PROGRAM SCHEMAS WITH SEMANTIC
RESTRICTIONS

by: Elaine Jessica Weyuker
DCS-TR-60

Dissertation for a Ph.D.
in Computer Science

Department of Computer Science
Hill Center
Rutgers University, New Brunswick
N.J.

May 1977
ABSTRACT

PROGRAM SCHEMATA WITH SEMANTIC RESTRICTIONS

by ELAINE JESSICA WEYUKER

Thesis director: Professor Ann Yasuhara

Paterson introduced the notions of freedom and liberality as semantic restrictions on the class of schemas. He felt that such restricted classes of schemas might have solvable decision problems, and also be realistic models of what would generally be considered "good" programs. With the latter viewpoint in mind, two new classes of schemas are introduced in this thesis: the reachable schemas and the semifree schemas. Several decision problems for these classes, and translatability between these classes and other semantically restricted schema classes are studied. Relationships between various classes of schemas are also considered.

An investigation is made of the preservation of semantic properties by isomorphism, functional similarity, strong equivalence, and weak equivalence. This allows a distinction to be made between properties which are inherent in the class of functions represented by a schema, regardless of the algorithm used to compute it, and those properties which are dependent upon the algorithm or its encoding.
Two types of interpretations for schemas are defined: pointwise interpretations and function interpretations. The relationships between these types of interpretations are studied. It is seen that corresponding to the set of all finite consistent paths of a schema executed under pointwise interpretations, there is a single effectively constructable function interpretation of that schema which has the same set of finite consistent paths. The equivalence of certain semantic properties of schemas follows from this result. This result also facilitates the consideration of the inheritance of schema properties by programs, and the inheritance of program properties by schemas.

Finally, several classes of programs which are the analogues of the schema classes are introduced. A study is made of translatability and decidability questions for these classes of programs, and it is demonstrated that results similar to those for schemas hold in most cases.
ACKNOWLEDGEMENTS

I am deeply grateful to Ann Yasuhara for her guidance, patience, and helpful discussions and suggestions. I am also indebted to Tom Ostrand for his constant encouragement, support, friendship, and willingness to listen and give advice.

My special thanks go to my parents for their love, encouragement, and understanding, and for providing the type of environment in which one grows up wanting to learn.
Table of Contents

1 Introduction and Definitions 1
2 Reachable Schemas 24
3 Semifree Schemas 44
4 Relationships Between Schema Classes 69
   4.1 Free and Liberal Schemas 74
   4.2 Relationships With Reachable Schemas 87
      4.2.1 Free Schemas 87
      4.2.2 Liberal Schemas 90
      4.2.3 Schemas Which Halt Under Every Pointwise Interpretation 93
      4.2.4 Semifree Schemas 96
   4.3 Relationships With Semifree Schemas 107
      4.3.1 Free Schemas 107
      4.3.2 Liberal Schemas 108
      4.3.3 Schemas Which Halt Under Every Pointwise Interpretation 110
5 Preservation of Semantic Properties by Sameness Relations 112
   5.1 Isomorphism 112
   5.2 Functional Similarity 115
   5.3 Strong Equivalence 116
   5.4 Weak Equivalence 118
   5.5 Decidability Results 120
6 Pointwise and Function Interpretations 122
7 Restricted Classes of Programs 138

7.1 Introduction 138

7.2 Reachable Programs 139

7.3 Semifree Programs 147

7.4 Free Programs 151

7.5 Programs Which Halt For Every Input 155

8 Concluding Remarks 162

References 167
CHAPTER 1

INTRODUCTION AND DEFINITIONS

A substantial amount of work has been done recently using program schemas as an abstract model of computation. Several authors have added data structures to the original model of Luckham, Park, and Paterson, with the intent of comparing the relative computational power of these augmented classes of schemas. Other authors have tried placing restrictions on the form that schemas may take. Both semantic and syntactic restrictions have been considered.

Paterson, for example, introduced the notion of progressive schemas, a type of schema with severe syntactic restrictions. Such limitations tend to seriously restrict the computational power of the class of schemas, and also significantly affect the degree to which the model realistically reflects properties of computer programs. Frequently, however, the added restrictions do have the positive affect of making the commonly considered decision problems solvable.

Paterson also introduced the notion of free schemas. In this case semantic restrictions are placed on the form of the schema. Not only are these restrictions sufficient to cause some of the decision problems which are unsolvable in the unrestricted model to be solvable,
but, Paterson also argued, this restricted class is really a better model of what one generally considers "good" or "efficient" programs than the unrestricted class.

Much of the work which follows was motivated by a similar desire to model "good" computer programs, and to determine under what circumstances one can detect whether or not a program possesses these desirable properties. We are also interested in determining under what circumstances programs lacking these characteristics can be automatically improved. We have found, in general, the same type of unsolvability results for decision problems, both for schemas and programs, that have been encountered in much of the previous work done with classes of schemas. Some of our results particularly highlight difficulties which seem inherent in the area of algorithmic program optimization or improvement. We also feel that some of our results have interesting implications which may be useful when considering program verification.

We will be using a program schema model based largely on the one formulated by Luckham, Park, and Paterson.

We have a formal language whose alphabet consists of the following sets of symbols:

(i) **Variable** or **Location** Symbols, denoted by the letters
u, v, w, x, y, z. The set of variables is divided into three disjoint subsets \(X, Y, \text{ and } Z\). The set \(X\) contains the input variables, \(Y\) is the set of program variables, and \(Z\) is the set of output variables. The value of a variable in \(X\) may be retrieved but never changed, whereas an element of \(Z\) may be assigned a value, but may never be retrieved from. Elements of \(Y\) may either be retrieved from or assigned to as long as the program variable has been assigned a value before it is retrieved from or tested.

(ii) Function Symbols, denoted by the letters \(f, g, h\).

(iii) Predicate Symbols, denoted by the letters \(p, q, r, s, t\).

(iv) Distinguished Symbols: START, HALT, (, ), \(\leq\), numerals, comma, TRUE, FALSE, (we sometimes use YES, NO, or 1, 0, or T, F).

Each of the symbols in i, ii, and iii may appear with or without a subscript.

The language has six types of statements:

(i) Start Statement

\[
\text{START}
\]

(ii) Assignment Statement

\[
y \leftarrow f(y_1, \ldots, y_n)
\]

where \(y_1, \ldots, y_n\) are elements of \(X \cup Y\), \(y\) is an element of \(Y \cup Z\), and \(f\) is an \(n\)-ary function symbol.

(iii) Test Statement

\[
p(y_1, \ldots, y_n)
\]

where \(y_1, \ldots, y_n\) are elements of \(X \cup Y\) and \(p\) is an \(n\)-ary predicate symbol.
(iv) **Input Statement** \( y \leftarrow x \)

where \( x \in X \) and \( y \in YUZ \).

(v) **Output Statement** \( z \leftarrow y \)

where \( y \in XUY \) and \( z \in Z \).

(vi) **Halt Statement** HALT

The above six types of statements are the **legal statements**.

A **flow diagram** is a labelled directed graph each of whose vertices is labelled by a legal statement and such that:

(i) A vertex labelled with a start statement has no edges entering it and one edge exiting from it.

(ii) A vertex labelled with an assignment statement, an input statement, or an output statement, has at least one edge entering it and exactly one edge exiting from it.

(iii) A vertex labelled with a test statement has at least one edge entering it and exactly two edges exiting from it which are labelled TRUE and FALSE (or \( T \) and \( F \), \( 1 \) and \( 0 \), or YES and NO). Occasionally we will extend our notation to include \( n \)-exit tests. This is simply a matter of notational convenience and could equally be represented by a series of \( n-1 \) two-exit tests.

(iv) A vertex labelled with a halt statement has at least one edge entering it and no edges exiting from it.
A node \( n \) of a flow diagram is the successor of a node \( m \) if there is an edge which leaves \( m \) and enters \( n \).

A node \( n \) of a flow diagram is the 0-successor of a node \( m \) if \( n \) is a successor of \( m \) and the edge from \( n \) to \( m \) is labelled 0. A 1-successor is defined analogously.

A trail \( T \) through a flow diagram \( P \) from a start statement is a sequence each of whose elements, \( T(i) \), is either \( s_k \) or a pair \((B,s_k)\), where \( s_k \) is a legal statement and \( B \in \{0,1\} \). It is defined inductively as follows:

(i) \( T(0) = \text{START} \)

(ii) If \( T(i) \) is a start statement, an assignment statement, an input statement, or an output statement, then \( T(i+1) \) is the statement which is its unique successor.

(iii) If \( T(i) \) is a test statement, then \( T(i+1) \) is a pair of the form \((B,s_k)\) for \( B \in \{0,1\} \) and \( s_k \) a legal statement. Then either

\[ T(i+1) = (0,s_k) \text{ for } s_k \text{ the 0-successor of } T(i), \text{ or} \]

\[ T(i+1) = (1,s_k) \text{ for } s_k \text{ the 1-successor of } T(i). \]

(iv) If \( T(i) \) is a halt statement, then the trail is finite, of length \( i+1 \), and \( T(i+1) \) is undefined.

A path through a flow diagram \( P \) is any not necessarily proper initial segment of a trail through \( P \).

A path will be denoted by \( s \), and \( s(i) \) will denote the \( i \)-th element of path \( s \). When we are speaking of an
element \( s(i) \) of a path \( s \) which is a pair consisting of
the outcome of the preceding test and a statement, we
will usually speak of \( s(i) \) being a statement rather than
the more precise expression that \( s(i) \) contains a
statement.

We note that we have defined paths in this somewhat
awkward manner in order to allow us to distinguish
between the two different routes which may be taken from
the test statement to the assignment statement in the
following example:

![Diagram](image)

This type of distinction will be of particular
importance when we consider free schemas, and is in fact
necessary in order to allow us to have a structural
characterization of freedom as well as a semantic one.

A program schema \( P \) is a finite flow diagram with the
following restrictions:
(i) There is exactly one vertex labelled START.
(ii) There are zero or more vertices labelled HALT. We
note that a schema usually contains at least one halt
statement (i.e. node labelled HALT) but occasionally it
is instructive to consider schemas which contain no halt
(iii) Each vertex lies on a path from the vertex labelled START.

(iv) On every path from the start statement, if \( u \) is a variable in \( YUZ \), then \( u \) is assigned a value before it is retrieved from or tested.

We note that the third restriction guarantees that we will never be considering a disconnected graph.

We shall use \( \mathcal{P} \) to denote the class of all schemas. We note that our definition does not include constants or interpreted function or predicate symbols (i.e., function or predicate constants) and does not permit the direct assignment of a variable value to another variable except in the case of input or output variables. We allow the value of an input variable to be assigned directly to a program or output variable, and allow the value of a variable other than an output variable to be assigned directly to an output variable.

As a convention we will use the variable \( x \), with or without subscripts, to denote input variables, and the variable \( z \), with or without subscripts, to denote output variables. Program variables are usually denoted by a \( y \).

We could alternately represent a program schema as a linear sequence of instructions with address labels, much like a conventionally written program, rather than as a
graph. This is the method of representation originally used by Luckham, Park, and Paterson[6]. We shall sometimes find it convenient when discussing schemas to represent them in this fashion and hence we introduce the following parallel definitions.

Our language has six types of statements, known as the legal statements.

(i) **Start statement**  
    k. \( y \leftarrow f(y_1, \ldots, y_n) \)

(ii) **Assignment statement**  
    k. \( y \leftarrow f(y_1, \ldots, y_n) \)

where \( k \) is a numeral denoting the address of the statement, \( y_1, \ldots, y_n \) are elements of \( X \cup Y \), \( y \) is an element of \( Y \cup Z \), and \( f \) is an \( n \)-ary function symbol.

(iii) **Test statement**  
    k. \( p(y_1, \ldots, y_n) \leftarrow l, r \)

where \( k, l, \) and \( r \) are numerals, \( y_1, \ldots, y_n \) are elements of \( X \cup Y \), and \( p \) is an \( n \)-ary predicate symbol. \( k \) denotes the address of the statement, while \( l \) and \( r \) denote the addresses of the next instruction which may be executed, called the transfer addresses.

(iv) **Input statement**  
    k. \( y \leftarrow x \)

where \( k \) is a numeral denoting the address of the statement and \( x \in X, y \in Y \cup Z \).

(v) **Output statement**  
    k. \( z \leftarrow y \)

where \( k \) is a numeral denoting the address of the statement and \( y \in X \cup Y, z \in Z \).

(vi) **Halt statement**  
    k. HALT

where \( k \) is a numeral denoting the address of the halt
A program schema \( P \) is a finite sequence of statements such that

(i) The schema begins with the statement 0. START
(ii) The address of each statement is its position in the sequence
(iii) All transfer addresses are addresses in \( P \)
(iv) The last statement is either a test statement or a halt statement
(v) Each statement is included in some path from the start statement
(vi) On every path from the start statement, if \( u \in YUZ \), then \( u \) is assigned a value before it is retrieved from or tested.

Throughout this thesis, we shall frequently use the expression "a statement \( k \)" or "a statement \( s_k \)". When we do, we are speaking of the actual statement, not simply the label or address of the statement.

Occasionally in this work, we shall extend our language to include constants, constant functions, or constant predicates. This will be done when the result differs from the general case, or because the method or form of the proof is particularly instructive. \( \mathcal{O}_C \) will be used to denote the class of all schemas over the language extended to include constants, while \( \mathcal{O}_E \) will be
used when the language has been extended to include equality tests. \( \mathcal{D}_{C_2} = \) will denote the class when both constants and equality tests have been added.

Now that we have defined the syntax of a program schema, we shall next discuss the semantics. We shall distinguish between two types of interpretations. The first type does not include an assignment of initial values to input variables. This type of interpretation, together with a schema, will constitute what we normally think of as a program. The second type of interpretation includes the assignment of initial values to input variables, and is the type of interpretation introduced by Luckham, Park, and Paterson in [6].

A function interpretation \( \mathcal{I} \) of a program schema \( P \) consists of:

1. A nonempty set of elements \( D \) called the domain.
2. An assignment to each \( n \)-ary function symbol \( f, n \geq 1 \), of a total \( n \)-ary function \( (\mathcal{I}f):D^n \to D \).
3. An assignment to each \( n \)-ary predicate symbol \( p, n \geq 1 \), of a total \( n \)-ary characteristic function \( (\mathcal{I}p):D^n \to \{0,1\} \).

A pointwise interpretation \( I \) of a program schema \( P \) consists of the three items of a function interpretation as well as:
(4) An assignment to each input \( x \) of an element \( I(x) \) \( \Sigma \).

We shall extend our notation so that \( I(\Sigma) \) describes the application of a pointwise interpretation \( I \) to a term \( \Sigma \) or an atomic formula as follows:

1. If \( \Sigma \) is \( x \), then \( I(\Sigma) \) denotes \((I x)\).
2. If \( \Sigma \) is of the form \( f(\Sigma_1, \ldots, \Sigma_n) \) where \( f \) is an \( n \)-ary function symbol, then \( I(f(\Sigma_1, \ldots, \Sigma_n)) \) denotes \((I f)(I \Sigma_1, \ldots, I \Sigma_n)\).
3. If \( \Sigma \) is of the form \( p(\Sigma_1, \ldots, \Sigma_n) \), where \( p \) is an \( n \)-ary predicate symbol, then \( I(p(\Sigma_1, \ldots, \Sigma_n)) \) denotes \((I p)(I \Sigma_1, \ldots, I \Sigma_n)\).

We shall use the symbol \( \mathcal{I} \), with or without subscripts or superscripts, to denote a function interpretation, and \( I \) to denote a pointwise interpretation.

We now introduce the notions of an execution sequence and the computation of a program schema \( P \) relative to an interpretation. Informally, an execution sequence of \( P \) under a pointwise interpretation \( I \) or a function interpretation \( \mathcal{I} \) with input \( d \), is the sequence of instructions executed under that interpretation. This is a path through the flow diagram when that representation is used. We will also compute the sequence of vectors of values assigned to the variables of \( P \) when \( P \) is executed under the specified
interpretation. This sequence will be called a computation.

Formally we have the following definitions.
We let $\sigma(P,I)$ denote the execution sequence of schema $P$
under pointwise interpretation $I$, and $\sigma(P,I,\langle i \rangle)$, $i \in \mathbb{N}$,
denote the $i$-th element of $\sigma(P,I)$. $\text{val}(y,i)$ denotes the
value of the variable $y$ after $\sigma(P,I,\langle i \rangle)$ has been
executed. Thus, using a linear representation of the
schema:

(i) $\sigma(P,I,\langle 0 \rangle) =$ START

$\sigma(P,I,\langle 1 \rangle)$ is the statement with address 1.

\[
\text{val}(u,0) = \begin{cases} 
I(u) \text{ if } u \text{ is an input variable} \\
\lambda \text{ otherwise}
\end{cases}
\]

where $\lambda$ indicates that a non-input variable has not yet
been assigned a value.

(ii) If $\sigma(P,I,\langle i+1 \rangle)$ is the assignment statement

$k \cdot y \leftarrow f(y_1, \ldots, y_n)$

then $\sigma(P,I,\langle i+2 \rangle)$ is the statement with address $k+1$.

\[
\text{val}(u,i+1) = \begin{cases} 
\text{val}(u,i) \text{ for } u \neq y \\
(\text{If})(\text{val}(y_1,i), \ldots, \text{val}(y_n,i)) \text{ for } u = y
\end{cases}
\]

(iii) If $\sigma(P,I,\langle i+1 \rangle)$ is the test statement

$k \cdot p(y_1, \ldots, y_n) = 1, r$

then $\sigma(P,I,\langle i+2 \rangle) =$

\[
\begin{cases} 
(0,\#l) \text{ if } (lp)(\text{val}(y_1,i), \ldots, \text{val}(y_n,i)) = 0 \\
(1,\#r) \text{ if } (lp)(\text{val}(y_1,i), \ldots, \text{val}(y_n,i)) \neq 0
\end{cases}
\]
where \#m denotes the statement with label m.
val(u,i+1) = val(u,i) for all variables u.
(iv) If \( \sigma(P,I,(i+1)) \) is the input statement
\[ k. y \leftarrow x \]
then \( \sigma(P,I,(i+2)) \) is the statement with address k+1.
\[ val(u,i+1) = \begin{cases} 
val(u,i) \text{ for } u \neq y \\
val(x,i) \text{ for } u = y 
\end{cases} \]
(v) If \( \sigma(P,I,(i+1)) \) is the output statement
\[ k. z \leftarrow y \]
then \( \sigma(P,I,(i+2)) \) is the statement with address k+1.
\[ val(u,i+1) = \begin{cases} 
val(u,i) \text{ for } u \neq z \\
val(y,i) \text{ for } u = z 
\end{cases} \]
(vi) If \( \sigma(P,I,(i+1)) \) is the halt statement
\[ k. \text{HALT} \]
then \( \sigma(P,I,(i+2)) \) is undefined.
\[ val(u,i+1) = val(u,i) \text{ for all variables } u. \]

The definition of an execution sequence is essentially
the same when the flow diagram representation is used.
In this case the next statement is determined by the
edges leaving the node which is labelled by the statement
currently being executed. Similarly, the values of the
variables are computed in the same manner. If \( \mathcal{L} \) is a
function interpretation, and \( d \) a vector of input values,
then the execution sequence of program schema P under
interpretation $\sigma(I,d)$ for input $d$ is denoted $\sigma(P,\sigma(I,d))$. The definition then proceeds as for the pointwise interpretation case. The values of the variables are also computed in the same manner as for the pointwise interpretation case, except for the assignment of values at stage 0. In this case we have:

$$\text{val}(u,0) = \begin{cases} 
    d_s & \text{if } u \text{ is the } s\text{-th input variable} \\
    \lambda & \text{otherwise} 
\end{cases}$$

We note that as in our definition of path, an element of an execution sequence may be either a statement or a pair consisting of the outcome of the preceding test statement as well as the statement. Again, in the latter case, we shall speak of $\sigma(P,I,(i))$ being the statement rather than containing the statement.

Let $P$ be a program schema which contains $r$ variables, $y_1,\ldots,y_r$. The computation of $P$ under pointwise interpretation $I$, denoted $(P,I)$, is a sequence whose $i$-th element is the vector:

$$(\sigma(P,I,(i)), \text{val}(y_1,i), \ldots, \text{val}(y_r,i))$$

The computation of $P$ under a function interpretation requires that a vector of input values, $d$, be specified. It is defined as above except that the first element of the $i$-th vector is $\sigma(P,\sigma(I,d),(i))$. We denote this computation by $(P,\sigma(I,d))$. Notice that specifying a schema and a function interpretation is not sufficient to define a computation. We call such a pair a program and denote
it by \((P, \omega)\). Once an input vector is specified, we have defined a computation.

