STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-158

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-250

PROGRAM SCHEMAS WITH EQUALITY

BY

ASHOK K. CHANDRA

ZOHAR MANNA

SPONSORED BY
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457
DECEMBER 1971

COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY

PROGRAM SCHEMAS WITH EQUALITY

by

Ashok K. Chandra and Zohar Manna

Computer Science Department

Stanford University

Abstract

We discuss the class of program schemas augmented with equality tests, that is, tests of equality between terms.

In the first part of the paper we discuss and illustrate the "power" of equality tests. It turns out that the class of program schemas with equality is more powerful than the "maximal" classes of schemas suggested by other investigators.

In the second part of the paper we discuss the decision problems of program schemas with equality. It is shown for example that while the decision problems normally considered for schemas (such as halting, divergence, equivalence, isomorphism and freedom) are solvable for Ianov schemas, they all become unsolvable if general equality tests are added. We suggest, however, limited equality tests which Can be added to certain subclasses of program schemas while preserving their solvable properties.

1. Introduction

In recent years the study of schunas has been widely pursued in an attempt to understand the power of programming languages. In the study of program schemas, the functions and predicates allowed are usually considered to be uninterpreted symbols. The reason for this is that very simple interpreted programs yield all the partial recursive functions, and therefore interpreted programs do not provide insight into the difficulty in programming; e.g. the difference between the essentially iterative nature of Fortran and the recursive structure of Algol or PL/1.

Earlier works in this area, e.g. Ianov [1960], Rutledge [1964], Paterson [1967, 1968] and Luckham, Park and Paterson [1970] essentially considered flowchart schemas, and emphasized the decision problems for schemas, viz. halting, divergence, equivalence, etc. Most of the recent papers, on the other hand, e.g. Paterson and Hewitt [1971], Strong [1971a], Constable and Gries [1971] and Garland and Luckham [1971] considered more powerful schemas, i.e., flowchart schemas with additional programming features like counters, recursion, push-down stacks and arrays; and were concerned mainly with the problem of translating program schemas from one class to another.

Several formalisms have been considered in the literature for the description of schemas. We define a flowchart schema as being a program with the following features: it has a finite number of program variables denoted by $y_1, y_2, \ldots,$ a finite number of uninterpreted function symbols $f_1, f_2, \ldots$ (which may be ccxnbined with the variables to form terms) and a finite number of predicate symbols denoted by $p_1, p_2, \ldots$. Some of the function symbols may be zero-ary. These stand for individual constants, and are denoted by $a_1, a_2, \ldots$. A statement in the program may be:

(a) an assignment statement of the form

$$y_i \leftarrow t$$

where t is any term, (b) a predicate statement of the form

$$\underline{\text{if}}\ p_i(t_1, t_2, \ldots, t_n)\ \underline{\text{then}}\ \underline{\text{goto}}\ L_1\ \underline{\text{else}}\ \underline{\text{got0}}\ L_2$$

where $t_1, \ldots, t_n$ are terms and: L_1, L_2 are labels, or (c) a terminal statement, i.e., a START statement, a HALT statement or a LOOP statement. A schema has a uniquc START statement as its first statement. Free use of goto statements is allowed; and all statements except the START statement may be labelled. In addition, for convenience and readability we describe schemas using ALGOL-like features, e.g. while-statements and block structures. These clearly do not add any "power" and every such ALGOL-like program can be translated to an equivalent program that uses goto-statements instead.

Certain features can be added to flowchart schemas, e.g. counters or arrays. A counter is a special variable that takes nonnegative integer values. The operations allowed on a counter are adding one, subtracting one, and testing for zero. An array is a one-dimensional semi-infinite sequence of variables that can be referenced by using a counter to subscript the array.

In addition, we also consider recursive schemas. A recursive schema is a set of recursive definitions of functionals $F_1, F_2, \ldots$ of the form

$$F_i(y_1, \ldots, y_j) \leftarrow \underline{\text{if}}\ p(t_1, \ldots, t_n)\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t'$$

where p is an n-ary predicate symbol and $t_1, \ldots, t_n$, t and t' are terms that may consist of function symbols, functionals and the variables $y_1, \ldots, y_j$.

The research was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government. Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information, Springfield, Virginia 22151. Price: Full size copy $3.00; microfiche copy $.95.

It is quite surprising, though, that people have so far neglected to mention one of the most useful features: equality tests between terms, i.e., statements of the form

if $t_1 = t_2$ then got0 L_1 else got0 L_2 ,

where t_1, t_2 are terms and L_1, L_2 are labels.

The extension of program schemas to allow equality is quite natural, much as is the extension of first order predicate calculus to first order predicate calculus with equality. The analogy Can be extended further in that in both cases equality tests can be treated as just any other binary predicate but with a partial interpretation which in turn involves all other predicates and functions used in the system. This tends to be an unnatural approach to the treatment of equality. Accordingly, we prefer the direct approach of allowing the equality test to be a basic operation in the system as is the operation of assignment to a variable.

The reason for the anission of equality tests in earlier papers can perhaps be traced to the following fact. All schemas discussed in the papers mentioned above have one very important common property: the behavior of a schema for all interpretations can be characterized by the behavior for a subset of all interpretations viz. the Herbrand interpretations. We therefore call all these schemas Herbrand schemas. To be somewhat more precise, in a Herbrand schema, for every interpretation there "corresponds" a Herbrand interpretation that follows exactly the same path of computation. Flowchart schemas with equality tests are in general non-Herbrand schemas, that is, they may behave quite differently for Herbrand and non-Herbrand interpretations. Consider, for example, the simple schema:

START
if a = b then HALT else LOOP .

This schema halts for some interpretations and loops for others. For all Herbrand interpretations, however, it always loops. It is therefore a non-Herbrand schema, and further, there can be no Herbrand schema that is equivalent to it. A non-Herbrand schema that has no equivalent Herbrand schema is said to be an inherently non-Herbrand schema

The use of equality tests does not necessarily make a schema non-Herbrand. Example 0 in Appendix A is an interesting instance of a Herbrand program schema with equality tests that has an equivalent Herbrand program schema without any equality test and also an equivalent non-Herbrand program schema (which does have equality tests).

There are several other features which in general give rise to non-Herbrand schemas: the use of quantified tests is one such. Unfortunately, it is not partially decidable if a given schema is a Herbrand schema. This result follows from the fact that it is not partially solvable whether or not any given flowchart schema (without equality tests) diverges for every interpretation. Given any flowchart schema T , replace every HALT statement by the statement

if y=a then HALT else LOOP ,

where a is a new individual constant. Now the new schema is a Herbrand schema if and only if T diverges for every interpretation.

