LOOP SCHEMATA

by

Robert L Constable
Cornell University
Ithaca, New York

Tech Report 71-94

Department of Computer Science
Cornell University
Ithaca, New York 14850
Loop Schemata

Robert L Constable
Cornell University
Ithaca, New York

ABSTRACT

We define a class of program schemata arising from the subrecursive programming language Loop. In this preliminary report on Loop schemata we show how to assign functional expressions to these schemata (as one aspect of the problem of assigning meaning to these programs), and we outline a solution to the schemata equivalence problem. Schemata equivalence is reduced to questions about formal expressions. Certain subcases of the problem are easily shown solvable, and although we claim that the general problem is solvable, we do not present the complete solution here because of its complexity.

KEY WORDS & PHRASES
program, program schemata, Algol, universal programming language, Loop language, subrecursive language, computable function, primitive recursive function, functional, general recursive functional, equivalence problem, unsolvability.

C.R. CATEGORIES
4.20, 4.22, 5.22, 5.24

1This work was supported in part by NSF Grant GJ-579.
§ Introduction

Consider the simple programming language having only assignment statements, "\(v \leftarrow f(w)\)" and conditional statements, "if \(T(v)\) then go to \(\ell\)" where \(v, w\) are variables, \(\ell\) is a label, \(f\) is a function symbol and \(T\) is a predicate symbol. By giving \(f\) and \(T\) specific values, say \(f_1(v) = v + 1\), \(f_2(v) = v^2 - 1\) and \(T(v)\) iff \(v \neq 0\), a specific language base results (called \(G_3\)). The programming language produced from this base (also called \(G_3\)) is defined to be the set of all finite sequences of uniquely labelled statements.

If instead of specifying \(f_1, f_2, T\) we left them as variables, then the finite sequences of uniquely labelled instructions are called program schemata\(^2\). Vaguely speaking, such schemata can be defined for any language with a base consisting of assignments and conditionals using simple variables, \(n\)-ary functions and \(n\)-ary predicates. (We shall be precise about scheme in §2.

One program scheme represents a whole class of programs, those obtained by supplying values for the function and predicate variables. These programs share the same "control structure". The class of all program schemata can be said to characterize the control structure of the language. We might say that this control structure defines the type of the language. So for instance, the type of \(G_3\) is given by the base: \(v \leftarrow f_1(v), v \leftarrow f_2(v), \) if \(T_1(v)\) then \(\ell\).

We say that two program schemata, say \(S_1, S_2\), are (strongly) equivalent iff they produce the same output for all values of their function, predicate and number variables. The equivalence

\(^2\)The terms "schema" or "scheme" both seem appropriate for the singular with the corresponding plurals "schemata" and "schemes", (but not schemas). We shall use both for the sake of variety and indecisiveness, but schemata will often connote a collective use of the term.
of two program schemata therefore implies the equivalence of program having these schemata but not conversely. So even if equivalence of programs is undecidable for a language, equivalence of schemata might be decidable.

If there were a decision procedure for schemata equivalence, it may have practical as well as theoretical interest. However, if the programming language is of sufficiently general type, then Paterson [6] (or [7]) has shown that schemata equivalence is unsolvable (but of degree \( \Sigma^0_1 \) where program equivalence is of degree \( \Pi^0_2 \)).

In the face of this unsolvability, it is natural to seek subcases which might be solvable. Paterson [6] discusses several interesting cases (including the pioneering work by Ianov [2]), but as he and Milner [4] point out there is a need to see more examples before interesting general principles emerge.

One natural place to look for interesting schemata is the realm of subrecursive programming languages (see [1] and [3]). These provide an example of a new type of programming language; one more restrictive than the universal type for which Paterson's results hold. The restrictions suggest that schemata equivalence is solvable.

We present in §2 one class of subrecursive program schemata, namely Loop schemata. Their syntax is given first and then they are interpreted as general recursive functionals. In §3 we provide a function theoretic notation for Loop schemata and an interesting sufficient condition for schemata equivalence. This condition has some nominal bearing on questions about code optimization and parallelism.

In §4 we extract a critical feature of the notation in §3 and use it to define calling sequences and calling expressions for Loop schemata. The schemata equivalence problem is formulated in terms of these expressions. We examine some easily solved but interesting subcases
of this problem and illustrate some deeper forms of equivalence than those detectable by the methods of §3.

We leave the solution of the necessary questions about calling expressions, hence the complete solution of Loop equivalence problem, to a complete report because at present it requires a considerable technical effort. The report concludes with a few untested open problems.
§2 Basic Definitions

(2.1) Loop schemata syntax

We describe first the class of Loop schemata syntactically.

<variable>:: = V|V<variable>

<function variable>:: = F|F<function variable>

<term>:: = <function variable>(<variable>)

<assignment scheme>:: = <variable>+</term>

<iterative scheme>:: = DO <variable>;<program>;END

<program scheme>:: = <statement scheme>|<statement scheme>;<program
scheme>

As abbreviations we use $V^n$ for $V \ldots V$ n-times and $F^n$ for $F \ldots F$ n-times. We also use $u_i$, $v_i$, $w_i$, to denote individual variables and $h_i$ to denote function variables. Also $S_i$ denote program schemata. With these abbreviations, the two statement schemes are

$v \gets h(w)$ and

DO v; S; END.

