of \( S \).
Step 3. \( M \) verifies that \( \Pi \) 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 \( S \). Moreover, \( M \) is polynomially time-bounded.
This follows since
(1) the lengths of each of the sentential forms \( F_1, b_1 F_{i_1} c_{i_1}, \ldots, b_n F_{i_n} c_{i_n} \) in \( \Pi \) is less
than \((2n_0 + 1) \cdot (m + 1) + 1\), where \( m \) is an upper bound on the lengths of the strings \( \alpha, \beta \)
in the defining equations of \( S \); and
(2) \( \Pi \) is legal for some interpretation \( I \), if and only if, for each pair \((b_i, F_{i_1} c_{i_1}, b_i, F_{i_1} c_{i_1})\)
of sentential forms in \( \Pi \), if
\[
b_i F_{i_1} c_{i_1} \rightarrow b_i \delta c_{i_1},
\]
\[
b_i F_{i_1} c_{i_1} \rightarrow b_i \delta' c_{i_1},
\]
\[
c_{i_1} = c_{i_1},
\]
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 \( \Pi \) of \( G_{S_1} \)
\[
F_1 \rightarrow \cdots \rightarrow b_{i_k} F_{i_k} c_{i_k} \rightarrow b_{i_{k+1}} c_{i_{k+1}}
\]
and a rightmost derivation \( \Pi' \) of \( G_{S'_1} \)
\[
F'_1 \rightarrow \cdots \rightarrow b'_{i_k} F'_{i_k} c'_{i_k} \rightarrow b'_{i_{k+1}} c'_{i_{k+1}},
\]
where \( k + 1 \) and \( l + 1 \) are less than \( 3n_3 m_3 \), and \( b_{i_{k+1}}, c_{i_{k+1}} \neq b'_{i_{k+1}}, c'_{i_{k+1}} \).
(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 \( \alpha, \beta \) in any of the defining
equations of \( S_1 \) and \( S'_1 \).)
Step 4. \( M \) verifies that both of \( \Pi \) and \( \Pi' \) are legal for some interpretation. If so, \( M \)
accepts \((S, S')\). Otherwise, \( M \) halts without accepting.
By Proposition 2.9 \( M \) accepts \( S_1 \). 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 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 $\text{val}(S, H)$. We denote the value of a scheme $S$ under interpretation $H$ starting with statement $L_0$ by $\text{val}(S, H, L_0)$. With every svp scheme $S$, we associate the three languages $L(S)$, $L^\times(S)$, and $L^{\times\text{#}}(S)$ 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 $\text{Comp}(S, H)$ is the (possibly infinite) string

$$
\cdots \alpha_m, \cdots \alpha_i P_{m, n}^\times \cdots P_{i, n}^\times \alpha_1
$$

such that each $\alpha_i$ is a (possibly empty) string of function symbols of $S$, $P_i^\times$ is either $P_i^+$ or $P_i^-$ where $P_i$ is a predicate symbol of $S$, and

$P_i^\times$ is $P_i^+$ if and only if, $(P_i)^{\text{if}}(\alpha_1 \cdots \alpha_i) = \text{True}.$

The **computation language** of $S$, denoted by $L^\times(S)$, is the set $\{\text{Comp}(S, H)|H$ is a Herbrand interpretation\}. The **terminating computation language** of $S$, denoted by $L^{\times\text{#}}(S)$, 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 $L^{\times\text{#}}(S_1) = L^{\times\text{#}}(S_2)$, 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, $\text{Comp}(S_1, H) = \text{Comp}(S_2, H)$.

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 $L_1. \text{If } P(x) \text{ then } L_1 \text{ else } L_2$, is said to be superfluous, if and only if, $\text{val} (S, H, L_1) = \text{val} (S, H, L_2)$ 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, $\text{val} (S_1, H, L_1) = \text{val} (S_2, H, L_2)$ 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 $L_1 \equiv (S_1) = L_2 \equiv (S_2)$.