In the rest of this paper, we illustrate the power of equality tests (Section 2) and the decision problems concerning program schemas that use them (Section 3). For the sake of clarity we merely give the "flavor" of the examples in the main part of the paper, and we state the theorems without proof. Details of the examples are given in Appendix A (Section 4) and the proofs are sketched in Appendix B (Section 5). Detailed proofs can be found in Chandra [1972b].

2. The "Power" of Program Schemas with Equality

The use of equality tests in program schemas raises an old question that has been asked several times and never been answered to our complete satisfaction -- just what is a schema? We do not, in this paper, propose to answer this question, but we can indicate that much remains to be studied. It has been suggested (Constable and Gries [1971], Strong [1971b]), for example, that the class of program schemas with arrays might be a "maximal" class of schemas, i.e., for every schema there exists an equivalent schema in this class. Now, it may be that the class of array-schemas is indeed maximal with respect to the Herbrand schemas, but nevertheless alischemas in this class are Herbrand schemas. It has been shown, however, that there exist certain schemas using equality tests that are inherently non-Herbrand. This means that the class of program schemas with arrays and equality tests is a strictly larger class.

A problem is said to be a Herbrand problem if it can be solved by a Herbrand schema. A non-Herbrand problem is one that can only be solved by inherently non-Herbrand schemas. The class of program schemas with arrays and equality tests can solve certain non-Herbrand problems (which by the definition of a non-Herbrand problem cannot be solved if only arrays are allowed).

We first illustrate this point with two examples of non-Herbrand problems.

Example 1: Inverse of a unary function

Consider the following problem: "Given a unary function symbol f , a finite number of other n-ary function symbols, $n \geq 0$, and an input variable x , write a program schema that under any interpretation will yield a value of $f^{-1}(x)$ as output. That is, it finds an element y that can be expressed in terms of the given function symbols and the input variable x , such that f(y) = x ; if no such element exists, the schema loops forever". This problem, which is essentially one of inverting a given unary function, is non-Herbrand, the reason being that if the input x is equal to the zero-ary function a then it has no inverse in any Herbrand interpretation, whereas for other interpretations it may have an inverse. It follows that the task cannot be performed by any Herbrand schema. The task cannot be performed by any Herbrand schema. The task is, however, well within the capability of flowchart schemas with arrays and equality tests. A schema in this class that solves this problem is described in Appendix A.

Example 2: Herbrand-like interpretations

Given a set of function and predicate symbols of which there is at least one zero-ary function, we say that an interpretation I for this set is Herbrand-like if there exists some Herbrand interpretation H such that there is a 1-1 homomorphism from H into I . In other words, an interpretation I is herbrand-like if and only if for every pair of distinct terms t_1 and t_2 (made up of the given functions) the elements in I corresponding to t_1 and t_2 are distinct.

Now, consider the following problem: "given an interpretation for a set of function and predicate symbols, of which at least one is a zero-ary function, determine if the interpretation is not Herbrand-like. If the interpretation is not Herbrand-like then halt with no output, else diverge." This problem is inherently non-Herbrand in nature since a schema that solves this problem must diverge for every Herbrand interpretation. But for certain other interpretations the schema should halt. A schema with equality tests that solves the stated problem is presented in Appendix A.

The problem presented above is an abstract model closely related to certain problems in real life programming. As an illustration, consider a directed graph (with an identified root node) in which each node has two identified pointers leading from it. Pointers may lead to a terminal node "NIL" . The problem is to determine whether or not the given graph is a tree. This problem may be modelled by the above problem with two monadic functions representing the two pointers, and with the difference that the search for the equality of two "terms" is conducted not for the entire set of all terms, but for those terms not representing NIL. The correspondence is that the interpretation is Herbrand-like for this set of terms if and only if the corresponding graph is a tree.

Another related problem is that of determining if a given list is circular. In this problem, too, the explicit use of equality in a schema model of the computation represents a more natural approach than the treatment of equality as an interpreted predicate.

While the main interest in equality tests stems from the fact that programmers frequently do use tests of equality between variables whose values are data elements and these tests are often of a non-Herbrand nature, equality tests find some interesting applications in problems that are really Herbrand in nature. We give two examples below.

Example 3: Translation of flowchart schemas with Counters

The recursive schema

$F(x) \leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ F(F(f(x)))\ \underline{\text{else}}\ f(x)$

can be translated to an "impure" flowchart schema by introducing a counter. It can also be translated to a rather horrendous flowchart schema without any explicit counter (Plaisted [1972]). However, the use of equality gives a relatively simple flowchart schema equivalent to the above while retaining the advantage of having a "pure" schema (all functions and predicates being left uninterpreted). Details are presented in Appendix A.

Example 4: Efficient translation of linear recursive schemas

Consider the recursive schema T :

F(a) where

$F(y) \leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ g(F(f(y)),y)\ \underline{\text{else}}\ y$.

Let I be an interpretation of T for which there exists an n , $n \geq 0$, such that $f^n(a)$ = FALSE and for all $k < n$, $f^k(a)$ = TRUE . The output of the computation $\langle T,I \rangle$ is the term

$$g(g(g(\ldots g(f^n(a),f^{n-1}(a))\ldots,f^2(a)),f(a)),a)$$

For usual implementations of recursion the computation of the interpreted schema (T,I) takes time (the number of operations on data structures performed) and space (the number of values stored) both proportionalto n . The recursive schema T can be translated to an equivalent flowchart schema using a fixed memory size (number of variables) and time proportional to n*n . Using equality tests, however, the time can be brought down to some constant times $n^{(1+\epsilon)}$, where ϵ is any arbitrarily small positive number. Details of the construction are given in Appendix A. For further discussion of this topic, see Chandra [1972a].

3. Decision Problems

We consider the following decision problems for classes of schemas:

(a) The halting problem -- to decide whether a given schema in the class halts on every interpretation.

(b) The divergence problem -- to decide whether a given schema in the class diverges on every interpretation.

(c) The equivalence problem -- to decide whether two given schemas in the class are equivalent.

(d) The inclusion problem -- given two schemas A and B to decide whether A includes B , i.e., for every interpretation either both schemas halt with the same output or schema B diverges.

(e) The isomorphism problem -- to decide whether two schemas are isomorphic to each other. (Two schemas are said to be isomorphic, or operationally equivalent, if the sequences of statements executed by both schemas are exactly alike for every interpretation.)

(f) The freedom problem -- to decide whether a given schema in the class is free.

(g) The translation problem -- to translate any schema in the class to an equivalent free flowchart schema (using any number of variables).

It should be noted that the translation problem is not strictly a decision problem. We include it in this list, however, because it is an interesting problem closely related to the others.

