

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 and Containment
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.

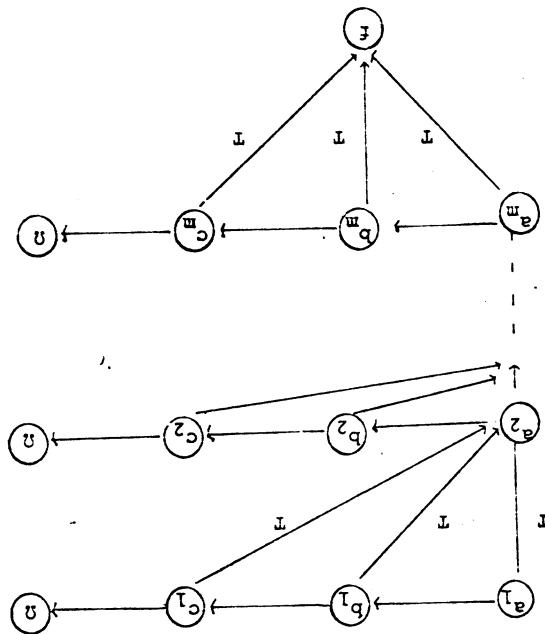
2. Preliminaries

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 $A(b) = \text{true}$ (false). The value mapping Val maps pairs of schemes and assignments to function symbols and is defined as follows:

$\text{Val}(S, A) = f$ iff the leaf reached by the path $t(A)$ 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 Ω 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 $\text{Val}(S_1, A) = \text{Val}(S_2, A)$.



S_1 :

schemes S_1 and S_2 are

Let $F' = (a_1 + b_1 + c_1) (a_2 + b_2 + c_2) \dots (a_m + b_m + c_m)$. Then the

restriction that $u_i = u_2 = \dots = u_p = v_1 = \dots = v_q$.

To show that BNCONT is NP-hard we reduce 3-CNF satisfiability to it. Let x_1, x_2, \dots, x_k be new variables and replace every occurrence of x_i by a distinct v_i . Let F' be the formula obtained by replacing every x_i in F by a distinct v_i . Similarly uncomplemented occurrence of x_i in F' is replaced by v_i . Let v_1, v_2, \dots, v_q be new variables and replace every complemented occurrence of x_i by a distinct u_i . We will construct two schemes obtained by replacing every x_i by a distinct s_i , and $s_1 \neq s_2$ will enforce the satisfiability of the formula F' , and $s_1 \neq s_2$ will enforce the satisfiability. Intuitively, when $s_1 \neq s_2$, s_1 will force the scheme s_1 and s_2 such that $s_1 \neq s_2$ if the original formula F is satisfiable.

PROOF: The usual guess and check method shows that BNCONT is in NP.

$$\text{BNCONT} = \{(s_1, s_2) \mid s_1 \text{ and } s_2 \text{ are free B-schemes and}$$

Theorem 3.1: The set

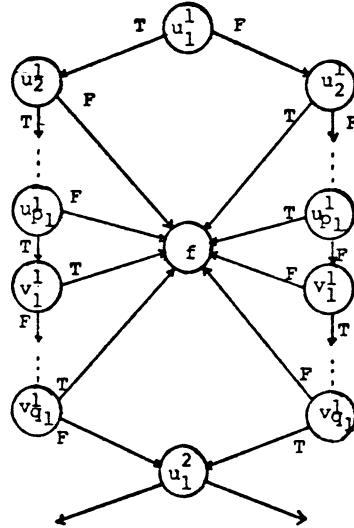
Here we show that the containment problem for free B-schemes is NP-complete, and that in certain cases we can find polynomial time algorithms for equivalence.

3. Containment and equivalence for free B-schemes

[CHS].

We note that if the leaves in a B-scheme are replaced by a HALT-statement, then we obtain the switching schemes of

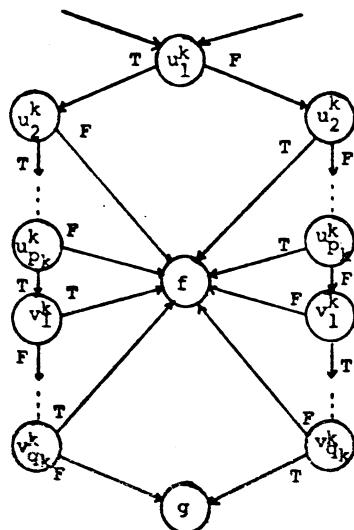
$s_2 =$



1

1

1



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 $S_1 \not\leq S_2$. Conversely, if $S_1 \not\leq S_2$, 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 , $u_1^i = u_2^i = \dots = u_{p_i}^i = v_1^i = \dots = v_{q_i}^i$. Hence assigning to each x_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 , BNCONT is 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 $b_1 \dots b_k$ 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 $b_1 \dots b_k$ 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

is ordered.