**Proof.** Let $S_1 \equiv S_2$. We show that $L_1 \equiv (S_1) = L_2 \equiv (S_2)$ by proof by contradiction. Suppose $L_1 \equiv (S_1) \neq L_2 \equiv (S_2)$. Let $x = x_1 \cdots x_t$ be a string in one of $L_1 \equiv (S_1)$ and $L_2 \equiv (S_2)$ but not both, say $x \in L_1 \equiv (S_1) - L_2 \equiv (S_2)$. Let $y_1$ be the subsequence of $x$ obtained by deleting all predicate tests. Then, there exists a string $y = y_m \cdots y_1$ in $L_2 \equiv (S_2)$ 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 $L_2 \equiv (S_2)$ satisfies (a) and agrees with $x$ on a longer final segment.

Such a string $y$ exists since $x_1 = \text{val} (S_1, H)$ for some Herbrand interpretation for which $S_1$ halts and $S_1 \equiv S_2$ by assumption.

Let $k$ $(1 \leq k \leq \min (n, m))$ be the least positive integer such that $x_k \cdots x_t \neq y_k \cdots y_t$. Let $\alpha$ be the string that results from $x_k \cdots x_t$ by deleting all predicate tests ($\alpha$ can be the empty string). Since $x_1 = y_1$ and $S_1$ and $S_2$ are strongly 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

$L_{00} : \text{If } P_i (x) \text{ then } L_1 \text{ else } L_2$ and $L_{01} : \text{If } P_j (x) \text{ then } L_1 \text{ else } L_3$, respectively. Then $L_1$ cannot be equivalent to both of $L_{01}$ and $L_{02}$, otherwise the occurrence of $P_i$ in statement $L_{00}$ is superfluous. So suppose that $L_1$ and $L_{01}$ are not equivalent. Then there is a Herbrand interpretation $H_0$ such that

$\text{val} (S_1, H_0, L_1) \neq \text{val} (S_2, H_0, L_{01})$.

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

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

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

Let $H_2$ be any Herbrand interpretation satisfying the following:

(A) For all proper suffixes $\alpha'$ of $\alpha$ and for all predicate symbols $P_i$ 

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

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

(C) $(P_i)^{H_2}(\alpha x) = \text{True};$ and
For all strings $a' = w', \alpha$ such that $\alpha$ is a proper suffix of $a'$ and for all predicate symbols $P$,

$$(P_{1})^{H_{S}}(a'x) \neq (P_{1})^{H_{S}}(w'x).$$

Clearly such Herbrand interpretations exist. For each such Herbrand interpretation $H_{S}$,

$$\text{val}(S_{1}, H_{S}) \neq \text{val}(S_{2}, H_{S}).$$

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 $L^H(S)$. 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

$$\Sigma = \{f, f_{p_{1}}, f_{p_{2}}, f_{p_{1}^{*}}, f_{p_{2}^{*}}, f_{p_{1}^{+}}, f_{p_{2}^{+}}\}.$$

![Diagram of $A(S_{0})$](image)

**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 \( j \)th 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 \( \Sigma \) not occurring as labels on outgoing edges.

Clearly the automaton \( A(S_0) \) accepts the language \( L''(S_0) \). Thus rephrasing Theorem 3.7, for reduced strongly free sfp schemes \( S_1 \) and \( S_2 \), \( S_1 = S_2 \), if and only if, the associated deterministic finite automata \( A(S_1) \) and \( A(S_2) \) are equivalent. Noting that, for a strongly free sfp scheme \( S_1 \), \( A(S_1) \) is constructible 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 sfp schemes is decidable deterministically in polynomial time.

### 3.2. Reducing strongly free lanov schemes in deterministic polynomial time.

In order to extend the equivalence algorithm to arbitrary strongly free lanov 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 \( x, y \in \{ f \}^* \). 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 \( \Sigma'' := \{ a_1, \cdots, a_n \} \). 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, (the predicate is false) and 1, (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 \( \delta \).

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 \( B_n^i \).

The algorithm starts with two blocks, \( B_0^0 := \{ \text{halt state} \} \), \( B_0^1 := \{ \text{all nonhalt states} \} \), and proceeds to split blocks into smaller blocks until no further splitting is possible. It is possible to split a block \( B_n^i \) as long as condition ** given below holds:

** \( \exists a \in \Sigma'' \exists s_1, s_2 \in B_n^i \text{ such that} \)

\[ \delta(s_1, a) \in B_n^i \text{ and } \delta(s_2, a) \in B_n^i; \]

for \( \delta \) 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 $\delta$’s transition on the label $a$.