The program schemata defined above are called Loop schemata. The collection of them is denoted $\mathcal{L}$.

Example 2.1 The following are examples of Loop schemes

\[
\begin{align*}
V & \gets F(V) \\
V_1 & \gets F(V_1) \\
DO V_1 & \\
V_2 & \gets F(V_2) \\
V_2 & \gets F(V_1) \\
V_1 & \gets F(V_2) \\
V_1 & \gets F(V_1) \\
END & \\
DO V_1 & \\
V_2 & \gets F(V_2) \\
V_2 & \gets F(V_1) \\
V_1 & \gets F(V_2) \\
END & \\
V & \gets F(V) \\
V_2 & \gets F(V_2) \\
END & \\
\end{align*}
\]
A specific programming language of the Loop-type is obtained by replacing function variables with function constants. The Loop language of Meyer & Ritchie [3] is obtained by fixing three function constants.

(2.2) Loop language

Let +1, 0, and ( ) be names for the following functions from integers, \( \{0,1,2,\ldots\} \), to integers.

(i) \( f(x) = x + 1 \)  
(ii) \( f(x) = 0 \) for all \( x \)  
(iii) \( f(x) = x \) for all \( x \)

successor  
zero  
identity

The Loop language usually comes with the added restriction that +1 can be used only in schemes of the form \( v+ h(v) \).

Notation: We shall denote Loop programs by \( \alpha_i \), the set of them by \( L \), the integers \( \{0,1,2,\ldots\} \) by \( \mathbb{N} \), variables over \( \mathbb{N} \) by \( x_i \), over the cartesian product \( \mathbb{N} \times \ldots \times \mathbb{N} = \mathbb{N}^n \) by \( x_i \), functions from integers to integers by \( f( ) \) or \( f( ) : \mathbb{N}^n \to \mathbb{N} \), vector valued functions by \( \langle f() \rangle \) or \( f( ) : \mathbb{N}^n \to \mathbb{N}^p \). The set of functions \( \mathbb{N}^n \to \mathbb{N}^p \) is denoted \( \mathcal{J}_{n,p} \). The subset of (general) recursive functions is denoted \( \mathcal{R}_{n,p} \).

The intended use of programs in theoretical investigations is the computation of functions from \( \mathbb{N}^n \) to \( \mathbb{N}^p \). The input/output (I/O) conditions on programs and the precise conventions which associate a function to a program vary slightly from author to author.

We shall use the convention that all variables \( w \) which appear first (in the order of statements in a program) in the form \( v+h(w) \) or \( v+h(w) \) are called right variables or input variables. All variables \( v \) which appear on the left hand side (lhs), as in \( v+h(w) \), are output variables. Usually the output variables are thought of as some subcollection of these, designated by a statement like "OUT \( v_1, \ldots, v_p \)". This convention would also be acceptable here.
It is also customary to define input variables as a subset of the right variables, using a statement like "\(\text{IN } v_1, \ldots, v_n\)". The difficulty with this approach is that the remaining right variables are undefined unless some convention is made about initial condition of such variables. If we assume they were all 0, then this alternate input convention would also be acceptable here.

Notice that the Loop schemata, hence the Loop type languages, do not have labels, conditional statements or predicates. There is no mechanism for testing or branching. We now discuss the details of the interpretation, or semantics, of Loop programs and program schemata.

(2.3) **semantics**

The mathematical interpretation of a loop scheme \(S_i\) having function variables \(h_1, \ldots, h_e\), input variables \(v_1, \ldots, v_n\) and output variables \(w_1, \ldots, w_p\) is a functional

\[ S_i : \mathcal{F}_i \times \mathbb{N}^n \rightarrow \mathbb{N}^p \]

The value is denoted \(S_i[f_1(), \ldots, f_e(), x_1, \ldots, x_n]_j\) \(j=1, \ldots, p\)

where \(f_i() \in \mathcal{F}_i\), \(x_i \in \mathbb{N}\), or more simply, \(S_i[<f()>, X]\).

A functional is simply a function which can take both function and number inputs and having number values. The class of all functionals (over \(\mathcal{F}, \mathbb{N}\)) is denoted \(\mathcal{F}\). The class of Loop schemata, \(\mathcal{L}\), defines a subclass, \(\mathcal{L}_L\) of the general recursive functionals (denoted \(\mathcal{R}\)). (See Roger's [8] for a discussion of functionals.)

(2.4)

To be precise about the interpretation in (2.4) we must see how a Loop program scheme, \(S_i\), can be regarded as a **functional computing program**. First each function variable is regarded as an input variable as are the right individual variables. Under these circum-
stances, the assignment

\[ v + h_i(w) \]

means, put into variable \( v \) the value of \( f_i(\) \) applied to
the contents of variable \( w \).

The iterative,

\[ \text{Do } v; \ S; \ \text{END} \]

is interpreted by the following code

\[
\begin{align*}
\bar{v} + v \\
1 \text{ if } \bar{v} = 0 \text{ then go to } l \\
\alpha \\
\bar{v} + \bar{v} = 1 \\
go \text{ to } l \\
l \\
\end{align*}
\]

where \( \bar{v} \) is a variable not occurring in the program containing this
iterative; \( \bar{v} \) is called the loop-control variable. \(^3\)

A precise formal semantics could be given for these informal
definitions by selecting a formal machine model such as the RASP.
In the context of a RASP-based theory of functionals one could then
prove that each Loop scheme \( S_i \) computes a general recursive functional,
\( S_i[ \ ] \).