We now present a polynomial time algorithm which solves the equivalence problem for two free B-schemes, provided one

process: Immediate.

$S_1[b=\text{true}] \in S_2[b=\text{true}]$ and $S_1[b=\text{false}] \in S_2[b=\text{false}]$

it and only if

Lemma 3.5: Let S_1 and S_2 be free B-schemes. Then $S_1 \in S_2$

$S[b=\text{false}]$ is defined analogously.

2. Delete any inaccessible vertices.

(w,u) and give it the label of (w,v) .

vertex w such that (w,v) was in S , insert edge

the root, make u the root. Otherwise for each

the vertex v and any edges connected to it. If v was

delete v and any edges connected to it. Let u be

1. For each vertex v labeled b in S , do the following.

setting b to be true. More precisely:

variable. Then $S[b=\text{true}]$ is the scheme obtained from S by

definition 3.4: Let S be a free B-scheme and b a Boolean

The method can be characterized as "graph pushing".

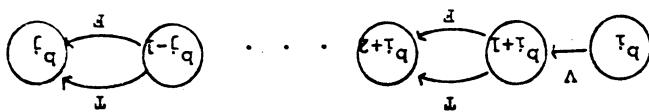
remains true in the case where just one scheme is ordered.

We close this section by proving that theorem 3.3

[AHU], there is a polynomial time algorithm for ordered schemes. ■

deterministic finite automata can be done in polynomial time

$L(M_1) = L(M_2)$ if $S_1 \subseteq S_2$. Since M_1 and M_2 can be computed in time polynomial in the size of S_1 and S_2 , and equivalence of
 since the Boolean variables are ordered it is clear that
 the start state, and the accepting node the only accepting state.
 edge labels are state transitions, the test labeled b_i
 resulting graph is the state graph of M_1 ; nodes are states,
 edge labeled f from the leaf to the accepting node. Then the
 we add a new accepting node and for each leaf labeled f an



is replaced with



the edge

labeled b_i to a leaf, and $i < k$. For example in the second case
 labeled b_j , and $j > i+1$, or (3) there is an edge from a test
 b_i , (2) there is an edge from a test labeled b_i to a test
 we may need to add extra tests if (1) the root is not labeled
 Boolean variable is tested on every path from root to leaf.
 M_1 is constructed as follows. We extend S_1 so that every

$$A(b_i) = \begin{cases} \text{false if } V_i = x \\ \text{true if } V_i = t \end{cases}$$

where A is the assignment

$$V_i \text{ is either } t \text{ or } x \text{ and } e \text{ is a function symbol if } Val(S_1, A) =$$

$$L(M_1) = L(M_2)$$

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 $(v, v_1), \dots, (v, v_m)$ be all the pairs of graphs on
 L in which v occurs;

comment since v, v_1, \dots, v_m are subgraphs of an ordered
 scheme, the method in Theorem 3.3 can be used to
 test their equivalence;

if $\neg (v_1 \equiv v_2 \equiv \dots \equiv v_m)$ 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 $(v, v_1), \dots, (v, v_m)$ 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

$$P: S_1 \equiv S_2 \Leftrightarrow \forall (v, v_i) \in L : v \equiv v_i$$

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_i) on L such that $v \not\equiv v_i$ 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. ■

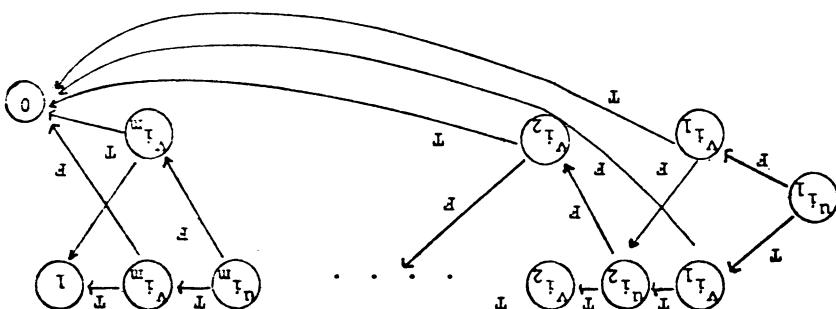
Note that the freedom of S_1 guarantees that the graph $v'(v'')$ in the statement labeled A in the algorithm is equal to $v[b=true](v[b=false])$.

