Subrecursive Program Schemata I & II

I. Undecidable Equivalence Problems

II. Decidable Equivalence Problems

by

R. L. Constable & S. S. Muchnick

TR 72 - 124 (Revised)

June 1972

Department of Computer Science
Cornell University
Ithaca, New York 14850

Table of Contents

Subrecursive Program Schemata I & II

I. Undecidable Equivalence Problems

II. Decidable Equivalence Problems

Robert L. Constable and Steven S. Muchnick
Cornell University

Abstract

The study of program schemata and the study of subrecursive programming languages are both concerned with limiting program structure in order to permit a more complete analysis of algorithms while retaining sufficiently rich computing power to allow interesting algorithms. In this paper we combine these approaches by defining classes of subrecursive program schemata and investigating their equivalence problems. Since the languages are all subrecursive, any scheme written in any one of them must halt (as long as we assume the basic functions and predicates are all total). Hence equivalence of schemes is the first question of interest we can ask about these languages.

We consider schematic versions of various subrecursive programming languages similar to the Loop language. We distinguish between Pre-Loop and Post-Loop languages on the basis of whether the exit condition in an iteration loop is tested before iteration, as in Algol (Pre-), or after iteration, as in FORTRAN (Post-). We show that at the program level all these languages have the same computing power (the primitive recursive functions) and all have unsolvable equivalence problems (of arithmetic degree Π_1^0). But at the level of schemes, Pre-Loop has an unsolvable equivalence problem, while at least one formulation of

Post-Loop has a solvable equivalence problem.

If L is a programming language or scheme language, then we denote by E(L) the equivalence problem in L.

The basic languages considered are

Loop (≡ Pre-Loop)	Loop language for primitive recursive functions
Post-Loop	Post-Loop language for primitive recursive functions
$\text{Loop}_{\Diamond}$	Loop language with restricted conditionals
L [D, ( )]	Loop schemata over D with identity
$L_{\Diamond}$ [D, ( )]	Loop schemata with conditionals
PL [D, ( )]	Post-Loop schemata over D
$PL_{\Diamond}$ [D, ( )]	Post-Loop schemata with conditionals
P	Program (flowchart) schemata
P_d	Program schemata with DO-statements.

In contrast to (pure) Loop schemata studied previously by the first author, some of these schemata languages contain the identity function so that a pure data transfer, X ← Y , is possible. Moreover, the equivalence algorithms given here are for the special case of linear schemes (to be defined below) with monadic function variables. Linear schemes are designated by placing L before the name of the more general class, thus LL for linear Loop, LPL for linear Post-Loop, etc. In all schemes considered here the functions are monadic, so no special designation of function rank is provided.

It is well known that $E(P)$ is recursively unsolvable and $E(P) \in \Pi_2^0$. We show that $E(\text{Loop})$, $E(\text{Post-Loop})$, $E(L_\diamond)$ (both with and without the pure data transfer), and $E(L)$ are recursively unsolvable, while $E(LPL)$ is recursively solvable.

The extension of the equivalence algorithm for LPL to polyadic functions appears at present to be a tedious but straightforward modification to the monadic algorithm. We are hopeful that a simpler and more generally applicable technique will emerge for demonstrating solvability or unsolvability of this class of equivalence problems. The algorithm and proofs given here are but a crude first step in delimiting this problem.

§1. Introduction

We assume the reader is familiar with <u>program schemata</u> say as presented in [3] or [5] (also called <u>flowchart</u> schemata [8] , or <u>abstract programs</u> [9]). The class of such schemata is denoted P . Briefly these schemata are finite sequences of <u>statement schemes</u> of the <u>assignment form</u>,

$$y \leftarrow f(x_1,\ldots,x_n)$$

where x_i , y are <u>individual variables</u> and f is a function variable, or of the <u>conditional form</u>,

$$\text{IF } P(x_1,\ldots,x_n) \text{ THEN } S_1 \text{ ELSE } S_2$$

where P is a predicate variable and S_1 and S_2 are statements, or of the <u>go-to</u> form,

GO TO L

where L is a statement label.

The semantics is conventional when the possible contents of individual variables are specified, say domain D , and when function variables are assigned functions from D^n into D and predicate variables predicates from D^n to {true,false}.

In this paper we consider a different type of program scheme. We fix some of the interpretation by defining two types of iterative (or DO) statements. We require a function $|\cdot| : D \to \mathbb{N}$ which specifies the number of times the statements within an iterative are to be executed. This results in the class of program schemata with DO-statements $P_d(D)$.

We then restrict the structure of $P_d(D)$ programs by eliminating GO TO's which cause branches to statements before the GO TO (such GO TO's are the only way to form loops, and are called backwards or negative GO TO's). Such a restricted language is called $L_{\Diamond\!\downarrow}$ (L for "loop", "$\Diamond$" for the usual flow chart symbol for conditional, and "$\downarrow$" for the downward direction of the GO TO). Usually we write only $\Diamond$ for $\Diamond\!\downarrow$.

When these schemata are provided with specific interpretations for their functions and predicates, specific programming languages result. For example, allowing only the functions $f(x) = x+1$, $f(x) = x\dot{-}1$ and the predicate $p(x)$ iff $x \neq 0$, the language P is equivalent to the programming language G_3 (see [2]), conveniently abbreviated as $P[+1, \dot{-}1, \neq 0]$. As Shepherdson & Sturges [6] and Minsky [11] show, this language is universal, i.e. it contains programs for all recursive functions. When $L_\Diamond$ is provided the same functions and predicates, it contains programs for precisely the primitive recursive functions $\mathcal{R}^1$ (see [2]). We denote the language by $L_\Diamond[+1, \dot{-}1, \neq 0]$. This language is similar to SR which is discussed at length in [2]; both languages compute $\mathcal{R}^1$.

At the level of specific programming languages like $P[+1, \dot{-}1, \neq 0]$ and $L_\Diamond[+1, \dot{-}1, \neq 0]$, the equivalence of programs is quite familiar, namely two programs ϕ_i and ϕ_j are equivalent, written $\phi_i \equiv \phi_j$, iff

$$\forall x_1,\ldots,x_n \; \phi_i(x_1,\ldots,x_n) = \phi_j(x_1,\ldots,x_n)$$

where the equality tacitly means that if ϕ_i halts on its input, then so does ϕ_j and conversely (for $L_\Diamond[+1, \dot{-}1, \neq 0]$ all programs halt so this convention is vacuous).

It is also well known that the equivalence problem is unsolvable (at level Π_2° for $P[+1, \dot{-}1, \neq 0]$ and at level Π_1° , for $L_\Diamond[+1, \dot{-}1, \neq 0]$) .

At the schema level, we say that two schemes Φ_i and Φ_j having as inputs functions $f_1,\ldots,f_\ell$, predicates $p_1,\ldots,p_m$ and elements of D, $x_1,\ldots,x_n$ are <u>equivalent</u> (over D) iff

$$\forall f_1,\ldots,f_\ell,\ p_1,\ldots,p_m,\ x_1,\ldots,x_n$$
$$\Phi_i\ [f_1,\ldots,f_\ell,\ p_1,\ldots,p_m,\ x_1,\ldots,x_n] =$$
$$\Phi_j\ [f_1,\ldots,f_\ell,\ p_1,\ldots,p_m,\ x_1,\ldots,x_n]\ .$$

It is known from [8] that schema equivalence for P is unsolvable (of arithmetic level Π_2^0). At this point we enter the scene with questions about equivalence in $L_\Diamond$ and various related languages.