Such a precise account would include a definition of a scheme-
computation on inputs \( f_i(\) \), \( x_i \), a terminating computation, and the
notion of the flow of control from statement scheme to statement

\(^3\)It is noteworthy that Loop schemata can not be translated into \( G \)
type schemata (discussed in §1) whereas Loop programs can be trans\(^3\)
lated to \( G_3 \) programs. This is because the semantics for Loop schemata
require the specific function \( v^21 \) and predicate \( v=0 \). Therefore Loop
schemata are not precisely a subset of the \( G_3 \) type schemata. They
are actually quasi-schemata.
scheme. We do not need such a formal account here. It suffices to notice that each Loop scheme computation on inputs from \( \mathcal{A}^t \) and \( \mathbb{N}^n \) halts leaving an output in \( \mathbb{N}^p \). The flow of control is always downward except in loops where it cycles. The reader seeking a precise account of this is referred to Paterson [7].

(2.5) **equivalence of schemes**

Given any two schemes \( S_1, S_2 \), we can assume they have the same number of inputs and outputs by adding dummy variables to the functional description. Thus assume

\[
S_i[\_]: \mathcal{A}^t \times \mathbb{N}^n \rightarrow \mathbb{N}^p \quad \text{for } i=1,2.
\]

We say that the two schemes are strongly equivalent, \( S_1 \equiv S_2 \), iff

\[
\forall f_1 (\_), \ldots, f_t (\_), \forall x_1, \ldots, x_n, \forall j \leq p
\]

\[
S_1[f_1 (\_), \ldots, f_t (\_), x_1, \ldots, x_n] = S_2[f_1 (\_), \ldots, f_t (\_), x_1, \ldots, x_n].
\]

Thus two schemes are equivalent iff they compute the same functional. The following example gives two equivalent schemes.

**Example 2.2**

\[
S_1
\]

\[
V_1 + f_1 (V_1) \\
\text{DO } V_2 \\
V_3 + f_1 (V_2) \\
\text{END}
\]

\[
S_2
\]

\[
\text{DO } V_2 \\
V_3 + f_2 (V_3) \\
V_3 + f_2 (V_3) \\
\text{END}
\]

Both schemes compute the same functional

\[
\mathcal{A}^t \times \mathbb{N}^2 \rightarrow \mathbb{N}^3
\]
although $S_2$ appears to have both $f_2(\cdot)$ and $V_4$ as additional inputs. The functional can be described mathematically as

$$S[f_1, V_1, V_2] = \langle f_1(V_1), V_2, f_2^{(V_2)}(V_2) \rangle$$

In the next section we describe this "mathematical description" in detail and see how it can be obtained systematically from the scheme.
§3 Assigning Mathematical Notation to Schemes

(3.1) function names in mathematics

The standard mathematical notation for the Loop program

DO X; X+X+1; END is 2 \cdot x and for DO X; DO X; X+X+1; END; END is

2^x \cdot x. If one is being careful to distinguish function names
(functors) from integer names, then the usual convention has

\lambda x[2 \cdot x] or f_2 ( ) with f_1(x)=2 \cdot x for 2 \cdot x, and \lambda x[2^x \cdot x] or \ f_2 ( )
with f_2(x)=2^x \cdot x for 2^x \cdot x.

If the two Loop programs, \alpha_1 and \alpha_2, compute functions \mathbb{N} \to \mathbb{N},
then the program \alpha_1; \alpha_2 is usually denoted by \overline{\alpha}_1 (\overline{\alpha}_2 ( )) where

\overline{\alpha}_1, \overline{\alpha}_2 are \ the \ functors \ for \ \alpha \ and \ \alpha_2. \ Thus

DO X; X+X+1; END; DO X; DO X; X+X+1; END; END is denoted by

2^{(2 \cdot x)} \cdot (2 \cdot x) or by \ f_2 (f_1 ( )). We see informally that con-
junction of programs corresponds to composition of the corresponding
functions.

The "application of DO v; _____ ; END to a program \alpha" resulting in
DO v; \alpha; END, also has a standard mathematical form. Again suppose
that \alpha( ): \mathbb{N} \to \mathbb{N}. Then for any function f( ): \mathbb{N} \to \mathbb{N}, define
its iteration by

(i) f^{(0)}(x)=x

(ii) f^{(n+1)}(x)=f(f^{(n)}(x))

Now it is easy to see that DO v; \alpha; END is represented by \alpha^v(x) ( )
where x is the value of v.

We shall exploit this correspondence between programs and functors.
To do so we must define it carefully and carry it over to schemes.
The only difficulty is providing a smooth mechanism for the iteration of vector functions.

There is also a small matter of terminology. If programs compute functions which are denoted mathematically by functors, then what do we call the mathematical names for functionals, which program schemes compute? We shall call them functorials and then avoid the term whenever possible. An example of one is $H[f,x,y] = f(x)(y)$, it is the mathematical notation for

$$\text{DO } x; y\leftarrow f(y); \text{ END}$$

Notice that to simplify life, we've allowed the same letters to name both program variables and the values they can assume, numbers; and function variables and the values they can assume, functions. Notice that the $\lambda$-expression for this functorials would be

$$\lambda f\lambda x,y[f(x)(y)].$$

We shall use the first form with the following explicit conventions.