The informal algorithm is this.

**REDUCTION ALGORITHM 3.10.** Start with $\Sigma$, $A(S)$. Form $\Sigma'$ as the set of labels with predicates removed and $A'(S')$ as the automaton with predicates removed from labels (but written on the states). Let $B^0_i$ contain the half state and $B^1_i$ all non-halt states. Let $N$ be the stage number, initially $N = 0$. Let $\delta_N$ be the (non-deterministic) transition function arising from the $\delta$ of $A(S)$.

BEGIN REDUCTION ALGORITHM

initialize (set $N = 0$, set up $B^0_1$, $B^1_1$).

while $\ast \ast$ do

begin

1. compute the output behavior of each state under $\Sigma^N$ (at stage $N$) using each possible transition of $\delta_N$.
2. locate the non-redundant states at stage $N$, i.e. $\delta_N(i, a) \in B^0_i$ and $\delta_N(i, b) \in B^1_i$, $i \neq j$ (possibly $a = b$).
3. form a new set of labels, $\Sigma^{N+1}$, 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 $\Sigma^{N+1}$, $\delta_{N+1}$.
5. split blocks $B^0_i$ to form blocks $B^{N+1}$ by grouping only those states of $B^0_i$ 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 $L$: if $P$, then $L_1$ else $L$,

and connect all incoming edges to $L_1$ (that is, replace any goto $L$ by goto $L_1$).

Combining this algorithm with the reduction algorithm, we have an algorithm for transforming strongly free Ianov schemes $S$ to reduced strongly free Ianov schemes $S$ (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 $O(|\Sigma| \cdot |K|)$. Consider the time for each step, the bounds are

1. $\leq |\Sigma| \cdot |K|$
2. $\leq |K|$
3. $\leq |\Sigma|$
4. $\leq |\Sigma| \cdot |K|$
5. $\leq |K|$

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.
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 $\tilde{S}$ given $S$. That is,

(i) $S = \tilde{S}$ 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 $P$ remains in $S$ then it is not redundant because the algorithm produces an interpretation under which the true and false branches from $P$ are distinct. So we need only show that if a predicate occurrence is removed, say at state $s$ as

$$s: \text{if } P \text{, then } L_t \text{, else } L_f,$$

then that occurrence is really redundant. To prove this, suppose some predicate occurrences were erroneously removed, say $P_1$ at state $s_1, \ldots, P_n$ at states $s_n$. 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 $x_1, x_2, \ldots, x_t$ or the false one, $y_1, y_2, \ldots, y_t$ either (a) do not contain any $P_i$ (erroneously removed predicates) or (b) if such a $P_i$ does occur, then the true branch from it to the halt state ($x_t$ or $y_t$) is the same as the false branch, because otherwise this $P_i$ 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 $x_1, \ldots, x_t$ and $y_1, \ldots, y_t$ appear already in some $A^*(S)$. 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 $S = \hat{S}$ 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 $\Sigma$ 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 $S_1$, $S_2$ 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 lanov schemes, the generation of multi-exit predicates may require exponential time.

If all the predicate clusters in a lanov 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, $E^N(s)$; that is, a stage $N$ there may be $i = 1, \ldots, n$ 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 $E^N(s)$ if for all sequences of predicate tests $z_i$ such that $z_iQ^+y \in E^N(s)$ 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 $z_2Q^+y \in E^N(s)$. 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

\[
\begin{align*}
A & \quad B & \quad C \\
P^+Q^+ & P^+Q^- & P^-Q^+ \\
& & P^-Q^-
\end{align*}
\]

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

begin
(1) locate $Q$ in the cluster (let $y$ be the path to $O$),
(2) list all prefixes of the form $z$ where $zQ^y$ is in $E_1^y$,
(3) for each $z$ in (2) check whether there is a prefix $w$ where
   (a) $wQ^y$ is in $E_1^y$,
   (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 $E$ if the multi-exit redundancy procedure generates $Q$ given $C$ and $E$.