Before launching into the technical results, let us consider some motivation for our interest in the subrecursive schemata equivalence problem. When the equivalence problem is solvable for a recursive class of programs, it is possible to find the minimum length program equivalent to any program, and more generally it is possible to put programs into any number of canonical forms. From such forms we may read off important structural properties of the original program (as in algebra, where the diagonal form of a matrix displays the eigenvalues). A sweeping view of the role of the equivalence problem is expressed by A. P. Ershov in [ ], "...only the fundamental problem (equivalence problem) solution will give the first adequate material for an algebra of programming." [parentheses ours].

From a less general point of view, we might expect that an algorithm for a non-trivial form of the equivalence problem will be an interesting combinatorial tool in the same way that the finite automaton equivalence algorithm is, e.g. in Hopcroft's application of automata equivalence to planar graph isomorphism [6].

With these hopes and goals in mind, we turn to a look at subrecursive schemata equivalence. The table below summarizes our results. An entry D means decidable, U means undecidable.

<table>
<tr><th></th><th></th><th>Non-linear</th><th>Linear</th></tr>
<tr><td rowspan="2">Conditional Pre-Loop Schemata $L_{\Diamond}$</td><td>With $Y \leftarrow X$</td><td colspan="2">U Thm. 3</td></tr>
<tr><td>Without $Y \leftarrow X$</td><td colspan="2">U Thm. 4</td></tr>
<tr><td rowspan="2">Conditional Post-Loop Schemata $PL_{\Diamond}$</td><td>With $Y \leftarrow X$</td><td colspan="2">U Thm. 5</td></tr>
<tr><td>Without $Y \leftarrow X$</td><td colspan="2">U Thm. 5</td></tr>
<tr><td rowspan="2">Pre-Loop Schemata L</td><td>With $Y \leftarrow X$</td><td>U Thm. 6</td><td>U Thm. 7</td></tr>
<tr><td>Without $Y \leftarrow X$</td><td>U Thm. 8</td><td>?</td></tr>
<tr><td rowspan="2">Post-Loop Schemata PL</td><td>With $Y \leftarrow X$</td><td>U Thm. 9</td><td>D Thm. 11</td></tr>
<tr><td>Without $Y \leftarrow X$</td><td>?</td><td>D Thm. 11</td></tr>
</table>

§2. Preliminaries

(2.1) Syntactic Categories

We describe the types of statement schemes from which the various classes of schemata will be defined (in an extended BNF format).

(i) <variable> :: = V | V <variable>

(ii) <function variable> :: = <function name>$^{\text{<rank>}}$
<function name> :: = F | F<function name>
Where <rank> is a positive integer.

(iii) <term> :: = <function variable> (<variable>,...,
<variable>) | <variable>
The number of occurrences of <variable> as an argument matches the rank of the <function variable> .

(iv) <assignment scheme> :: = <variable> ← <term>

(v) <predicate variable> :: = <predicate name>$^{\text{<rank>}}$
<predicate name> :: = P | P<predicate name>
Where <rank> is a positive integer.

(vi) <predicate> :: = <predicate variable> (<variable>,
...,<variable>)
The number of occurrences of <variable> matches the rank of the <predicate variable>.

(vii) <conditional scheme> :: = IF <predicate> THEN
<label> ELSE <label>

(viii) <go to> :: = <go to +> | <go to ->

(ix) <go to +> :: GO TO + <label>

(x) <go to -> :: GO TO - <label>

(xi) <label> :: = L|L <label>

(xii) <iterative scheme> :: = DO<variable>; <scheme>; END

(xiii) <post-iterative scheme> :: = DO;<scheme>; TEST <variable>

(xiv) <halt scheme> :: = HALT

(xv) <null scheme> ::=

The definitions of <statement scheme> and <scheme> will depend on the class of schemes being constructed.

(2.2) Abbreviations:

As abbreviations we use V_n for V...V n-times, F_i^n for $F...F^n$ i-times, P_i^n for $P...P^n$ i-times, and L_n for L...L n-times. We also use u_i, v_i, w_i to denote individual variables and h_i to denote function variables.

The statement types are also abbreviated as follows: $\leftarrow$ for <assignment>, $\Diamond$ for <conditional>, $\begin{array}{c}\text{DO}\\ \text{END}\end{array}$ for <iterative> $\begin{array}{c}\text{DO}\\ \text{TEST}\end{array}$ for <post-iterative>, $\downarrow$ for <go to +>, $\uparrow$ for <go to -> .

We will also let S_i, T_i and capital Greek letters Φ , Ψ ,... denote schemes. Lower case Greek letters ϕ , ψ ,... will denote programs.

(2.3) Classes of Schemata

(i)* The program schemata P are defined by allowing

<unlabeled statement scheme> :: = <assignment scheme> | <go to +> | <go to -> | <conditional scheme> | <null scheme>

<statement scheme> :: = <unlabeled statement scheme> | <label> : <statement scheme>

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

Symbolically and informally, $P = [\leftarrow, \downarrow, \uparrow, \Diamond]$. We assume here (and below) that every label used in a go to or conditional also occurs exactly once as the label of a statement.

(ii) The program schemata with DO-statements P_d are defined by changing the definition of <unlabeled statement scheme> in the specification of P to

<unlabeled statement scheme> :: = <assignment scheme> | <go to +> | <go to -> | <conditional scheme> | <iterative scheme> | <null scheme>

* Sometimes, see [8], the class P is not allowed the pure transfer, $v \leftarrow w$, but this addition does not significantly affect the theory presented here.

No branch may be into an iterative from outside it in P_d, or in what follows.

(iii) The conditional (Pre-) Loop schemata $L_\Diamond$ are defined by allowing

<unlabeled statement scheme> :: = <assignment scheme> | <conditional scheme> | <iterative scheme> | <go to ↓> | <halt scheme> | <null scheme>

<statement scheme> :: = <unlabeled statement scheme> | <label> : <statement scheme>

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

No label in a go to or conditional can refer to that same statement or one preceeding it.

$$L_\Diamond = [\leftarrow, \downarrow, \Diamond, {}^{\mathrm{DO}}_{\mathrm{END}}]$$

(iv) The (Pre-) Loop schemata L are defined by allowing

<statement scheme> :: = <assignment scheme> | <iterative scheme>

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

$$L = [\leftarrow, {}^{\mathrm{DO}}_{\mathrm{END}}]$$

(v) The <u>conditional</u> <u>Post-Loop</u> <u>schemata</u> $PL_{\Diamond}$ are defined by allowing

<unlabeled statement scheme> :: = <assignment scheme> |
<conditional scheme> | <post-iterative scheme> |
<go to +> | <halt scheme>

<statement scheme> :: = <unlabeled statement scheme> |
<label> : <statement scheme>

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

$$PL_{\Diamond} = [\leftarrow, \downarrow, \Diamond, {}^{\text{DO}}_{\text{TEST}}]$$

(vi) The <u>Post-Loop</u> <u>schemata</u> PL are defined by allowing

<statement scheme> :: = <assignment scheme> |
<post-iterative scheme>

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

$$PL = [\leftarrow, {}^{\text{DO}}_{\text{TEST}}]$$

(2.4) Examples:

In this section we give èxamples of each type of scheme.

