The Complexity of Equivalence and Containment for Free Single Variable Program Schemes

Steven Fortune
John Hopcroft
Erik Meineche Schmidt

TR 77-310

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853
The Complexity of Equivalence andContainment
for Free Single Variable Program Schemes

Steven Fortune
John Hopcroft
Erik Meineche Schmidt
Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

Non-containment for free single variable program schemes is shown to be NP-complete. A polynomial time algorithm for deciding equivalence of two free schemes, provided one of them has the predicates appearing in the same order in all executions, is given. However, the ordering of a free scheme is shown to lead to an exponential increase in size.

This research was supported in part by the office of Naval Research under contract N00014-76-0018 and by Aarhus University, Aarhus, Denmark.
1. Introduction

Much work in the theory of program schemes has gone into the investigation of decidability properties for different classes of schemes \([G,M]\). In the cases where a problem is decidable, a natural question is to determine the complexity of the decision procedure. Some of those questions were answered in \([CHS]\) where it was shown that noncontainment and nonequivalence for single variable program schemes and for monadic linear recursion schemes are NP-complete.

In this paper we investigate the complexity of these two problems for the class of free single variable program schemes. The requirement of freedom (i.e. absence of pieces of code which cannot possibly be executed), is a very natural one if we want to consider schemes which are models of real programs. Although most real programs have more than one variable, we show that even in the single variable case the equivalence problem is difficult.

We show that the noncontainment problem for free schemes remains NP-complete. We do not know the complexity of the equivalence problem for free schemes (except that inequivalence is in NP), but we can reduce it to the problem of determining equivalence of acyclic schemes involving only predicates and terminal assignment statements. We present a partial solution to the equivalence problem by showing that if one of the schemes
has all predicates appearing in the same order, then there is a polynomial time algorithm. However, we show that there are schemes in which ordering the predicates causes an exponential increase in size, indicating that preprocessing by ordering one of the schemes cannot lead to a polynomial time algorithm.

The paper is organized in 5 sections. In section 2 we introduce the notion of a B-scheme, which is an acyclic single variable program scheme containing only predicates and terminal assignment statements. Section 3 contains the proof that noncontainment for free B-schemes is NP-complete as well as the polynomial time algorithm for the case where one scheme is ordered. In section 4 we present an unordered B-scheme with no small equivalent ordered scheme, and in section 5 we show that equivalence for the full class of free single variable schemes is decidable in polynomial time if and only if the equivalence problem for free B-schemes is decidable in polynomial time.

Although this is a paper about program schemes, some of the results, notably the exponential blow-up in section 4, are of interest in their own right. Since these results are formulated in terms of standard concepts from graph theory, no particular knowledge from program scheme theory is required.
2. Preliminaries

A B-scheme is a labeled rooted dag whose vertices have outdegree 2 or 0. Vertices with outdegree 2 are called tests and are labeled with Boolean variables; vertices with outdegree 0 are called leaves and are labeled by function symbols. One edge from a test is labeled T, the other F. \(|S|\) denotes the number of nodes in scheme S. A B-scheme is free if there is no path from the root to a leaf which contains two or more tests with the same label.

Let S be a B-scheme. A B-assignment A (assignment for short) is a mapping from the Boolean variables of S to (true, false). \(t(A)\) is the path constructed by starting at the root and selecting the edge labeled T (F) whenever encountering a test-labeled b where A(b) = true (false). The value mapping Val maps pairs of schemes and assignments to function symbols and is defined as follows:

\[
\text{Val}(S,A) = f \text{ iff the leaf reached by the path } t(A) \text{ has label } f.
\]