The value of the computation of schema \(P\) under pointwise interpretation \(I\), denoted \(\text{val}(P, I)\), is the final vector of values of the output variables of \(P\) if \(\sigma(P, I)\) includes a halt statement. That is, it is the vector of values of the variables of \(Z\) if \(P\) halts under interpretation \(I\). If \(\sigma(P, I)\) does not include a halt statement, \(\text{val}(P, I)\) is undefined. This is denoted by \(\text{val}(P, I) \uparrow\). Similarly, the value of the computation of \(P\) under function interpretation \(\omega\) for input \(d\) is denoted \(\text{val}(P, \omega, d)\) and is defined as for the pointwise case.

We next introduce a particular type of pointwise interpretation, called a free or Herbrand interpretation. The elements of the domain are essentially well-formed terms made up of syntactic elements of the language of the program schema being interpreted. Let \(P\) be a program schema. We define the Herbrand universe of \(P\), denoted \(U(P)\), as follows:

(i) If \(x\) is an input variable appearing in \(P\), then \(x \in U(P)\).

(ii) If \(f\) is an \(n\)-ary function letter, and \(t_1, \ldots, t_n \in U(P)\), then \(f(t_1, \ldots, t_n) \in U(P)\).

An interpretation \(H\) is a free or Herbrand interpretation of a schema \(P\) if:
(i) The domain of $H$ is $U(P)$.
(ii) $H(x) = x$ for every input variable $x$.
(iii) If $f$ is an $n$-ary function symbol of $P$, $n > 1$, then 
$$(Hf):(U(P))^n \rightarrow U(P)$$
is defined by 
$$(Hf)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n),$$
where $t_1, \ldots, t_n \in U(P)$.
(iv) If $p$ is an $n$-ary predicate symbol of $P$, $(Hp)$ is any function such that 
$$(Hp):(U(P))^n \rightarrow \{0, 1\}.$$ 

We next introduce the syntactic notion of the record of a path. This parallels the semantic notion of a computation.

Let $P$ be a program schema, and $\mathcal{L}$ the language for $P$.
Let $U(\mathcal{L})$ denote the set of all well-formed terms on $X$ and the function symbols of $\mathcal{L}$.
Assume $|X|=n$ and $|Y U Z|=m$. Let $b=n+m$.

Let $s$ be a path through $P$.

The record of $s$, denoted $\text{rec}(s)$, is a sequence whose components are either $b$-tuples of elements of $U(\mathcal{L}) \cup \{\lambda\}$, or literals of $\mathcal{L}$.

$\text{rec}(s,i)$ denotes the $i$-th component of $\text{rec}(s)$, $i=0, 1, \ldots$.

We define record inductively, using the notion of a base to facilitate the definition.

For $s(0) = \text{START}$,

$\text{rec}(s,0) = <x_1, \ldots, x_n, \lambda, \ldots, \lambda>.$

If $\text{rec}(s,k) = <\tau_1, \ldots, \tau_b>$ for $\tau_i \in U(\mathcal{L}) \cup \{\lambda\}$
then $\text{base}(s,k) = \text{rec}(s,k)$.

If $\text{rec}(s,k)$ is a literal, then there is a $t<k$ such that
\[ \text{rec}(s,t) = \langle \mathcal{T}_i, \ldots, \mathcal{T}_b \rangle \quad \mathcal{T}_i \in \mathcal{U}(\mathcal{L}) \cup \{ \lambda \}. \]

Let \( v \) be the largest such \( t \).

Then \( \text{base}(s,k) = \text{rec}(s,v) \).

Let \( \text{base}(s,k) = \langle \mathcal{T}_i, \ldots, \mathcal{T}_w, \ldots, \mathcal{T}_b \rangle \).

(i) If \( s(k+1) \) is the assignment statement

\[ u \leftarrow f(u_{i_1}, \ldots, u_{i_h}) \]

then \( \text{rec}(s,k+1) = \langle \mathcal{T}_i, \ldots, f(\mathcal{T}_{i_1}, \ldots, \mathcal{T}_{i_h}), \ldots, \mathcal{T}_b \rangle \).

(ii) Suppose \( s(k+1) \) is the test statement

\[ p(u_{i_1}, \ldots, u_{i_h}) l, r \quad 1 \neq r. \]

If \( s(k+2) \) is the statement labelled \( 1 \),

then \( \text{rec}(s,k+1) = \neg p(\mathcal{T}_{i_1}, \ldots, \mathcal{T}_{i_h}) \)

and if \( s(k+2) \) is the statement labelled \( r \),

then \( \text{rec}(s,k+1) = p(\mathcal{T}_{i_1}, \ldots, \mathcal{T}_{i_h}) \).

(iii) If \( s(k+1) \) is the test statement

\[ p(u_{i_1}, \ldots, u_{i_h}) l, r \quad l = r \]

then \( \text{rec}(s,k+1) = \text{base}(s,k) \).

(iv) If \( s(k+1) \) is the input or output statement

\[ u_{i_1} \leftarrow u_i \]

then \( \text{rec}(s,k+1) = \langle \mathcal{T}_i, \ldots, \mathcal{T}_{i_1}, \ldots, \mathcal{T}_b \rangle \).

(v) If \( s(k+1) \) is a halt statement,

then \( \text{rec}(s,k+1) = \text{base}(s,k) \).

A path \( s \) is consistent if and only if there is no atomic formula \( A \) such that both \( A \) and \( \neg A \) are components of \( \text{rec}(s) \).
**Lemma A** (Paterson) Let $P$ be a schema, $I$ a pointwise interpretation, and $\sigma(P,I)$ the execution sequence of $P$ under $I$. Then there is an Herbrand interpretation $H$ such that $\sigma(P,I) = \sigma(P,H)$ and $\text{val}(P,I) = I(\text{val}(P,H))$. □

**Corollary B** (Paterson) A path in $P$ from the start statement is consistent if and only if it is an execution sequence. □

Schemas $P$ and $Q$ are **strongly equivalent**, denoted $P \equiv Q$, if and only if for every pointwise interpretation $I$, $\text{val}(P,I) = \text{val}(Q,I)$ whenever either value is defined. This definition requires that the two schemas produce the same output under every interpretation and is thus sometimes known as output equivalence. Although the definition does not apparently place any restrictions on the manner in which the output is produced, i.e. the actual computation method, in fact the free interpretation lemma tells us that $P$ and $Q$ must compute their output variables in essentially the same manner. In spite of this observation, there are even stronger notions of "sameness" which are sometimes of interest. We introduce next one such highly restrictive characterization of sameness. We begin by computing a sequence which contains a record of the functions and tests which have been applied to elements of the domain of an interpretation and which reflects the order in
which these operations have been applied.

If P is a program schema and I is a free interpretation, then we define the (possibly infinite) sequence seq(P, I), whose i-th element is denoted seq(P, I, (i)) as follows:

(i) If \( \sigma(P, I, (i)) \) is a start statement, then seq(P, I, (i)) = START.

(ii) If \( \sigma(P, I, (i)) \) is the assignment statement

\[
 k. y \leftarrow f(y_1, \ldots, y_n)
\]

then seq(P, I, (i)) = f(val(y_1, i-1), \ldots, val(y_n, i-1)).

(iii) If \( \sigma(P, I, (i)) \) is the test statement

\[
 k. p(y_1, \ldots, y_n) \text{ l, r}
\]

then seq(P, I, (i)) = p(val(y_1, i-1), \ldots, val(y_n, i-1)).

(iv) If \( \sigma(P, I, (i)) \) is the input statement

\[
 k. y \leftarrow x
\]

then seq(P, I, (i)) = val(x, i-1)

(v) If \( \sigma(P, I, (i)) \) is the output statement

\[
 k. z \leftarrow y
\]

then seq(P, I, (i)) = val(y, i-1)

(vi) If \( \sigma(P, I, (i)) \) is a halt statement then seq(P, I, (i)) = HALT

We next define isomorphism of two schemas, P and Q. We see that isomorphism requires a stronger degree of "sameness" than strong equivalence, but considerably less than a requirement of structural identity.
Two schemas, $P$ and $Q$, are isomorphic, denoted $P \cong Q$, if for every pointwise interpretation $I$,

$$\text{seq}(P, I) = \text{seq}(Q, I)$$

and

$$\text{val}(P, I) = \text{val}(Q, I)$$

Thus it follows immediately from the above definition that $P \cong Q$ implies that $P \equiv Q$. The converse, however is not true as the following example shows:

$$
\begin{array}{ll}
\text{P} & \text{Q} \\
0. \text{START} & 0. \text{START} \\
1. y_1 \leftarrow fx & 1. y_2 \leftarrow gx \\
2. y_2 \leftarrow gx & 2. y_1 \leftarrow fx \\
3. z \leftarrow y_1 & 3. z \leftarrow y_1 \\
4. \text{HALT} & 4. \text{HALT} \\
\end{array}
$$

By the free interpretation lemma, we need only consider free interpretations, and since neither $P$ nor $Q$ contain any test statements, there is precisely one free interpretation for $P$ and $Q$. Call that interpretation $H$.

$$\text{val}(P, H) = \text{val}(Q, H) = fx$$
and hence $P \equiv Q$. However,

$$\text{seq}(P, H) = (\text{START}, fx, gx, fx, \text{HALT})$$
and

$$\text{seq}(Q, H) = (\text{START}, gx, fx, fx, \text{HALT})$$
and hence $P$ is not isomorphic to $Q$.

It is perhaps instructive to indicate why we must require that $\text{val}(P, I) = \text{val}(Q, I)$. It is necessary to insure that isomorphism is a stronger property than strong equivalence. Consider the two schemas $P$ and $Q$: 

...
Again, by the free interpretation lemma, we only have to consider free interpretations. Furthermore, since neither P nor Q contains a test statement, there is precisely one free interpretation, call it H, for P and Q. Then \( \text{val}(P,H) = (gx,f^2x) \) and \( \text{val}(Q,H) = (f^2x,gx) \) and hence P is not strongly equivalent to Q.

\[
\text{seq}(P,H) = (\text{START},fx,gx,f^2x,\text{HALT}) \text{ and }
\text{seq}(Q,H) = (\text{START},fx,gx,f^2x,\text{HALT})
\]

and hence if we did not require that \( \text{val}(P,I) = \text{val}(Q,I) \) for every pointwise interpretation I, we would have P isomorphic to Q, but not strongly equivalent to Q. We mention that Chandra [2] uses a similar definition of isomorphism, and indeed our definition is based on his. The fact that his model associates output statements with halt statements enables him to omit the explicit requirement that

\[
\text{val}(P,I) = \text{val}(Q,I)
\]

in his definition of isomorphism. In his formulation,

\[
\text{seq}(P,I) = \text{seq}(Q,I)
\]

implies \( \text{val}(P,I) = \text{val}(Q,I) \)

We introduce one other notion of structural sameness at this time. We define \( \text{funcseq}(P,I) \) to be the sequence of non-test statements of P which are executed under free
interpretation $I$. Thus $\text{funcseq}(P,I)$ is formed exactly as $\text{seq}(P,I)$ without including the test statements encountered under $I$.

We say that two schemas $P$ and $Q$ are functionally similar, denoted $P \sim Q$, if for every pointwise interpretation $I$,

$$\text{funcseq}(P,I) = \text{funcseq}(Q,I)$$

and

$$\text{val}(P,I) = \text{val}(Q,I)$$

Thus we have that $P \equiv Q$ implies $P \sim Q$ implies $P \equiv Q$. None of the converses hold. We demonstrated above that $P \equiv Q$ does not imply that $P \sim Q$. The same example verifies that there are schemas $P$ and $Q$ such that $P \equiv Q$ but $P \not\sim Q$. The following pair of schemas are functionally similar, but not isomorphic.

$$\begin{array}{l}
P \\
0. \text{START} \\
1. p(x)2,2 \\
2. z \leftarrow f(x) \\
3. \text{HALT}
\end{array} \quad \begin{array}{l}
Q \\
0. \text{START} \\
1. z \leftarrow f(x) \\
2. \text{HALT}
\end{array}$$

Garland and Luckham [4], introduced the notion of translatability for classes of schemas. We say a class of schemas $A_1$ is translatable into a class of schemas $A_2$, denoted $A_1 \rightarrow A_2$, if for every $P_1 \in A_1$, there is a strongly equivalent $P_2 \in A_2$. 
In the chapters which follow, we investigate decision problems and translatability questions for two classes of schemas—the reachable and the semifree schemas. We then consider questions about the relationship between these two classes of schemas and several other classes. In particular we look for interesting subclasses for which reachability and semifreedom are decidable properties and investigate equivalence problems in the case that the two schemas being considered are known to be in different classes.

We study the preservation of semantic properties by various sameness relations. If two schemas are similar in one of several ways, and one of the schemas has some property, we investigate under what circumstances the other schema must also have that property.

In Chapter 6 we show a fundamental relationship between function and pointwise interpretations. We then use that relationship as a basis for introducing three additional classes of schemas and investigating the relationship between these classes and the reachable, semifree, and free schemas.

Finally, in Chapter 7, we define classes of programs which are the analogues of the classes of schemas which we have studied. We investigate decision problems and translatability questions for these classes. We also study the inheritance of schema properties by programs and the inheritance of program properties by schemas.
CHAPTER 2

REACHABLE SCHEMAS

This thesis examines classes of schemas which have been defined by requiring that the members of a class fulfill certain semantic requirements. We shall call such a class a "semantically restricted class of schemas". In this chapter, we introduce and study in detail one such class, the reachable schemas. In subsequent chapters we introduce other semantically restricted classes and consider the relationships between such classes.

Paterson [9] introduced the concept of a free schema; it is a property of schemas which is interesting for several reasons. Paterson felt that a common characteristic of schemas with unsolvable decision problems was an inherent inefficiency characterized by repeated calculations. He hypothesized that if one added restrictions to the general model which eliminated these repetitions, the more limited class might have solvable decision problems. Furthermore the restrictions would eliminate from consideration program schemas which had undesirable features and hence were not models of what we would generally consider a good program. An additional characteristic of freedom is that although it is a semantic property of a schema, it is easy to show that it
has an equivalent structural characterization. With this motivation we define free schemas, and introduce the notion of reachability for schemas. In Chapter 7, we shall introduce similar properties for programs. We feel that the notion of reachability is almost a necessary, although certainly not sufficient, condition for a program to be considered good or even acceptable. We had hoped, also, much as Paterson had, that the restricted class of schemas would have solvable decision problems. This is not in general the case.

A schema \( P \) is free if every finite path through its flow diagram from the start statement is an initial segment of some execution sequence \( \sigma(P,I) \).

A schema \( P \) is uniformly free if there is a function interpretation \( \mathcal{I} \) with domain \( D \) such that for every finite path \( s \) through its flow diagram from START, there is an input \( d \in D^s \) such that \( s \) is an initial segment of \( \sigma(P,\mathcal{I},d) \).

A statement labelled \( s_k \) in a schema \( P \) is reachable if there is a pointwise interpretation \( I \) such that \( s_k \) is a statement in \( \sigma(P,I) \).

A schema \( P \) is reachable if every statement in \( P \) is reachable. Let \( \mathcal{P} \) denote the class of reachable schemas.
A schema $P$ is uniformly reachable if there is a function interpretation $\ll$ such that for every statement $s_K$ there is an input $d$ such that $s_K$ is a statement in $\sigma(P,\ll,d)$.

We shall compare the relative power of reachability and uniform reachability in Chapter 6.

Since we consider reachability a necessary property of any program which we would wish to consider good, the following proposition tells us that at least these good versions of a schema always exist; we need simply to remove the inaccessible code and connect any "loose ends" (i.e. edges leaving tests which now have no successor statements) to an arbitrary statement such as the successor of the start statement.

**PROPOSITION 2.1** Every schema is isomorphic to some reachable schema.

We shall see shortly, however, that although these improved versions of nonreachable schemas exist, we cannot in general hope to obtain them effectively.

**LEMMA 2.2** It is decidable whether a reachable schema $P$ halts under some pointwise interpretation.

**Proof:** A reachable schema halts under some pointwise
interpretation if and only if it contains a halt statement.

**Theorem 2.3** There is no algorithm which given an arbitrary program schema P, constructs a reachable schema Q such that $P = Q$.

**Proof:** Assume there was such an algorithm. Then by lemma 2.2 we could decide whether P halts under some interpretation, thus contradicting theorem 4.1 of Luckham, Park, and Paterson [6].

**Lemma 2.4** It is decidable of a finite path through a schema P from the start statement, whether it is consistent.

Thus we have that if $\mathcal{C}_P$ is the set of finite consistent paths through the schema P, then $\mathcal{C}_P$ is recursive, and hence recursively enumerable.

**Lemma 2.5** Let P be a program schema and $s_K$ a statement in P. Then $s_K$ is an element of some consistent path if and only if $s_K$ is an element of a finite consistent path.

**Lemma 2.6** It is partially decidable of a program schema P and statement $s_K$, whether:
a) \( s_K \) is a reachable statement in \( P \)
b) \( P \) is a reachable schema

**Proof:**  
a) By lemma 2.4 we can enumerate finite consistent paths through \( P \). By lemma 2.5 and corollary B, we know that if \( s_K \) is reachable, it is an element of some finite consistent path through \( P \).

b) By definition, \( P \) is reachable if and only if every statement \( s_K \) in \( P \) is reachable. Thus we need only apply the partial decision procedure of part a to each of the finitely-many statements of \( P \). \( \square \)

**COROLLARY 2.7**  
a) It is not partially decidable whether an arbitrary statement of a schema is unreachable.
b) It is not partially decidable whether an arbitrary assignment statement of a schema is unreachable.

**Proof:**  
a) Let \( P \) be an arbitrary schema. Let \( s_K \) be an arbitrary statement of \( P \). Assume without loss of generality that \( P \) contains a single halt statement. If it were partially decidable whether \( s_K \) were unreachable, then by lemma 2.6 we could decide whether an arbitrary statement in a schema was reachable. In particular we could decide whether \( P \) halts under some pointwise interpretation. But this has been shown by Paterson [9] to be undecidable.

b) Let \( P \) be an arbitrary schema. We assume without loss
of generality that $P$ contains a single halt statement. We construct schema $Q$ from $P$ by replacing the halt statement of $P$ by an assignment statement $s$ followed by a halt statement. Clearly $P$ halts under some interpretation if and only if $s$ is reachable in $Q$. Thus if we could decide whether $s$ were reachable in $Q$, we could decide whether $P$ halts under some interpretation. \hfill $\Box$

**Theorem 2.8** It is not decidable whether a program schema $P$ is reachable.

*Proof:* We will reduce assignment statement reachability to schema reachability. Let $P$ be a schema containing $m$ statements. Let $s_K$ be an assignment statement of $P$. Let $p$ be an $n$-exit predicate (or alternately a series of $n-1$ two-exit predicates) which does not appear in $P$. We construct the schema $Q$ from $P$ by inserting an initializing assignment statement $\vec{y} \leftarrow x$ immediately after START. $\vec{y}$ denotes the vector of program variables and $x$ is some input variable. This notation indicates that every program variable is initially assigned the value of $x$. The purpose of the initializing assignment statement is simply to guarantee that every program variable has been assigned a value before it is retrieved from or tested. We also insert a copy of the test $p(x)$ after statement $s_K$. The branch from the $m$-exit of $p(x)$ enters statement $s_m$. Figures 1 and 2 contain the
**SCHEMA P**

**FIGURE 1**

**SCHEMA Q**

**FIGURE 2**
outlines of schemas P and Q.
We see that statement $s_K$ is reachable in P if and only if Q is a reachable schema. If $s_K$ is reachable in P, then clearly $s_K$ is reachable in Q and hence the test p is reachable in Q. Then every statement of Q is reachable and hence Q is a reachable schema. If $s_K$ is unreachable in P, then $s_K$ is unreachable in Q and so Q is not a reachable schema. Thus if we could decide reachability for Q, we could decide assignment statement reachability for $s_K$.□

The next theorem will demonstrate that if we are given an arbitrary schema and reachable one, we cannot even decide whether they are strongly equivalent.

**Theorem 2.9** There is no algorithm to decide of an arbitrary schema P, and a reachable schema whether they are strongly equivalent.

**Proof:** Assume there is such a decision procedure. Let P be an arbitrary schema. By lemma 2.6 we can enumerate the reachable schemas. Using the hypothesized decision procedure, we could then check each reachable schema $P_K$ to see if $P \equiv P_K$. Since every schema is strongly equivalent to some reachable schema, we would have an algorithm which for any program schema produces a strongly equivalent reachable one. This contradicts
Occasionally in this work, we include proofs of theorems even though the result is an immediate consequence of a result which appears later in this work. This is done in those cases for which we felt that there is a natural progression of questions to consider and that the rearrangement of this order would detract from the coherence of the work. The preceding result is an example of such a theorem—we shall see that it follows immediately from corollary 2.11.

Our next theorem is both interesting in its own right, and an important tool for our consideration of the equivalence problem for reachable schemas.

**LEMMA ** It is not decidable whether an arbitrary schema \( P \) halts under every pointwise interpretation.

We shall use this lemma to show that the same property is undecidable for reachable schemas. We will do this by constructing an algorithm which when applied to an arbitrary schema \( P \), produces a reachable schema \( Q \), such that \( Q \) halts under every pointwise interpretation if and only if \( P \) does. Then, if we could decide whether a reachable schema halts under every pointwise
interpretation, we could decide this for arbitrary schemas, thus contradicting lemma C.