All thccc questions can be answered in the affirmative for the class of Ianov schemas which consists of one-variable flowchart schema6 using only monadic function and predicate constants (Ianov [1960], Rutledge [1964]). In view of this it is somewhat unexpected that the addition of general equality tests to Ianov schema6 renders all these decision problems unsolvable. On the other hand, we show that these problems for Ianov schemas extended even to nonmonadic functions and resets but with limited equality tests are solvable.

It should be stated that for all "conventional" schemas, i.e., all schemas mentioned in this paper and in earlier works, the following problems are at least partially solvable:

(a') The halting problem -- to decide whether a given schema in the class halts on every interpretation.

(b') The non-divergence problem -- to decide whether a given schema ever halts,

(e') The non-isomorphism problem -- to decide if two schema6 are not isomorphic to each other.

(f') The non-freedom problem -- to decide if a given schema is not free.

The notable exception6 are the equivalence and inclusion problems. In general, the equivalence and inclusion problem6 as well as their negation6 are all not partially solvable.

3.1 Notation

We use the symbols

(1) $a, a_1, a_2, \ldots$ to represent individual constants (or zero-ary functions, if you will),

(2) $y, y_1, y_2, \ldots$ to represent program variables,

(3) $f, f_1, f_2, \ldots$ to represent functions, and we use

(4) $p, p_1, p_2, \ldots$ to represent predicates.

The set of terms is defined by the smallest set containing 8'6 , y's and closed under the following operation: if $t_1, t_2, \ldots, t_n$ are terms, and f_i is an n-ary function symbol, then $f_i(t_1, \ldots t_n)$ is also a term.

We use the notation $t(y_1, y_2, \ldots, y_n)$ to represent that $y_1, y_2, \ldots, y_n$ are the only variables that may be present in t . Thus a term t(y) may or may not contain the variable y , but contain6 no other variable. A term t() indicates therefore a constant term, that is, a term that has no occurrences of y's at all.

Given a nonconstant term t(y) , i.e., one containing the variable y , a common subterm t'(y) of t(y) is one such that if every occurrence of t,(y) in t(y) is replaced by an individual constant then t(y) is reduced to a constant term. Clearly the terms y itself and t(y) are common subterms of t(y) . Also, if t'(y) and t"(y) are common subterms of t(y) then t'(y) is a common sub-term of t"(y) or vice versa.

The assignment depth $\|t(y)\|$ of a term t(y) is defined to be the number of common subterms in t(y) excluding y itself. By convention, for a constant term t() , $\|t()\| = 0$.

The depth $|t(y)|$ of a term t(y)+ is the maximum depth of nesting in the term, and is defined by:

$|t()| = 0$,

$|y| = 0$,

$|f(t_1, t_2, \ldots$ [illegible] $| =$ [illegible] $\max(|t_1|, \ldots, |t_n|)+1$

Note that for monadic terms $\|t\| = |t|$, and in general $\|t\| < |t|$. A few examples illustrate this point. In-the following table

(a) stands for t(y) ;

(b) stand6 for common subterms of t(y) (excluding yy .itself);

(c) stand6 for $\|t(y)\|$;

(d) stands for $|t(y)|$.

(a)	(b)	(c)	(d)
Y		0	0
f(a)	-	0	0
f(y)	f(y)	1	1
f(g(h(y)))	h(y) ;gh(y) ;fgh(y)	3	3
f(g(a,y),g(a,y))	g(a,y) ;f(g(a,y), g(a,y))	2	2
f(y, g(a,y))	f(y,g(a,y))	1	2

3.2 Solvable Classes

Consider the rather general class S_1 of flowchart schemas with one variable. Schemas in S_1 contain the following statement types (L1 and L_2 are arbitrary labels in the definitions below):

START statement:	START $y \leftarrow a_1$
Final statements:	HALT or LOOP
Assignment statement:	$y \leftarrow t(y)$

predicate-test st.:	if $p(t_1(y), \ldots, t_n(y))$
	then goto L_1
	else got0 L_2

Equality-test st.:	if $t_1(y) = t_2(y)$
	then goto L_1
	else goto L_2

The equality tests allowed must, however, satisfy the condition that either t,(y) or t,(y) is a constant term, or else both $\|t_1(y)\|$ and $\|t_2(y)\|$ are less than or equal to 1 .

THEOREM 1 (Solvability of S_1) . For the class S_1

1(a) the halting problem is solvable

1(b) the divergence problem is solvable

1(c) the equivalence problem is solvable

1(d) the inclusion problem is solvable

1(e) the icomorphism problem is solvable

1(f) the freedom problem is solvable

1(g) any schema can be effectively translated to an equivalent free schema (with the addition of extra program variables).

This theorem includes as special cases the results of Ianov [1960], Rutledge [1964], and also recent extensions by Pnueli [private communication] and Garland and Luckham [1971].

As a special case, the problem6 (a)-(g) are solvable for the class of 1-variable monadic schema6 allowing resets and equality tests of the forms:

$t_1() = t_2()$, $y = t()$, $y = f_i(y)$, and $f_i(y) = f_j(y)$

Consider, next, the class S_2 of schernas, similar to the class S_1 , but with a change in the form of equality tests allowed, viz. the equality test statements allowed are of the form:

if $t_1(y) = t_2(y)$ then goto L_1 else goto L_2 ,

but this time the restriction is that $\|t_1(y)\| = \|t_2(y)\|$.

THEOREM 2 (Solvability of S_2) :

Problem6 (a)-(g) solvable for the class S_2 .

As a special case, the problem6 (a)-(g) are solvable for the class of 1-variable monadic schemas allowing resets and equality tests of the form:

$t_1(y) = t_2(y)$ where $|t_1(y)| = |t_2(y)|$.

3.3 Unsolvable Classes

It should well be asked why we have the "strange" restrictions on the form of equality tests above. The answer is that even slight generalizations of the restrictions above yield, astonishingly, classes whose problem6 are unsolvable. We demonstrate this on two classes.

Consider the class S_3 consisting of one variable y , one constant a , no predicates and only monadic function constants. Statement6 in schemas of S_3 are of the forms:

START statement:	START 3 - a
Final statements:	HALT or LOOP
Assignment statement:	$y \leftarrow f_i(y)$

Equality-test st.:	if $f_i(y) = f_j(f_k(y))$
	then goto L_1
	else goto L_2

S_3 differs fran S_1 in that nonconstant terms of depth 2 are used in &quality tests; and it differs from S_2 in that terms tested for equality do not have the same assignment depth.

THEOREM 3 (Unsolvability of S_3) : For the class S_3 :

3(a) the halting problem is unsolvable

3(b) the divergence problem is not partially solvable

3(c) the equivalence problem is not partially solvable

3(d) the inclusion problem is not partially solvable

3(e) the isomorphism problem is not partially solvable

3(f) the freedom problem is not partially solvable

3(g) there exists no effective translation to equivalent free schemas.