The B-schemes \(S_1\) and \(S_2\) are equivalent, \((S_1 \equiv S_2)\), if and only if for each assignment A, whose domain contains all Boolean variables in \(S_1\) and \(S_2\), \(\text{Val}(S_1,A) = \text{Val}(S_2,A)\). One function symbol \(\Omega\) is designated as a special symbol and represents the undefined function. \(S_1\) is contained in \(S_2\), \((S_1 \subseteq S_2)\), if and only if for each assignment A whose domain contains all Boolean variables in \(S_1\) and \(S_2\), either \(\text{Val}(S_1,A) = \Omega\) or \(\text{Val}(S_1,A) = \text{Val}(S_2,A)\).
schemes $S_1$ and $S_2$ are then the

\[ \left( \omega^c, \omega_p, \omega_a \right) \cdots \left( \omega^c, \omega_p, \omega_a \right) \left( \Omega^c, \Omega^q, \Omega_a \right) = \left( \Omega^c, \Omega^q, \Omega_a \right) \]

\[
\begin{align*}
\text{restriction that } u & = \frac{I}{I} \\
\text{and } v & = \frac{I}{I} \\
\text{and } w & = \frac{I}{I} \\
\text{and } z & = \frac{I}{I} \\
\end{align*}
\]

-6-
satisfiability of the formula \( \phi \) and \( \psi \) will enforce the
satisfiability intuitively, when \( S_1 \neq S_2 \), \( S_1 \) will force the
\( S_1 \) and \( S_2 \) such that \( S_1 \neq S_2 \) iff the original formula \( \phi \) is
obtained by replacing every \( x_i \). We will construct two schemes
occurrence of \( x_i \) by a distinct \( y_i \). Let \( P_i \) be the formula
occurrence of \( x_i \) in \( P \) by a distinct \( y_i \). Similarly
uncompimented occurrence of \( x_i \) in \( P \) times. Let \( \{ x_1, \ldots, x_n \} \) be new variables and replace every
g times. Let \( \{ z_1, \ldots, z_m \} \) be new variables and replace every
\( x_i \) appear uncompimented in \( P \) times and compensated
to it. Let \( P \) be a 3-CNF formula with variables \( x_1, x_2, \ldots, x_n \)
and
To show that \( \text{BNCNT} \) is NP-hard we reduce 3-CNF satisfiability
is in NP.

Proof: The usual guess and check method shows that \( \text{BNCNT} \)
is NP-complete.

\[ \{ S_1 \neq S_2 \} \]

\[ \text{BNCNT} = \{ (S_1, S_2) \mid S_1 \text{ and } S_2 \text{ are free } P\text{-schemes and} \] 

\[ \text{Theorem 3.1: The set} \]

\[ \text{Find polynomial time algorithms for equivalence.} \]

\[ \text{B-schemes is NP-complete, and that in certain cases we can} \]

\[ \text{Here we show that the containment problem for free} \]

\[ \text{3. Containment and equivalence for free } P\text{-schemes} \]

\[ \text{(CHS)} \]

\[ \text{By a HALT-statement, then we obtain the switching schemes of} \]

\[ \text{we note that if the leaves in a } P\text{-scheme are replaced} \]

-
Now if the original formula \( F \) was satisfiable we can find an assignment \( A \) such that \( \text{Val}(S_2, A) = g \) and \( \text{Val}(S_1, A) = f \), so \( S_1 \not\equiv S_2 \). Conversely, if \( S_1 \not\equiv S_2 \), then there is an assignment \( A \) such that \( \text{Val}(S_1, A) = f \) and \( \text{Val}(S_2, A) = g \). But \( \text{Val}(S_2, A) = g \) only if, for each \( i \), \( u_1^i = u_2^i = \ldots = u_1^i = \ldots = u_1^i = \ldots = v_1^i = \ldots = q_1^i \). Hence assigning to each \( x_1^i \) the value \( A(u_1^i) \) satisfies \( F \). Since \( S_1 \) and \( S_2 \) can be written down in time polynomial in the length of \( F \), \( \text{BNCONT} \) is \( \text{NP-hard} \).

We now turn to the equivalence problem for free B-schemes. First we show that if the two schemes are ordered, then there is a polynomial time algorithm for deciding equivalence.

**Definition 3.2:** A B-scheme with Boolean variables \( b_1 \ldots b_k \) is ordered if whenever a test labeled \( b_i \) is a predecessor of a test labeled \( b_j \) then \( i < j \).