(i) program schemes

(a) a sample P scheme

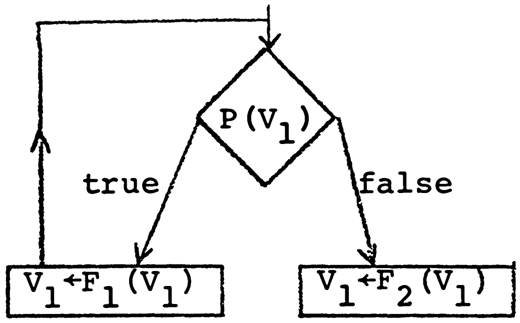
L_1 : IF $P(V_1)$ THEN $+L_2$ ELSE $+L_3$;

L_2 : $V_1 \leftarrow F_1(V_1)$;

GO TO $-L_1$;

L_3 : $V_1 \leftarrow F_2(V_1)$

the flow chart for this scheme is simply



(b) a more complex example of a P scheme

L_1 : $V_1 \leftarrow F_2(V_1)$;

L_2 : IF $P_1(V_1)$ THEN $+L_3$ ELSE $+L_4$;

L_3 : $V_1 \leftarrow F_2(V_1)$;

IF $P_2(V_1)$ THEN $+L_6$;

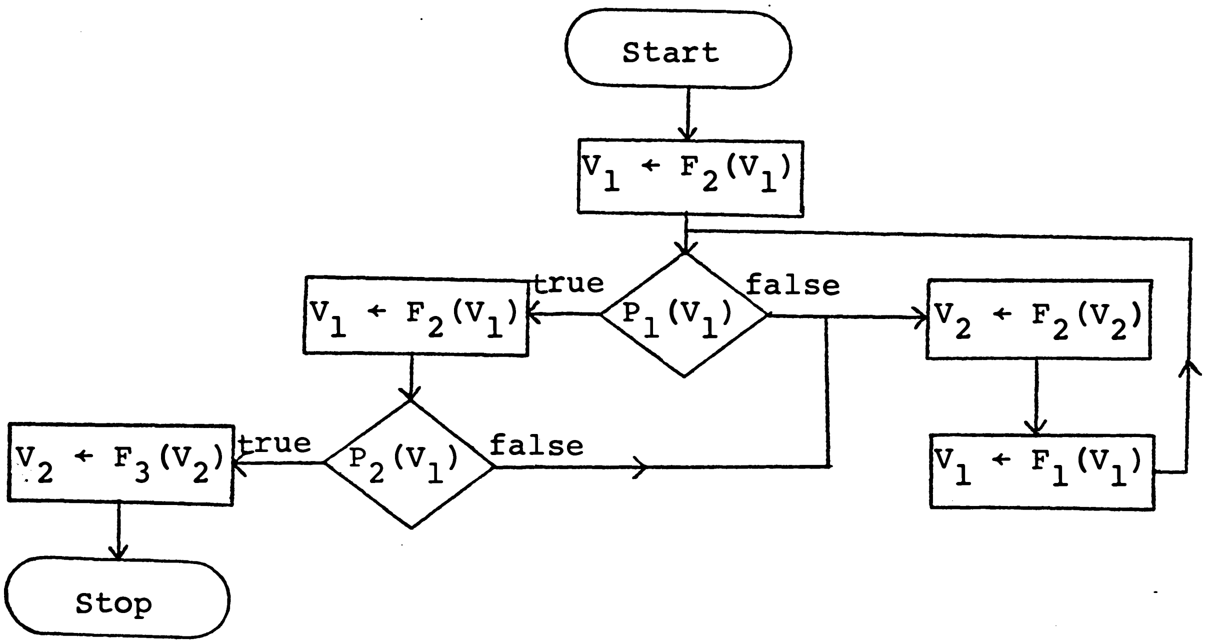
L_4 : $V_2 \leftarrow F_2(V_2)$;

$V_1 \leftarrow F_1(V_1)$;

L_5 : GO TO $-L_2$;

L_6 : $V_2 \leftarrow F_3(V_2)$

This scheme can be represented by the flowchart



(ii) a P_d scheme

$V_1 \leftarrow F_2(V_1)$;

L_1 : DO V_1;

$V_2 \leftarrow F_2(V_2)$;

END;

$V_1 \leftarrow F_2(V_1)$;

IF $P_2(V_2)$ THEN $+L_2$ ELSE $+L_3$;

L_2 : $V_1 \leftarrow F_1(V_1)$;

GO TO $-L_1$;

L_3 : $V_2 \leftarrow F_2(V_2)$

(iii) An L scheme (to simulate (i.a) for V_2 steps)

```
      DO V2;
      IF P(V1) THEN +L2 ELSE +L3;
L2 :  V1←F1(V1);
      GO TO +L1;
L3 :  V1←F2(V1);
      GO TO +L4;
L1 :  END;
L4 :
```

(iv) An L scheme to simulate (iii) under the assumption that F_P is a characteristic function for P .

```
DO V2;
   V3 ← FP(V1);
   V4 ← V1;
   DO  V3;
        V1 ← V4;
        V1 ← F1(V1);
   END;
END;
V1 ← F2(V1)
```

(v) A $PL_{\Diamond}$ scheme

```
        V_2 ← F_3(V_2);
        DO;
          IF P(V_1) THEN +L_1 ELSE +L_2;
   L_1 :  V_1 ← F_1(V_1);
        TEST    V_2;
   L_2 : V_1 ← F_2(V_1)
```

(2.5) Semantics

The meaning of assignments and conditionals should be clear. Once a domain D is specified, then individual variables V_i hold contents from D, and $v \leftarrow h(w)$ means that when h is interpreted as a function from D to D, then v receives the value of h applied to the contents of w. Likewise, when predicate variables, p, are assigned predicates from D to {true, false}, the conditional, IF p(v) THEN L_1 ELSE L_2, means if the predicate assigned to p is true for the contents of v then GO TO L_1, otherwise GO TO L_2. For example, the function $f(x) = x+1$ might be assigned to h and the predicate $x \neq 0$ might be assigned to p.

The iterative occurring in the context of schemes S_1, S_2, and S_3, S_1; DO v; S_2; END; S_3, means

$$
\begin{array}{ll}
 & S_1; \\
 & \bar{v} \leftarrow |v|; \\
L_1 : & \text{IF } \bar{v} = 0 \text{ THEN } +L_2; \\
 & S_2; \\
 & \bar{v} \leftarrow \bar{v} \dot{-} 1; \\
 & \text{GO TO } -L_1; \\
L_2 : & S_3
\end{array}
$$

where $\bar{v}$ assumes values in $\mathbb{N}$.