For the sake of completeness we should mention that the nonequivalence and the noninclusion problems for this class too are not partially solvable. Of course, the halting, nondivergence and nonisomorphism problem6 are partially solvable, which follows from the general result mentioned in the earlier parts of Section 3.

We introduce next the class S_4 of 1-variable monadic schemas similar to S_3 but with the difference that equality tests allowed have the following form:

if $y = t(y)$ then goto L_1 else goto L_2

where $1 \leq |t(y)| \leq 3$, i.e., tests may have any of the forms:

$y = f_i(y)$,

$y = f_i(f_j(y))$, or

$y = f_i(f_j(f_k(y)))$.

THEOREM 4 (Unsolvability of S_4) :

Problems (a)-(g) for the class S_4 are unsolvable.

A class of schemas is said to be solvable if its decision problems (a)-(e) are solvable; similarly, a class is unsolvable if its decision problems (a)-(e) are unsolvable. Classes S_1 and S_2 are solvable whereas S_3 and S_4 are unsolvable. On comparing these classes it is clear that there is a very sharp demarcation between classes of one-variable schemas that are solvable and those that are unsolvable, depending on the form of equality tests allowed. It should perhaps be asked how many function symbols suffice to render a class unsolvable. It can be shown, for example, that for the class S_3 , merely 4 functions are sufficient.

It is more interesting to note, however, that these function symbols can be "coded" using only 2 function symbols so that cchemas with one variable, two functions and general equality tests, i.e., tests of the form $t_1(y) = t_2(y)$, are unsolvable.

So far we have restricted our consideration to schemas that have only one variable. The reason is obvious: one-variable schemas provide the most interesting solvable classes. When more variables are allowed, even a very few features tend to make the schcmas unsolvable. For example, schemas with two variables, two functions and tests only of the form $y_i = f(y_i)$ are unsolvable.

It is even more interesting, though probably not surprising, that schemas with a single function too are unsolvable; for example, the class of one-function schemas having tests only of the form $y_i = y_j$ is unsolvable (5 variables suffice in this case) .

The proofs of these secondary results are also presented in Appendix B.

4. Appendix A -- Detailed Examples

Example 0: A Herbrand schema with equality

Not all schemas that use equality tests are non-Herbrand. Consider, for example, the schema

```
START
$y_1 \leftarrow y_2 \leftarrow a$;
L:  if $p(y_1)$ then
       if $p(y_2)$ then
          begin
          $y_1 \leftarrow f(y_1)$;
          $y_2 \leftarrow f(y_2)$;
          goto L;
          end
       else if $y_1 = a$ then HALT else LOOP
    else if $y_1 = y_2$ then HALT else LOOP .
```

This is a Herbrand schema because the equality test $y_1 = y_2$ must always be true, and the equality test $y_1 = a$ can never be entered. The given schema is hence equivalent to the following schema, which has no equality test.

```
START
$y \leftarrow a$;
L:  if $p(y)$ then
       begin
       $y \leftarrow f(y)$;
       goto L
       end
    else HALT.
```

The following schema is also equivalent to the above schemas, but it is a non-Herbrand schema because the LOOP statement in it can never be entered for any Herbrand interpretation. The schema is, however, not inherently non-Eerbrand.

```
START
$y \leftarrow a$;
L:  if $p(y)$ then
       if $y = f(y)$ then LOOP
       else begin
          $y \leftarrow f(y)$;
          goto L
          end
    else HALT .
```

Example 1: Inverse of a unary function

For simplicity we assume that the only functions are a single zero-ary function a , the given unary function f and a binary function g . The possible terms are therefore:

x , a , $f(x)$, $g(x,x)$, $f(a)$, $g(a, a)$, $g(x,a)$, $g(a,x)$, $f(f(x))$, . ..

The schema for any other set of functions is similar to the one for this particular case.

Symbols c_1, c_2, c_3 stand for counters.

Strictly, the only operations allowed on counters are adding and subtracting one, and testing for zero. For convenience, however, we will also allow other statements such as $c_i \leftarrow 0$, $c_i \leftarrow c_j$, and tests like $c_i = c_j$, as it is clear that these operations can be performed using only the legal operations and additional counters.

```
(1) --  START
        $A[0] \leftarrow x$;
        $c_1 \leftarrow 0$;
(2) --  $c_2 \leftarrow 1$; $A[c_2] \leftarrow a$;
(3) --  REPEAT:  $y \leftarrow A[c_1]$;
(4) --  if $f(y) = x$ then HALT(y);
        $c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow f(y)$;
        $c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow g(y,y)$;
        $c_3 \leftarrow c_1$;
        while $c_3 \neq 0$ do
           begin
           $c_3 \leftarrow c_3-1$;
           $c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow g(A[c_3],y)$;
           $c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow g(y,A[c_3])$;
           end;
        $c_1 \leftarrow c_1+1$;
(5) --  goto REPEAT .
```

After the initialization phase (lines (1) to (2))

$A[0] = x$, $A[1] = a$, $c_1 = 0$, $c_2 = 1$.

After completing one pass through the outer loop of the program (lines (3) to (5))

$A[2] = f(x)$, $A[3] = g(x,x)$, $c_1 = 1$, $c_2 = 3$,

and after a second pass

$A[4] = f(a)$, $A[5] = g(a,a)$,

$A[6] = g(x,a)$, $A[7] = g(a,x)$, $c_1 = 2$, $c_2 = 7$.

The algorithm works as follows: two pointers c_1 and c_2 reference the array. $A[c_1]$ represents the "current" value. If the current value is not the inverse, as determined by line (4), it is composed with values preceding it in the enumeration by function applications, and the new values obtained are added to the array.

It can be shown by induction that the process of enumeration generates and tests each possible term exactly once. This means that the inverse will be found if it exists. The point at which the test of the inverse is made could be changed to effect time efficiency but without altering the main features of the program.

Example 2: Herbrand-like interpretations

We assume that the only functions are a single zero-ary function a , a unary function f and a binary function g . Therefore the set of terms includes

a , $f(a)$, $g(a,a)$, $f(f(a))$, $g(f(a),f(a))$, $g(a,f(a))$, ...

The required schema is:

```
(1)     START
   --   A[0] ← a;
(2) --  c1 ← c2 ← 0;
(3) --  REPEAT: y ← A[c1];
       ┌ - - - - - - - - - ┐
       | c4 ← c1;          |
       | while c4 ≠ 0 do   |
(4) -- |    begin          |
       |    c4 ← c4-1;     |
       |    if A[c4] = y then HALT;
       |    end;           |
       └ - - - - - - - - - ┘
        c2 ← c2+1; A[c2] ← f(y) ;
        c2 ← c2+1; A[c2] ← g(y,y);
        c3 ← c1;
        while c3 ≠ 0 do
           begin
           c3 ← c3-1;
           c2 ← c2+1; A[c2] ← g(A[c3],y);
           c2 ← c2+1; A[c2] ← g(y,A[c3]);
           end;
        c1 ← c1+1;
(5) --  @ REPEAT  .
```

This program is quite similar to the previous one in the manner of enumeration of terms. The fact that each term is generated exactly once is used in making the test (4) to check if a value is repeated.