**Theorem 2.10** It is not decidable whether a reachable schema halts under every pointwise interpretation.

*Proof:* We present an algorithm which for a given schema P, produces a reachable schema Q which halts under every pointwise interpretation if and only if P does.

Let P be an arbitrary schema. Assume P contains n+1 instructions labelled 0, 1, ..., n with 0 the start statement and 1 the unique successor of the start statement.

Let p_1, q_1, ..., q_n, r_1, ..., r_{n-1} be predicate symbols which do not appear in P.

We construct Q as follows.

We call the above subschema the initial subschema. \( \bar{y} \leftarrow x \) is an initializing assignment statement, as described in theorem 2.8.
We let $s(i)$ denote the instruction in $P$ labelled $i$.

**Case 1** If $s(i)$ is an assignment, input, or output statement in $P$ such that the successor of $s(i)$ is $s(j)$, then in $Q$ we have the subschema:

![Diagram]

for $i = 1, \ldots, n-1$.

If the statement labelled $n$ is one of the three types of statements considered in case 1, then in $Q$ we have the subschema:

![Diagram]
case 2 If $s(i)$ is a test statement in $P$ with $0$-successor $s(k)$, and $1$-successor $s(j)$, then in $Q$ we have the subschema:

![Diagram](image)

for $i = 1, \ldots, n-1$.

If the statement labelled $n$ in $P$ is a test statement, in $Q$ we will have a subschema labelled $n$ which is exactly like the above subschema except that the branches to subschema $i+1$ are replaced by halt statements.
case 3 If $s(i)$ is a halt statement in $P$, then in $Q$ we have the subschema:

![Diagram]

for $i = 1, \ldots, n-1$.

If the statement labelled $n$ in $P$ is a halt statement, then in $Q$ we have the subschema:

![Diagram]

We point out here that a schema $Q$ will never contain all of the predicate symbols $q_1, \ldots, q_n, r_1, \ldots, r_{n-1}$ but will contain one $q_i$ for each test statement in $P$ and one $r_i$ for each halt statement in $P$ which is not labelled $n$. We have chosen to include the extra predicate symbols simply to facilitate the notation.

It remains to show that the schema $Q$, so
constructed, has the required properties. Before doing so, however, we shall describe informally how the schema Q functions.

Intuitively, Q can be in one of two distinct modes, depending on the interpretation of the test p(x). Furthermore, since x is an input variable, and thus can never be assigned a new value, once a mode is determined, it cannot be changed. If (Ip)(x) = 1, then the schema is in what we shall call "reachability mode". That is, we are simply guaranteeing in this case that every instruction of Q can be reached.

If (Ip)(x) = 0, then the schema is in what we shall call "simulation mode", and is effectively simulating the computation of P under interpretation I. In either case, we are not really interested in what value is calculated by Q. If (Ip)(x) = 0, then P halts under interpretation I if and only if Q halts under interpretation I. If (Ip)(x) = 1 then Q halts regardless of P's behavior under interpretation I.

We also point out here the reason for the test statements q_x and r_x. They are only encountered under interpretations which put the schema into reachability mode and are in fact used to guarantee that every statement is reachable. In the case that s(i) is a test statement, we cannot guarantee that there is some interpretation which makes the value of s(i) under that
interpretation 1 and some other interpretation such that
the value of \( s(i) \) under this latter interpretation is 0.
Hence it is possible that one of the two test statements
\( p(x) \) might not be reachable. By adding the tests \( q_i(x) \),
where \( q_i \) does not appear anywhere else in the schema \( Q \),
we can guarantee that both \( p(x) \) tests are reachable.

If \( s(i) \) is a halt statement, and \( i \) is not \( n \), we add
the test \( r_{\xi}(x) \) to guarantee that the halt statement can
be reached. Since the predicate symbol \( r_{\xi} \) does not
appear anywhere else in the schema \( Q \), for those
interpretations for which \( (Ir_{\xi})(x) = 1 \), the halt
statement is reachable.

Formally, we shall first show that \( Q \) is a reachable
schema. To do this we must first show that every
subschema of \( Q \) is reachable, and that every statement
within a subschema is reachable. To do this, we need
only consider those interpretations for which
\( (Ip)(x) = 1 \).

(i) Subschema 1 is reachable. We begin at the start
statement, test \( p(x) \) and then enter subschema 1.

(ii) Assume subschema \( i, 1 \leq i \leq n-1 \), is reachable.

**case 1** Statement \( i \) of schema \( P \) is an assignment,
input, or output statement. Then by our construction, we
execute \( s(i) \), test \( p(x) \), and enter subschema \( i+1 \).

**case 2** Statement \( i \) of schema \( P \) is a test
statement.
If \( I(s(i)) = 1 \) and \((Iq_\downarrow)(x) = 1\),
or if \( I(s(i)) = 0 \) and \((Iq_\downarrow)(x) = 0\),
then the test statements \( s(i), p(x), \) and \( q_\downarrow(x) \) are
executed in that order, and then control goes to
subschema \( i+1 \).
If \( I(s(i)) = 1 \) and \((Iq_\downarrow)(x) = 0\),
or if \( I(s(i)) = 0 \) and \((Iq_\downarrow)(x) = 1\),
then the test statements \( s(i), p(x), q_\downarrow(x), p(x), \) and
\( q_\downarrow(x) \) are executed in that order, and then control goes
to subschema \( i+1 \).

case 3 Statement \( i \) of schema \( P \) is a halt
statement. For every interpretation in which
\((Ir_\downarrow)(x) = 0\), we execute \( p(x) \) followed by \( r_\downarrow(x) \) and then
enter subschema \( i+1 \).

Clearly every statement of the initial subschema is
reachable. We must now show that every instruction
within a subschema \( i \) is reachable.

case 1 If statement \( i \) of schema \( P \) is an
assignment, input, or output statement, this is
immediate.

case 2 If statement \( i \) of schema \( P \) is a test
statement, we can do an exhaustive case study to verify
that both copies of the tests \( p(x) \) and \( q_\downarrow(x) \) are
reachable under some interpretation. This has been
discussed informally earlier.
case 3 If statement \( i \) of schema \( P \) is a halt statement, we see that for any interpretation \( I \) for which \((\text{Ir}_I)(x) = 1\), the halt statement is reachable. Clearly the other two statements in the subschema are reachable.

Our final task is to show that \( Q \) halts under every pointwise interpretation if and only if \( P \) does. Assume \( Q \) halts under every pointwise interpretation. Then in particular, it halts for those interpretations \( I \) for which \((\text{Ip})(x) = 0\). But we have already seen in our informal discussion, that for those interpretations of \( Q \) for which \((\text{Ip})(x) = 0\), \( Q \) simply simulates \( P \)'s actions. Thus \( P \) must also halt for these interpretations.

Furthermore, \( p(x) \) is not a predicate symbol in \( P \)'s language, and hence \( P \)'s behavior under an interpretation must be independent of the interpreted value of \( p(x) \) and hence \( P \) must halt under every pointwise interpretation since it halts under every pointwise interpretation for which \((\text{Ip})(x) = 0\).

Assume \( P \) halts under every pointwise interpretation. Then by the above discussion, \( Q \) halts under every pointwise interpretation \( I \) such that \((\text{Ip})(x) = 0\). If \((\text{Ip})(x) = 1\), we have shown that after executing the initial subschema of \( Q \), we execute subschemas \( 1, 2, \ldots, m \). If \((\text{Ir}_I)(x) = 0\) for every \( 1 \leq i \leq n-1 \) for which statement \( s(i) \) of \( P \) is a halt statement, then \( m = n \). By our construction, regardless of what type of statement \( s(n) \)
of \( P \) is, in \( Q \) we halt after executing this subschema. If there is an \( 1 \leq i \leq n-1 \) such that \( s(i) \) is a halt statement in \( P \) and \( (\text{Ir}_\xi)(x) = 1 \), then let \( k \) be the smallest such \( i \). Then \( m = k \) and by our construction \( Q \) halts under such an interpretation after performing subschema \( k \). Thus \( Q \) halts under every pointwise interpretation.

Thus we have shown that \( Q \) is a reachable schema which is constructed effectively from an arbitrary schema \( P \) and such that \( Q \) halts under every pointwise interpretation if and only if \( P \) halts under every pointwise interpretation. Therefore, if we could decide whether a reachable schema halts under every pointwise interpretation, we could decide whether an arbitrary schema halts under every pointwise interpretation, contradicting lemma C. \( \Box \)

**Corollary 2.11** It is not decidable whether two reachable schemas are strongly equivalent.

**Proof:** Let \( P \) be a reachable schema with output variables \( z_1, \ldots, z_n \) and input variable \( x \).

Let \( P' \) be the reachable schema of figure 3. Clearly \( P' \) halts under every pointwise interpretation.

We construct the schema \( P'' \) by replacing each halt statement of \( P \) by the sequence of instructions shown in figure 4. Clearly \( P'' \) is reachable since \( P \) is reachable. Furthermore
FIGURE 3

START

\( Z_1 \leftarrow x \)

\vdots

\( Z_n \leftarrow x \)

HALT

FIGURE 4

\( Z_1 \leftarrow x \)

\vdots

\( Z_n \leftarrow x \)

HALT
\[ P' \equiv P'' \text{ iff } P'' \text{ halts under every pointwise interpretation} \]
\[ \text{iff } P \text{ halts under every pointwise interpretation.} \]

Thus, if strong equivalence was decidable for reachable schemas, we could decide whether a reachable schema \( P \) halts under every pointwise interpretation. \( \square \)

Paterson showed that it is partially decidable whether an arbitrary schema halts under every pointwise interpretation. Thus although we have just shown that it is not decidable whether a reachable schema halts under every pointwise interpretation, it is at least partially decidable.

We shall see in Chapter 7, that the analogous problem for reachable programs is not even partially decidable.
CHAPTER 3

SEMIFREE SCHEMAS

In this chapter we introduce another class of schemas, called the semifree schemas, which we denote by \( \mathcal{S} \). As in our definition of reachability, we are motivated by a desire to model a property of schemas and programs which we feel characterizes a certain type of optimization of code. Once again it is a semantic property which is of interest, although it has a rather structural flavor.

We call an edge connecting two nodes in \( P \), a path segment of length one.

A schema \( P \) is semifree if for each path segment \( s \) of length one in \( P \), there is a pointwise interpretation \( I \) such that \( s \) is traversed under \( I \).

In the next chapter we shall see that \( \mathcal{I} \subseteq \mathcal{S} \subseteq \mathcal{R} \subseteq \mathcal{P} \), where \( \mathcal{P} \) denotes the class of free schemas. Paterson showed that a schema is free if and only if it does not contain any repeated tests, in the sense that no predicate symbol is ever applied to the same n-tuple of elements of the Herbrand universe more than once within an execution sequence. Furthermore, there are schemas which are inherently nonfree in the
sense that they are not strongly equivalent to any free schema, and thus must repeat some tests in order to do the desired computation. Our motivation for defining semifreedom is to somehow distinguish between schemas which have redundant (i.e. unnecessarily repeated) tests, and those which contain a repetition of a test only when it is actually required for the calculation. Of course it is difficult to say exactly what it means for a test to be required.

We shall say a test statement $t$ of a schema is necessary if there are pointwise interpretations $I_1$ and $I_2$ such that the true exit of $t$ is taken under $I_1$ and the false exit is taken under $I_2$.

Intuitively, this says that for some interpretations one exit of the test is taken, and for others, the other exit is taken. Thus, one feels that the test is really testing something, and in that sense is necessary for the computation. The simple example of figure 5 illustrates a schema $\mathcal{P}$ which contains a test which by this definition is unnecessary.
Although this schema is reachable, the test designated \( \mathbf{b} \) is unnecessary since under no interpretation is the 0-exit ever taken. Thus it can be removed and we would then have the schema \( Q \) shown in figure 6, which is strongly equivalent to \( P \).
FIGURE 6--SCHEMA Q

We note that this definition does not exclude all tests which could be removed from a schema while preserving strong equivalence, but it does encompass many such instances.

The schema R of figure 7 illustrates the above point. Although it has no tests which are unnecessary under the above definition, there is a test which really serves no purpose.
Having described the limitations of this definition, we next relate the notions of semifreedom and necessary tests. The following proposition follows directly from the definitions.

**Proposition 3.1** A schema $P$ is semifree if and only if every test in $P$ is necessary. \(\square\)

As in the case of reachable schemas, we shall demonstrate that $C$ is translatable into $A$. Once again there is no effective procedure to do the translation. The result in this case, however, is not quite as strong as the analogous one for reachable schemas. We shall show that every schema is strongly equivalent to some
semifree schema, but there are schemas which are not isomorphic to any semifree schema.

It is immediately clear from our definitions that every semifree schema is reachable, for certainly if every exit from a node is traversed under some interpretation, the node itself must be reachable.

**PROPOSITION 3.2** If $P \in \mathcal{J}$ then $P \in \mathcal{R}$. □

**LEMMA 3.3** It is decidable whether a semifree schema halts under some pointwise interpretation.

**Proof**: This follows directly from proposition 3.2 and lemma 2.2. □

**THEOREM 3.4** $\mathcal{R}$ is translatable into $\mathcal{J}$.

**Proof**: We shall prove this result by describing a procedure to convert any schema into a strongly equivalent semifree schema. We point out here that this procedure is not effective as it requires knowledge of whether or not a given path segment of length one is ever traversed under some pointwise interpretation. We shall demonstrate in theorem 3.8 that this is not in general decidable.
PROCEDURE

1) Delete any edges which cannot be traversed under any pointwise interpretation. Note that the resulting graph may not be a program schema, and may in fact be a disconnected graph.

2) In the resulting graph, delete any node, other than the start statement, which has no incoming edges. It will of necessity also have no outgoing edges as they will have been removed in step 1. We now have a single entry connected graph.

3) If n is a node labelled with a test statement, which as a result of the above procedure has only one exiting edge, delete n and connect all edges which enter n to n's unique successor node.

The above procedure clearly produces a schema which is strongly equivalent to the original schema as it deletes only inaccessible path segments and nodes, and unnecessary tests. Furthermore, since it contains only traversable path segments, it is semifree. □

The following example may serve to illustrate the above procedure. We begin with schema P of figure 8 which is not semifree.
FIGURE 8--SCHEMA P
Applying step 1 we get the flow diagram shown in figure 9.

![Flow Diagram](image-url)
We next apply step 2, and delete the disconnected node designated (a).

Finally we apply step 3 and get rid of the unnecessary test designated (b), obtaining the semifree schema Q of figure 10 which is strongly equivalent to P.

FIGURE 10—SCHEMA Q
THEOREM 3.5  There is no effective procedure which given an arbitrary schema P, constructs a strongly equivalent semifree schema Q.

Proof: Assume there was such an effective procedure. Then by lemma 3.3, we would have an algorithm for deciding whether an arbitrary schema halts under some pointwise interpretation.□

THEOREM 3.6  There is a schema which is not isomorphic to any semifree schema.

Proof: Consider schema P of figure 11. Clearly P is not semifree as the branch designated cannot be traversed under any pointwise interpretation.

We shall demonstrate that if Q is a schema such that \( P \uplus Q \), then Q is not semifree.

Let \( I_1 \) and \( I_2 \) be pointwise free interpretations such that

\[
I_1(p(fx)) = 0 \\
I_2(p(fx)) = 1
\]

Then

\[
\text{seq}(P, I_1) = \langle \text{START}, fx, p(fx), p(fx), \text{HALT} \rangle = \text{seq}(Q, I_1)
\]

\[
\text{seq}(P, I_2) = \langle \text{START}, fx, p(fx), \text{HALT} \rangle = \text{seq}(Q, I_2)
\]

So Q must include the schema segment shown in figure 12, where \( y' \) denotes some program variable. If Q is to be semifree, then there must be an interpretation I such that \( \sigma(Q, I) \) includes the edge designated. But if
FIGURE 11 -- SCHEMA P

FIGURE 12
1(p(fx)) = 0, then after testing p(fx) twice, the schema halts at \(d\), and if \(I(p(fx)) = 1\), then after testing p(fx) once, the schema halts at \(c\). Thus in no case can branch \(b\) be traversed. Thus \(Q\) cannot be semifree. □

We mention here that there is a reachable schema \(R\) such that \(P \not\subseteq R\) as guaranteed by proposition 2.1. It is shown in figure 13. Although branch \(c\) of schema \(R\) can never be traversed, the schema is nonetheless reachable.

We shall next consider membership in \(A\). We shall demonstrate that it is partially decidable, but not decidable whether a schema is semifree.

**Theorem 3.7** It is partially decidable whether a schema is semifree.

**Proof:** Enumerate the finite consistent paths. By lemma 2.5, each path segment appears in some consistent path if and only if it appears in some finite consistent path. The procedure terminates if and when every edge is included in some finite consistent path. □

**Theorem 3.8** It is not decidable whether an arbitrary schema is semifree.

**Proof:** We present an algorithm which given an arbitrary
schema $P$ and assignment statement $s_K$ of $P$, constructs a
schema $Q$ such that $Q$ is semifree if and only if
assignment statement $s_K$ of $P$ is reachable. By
corollary 2.7b, it is not decidable whether an assignment
statement in an arbitrary schema is reachable. We shall
demonstrate that $Q$ is semifree by verifying that each of
its tests is necessary.

Assume $P$ contains $n$ test statements:

\[ P_i(y_{i1}, \ldots, y_{i\alpha_i}) \quad i = 1, \ldots, n \]

We shall designate these tests $t_1, \ldots, t_n$.

Let $s_K$ denote an arbitrary assignment statement in $P$.
Let $f_i \ i = 1, \ldots, n$ be $n$ distinct function symbols not
appearing in $P$.

Let $q$ be an $n+1$ exit predicate symbol not in $P$.

We construct $Q$ from $P$ by inserting the test statement $q$
after statement $s_K$. Each of the first $n$ exits from $q$
enters one of the $n$ test statements of $P$ after first
altering the value of one of the arguments of the test.

This is done to guarantee that for each exit from a test $t_i$, there is some interpretation which selects that exit.
The $(n+1)$-st exit from $q$ enters the statement which is
the unique successor of $s_K$ in $P$, denoted $s_{K-1}$. We also
insert an initializing assignment statement immediately
following the start statement.

An outline of schema $Q$ is shown in figure 14.

It remains to demonstrate that $Q$ is semifree if and only
START

\[ \overline{y} \leftarrow x \]

[Diagram]

\[ y_{11} \leftarrow f_1(y_{11}) \]

[Diagram]

\[ y_{m1} \leftarrow f_m(y_{m1}) \]

[Diagram]

\[ y_{n1} \leftarrow f_n(y_{n1}) \]

[Diagram]

FIGURE 14--SCHEMA O
if statement $s_k$ of $P$ is reachable. $s_k$ is reachable in $Q$ if and only if $s_k$ is reachable in $P$, since the code of schemas $P$ and $Q$ is identical up to statement $s_k$. It is only after $s_k$ has been executed in $Q$ that the execution sequence through $Q$ is in any way different from the corresponding execution sequence through $P$. If $Q$ is semifree, then $Q$ is reachable. Thus $s_k$ is reachable in $Q$ and hence in $P$.

If $s_k$ is reachable in $P$ then clearly $s_k$ is reachable in $Q$. Since $q$ is a new predicate symbol, each of the $n+1$ exits from $q$ may be taken. Furthermore, since the function symbols $f_k$ appear nowhere else in the schema $Q$, we are guaranteed that the value being tested by predicate symbol $p_k$ has never been tested previously, and hence either exit may be taken from each test statement $t_k$. Thus $Q$ is semifree.

Therefore, $Q$ is semifree if and only if $s_k$ is reachable in $P$ and thus it is undecidable whether an arbitrary schema is semifree.

We have seen so far that although every schema is strongly equivalent to some semifree schema, there is no algorithm to produce the more desirable form of the schema. Furthermore, we have demonstrated that we cannot decide whether a schema is semifree, and hence already possesses the desired properties. We next consider whether presented with an arbitrary schema and a semifree
schema, we can decide whether they are equivalent. We shall see that this too is not a solvable problem.

**THEOREM 3.9** There is no algorithm to decide of an arbitrary schema and a semifree schema whether they are strongly equivalent.

**Proof:** Assume there is such a decision procedure. Let $P$ be an arbitrary schema. By theorem 3.7 we can enumerate the semifree schemas. Using the hypothesized decision procedure, we could then check each semifree schema $P_K$ to see if $P = P_K$. Since every schema is strongly equivalent to some semifree schema, we would have an algorithm which for any schema produces a strongly equivalent semifree schema. This contradicts theorem 3.5.

The last problems we shall consider in this chapter are the halting and equivalence problems for semifree schemas.