Notation: Capital Latin letters, $F_1, G_1, H_1$ denote functorials and $F_1[ ]$, $G_1[ ]$, $H_1[ ]$ denote the corresponding functionals.

To simplify the connection between schemes, $S_1$, and their functorials, we use number and function variables $(x_1,f_1)$ for variables and function variables $(v_1,h_1)$ in schemes.

(3.2) **Iteration of vector functions**

A key piece of notation needed below is that for the iteration of vector valued functions and functionals. We need iteration only into number arguments, so it suffices to treat the case of functions.  

---

"Notation for the iteration of functions $f(\cdot): \mathbb{N}^n \to \mathbb{N}$ is difficult so one might expect that for $f(\cdot): \mathbb{N}^n \to \mathbb{P}$ it would be terrible. Happily, it is not. It is easier than the single value case and in fact, clarifies that case."
Notation for vectors is critical, so we consider it with care. A vector \( <x_1, \ldots, x_n> \) will be denoted by \( <x> \), or by \( X \) when no confusion results. The number of elements is indicated by writing \( <x> \in \mathbb{N}^n \) or \( X \in \mathbb{N}^n \) in the context. The \( i \)-th component is denoted \( <x>_i \) or \( X_i \), thus \( X_i = x_i \).

A value of a vector valued function is denoted \( <f(X)> \), or \( f(X) \) when no confusion is possible. The function itself is denoted \( <f()> \) or \( f() \). Thus in the worst case when it is important to avoid confusion with number valued functions, \( f_1() \), and vector valued functions, \( f_2() \), we can write \( f_1(X) \) versus \( <f_2(X)> \) or even \( <f_2(<x>)> \). Now for iteration.

(1) Given \( f() : \mathbb{N}^n \rightarrow \mathbb{N}^p \) and \( X \in \mathbb{N}^n \) define \( \hat{f}(()) : \mathbb{N}^{n+1} \rightarrow \mathbb{N}^{n+p} \) as follows

\[
\begin{aligned}
\hat{f}(0)(X) &= \begin{cases} 
X_j & \text{if } j = 1, \ldots, n \\
0 & \text{otherwise (ow)}
\end{cases} \\
\hat{f}(n+1)(X) &= \begin{cases} 
f(\hat{f}(n)(X)_1, \ldots, \hat{f}(n)(X)_n) & \text{if } n < j \leq p \\
g_j(\hat{f}(n)(X)_1, \ldots, \hat{f}(n)(X)_p) & 1 \leq j \leq n
\end{cases}
\end{aligned}
\]

The functions \( g_j() \) are the feedback functions determining which outputs affect the \( j \)-th input.

This class of vector iterations is more than adequate for our purposes. We are concerned only with iterations which occur in the form \( \text{DO } x; S; \text{END} \) for some scheme \( S \). In such a case the feedback is built into the scheme \( S \). Conversely given \( S \), the function corresponding to it has the feedback built in. This allows for a much simpler notation as we shall see below in (2).

(2) Given \( f() : \mathbb{N}^n \rightarrow \mathbb{N}^p \), the required idea of iteration, 

\[ f(x)(()) \]

is

\[
\begin{array}{c}
\text{DO X} \\
f \\
\text{END}
\end{array}
\]

where \( f \) has \( n \) inputs and \( p \) outputs. We notice that unless some output is also an input, the above fixed
expression is just \( f(\cdot) \) itself. In fact, the only outputs of consequence are those which are also inputs (feedback variables). Therefore, we assume that \( p \geq n \) and we regard each input as an output (if it does not appear on the lhs of any assignment, then it is an implicit output). The definition of iteration given under these conditions can be seen to be adequate for the first notion of vector iteration, but we do not need such a result here. The definition follows:

(i) \( f^{(0)}(x)_j = \begin{cases} x_j & \text{if } j \leq n \\ 0 & \text{ow} \end{cases} \)

(ii) \( f^{(n+1)}(x)_j = f(f^{(n)}(x)_1, \ldots, f^{(n)}(x)_n)_j \)

An example showing the connection between these iteration functors and iteratives in schemes should be helpful.

Example 3.1 Consider the following scheme

\[
\begin{align*}
&\text{DO } x_1 \\
&\quad y \leftarrow f_1(x_1) \\
&\quad x_1 \leftarrow f_2(x_2) \\
&\quad x_2 \leftarrow f_3(y) \\
&\text{END}
\end{align*}
\]

The scheme defines the vector function \( S[f_1, f_2, f_3, x_1, x_2] = \langle f_2(x_2), f_3(f_1(x_1)), f_1(x_1) \rangle \). The simple function notation for the function can be

\( S[x_1, x_2] = \langle f_2(x_2), f_3(f_1(x_1)), f_1(x_1) \rangle \).
The iteration of $S$, denoted $S(x)[f_1, f_2, f_3, x_1, x_2]$ or more simply
\[ S(x)[x_1, x_2], \] is defined by
\[ S(0)[x_1, x_2] = \langle x_1, x_2, 0 \rangle \]
\[ S(n+1)[x, x] = S[S(n)[x_1, x_2]]_1, S[S(n)[x_1, x_2]]_2, S[S(n)[x_1, x_2]]_3 > \]

(3.3) assigning functors to schemes

The process of assigning a functional expression to a scheme is simple. The idea is to follow the flow of control backwards from the output variable. A simple informal routine for this is given below.