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

- Note that if A satisfies all equalities then the node labeled 1 is reachable via A.
- The scheme S_0 is now constructed in two stages
- a) The base of S_0 is a complete binary tree with $n-1$ interior nodes labelled with u_1, \dots, u_{n-1} . The leaves are numbered from 0 to $n-1$.
- b) The i^{th} leaf is replaced by the column C_i , obtained as follows. Remove from the set of equalities all equalities involving variables that occur on the path from the root to Leaf i , and construct C_i from the remaining equalities. Note that the sets of equalities are just cyclic permutations of equalities between $\{u_1, \dots, u_{n-1}\}$ and $\{v_1, \dots, v_n\}$.
- The following facts about S_0 are evident
- S_0 is free and has $n-1+3(n-1-\log n) \cdot n+2n < 3n^2$ nodes.
 - No equality constraint appears more than once.



$A_1 \cup A_2$, is defined to be
 A_1 and A_2 , $A_1 \cup A_2$, is defined to be
 A_2 are consistent if they have the same value whenever they
 are both defined. The union of two consistent partial assignments
 A_1 and A_2 , $A_1 \cup A_2$, is defined to be
 $A_1(b) \cup A_2(b)$ if $A_1(b)$ is defined
 $\{A_1(b), A_2(b)\}$ if $A_1(b)$ is defined
 $A_1 \cup A_2(b) =$
 $A_2(b)$ if $A_2(b)$ is defined
 $A_1 \cup A_2$ is an extension of A_2 if for each
 A_1 partial assignment A_1 is an extension of A_2 if for each
 Boolean variable b , $A_2(b)$ defined implies $A_1(b) = A_2(b)$.
 Let S be a scheme. A partial assignment A determines a
 path from the root to a node which is either a leaf or a test
 with a label on which A is not defined. Nodes on this path
 are said to be specified by A . Any node specified by some
 extension of A is said to be reachable via A . Note that the path
 determined by A can not be extended arbitrarily by an extension
 of A since certain tests not on the path may already be specified
 by A .

Assume that n is a power of 2. The scheme S^0 will contain
 2^{n-1} Boolean variables $u_1, \dots, u_{n-1}, v_1, \dots, v_n$. We say that a
 partial assignment A satisfies an equality $u_j = v_j$ if $A(u_j)$ and
 $A(v_j)$ are both defined and are equal. Given a set of equalities
 $\{u_1 = v_1, \dots, u_i = v_i\}$ we construct the scheme, called a column,
 shown below

- 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 $Y = \{y_1, \dots, y_{\frac{\sqrt{n}}{2}}\}$ and let the remaining variables be $Z = \{z_1, \dots, z_{2n-1-\frac{\sqrt{n}}{2}}\}$. 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 $(\sqrt{\frac{n}{2}})^2 = \frac{n}{2}$ 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, $y_i \in Y$, 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 inductive hypotheses

$$- A(x, g_x^x, k-1) A(g_x^x, k-1)$$

$$A(m, g, k) = 2^k A(x, g_x^x, k-1) + 2^k A(g_x^x, k-1)$$

$$g+x = m. \quad \text{Now}$$

1) The root is not labeled with a variable in M , hence

2) and in the right subtree x . There are two cases to consider.
Let the number of variables from M in the left subtree be g_x^x . Let the number of variables from M in the right subtree be g_x^y .
Labelled 1 in the left subtree be g_x^y and in the right subtree
complete binary trees with 2^k leaves. Let the number of leaves
in the left subtree be $2^{g_x^x}$ and in the right subtree be $2^{g_x^y}$.
Induction step: Assume that $A(m, g, k-1) \leq 2^m/2^{k-1}$ and consider

Basics: The result is immediate for $k=0$.