**THEOREM 3.10** It is not decidable whether a semifree schema halts under every pointwise interpretation.

**Proof:** We present an algorithm which for a given reachable schema $P$, constructs a semifree schema $Q$ which halts under every pointwise interpretation if and only if $P$ does.
Let $P$ be a reachable schema. Then it follows from theorem 2.10 that it is undecidable whether $Q$ halts under every pointwise interpretation.

Assume $P$ contains $n$ test statements denoted by $t_1, \ldots, t_n$, where $t_1$ is the first test statement encountered under every interpretation.

Let $k_i$ denote the statement which is the $i$-successor of test $t_k$, $k=1, \ldots, n; i=0, 1$.

Let $p$ be a predicate symbol not appearing in $P$.

Let $q$ be an $n$-exit predicate symbol not appearing in $P$.

We begin by constructing $P'$ from $P$. This is done by inserting a test statement $p_x$ as the 0-successor of each test $t_k$ and as the 1-successor of each test $t_k$ (thus $2n$ copies of the test $p_x$ are added to $P$). The 0-exit of the $p_x$ statement which is the $i$-successor of $t_k$ enters statement $k_i$. The 1-exit enters a halt statement. All edges which enter the node $k_i$ in $P$, enter the $p_x$ test preceding $k_i$ in $P'$. Thus, if in $P$ we have:

\[ \begin{array}{c}
K_0 \\
K_i
\end{array} \]

where $\rightarrow$ denotes the set of edges entering $k_i$ in $P$.

Then in $P'$ we have:
We next construct $Q$ from $P'$.

Let 1 designate the successor to the start statement in $P'$.

Immediately following the start statement, we insert an initializing statement $y \leftarrow x$. As in theorem 3.8, we are assigning an initial value to each program variable to assure that each non-input variable is assigned to before it is used. Following this initializing statement, we insert a copy of the test statement $px$. The 0-successor of this test will be statement 1 of $P'$. The 1-successor of this test is the n-exit test $q$. The exits of $q$ are labelled 1, ..., n for convenience of reference.

Exit k of test $q$ has a branch to statement $t_k$.

Schema $Q$, constructed as described above, is shown in figure 15. As in theorem 2.10, we can think of schema $Q$ as having two modes. If $(Ip)(x) = 0$ then $Q$ simulates the computation of $P$ with the addition of repeated testing of $px$. Since $x$ is an input variable, the value is never
FIGURE 15—SCHEMA Q
reassigned, and therefore the 0-exit will always be taken from the test px.
If \((Ip)(x) = 1\)
and \((Iq) = k\) \(k=1, \ldots, n\)
and \((It_K) = i\) \(i=0, 1\)
Then, after the initializing assignment, the computation will proceed as follows:
px is tested \(
\begin{align*}
1\text{-exit taken} \\
\end{align*}
\)
q is tested \(
\begin{align*}
k\text{-exit taken} \\
\end{align*}
\)
t_K is tested \(
\begin{align*}
i\text{-exit taken} \\
\end{align*}
\)
px is tested \(
\begin{align*}
1\text{-exit taken} \\
\end{align*}
\)
Q halts.

We must now demonstrate that Q is semifree. This is done by verifying that each test is necessary. This is immediate for the test q and the initial instance of the test px. Thus it remains to show that each \(t_K, k=1, \ldots, n\) is necessary and the 2n remaining copies of px are necessary.

1) Statements \(t_K\)
Let \(I_K^1\) be a pointwise free interpretation such that
\[(I_K^1p)(x) = 1\]
\[(I_K^1q) = k\]
\[(I_K^1t_K) = 0\]
This interpretation causes the 0-exit of test \(t_K\) to be selected.
Similarly for interpretation \(I_K^2\) such that
\((I_{k}^{2}p)(x) = 1\)
\((I_{k}^{2}q) = k\)
\((I_{k}^{2}t_{k}) = 1\)

the 1-exit of \(t_{k}\) is selected.

2) Statements \(p_{x}\)

We consider the instance of \(p_{x}\) preceding statement \(k_{i}\), \(k=1,...,n;i=0,1\). Since \(k_{i}\) is reachable in \(P\), this statement \(p_{x}\) is reachable in \(P'\) and hence in \(Q\), for some pointwise interpretation \(I\) with \((Ip)(x) = 0\). Thus the 0-exit from the test will be taken under such an interpretation. We verify that the 1-exit of this test may indeed be selected under some pointwise interpretation, by considering a free interpretation \(I_{k_{i}}\) such that:

\((I_{k_{i}}^{2}p)(x) = 1\)
\((I_{k_{i}}^{2}q) = k\)
\((I_{k_{i}}^{2}t_{k}) = i\)

It still remains to demonstrate that \(P\) halts under every pointwise interpretation if and only if \(Q\) halts under every pointwise interpretation.

Assume \(P\) halts under every pointwise interpretation.

Let \(I\) be a pointwise interpretation of \(Q\).

\textbf{case 1} \((Ip)(x) = 0\)

Then after the initializing assignment statement, and the initial testing of \(p_{x}\), \(Q\) performs exactly as \(P\) does under interpretation \(I\), except that \(p_{x}\) is retested immediately.
preceding each execution of a statement \( k \), \( k=1,\ldots,n; \ i=0,1 \).

Hence \( Q \) halts under \( I \) since \( P \) halts under \( I \).

**case 2** \((Ip)(x) = 1\)

Assume \((Iq) = k\) \( k \in \{1,\ldots,n\}\)

\((It_w) = i\) \( i \in \{0,1\}\)

Thus after initializing the program variables, testing \( px, q, t_k \) and \( px \), in that order, \( Q \) halts.

Thus if \( P \) halts under every pointwise interpretation, \( Q \) halts under every pointwise interpretation.

Assume \( Q \) halts under every pointwise interpretation.

Let \( I \) be an arbitrary pointwise interpretation of \( P \).

Let \( I' \) be an extension of \( I \) such that \( I' \) is an interpretation of \( Q \) and \((I'p)(x) = 0\).

Assume \( P \) diverges under interpretation \( I \). Then clearly \( P \) diverges under \( I' \). But for any interpretation of \( Q \) such that \((I'p)(x) = 0\), \( Q \) simply simulates the action of \( P \).

Thus if \( P \) diverged under \( I' \), \( Q \) would diverge under \( I' \), contradicting our assumption that \( Q \) halts under every interpretation.

We have thus demonstrated that for any reachable schema \( P \), we can construct a semifree schema \( Q \), such that \( P \) halts under every pointwise interpretation if and only if \( Q \) halts under every pointwise interpretation. Thus if we could decide whether a semifree schema halts under every
pointwise interpretation, we could decide this for reachable schemas. This would contradict theorem 2.10. □

**COROLLARY 3.11** It is not decidable whether two semifree schemas are strongly equivalent.

**Proof:** The proof of this corollary follows directly from theorem 3.11, the proof of corollary 2.11, and the observation that the schema P' of that proof is semifree. □
CHAPTER 4

RELATIONSHIPS BETWEEN SCHEMA CLASSES

In the previous two chapters we introduced the classes $\mathcal{R}$ and $\mathcal{S}$ which are respectively the reachable and the semifree schemas. We discussed decidability and translatability properties of these two classes in relation to $\mathcal{P}$, the class of all program schemas. Both $\mathcal{R}$ and $\mathcal{S}$ are subsets of $\mathcal{P}$ which are defined in terms of semantic restrictions on $\mathcal{P}$. That is, in order for a schema $P$ to be a member of $\mathcal{R}$ or $\mathcal{S}$, we require that certain semantic properties hold for $P$.

In this chapter we consider other classes of schemas with semantic restrictions, and the relationship between these classes and $\mathcal{R}$ and $\mathcal{S}$.

We have defined $\mathcal{F}$, the class of free schemas, in Chapter 2. In this chapter, we define $\mathcal{L}$, the class of liberal schemas. Both $\mathcal{F}$ and $\mathcal{L}$ were originally defined by Paterson in his thesis, and several other people have extended his results for these two classes. We shall also be interested in the class of schemas which halt under every pointwise interpretation, which we shall denote by $\mathcal{H}$.

We saw in the previous two chapters that most of the decidability questions for reachable and semifree schemas are unsolvable. We shall investigate in this chapter
whether the situation is considerably altered by placing additional restrictions on the types of schemas to be considered, and also under what circumstances knowing that a schema possesses certain desirable properties is sufficient to allow us to decide whether the schema has other desirable properties.

We first define the notion of a value language, which is a particularly valuable tool for proving results about free and liberal schemas.

We say a schema is monadic if each of its functions and predicates are monadic. If $P$ is a monadic program schema with a single output variable, we define its value language, $L(P)$, as follows:

$$L(P) = \{w | \exists \text{ free interpretation } H, \exists x \in X \ni \text{val}(P,H) = wx\}$$

Garland and Luckham [4] observed that if two program schemas are strongly equivalent, then they have the same value language. They also demonstrated that the converse of this statement is not true.

If $\mathcal{C}$ is a class of schemas, we denote the class of value languages of schemas in $\mathcal{C}$ by $L(\mathcal{C})$, defined as follows:

$$L(\mathcal{C}) = \{L(P) | P \in \mathcal{C}\}.$$ 

**Theorem D** (Chandra) Let $\mathcal{F}$ be the class of all monadic free schemas which have a single output variable and whose function letters are in a set $F$. Then $L(\mathcal{F})$ is the
class of all regular sublanguages of $F^*$.

The following proposition follows immediately from our definitions and is thus offered without proof.

**PROPOSITION 4.1** \( \mathcal{I} \subseteq \mathcal{A} \subseteq \mathcal{R} \subseteq \mathcal{P} \). \( \square \)

We shall now demonstrate that each of the above inclusions are strict inclusions.

**LEMMA 4.2**

a) There is a schema which is not reachable.
b) There is a reachable schema which is not semifree.
c) There is a semifree schema which is not free.

**Proof:** In each case we shall exhibit a schema having the required properties.

a)
The statement designated (a) is unreachable and hence the schema is unreachable.

b)

The schema is reachable, but the edge designated (b) is traversed under no pointwise interpretation, and hence the schema is not semifree.
This schema is semifree. Since the value language of this schema is \( \{g^n f^n | n \geq 1\} \), a non-regular language, it follows from theorem D that the schema is not free.

Before considering the relationships between several classes of schemas defined by placing semantic restrictions on \( P \), we first discuss some properties of \( \mathcal{I} \) and \( \mathcal{L} \). Both of these classes were introduced by Paterson [9], and additional results about \( \mathcal{I} \) may be found in Chandra's thesis [2]. We have proved some
additional results which are both useful in extending our insight into the classes $\mathcal{I}$ and $\mathcal{L}$, and are particularly important when studying the relationship between these classes and $\mathcal{R}$ and $\mathcal{S}$.

**Theorem E (Greibach)** The following statements regarding a program schema $P$ are equivalent:

1. $P$ is free.
2. Every path $s$ through $P$ is consistent.
3. Under any free interpretation $H$, an $n$-ary test $p$ is never applied twice to the same $n$-tuple of elements of $U(P)$.

**Lemma 4.3** It is partially decidable whether a schema $P$ is nonfree.

**Proof:** By corollary B (Chapter 1) we can rephrase our definition of freedom as follows: $P$ is free if and only if every finite path is consistent. Thus our procedure is to enumerate finite paths through the flow diagram of $P$ from the start statement and apply the decision procedure of lemma 2.4 to determine whether a path is consistent. If there is a path which is not consistent, then $P$ is not free.

**Lemma F (Paterson)** It is not decidable whether an arbitrary schema is free.
THEOREM 4.4  It is not decidable whether an arbitrary schema $P$ and a free schema $Q$ are strongly equivalent.

Proof: Let $P$ be an arbitrary schema.
Let $Q$ be the schema of figure 16.

\[
\text{START}\quad \gamma \leftarrow f_x \\
\gamma \leftarrow f_y \\
\exists \quad P(\gamma) \quad \text{1}
\]

FIGURE 16

$Q$ is free and diverges under every pointwise interpretation.

$P \equiv Q$ iff $P$ diverges under every pointwise interpretation.

Therefore, if we could decide strong equivalence for schemas $P$ and $Q$, we could decide whether $P$ diverges under every pointwise interpretation. But this has been shown by Paterson [9] to be undecidable. $\square$

The following proposition is an immediate consequence of Garland and Luckham's observation of the relationship between strongly equivalent schemas and their value languages, which we mentioned earlier, and theorem D.
PROPOSITION 4.5 If an arbitrary monadic schema P with a single output variable is strongly equivalent to some free schema, then L(P) is regular. □

In view of theorem D, it is perhaps a bit surprising that the converse of this proposition is not true.

THEOREM 4.6 There is a schema P with regular value language L(P) such that there is no free schema Q which is strongly equivalent to P.

Proof: Let P be the schema of figure 17.

![Diagram of schema P]

FIGURE 17--SCHEMA P
We have introduced the notation

\[ \text{LOOP} \]

to represent an always divergent subschema such as

\[ y \leftarrow fx \]

\[ y \leftarrow fy \]

\[ o \leftarrow \rho(y) \]

We note that the left-hand side of the schema \( P \)
(0-exit of the \( r(u) \) test) computes \( \{g^n f^m | n \geq 1\} \) while the right-hand side of the schema computes \( \{g^n f^m | n \neq m, n, m \geq 1\} \). The value language of \( P \),
\[ L(P) = \{g^n f^m | n, m \geq 1\} \]
\[ = \{g^n f^n | n \geq 1\} \cup \{g^n f^m | n \neq m, n, m \geq 1\}, \]
which is a regular language. We see that for any free interpretation \( I \), if \( I(r(fx)) = 0 \), then \( \text{val}(P, I) = g^n f^x \) for some \( n \geq 1 \), and if \( I(r(fx)) = 1 \), then either \( \text{val}(P, I) = g^n f^m \), for some \( m \) and \( n \) where \( n \neq m \) and \( n, m \geq 1 \), or \( \text{val}(P, I) \) is undefined. Furthermore, \( P \) is clearly not a free schema.

Let \( Q \) be a schema such that \( P \equiv Q \). We must show that \( Q \) cannot be a free schema.
Let $I_1$, $I_2$ be pointwise interpretations of $P$ and $Q$ such that

1. They have the same domain $D$.
2. For each function symbol $h$ of $P$ or $Q$,
   \[ I_1 h = I_2 h \]
3. For each predicate symbol $s$ of $P$ or $Q$ other than $r$,
   \[ I_1 s = I_2 s \]
4. $\forall d \in D$ except $d = I_1(fx) = I_2(fx)$,
   \[ (I_1 r)(d) = (I_2 r)(d) \]
   and $(I_1 r)(fx) = 0$
   and $(I_2 r)(fx) = 1$.

Thus, $val(P, I_1) \neq val(P, I_2)$.

Assume $Q$ does not contain the predicate symbol $r$ applied to the value $fx$. Then
\[ val(Q, I_1) = val(Q, I_2) \]
since the interpretations $I_1$ and $I_2$ are the same everywhere except on the predicate symbol $r$ applied to $fx$. But $P \equiv Q$ so $val(P, I_1) = val(Q, I_1)$
and $val(P, I_2) = val(Q, I_2)$
but $val(P, I_1) \neq val(P, I_2)$.

Thus we have a contradiction and hence $Q$ must contain the predicate symbol $r$, applied to the value $fx$.
Furthermore, for every pointwise interpretation $I$ of $Q$,
if $I(r(fx)) = 0$ then if $val(Q, I)$ is defined,
\[ val(Q, I) = g^r f^x \] for some $n > 1$ and if $I(r(fx)) = 1$ then
if $val(Q, I)$ is defined, $val(Q, I) = g^r f^x$ for some $m, n$
where \( n \neq m, n,m \geq 1 \).

Thus \( Q \) must have the form outlined in figure 18. Notice that the left-hand side of \( Q \) (the subschemas \( S_1, S_2, S_4 \)) corresponds to the left-hand side of \( P \) and the right-hand side of \( Q \) (the subschemas \( S_1, S_3, S_4 \)) corresponds to the right-hand side of \( P \).

![Flowchart](image)

**FIGURE 18—SCHEMA Q**

where \( s_1, s_2, s_3, s_4 \) are (possibly empty) subschemas, and \( w \) is a program variable with value \( fx \). We now consider the schema \( R \) of figure 19, which is essentially the schema composing the left-hand side of schema \( Q \).
FIGURE 19—SCHEMA R

L(R) = \{g^n f^n | n \geq 1\}.

But this is not a regular language. Thus by theorem D, R is not free. By theorem E, this says that there is some n-tuple of elements of the Herbrand universe to which some predicate symbol \( p \) may be applied twice. But then either that is also true in \( Q \), or the 0-exit of the test \( r(w) \) may never be taken. In either case, \( Q \) is not free.  

We have seen so far that it is undecidable whether an arbitrary schema is free, and whether a given free schema is strongly equivalent to an arbitrary schema. We
next show that we cannot even decide whether there is some free schema strongly equivalent to an arbitrary schema.

**THEOREM 4.7** There is no algorithm to decide of an arbitrary schema whether there is a free schema which is strongly equivalent to the given schema.

**Proof:** Let $P$ be a monadic, single output variable schema with input variable $x$. We construct the schema $Q$ by replacing each halt statement of $P$ by the subschema of figure 20.

![Diagram](image)

**FIGURE 20**
\[
L(Q) = \begin{cases} 
g^n & n \geq 1 \text{ if } P \text{ halts under some interpretation} \\
\emptyset & \text{if } P \text{ diverges under every interpretation} 
\end{cases}
\]

By proposition 4.5, if a schema is strongly equivalent to some free schema, then its value language is regular. Furthermore, if the value language of a schema is \(\emptyset\), then the schema diverges under every interpretation and is strongly equivalent to the free schema shown in figure 21.

![Diagram](image)

Thus if we could decide whether \(Q\) was strongly equivalent to some free schema, we could decide whether \(P\) diverges under every interpretation. But this is a well-known undecidable property of schemas. \(\square\)

We shall now introduce another type of restriction which may be placed on a schema. This restriction was also originally defined by Paterson in [9], and was
motivated by the same type of concerns we discussed when we introduced reachability, semifreedom, and freedom. Again it was felt that redundant calculations should be removed and it was hoped that the class of schemas which does not contain these types of inefficiencies might well have solvable decision problems.

We say a schema $P$ is \textbf{liberal} if for every free interpretation $I$ of $P$, no function symbol $f$ is ever applied to the same $n$-tuple of elements of the Herbrand universe more than once within an execution sequence.

It might seem that if an expression is recomputed, then we have an unnecessary calculation and hence a rectifiable inefficiency. We shall see, however, that as in the case of freedom, this is not the case. There are schemas which are inherently nonliberal; that is, they are not strongly equivalent to any liberal schema.

\textbf{Lemma G} (Paterson) There is a procedure which given a liberal schema $P$, effectively constructs a strongly equivalent schema $Q$ which is liberal and free. $\square$

The following corollary is an immediate consequence of lemma G and the construction used by Chandra to prove theorem D.
COROLLARY 4.8 Let $B$ be the class of all monadic liberal schemas which have a single output variable and whose function letters are in a set $F$. Then $L(B)$ is the class of all regular sublanguages of $F$. □

Again, as in the case of free schemas, it follows that if a monadic schema $P$ is strongly equivalent to some liberal schema, then its value language $L(P)$ must be regular. This is an immediate consequence of the above results. Again, as in the case of free schemas, however, the converse is not true.

COROLLARY 4.9 If an arbitrary monadic schema $P$ with a single output variable is strongly equivalent to some liberal schema, then $L(P)$ is regular. □

COROLLARY 4.10 There is a schema $P$ with regular value language $L(P)$ such that there is no liberal schema $Q$ which is strongly equivalent to $P$.

Proof: Let $P$ be the schema used in the proof of theorem 4.6. $L(P)$ is regular. Assume $P$ is strongly equivalent to some liberal schema $Q$. By lemma G, $Q$ is strongly equivalent to some free liberal schema $R$. Thus $P$ is strongly equivalent to a free schema $R$, contradicting theorem 4.6. □
We are now ready to begin our consideration of the relationship between several semantically restricted classes of schemas. We shall be particularly interested in seeing when such additional restrictions are sufficient to enable us to decide membership in the classes or cause there to be effective translations between the classes.

We begin by considering the closely related classes $\mathcal{L}$ and $\mathcal{I}$. We have already mentioned (lemma 6) that $\mathcal{L}$ is effectively translatable into $\mathcal{I}$.

We introduce the notion of a syntactic repetition. This is intended to represent not only the repeated application of a predicate to a set of values, but also the repetition of the application of a predicate to the same set of syntactic objects. A test statement $k. p(y_1, \ldots, y_n)l_1, r_1$ is a syntactic repetition if there is a test statement $m. p(y_1, \ldots, y_n)l_2, r_2$ and there is a path segment of length at least one from statement $m$ to statement $k$, and $y_1, \ldots, y_n$ are not assigned to on this segment. We note here that this is even more restrictive than Paterson's definition of an immediate repetition which required only that a predicate be reapplied to the same set of values within an execution sequence. Thus an immediate repetition represented an instance of illiberality. A syntactic repetition as we have defined it represents an
instance of illiberality only if the segment from statement $m$ to statement $k$ is contained in an execution sequence. We further note that it is clearly decidable whether a schema contains a syntactic repetition.