The post-iterative occurring in the context of S_1, S_2, S_3, S_1; DO; S_2; TEST v; S_3, means

$$
\begin{array}{ll}
 & S_1; \\
 & \bar{v} \leftarrow |v|; \\
L_1 : & S_2; \\
 & \bar{v} \leftarrow \bar{v} \dot{-} 1; \\
 & \text{IF } \bar{v} = 0 \text{ THEN } +L_2 \text{ ELSE } -L_1; \\
L_2 : & S_3
\end{array}
$$

where $\bar{v}$ assumes values in $\mathbb{N}$.

Notice that for $v > 0$, DO v; S; END and DO; S; TEST v have equivalent meaning.

The directional go to's also have a straightforward meaning. GO TO +L means that the label L must be below the GO TO in the linear sequence of program statements; GO TO -L means the label L must lie above the GO TO in the sequence of statements.

We regard schemes as programs for computing mappings. So inputs are given and certain variables are regarded as outputs when (and if) the program halts (produces a finite computation). To be precise about this, one should specify the inputs and define a computation expressed by a scheme. We will be precise about the first point because it is not often treated in the literature, but we leave the definition of computation to the reader since it is similar to the case of defining computations for register machines which is done well in the literature (see [12] or [16]).

To be precise about inputs to a scheme, let $\mathcal{F}^n(D)$ be the set of functions $f(\ ) : D^n \to D$ and let $\mathcal{F}(D) = \bigcup_{i=1}^{\infty} \mathcal{F}^i(D)$. Also let $\mathcal{P}^n(D)$ be the set of predicates $p(\ ) : D^n \to \{\underline{\text{true}}, \underline{\text{false}}\}$ and put $\mathcal{P}(D) = \bigcup_{i=1}^{\infty} \mathcal{P}^i(D)$.

A mapping from special subsets* of $\mathcal{F}(D)^{n_1} \times \mathcal{P}(D)^{n_2}$ to D is called a functional.

A scheme computes a <u>functional</u> when we tell how to assign inputs. To do this we list all function and predicate variables in order of occurrence. We then select a set of individual variables as inputs. (These are all the variables w whose first occurrence is not on the left hand side of an assignment; in $v \leftarrow f(v)$ we say that v occurs first on the right. They are also called <u>right variables</u>.) We then list these arguments of the scheme and write $\Phi[f_1,\ldots,f_{n_1},$

---

* Note that $\mathcal{F}^m(D)^n$ means $\mathcal{F}^m(D) \times \mathcal{F}^m(D) \times \ldots \times \mathcal{F}^m(D)$ n-times. The subsets of interest are of the form

$$\mathcal{F}^{\ell_1}(D) \times \mathcal{F}^{\ell_2}(D) \times \ldots \times \mathcal{F}^{\ell_{n_1}}(D) \times \mathcal{P}^{m_1}(D) \times \ldots \times \mathcal{P}^{m_{n_2}}(D) \times D^{n_3}$$

but these are too complex to write, so we leave them incompletely described.

$p_1,\ldots,p_{n_2},x_1,\ldots,x_{n_3}]$. The scheme then computes a functional from

$$\mathcal{F}^{rf_1}(D) \times\ldots\times \mathcal{F}^{rf_{n_1}}(D) \times \mathcal{P}^{rp_1}(D) \times\ldots\times \mathcal{P}^{rp_{n_2}}(D) \times D^{n_3} \text{ to } D$$

where rf_i and rp_j are the ranks of f_i and p_j , respectively.

So in short, a scheme computes a functional and a functional is simply a function which takes both individual and function inputs to produce an individual output.

The class of all functionals over D is here denoted $\mathbb{F}(D)$. The class of all recursive functionals over D (see Rogers [14]) is denoted $R\mathbb{F}(D)$. All the scheme languages presented compute subclasses of $R\mathbb{F}(D)$.

(2.6) Concrete Programming Languages

Given a scheme $\Phi[f_1,\ldots f_{n_1}, p_1,\ldots,p_{n_2}, x_1,\ldots,x_{n_3}]$, if all the functions and predicates are interpreted, the result is a program in the usual sense, and the program computes a function $D^{n_3} \to D$. So if a finite class of functions and predicates is provided as a fixed interpretation for the variables in a class of schemes, the result is a programming language.

A convenient way to denote such languages is to use the scheme letter, e. g. P , followed by a list of the fixed functions and predicates. So $P[+1, \dot{-}1, \neq 0]$ describes a programming language. (Where +1 abbreviates $f(x) = x+1$*, $\dot{-}1$ abbreviates $f(x) = x\dot{-}1$ and $\neq 0$ abbreviates $p(x)$ iff $x \neq 0$).

With this convention we can describe several programming languages of special interest.

(i) $P[+1, \dot{-}1, \neq 0]$ is equivalent to G_3 of [2],

(ii) $L_{\diamond}[+1, \dot{-}1, +, -, \cdot, \div, =0, \neq 0]$ is SR of [2],

(iii) $P[+1, 0]$ is the Loop language of [2] and [10], (here 0 means $v \leftarrow 0$).

The objects in these programming languages are programs and are denoted with lower case Greek letters, e.g. ϕ_i .

(2.7) Linear Loop Languages and Their Power

The fact that the class of functions computed by Loop programs is $\mathcal{R}^1$ means that arbitrary n-argument primitive recursive functions can be generated by the Loop operations starting from unary functions ($f(x) = x+1$, $f(x) = x$, and $f(x) = 0$). We want to establish this fact

* To be precise, only the schemes having unary function and predicate letters are interpreted. We might denote this subset of P by P^1 . Moreover, the assignment of functions to letters must be specified, e.g. F_1 is +1, F_2 is $\dot{-}1$, F_3 is +1, F_4 is $\dot{-}1$, etc. (and all P_i are $\neq 0$).

for PL programs and show besides that $\mathcal{R}^1$ can be obtained from a special form of Loop and Post-loop schemes called <u>linear schemes</u>. (It is for the class of linear Post-Loop schemes that we later solve the equivalence problem explicitly.)

<u>Definition 1:</u> We say that a class of programs (schemes) C <u>computes a class of functions (functionals)</u> $\mathcal{C}$ iff every program (scheme) in C computes a function (functional) <u>in</u> $\mathcal{C}$, and for every function (functional) in $\mathcal{C}$ there is a program (scheme) in C computing it.

Two classes of programs (schemes) have the same <u>power</u> iff they compute the same class of functions (functionals).

<u>Definition 2:</u> Let $\lfloor x \rfloor$ be the greatest integer less than or equal to x, i.e. the integer part of x . Define quadres (x) $= x \dot{-} \lfloor \sqrt{x} \rfloor^2$, called the <u>quadratic residue</u> of x .

<u>Theorem 1:</u> PL[+1, $\dot{-}$1, quadres] computes $\mathcal{R}^1$.