Proof: The proof is by induction on k , the height of the tree.

number of acceptable assignments. Then $A(m, g, k) \leq 2^m/2^k$.
a leaf labeled 1 is reachable from it, and denote by $A(m, g, k)$ the
be g . Call an assignment to the variables in M acceptable if
of the variables of size m and let M be any subset
 u_1, \dots, u_{2^k-1} and 2^k leaves labeled over $\{0, 1\}$. Let M be any subset
tree, with 2^{k-1} interior nodes labeled with variables
Lemma 4.2: Let S be a B-scheme whose graph is a complete binary
assignments is big.

following lemma which states that the total number of acceptable
which differ by more than $\log n$ of the variables we prove the
Before we can show that there are many acceptable assignments

Let C_1 be an acceptable column reacchable via A_1 and let A
 be the minimum partial assigment such that A_1 satisfies C_1
 and all equations in C_1 involving variables in X are satisfied
 (this is always possible since A_1 is acceptable). S_0 is free and
 $\sqrt{n}/2 + \log n$ variables. Perform the following step while A_2
 does not specify some column: Let z^e be the label of the last
 node specified by A_2 . Extend A by setting z^e to be false, and
 if $z^e = z^e$ appears in C_1 , extend A to set z^e to false. (Setting
 z^e to true would work equally well.) This process
 terminates after adding at most $2 \log n$ variables to A , after
 which A_2 specifies some column C_2 (C_2 is not necessarily
 acceptable). Note that all equalities in C_1 involving variables
 in A_1 are still satisfied. There are at least $(n - \log n - |A|)/2$ =
 $(n - \log n - \sqrt{n} - 3 \log n)/2$ equalities in C_1 all of whose variables
 are unassigned by A_1 . There are only $2 \log n$ variables not
 appearing in C_2 , thus there is a $z^e = z^e$ in C_1 , z^e and z^e not
 assigned in A_1 , and $z^e = x^e$, some x^e , is in C_2 . x^e is not z^e
 by the construction of S_0 . Now by extending A so that all equalities
 in C_1 are satisfied, and $A(z^e) = A(z^e) \neq (A_2)_{A_1}(x^e)$, we can
 ensure that A_1 satisfies C_1 whereas A_2 does not satisfy C_2 .
 This completes the proof of the lemma.

$$\begin{aligned} A(m, g, k) &\geq 2^l(2^r g_r / 2^{k-1}) + 2^r(2^l g_l / 2^{k-1}) \\ &\quad - (2^r g_r / 2^{k-1})(2^l g_l / 2^{k-1}) \\ &= 2^{l+r}[(g_l + g_r)/2^{k-1} - g_l g_r / 2^{2(k-1)}] \\ &= 2^m[g/2^k + g/2^k - g_l g_r / 2^{2(k-1)}] \\ &\geq 2^m g / 2^k \text{ as } g_l, g_r \leq 2^{k-1} \end{aligned}$$

- 2) The root is labeled with a variable from M. Then
 $i+r+1 = m$ and

$$\begin{aligned} A(m, g, k) &= 2^l A(r, g_r, k-1) + 2^r A(l, g_l, k-1) \\ &\geq 2^l(2^r g_r / 2^{k-1}) + 2^r(2^l g_l / 2^{k-1}) \\ &= 2^{l+r}(g_l + g_r) / 2^{k-1} \\ &= 2^m g / 2^k \end{aligned}$$

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

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

Proof: From the discussion preceding Lemma 4.1 we know that S_0 contains at least $n/2$ acceptable columns. Since Y contains m variables there are at least $A(m, n/2, \log n)$ 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 $\log n$ $\sum_{i=0}^{\log n} \binom{m}{i} < \sum_{i=0}^{\log n} m^i < m^{\log n+1}$ 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

$$\begin{aligned}|S_1| &\geq (2^m \cdot (n/2) / 2^{\log n}) / m^{\log n+1} \\&= 2^{m-1/2(\log n+1)} \log m \\&= 2^{m-1-(\log n+1)(\log n-1)/2} \quad (\text{recall that } n = \sqrt{r}/2) \\&= 2^{m-\frac{1}{2}(\log^2 n + 1)}\end{aligned}$$

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 Ω . Edges

Having shown how to handle k -equivalence for all k we now

made altogether.