Given a scheme $S_i = s_1, s_2, \ldots, s_k$, the last occurrence of a variable $x$ in $S$ is its occurrence in $s_m$ where $m = \max\{i \mid x \text{ occurs in } s_i\}$.

We now give an algorithm for translating a scheme to a functorial.

**Routine A**

1. Locate all input variables of $S$, i.e. all right variables. List them as an output vector, $\langle y_1, \ldots, y_p \rangle$.

2. For each $y_i$, find the last occurrence of $y_i$ on the left hand side (lhs) in some statement of $S$. The occurrence is one of two types
   (a) $y_i = f_1(w_i)$

   (b) $y_i$ occurs on the lhs in the scope of an iterative, say is DO $v$; $H$; END.

Follow a separate procedure for each case.
(a) procedure: Write \( f_i(w_i) \) for \( y_i \) in the output vector.

(b) procedure: Let \( s_m \) be the statement of last occurrence. Select an \( H_i \), locate inputs and outputs to \( s_m \), order them and write

\[
H(v)[v_{i_1}, \ldots, v_{i_q}]_k
\]

for \( y_i \) in the output vector (where the subscript on the functorial selects the output \( y_i \) from the vector of outputs for \( s_m \)).

(3) For each input in the output vector resulting from (1), apply the process of step (1). Continue in this way until only input variables, \( x_i \) to \( S_i \) remain as inputs and none of them occur on the lhs of any statements \( s_j \) for \( j < m \) where \( s_m \) contains the last occurrence of \( x_i \). Notice that this process must stop after \( \ell \) steps (\(|S| = |s_1; \ldots; s_\ell| \leq \ell\)).

Routine A produces a number of new functorial letters \( H_i \). Each of them has vector input/output and is associated with an iterative statement scheme of \( S \). Step 2 in the translation from \( S \) to a functorial is the application of Routine A to each iterative statement scheme \( s_i \) and associated functorial \( H_i \). The result of this step is a set of functional expressions and a new set of letters associated with iteratives contained in each iterative of \( S \). Routine A is applied to these and the process is repeated until there are no new functorial letters introduced. This procedure requires only \( d \) iterative where \( d \) is the maximum depth of nesting of iteratives in \( S \).

Step 3 of the translation is the substitution of functional expressions for functorial letters until only one expression remains. We provide an illustration of the method below.
Example 3.1

schema S

\[ S = s_1 ; s_2 ; s_3 ; s_4 \]

\[ V + F (V) \]

\[ s_1 \]

DO \( V \)

\[ V + F (V) \]

\[ s_2 \]

\[ V + F (V) \]

\[ s_3 \]

END

DO \( V \)

\[ V + F (V) \]

\[ s_4 \]

END

\[ V + F (V) \]

\[ s_5 \]

input \( V_1 , V_2 \)

output \( V_1 , V_2 , V_3 \)

output vector after one application of Routine A to S:

\[ < H(V_1) [V_1, V_2, V_3], H(V_1) [V_1, V_2, V_3], F(V_1) > \]
Hereafter we present the translation only as it applies to the first component, $V_1$.

After two applications of Routine A to $S$ the first component is

$$H(G(V_1) [V_1, V_2] ) [G(V_1) [V_1, V_2], G(V_1) [V_1, V_2] , F(V_2) ]$$

After three applications of Routine A to $S$ the first component is

$$H(G(F(V_1)) [F(V_1), V_2] ) [G(F(V_1)) [F(V_1), V_2], G(F(V_1)) [F(V_1), V_2] , F(V_2) ]$$

Now in step 2, the following statement, $s_4$, of $S$ is reduced.

```plaintext
statement schema $s_4$

DO $V_1$

$V_2 + F(V_1)$

DO $V_2$

$V_2 = F(V_1)$

$V_1 = F(V_2)$

END

$V_1 + F(V_1)$

END
```

input $V_1, V_2, V_3$

output $V_1, V_2, (V_3$ implied output)

Output vector for $s_4$ after one application of Routine A

$$<F(V_1), H(V_1) [V_1], V >$$
After two applications

\[ \langle F_2 (V_1), H_1 (F_1 (V_2)) [F_1 (V_3)], V_3 \rangle. \]

\[ H_1 [V_2] = \langle V_1, V_2 \rangle \]

is associated with

DO V_2

\[ V_2 + F_1 (V_1) \]

\[ V_1 + F_1 (V_2) \]

END

so the expression for \( H_1 \) is

\[ \langle F_1 (F_1 (V_2)), F_1 (V_2) \rangle \]

Upon substitution the expression \( H_1 (V_2) [V_2] \) becomes

\[ \langle F_1 (F_1 (V_2)), F_1 (V_2) \rangle (V_2) [V_2] \]

which means that the vector function \( \langle F_1 (F_1 (V_2)), F_1 (V_2) \rangle \) is iterated \( V_2 \) times.