<u>Proof:</u> For the proof we rely on a combinatorial fact about $\mathcal{R}^1$ proved by Péter [13]. We say that a function f( ) is defined from h( ) by pure iteration iff f(0) = 0 and f(n+1) = h(f(n)) .

<u>Fact 1:</u> The class $\mathcal{R}^1$ can be generated from the functions x+1, x+y, quadres (x) by the operations of function composition and pure iteration.

To obtain X ← 0 use the program

```
DO;
X ← QUADRES(X);
TEST X
```

The basic functions, except for $x+y$, are in our class. To obtain $x+y$ use the program

```
Y ← Y + 1;
DO;
X ← X + 1;
TEST Y;
X ← X ∸ 1;
Y ← Y ∸ 1
```

Given two functions $f_1(\ )$, $f_2(\ )$ computed by programs f_1 with inputs $x_1,\ldots,x_n$ and output y_1 and f_2 with output y_2, the composition $f_1(x_1,\ldots,x_{k-1},f_2(\ ),$ $x_{k+1},\ldots,x_n)$ is computed by f_2; $x_k \leftarrow y_2$; f_1 which is again a program in the class.

If $f_1(\ )$ is computed by f_1 with input x and output y, then the function $f_2(\ )$ defined by $f_2(0) = 0$, $f_2(n+1) = f_1(f_2(n))$ is computed by DO X; f_1; X ← Y; END . if $x > 0$ this is equivalent to DO; f_1; X ← Y; TEST X . So we need only consider the case when $x = 0$. Then the value of the output, y , should be 0, not $f_1(x)$.

To arrange this we use the function $sg(x) = \text{if } x > 0$ then 1 else 0 (for reference we mention its companion $\overline{sg}(x) = \text{if } x > 0$ then 0 else 1).

Then we calculate $sg(x) \cdot \hat{f}(x)$ where $\hat{f}(x)$ is the function defined by DO; f_1; TEST X . The product, $x \cdot y$, is calculated in LPL as

```
Z ← 0;
Y ← Y + 1;
DO;
Z ← Z + X;
TEST Y;
Z ← Z ∸ X
```

We did not need the function $\overline{sg}(\ )$ here, but to conclude the proof we show how to calculate both $sg(\ )$ and $\overline{sg}(\ )$.

Program for $sg(\ )$	Program for $\overline{sg}(\ )$
W ← 0;	W ← 0;
X ← X+1;	W ← W+1;
DO;	X ← X+1;
Z ← Z∸1;	DO;
Z ← W;	Z ← W;
W ← W∸1;	W ← W∸1;
W ← W+1;	TEST X;
TEST X;	X ← X∸1
X ← X∸1	
input is X, output is Z.	input is X, output is Z.

Q.E.D.

The above theorem shows how to translate a set of function defining equations into programs. It is interesting to consider the reverse process. This topic will be discussed in detail in §5. For now, we concentrate on one aspect.

Consider the simple scheme A:

<u>scheme A</u>

DO Z;

$Y \leftarrow f_1(X);$

$X \leftarrow f_2(Y);$

END

Taking X as the output variable in scheme A, the function being computed is the iteration of $f_2(f_1(x))$. If we use the notation $f^{(x)}(\ )$ for self-composition x-times, where $f^{(0)}(y) = y$ and $f^{(x+1)}(y) = f(f^{(x)}(y))$, then the mathematical expression corresponding to the scheme A with inputs X, Z and output X is $f^{(z)}(x)$ where $f(x) = f_2(f_1(x))$. Note that we use the corresponding lower case variables to denote the mathematical variables representing X, Y and Z.

If we attempt to write a similar function definition for this scheme with Y as output a difficulty arises; the final value of Y depends on a value of X computed during the previous iteration. So the best type of function expression we can write is $f_1(f^{(z-1)}(x))$ where $f(x) = f_2(f_1(x))$. Recalling that $f^{(0)}(f_1(x)) = f_1(x)$, we have a valid expression for Y , but because the <u>form of the expression changes</u> (<u>in general</u>, <u>possibly in a drastic manner</u>) depending on whether $|z| = 1$ or $|z| > 1$.

Another example of this lack of "form invariance" is the following

scheme B

```
Y ← f0(X);
DO Z;
Y ← f1(X);
X ← f2(Y);
END
```

When $|Z| = 0$, the form $f_1(f^{(z-1)}(x))$ for the value of Y , where $f(x) = f_2(f_1(x))$, is not correct, since then it computes $f_0(x)$.

Another example for PL is

scheme C

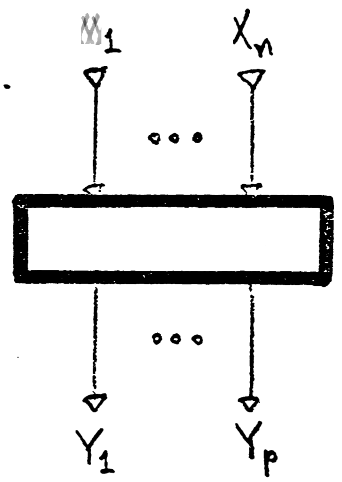
```
DO;
X ← F1(Y);
Y ← F2(Y);
TEST Z
```

If $|Z| = 0$ or 1, then the value of X is $F_1(Y)$ but if $|Z| = 2$, then the value of X is $F_1(F_2(Y))$ (in general if $|Z| = n > 0$, then X is $F_1(F_2^{n-1}(Y))$). Thus the <u>form</u> of <u>X</u> <u>changes</u> as a function of Z . Notice that the form of the value of Y does not change. It is always $F_2^n(Y)$. Thus with respect to Y , the scheme is "form invariant".

The informal idea of "form invariance" is relative to the mathematical notation being used. We are thinking of invariance with respect to "iterative vector function forms". Since such forms are amenable to analysis, we would like to first restrict our attention to them and produce an equivalence algorithm. We do this in §6 for a certain type of form invariant schemes which we call <u>linear schemes</u>.

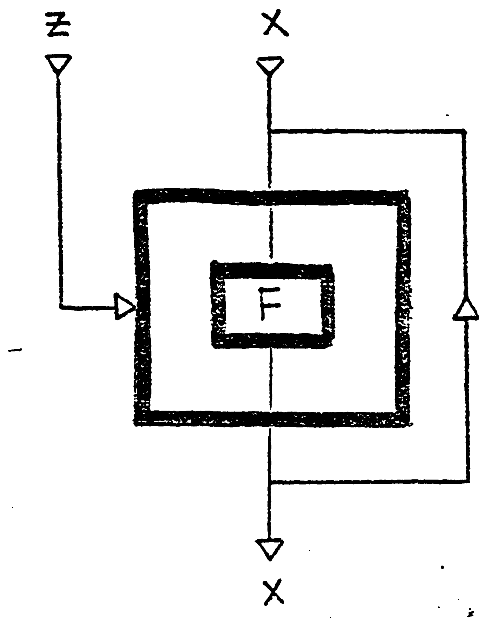
The term "linear" is derived from a geometric view of schemes and it is worth some time to convey this idea informally before we offer a precise formal definition of linear scheme (in Definition 3 below).