Example 3: Translation of flowchart schemas with Counters

The recursive schema

F(a) where

$F(y) \leftarrow$ if $p(y)$ then $F(F(f(y)))$ else $f(y)$,

can be translated to a flowchart schema with one program variable y and one counter c .

```
          START
          y ← a;
(1) --   c ← 0;
         while true do
             if p(y)
                then begin
                     y ← f(y);
(2) --               c ← c+1;
                     end
                else begin
                     y ← f(y);
(3)                  if c = 0 then goto DONE;
(4) --               c ← c-1;
                     end;
         DONE: HALT(y) .
```

Note that the test " c = 0 " above is not a test of equality between two data structures but rather between an interpreted variable, i.e., c , and an interpreted constant, i.e., 0 .

The corresponding equivalent flowchart schema with equality tests instead of counters uses three variables:

y plays the same role as the variable y above,

z effectively simulates a counter, and

w is a temporary variable.

The idea behind the method is that the variable z simulates a counter, where $f^i(a)$ stands for the integer i . Therefore, the statement $z \leftarrow a$ stands for the statement $c \leftarrow 0$, $z \leftarrow f(z)$ stands for $c \leftarrow c+1$, and the statements [$w \leftarrow a$; while $f(w) \neq z$ do $w \leftarrow f(w)$; $z \leftarrow w$] stand for $c \leftarrow c-1$. We have to be careful, however.

The term $f^n(a)$ stands for the integer n , $n \geq 0$, only if for no two distinct numbers $i,j \leq n$ are the terms $f^i(a)$ and $f^j(a)$ equal. Interpretations for which the counter is required to count up to an integer n where there exist $i,j \leq n$, $i \neq j$, such that $f^i(a) = f^j(a)$ are called looping interpretations. It can be shown that for looping interpretations the given recursive schema never halts. The required program schema is therefore easy to construct:

```
        START
        y ← a;
(1) --  z ← a;
        while true do
            if p(y)
                then begin
                     y ← f(y);
            ┌ - - - - - - - - - - - - - ┐
            |        w ← a;             | check
            |        while w ≠ z do     | for a
            |        if w = f(x)        | looping
            |           then LOOP       | inter-
            |           else w ← f(w);  | preta-
            |        if w = f(x) then LOOP;| tion
            └ - - - - - - - - - - - - - ┘
(2) --               z ← f(z);
                     end
                else begin
                     y ← f(y);
(3) --               if z = a then goto DONE;
            ┌ - - - - - - - - - - - - - - - - - ┐
            |        w ← a;                     |
(4) --      |        while f(w) ≠ z do w ← f(w); |
            |        z ← w;                     |
            └ - - - - - - - - - - - - - - - - - ┘
                     end;
        DONE: HALT(y) .
```

Example 4: Efficient translation of linear recursive schemas

Consider the recursive schema T :

F(a) where

F(y) ← if p(y) then g(F(f(y)),y) else y .

Let I be an interpretation of T for which there exists an n , n > 0 , such that $f^n(a)$ = FALSE , and $f^k(a)$ = TRUE for all k < n . The output of the computation of ⟨T,I⟩ is

$g(g(g(\ldots g(f^n(a),f^{n-1}(a)) \ldots f^2(a)),f(a)),a)$.

The computation of (T,I) takes time and space proportional to n -- for usual implementations of recursion. The recursive schema can be translated to an equivalent flowchart schema T' using a fixed memory size (number of variables) such that the computation of ⟨T',I⟩ takes time proportional to n^2, as follows:

```
START
y ← a;
while p(y) do y ← f(y);          -- y = P(a)
x ← a;
while p(x) do
    begin
    x ← f(x);
    x1 ← x;                      -- x = f^i(a)   i > 1
    z ← a;
    while p(x1) do
        begin
        x1 ← f(x1);
        z ← f(z);
        end;
    y ←  da;                     -- z = f^{n-i}(a)
    end
HALT(y) .
```

Using equality tests, however, the time can be brought down to $n^{1+\epsilon}$ where ϵ is an arbitrarily small number. We first describe an equivalent flowchart schema with equality tests with a time bound of $n^{3/2}$.

Intuitively, the idea is the following. The earlier flowchart schema spends most of its time trying to find the inverse of the function f (i.e., given $f^i(a)$, to find $f^{i-1}(a)$) -- though this operation is somewhat hidden in the program. We can speed up this by planting a value at a "distance" of about $\sqrt{n}$ from the end and compute inverses from this planted value. Time taken to find the square root is of the order of $n^{3/2}$, average time to find the inverse is $n^{1/2}$ (done n times) and time to reset the planted value is of the order of n (done $n^{1/2}$ times). In general, by planting (k-1) values (instead of just one) at distances

$$n^{1/k} , n^{2/k} , n^{3/k} , \ldots , n^{(k-1)/k}$$

from the end we get a time bound of $n^{1+(1/k)}$.

```
        START
        y ← a;
(1) --  while p(y) do y ← f(y);
(2) --  if y = a then HALT(a);
        x ← f(a);
(3) --  CHECK: y1 ← y2 ← a;
        while y1 ≠ x do
            begin
                y2 ← a;
                while y2 ≠ x do
                    begin
                        y2 ← f(y2) ;
                        Y3 ← f(y3);
                        if y3 = y then goto FOUND;
                    end;
                y1 ← f(y1) ;
            end;
        x ← f(x);
(4) --  goto CHECK;
        FOUND :  z ← y;                  --  x = f^m(a)
        x2 ← x;
(5) --  REPEAT: x1 ← a;
        while x2 ≠ z do
            begin
            x1 ← f(x1) ;
            x2 ← f(x2) ;
(6) --      end;
```

```
(7) --  while $z \neq x_1$ do
            begin
            $x_3 \leftarrow x_1$;
            while $f(x_3) \neq z$ do $x_3 \leftarrow f(x_3)$;
            $y \leftarrow g(y,x_3)$;
            $z \leftarrow x_3$;
(8) --      e ;

        TEST:  if $z = a$ then HALT(y);
(9) --  $x_2 \leftarrow a$;  while $(x_2 (\neq_2 z)$   $\neq$   $x)$
                do $x_2 \leftarrow f(x_2)$;
        goto REPEAT .
```

Line (1) detects if there exists an $n \geq 0$ such that $f^n(a)$ = FALSE and $f^k(a)$ = TRUE for all $k < n$. If such an n does not exist the program loops forever which is the desired operation. If n exists it follows that for all $i,j \leq n$, if $i \neq j$ then $f^i(a) \neq f^j(a)$. At this point $y = f^n(a)$.