On making the substitutions for letters \( H_1 \) it is simpler to allow substitution of inputs directly into the functional expression for \( H_1 \). This renders the terms in square brackets redundant, and they are dropped. Thus the above form for \( H_1 \) is simplified to

\[ \langle F_1 (F_1 (V_2)), F_1 (V_2) \rangle (V_2) \]

Upon substitution for \( V_2 \) we get the complete \( H_1 \) expression as it occurs in translating \( s_4 \).

\[ \langle F_1 (F_1 (F_1 (V_1))), F_1 (F_1 (V_1)) \rangle (F_1 (V_1)) \]
(3.4) **functional expressions and the equivalence problem**

The first step in analysing schemes for equivalence is a translation to a functional expression. If two schemes have identical functional expressions, then they are equivalent. Thus letting $F_i$ be the functional expression for $S_i$, we know $F_i = F_j$ implies $S_i = S_j$. Although this test is sufficient, it is not necessary and we shall examine more subtle forms of schemata equivalence.

The "functional expression test" described here does isolate several interesting types of equivalence. For example, this test will locate schemata which differ for the following reasons:

(a) one scheme has redundant code (see $S_2$ of Example 2.2)

(b) the schemes differ only in the order of performing parallel operations (see Example 2.2)

The results of this section do not depend on the fact that the input functions are single argument, but in the next section this fact is critical.

§4 Callling Expressions for Loop Schemata

We consider now a family of simple formal languages which can be associated with schemata. In certain special cases the techniques of language theory give clear decision techniques for the equivalence problem and finiteness problem for the family. These lead to conceptionally simple solvable cases of the equivalence problem.

(4.1) **calling expressions**

From the methods of §3 it should be clear that the output of a single variable, $v$, of a Loop scheme $S$ depends on a sequence of "function calls", 
\[ f_1 \circ f_2 \circ \ldots \circ f_m (w) \]

where \( f_i \) are input functions and where \( f \circ g \) denotes the composition \( f(g(\cdot)) \). The calling sequence, \( i_1, i_2, \ldots \), is determined by the inputs and the schematic structure, but the overall form of such sequences is determined by a rather simple expression.

Given a finite alphabet \( \Sigma = \{a_1, \ldots, a_n\} \) define the following expressions.

**BNF**

\[
\text{<integer> ::= } \mathbb{N}^+ = \{1, 2, 3, \ldots \} \\
\text{<variable> ::= } n \text{<integer>} \\
\text{<superscript> ::= } \text{<variable>} | \text{<integer>} \\
\text{<letter> ::= } a_1 | a_2 | \ldots | a_n \\
\text{<term> ::= } \text{<letter>} | \text{<term>} <\text{term}> | (\text{<term>}) \text{<superscript>}
\]

**schematic form**

\[
c_1, c_2, \ldots \\
n_1, n_2, \ldots \\
s_1, s_2, \ldots \\
a_1, a_2, \ldots, a_n \\
\text{each } a_i \text{ is a term, and if } t_i \text{ are a terms then so are } t_i^s, t_i^t.
\]

The following are expressions over \( \Sigma = \{a, b\} : ab, a^n b^n, a^2 b^3 (ab)^n a, a^2 ((a^2 b)^n a (aba)^2 bab)^m b a \).

We say that these expressions are calling expressions, and they denote subsets of \( \Sigma^* \) in the usual manner, i.e., product is concatenation and exponentiation is limited closure. These sets are formal languages whose elements are calling sequences. The family of all such languages is the calling family for Loop.

(4.2) **calling expressions and combinatorial equivalence**

A simple variety of Loop schemata equivalence which is "deeper" than the equivalence of §3 arises directly from calling expressions. For each calling expression \( E \) there is a Loop scheme whose behavior is characterized by \( E \). The general idea should be clear from an example.
Example 4.1

Given an expression $a^2 b a^n b a^m$ over $L=\{a,b\}$ the corresponding scheme is given below where $f_1$ is $a$ and $f_2$ is $b$.

\[
\begin{align*}
V & \leftarrow F_1 (V) \\
V & \leftarrow F_1 (V) \\
V & \leftarrow F_2 (V) \\
& \text{DO } V_2 \\
V & \leftarrow F_1 (V) \\
& \text{END} \\
V & \leftarrow F_1 (V) \\
V & \leftarrow F_2 (V) \\
& \text{DO } V_3 \\
V & \leftarrow F_2 (V) \\
& \text{END}
\end{align*}
\]

Because of this correspondence, one necessary condition for recognizing Loop schemata equivalence is recognition of strong type of equivalence between calling expressions. In the case of calling expressions with only one variable, e.g., $a^n b^n$ or $a^2 b a^3 a^n b a^b$, etc, we need to test whether two formal languages, $L_1$ and $L_2$, in the calling family satisfy the condition that $(L_1 - L_2) \cup (L_2 - L_1)$ is finite. In a number of simple cases one can reduce this to the finiteness problems for context-free languages. It would be nice if the decision problem for the general case of schemata equivalence could be reduced to known solved problems, but that is not presently the case.

From the above correspondence the reader can construct examples of schemata equivalence even when the functional expressions are not identical. In the next subsection we outline a treatment of these "combinatorial" equivalence phenomena.