In the proof of the next theorem we use the observation that if a scheme is ordered, then the size of the finite automaton accepting the interpreted value language [G] is polynomial in the size of the scheme.

**Theorem 3.3:** There is a polynomial time equivalence algorithm for ordered schemes.

**Proof:** Let \( S_1 \) and \( S_2 \) be schemes in which the Boolean variables \( b_1 \ldots b_k \) appear. We will construct deterministic finite automata \( M_1 \) and \( M_2 \) from \( S_1 \) and \( S_2 \) such that \( S_1 \equiv S_2 \) iff
The equivalence problem for two free B-schemes, provided one
is ordered, is decidable and deterministic.

Proof:
1. Let S₁ = true and S₂ = false. Then S₁ = S₂ if and only if

2. Define a free B-scheme S, and let S₁ and S₂ be free B-schemes.

3. Definition: A free B-scheme S is ordered if:

   a. There is a polynomial time algorithm for ordered B-schemes.

   b. Every scheme is ordered.

   c. The method can be characterized as "graph pushing."

   d. In the case where just one scheme is ordered, we choose this section by proving that Theorem 3.3

   e. In any free polynomial time algorithm, the deterministic finite automata can be done in polynomial time.
time polynomial in the size of $S$ and $\Sigma$, and each transition of
\( M \) and $L$ can be computed in \( \mathcal{O}(n^2) \). Since $M$ and $L$, $\forall \mathcal{S} \in \mathcal{T}$, $\exists \mathcal{T} \in \mathcal{I}$ and $\mathcal{T}$ can be computed in \( \mathcal{O}(n^2) \).

The boolean variables are ordered $\{ b_1, b_2, \ldots, b_n \}$, and for each accepting state $q$, we add a new accepting node $q^+$ and for each edge labeled $e$, we replace $q^+ \rightarrow q$ with $q^+ \rightarrow q^+$.

The edge labeled $p$ to a leaf, and for each state $q$, there is an edge from a test to a test if there is an edge from a test labeled $p$ to a test labeled $p$. We may need to add extra tests if the root is not labeled. Boolean variable $b_i$ is tested on every path from root to leaf.

$M$ is constructed as follows: We extend $S$ so that every

\[
\begin{align*}
\text{false if } & \mathcal{T} \text{ is a string accepted by } M \\
\text{true if } & \mathcal{T} \text{ is a string accepted by } M
\end{align*}
\]

where $\mathcal{T}$ is the assignment of $b_i$. We need to check if $\mathcal{T}$ is either $\bot$ or $\bot$ and $\mathcal{T}$ is a function symbol ($\forall \mathcal{T} \in \mathcal{T}$). Then $M$ will accept the string $\mathcal{T}$.
Algorithm 3.6:

Input: Free B-scheme $S_1$ and ordered B-scheme $S_2$.
Output: "Yes" if the schemes are equivalent, "No" otherwise.

begin
    comment $L$ is a list of pairs of graphs which must be equivalent in order that $S_1$ and $S_2$ be equivalent;
    initialize $L$ to $(S_1, S_2)$;
    repeat
        let $n$ be a node of $S_1$ all of whose predecessors have been marked and let $v$ be the subgraph with root $n$;
        let $(v, v_1), \ldots, (v, v_m)$ be all the pairs of graphs on $L$ in which $v$ occurs;
        comment since $v_1, v_2, \ldots, v_m$ are subgraphs of an ordered scheme, the method in Theorem 3.3 can be used to test their equivalence;
        if $\neg (v_1 \equiv v_2 \equiv \ldots \equiv v_m)$ then output ("No") and halt;
        if $v$ is a leaf then
            comment since $v$ is trivially ordered, the method in Theorem 3.3 can again be used to test equivalence of $v$ and $v_1$;
            if $\neg (v \equiv v_1)$ then
                output ("No") and halt;
        else
            A: add to $L$ the pairs $(v', v_1 [b=true])$ and $(v'', v_1 [b=false])$ where $b$ is the label of $v$'s root $n$ and $v'(v'')$ is the subgraph of $S_1$ reachable via $n$'s outgoing T-edge (F-edge)
            fi;
            remove the pairs $(v, v_1), \ldots, (v, v_m)$ from $L$;
            mark $n$;
        until all nodes of $S_1$ have been marked;
        output ("Yes") and halt;
Theorem 3.7: Algorithm 3.6 works correctly and runs in polynomial time.