If n = 0 the program halts with output a (line 2). If $n \geq 1$ the CHECK loop segment of the program from lines (3) to (4) finds the smallest positive integer m such that $m*m \geq n$. This is done by successively trying larger and larger values $i = 1,2,3,\ldots$ for m until one'is found such that $i*i \geq n$. This is the required value for m. We use the variable x to store the value of $f^i(a)$ and the variable y_3 to "count" up to $i*i$ by successively taking values $a, f(a), \ldots, f^{i*i}(a)$. The final value of x is $f^m(a)$ and it remains unchanged for the rest of the program.

Execution of lines (5) to (6) now causes the variable x_1 to be "planted" at $f^{n-m}(x)$. The while statement between lines (7) and (8) constitutes the main part of the program. The variable y takes on values in the sequence

$$f^n(a) ,$$
$$g(f^n(a), f^{n-1}(a)) ,$$
$$g(g(f^n(a),f^{n-1}(a)),f^{n-2}(a)) ,$$
$$\vdots$$
$$g(g(\ldots g(f^n(a),f^{n-1}(a)), \ldots),f^{n-m}(a))$$

On exit from this while-loop the value of z is $f^{n-m}(a)$.

Lines (9) and (5) to (6) are then used to reset the planted value to $f^{n-2m}(a)$ and the process is repeated. After it, the planted value is reset to $f^{n-3m}(a)$, and so on. A special case is encountered when the integer corresponding to z becomes less than m. In this case, the next planted value should be simply a, and hence the use of line (9) instead of simply setting $x2 \leftarrow x$.

5. Appendix B -- Proof of Theorems

We use the terminology $T_1 \equiv T_2$ to mean the schemas T_1 and T_2 are equivalent, and $T_1 \supset T_2$ to mean T_1 includes T_2.

Proof of Theorem 1 (Solvability of S_1)

1(a),(b),(c): The solvability of the halting, divergence and equivalence problems follows from the solvability of inclusion:

(a) Given a schema T of S_1, T halts if and only if $T' \supset H$ where H represents the schema [START; HALT(a)] that always halts with output a, and T' is the schema T with all HALT statements changed to HALT(a) .

(b) Given a schema T of S_1, T diverges if and only if $L \supset T$, where L represents the schema [START ; LOOP] that always loops,

(c) Given two schemas T_1 and T_2 of S_1, $T_1 \equiv T_2$ if and only if $T_1 \supset T_2$ and $T_2 \supset T_1$.

1(d): We give below only the intuitive idea behind the proof of solvability of the inclusion problem. Given two schemas T_1 and T_2 of S_1, to decide if $T_1 \supset T_2$, an automaton is constructed that simulates the computations of T_1 and T_2 in parallel. The input tape of the autunaton represents an interpretation for T_1 and T_2. The input tape is rejected if T_1 and T_2 both halt but with different outputs, or if T_2 halts and T_1 diverges, under the interpretation corresponding to the input tape; otherwise, the tape is accepted.

To describe the operation of the automaton we first introduce the notion of the "specification state" of a variable y . The specification state represents the outcomes of all possible tests that could be performed by a schema without changing the value of the variable y (and using terms no "larger" than the "largest" term used in the schemas T_1 and T2). The automaton simulates the computations of T_1 and T2 not just for the main-line computation, but for a large number of "instances" of the variable y . There is one instance for each assignment statement and each constant term (no larger than the largest term). The computation of an instance (for an assignment statement and a term) represents what the schema would really do if its main-line variable happened to equal that constant term after that assignment statement.

The computation on each instance is kept in step, and the automaton keeps track of which instances have equal values at each step. This enables the automaton to detect whether the input tape really represents a feasible interpretation.

The reason that this specification state approach works with limited equality tests is that the finite specification state carries sufficient information to allow it to be updated. This is not true for general equality tests, e.g. in the classes S_3 and S_4, if a specification state were to carry all information necessary to update it, the amount of information would grow without bound as the computation proceeded.

1(e): The proof of isomorphism is similar to the proof of inclusion, except that the automaton not only keeps track of which instances are equal in value at each step, but also which equal instances have an isomorphic history. The automaton can then detect if for any input tape the computations of the two schemas are not isomorphic.

1(f): Freedom or nonfreedom is detected by the algorithm 1(g) that translates a given schema in S_1 to an equivalent free schema; if ever a test statement is detected for which some exit is not feasible the schema is not free, else it is free.

1(g): We give below a short outline for the translation of a given schema T in S_1 to an equivalent free schema T_1 (using several variables).

A "partial specification state" is like a specification state but with the possibility that the values of certain predicate and equality tests may be unknown. The schema T_1 has a (large) number of variables, one variable for each assignment statement and each constant term (no larger than the largest term used in T).

The schema T_1 begins by assigning all variables their corresponding initial values. The schema T_1 has a (large) number of "chunks" of statements. Each chunk updates the variables. This corresponds to one step of the automaton in the proof of inclusion. This updating can be performed without introducing any nonfreedom. Each chunk is associated with the following information (line (iii) is unnecessary for this problem, but it is required to solve the freedom problem).

(i) The statement in T corresponding to each variable in T_1.

(ii) Which variables have equal values.

(iii) Which pairs of variables have the property that they both would have tested the same value if we hadn't explicitly avoided that (i.e., if both variables are "entered" by the main-line computation, nonfreedom would result).

When updating is performed, no predicate or equality test is introduced whose outcome is known from the information corresponding to the chunk. Loops are detected as before; and some variables may become "inactive" either by looping or halting.

Proof of Theorem 2 (Solvability of S_2)

The proof of Theorem 2 is similar to the proof of Theorem 1 except that the formal definition of the specification state reflects the different kind of equality tests allowed.

Proof of Theorem 3 (Unsolvability of S_3)

3(a),(b): We define a class S_5 of schemas having two variables y_1 and y_2, and whose statements consist of the following:

Start statement: START
$y_1 \leftarrow y_2 \leftarrow a;$

Final statements: HALT or
LOOP

Test statement: $Y \leftarrow f(y_i);$
if $p(y_i)$ **then goto** L_j
else goto L_k;

It was shown by Luckham, Park and Paterson [1970] that the halting problem for the class S_5 is unsolvable, and that the divergence problem is not partially solvable.