**LEMMA 4.11** Let $P$ be a liberal schema. Then $P$ is free if and only if it does not contain any syntactic repetition of a test.

**Proof:** If $P$ is liberal and free, then every path through $P$ is an execution sequence of $P$ and hence by theorem E, $P$ cannot contain a syntactic repetition. Let $P$ be liberal and assume it contains no syntactic repetition of a test. Then by the definition of liberality, no value may be recomputed within any execution sequence. Thus for any free interpretation, if $\text{val}(y_i) = \text{val}(y_j)$ then $i = j$. Thus, we have ruled out the possibility of having a repetition of a test $p$ on different syntactic objects having the same value. Furthermore, since $P$ has no syntactic repetitions, it cannot have a repetition of a test on the same syntactic objects with no intervening assignment to one of these elements. But since $P$ is liberal, if there were an intervening assignment statement, the value to be assigned to the variable must never have been computed before. Thus we cannot have any type of repeated test within an execution sequence and therefore by theorem E,
P is free. □

**COROLLARY 4.12** It is decidable whether a liberal schema is free.

**Proof:** The decision procedure is simply to look for any syntactic repetitions of a test in any path through the schema. If there is one, then there must be one for which the number of intervening instructions is no more than the number of instructions in the schema. Thus the search procedure is clearly finite. □

### 4.2 Relationships With Reachable Schemas

We now turn our attention to the class of reachable schemas, and study the affect on other decision problems, of knowing that a schema has this property. We also consider under what conditions knowing that a schema P is in a class \( C \) enables us to decide whether P is in \( \mathcal{R} \) or can be effectively transformed into a schema Q such that \( P \equiv Q \) and \( Q \in \mathcal{R} \).

#### 4.2.1 Free Schemas

**THEOREM 4.13** It is not partially decidable whether a reachable schema is free.
Proof: Assume there was such a partial decision procedure. By lemma 2.6, the set of reachable schemas is recursively enumerable. We can then apply the hypothesized partial decision procedure and select free reachable schemas. But by proposition 4.1, every free schema is reachable, and hence we have a partial decision procedure for freedom, contradicting lemma 4.3 and lemma F. □

THEOREM 4.14 There is no algorithm to decide of a reachable schema and a free schema, whether they are strongly equivalent.

Proof: We present an algorithm which given an arbitrary reachable schema P, and a free schema Q, produces a reachable schema R such that Q is strongly equivalent to R if and only if P halts under every pointwise interpretation.

Let P be a reachable schema with input variable x and output variables \( z_1, \ldots, z_n \).

Let Q be the free schema outlined in figure 22a. This schema halts under every pointwise interpretation.

We construct the schema R by replacing each halt statement in P by the sequence of instructions shown in figure 22b. It follows that R is a reachable schema since P is. Furthermore,
**FIGURE 22a--SCHEMA Q**

**FIGURE 22b**
Q \equiv R \text{ iff } R \text{ halts under every pointwise interpretation}

\text{iff } P \text{ halts under every pointwise interpretation.}

Therefore, if we could decide strong equivalence for a reachable schema and a free schema, we could decide whether a reachable schema halts under every pointwise interpretation. But that would contradict theorem 2.10. \square

4.2.2 Liberal Schemas

We now consider the relationship between \( \mathcal{L} \) and \( \mathcal{R} \).

**Lemma 4.15** It is decidable whether a liberal schema halts under some interpretation.

**Proof:** Let \( P \) be a liberal schema. Let \( Q \) be the liberal schema guaranteed by lemma 6. \( P \equiv Q \). Clearly \( P \) halts under some interpretation if and only if \( Q \) halts under some interpretation. But \( Q \) halts under some interpretation if and only if it contains a halt statement. \square

**Theorem 4.16** It is decidable for a liberal schema, whether an arbitrary statement is reachable.

**Proof:** Let \( P \) be a liberal schema with statements \( s_1, \ldots, s_n \). For each \( k, 1 \leq k \leq n \), we shall construct a
liberal schema \( P_k \) from \( P \), such that \( P_k \) halts under some interpretation, if and only if \( s_k \) is reachable in \( P \). To construct \( P \), we replace each halt statement of \( P \) by a LOOP statement and statement \( s_k \) by a halt statement. If \( s_k \) was anything other than a halt statement, we remove all edges leaving \( s_k \) and delete any portions of the flow diagram which become disconnected as a result of this replacement. Thus we have:

```
START
\downarrow
\vdots
\downarrow
S_k
\downarrow
\vdots
\downarrow
HALT
```

```
START
\downarrow
\vdots
\downarrow
HALT
```

SCHEMA \( P \)

SCHEMA \( P_k \)

We shall now verify that \( P_k \), so constructed, is a liberal schema. This follows from the fact that \( P \) is a liberal schema and hence does not contain any repeated calculations, and the fact that our construction of \( P \)
from $P$ did not add to $P_k$ any calculations not appearing in $P$. Thus $P_k$ contains no repeated calculations, and is therefore liberal.

We shall now demonstrate that $P_k$ halts under some interpretation if and only if $s_k$ is reachable in $P$. $P_k$ contains exactly one halt statement located in the same position that $s_k$ was located in schema $P$. Thus the HALT of $P_k$ is reachable if and only if $s_k$ of $P$ is reachable. Thus $P_k$ has the desired properties.

Therefore it follows from lemma 4.15 that since $P_k$ is liberal, there is a decision procedure to determine whether $P_k$ halts under some interpretation, and hence whether $s_k$ is reachable.  

The next corollary is an immediate consequence of theorem 4.16 and the observation that a schema contains finitely-many statements.

**COROLLARY 4.17** It is decidable whether a liberal schema is reachable. 

Furthermore, we have as an immediate corollary to theorem 4.16, the following result. It is important to notice that the procedure will do just what we want, namely remove unreachable code. Thus, both the letter and the spirit of the definition are fulfilled. We shall see (theorem 4.22) that in some cases we are able to
satisfy the definition without really having the type of
algorithm which we want.

**COROLLARY 4.18** There is a procedure which given a
liberal schema, effectively constructs a strongly
equivalent schema which is liberal and reachable.

We feel that the preceding results are particularly
interesting and even encouraging. They say that if a
schema has a certain desirable property (liberality),
then we can decide whether or not it has other desirable
properties such as freedom and reachability. Furthermore,
we can effectively construct schemas which are strongly
equivalent and possess these properties. It is
especially important since liberality itself is
decidable, and as we have seen, it is one of the few
semantic properties which have been considered which is
decidable.

4.2.3 Schemas Which Halt Under Every Pointwise
Interpretation

We now continue our investigation of the
relationship of the class of reachable schemas to other
semantically restricted classes by considering $\mathcal{H}$. This
is the class of schemas which halt under every pointwise
interpretation. Paterson demonstrated that strong
equivalence is decidable for schemas in $\mathcal{H}$. We have previously demonstrated that being a member of $\mathcal{R}$ is not sufficient to enable us to decide if the schema halts under every pointwise interpretation (Theorem 2.10).

**Lemma 4.19** There is an algorithm which given a schema $\mathcal{P}$ that halts under every pointwise interpretation, produces a schema $\mathcal{T}'(\mathcal{P})$ such that $\mathcal{P} \sim \mathcal{T}'(\mathcal{P})$ and $\mathcal{T}'(\mathcal{P})$ is reachable.

**Proof:** The construction outlined below is essentially the construction used by Paterson [9] as modified by Greibach [5], for going from a schema which halts under every pointwise interpretation to a finite loop-free schema. In particular, the construction will guarantee that the flow diagram of the schema constructed is a tree. We begin with $\mathcal{P}$, an always halting schema, and construct a tree $\mathcal{T}(\mathcal{P})$ of all possible execution sequences for $\mathcal{P}$ under free interpretations. We know that this can be done effectively by Lemma 2.4 and Corollary B. Each node of $\mathcal{T}(\mathcal{P})$ is labelled with an execution sequence such that if a node $n_j$ is an ancestor of $n_j$, then the execution sequence labelling $n_j$ is an initial segment of the execution sequence labelling $n_j$. The tree is finite since it is finitely branching and every execution sequence of $\mathcal{P}$ is finite.

We next construct a tree $\mathcal{T}'(\mathcal{P})$, which is graphically isomorphic to $\mathcal{T}(\mathcal{P})$, and whose nodes are labelled by
instructions of P as follows: if a node of $T(P)$ is labelled with execution sequence $s = (k_1, \ldots, k_m)$, then the corresponding node of $T'(P)$ is labelled by the statement $k_m$. If $k_m$ is a test statement, the branches to its immediate successors are labelled 0 or 1 to correspond to $P$. There is one more step to the required construction. We shall construct a tree $T''(P)$ such that $P \leadsto T''(P)$ and $T''(P)$ is reachable. We note that $T'(P)$ almost has these required properties except that $T'(P)$ may not be a schema. If a node of $T'(P)$ is labelled by a test statement, it may have either one or two successors depending on the number of consistent outcomes of the test at that point. If there is but one successor of a test statement $k_j$ in $T'(P)$, we delete that node when constructing $T''(P)$ and include a branch from its immediate predecessor to its (unique) immediate successor. The tree $T''(P)$ obviously has the required properties since only paths which are execution sequences were considered, and only test statements were pruned from the tree $T'(P)$. □

THEOREM 4.20 It is decidable whether a schema $P$ which halts under every pointwise interpretation is reachable.

Proof: We begin by prefixing each instruction of $P$ with a unique label. We next construct the tree $T'(P)$ as outlined in the proof of lemma 4.19, using the prefixed
instructions as labels on the nodes of $T'(P)$. We note that $T'(P)$ is not necessarily a schema as some test statements may have only a single immediate successor. However, by our construction, we are guaranteed that each instruction in $T'(P)$ is a reachable instruction of $P$. We let $M$ denote the set of prefixes appearing in $P$, and $M'$ the corresponding set for $T'(P)$. Then $P$ is reachable if and only if $M = M'$. \(\square\)

4.2.4 Semifree Schemas

We know that $\mathcal{D}$ is a proper subset of $\mathcal{R}$. We now investigate this relationship.

**Theorem 4.21** It is not decidable whether a reachable schema is semifree.

**Proof:** Let $P$ be an arbitrary schema. We construct a reachable schema $Q$ such that $Q$ is semifree if and only if $P$ is semifree. Therefore, we cannot decide whether a reachable schema is semifree, else we could decide whether an arbitrary schema is semifree, contradicting theorem 3.8.

Assume $P$ contains $n$ test statements designated $t_1, \ldots, t_n$. The 0-successor of statement $t_k$ will be denoted by $k_0$, and the 1-successor will be denoted by $k_1$. Let $t_1$ be the first test statement encountered under every
interpretation (i.e. \( t_1 \) is the test statement nearest to START).

Let \( p \) be a predicate symbol not appearing in \( P \).
Let \( q \) be a 2n-exit predicate symbol not appearing in \( P \).
The exits of \( q \) are labelled \( t_10, t_20, \ldots, t_n0, t_11, \ldots, t_n1 \).

We shall construct \( Q \) from \( P \) by properly inserting \( 2n+1 \) copies of the test statement \( p(x) \) and one copy of the test statement \( q(x) \) in the flow diagram of \( P \) as follows:

1. Insert an initializing statement \( \overline{y} \leftarrow x \) followed by the statement \( p(x) \) immediately after the start statement of \( P \). The 0-successor of this test is the statement which is the successor of the start statement in \( P \); we shall designate that statement \( a \). The 1-successor of the \( p(x) \) statement is the test statement \( q(x) \).

2. Insert one copy of the test statement \( p(x) \) as the 0-successor of each test statement \( t_k \), \( k = 1, \ldots, n \). Both the 0- and 1-successors of \( p(x) \) will be the statement \( k \) of \( P \). Similarly, we insert one copy of \( p(x) \) as the 1-successor of \( t \). Both the 0- and 1-successors of this copy of the test statement \( p(x) \) will be statement \( k_1 \) of \( P \).

3. The \( 2n \) successors of the test statement \( q(x) \) are the \( 2n \) copies of \( p(x) \) inserted in step 2 above.

Thus we have constructed the schema \( Q \) whose outline is shown in figure 23. Intuitively we can think of \( Q \) as being divided into \( 2n+1 \) segments of code, the entrance to each segment being controlled by the interpretation of
FIGURE 23--SCHEMA Q
the initial p(x) test and the q(x) test.

We shall say a statement s of schema Q is in the segment controlled by the test t_k-0 if there is a path segment from the 0-exit of t_k to s which includes no test t_j, 1 ≤ j ≤ n, except possibly s.

We define the segment controlled by t_k-1 analogously.

We shall now demonstrate that Q so constructed is indeed a reachable schema. Let s be an arbitrary statement of Q.

**case 1 s is the start statement; it is thus reachable.**

**case 2 s is an assignment, input, output, halt statement, or test statement other than q(x).**

2a) If s is a statement in the initial segment of Q, i.e. the subschema from START through t_4 including the test q, then any pointwise interpretation I such that (Ip)(x) = 0 causes s to be executed. Hence such a statement s is reachable.

2b) If s is a statement in the segment controlled by t_k-0, k=1,...,n, then any pointwise interpretation I such that (Ip)(x) = 1 and (Iq)(x) = t_k0 will cause s to be executed. Hence such a statement s is reachable.

2c) If s is a statement in the segment controlled by t_k-1, k=1,...,n, then any pointwise interpretation I such that (Ip)(x) = 1 and (Iq)(x) = t_k1 will cause s to be executed. Hence such a statement s is reachable.
case 3: s is the test statement q(x). Any pointwise interpretation such that (Ip)(x) = 1 will cause s to be executed and therefore it is reachable.

It remains to demonstrate that Q is semifree if and only if P is semifree. To do this, we will first show that except for path segments in the initial segment of Q, or path segments of the form \( \langle (px, 0), k_j \rangle \) \( k=1, \ldots, n; j=0,1 \), every path segment of length one in Q is contained in some execution sequence \( \sigma(Q, I) \) for which \((Ip)(x) = 1\).

We note that since x is an input variable and hence may never be assigned a new value, the value of \((Ip)(x)\) is constant throughout a computation.

We let \( I_{t_{k,j}} \) denote a pointwise interpretation such that

\[(I_{t_{k,j}} p)(x) = 1\]

and

\[(I_{t_{k,j}} q)(x) = t_{k,j} \]

Then, since under interpretation \( I_{t_{k,j}} \), after executing the initializing statement followed by p(x) and q(x), the next code to be executed will be the segment of Q controlled by \( t_{k-j} \), if \( t_m \) is the test statement in that segment, then clearly either exit may be taken from it. Furthermore, for any interpretation I such that \((Ip)(x) = 0\), all paths of length one in the initial segment may be executed as well as either exit from test \( t_1 \).
We shall now demonstrate that $P$ is semifree if and only if $Q$ is semifree. As described in the preceding discussion, it is only necessary to show that $P$ is semifree if and only if every path segment of length one in $Q$ of the form $<(p_x, 0), k_0; >$ $k=1, \ldots, n; i=0, 1$ is included in some execution sequence. Assume $P$ is semifree. Then for the $i$-exit of each test $t_k$, $1 \leq k \leq n$, $i=0, 1$, there is an interpretation $I$ of $P$ such that the path segment from $t_k$ to $k_0$ is included in the execution sequence $\sigma(P, I)$. Let $I'$ be an interpretation of $Q$ which agrees with $I$ on the symbols of $P$ and for which $(I'p)(x) = 0$. Then the execution sequence for $Q$ under $I'$ will be exactly the same as the execution sequence for $P$ under $I$ except that $\sigma(Q, I)$ will include repeated tests of $p(x)$. Since $x$ is an input variable and $(Ip)(x) = 0$, then the 0-exit will always be taken from the test $p(x)$. Thus $<(p_x, 0), k_0; >$ and $<(p_x, 0), k_i; >$ $k=1, \ldots, n$ are included in some execution sequence of $Q$ and therefore $Q$ is semifree.

Assume $P$ is not semifree. Then there is a path segment $<(t_k, i), k_0; >$ $k=1, \ldots, n; i=0, 1$ in $P$ which is not included in any execution sequence of $P$. Thus in $Q$ the path segment $<(t_k, i), (p_x, 0), k_0; >$ is not included in any execution sequence of $Q$. Therefore the path segment of length one from the 0-exit of the test $p(x)$ following $t_k-i$, to $k_0$ is not included in any execution sequence.
Thus $Q$ is not semifree.

Thus we have demonstrated that for an arbitrary schema $P$, we can construct a reachable schema $Q$ such that $P$ is semifree if and only if $Q$ is semifree. Therefore we cannot decide whether a reachable schema is semifree, else we could decide whether an arbitrary schema is semifree, contradicting theorem 3.6. □

We shall now discuss an interesting situation which underscores the necessity of considering both the letter and the spirit of a result. Our next theorem demonstrates that if we augment our model to include equality tests and constants, then for any reachable schema, we can construct a strongly equivalent semifree schema. At first glance, that seems like a very desirable situation. It seems to say that if we know that we do not have any unreachable code, then we can effectively get rid of any unnecessary tests. In fact the theorem does not really say that at all. Consideration of the construction used in the proof will verify that in fact just the opposite is happening! Instead of deleting unnecessary tests, we have added additional tests in order to force the newly constructed schema to be semifree.

We would really like to begin with a reachable schema, and if it is not semifree, delete exactly the
unnecessary tests and be left with a "reduced" strongly equivalent semifree schema. But we know that that is not possible, else we could decide whether a reachable schema is semifree simply by seeing whether or not the schema is modified by the procedure.

**THEOREM 4.22** There is an algorithm which given a reachable schema $P$ in $\mathcal{C}_r$, constructs a semifree schema $Q$ in $\mathcal{C}_r$ such that $P \equiv Q$.

**Proof:** Assume $P$ contains $n$ test statements, designated $t_1, \ldots, t_n$. Let $k_o$ and $k_1$ denote test $t_k$'s 0- and 1-successor respectively, $k=1, \ldots, n$.

Let $SW$ be a variable which does not appear in $P$.

Let $a$ denote the statement which is the successor statement of START in $P$.

Let $q$ be an $n$-exit predicate not in $P$, with exits labelled $1, \ldots, n$.

We shall construct $Q$ from $P$ as follows:

For each test statement $t_k$ of $P$, insert the test statement $SW = 0$ between $t_k$ and $k_o$. The 0-successor (false exit) of the newly inserted test is $k_o$, while the 1-successor (true exit) is a new assignment statement $SW \leftarrow 1$. The successor of this assignment statement is $a$. Similarly, we insert a copy of the test $SW = 0$ between $t_k$ and $k_1$, the 0-successor of this test being $k_1$, and the 1-successor being another copy of the assignment.
statement \( SW \leftarrow 1 \). Again, the successor of this assignment statement is \( a \). All edges which enter \( k_i, k=1, \ldots, n; i=0,1 \) in \( P \), enter the \( SW = 0 \) test which precedes \( k_i \) in \( Q \).

Insert an initializing statement \( \bar{y} \leftarrow x \) and an assignment statement \( SW \leftarrow 0 \) immediately following the start statement. The successor of the statement \( SW \leftarrow 0 \) will be the \( n \)-exit test \( q(x) \). Exit \( k \) of this test will enter test \( t_k \). The schema \( Q \), so constructed from \( P \), is outlined in figure 24. We note that the sole purpose of the addition of the variable \( SW \), and its assignment and testing is to assure that \( Q \) be semifree. Regardless of which exit of the test \( q \) is taken, after testing the appropriate statement \( t_k \), \( SW \) is tested and since it was initialized to 0, the 1-exit is taken from the test. After resetting \( SW \), control goes to statement \( a \) and then simulation of schema \( P \) begins.

A detailed argument similar to the one used in lemma 3.10 can be made to verify that schema \( Q \) is indeed semifree.

Intuitively, the test \( q \) guarantees that either exit of each test \( t_k \) can be taken, and that the 1-exit of each copy of test \( SW = 0 \) can be taken. Since \( P \) is a reachable schema, there is an edge in \( P \) which enters statement \( k_i, k=1, \ldots, n; i=0,1 \) which is traversed under some interpretation of \( P \). Thus in \( Q \), each test \( SW = 0 \) is reachable after \( SW \) has been assigned the value 1 and simulation of \( P \) has begun. Therefore, the 0-exit of each
FIGURE 24—SCHEMA Q
test SW = 0 may be traversed under some interpretation and hence Q is semifree.

It remains to demonstrate that P = Q. Let I be a pointwise interpretation such that

(Iq)(x) = k \quad k \in \{1, \ldots, n\}

and (It_k) = i \quad i \in \{0, 1\}

After the initializing segment of Q in which all of the program variables are initialized to x, SW is initialized to 0, the k-exit is selected from q, the i-exit is taken from t_k, SW is tested and SW is reset to 1, control goes to statement a. Once the value of SW is set to 1 in Q, it can never be reset to 0 within the schema. Therefore, Q will perform exactly as schema P would under interpretation I, with the addition of periodic testing of the value of SW. Thus, for every I, val(P, I) = val(Q, I) and hence P = Q. \square

The following corollary is an immediate consequence of corollary 3.11.