Proof: It follows from Lemma 3.5 that the property

\[ P: S_1 \equiv S_2 \iff \forall (v, v_1) \in L : v \equiv v_1 \]

is an invariant for the loop. To show correctness then, it is sufficient to note that \( P \) is true initially and that when the algorithm stops, one of the following is true:

a) all nodes have been marked, the list \( L \) is empty and the answer is "Yes".

b) not all nodes have been marked, there is a pair \((v, v_1)\) on \( L \) such that \( v \not\equiv v_1 \) and the answer is "No".

To see that the algorithm runs in polynomial time observe that the loop is executed at most \(|S_1|\) times and each execution of the loop requires at most \(|S_2|\) equivalences of ordered schemes which can be done in polynomial time by Theorem 3.3.

Note that the freedom of \( S_1 \) guarantees that the graph \( v'(v") \) in the statement labeled A in the algorithm is equal to \( v[b=\text{true}](v[b=\text{false}]) \).

4. A scheme with no small equivalent ordered scheme

Here we construct a free B-scheme \( S_0 \) whose smallest ordered equivalent has size "exponential" in \(|S_0|\). First we need some extra notation.

Let \( S \) be a B-scheme. A **partial B-assignment** (partial assignment for short) is a partial mapping from the Boolean variables of \( S \) to \{true, false\}. Two partial assignments \( A_1 \) and
The following facts about $S_0$ are evident:

\[ \{ \{ \ldots, T_{n+1}, \ldots, T_1 \} \} \text{ and } \{ \{ \ldots, T_{n-1}, \ldots, T_1 \} \} \]

are just cyclic permutations of equivalent sets. Note that the sets of equivalent terms in the $i$th and $j$th columns of $C$'s obtained as follows. Remove from the set of equivalent terms the $i$th leaf is replaced by the column \( C_{n-i} \) obtained as every element from $0$ to $n-1$.

The leaves are numbered from the root to $n$. The leaves of the tree labeled with $n+1$.

The scheme $S_0$ is now constructed in two stages.

1. $T_1$ is reachable via $A$.

2. Note that if $A$ satisfies all equivalences then the node labeled $1$.
shown below.

\[ \{ T_\Lambda = T \, \cdots = T_n \} \]

we construct the scheme, called a column.

\[ (V) \Lambda \]

are both determined and are equal. Given a set of equations

\[ (T_n) V \]

and

\[ \Lambda = T \, \cdots = T_n \]

partial assignment \( \Lambda \) satisfies an equality, we say that a

\[ \Lambda = T \, \cdots = T_n \]

2n-1 boolean variables \( u_1, \ldots, u_n \). We say that a

Assume that \( n \) is a power of 2. The scheme \( 0 \) with contam

By \( n \):

of a scheme certain tests not on the path may already be specified
determined by \( n \) can not be extended arbitrarily by an extraction
extension of \( n \) is said to be reachable via \( n \). Note that the path

are said to be specified by \( n \). Any node specified by some

with a level on which \( n \) is not determined. Nodes on this path

path from the root to a node which is either a leaf or a test

let \( S \) be a scheme. A partial assignment \( \Lambda \) determines a

boolean variable \( p \), \( \Lambda(p) \) determined implies \( \Lambda p \) is an extension of \( \Lambda \) for each

\[ \left\{ \begin{array}{c} \forall \Lambda \, (\forall p) (\forall \Lambda) \Lambda(p) \text{ is an extension of } \Lambda \text{ for each} \\ \text{undetermined otherwise} \end{array} \right\} = (\forall) (\forall n) (\forall p) n(p) \]

\[ \Lambda(p) \text{ is determined to be } \Lambda(p) \text{ and } \forall n \forall p \Lambda(n) \text{ is determined to be } \Lambda(p) \]

are both determined. The union of two consistent partial assignments

\[ \forall n \text{ are consistent if they have the same value whenever they} \]

-13-
c) Every path from the root to a leaf labeled 1 is missing \( \log n \) variables among the \( v \)'s.