(4.3) **Strong equivalence of calling expressions**

Given two calling expressions $E_1, E_2$, (and a 1-1 correspondence $\pi$, ...
between variables) we form their difference, \( E_2^1 - E_2^1 \), as follows. Let \( e_i^j (n_1, \ldots, n_p) \) be the \( i \)-th letter in the sequence from \( E_j \) with values \( n_1, \ldots, n_p \) for the variables. Then \( e_2^1 (n_1, \ldots, n_p) - e_2^1 (n_1, \ldots, n_p) = (\text{if } e_2^1 (n_1, \ldots, n_p) = e_2^1 (n_1, \ldots, n_p) \text{ then } 1 \text{ else } 0) \) and

\[ E_2^1 - E_2^1 = \{ e_2^1 (n_1, \ldots, n_p) - e_2^1 (n_1, \ldots, n_p) \mid n_i \in \mathbb{N} \}. \]

We are interested in the problem of finding for any \( E_1, E_2 \) (and any 1-1 correspondence between their variables) the largest \( k \) such that if \( w \in E_2^1 - E_2^1 \) and \( |w| \geq k \), then \( w \in \{1\}^* \), i.e. we want the largest \( k \) above which the calling expressions are always equal. If there is no such finite \( k \), we expect to get the answer, \( k = \infty \). If \( k < \infty \) we say \( E_1 \) and \( E_2 \) are strongly equivalent beyond \( k \).

To apply the concept of a calling sequence to schemata, observe that a calling sequence \( f_1^{\circ} \ldots \circ f_i^{\circ} \) over \( \Sigma = \{f_1, \ldots, f_n\} \) is a functional \( F[\_]: \mathcal{F}^n \times \mathbb{N} \rightarrow \mathbb{N} \).

We then have

**Lemma 4.1** Two calling sequences \( f_1^{\circ} \ldots \circ f_i^{\circ} \) and \( f_1^{\circ} \ldots \circ f_j^{\circ} \) are equivalent (as functionals) iff they are identical.

**Proof:** The proof is by induction on the length of the sequences. If \( p = 1 \), say \( f_1^{\circ} = f_1^{\circ} \) then if \( q = p \) and \( f_1^{\circ} = f_1^{\circ} \), then the functionals are clearly equivalent. If \( q = p \) and \( f_1^{\circ} = f_1^{\circ} \), then pick a value of \( f_1^{\circ} \) different than \( f_1^{\circ} \), say \( f(0) \neq f_1^{\circ}(0) \). In case \( q > 1 \) and \( f_1^{\circ} = f_1^{\circ} \), pick \( f_1^{\circ}, f_2^{\circ} \) to be strictly increasing. Then clearly \( f_1^{\circ}(0) \neq f_1^{\circ}(0) \).

The case for \( p > 1 \) is nearly identical to the base step and is left to the reader.
Again one uses strictly increasing $f(\quad)$'s to produce values where the functionals differ. Q.E.D.

We now state without proof a basic lemma.

**Lemma 4.2** There is an algorithm to determine for any two calling expressions $E_1, E_2$ (and correspondence between variables, $\pi$) the largest $k$ (including $\infty$) for which $|w|^\geq k$ and $w \in E_1 - E_2$ implies $w \in \{1\}^\ast$.

The essential idea behind the lemma is that the behavior of $e_{1_1, \ldots, 1_p} - e_{2_1, \ldots, 2_p}$ is periodic in the length of the subscripted terms and thus any potentially infinite differences can be detected in a finite set of calling sequences.

Finally, in the next section we need the notion of nested calling expressions. These arise by allowing substitutions of calling expressions for variables. The following is a nested calling expression.

$$a_{(ba)}(a^2b^p)^a_{ba(ba)(aba^4)}$$

It is nested to depth two (over the letter $a$).

**4.4 Loop schemata equivalence**

Using the two lemmata of the previous subsection we indicate a method of solving the schemata equivalence problem. Given two schemes

$$S_1[f_1, \ldots, f_{t_1}, x_1, \ldots, x_{n_1}]$$

and

$$S_1[f_1', \ldots, f_{t_2}, x_1', \ldots, x_{n_2}]$$

we consider all possible correspondences between variables (inputs and outputs) adding dummy variables if necessary. Without loss of generality then we consider only one assignment and assume $t_1 = t_2 = t$, $n_1 = n_2 = n$. Thus each $S_i$ satisfies

$$S_i[\quad]: \mathcal{F}^t \times \mathbb{N}^n \rightarrow \mathbb{N}^p$$

For each output variable $y_i$ we follow the same plan, so consider only one variable, say $y$. We outline a method of testing which at each stage reduces the length or the depth of nesting of the expressions
being tested. Thus eventually the process stops. In a formal proof we would set up an induction on length and depth.

**Equivalence test**

In the process of carrying out the "functional identity test" of §3, we form nested calling expressions for each input variable in the obvious manner. For instance, the nested calling expression for \( v \) in Example 3.1 is

\[
(f \circ (f \circ f)^{(1)} \circ (V_1)^{(3)} \circ f)^{(2)} \circ f \circ (f \circ f)^{(1)} \circ (V_2)^{(2)}
\]

The first step in the equivalence test is to compare the nested calling expressions in \( S_1 \) and \( S_2 \) after one application of Routine A. Three possible cases can occur depending on the form of the expression.

1. each expression begins with a letter

\[
S_1 \quad \quad S_2
\]

\[
f_{i_0} \quad \quad f_{j_0}
\]

2. one expression begins with a letter and one with an iterative, say the case is as illustrated

\[
S_1 \quad \quad S_2
\]

\[
f_{i_0} (W_1) \quad (E_2)^{(2)} (W_2)
\]

3. each begins with an iterative

\[
S_1 \quad \quad S_2
\]

\[
(V) \quad (E_1)^{(2)} (W_2)
\]

For example, the first stage expression for Example 3.1 is

\[
(f \circ (f \circ f)^{(1)} \circ (V_1)^{(2)} \circ f)^{(2)} \circ f \circ (f \circ f)^{(1)} \circ (V_2)^{(2)}.
\]

We consider the cases separately.
(1) In the first case, if \( f_{i_0} \neq f_{j_0} \) the test is complete and the schemes are not equivalent. If \( f_{i_0} = f_{j_0} \), then continue to the next stage by producing an expression for the next application of Routine A.