**COROLLARY 4.23** There is no algorithm to decide of a reachable schema P and a semifree schema Q, whether P = Q. \square
4.3 Relationships With Semifree Schemas

Having discussed the relationship between the class \( R \) and the classes \( J, L, H, \) and \( A \), we now consider the relationship between \( A \) and these other semantically restricted classes. We begin by considering the relationship between \( A \) and \( J \). The next two results actually subsume theorems 4.13 and 4.14. We have included proofs of 4.13 and 4.14 because we felt it was part of the natural progression of questions to be considered. Furthermore, since results of our inquiry into the relationship between \( R \) and \( L \) are used in the proofs of results about the relationship between \( A \) and \( L \), we felt it better to consider the class \( R \) before considering \( A \).

4.3.1 Free Schemas

**Theorem 4.24** It is not partially decidable whether a semifree schema is free.

**Proof:** Assume such a partial decision procedure existed. We have shown (Theorem 3.7) that it is partially decidable whether a schema is semifree. Thus we can apply the hypothesized partial decision procedure and select the semifree, free schemas. But by proposition 4.1, every free schema is semifree, and hence
we have a partial decision procedure for freedom, contradicting lemmas 4.3 and F. □

**THEOREM 4.25** There is no algorithm to decide of a semifree schema and a free schema, whether they are strongly equivalent.

**Proof:** The proof is similar to that of theorem 4.14 and thus we just outline the argument here. Let $P$ be a semifree schema with input variable $x$ and output variables $z_1, \ldots, z_n$. $Q$ is the free schema of theorem 4.14. We construct $R$ from $P$ as in theorem 4.14. Then $Q \equiv R \iff R$ halts under every pointwise interpretation 

iff $P$ halts under every pointwise interpretation.

Thus, if we could decide whether a semifree schema were strongly equivalent to a free schema, we could decide whether a semifree schema halts under every pointwise interpretation. But this contradicts theorem 3.10. □

4.3.2 **Liberal Schemas**

We shall now consider the relationship between semifree schemas and liberal schemas.

**LEMMA 4.26** It is decidable whether an arbitrary test statement of a liberal scheme is necessary.
Proof: Let $P$ be a liberal schema.
Let $t_k$ be an arbitrary test statement of $P$.
Let $y$ be a program variable which does not appear in $P$.
Let $f$ be a function symbol which does not appear in $P$.
Let $x$ be an input variable of $P$.
We construct a liberal schema $P'$ from $P$ as follows:
1) Insert between the start statement and its successor, the assignment statement
   \[ y \leftarrow fx \]
2) If we have the subschema:

![Diagram]

in schema $P$, we replace it with the following subschema in schema $P'$:

![Diagram]
$P_K$ is liberal because $P$ is liberal and the added assignment statements update $y$ using a function symbol $f$ which does not appear in $P$. We see that $t_K$ is a necessary test in $P$ if and only if both copies of the instruction $y \leftarrow fy$ are reachable in $P$.

By theorem 4.16 it is decidable whether these instructions are reachable, and hence it is decidable whether $t_K$ is a necessary test in $P$. □

The following corollary is an immediate consequence of the preceding lemma and proposition 3.1.

**COROLLARY 4.27** It is decidable whether a liberal schema is semifree. □

**COROLLARY 4.28** There is a procedure which given a liberal schema, effectively constructs a strongly equivalent schema which is liberal and semifree.

**Proof:** This is an immediate consequence of lemma 4.26. □

4.3.3 Schemas Which Halt Under Every Pointwise Interpretation

Finally we consider the relationship between semifree schemas, and schemas which halt under every pointwise interpretation.
THEOREM 4.29  There is an algorithm which given a schema P that halts under every pointwise interpretation, produces a schema \( T'(P) \) such that \( P \sim T'(P) \) and \( T'(P) \) is semifree.

Proof: The proof of this lemma follows directly from the proof of lemma 4.19 with the observation that \( T'(P) \) is a semifree schema. The construction of \( T'(P) \) from \( T'(P) \) consists of removing from \( T'(P) \) those test statements from which only one exit may be taken. These are exactly the unnecessary tests. \( \square \)

COROLLARY 4.30  It is decidable whether a schema which halts under every pointwise interpretation is semifree.

Proof: This follows immediately from theorem 4.20 and proposition 4.1. \( \square \)
CHAPTER 5

PRESERVATION OF SEMANTIC PROPERTIES BY SAMENESS RELATIONS

So far we have considered several classes of schemas whose members possess certain semantic properties. We now consider four notions of "sameness", three of which were introduced in Chapter 1; the fourth we shall introduce shortly. We discuss the following question relative to these relations: If P is a schema in the class C, and rel(P,Q) where rel is one of the sameness relations, does it follow that Q ∈ C?

We say that P and Q are weakly equivalent, denoted \( P \equiv Q \), if and only if for every pointwise interpretation \( I \), \( \text{val}(P,I) = \text{val}(Q,I) \) whenever both values are defined.

5.1 Isomorphism

We begin our discussion by considering isomorphism which is the strongest of the sameness relations to be investigated.

Theorem 5.1 Let P and Q be schemas such that \( P \equiv Q \). Then

a) If P is free, then Q is free.

b) If P is liberal, then Q is liberal.

c) If P halts under every interpretation, then Q halts under every interpretation.

d) P semifree does not necessarily imply that Q is
semifree.

e) $P$ reachable does not necessarily imply that $Q$ is reachable.

Proof: a) Assume $Q$ is not free. Then there is a free interpretation $H$, such that

$$(\text{START}, s_1, \ldots, s_{k-1}, p(u_1, \ldots, u_m), s_k, \ldots, s_{k-1}, p(v_1, \ldots, v_m))$$

is an initial segment of $\sigma(Q, H)$ and

$$\text{val}(u_1, k) = \text{val}(v_1, t)$$

$$\vdots$$

$$\text{val}(u_m, k) = \text{val}(v_m, t).$$

But $P \not\subseteq Q$ so this sequence must also be an initial segment of $\sigma(P, H)$. Thus $P$ contains a repeated test under interpretation $H$, and hence $P$ is not free.

b) The argument is essentially the same as the one used in part a, except that in this case there would be a repeated calculation, rather than a repeated test.

c) This result is immediate and is included only for the purpose of comparing $H'$ to the other semantically restricted classes.

d) Let $P$ be the semifree schema shown in figure 25.

Let $Q$ be the schema shown in figure 26. $Q$ is not semifree, as the test statement designated (a) is not necessary.

We see that $P \not\subseteq Q$, for if $H$ is a free interpretation such that $(Hp)(fx) = 0$, then

$$\text{seq}(P, H) = \text{seq}(Q, H) = \langle \text{START}, fx, p(fx), f^2x, \text{HALT} \rangle$$
FIGURE 25—SCHEMA P

FIGURE 26—SCHEMA Q
and \( \text{val}(P,H) = \text{val}(Q,H) = f^2 x \).

and if \((Hp)(fx) = 1\) then

\( \text{seq}(P,H) = \text{seq}(Q,H) = \langle \text{START}, fx, p(fx), p(fx), p(fx), \ldots \rangle \)

and \( \text{val}(P,H), \text{val}(Q,H) \) are both undefined.

e) This follows directly from the proof of part d and the observation that Q is not reachable as the statement designated \( b \) is not reachable. \( \square \)

5.2 Functional Similarity

**THEOREM 5.2** Let \( P \) and \( Q \) be schemas such that \( P \sim Q \), then

a) If \( P \) is liberal, then \( Q \) is liberal.

b) If \( P \) halts under every interpretation, then \( Q \) halts under every interpretation.

c) \( P \) free does not necessarily imply that \( Q \) is free.

d) \( P \) semifree does not necessarily imply that \( Q \) is semifree.

e) \( P \) reachable does not necessarily imply that \( Q \) is reachable.

**Proof:** a) This follows by the argument used in theorem 5.1a,b.

c) Let \( P \) and \( Q \) be the schemas used in the proof of theorem 5.1d. We note that \( P \) is free and \( Q \) is not free.

If \( H \) is a free interpretation such that \((Hp)(fx) = 0\), then

\( \text{funceq}(P,H) = \text{funceq}(Q,H) = \langle \text{START}, fx, f^2 x, \text{HALT} \rangle \)

and \( \text{val}(P,H) = \text{val}(Q,H) = f^2 x \).
and if \((H_p)(f^x)=1\) then
\[\text{funcseq}(P,H)=\text{funcseq}(Q,H)=<\text{START},f^x>\]
and \(\text{val}(P,H), \text{val}(Q,H)\) are both undefined.

d,e) Follow as immediate corollaries to theorem 5.1d and e and the fact that \(P \not\equiv Q\) implies \(P \sim Q\). □

5.3 Strong Equivalence

**THEOREM 5.3** Let \(P\) and \(Q\) be schemas such that \(P \equiv Q\), then

a) If \(P\) halts under every interpretation then \(Q\) halts under every interpretation.

b) \(P\) liberal, does not necessarily imply that \(Q\) is liberal.

c) \(P\) free does not necessarily imply that \(Q\) is free.

d) \(P\) semifree does not necessarily imply that \(Q\) is semifree.

e) \(P\) reachable does not necessarily imply that \(Q\) is reachable.

**Proof:** b) Let \(P\) be the liberal schema shown in figure 27, and let \(Q\) be the schema of figure 28. \(Q\) is clearly illiberal.

Let \(H\) be a free interpretation. If \((H_p)(f^nx)=1\) for all \(n \geq 1\) then \(\text{val}(P,H)\) and \(\text{val}(Q,H)\) are both undefined.

If \(k = \not\in n\) such that \((H_p)(f^nx)=0\) then
\[\text{val}(P,H)=\text{val}(Q,H)=(f^{K_f+1}x, f^Kx)\]
Thus \(P \equiv Q\).
FIGURE 27--SCHEMA P

FIGURE 28--SCHEMA Q
c, d, e) Follow as immediate corollaries to theorem 5.2

5.4 Weak Equivalence

Theorem 5.4 Let P and Q be schemas such that P $\equiv$ Q, then

a) P halts under every interpretation does not necessarily imply that Q halts under every interpretation.
b) P liberal does not necessarily imply that Q is liberal.
c) P free does not necessarily imply that Q is free.
d) P semifree does not necessarily imply that Q is semifree.
e) P reachable does not necessarily imply that Q is reachable.

Proof: a) Let P be the schema shown in figure 29 which halts under every interpretation and let Q be the schema of figure 30. Q does not halt under any interpretation H for which $(Hp)(fx)=1$.

However, for any interpretation H such that $(Hp)(fx)=0$, Q halts and $\text{val}(P,H)=\text{val}(Q,H)=fx$.

b, c, d, e) These results follow as immediate corollaries to theorem 5.3 b, c, d and e and the fact that $P \equiv Q$ implies $P \sim Q$. □
FIGURE 29--SCHEMA P

FIGURE 30--SCHEMA Q
5.5 Decidability Results

There are several interesting observations which follow more or less immediately from the preceding results. The following four corollaries are examples.

**COROLLARY 5.5** It is not decidable whether an arbitrary schema is isomorphic to some free schema.

**Proof:** This follows immediately from theorem 5.1a. If \( P \not\cong Q \) then \( P \) is free if and only if \( Q \) is free. Therefore if it were decidable whether an arbitrary schema \( S \) is isomorphic to some free schema, it would be decidable whether \( S \) is free. □

**COROLLARY 5.6** It is not decidable whether a reachable schema or a semifree schema is isomorphic to some free schema. □

**COROLLARY 5.7** It is decidable whether an arbitrary schema is isomorphic to some liberal schema.

**Proof:** By theorem 5.1b, if \( P \not\cong Q \), then \( P \) is liberal if and only if \( Q \) is liberal. But it is decidable (Paterson [9]) whether a schema is liberal. □

**COROLLARY 5.8** It is not decidable whether an arbitrary schema is strongly equivalent to some schema which halts
under every interpretation.

Proof This follows immediately from theorem 5.3a and the fact that it is undecidable whether a schema halts under every interpretation.

What do the preceding results tell us about the nature of these semantic properties? Semifreedom and reachability are apparently properties which are not associated with the function being computed, but are rather properties of the method of calculation. The separation here is quite distinct.

Halting under every interpretation, on the other hand, is intimately associated with the function being computed, and is independent of the method of computation.

Liberality and freedom fall somewhere in between. We know that there are schemas which are inherently nonfree or nonliberal in the sense that they are not strongly equivalent to any free or liberal schema. Furthermore, both properties are preserved by isomorphism, while neither property is preserved by strong equivalence.
CHAPTER 6

POINTWISE AND FUNCTION INTERPRETATIONS

Earlier in this work, we introduced two different types of interpretations. Most of the properties of schemas which we have discussed so far are defined in terms of pointwise interpretations. The results of this chapter compare pointwise and function interpretations of schemas.

It is clear that corresponding to every function interpretation and input vector, there is a pointwise interpretation. Thus we have the following results as immediate corollaries of this observation.

COROLLARY 6.1  a) If $P$ is a uniformly reachable schema, then $P$ is reachable.
b) If $P$ is a uniformly semifree schema, then $P$ is semifree.
c) If $P$ is a uniformly free schema, then $P$ is free.

Our next theorem will show that corresponding to the set of pointwise interpretations, there is a single effectively constructable function interpretation. The converses of the results in corollary 6.1 follow as corollaries to this theorem.
THEOREM 6.2 Let $P$ be a program schema with $n$ input variables and let $C_P$ be the set of all finite consistent paths through its flow diagram from the start statement. Then there exists an effectively constructable function interpretation $\mathcal{J}$ with domain $D$ such that for every $s_j \in C_P$, there is a $d \in D^*$ such that $s_j$ is an initial segment of $\sigma(P, \mathcal{J}, d)$.

Proof: We shall first outline the construction briefly; a detailed proof will follow this discussion. Our procedure will construct $\mathcal{J}$ in terms of $D$, as the union of chains of partially defined functions and predicates. The chains are defined inductively by stages, each stage corresponding to a finite consistent path. $D$ is well-ordered, with elements $d_0, d_1, \ldots$.

If $P$ has infinitely many finite paths from $\text{START}$, then the domain $D$ of the function interpretation $\mathcal{J}$ will be countably infinite. If, however, there are only finitely many finite paths from $\text{START}$, then clearly $P$ diverges under no interpretation. This is equivalent to saying that $P$ halts under every interpretation. Paterson [9] has shown that in that case there is a uniform bound $N$ such that the length of any execution sequence in $P$ is no longer than $N$. There are then at most $F(N)$ finite paths from $\text{START}$ where $F$ is a calculable function. If the schema requires $k$ input variables, then

$$|D| \leq [k^*F(N)] + [N^*F(N)] + 1.$$
so the domain will be finite and of determinable size. Let \( f_1, \ldots, f_w \) be the function symbols of \( P \), with \( a_1, \ldots, a_w \) their respective arities. Let \( p_1, \ldots, p_v \) be the predicate symbols of \( P \), with \( b_1, \ldots, b_v \) their respective arities. Let \( \mathcal{C}_P = \{ s_j \} \) be the enumeration guaranteed by lemma 2.4.

The construction will define by stages, where stage \( j \) corresponds to path \( s_j \), the sets:

\[
F(i, j) \subseteq D^{a_i} + 1 \quad i = 1, \ldots, w; \ j = 0, 1, \ldots \\
P(i, j) \subseteq D^{b_i} \times \{0, 1\} \quad i = 1, \ldots, v; \ j = 0, 1, \ldots 
\]

and will define

\[
\mathcal{A}_h = \bigcup_{j \in N} F(h, j) \\
\mathcal{A}_h = \bigcup_{j \in N} P(h, j)
\]

For \( s_j \in \mathcal{C}_P \) where \( s_j \) has length \( m \), let \( s_j(1), \ldots, s_j(m) \) denote the sequence of instructions executed in \( s_j \).

At each stage \( j \), we have a list \( L(j) \). Intuitively, \( L(j) \) is the set of elements of \( D \) which have been assigned as input values or as the value of a function symbol \( f_k \) applied to an \( a_k \)-tuple of values by the end of stage \( j \) and \( \text{val}_j(u, k) \) will denote the value which the variable \( u \) is to be assigned under interpretation \( \mathcal{I} \), after instruction \( s_j(k) \) is executed. Formally, we assume \( P \) contains \( c \) variables \( u_1, \ldots, u_c \). If \( u_k \) is the \( b \)-th input variable, then \( \text{val}_j(u_k, b) = a_{t \cdot b - 1} \), else \( \text{val}_j(u_k, 0) \) is undefined.

Let \( s_0 \) be the sequence \( \langle \text{START} \rangle \).
Stage 0 \[ F(i,0) = \{ \langle d_0, \ldots, d_i, d_0 \rangle \} \quad 1 \leq i \leq w \]
\[ P(i,0) = \{ \langle d_0, \ldots, d_i, 0 \rangle \} \quad 1 \leq i \leq v \]
\[ L(0) = \{ d_0 \} \]

Stage \( j \)

Let \( \vec{d}_t \) denote the \( n \) place input vector \( \langle d_{t_1}, d_{t_2}, \ldots, d_{t_{n-1}} \rangle \) where \( t = \min \{ m : d_m \in D-L(j-1) \} \).

We use \( F(i,j,k) \), \( P(i,j,k) \), and \( L(j,k) \) to indicate that we are considering the \( k \)-th instruction of \( S_j \).

Initialize \( L(j,0) = L(j-1) \cup \{ d_t, \ldots, d_{t+n-1} \} \)
\[ F(i,j,0) = F(i,j-1) \quad i = 1, \ldots, w \]
\[ P(i,j,0) = P(i,j-1) \quad i = 1, \ldots, v \]
\[ \text{val}_j(u_{i,k},0) = d_{t-b-1} u_k \text{ the } b \text{-th input variable} \]

Suppose we have defined \( L(j,k-1) \),
\[ F(1,j,k-1), \ldots, F(w,j,k-1), P(1,j,k-1), \ldots, P(v,j,k-1), \]
\[ \text{val}_j(u_{1,k-1}), \ldots, \text{val}_j(u_{c,k-1}). \]

By definition, \( s_j(0) \) is the start statement. For \( k' = 1, \ldots, m \):
case 1 \[ s_j^*(k) \] is the assignment statement
\[ y \leftarrow f_h(y_1, \ldots, y_{\alpha_h}) \]

\[ F(h, j, k) = \begin{cases} 
F(h, j, k-1) & \text{if there is an } \alpha_h+1 \text{-tuple in} \\
F(h, j, k) & \text{whose first } \alpha_h \\
\text{components are} \\
\text{val}_j^*(y_1, k-1), \ldots, \text{val}_j^*(y_{\alpha_h}, k-1) \\
\end{cases} \]

\[ F(t, j, k) = F(t, j, k-1) \quad t = 1, \ldots, h-1, h+1, \ldots, v \]

\[ L(j, k) = \begin{cases} 
L(j, k-1) & \text{if } F(h, j, k) = F(h, j, k-1) \\
L(j, k-1) \cup \{d!\} & \text{if } F(h, j, k-1) = F(h, j, k-1) \\
\quad \cup \{\text{val}_j^*(y_1, k-1), \ldots, \text{val}_j^*(y_{\alpha_h}, k-1), d'\} \end{cases} \]

\[ P(t, j, k) = P(t, j, k-1) \quad t = 1, \ldots, v \]

\[ \text{val}_j^*(y, k) = d' \]

\[ \text{val}_j^*(u_{j}, k) = \text{val}_j^*(u_{j}, k-1) \quad \text{for } u_{j} \neq y \]