Now let \( S_1 \) be an ordered \( B \)-scheme which is equivalent to \( S_0 \), and let \( Y \) be the \( \sqrt{n/2} \) Boolean variables which come first in the ordering. We shall show that there are "exponentially" many assignments to variables in \( Y \) which compute different functions of the remaining variables. Since each of these different functions must be represented by different nodes in \( S_1 \), \( S_1 \) must have "exponentially" many nodes.

Relabel the variables such that \( Y = \{ y_1, \ldots, y_{\sqrt{n/2}} \} \) and let the remaining variables be \( Z = \{ z_1, \ldots, z_{2n-1-\sqrt{n/2}} \} \). Call a column in \( S_0 \) acceptable if there is no equality \( y_i = y_j \) between two elements of \( Y \) appearing in the column. There are at most \( (\sqrt{2})^2 = n/2 \) unacceptable columns. Call an assignment \( A \) to variables in \( Y \) acceptable if there is some acceptable column reachable via \( A \).

Now we show the key result of this section, that if two acceptable assignments are "a little different" then they can be extended such that one of them specifies a node labeled 1 and the other a node labeled 0.

**Lemma 4.1:** Let \( A_1 \) and \( A_2 \) be acceptable assignments (to the variables in \( Y \)) which differ in more than \( \log n \) variables. Then there is an assignment \( A \) to the variables in \( Z \) such that \( \text{Val}(A_1 \cup A, S_0) \neq \text{Val}(A_2 \cup A, S_0) \).
Proof: Since $A_1$ and $A_2$ are acceptable assignments, we can always reach acceptable columns via $A_1$ and $A_2$. There are two cases to consider:

1) Assume that some acceptable column $C$ is reachable via both $A_1$ and $A_2$. There are $2 \log n$ variables which do not appear in $C$. Half of them are $u$'s which appear on the path from the root to the column. The other half consists of $v$'s. $A_1$ and $A_2$ cannot differ on the variables on the path from the root to $C$ since $C$ is reachable via both $A_1$ and $A_2$. Thus even if $A_1$ and $A_2$ differ on all the $\log n$ $u$'s missing from column $C$, there is at least one variable, $y_i$, which appears in an equality of $C$ on which $A_1$ and $A_2$ differ. (The variable $y_i$ may be either a $u$ or a $v$, we don't care which.) The equality in which $y_i$ appears must be of the form $y_i = z_j$, $z_j \in Z$ since the column is acceptable, that is, the column has no equality between two $y$'s. Since $S_0$ is free, $z_j$ does not appear on the path from the root to $C$. Hence we can find an assignment $A$ to the variables in $Z$ such that $A_1 \cup A$ and $A_2 \cup A$ both specify $C$ and $A_1 \cup A$ satisfies all equations in $C$. However, $A(z_j) = A_1(y_i) \neq A_2(y_i)$ so $\text{Val}(A_1 \cup A, S_0) = 1$ and $\text{Val}(A_2 \cup A, S_0) = 0$.

2) Assume that there is no acceptable column $C$ which is reachable via both $A_1$ and $A_2$. We first find a partial assignment $A$ to the variables in $Z$ such that $A_1 \cup A$ specifies a column which can be satisfied by some extension, $A'$, of $A_1 \cup A$. Then we show that we can choose the extension $A'$ such that it satisfies the column specified by $(A_1 \cup A)$ but the column specified by $(A_2 \cup A) \cup A'$ is not satisfiable.
and using the induction hypothesis

\[ (T_{m+1})^V = (T_{m})^V \cdot (T_{m})^{V'} + (T_{m})^{V'} \cdot (T_{m-1})^{V''} \]

Now \( m = x+1 \)

The root is not labeled with a variable \( x \). Hence \( x \) and \( z \) in the right subtree \( l \). There are two cases to consider.