(2) In the second case, continue applying Routine A to \( S \) until an iterative is discovered. Suppose the form is then \( f_{i_0} \circ \ldots \circ f_{i_p} (E_1)(w_1) \).

Now apply the equivalence test to the nested calling expressions for \( v_1 \) and \( v_2 \) (causing a recursive call to the procedure but a reduction in the level of composition). Determine the maximum \( k \) for which they differ. If this \( k \) is finite, then check equivalence for each of the possible values (now the depth of nesting is less). If none of them produce a non-equivalent interpretation, then consider the case "\( v_1 = v_2 \)" below.

If \( v_1 \neq v_2 \) infinitely often (i.e., \( k \) is infinite), then \( S \notin S \). We leave this non-trivial case for the reader. (Again one needs strict monotonicity of the \( f_j \) to have the freedom to choose values for the \( f_{i_j} \) after fixing values of \( v_{i_1} \) and \( v_{i_2} \) which produce calling sequences of different lengths. The argument is a more subtle form of Lemma 4.1).

If \( v_1 = v_2 \), then the only way to have equivalence is to have \( E_2 \) begin with \( f_{i_0} \circ \ldots \circ f_{i_p} \). If it continues, say with nested expression \( T \), then \( E \) must begin with something equivalent to \( T \), and this can be checked by applying this procedure (again the degree of nesting is lower). If this happens, then we check whether "\( T \circ w_2 \in w_1 \)", i.e., the expression for \( w_2 \) composed with \( T \) is equivalent to the expression for \( w_1 \). This is checked by applying this procedure (now the level of composition has been reduced).

(3) The third case proceeds in a manner similar to case (2), and its outline is left to the reader.

end of procedure
§5 Conclusion and open problems

We have defined a new class of program schemata based on subrecursive programming languages. These Loop schemata are fundamentally different than program schemata as defined in Paterson [6] because their interpretation requires specific functions (\(\cdot\)-1) and tests (v=0). They represent a wide and interesting class of programs, but even more interesting classes are suggested by the general principle of defining schemata from subrecursive languages. We consider another example below.

We are able to uniformly assign mathematical expressions to these schemata. Such a task is not yet accomplished for more general program schemata, (and an effort on that task appears worthwhile), but in the case of Loop, the idea is quite simple and can be applied also directly to the Loop languages.

Finally, we have outlined a solution to the Loop schemata equivalence problem although the equivalence problem for Loop programs is unsolvable (of level \(\Pi^0_1\)). We have indicated two levels of schemata equivalence, "functional" and "deep" (or combinatorial) equivalence.

Carrying over these ideas to a language like Conditional Loop (described below) and developing a good equivalence algorithm might have practical merit.

(5.1) more general subrecursive schemata

We can extend Loop schemata to permit nested conditionals. If we adjoin the following categories to Loop schemata, the resulting programs are called Conditional Loop Schemes.

\[
\begin{align*}
\text{<predicate variable><:}= & P | P \text{<predicate variable>}
\text{<predicate><:}= & \text{<predicate variable><(variable)}> \\
\text{<conditional><:}= & \text{if <predicate> then <program> else <program>}
\end{align*}
\]
Example 5.1 (1) if P(V) then V+F(V) else V+FF(V)

(2) DO VI
    V  + F (V )
    1   1 2
    if P(V1) then DO V2
      V  + F (V )
      2   1 1
      END
    else DO V1
      V  + F (V )
      2   2 2
      END
    END

V  + F (V )
2   1 2

If in addition to the functions +1, 0, ( ), we specify (as in §2) the predicates P1(x) iff "x=0" and P1(x) iff "x≠0", in the Conditional Loop schemata, the resulting language is called the Conditional Loop language. By the results of Constable and Borodin [1], this language computes the same class of total functions as the Loop language. The similarity of this type of language to interesting subsets of Algol is obvious. Investigation of the schemata equivalence problem for Conditional Loop (allowing n-ary function inputs) would appear to be an interesting non-trivial (but tractable) problem.

(5.2) equivalence over special function domains

One reason for the solvability of Loop schemata equivalence is our freedom to choose values for the function inputs, i.e., the schema does not provide much information. Therefore, one reason that Loop program equivalence is unsolvable is that specific inputs, like +1, -1, ( ), provide too much information. It would be interesting to consider the effects of a more gradual increase in information.

In this regard, it is noteworthy that +1, -1, are inverse and ( )
is an identity in the function space $\mathcal{F}$. What is the consequence for schemata equivalence of requiring the function inputs to come from a group? Such a restriction presents an increase of information (we at least know the symmetry but not its exact nature).

These observations suggest an investigation of the schemata equivalence problem for various algebraic function domains such as groups and rings, etc.

Acknowledgements

The author would like to thank John Cherniavsky and Professor Robert Wagner of Cornell for their helpful discussions on the topics in this paper and Diane Goolsby for her excellent typing.

References