\text{case 2.1} \[ s_j^*(k) \] is the test statement:
\[ p_h(y_1, \ldots, y_{\beta_h}), 1, r \quad 1 \neq r \text{ and } k \neq m. \]

\[ P(h, j, k) = \begin{cases} 
P(h, j, k-1) \cup \{\text{val}_j^*(y_1, k-1), \ldots, \text{val}_j^*(y_{\beta_h}, k-1), 0\} & \text{if } s_j^*(k+1) \text{ is the instruction with address } 1 \\
P(h, j, k-1) \cup \{\text{val}_j^*(y_1, k-1), \ldots, \text{val}_j^*(y_{\beta_h}, k-1), 1\} & \text{if } s_j^*(k+1) \text{ is the instruction with address } r \end{cases} \]
\[ \text{case} \ 2.2 \ \ y_i(k) \ \text{is the test statement:} \]
\[ p_h(y_1, \ldots, y_{b_h})_l, r \quad 1 \neq r \ \text{and} \ k = m. \]
\[ L(j,k) = L(j,k-1) \]
\[ F(t,j,k) = F(t,j,k-1) \quad t = 1, \ldots, w \]
\[ P(t,j,k) = P(t,j,k-1) \quad t = 1, \ldots, v \]
\[ \text{val}_d(u_i,k) = \text{val}_d(u_i,k-1) \ \text{for all} \ u_i, \ 1 \leq i \leq c \]

\[ \text{case} \ 3 \ \ y_i(k) \ \text{is the unconditional branch statement:} \]
\[ p_h(y_1, \ldots, y_{b_h})_r, r \]
\[ L(j,k) = L(j,k-1) \]
\[ F(t,j,k) = F(t,j,k-1) \quad t = 1, \ldots, w \]
\[ P(t,j,k) = P(t,j,k-1) \quad t = 1, \ldots, v \]
\[ \text{val}_d(u_i,k) = \text{val}_d(u_i,k-1) \ \text{for all} \ u_i, \ 1 \leq i \leq c \]

\[ \text{case} \ 4 \ \ y_i(k) \ \text{is the input statement:} \]
\[ y \leftarrow x \]
\[ L(j,k) = L(j,k-1) \]
\[ F(t,j,k) = F(t,j,k-1) \quad t = 1, \ldots, w \]
\[ P(t,j,k) = P(t,j,k-1) \quad t = 1, \ldots, v \]
\[ \text{val}_d(y,k) = \text{val}_d(x,k-1) \]
\[ \text{val}_d(u_i,k) = \text{val}_d(u_i,k-1) \ \text{for} \ u_i \neq y \]
case 5 \( s_j(k) \) is the output statement:
\[
z \leftarrow y
\]
\[
L(j, k) = L(j, k-1)
\]
\[
F(t, j, k) = F(t, j, k-1) \quad t = 1, \ldots, w
\]
\[
P(t, j, k) = P(t, j, k-1) \quad t = 1, \ldots, v
\]
\[
\text{val}_d(z, k) = \text{val}_d(y, k-1)
\]
\[
\text{val}_d(u, k, k) = \text{val}_d(u, k, k-1) \text{ for } u \neq z
\]

case 6 \( s_j(k) \) is a halt statement
\[
L(j, k) = L(j, k-1)
\]
\[
F(t, j, k) = F(t, j, k-1) \quad t = 1, \ldots, w
\]
\[
P(t, j, k) = P(t, j, k-1) \quad t = 1, \ldots, v
\]
\[
\text{val}_d(u, k, k) = \text{val}_d(u, k, k-1) \text{ for all } u \quad 1 \leq i \leq c
\]

After considering statement \( s_j(m) \), we define:
\[
L(j) = L(j, m)
\]
\[
F(t, j) = F(t, j, m) \cup \{<d_{i_1}, \ldots, d_{i_\alpha}, d> \mid \text{for every } a_\alpha \text{-tuple of elements of } L(j) \text{ such that there is no } d \in D \text{ such that }<d_{i_1}, \ldots, d_{i_\alpha}, d> \in F(t, j, m)\} \quad t = 1, \ldots, w
\]
\[
P(t, j) = P(t, j, m) \cup \{<d_{i_1}, \ldots, d_{i_\beta}, c> \mid \text{for every } b_\beta \text{-tuple of elements of } L(j) \text{ such that there is no } c \in \{0, 1\} \text{ such that }<d_{i_1}, \ldots, d_{i_\beta}, c> \in P(t, j, m)\} \quad t = 1, \ldots, v
\]

This ends the construction of \( P(t, j) \), \( F(t, j) \) and \( L(j) \) for stage \( j \).

We define \( \mathcal{F}_t = \bigcup_{j \in N} F(t, j) \quad t = 1, \ldots, w \).
\[ \mathcal{L}_t = \bigcup_{j \in \mathcal{L}} P(t, j) \quad t = 1, \ldots, v \]

We have still to show that there are indeed single-valued and total. Both of these facts will follow directly from the construction of \( \mathcal{L} \). Notice that \( F(t, j) \subseteq F(t, j+1) \) for \( t = 1, \ldots, w \) and \( j = 1, 2, \ldots \) and \( P(t, j) \subseteq P(t, j+1) \) for \( t = 1, \ldots, v \) and \( j = 1, 2, \ldots \).

First we shall show that \( D \subseteq \mathcal{UL}(j) \). Let \( d_j \in D \). We wish to show that there is some \( k \) such that \( d_j \in L(k) \). At each stage \( j \geq 1 \), the construction adds the next \( n \) elements of \( D \) to \( L(j) \). Thus, in the worst case \( d_j \in L[j^{-}] \) and hence \( d_j \in \mathcal{UL}(j) \).

Since the only things added to an \( L(j) \) are elements of \( D \), it follows immediately that \( \mathcal{UL}(j) \subseteq D \). Thus we have that \( D = \mathcal{UL}(j) \).

We are now ready to show that for each \( t \), \( \mathcal{L}_t \) is single-valued. Assume the contrary. That is, for some \( t \),

\[ \langle d_{i_1}, \ldots, d_{i_{\alpha_t}}, d \rangle \in \mathcal{L}_t \]

and \( \langle d_{i_1}, \ldots, d_{i_{\alpha_t}}, d' \rangle \in \mathcal{L}_t \)

and \( d \neq d' \).

Then there is a \( j \in \mathbb{N} \) such that

\[ \langle d_{i_1}, \ldots, d_{i_{\alpha_t}}, d \rangle \in F(t, j) \]

and \( \langle d_{i_1}, \ldots, d_{i_{\alpha_t}}, d' \rangle \in F(t, j) \)

Let \( z \) denote the smallest such \( j \).

That is, \( \langle d_{i_1}, \ldots, d_{i_{\alpha_t}}, d \rangle \in F(t, z) \)

\[ \langle d_{i_1}, \ldots, d_{i_{\alpha_t}}, d' \rangle \in F(t, z) \]
but \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d \rangle \notin F(t, z-1) \)
or \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d' \rangle \notin F(t, z-1) \).

There are only two cases in which an \( F(t, j) \) may be modified. The first case is if there is a \( k \) such that \( s_{\alpha t}(k) \) is an assignment statement. The second case is if we have completed stage \( z \) and there is no \( c \) such that \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, c \rangle \in F(t, z, m) \) where \( m \) is the length of path \( s_{\alpha t}^t \).

**Case 1.** There is a \( k \) such that \( s_{\alpha t}(k) \) is an assignment statement of the form:

\[
y \leftarrow f_{\alpha t}(y_1, \ldots, y_{\alpha t})
\]

and \( \text{val}_{z}(y_1, k-1) = d_{i_1} \)

\[\vdots\]

\( \text{val}_{z}(y_{\alpha t}, k-1) = d_{i_{\alpha t}} \)

and \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d \rangle \in F(t, z, k-1) \)

but \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d' \rangle \notin F(t, z, k-1) \)

Then by our construction \( F(t, z, k) = F(t, z, k-1) \) and hence \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d' \rangle \notin F(t, z, k) \) and hence \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d' \rangle \notin F(t, z, m) \) in this case.

**Case 2.** \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d \rangle \in F(t, z, m) \)

and \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d' \rangle \notin F(t, z, m) \)

Then by our construction, \( \langle d_{i_1}, \ldots, d_{i_{\alpha t}}, d' \rangle \notin F(t, z) \).

Thus, since these two cases represent the only two methods by which an \((\alpha t+1)\)-tuple may be added to an \( F(t, j) \), we have a contradiction to our hypothesis and thus for every \( t \in \{1, \ldots, w\}, \mathcal{S}_{\alpha t} \) is single-valued.
We will next show that for each t, \( \neg \mathcal{P}_t \) is total. Assume the contrary. That is, there is an \( a_t \)-tuple of elements of \( D \), \((d_{i_t}, \ldots, d_{\alpha_t})\) such that there is no \( d \in D \) such that \( \langle d_{i_t}, \ldots, d_{\alpha_t}, d \rangle \in \mathcal{P}_t \).

Let \( i_{sp} \) be the largest subscript of the tuple. Since \( U L(j) = D \), there is a \( k \) for which \( d_{i_{sp}} \in L(k) \). Let \( c \) be the smallest such \( k \). That is \( d_{i_{sp}} \in L(c) \) but \( d_{i_{sp}} \notin L(c - 1) \). Since \( P \) takes \( n \) input values, \( c \leq \left\lceil \frac{L_{sp}}{n} \right\rceil \).

By assumption there is no \( d \) such that \( \langle d_{i_t}, \ldots, d_{\alpha_t}, d \rangle \in F(t, c, m) \) but \( d_{i_t}, \ldots, d_{\alpha_t} \in L(c) \).

Under these circumstances our procedure makes \( \langle d_{i_t}, \ldots, d_{\alpha_t}, d_0 \rangle \in F(t, c) \) and therefore \( \langle d_{i_t}, \ldots, d_{\alpha_t}, d_0 \rangle \in \mathcal{P}_t \). This contradicts our assumption, and hence \( \neg \mathcal{P}_t \) is total for each t.

We will next demonstrate that for each t, \( \neg \mathcal{P}_t \) is single-valued. Assume the contrary. That is, for some t \( \langle d_{i_t}, \ldots, d_{b_t}, 0 \rangle \in \neg \mathcal{P}_t \) and \( \langle d_{i_t}, \ldots, d_{b_t}, 1 \rangle \in \neg \mathcal{P}_t \).

Then there is a \( j \in N \) such that \( \langle d_{i_t}, \ldots, d_{b_t}, 0 \rangle \in P(t, j) \) and \( \langle d_{i_t}, \ldots, d_{b_t}, 1 \rangle \in P(t, j) \).

Let \( z \) denote the smallest such \( j \). There are only two ways in which a \((b_t + 1)\)-tuple may be added to some \( P(t, j) \). The first case is if there is a \( k \) such that \( s_{i_t}(k) \) is a test statement

\[ P_x(y_1, \ldots, y_{b_t})_{l, r} \quad l \neq r \text{ and } k \neq m \]
where \( m \) is the length of path \( s_Z \). Since our procedure uses distinct elements of the domain for each path, and since our program schema model does not include constants, there is only one stage of the construction during which a particular \( b_t+1 \)-tuple may be added to any \( P(t,j) \). Furthermore, since each path is consistent, we are guaranteed that within a single path, a given predicate symbol \( p_t \) will not be applied to the same \( b_t \)-tuple of values with contradictory outcomes. Notice that we distinguish between test statements which have two distinct next statements, and those which are effectively unconditional branch statements. We point out that in the latter case we defer the assignment of a value to the \( b_t \)-tuple \( \langle d_{i_1}, \ldots, d_{i_{b_t}} \rangle \) under the predicate \( p_t \). This is done since there may be a test statement

\[
p_t(d_{i_1}, \ldots, d_{i_{b_t}}) \quad l', r'
\]

where \( l' \neq r' \) which is encountered later in the path. If we had arbitrarily selected a value to be assigned to this \( b_t \)-tuple during the unconditional branch, we could now have a conflicting assignment.

The second case is if stage \( z \) has been completed and there is no \( c \in \{0, 1\} \) such that \( \langle d_{i_1}, \ldots, d_{i_{b_t}}, c \rangle \in P(t,z,m) \). In this case \( \langle d_{i_1}, \ldots, d_{i_{b_t}}, 0 \rangle \) is added to \( P(t,z) \).

**case 1** There is a \( k \neq m \) such that \( s_Z(k) \) is a test statement of the form
\[ p_t(y_1, \ldots, y_{b_t})^{1,r} \quad 1 \neq r \]
and \[ \text{val}_Z(y_1, k-1) = d_{i_1} \]
\[ \vdots \]
\[ \text{val}_Z(y_{b_t}, k-1) = d_{i_{b_t}} \]
and \[ \langle d_{i_1}, \ldots, d_{i_{b_t}}, 0 \rangle \in P(t, z, k-1) \]
and \[ s_z(k+1) \] is the instruction with address \( r \).
But \[ \langle d_{i_1}, \ldots, d_{i_{b_t}}, 0 \rangle \in P(t, z, k-1) \] if and only if \( \exists j < k \) such that \[ s_z(j) \] is a test statement of the form:
\[ p_t(u_1, \ldots, u_{b_t})^{l', r'} \quad l' \neq r' \]
where \[ \text{val}_Z(u_1, j-1) = d_{i_1} \]
\[ \vdots \]
\[ \text{val}_Z(u_{b_t}, j-1) = d_{i_{b_t}} \]
and \( s_z(j-1) \) is the instruction with address \( l' \).
But if this is the case, then path \( s_z \) is inconsistent, contradicting our hypothesis that \( s_z \) is a consistent path. A similar situation occurs if
\[ \langle d_{i_1}, \ldots, d_{i_{b_t}}, 1 \rangle \in P(t, z, k-1) \] and \[ s_z(k-1) \] is the instruction with address \( l \). As mentioned earlier, we need only consider other test statements within path \( s_z \), for our procedure uses distinct elements of the domain for each path, and our schema language does not include constants. Thus it is only at stage \( r \) that the predicate \( p_t \) may be applied to arguments with the required values.

case 2 \[ \langle d_{i_1}, \ldots, d_{i_{b_t}}, 1 \rangle \in P(t, z, m) \]
and \[ \langle d_{i_1}, \ldots, d_{i_{b_t}}, 0 \rangle \notin P(t, z, m) \]
Then by our construction \[ \langle d_{i_1}, \ldots, d_{i_{b_t}}, 0 \rangle \notin P(t, z) \].
Thus, since these two cases represent the only two methods by which a \((b_t+1)\)-tuple may be added to a \(P(t, j)\), we have a contradiction to our hypothesis and thus for every \(t \in \{1, \ldots, v\}\), \(\mathcal{I}P_t\) is single-valued.

Our next task is to show that for each \(t\), \(\mathcal{I}P_t\) is total. The argument is essentially the same as the one to show that each \(\mathcal{I}f_t\) is total. Every element \(d_x\) of the domain is in some \(L(j)\). So for every \(b_t\)-tuple \(<d_{i_1}, \ldots, d_{i_{b_t}}\>\) there is an \(L(k)\) such that \(d_{i_q}\), \(q = 1, \ldots, b_t\) is in \(L(k)\). Then by our construction, if there is no \(c \in \{0, 1\}\) such that \(<d_{i_1}, \ldots, d_{i_{b_t}} c> \in P(k, t, m)\) where \(m\) is the length of path \(s_\kappa\), we add \(<d_{i_1}, \ldots, d_{i_{b_t}} 0>\) to \(P(k, t)\) and hence ultimately to \(\mathcal{I}P_t\). Thus for each \(t\), \(\mathcal{I}P_t\) is total.

We have now verified that \(\mathcal{I}\) is indeed an interpretation. For every \(s_j \in \mathcal{C}_P\), there is a \(\overline{d} \in D^n\) such that \(s_j\) is the initial segment of \(\sigma(P, \mathcal{I}, \overline{d})\). This is true because \(\text{val}_{\mathcal{I}}(u, k)\) is exactly the value which the variable \(u\) is assigned under \(\mathcal{I}\) after the execution of instruction \(s_j(k)\) and \(\mathcal{I}\) has been constructed so that each successive path of \(\mathcal{C}_P\) is considered and simulated by a piece of \(\mathcal{I}.\)

We shall call the function interpretation constructed as in theorem 6.2 the uniform interpretation of a schema \(P\).
COROLLARY 6.3 a) If \( P \) is a reachable schema, then \( P \) is uniformly reachable.
b) If \( P \) is a semifree schema, then \( P \) is uniformly semifree.
c) If \( P \) is a free schema, then \( P \) is uniformly free.

Proof a) By theorem 6.2, it follows that exactly the same set of finite paths through \( P \) are followed under the set of all pointwise interpretations as are executed under the uniform function interpretation \( \sim I \). Thus it follows that exactly the same set of instructions are executed. If \( P \) is reachable, then every instruction is executed under some pointwise interpretation, and hence every instruction is executed under the uniform function interpretation \( \sim I \) for some input \( d \). Thus \( P \) is uniformly reachable.

b) Since the uniform function interpretation \( \sim I \) traverses exactly the same set of paths through \( P \) as are traversed under the set of all pointwise interpretations, it follows that if \( P \) is semifree, and hence each path segment of length one is traversed under some pointwise interpretation, then each path segment of length one is traversed under \( \sim I \) for some input \( d \). Therefore \( P \) is uniformly semifree.

c) If \( P \) is free, then the set \( C_P \) of theorem 6.2 is the set of all finite paths through \( P \) from the start statement. Therefore by theorem 6.2, \( P \) is uniformly...
The results of corollaries 6.1 and 6.3 allow us to use the properties of reachability and uniform reachability interchangeably. Similarly for the properties of semifreedom and uniform semifreedom, and freedom and uniform freedom. This will be particularly useful in the next chapter when we consider the inheritance of schema properties by programs, and conversely the inheritance of program properties by schemas.

If we consider program schemas with constants, the situation is markedly different. The next theorem shows that theorem 6.2 does not hold if we modify our model slightly to allow constants. Before we state and prove this result, however, we must extend our definition of an interpretation to include constants. Thus an interpretation \( I \), either pointwise or function, contains an assignment to each constant symbol \( c \), of an element \( I(c) \in \mathcal{D} \).

**THEOREM 6.4** There is a program schema \( P \) with constants such that there are pointwise interpretations \( I_1 \) and \( I_2 \) with \( \sigma(P, I_1) = s_1 \) and \( \sigma(P, I_2) = s_2 \), and there is no function interpretation \( \mathcal{J} \) with domain \( \mathcal{D} \) such that for some \( d_1, d_2 \in \mathcal{D}^n \), \( \sigma(P, \mathcal{J}, d_1) = s_1 \) and \( \sigma(P, \mathcal{J}, d_2) = s_2 \).
Proof: Let c be a constant symbol. Consider the schema:

0. START
1. p(c) 2,4
2. y ← f(x)
3. HALT
4. y ← g(x)
5. HALT

We have written the schema linearly to facilitate our description of execution sequences. Clearly s = 0 1 2 3 and s = 0 1 4 5 are execution sequences. We simply define I_4(c) = d_4 and I_1(p(d_4)) = 0, I_2(c) = d_2 and I_2(p(d_2)) = 1. However, there can be no function interpretation → for which →(c) = d and →(p(d)) = 0 and →(p(d)) = 1. □

COROLLARY 6.5  a) There is a schema with constants which is reachable but not uniformly reachable.

b) There is a schema with constants which is semifree but not uniformly semifree.

c) There is a schema with constants which is free but not uniformly free. □
CHAPTER 7

RESTRICTED CLASSES OF PROGRAMS

7.1 Introduction

We have considered several classes of schemas with semantic restrictions, and have discussed briefly the implications of our results on the areas of program optimization and verification. In this chapter we introduce classes of programs with semantic restrictions, each such program class being the analogue of one of the schema classes we have considered. We shall see that many of the properties of the schema classes also hold for the program classes, thus making the implications for software engineering that much more relevant.

A program \((P, S)\) is free if for every finite path \(s\) there is an input \(d \in D^n\), such that \(s\) is an initial segment of \(\sigma(P, S, d)\).

A program \((P, S)\) is reachable if for every statement \(s_k\), there is an input \(d \in D^n\) such that \(s_k\) is executed.

A program \((P, S)\) is semifree if for each path segment \(\tau\) of length one in \((P, S)\), there is an input \(d \in D^n\) such that \(\tau\) is traversed.

We also define strong equivalence for programs. If \((P, S)\) and \((Q, S)\) are programs with the same domain \(D\), then \((P, S)\) is strongly equivalent to \((Q, S)\), denoted
\[(P, \mathcal{J}) = (Q, \mathcal{J}), \text{ if and only if for every input vector } d \in D^n,\]
\[\text{val}(P, \mathcal{J}, d) = \text{val}(Q, \mathcal{J}, d) \text{ whenever either value is defined.}\]

7.2 Reachable Programs

We begin our investigation with reachable programs.
We have already stated that we consider reachability a minimal requirement for a program to be considered acceptable. We have seen that when looking at abstract programs, most decidability questions are unsolvable. We now discuss the effect on the solvability of such problems of having a specified interpretation. We shall only state a few results explicitly, particularly those for which the result for programs is different than the analogous one for schemas. Most other results hold for programs with a proof similar to the one for schemas.

**Proposition 7.1** Every program is strongly equivalent to some reachable program. $\Box$

As in the case for schemas, however, we shall see that the strongly equivalent reachable version of the program, is not effectively constructable.

**Lemma 7.2** It is decidable whether a reachable program halts for some input.
Proof: The program halts for some input if and only if it contains a halt statement. □

LEMMA H (Minsky[8], Turing[12]): It is not decidable whether an arbitrary program halts for some input. □

THEOREM 7.3 There is no algorithm which given an arbitrary program, constructs a strongly equivalent reachable program.

Proof: Such an algorithm would yield an algorithm to decide whether an arbitrary program halts for some input, contradicting lemma H. □

THEOREM 7.4 It is not decidable whether an arbitrary program is reachable.