\( a \). Let the number of variables from \( m \) in the left subtree be \( \beta \). Let the number of leaves labeled \( I \), and in the right subtree be \( \beta' \), and in the right subtree be \( \beta' \).

Complete binary trees with \( X \) leaves. Let the number of leaves \( n \).

Induction step: Assume that \( \lambda(m', x-1) \geq \lambda(m, x-1) \)

Base case: The result is immediate for \( x = 0 \).

**Proof:** The proof is by induction on \( x \), the height of the tree.

- The number of acceptable assignments. Then \( \lambda(m', x) \geq \lambda(m, x-1) \) and \( \lambda(m', x) \geq \lambda(m, x-1) \).
- A leaf labeled \( I \) is reachable from \( l \), and hence by \( \lambda(m', x) \) the assignment is \( \lambda \). Call an assignment to the vertices in \( \lambda \) acceptable.
- If \( n \) is any subset of \( T_{m-1} \) and \( T_{m-1} \) leaves in \( T_{m-1} \) over \{0, 1\}, let \( T \) be an subset.
- In a complete binary tree, with \( T \) labeled nodes labeled with variables.

Lemma 4.2: Let \( G \) be a graph whose adjacency matrix is a complete binary assignment is \( b \).

Following Lemma which states that the total number of acceptable

Lemma 4.2: Let \( G \) be a graph whose adjacency matrix is a complete binary assignment is \( b \).

Before we can show that there are many acceptable assignments,
This completes the proof of the lemma.

Let $\mathcal{C}_1$ be an acceptable column reachable via $\mathcal{A}$, and let $\mathcal{A}$
-19-

\[ A(m,g,k) \geq 2^l \left( 2^r g_+ \cdot 2^{k-1} \right) + 2^r \left( 2^l g_- \cdot 2^{k-1} \right) \]
\[ - (2^r g_+ \cdot 2^{k-1}) \cdot (2^l g_- \cdot 2^{k-1}) \]
\[ = 2^{l+r} \left( \left( g_+ + g_- \right) / 2^k - \frac{g_+ g_-}{2^{2(k-1)}} \right) \]
\[ = 2^m \left( g/2^k + g/2^k - \frac{g_+ g_-}{2^{2(k-1)}} \right) \]
\[ \geq 2^m g/2^k \text{ as } g_+ \leq 2^{k-1}, g_- \leq 2^{k-1} \]

2) The root is labeled with a variable from M. Then

\[ l+r+1 = m \text{ and} \]
\[ A(m,g,k) = 2^l A(r,g_+,k-1) + 2^r A(l,g_-,k-1) \]
\[ \geq 2^l \left( 2^r g_+ \cdot 2^{k-1} \right) + 2^r \left( 2^l g_- \cdot 2^{k-1} \right) \]
\[ = 2^{l+r} \left( g_+ + g_- \right) / 2^k \]
\[ = 2^m g/2^k \]

Now we can prove that any ordered scheme equivalent to \( S_0 \) must be big.

**Theorem 4.3:** Let \( S_1 \) be an ordered B-scheme which is equivalent to \( S_0 \). Then

\[ |S_1| \geq 2^{m-(\log^2 n+1)/2} \quad \text{where } m = \sqrt{n/2} \]

**Proof:** From the discussion preceding Lemma 4.1 we know that \( S_0 \) contains at least \( \frac{n}{2} \) acceptable columns. Since \( Y \) contains \( m \) variables there are at least \( A(m,\frac{n}{2},\log n) \) acceptable assignments to variables in \( Y \). From Lemma 4.1 we know that if two of these assignments differ by more than \( \log n \) of the variables then they must lead to two different nodes in \( S_1 \). Now there are at
most \( \binom{m}{i} \) assignments to \( m \) variables which differ from a given assignment in \( i \) variable values. Hence there can be at most
\[
\log n \sum_{i=0}^{m} \binom{m}{i} < \sum_{i=0}^{m} m^i < m^{\log n + 1}
\]
assignments which differ from a given assignment by at most \( \log n \) variables. Therefore, there are at least \( A(m, n/2, \log n)/m^{\log n + 1} \) acceptable assignments which differ by more than \( \log n \) variables and hence \( |S_1| \geq A(m, n/2, \log n)/m^{\log n + 1} \). By lemma 4.2 we now get
\[
|S_1| \geq (2^m \cdot (n/2)/2 \log n)/m^{\log n + 1} = 2^{m-1/2} (\log n + 1) \log m = 2^{m-1} (\log n + 1) (\log n - 1)/2 \text{ (recall that } m = \sqrt{n}/2) = 2^{m-1/2} (\log^2 n + 1)
\]
and the theorem is proved.