To show the halting problem for S_3 to be unsolvable we reduce the halting problem for S_5 to that for S_3; that is, we describe an algorithm that takes any schema T_5 in the class S_5 as input and yields a schema T_3' in the class S_3 such that T_3' halts if and only if T_5 . halts. Similarly, to show that the divergence problem for S_3 is not partially solvable we describe an algorithm that takes T_5 as input and yields as output a schema T_3'' in the class S_3 such that T_3'' diverges if and only if T_5 diverges. We will unify the construction for the two cases by constructing for both cases a schema T_3 in the class S_3 but augmented with a special final statement called the REJECT statement:

REJECT statement: REJECT .

The REJECT statement signifies that the interpretation is unacceptable and is rejected. Loosely the idea is the following. There exists a map from interpretations of T_3 that are not rejected onto the interpretations of T_5 such that the computation for T_3 under an interpretation halts if and only if the computation for T_5 under the corresponding interpretation halts.

Now it is clear that if we replace all REJECT statements in T_3 by HALT statements to get T_3', then T_3' halts on every interpretation if and only if T_5 halts on every interpretation. Similarly, if we replace all REJECT statements by LOOP statements to get T_3'' then T_3'' diverges on every interpretation if and only if T_5 diverges on every interpretation.

Given a schema T_5 in S_5 we construct the corresponding schema T_3 in S_3 (with the addition of REJECT statements) as follows. We use the variable y of T_3 to represent the latest variable tested in T_5, i.e., y_1 or y_2. The function f plays the same role in T_3 as in T_5. We use a new function g called a "test function"; and tests of the form

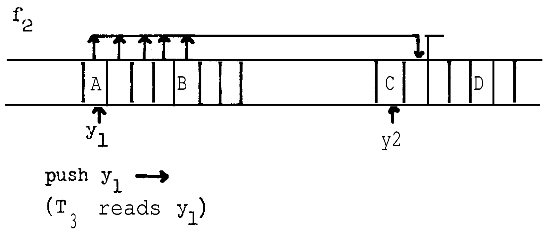
if $p(y)$ then . . . else . . .

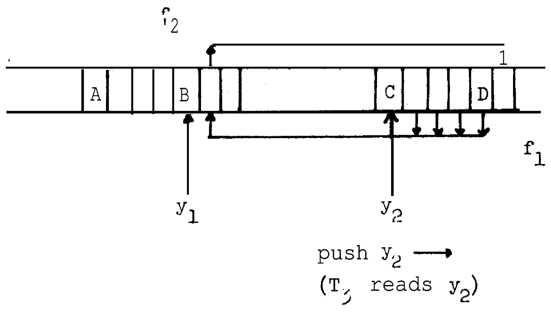
in T_5, will take the form

if $g(y) = g(g(y))$ then . . . else . . .

in T_3. In addition we use two "control" functions f_1 and f_2. Their roles are the following: if y stands for y_2 (of S_5) then $f_1(y)$ will equal the value of $f(y_1)$ at that instant in the computation unless, of course, a REJECT statement is reached earlier. The role of f_2 is analogous, i.e., if y stands for y_1 then $f_2(y)$ will equal the value of $f(y_2)$.

The schema T_3 simulates a computation of T_5 as follows. In the diagram below the elements a, $f(a)$, $f(f(a))$, $f(f(f(a)))$ are represented by contiguous squares from left to right. We superimpose on this diagram the computations of both T_3 and T_5. Suppose, at some instant in the computation of T_5, y_1 is at point A, and y_2 is at C, and suppose y_1 is being "read". T_3 makes certain that the f_2 pointers from the squares scanned, point to the right of y_2. Suppose that we continue to "read" from y_1 until y_1 reaches point B where the schema T_5 starts "reading" from y_2. T_3 checks that the f_1 pointers from the squares scanned, point to the right of B.





We are now in a position to describe the construction of T_3. Without loss of generality, we will assume that in T_5 the first test statement tests the variable y_1. T_3 will effectively contain 2 copies of T_5 except there is only one start statement. We will call these copies A and B. We will label statements of T_5 by numbers 1,2,3,... . The corresponding statements in T_3 will be labelled 1-A, 1-B, 2-A, 2-B, 3-A, 3-B, ...

(i) The start statement in T_5 is

```
START
$y_1 \leftarrow y_2 \leftarrow a$;
goto i;
```

The corresponding statements in T_3 are:

```
START
$y \leftarrow a$;
if $f(y) \neq f_2(y)$ then REJECT else goto i-A;
```

Note that the test $f(y) \neq f_2(y)$ is not strictly an allowed statement. We use this form for clarity: it can really be "simulated" by the statements:

```
if $f(y) \neq f_1(f_1(y))$ then REJECT;
if $f_2(y) \neq f_1(f_1(y))$ then REJECT
                else goto i-A;
```

(ii) For any test statement i in T_5, if i is of the form:

```
i: $y_1 \leftarrow f(y_1)$;
   if $p(y_1)$ then goto j else goto k;
```

the corresponding statements i-A and i-B are:

```
i-A: if $f_2(y) \neq f_2(f(y))$ then REJECT;
     $y \leftarrow f(y)$;
     if $g(y) = g(g(y))$ then goto j-A
                  else goto k-A;
and
i-B: if $f(y) \neq f_2(f_1(y))$ then REJECT;
     $y \leftarrow f_1(y)$;
     if $g(y) = g(g(y))$ then goto j-A
                  else goto k-A;
```

(iii) For any test statement i in S of the form:

```
i: $y_2 \leftarrow f(y_2)$;
   if $p(y_2)$ then goto j else goto k;
```

i-A and i-B are similar to the above, except, one has to interchange f_1 with f_2 and A with B.

(iv) HALT and LOOP statements remain unchanged.

This completes the construction.

The main reason that the schema T_3 can simulate the computation of T_5 is that each f_1, f_2 "pointer" is checked at most once from each square. If pointers were to be checked twice and it turned out that they were required to point to different values there might exist no interpretation satisfying this condition -- the result would be that all interpretations of T_3 would be rejected.

3(c): The non-partial solvability of the equivalence problem follows directly fran the non-partial solvability of the divergence problem (Part (b)), since a program schema in S_3 diverges if and only if it is equivalent to the schema:

```
START
y ← a;
LOOP .
```

3(d): The non-partial solvability of the inclusion problem follows immediately from the non-partial solvability of the equivalence problem since $T_1 \equiv T_2$ if and only if $T_1 \supset T_2$ and $T_2 \supset T_1$.

-S(e): The non-partial solvability of the isomorphism problem also follows directly from the non-partial solvability of the divergence problem. Given a schema T in the class S_3 , construct a new schema T' also in S_3 obtained by replacing each HALT statement in S_3 by the statements:

```
y ← f(y) ;
HALT .
```

Then T and T' are isomorphic if and only if T diverges.

3(f): The non-partial solvability of the freedom problem is shown by reduction of Post's Correspondence Problem for nonempty strings (PCP) to the nonfreedom problem for schemas in S_3, The proof follows along lines similar to a related proof in Paterson [1967] with the mechanism for effectively simulating two variables while using only one (as described in the proof of Z(a),(b)).