made for each value of k , hence at most $\binom{k}{2}$ B-scheme tests are made for k -equivalence for $k = 1, 2, \dots, t$. At most $\binom{t}{2}$ B-scheme tests are nodes must have the same label), we can use Lemma 5.1 to compute t -equivalent. Since 0-equivalence is easy to determine (the nodes are k -equivalent for all k if and only if they are

Proof: It follows trivially from the preceding lemma that two

in S are k -equivalent for all k .

a polynomial time algorithm for determining if two function nodes oracle for determining equivalence of free B-schemes, there is theorem 5.2: Let S be a free I-scheme with t nodes. Given an

is of B-schemes.

and n_2 are $(k-1)$ -equivalent and $V_1 \equiv V_2$, where the last equivalence relation. Then n_1 and n_2 are k -equivalent if and only if n_1 in V_1 by its equivalence class $[k-1]$ in the $(k-1)$ -equivalence $i=1$ or 2 (V_i may be simply a function node). Label each leaf i , n_2 . Let V_i be the B-scheme whose root is the descendant of n_i , Lemma 5.1: Let S be a free I-scheme with function nodes n_1 and some equivalence tests on B-schemes.

The next lemma, the proof of which we leave to the reader, states that k -equivalence can be determined from $(k-1)$ -equivalence and some equivalence tests on B-schemes.

If they have the same label. Thus for example two function nodes are 0-equivalent $p_1(S^2, A)$.

leaving tests are labeled with T and X as in B-schemes. An I-scheme is free if every B-scheme which is a subgraph is free. We shall only be interested in the behavior of our schemes under Herbrand interpretations (free interpretations [G]) where the values of the Boolean variables can change after each function step. We extend the notion of B-assumptions in the following way. Let F be a set of function symbols. An I-assignment A maps elements from $(F - \{A\})^*$ into B-assumptions. The interpretation of $A(w)$ is the mapping defining the values of the Boolean variables in state w (the state after computing the functions in w). The path determined by A in S is the obvious generalization of the trace $t(A)$ defined for B-schemes. The path determined by A in S is very similar to the minimum-B-schemes uses a procedure which is very similar to the minima- Z -schemes for determining equivalence of free I-schemes in polynomial time given an oracle for equivalence of free I-schemes in $[AU]$.

The proof that we can determine equivalence of free I-schemes in polynomial time over $F - \{A\}$ of length k or less. A set of all strings over $F - \{A\}$ of length k or less. A domain is $(F - \{A\})^*$ rather than $(F - \{A\})^*$. The path I -assumption is defined as a I-assumption except that its length is less than k if the path reaches a leaf. The string may be of length less than k if the path determines by A . (The string may be of length less than k if the path appears along the path A , is the string of function symbols appearing along the path A , and n_1 and n_2 appear in the path I labeled $p_1(S, A)$ for I-scheme S and I-assumption A , $p_2(S, A) =$ n_1 and I and S_1 and S_2 be the (sub)-schemes with n_1 and n_2 as roots. Then n_1 is equivalent to n_2 if for each I-assumption A , $p_1(S_1, A) =$

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 $(F - \{\Omega\})^*$ to B-assignment). The value mapping Val is defined as follows.

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

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 Ω 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 Ω .

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

Proof: Immediate. ■

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 .