5. Extension to single variable program schemes

In this section we show that the equivalence problem for free single variable program schemes (free Ianov schemes) is polynomial time equivalent to the equivalence problem for free B-schemes.

A single variable program scheme (an I-scheme) is a rooted directed graph (not necessarily acyclic) whose nodes have outdegree 0, 1 or 2. Nodes with outdegree 2 are tests and are labeled with Boolean variables. Nodes with outdegree 0 and 1 are called function nodes and are labeled with function symbols. Only vertices with outdegree 0 may be labeled with \( \Omega \). Edges
Having shown how to handle $k$-equivalence for all $k$ we now

made altogether.

made for each value of $k$, hence at most $\frac{g}{2}$-scheme tests are

$k$-equivalent for $k = 1, 2, \ldots, t$. At most $\frac{g}{2}$-scheme tests are

nodes must have the same label), we can use Lemma 5.1 to compute

$\varepsilon$-equivalent. Since $\varepsilon$-equivalent is easy to determine the

nodes are $k$-equivalent for all $k$ and only if they are

Proof: It follows trivially from the preceding lemma that the two

in $s$ are $k$-equivalent for all $k$.

a polynomial time algorithm for determining if two function nodes

oracle for determining equivalence of $\frac{g}{2}$-schemes, there is

Theorem 5.2: Let $s$ be a free $I$-scheme with $t$ nodes. Given an

is of $\frac{g}{2}$-schemes,

and $u \equiv v \Rightarrow (x-1)$-equivalent and $v \equiv z \Rightarrow (x-2)$-equivalent where the last equivalence

In $V$ and $u \equiv v \Rightarrow (x-1)$-equivalent if and only if $u \equiv v$. Then $u$ and $v$ are $k$-equivalent if and only if $u \equiv v$ in $V$.

In $V$ let $\equiv$ be the equivalence class of the $\frac{g}{2}$-scheme whose root is the descendant of $u$.

Lemma 5.1: Let $s$ be a free $I$-scheme with function nodes $t$ and

and some equivalence tests on $\frac{g}{2}$-schemes.

states that $k$-equivalence can be determined from $(x-1)$-equivalence

The next lemma, the proof of which we leave to the reader,

Thus for example two function nodes are $k$-equivalent

pr($S^2_2$,$\alpha$).
Then $T$ is $\lambda$-equivalent to $\mathcal{L}$ if for each $\lambda$-assignment $\mathcal{A}$, $\pi(T, \mathcal{A}) = \pi(\mathcal{L}, \mathcal{A})$. Let $\mathcal{S}$ and $\mathcal{S}'$ be the (sub-)schemes with $\mathcal{T}$ and $\mathcal{T}'$ as roots. Let the path reach a leaf. Let function nodes $\mathcal{T}$ and $\mathcal{T}'$ appear in the path determined by $\mathcal{A}$. The string may be of length less than $\mathcal{A}$. If $\mathcal{A}$ is the string of function symbols appearing along the path $\mathcal{T}$, let the label $\pi(\mathcal{T}, \mathcal{A})$ for $\mathcal{T}$-scheme $\mathcal{T}$ and $\mathcal{A}$-assignment $\mathcal{A}$.

The domain of a $\mathcal{A}$-assignment is denoted as $\mathcal{A}$-assignment except that its set of all strings over $\mathcal{A}$ of length $\mathcal{A}$ or less. Let $\mathcal{A}$ be a set of function symbols, and denote by $\pi(\mathcal{A})$.