A typical (and in a sense canonical) non-linear scheme is scheme C above. We can draw a "circuit diagram" which represents that scheme. Let



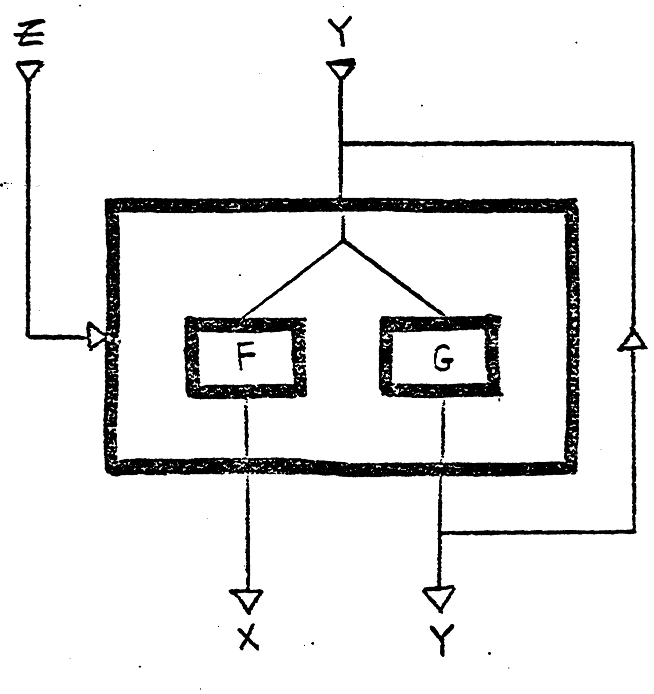
be a module with input lines $X_1,\ldots,X_n$ and outputs $Y_1,\ldots,Y_p$. We can combine such modules into networks in the usual fashion, and we can <u>iterate</u> a network.

To iterate a network, draw a box around it, draw a line into the side of the box representing the iteration variable and draw feedback connections between variables. Thus



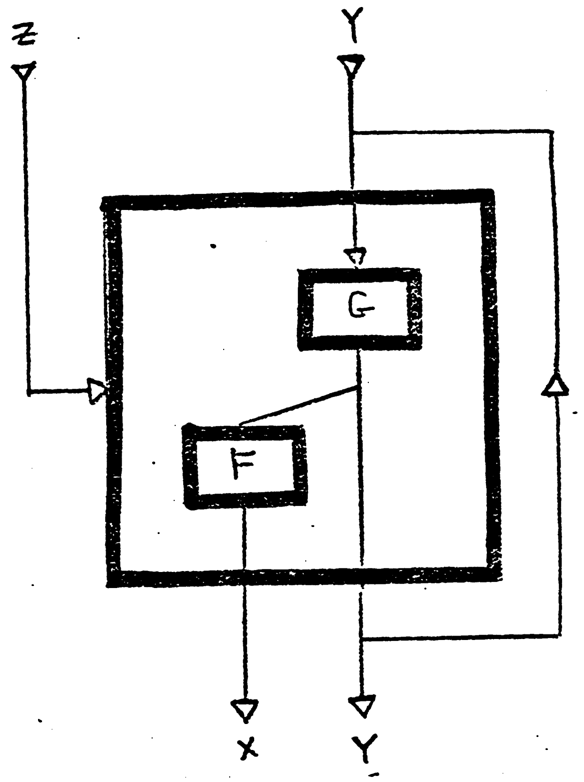
represents DO; X ← F(X); TEST Z.

The non-linear scheme above has the diagram



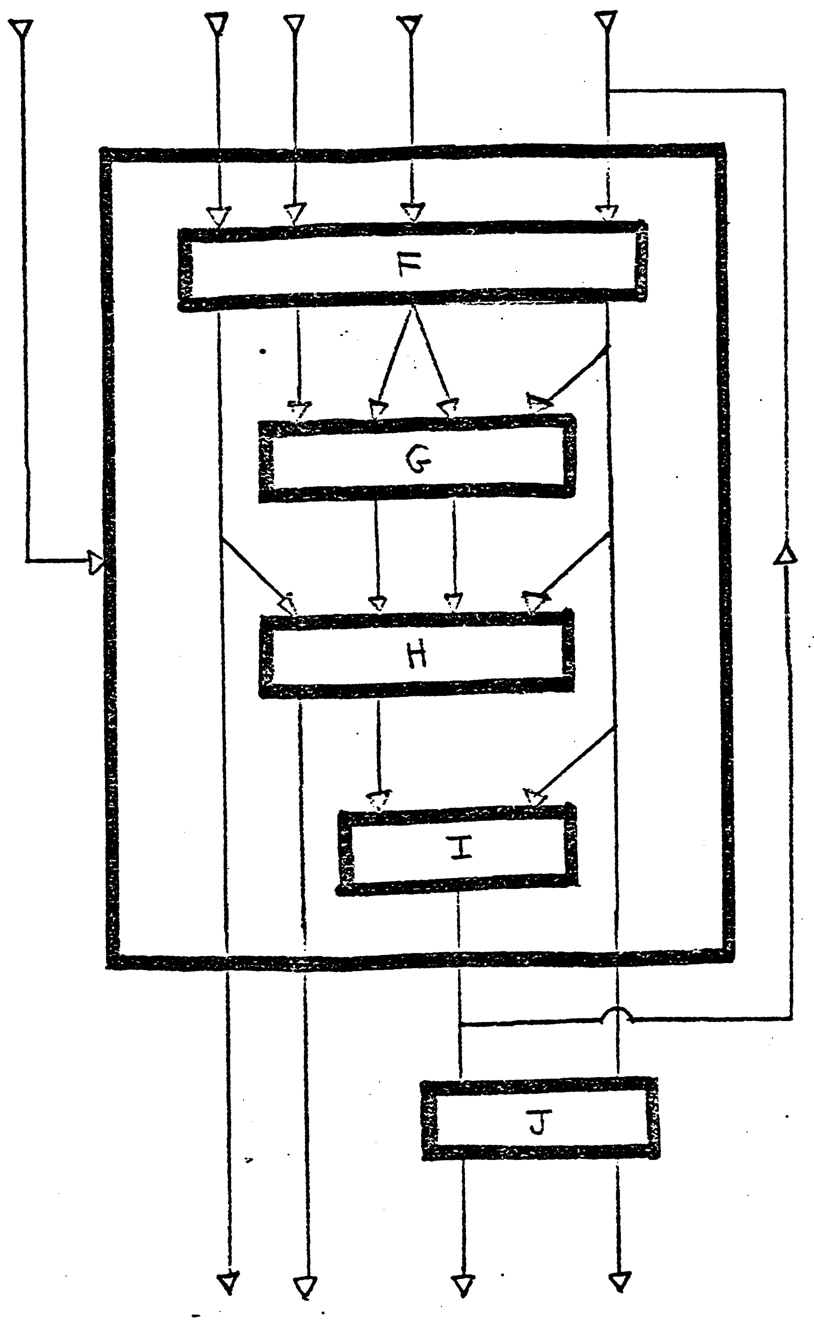
We say that this scheme is <u>non-linear in X</u>, hence non-linear, because in tracing back from X we arrive at a branch point at which it is possible for the F module to receive inputs either of the form Y or $G^{(n)}(Y)$.