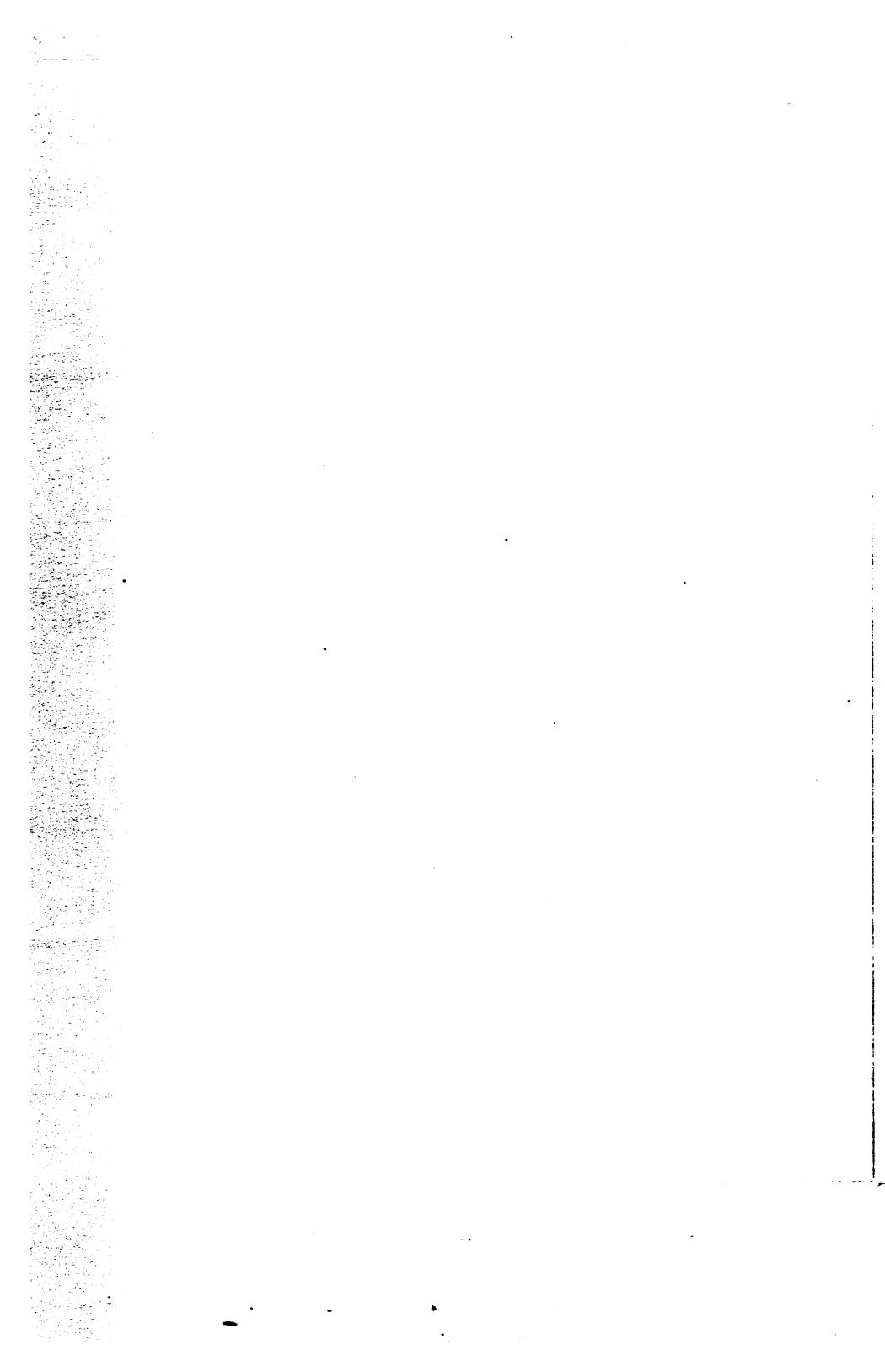
Proof: It is clear that if n_1 and n_2 are k -equivalent for all k , then S_1 is equivalent to S_2 . Conversely, suppose S_1 is equivalent to S_2 and let k be the smallest value for which there is a k -assignment A such that $p\ell(S_1, A) \neq p\ell(S_2, A)$. Not both of $p\ell(S_1, A)$ and $p\ell(S_2,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

We can extend A to an ℓ -assignment A' , $\ell \geq k$ with $A'(w) = A(w)$ for all w , $|w| \leq k$, such that A' defines a path to a leaf not labeled Ω 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 $\text{Val}(S_1, A') \neq \Omega$, we must have S_1 not equivalent to S_2 , a contradiction. ■

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. ■

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].



References

- [AHU] A.V. Aho, J.E. Hopcroft, J.D. Ullman: "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Company, 1974.
- [AU] A.V. Aho, J.D. Ullman: "The Theory of Parsing, Translation and Compiling", Volume I: Parsing, Prentice-Hall, Inc., Englewood Cliffs, N.J. 1972.
- [CHS] R.L. Constable, H.B. Hunt III, S. Sahni: "On the Computational Complexity of Scheme Equivalence", Proc. Eighth Ann. Princeton Conference on Information Sciences and Systems, Princeton University, 1974. Also submitted to SIAM J. Computing.
- [G] S.A. Greibach: "Theory of Program Structures: Schemes, Semantics, Verifications", Lecture Notes in Computer Science 36, Springer-Verlag, 1975.
- [M] Z. Manina: "Mathematical Theory of Computation", McGraw-Hill Inc. 1974.