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 I 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 $i = 0, 1, 2, \ldots$, $L_i$ 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 ($L_i$) is $NP$-complete.
Proof. To show that Inequiv ($L_1$) is NP-hard, it suffices to show that $\mathcal{F}_2$, 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 $\Pi_f$ 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 $f = c_1 \land c_2 \land \cdots \land c_k$ where $c_i = c_{i1} \lor c_{i2} \lor c_{i3}$ and each $c_{ij}$ is a literal. Let the propositional variables of $f$ be $x_1, \ldots, x_m$. Then, the $L_1$ program $\Pi_f$ is constructed as follows. The input variables of $\Pi_f$ are $x_1, \ldots, x_m$ and the output variable of $\Pi_f$ is $z$. The program $\Pi_f$ has the form—$(A_1, A_2, \ldots, A_m, B_1, \ldots, B_k, D_1, \ldots, D_k, z \leftarrow 0, z \leftarrow z + 1, \text{Do } C', z \leftarrow 0, \text{End}, \text{Do } C_k z \leftarrow 0, \text{End})$, where $A_i, B_i$, and $D_i$ are program blocks defined as follows:

1. $A_i$ computes the value of $\bar{x}_i$, the negation of $x_i$

$$A_i \text{ is } \bar{x}_i \leftarrow 0$$
$$\bar{x}_i \leftarrow \bar{x}_i + 1$$
$$\text{Do } x_i$$
$$\bar{x}_i \leftarrow 0$$
$$\text{End.}$$

2. $B_i$ computes the value of the clause $c_{ij}$ for any given values of $x_1, \ldots, x_m, \bar{x}_1, \ldots, \bar{x}_m$. We illustrate the construction of $B_i$ by an example. Suppose $c_i$ is $x_1 \lor \bar{x}_2 \lor \bar{x}_3$, then

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

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

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

We leave it to the reader to verify that the program $\Pi_f$ outputs 1 for some assignment of values to its input variables, if and only if, $f$ is satisfiable. Otherwise, $\Pi_f$ 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 $L_3$ programs is undecidable [12].
DEFINITION 4.4. Let \( x \) and \( y \) be nonnegative integers. Then,

\[
x - y = \begin{cases} 
  x - y, & \text{if } x \geq y, \\
  0, & \text{otherwise}.
\end{cases}
\]

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

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

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

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

Let \( f = c_1 \land c_2 \land \cdots \land c_k \), where \( c_i = c_{i1} \lor c_{i2} \lor c_{i3} \) and each \( c_{ij} \) is a literal. Let the propositional variables of \( f \) be \( x_{i1}, \ldots, x_{in} \). The \( \mathcal{P}_1 \) program \( \Pi \) has \( n \) input variables \( x_{1}, \ldots, x_{n} \), output variable \( \mathcal{P} \), and has the form—\( A_1, \ldots, A_m, B_1, \ldots, B_k, D_1, \ldots, D_m, P \leftarrow 1 - c_1, \ldots, P \leftarrow 1 - c_k \). Here, \( A_1, B_1, \) and \( D_m \) are—

(a) \( A_i \) is \( x_i \leftarrow 1 - x_i \),

(b) letting \( c_i = x_{i1} \lor x_{i2} \lor x_{i3} \), \( B_i \) is \( x_i \leftarrow x_{i1} + x_{i2} + x_{i3} \), and

(c) \( D_m \) is \( x_m \leftarrow 1 - c_m \).

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

\( M \) operates as follows. It replaces all instruction of the form \( x_i \leftarrow 1 - x_i \) in \( \Pi \) by the program fragment

\[
x_i \leftarrow 0 \\
x_i \leftarrow x_i + 1 \\
\text{Loop } x_i \\
x_i \leftarrow 0 \\
\text{End}
\]

It replaces all instructions of the form \( x_i \leftarrow x_i - x_i \) in \( \Pi \) by the \( L_1 \) program fragment

\[
x_i \leftarrow x_i \\
\text{Loop } x_i \\
x_i \leftarrow x_i + 1 \\
\text{End}.
\]

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

REFERENCES