By simply changing the order of the assignments, we arrive at the scheme DO; Y ← G(Y), X ← F(Y); TEST Z whose diagram is



This scheme is linear.

Pictorially a scheme will be linear whenever there is only one feedback variable and there are no modules which are not "directly in line", i.e. when there are no modules side by side. Below is a diagram for a linear scheme.



As preparation for a precise definition of linear schemes we want to treat all scheme statements uniformly as a type of assignment representing a function computation. Then each scheme is simply a sequence of generalized assignments. We need only reinterpret the iterative statement as an assignment.

Any iterative statement can be regarded as computing a vector-valued function, so let

$$\langle Y_1,\ldots,Y_p\rangle \leftarrow H(X_1,\ldots,X_n)$$

represent an iterative statement with outputs $Y_1,\ldots,Y_p$ and inputs $X_1,\ldots,X_n$. We think of one execution of that statement as making an assignment to $Y_1,\ldots,Y_p$. For our purposes, we can regard the assignment to the Y_i's as occurring simultaneously, but in fact we order them in the vector, from left to right, in order of assignment. For example,

DO; X ← F(X); TEST Z

becomes

$$\langle Z,X\rangle \leftarrow H(Z,X) \ .$$

Now any loop scheme can be regarded as a sequence of generalized assignments, say

$$\langle Y_{11},\ldots,Y_{1p_i}\rangle \leftarrow H_1(X_{11},\ldots,X_{1n_1});$$

$$\langle Y_{21},\ldots,Y_{2p_2}\rangle \leftarrow H_2(X_{21},\ldots,X_{2n_2});$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$\langle Y_{m_1},\ldots,Y_{mp_m}\rangle \leftarrow H_m(X_{m_1},\ldots,X_{mn_m})$$

To say when a scheme is linear, we need the concept of one variable influencing another.

We build up the definition of influence and direct influence inductively on the depth of nesting. At the lowest depth, if $Y \leftarrow F(X)$, we say that X directly influences Y . Given DO; B; TEST X or DO X; B; END, if there is a variable Y in B which influences itself (i.e. is a feedback variable), then we say that X directly influences Y at this depth. Given the sequence of generalized assignments above, X_{ij} will influence some Y_{ik} . If X_{ij} also occurs on the left hand side, say as $Y_{k\ell}$, then there are variables, say X_{kq} influencing X_{ij} (so if an input is also an output at some line, then it too is influenced). If $k < i$, then Y_{ik} is also influenced by X_{kq} , but even if $k > i$, we say that X_{kq} <u>can influence</u> Y_{ik} (because if the scheme is put into a loop, then influence can spread backward).

For example, in

$$X_1 \leftarrow F_1(Y);$$
$$X_2 \leftarrow F_2(X_1);$$
$$Y \leftarrow F_2(Y)$$

Y directly influences X_1 and influences X_2 .

We now define the crucial concept of a linear scheme.

Definition 3: A scheme S is linear iff for any variable Y no variable influencing Y occurs on the left hand side of a generalized assignment below the last occurrence of Y on the left, and only one variable Y influences itself and each generalized assignment (i.e. each H_i) within S is itself linear.[†]

Examples:

(1)
```
DO;
    X2 ← F1(X1);
    X3 ← F2(X2);
    X1 ← F1(X1);
    Y  ← F2(X1);
TEST Z
```

This scheme is linear.

(2)
```
DO;
    DO;
    Z ← F1(Z);
    TEST X;
X ← F2(X);
TEST Z
```

[†] This means that in any DO,TEST block there is only one feedback variable at the outermost level, but each generalized assignment may have its own feedback variable and so on down into the levels of nesting.

This can be written as

```
DO;
<X,Z> ← H(X,Z);
X ← F2(X);
TEST Z
```

and it is non-linear because both Z and X are feedback variables (each influences itself).

A more complex example of the same sort is

(3)
```
DO ;
  DO ;
  Z ← F1(Z);
  TEST X;
  DO;
  X ← F2(X);
  TEST Z;
TEST Y
```

We now show that the classes of linear schemes, LL for <u>linear</u> <u>Loop</u> and LPL for <u>linear</u> <u>Post-Loop</u>, are adequate to generate $\mathcal{R}^1$.

<u>Theorem 2</u>: LL$[+1, \dot{-}1]$ and LPL$[+1, \dot{-}1$, quadres( ), sg( )$]$ compute $\mathcal{R}^1$.

<u>Proof</u>: We again rely on a result by Péter, namely that $\mathcal{R}^1$ can be generated from the unary functions $n+1$ and quadres(n) using the operations of substitution, addition, and pure

iteration. Now, instead of using $x+y$ as a basic function, we are only free to use the operation $+$ in forming a function $f(\ )$ as in $f(n) = f_1(n) + f_2(n)$.

We must show that if a function is generated using the basic operations, then it can be done in $LL[+1, \dot{-}1]$ and in $LPL[+1, \dot{-}, \text{quadres}(\ ), \text{sg}(\ )]$. (The reason we need $\dot{-}$ in LL is that the schemes for subtraction in L are not linear.) The only questionable point might be the addition operator, but the reader can check that the schemes given in Theorem 1 for addition are linear.

Q. E. D.

(2.8) Equivalence Problems

Given a class of programs over D , say $C = \{\phi_i\}$, the equivalence problem (for unary programs*) is

$$\phi_i(x) \simeq \phi_j(x) \text{ for all } x \in D, \text{ abbreviated } \phi_i \equiv \phi_j ,$$

where $\phi_i(x) \simeq \phi_j(x)$ means $\phi_i(x)\downarrow$ iff $\phi_j(x)\downarrow$ and $\phi_j(x)\downarrow$ implies $\phi_i(x) = \phi_j(x)$ (where $\phi_i(x)\downarrow$ is the common abbreviation for ϕ_i halts on input x).

Given a class of schemes (over D , unary in each input*) $C = \{\Phi_i\}$ two schemes are equivalent, written

* We consider the unary case only for simplicity of notation. The definitions are identical for general domains.

$\Phi_i \equiv \Phi_j$, iff

$$** \quad \Phi_i(f,p,x) \simeq \Phi_j(f,p,x)$$

for all $f \in \mathcal{F}^n(d)$, $p \in \mathcal{P}^m(d)$ and $x \in D$.

Clearly if $\Phi_i \equiv \Phi_j$, then $\phi_i \equiv \phi_j$ for any programs derived from the schemes by interpreting the function and predicate variables.

The following facts are well known.

Fact 2: The equivalence problem for $G_3 = P[+1, \dot{-}1, \neq 0]$ is recursively unsolvable.

Remark: The halting problem, does $\phi_i(i)\downarrow$, is easily reducible to this equivalence problem, which establishes the fact.

Fact 3: The equivalence problem for Loop = L[+1, 0] is recursively unsolvable.