Proof: Let \{<u_i, v_i>|i=1,...,n\} be an arbitrary nontrivial Post correspondence problem, where a PCP is trivial if it is the case that for every pair <u_i, v_i>, u_i = v_i. We will construct a program which is reachable if and only if the PCP has a solution. The program is shown in figure 31.

Let \(y_1\) be a positive integer. \(y_1\) and \(y_2\) will be used to accumulate the words formed by concatenation (denoted \\^{**}\\) from the pairs of words \(<u_i, v_i>\). "Rem" denotes the remainder of a division operation, while
"\lfloor \rfloor" denotes the integer quotient of a division operation. If we wish to follow the path corresponding to the application of pairs \(i_1, i_2, \ldots, i_m\) in that order, we need input the number
\[((n+1)\times\cdots\times((n+1)\times i_m+i_{m-1})+i_{m-2})\ldots+i_2)+i_1.\]
This program is not reachable if and only if the nontrivial PCP has no solution. In particular the statement which outputs \(y_i\) can be reached if and only if there is a solution to the Post correspondence problem for \(\langle u_i, v_i \rangle \mid i=1, \ldots, n\). \(\square\)

We saw in Chapter 2 that it is partially decidable, but not decidable whether a reachable schema halts under every pointwise interpretation. Our next theorem shows that the analogue for reachable programs is not even partially decidable.

**THEOREM 7.5** It is not partially decidable whether a reachable program halts for every input in the domain.

**Proof:** Let \(\langle u_i, v_i \rangle \mid i=1, \ldots, n\) be an arbitrary nontrivial Post correspondence problem. We will construct a reachable program which halts for every input in the domain, if and only if the given Post correspondence problem has no solution. We shall use the same notation as in theorem 7.4, and the input value for \(y_i\) is determined in the same manner. The input value for
START

$Y_1 \leftarrow \lambda$
$Y_2 \leftarrow \lambda$

$Y_3 \leftarrow X$

$Y_4 = 0$

$Y_1 = Y_2$

HALT

$Y_5 = \text{Rem}(Y_3)$

$X = \lceil \frac{Y_4}{n+1} \rceil$

$Y_5 = n+1$

$Y_3 = 0$

$Y_3 = 1$

$\cdots$

$Y_3 = n-2$

$Y_1 = Y_1 \times U_1$

$Y_1 = Y_1 \times U_2$

$Y_1 = Y_1 \times U_{n-1}$

$Y_1 = Y_1 \times U_n$

$Y_2 = Y_2 \times V_1$

$Y_2 = Y_2 \times V_2$

$Y_2 = Y_2 \times V_{n-1}$

$Y_2 = Y_2 \times V_n$

FIGURE 32
$y_4$ is restricted to the positive integers.
Consider the program of figure 32.

This program is reachable. It halts for every input value in the domain if and only if the given Post correspondence problem has no solution. □

**COROLLARY 7.6** It is not decidable whether two reachable programs are strongly equivalent.

**Proof:** Let $(P, \mathcal{J})$ be a reachable program with output variables $z_1, \ldots, z_n$, and input variable $x$.

Let $(Q, \mathcal{J})$ be the reachable program of figure 33.

![Flowchart](image)

*(Q, \mathcal{J})* clearly halts for every input value.

We construct the program $(R, \mathcal{J})$ by replacing each halt statement in $(P, \mathcal{J})$ by the sequence of instructions shown
Since \((P, \exists)\) is a reachable program, it follows that
\((R, \exists)\) so constructed is also reachable. But
\((Q, \exists) \simeq (R, \exists)\) iff \((R, \exists)\) halts on every input
iff \((P, \exists)\) halts on every input.
Thus if we could decide strong equivalence for reachable
programs, we could decide whether a reachable program
halts on every input. \(\square\)

An interesting question is under what circumstances
are properties of programs inherited by schemas and
conversely, in which cases are properties of schemas
inherited by a program which is an instance of the
schema. The next theorems say that reachability is a
property which is inherited by schemas from programs, but
that the converse does not hold.
THEOREM 7.7 Let \((P, \mathcal{J})\) be a reachable program. Then \(P\) is a reachable schema.

**Proof:** If \((P, \mathcal{J})\) is a reachable program, then \(P\) is uniformly reachable for the function interpretation \(\mathcal{J}\). Thus by corollary 6.3, \(P\) is reachable. \(\Box\)

THEOREM 7.8 Let \(P\) be a reachable schema. Then there is a function interpretation \(\mathcal{J}\) such that \((P, \mathcal{J})\) is a reachable program.

**Proof:** If \(P\) is a reachable schema, then by corollary 6.3 \(P\) is uniformly reachable. Thus there is a function interpretation \(\mathcal{J}\) such that \((P, \mathcal{J})\) is a reachable program. \(\Box\)

THEOREM 7.9 There is a reachable schema \(P\) and a function interpretation \(\mathcal{J}\), such that \((P, \mathcal{J})\) is not a reachable program.

**Proof:** Consider the schema of figure 35,
and function interpretation $\mathcal{J}$:

$\text{domain} : N$

$(\mathcal{L}f)(u) : u+1$

$(\mathcal{L}g)(u) : u-1$

$(\mathcal{L}p)(u) : u=0$

$P$ is a reachable schema, but $(P,\mathcal{L})$ is not a reachable program. When control is at the test statement, $y \geq 1$ and hence the false branch of the test will always be taken and thus the instruction labelled $a$ is not reachable.

7.3 Semifree Programs

We now focus our attention on semifree programs. We shall state several results for such programs. In most cases the arguments used to prove the result are essentially the same as the ones used for the analogous
result about reachable programs.

**PROPOSITION 7.10** Every program is functionally similar, and hence strongly equivalent, to some semifree program. □

**THEOREM 7.11** There is no algorithm which given an arbitrary program, constructs a strongly equivalent semifree program. □

**THEOREM 7.12** It is not decidable whether an arbitrary program is semifree.

**Proof:** This follows directly from the proof of theorem 7.4 and the observation that the program in that proof is semifree if and only if the PCP has a solution. □

**THEOREM 7.13** It is not partially decidable whether a semifree program halts for every input in its domain.

**Proof:** Let \( \{<u_i,v_i>|i=1,...,n> \) be an arbitrary nontrivial Post correspondence problem. We construct a semifree program which halts for every input in its domain, if and only if the given PCP has no solution. We again use the notation of theorem 7.4 and the input value for \( x \), a positive integer, is determined in the same manner. Consider the program shown in figure 36. This
program is semifree. We have added a switch variable, SW, to the program of theorem 7.5 in order to guarantee that the program is semifree. If the PCP has a solution, then for the input value which "selects" the solution, the program will eventually enter the loop consisting of the four instructions designated \[\text{(a), (b), (c), (d)}.\] If there is no solution for PCP, then the program halts for every positive integer. 

The next result follows from theorem 7.13 by the argument used to prove corollary 7.6 and the observation that program \((Q, \mathcal{L})\) of that proof is semifree.

**Corollary 7.14** It is not decidable whether two semifree programs are strongly equivalent. 

**Theorem 7.15** Let \((P, \mathcal{L})\) be a semifree program. Then \(P\) is a semifree schema.

**Theorem 7.16** Let \(P\) be a semifree schema. Then there is a function interpretation \(\mathcal{I}\) such that \((P, \mathcal{I})\) is a semifree program.

**Theorem 7.17** There is a semifree schema \(P\) and a function interpretation \(\mathcal{I}\), such that \((P, \mathcal{I})\) is not a semifree program.

**Proof:** The schema \(P\) used in the proof of theorem 7.9 is
semifree, but program \((P, \emptyset)\) is not reachable, and hence not semifree. \(
\)

### 7.4 Free Programs

We have seen that the classes of reachable programs and semifree programs are very similar in terms of their decidability properties and their relationship to the respective classes of schemas.

We shall now examine a few questions for free programs, paying particular attention to the relationship between this class of programs and the class of free schemas. First, however, we note that membership in this class of programs, like the other two classes discussed, is not decidable.

**THEOREM 7.18** It is not partially decidable whether a program is free.

**Proof:** Let \(\{<u_i, v_i>|i=1, \ldots, n\}\) be an arbitrary nontrivial Post correspondence problem. We shall construct a program which is free if and only if the Post correspondence problem has no solution. Again, we shall use the same notation as in theorem 7.4 and the input value for \(v_x\) is again restricted to the non-negative integers and is determined in the same manner as in the proof of that theorem. Consider the program shown in
START

\[ y_1 \leftarrow \lambda \]
\[ y_2 \leftarrow \lambda \]

\[ y_4 \leftarrow x_1 \]
\[ sw \leftarrow x_2 \]

\[ y_4 = 0 \]
\[ \text{yes} \]
\[ y_1 + y_2 \]
\[ d_{sw} = 1 \]
\[ \text{yes} \]
\[ z \leftarrow y_1 \]

\[ y_6 \leftarrow \text{Rem}(y_4) \]
\[ \text{yes} \]
\[ z \leftarrow y_2 \]
\[ \text{HALT} \]

\[ y_4 \leftarrow \left\lfloor \frac{y_4}{n+1} \right\rfloor \]

\[ y_3 = 0 \]
\[ \text{no} \]
\[ y_3 = 1 \]
\[ \text{no} \]

\[ y_1 \leftarrow y_1 \times u_1 \]
\[ y_2 \leftarrow y_2 \times v_1 \]

\[ y_3 = n-2 \]
\[ \text{no} \]

\[ y_1 \leftarrow y_1 \times u_n \]

\[ y_2 \leftarrow y_2 \times v_1 \]

\[ y_3 = n-2 \]
\[ \text{no} \]

\[ y_1 \leftarrow y_1 \times u_n \]

\[ y_2 \leftarrow y_2 \times v_1 \]

\[ z \leftarrow y_2 \times v_n \]

FIGURE 37
figure 37.
If there is no solution to the given Post correspondence problem, then clearly, by the proper setting of SW at input, any finite path through the flow diagram may be followed, and hence the program is free.
If however, there is a solution to the Post correspondence problem, for the input value which calculates the solution, \(y_1 = y_2\) and hence regardless of the value of SW, the "no" branch of the test \((y_1 \neq y_2 \& SW = 1)\) is taken. Thus, the path which computes a solution and then takes the "yes" exit from the test \((y_1 \neq y_2 \& SW = 1)\) can never be followed if there is a solution to the PCP. Therefore, the program is not free. □

Our next result will demonstrate that the analogue of the basic theorem for free schemas (theorem 5) which provides us with a structural characterization of freedom, does not hold for free programs.

**THEOREM 7.19** There is a program \((P, \mathcal{A})\) in which no test is ever applied twice to the same n-tuple of elements of the domain, but \((P, \mathcal{A})\) is not free.

**Proof:** We consider the program shown in figure 38 which obviously does not contain a repeated application of a test, but which is nonetheless not free.
It is however, obviously true that if a program is free, then no test is ever applied twice to the same n-tuple of elements of the domain, within an execution sequence.

We saw earlier in this chapter that reachability and semifreedom are properties which are inherited by schemas from programs, but not conversely. The same situation prevails in the case of freedom.

THEOREM 7.20 Let \((P, \alpha)\) be a free program. Then \(P\) is a free schema. \(\Box\)
THEOREM 7.21 Let $P$ be a free schema. Then there is a function interpretation $\mathcal{I}$ such that $(P, \mathcal{I})$ is a free program. □

THEOREM 7.22 There is a free schema $P$ and a function interpretation $\mathcal{I}$, such that $(P, \mathcal{I})$ is not a free program.

Proof The schema $P$ used in the proof of theorem 7.9 is free, but program $(P, \mathcal{I})$ is not reachable, and hence not free. □

7.5 Programs Which Halt For Every Input

The last class of programs we shall discuss is the class of programs which halt for every input. This is the analogue of schemas which halt under every pointwise interpretation and we shall consider in particular the relationship between these two classes.

First we note that, as was the case for the other three classes of programs considered, membership in this class is not decidable.

LEMMA I (Minsky [8]) It is not decidable whether a program halts for all inputs. □
Before continuing our discussion of this class of programs, we define an execution sequence tree. We have already encountered a similar notion for schemas, when we constructed $T(P)$ in lemma 4.19.

Let $(P, J)$ be a program. Then the execution sequence tree for $(P, J)$ is a tree whose nodes are labelled as follows:

1. The root is labelled START.
2. If a node $q$ is labelled with an execution sequence $s=(k_1, \ldots, k_n)$ then
   a. If $k_n$ is a START, assignment, input, output, or unconditional branch statement, then $q$ has only one son labelled with the execution sequence $s'=(k_1, \ldots, k_n, k_{n+1})$ where $k_{n+1}$ is the unique successor of $k_n$.
   b. If $k_n$ is a test statement, then $q$ has either one or two sons. If $k'_{n+1}$ and $k''_{n+1}$ are the 0- and 1-successors of $k_n$, then $q$ will have two successors, labelled $s'=(k_1, \ldots, k_n, k'_{n+1})$ and $s''=(k_1, \ldots, k_n, k''_{n+1})$ if both $s'$ and $s''$ are consistent paths. If only one of the two possible paths is consistent, then $q$ has only one son, labelled appropriately.
   c. If $k_n$ is a halt statement, then $s$ is a complete execution sequence and hence $q$ has no sons.
An execution sequence tree is defined analogously for a program schema P. In this case the nodes are labelled with all possible execution sequences corresponding to a free interpretation I.

Paterson has proven the following theorem using a construction similar to the one we outlined in our proof of lemma 4.19.

**THEOREM J** (Paterson [9]) The execution sequence tree for a schema P is finite if and only if P halts under every pointwise interpretation.

Again, it is obviously true that if a program \((P, \mathcal{A})\) has a finite execution sequence tree, it halts for every input in its domain. We shall now demonstrate that the converse of this statement is not true.

**THEOREM 7.23** There is a program \((P, \mathcal{A})\) which halts for every input in its domain, but the execution sequence tree for \((P, \mathcal{A})\) is not finite.

**Proof:** We consider the program over the integers, shown in figure 39:
If $x \leq 0$ then the body of the loop is never executed and hence $(P, \emptyset)$ halts immediately. If $x = n \geq 1$, then the loop body is executed exactly $n$ times, after which $y = 0$ and the loop is exited and the program halts. Thus $(P, \emptyset)$ halts for every input in the domain. We next consider the execution sequence tree for $(P, \emptyset)$. We have labelled the instructions in our program and modify our notation slightly as follows: a node in the tree will be labelled with a sequence consisting of the names of the instructions, rather than the instructions themselves.
This execution sequence tree for the program of Figure 39 is not finite.

We have seen in this chapter, that reachability, semifreedom and freedom are all properties which are inherited by schemas from programs, but not conversely. In the case of always halting programs and schemas, the situation is reversed.

**Theorem 7.24** There is a program \((P, \mathcal{I})\) which halts for every input, but \(P\) diverges under some pointwise interpretation.

**Proof:** Consider the schema shown in figure 40, and function interpretation \(\mathcal{I}\):
**FIGURE 40**

```
START
\[
y \leftarrow x
\]

\[
p(y)
\]

\[
\begin{align*}
T & \quad y \leftarrow f(y) \\
F & \quad y
\end{align*}
\]

\[
z \leftarrow x
\]

HALT
```

**FIGURE 41**

```
START
\[
y \leftarrow x
\]

\[
y > 0
\]

\[
\begin{align*}
T & \quad y \leftarrow y - 1 \\
F & \quad z \leftarrow x
\end{align*}
\]

HALT
```
domain: integers

\((-\emptyset f)(u): u-1\)

\((-\emptyset p)(u): u>0\)

Then \((P, \emptyset)\) is the program shown in figure 41. If \(d=0\) where \(d\) is the input value, then \((P, \emptyset)\) halts without ever executing the loop body, and if \(d=n>0\) then the loop body is executed \(n\) times before the loop is exited, and then the program halts. Thus \((P, \emptyset)\) halts for every integer input.

But \(P\) does not halt under every pointwise interpretation.

In particular, if \((Ip)(u)\) is false for every \(u\) in the domain, then \(P\) does not halt under pointwise interpretation \(I.\)

Before stating our final result we introduce one additional term. We shall say a schema \(P\) halts under every function interpretation if for every function interpretation \(\emptyset\) and input vector \(d\) in the universe, \(\sigma(P, \emptyset, d)\) is finite.

**Theorem 7.25** If a schema \(P\) halts under every pointwise interpretation, then any program \((P, \emptyset)\) which is an interpretation of \(P\), halts for every input.

**Proof:** If \(P\) halts under every pointwise interpretation, then it follows from theorem 6.4 that \(P\) halts under every function interpretation. Thus, for every \(\emptyset\), \((P, \emptyset)\) halts for every input. \(\square\)
CHAPTER 8

CONCLUDING REMARKS

In Chapters 2 and 3 we introduced two new classes of program schemas—the reachable and the semifree schemas. They were intended to model programs which have nice properties, and to represent the type of simple structural properties we would hope a compiler could identify. If these properties were lacking, we had hoped that a compiler could recognize that fact and translate the schema into an equivalent improved version. We demonstrated that this is not possible and that most of the commonly considered decision problems for these classes are not solvable.

In Chapter 4 we considered the relationships between several classes of schemas. Particular attention was paid to the question of when knowledge that a schema has certain desirable properties allows one to decide whether or not it has other desirable properties. We also studied the question of under what circumstances we could verify whether or not two schemas in different classes are strongly equivalent. We obtained the most encouraging results of this nature when we considered the liberal schemas in relationship to other classes. We found, for example, that reachability and semifreedom are decidable for liberal schemas and that every liberal
schema can be effectively translated into a liberal semifree schema.

These results suggest several problems which warrant future investigation. One possible direction to consider is whether we can find other classes of schemas for which reachability and semifreedom are decidable. It would be particularly nice if, as is the case for liberality, we could decide whether or not an arbitrary schema had the required properties. Certainly we would like such a class to be quite broad in representational power. Ideally, the class of all schemas would be translatable into the newly defined classes.

Another interesting question is whether we can characterize the type of information necessary to decide various properties. We saw, for example, that knowing that a schema is reachable is not sufficient to allow one to decide whether it is semifree or free, whereas if a schema is liberal, then reachability, semifreedom, and freedom are all decidable properties. We would like not only to be able to specify such particular instances, but also to be able to specify that if a schema has any property with certain characteristics, then other specific properties are decidable.

We would also like to investigate whether the structural requirements of WHILE schemas [1] are sufficient to allow us to decide properties which are not decidable for the general, unrestricted case. Our
interest in this question was part of our motivation for considering the question of the preservation of properties by various types of "sameness" relations. This was done in Chapter 5. We recognized that since the class of all schemas is effectively translatable into the class of WHILE schemas, any property which is preserved by strong equivalence could not be decidable for WHILE schemas. Thus we wished to see which properties were at least reasonable candidates for consideration.

The other reason for our interest in the results of Chapter 5 is that they allow a distinction to be made between properties which are inherent in the class of functions represented by a schema, regardless of the algorithm used to compute it, and those properties which are dependent upon the algorithm or its encoding. We saw, for example, that halting is a property which is totally independent of the algorithm used to do a computation. Reachability, on the other hand, is a property which is entirely dependent on the algorithm used to do the computation and the form in which the algorithm is written. Thus we can have two schemas which are isomorphic, such that one is reachable and the other is not.

In Chapter 6 we studied the relationship between pointwise and function interpretations. We discovered that certain classes of schemas which were intuitively different in computational power, were in fact
equivalent. That is, one would expect that any uniformly reachable schema is reachable. We had not expected the converse to hold. As a corollary to the theorem which relates function and pointwise interpretations, however, we discovered that in fact these two classes are equivalent. We also observed that when our model was modified slightly to include constants, the situation altered radically and the classes were no longer equivalent. It was surprising that such a minor modification would cause such dramatic changes. This was also surprising in view of the fact that this modification of the Luckham, Park, and Paterson model was made by several people ([2], [5], [7] for example) who nonetheless built upon the Luckham, Park, and Paterson results.

In Chapter 7 we investigated programs. We defined classes of programs which are analogous to our schema classes, and asked translatability and decidability questions similar to the ones asked for schemas. We saw directly the impossibility of doing the simple types of code improvements which one would hope could be done algorithmically by compilers. This is disappointing both in view of the simplicity and structural nature of the properties considered. It suggests strongly the likelihood that less structural properties will be undecidable and encourages us to look for subsets of programs for which such improvements can be automatically
performed. Alternatively, we could consider heuristic code improvement. An interesting related question is whether we can define classes of program schemas based on the properties currently used in optimizing compilers, and investigate their decidability and translatability properties. This may give us insight into some problems commonly encountered when using optimizers.

In Chapter 7 we also studied the inheritance of program properties by schemas and conversely, the inheritance of schema properties by programs. The consideration of these problems was facilitated by our results of Chapter 6 relating pointwise and function interpretations. These results provide insight into the general applicability to programs of results about schemas.

We feel that the ideas presented here have laid a theoretical foundation for the consideration of properties of programs. This should be useful in obtaining candidates for normal form classes of programs. It should also prove helpful in the design of automatic code improvers.
REFERENCES