3(g): There can exist no effective translation to a free schema since if there did exist such an algorithm we could decide whether or not a given schema of S_3 halts since the halting problem for free schemas is trivially solvable.

Proof of Theorem 4 (Unsolvability of S_4)

The proof goes along lines quite similar to the proof for Theorem 3. We first define a subset S_6 of the class of schemas S_5 . S_6 , like S_5 , has two variables y_1 and y_2 , one function symbol f , and one predicate symbol p . However, S_6 has the constraint that in any path through a schema of S_6 , after each statement that tests the variable y_1 there must be either one or two statements that test y_2 (followed by a final statement or another test of y_1) -- note the form of the test statement of S_5 defined in the proof of 3(a),(b). The halting and divergence problems of S_6 can be shown to be unsolvable, and the halting and divergence problems of S_6 can be reduced to those of S_4 . This implies the unsolvability of problems (a)-(e) and (g) for S_4 . The freedom problem (f) can be shown to be unsolvable on lines similar to the proof for 3(f), i.e., by reducing PCP to the non-freedom problem and effectively simulating two variables while actually using only one.

Proofs of Secondary Results

In the following results the number of functions does not include the individual constants.

(i) Schemas with One Variable, Two Functions and General Equality Tests

The class of flowchart schemas with one variable, two functions (no predicates) and general equality tests is unsolvable.

If completely general equality tests are allowed it is easy to see that two function constants suffice to render the class of schemas unsolvable because more function letters can be "coded" in terms of two functions. For example, in 3b we could use only two functions f and g by making in the construction of T_3 from T_5 the following substitutions: for all terms t simultaneously substitute:

$f(f(t))$	for	$f(t)$
$f(g(t))$	for	$g(t)$
$g(f(t))$	for	$f_1(t)$
$g(g(t))$	for	$f_2(t)$

All the unsolvability results go through on making this substitution. Similar substitutions can be made to show the unsolvability of freedom.

(ii) Schemas with Two Variables, Two Functions and Restricted Equality Tests

The class of flowchart schemas with two variables and two functions (no predicates) with tests only of the form $y_i = f(y_i)$ are unsolvable.

Consider the class S_7 which is the same as S_5 but with the difference that there are two function constants f_1 and f_2 , and no predicate constant.

The computation of any schema T_5 in S_5 can be simulated by a corresponding schema T_7 in S_7, obtained by replacing every test statement of the form

```
y_i ← f(y_i) ;
if p(y_i) then goto L_j else goto L_k
```

by a test statement of the form

```
y_i ← f(y_i);
if y_i = g(y_i) then goto L_j else goto L_k .
```

It is easy to see that for any path, finite or infinite, through T_5, if there exists an interpretation for which T_5 executes statements along this path, then there is an interpretation for which T_7 executes statements along the corresponding path. This establishes the unsolvability of (a)-(e) and (g) for the class S_7 (note that the unsolvability of (c)-(e) and (g) follows from the unsolvability of (b)).

Further, the freedom problem too can be shown to be unsolvable by reducing FCP to it. The reduction is related to the corresponding reduction in Paterson [1967], but to do it with 2 function symbols we need the additional "cleverness' of padding each symbol of the PCP with enough "bits" in order to allow for testing, to effect a non-deterministic search.

(iii) Schemas with One Function, Restricted Equality Tests

Schemas with one function using tests only of the form $y_i = y_j$ are unsolvable.

The halting and divergence problems for two-counter automata are known to be unsolvable (Hopcroft and Ullman [1969]), and can be reduced to the halting and divergence problems for one-function schemas in a rather direct manner. In the reduction process the only care that has to be taken is for the operation of incrementing one to a counter, in which case the schema checks for a looping interpretation as in Example 3 of Appendix A. The unsolvability of the equivalence, inclusion, and isomorphism problems follows from the unsolvability of the halting and divergence problems.

6. References

Ashcroft, Manna and Pnueli [1971] -- E. Ashcroft, Z. Manna and A. Pnueli, "Decidable properties of monadic functional schemas", in Theory of Machines and Computations (Kohavi and Paz, Eds.), Academic Press, pp. 3-18.

Chandra [1972a] -- A. K. Chandra, "Efficient compilation of linear recursive programs", Report, Computer Science Dept., Stanford Univ. (to appear).

Chandra [1972b] -- A. K. Chandra, "Properties and applications of program schemas", Ph.D. Thesis, Computer Science Dept., Stanford Univ. (to appear).

Constable and Gries [1971] -- R. L. Constable and D. Cries, "On classes of program schemata", Report, Computer Science Dept., Cornell Univ. (August 1771).

Garland and Luckham [1971] -- S. J. Garland and D. C. Luckham, "Program schemes, recursion schemes, and formal languages", UCLA report (June 1971).

Hewitt [1970] -- C. Hewitt, "More comparative schematology", A.I. Memo 207, Project MAC, M.I.T. (August 1970).

Hopcroft and Ullman [1969] -- J. E. Hopcroft and J. D. Ullman, "Formal languages and their relation to automata", Addison-Wesley, 1969.

Ianov [1960] -- Y. I. Ianov, "The logical schemes of algorithms". English translation in Problems of Cybernetics, Vol. 1, Pergamon Press, New York, 1960, pp. 82-140.

Luckham, Park and Paterson [1970] -- D. C. Luckham, D. M. R. Park and M. S. Paterson, "On formalized computer programs", J. of Computer and System Science, Vol. 4, No. 3 (June 1970), pp. 220-249.

Paterson [1967] -- M. S. Paterson, "Equivalence problems in a model of computation", Ph.D. Thesis, University of Cambridge, England (August 1967). Also A.I. Memo No. 1, M.I.T. (1970).

Paterson [1968] -- M. S. Paterson, "Program schemata", in Machine Intelligence 3 (Michie, Ed.), Edinburgh Univ. Press, pp. 19-31.

Paterson and Hewitt [1970] -- M. S. Paterson and C. E. Hewitt, "Comparative schematology", in Record of Project MAC Conference on concurrent systems and parallel computation, ACM, New York (December 1970), pp. 119-128.

Plaisted [1972] -- D. Plaisted, "Program schemas with counters", Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing, Denver, Colorado (May 1972).

Rutledge [1964] -- J. D. Rutledge, "On Ianov's program schemata", J.ACM, Vol. 11, No. 1 (January 1964), pp. 1-7.

Strong [1971a] -- H. R. Strong, "Translating recursion equations into flowcharts", J. of Computer and System Science, Vol. 5 (June 1971), pp. 254-285.

Strong [1971b] -- H. R. Strong, "High level languages of maximum power", IBM Research Report.