In [47], a decision procedure for deterministic finite automata on $\mathcal{P}$. [24-17]

The path determined by $\mathcal{A}$ in $\mathcal{S}$ is the obvious generalization of the state after computing the functions in $\mathcal{A}$. Variables in state $\mathcal{A}$ are the state after computing the functions in $\mathcal{A}$ of $\omega$. In $\mathcal{A}$, the state after mapping defining the values of the Boolean expressions from $\mathcal{A}$ into $\mathcal{A}$-assignments. The interpretation maps.

Let $\mathcal{A}$ be a set of function symbols. An $\mathcal{A}$-assignment $\mathcal{A}$ maps.

Step. We extend the notion of $\mathcal{A}$-assignment in the following way. The values of the Boolean variables can change after each function under Herbrand interpretations (free interpretations $\mathcal{G}$) where we shall only be interested in the behavioural of our schemes.

I-scheme is free if every $\mathcal{G}$-scheme which is a subgraph of $\mathcal{G}$.

Learning tests are labeled with $\mathcal{T}$ and $\mathcal{A}$ in $\mathcal{A}$-schemes.
define what it means for two I-schemes to be equivalent.

Let $S$ be an I-scheme and $A$ an I-assignment (i.e. $A$ maps elements from $(F-\{v\})^*$ to B-assignment). The value mapping $\text{Val}$ is defined as follows.

$$\text{Val}(S,A) = \begin{cases} 
\text{the function symbols on the path determined by } A \text{ if the path is finite and does not end in } \top \\
\top \text{ otherwise}
\end{cases}$$

Two I-schemes $S_1$ and $S_2$ are equivalent if $\text{Val}(S_1,A) = \text{Val}(S_2,A)$ for all I-assignments $A$. It is clear that this definition means equivalence under all Herbrand interpretations (free interpretations) and it is well known that this implies equivalence under all interpretations [G].

We would like to show that two schemes are equivalent iff their root nodes are $k$-equivalent for all $k$. Unfortunately this is not quite true; the problem is that the schemes may both compute $\top$ but do so in different ways.

A free I-scheme is compact if from every non-leaf node there is a path to a leaf not labeled $\top$.

**Lemma 5.3:** There is a polynomial time algorithm to transform any free I-scheme into an equivalent compact free scheme.

**Proof:** Immediate. $\blacksquare$

**Lemma 5.4:** Two free compact I-schemes $S_1$ and $S_2$ are equivalent iff their roots $n_1$ and $n_2$ are $k$-equivalent for every $k$. 


Proof: It is clear that if $n_1$ and $n_2$ are $k$-equivalent for all $k$, then $S_1$ is equivalent to $S_2$. Conversely, suppose $S_1$ is equivalent to $S_2$ and let $k$ be the smallest value for which there is a $k$-assignment $A$ such that $pl(S_1, A) \neq pl(S_2, A)$. Not both of $pl(S_1, A)$ and $pl(S_2, A)$ can end in $\Omega$, so assume $pl(S_1, A)$ does not.

We can extend $A$ to an $l$-assignment $A'$, $\exists k$ with $A'(w) = A(w)$ for all $w$, $|w| \leq k$, such that $A'$ defines a path to a leaf not labeled $\Omega$ in $S_1$. Now since the $k^{th}$ symbol on the path defined by $A'$ in $S_2$ is different from the $k^{th}$ symbol on the path in $S_1$, and $Val(S_1, A') \neq \Omega$, we must have $S_1$ not equivalent to $S_2$, a contradiction. \hfill \qed

Now the following theorem is an immediate corollary of the preceding lemmas.

Theorem 5.5: There is a polynomial time algorithm to decide equivalence of free I-schemes if and only if there is a polynomial time algorithm to decide equivalence of free B-schemes. \hfill \qed

We close this section with the remark that non-inclusion for I-schemes is NP-complete. Inclusion for I-schemes is defined exactly as for B-schemes with "I-assignment" replacing "B-assignment". That the problem is NP-hard is clear from Theorem 3.1. That it is in NP is shown in [CHS].