Remark: This is proved by showing that for any $G_3 = P[+1, \dot{-}1, \neq 0]$ program ϕ_i , there is a Loop program α_i , which on input x simulates $\phi_i(i)$ for x steps and, if $\phi_i(i)\downarrow$ in that time, outputs a 0, otherwise outputs a 1. Now if it could be decided whether $\alpha_i(x) = 1$ for all x , the halting problem for G_3 programs could be solved.

In brief, two schemes are equivalent iff they compute the same functional. The following example gives two equivalent L schemes.

Example

$$
\begin{array}{ll}
S_1 & S_2 \\
V_1 \leftarrow F_1(V_1); & DO\ V_2; \\
DO\ V_2; & V_3 \leftarrow F_2(V_4); \\
V_3 \leftarrow F_1(V_2); & V_3 \leftarrow F_2(V_2); \\
END & V_3 \leftarrow F_1(V_2); \\
 & END; \\
 & V_1 \leftarrow F_1(V_1)
\end{array}
$$

Both schemes compute the same functional

$$\mathcal{J}(D)_1 \times D^2 \to D^3$$

although S_2 appears to have both $f_2(\ )$ and V_4 as additional inputs. The functional can be described mathematically as

$$S[f_1, v_1, v_2] = \langle f_1(v_1),\ v_2,\ f_1^{(v_2)}\ (v_2)\rangle$$

§3. Conditional Loop Schemata Equivalence

(3.1) The Language $L_\Diamond$:

We now consider the conditional Loop schemata language $L_\Diamond$ and show that its equivalence problem is undecidable. In the following section we show how to translate a conditional Loop scheme into an equivalent Loop scheme, and thus show the undecidability of the equivalence problem for Loop schemata in general. We choose this method of presentation because the downward transfer, conditional, and halt statements of $L_\diamond$ make the schemes easier to write and understand.

The conditional Loop schemata language $L_\diamond$ is defined in (2.3) (iii).

(3.2) <u>One-way Two-headed Finite Automata</u>:

Rosenberg [15] and Luckham, Park, and Paterson [8] have proved a number of undecidability results about one-way two-headed finite automata and the latter authors have demontrated a sense in which flowchart schemata simulate these automata.

We shall provide here only a broad outline of the functioning of these automata and an example and refer the interested reader to [8] for the details of their operation. A one-way two-headed finite automaton A is a 6-tuple (Σ, Q, T, q_0, Q′, $\underline{a}$) where

$\Sigma = \{b_1, b_2, \ldots, b_n\}$ is a finite input alphabet

Q is a set of states which is partitioned into

$Q' = \{q_0, q_1, \ldots, q_m\}$ the live states, with $q_0 \varepsilon Q$ the distinguished <u>initial</u> state, and $Q-Q'$ the <u>dead</u> states, with $\underline{a} \varepsilon Q-Q'$ the <u>accept</u> state. Q' is partitioned into disjoint subsets Q_1, Q_2, one for each reading head, and for each $q_i \varepsilon Q_j$ we write q_i^j to

indicate that A reads from head j in state q_i .

T is a transition table, as described below. The rows of T have the form

$$q_i^j : s_1 \to r_{i1}, \ldots, s_t \to r_{it}$$

specifying that the next state is r_{ik} if the symbol s_k is read by head j in state q_i . The device operates by moving its head, reading a symbol, and then changing state. On reaching a dead state the automaton stops (and accepts if the state reached is $\underline{a}$).

As an example, we exhibit a one-way two-headed finite automaton A which accepts the context-free language $\{0^n 1^n 0 (0 \vee 1)^* | n \geq 1\}$:

$$A = (\{0, 1\}, \{q_0, q_1, q_2, q_3, q_4, \underline{a}, \underline{r}\}, T,$$
$$q_0, \{q_0, q_1, q_2, q_3, q_4\}, \underline{a})$$

where T is given by

$$q_0^2: \ 0 \to q_1, \ 1 \to \underline{r}$$
$$q_1^2: \ 0 \to q_1, \ 1 \to q_2$$
$$q_2^1: \ 0 \to q_3, \ 1 \to \underline{r}$$
$$q_3^2: \ 0 \to q_4, \ 1 \to q_2$$
$$q_4^2: \ 0 \to \underline{r}, \ 1 \to \underline{a}$$

The machine operates by scanning across the tape with head two until it encounters a "1" and then alternately moving heads one and two to match each "0" and "1", halting when the first head encounters a "1" and the second another "0".

The basic undecidability result given by Luckham, Park, and Paterson (their Theorem 3.1) is that it is recursively undecidable whether a one-way two-headed finite automaton accepts any tapes at all (i.e. the <u>emptiness problem</u> is recursively unsolvable).

(3.3) <u>Simulation of One-way Two-headed Automata by Flowchart Schemes</u>

Luckham, Park, and Paterson [8] show that a flowchart scheme can simulate a one-way two-headed finite automaton with a binary alphabet in that if the computation of the automaton B is given by the state-symbol sequence

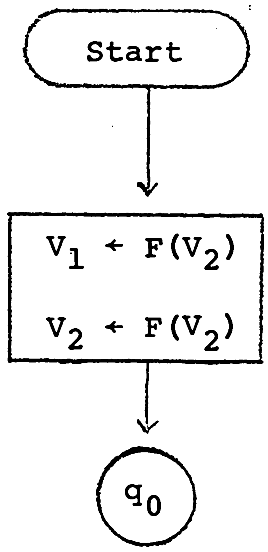
$$q_0 \varepsilon_1 q_{i_1} \varepsilon_2 q_{i_2} \varepsilon_3 \ldots,$$

then the execution sequence of the scheme P(B) is given by the instructions corresponding to the sequence of states

$$q_0,\ q_{i_1},\ q_{i_2}, \ldots,$$

and vice versa. The simulator P(B) is obtained from the transition table of B as follows:

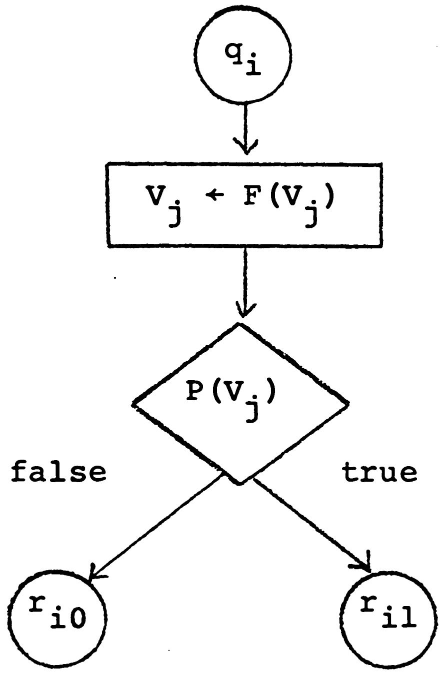
1. Construct the scheme,



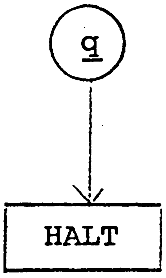
2. For each row of the transition table

q_i^j: $0 \rightarrow r_{i0}$, $1 \rightarrow r_{i1}$ $\quad 0 \leq i \leq n$

construct the scheme

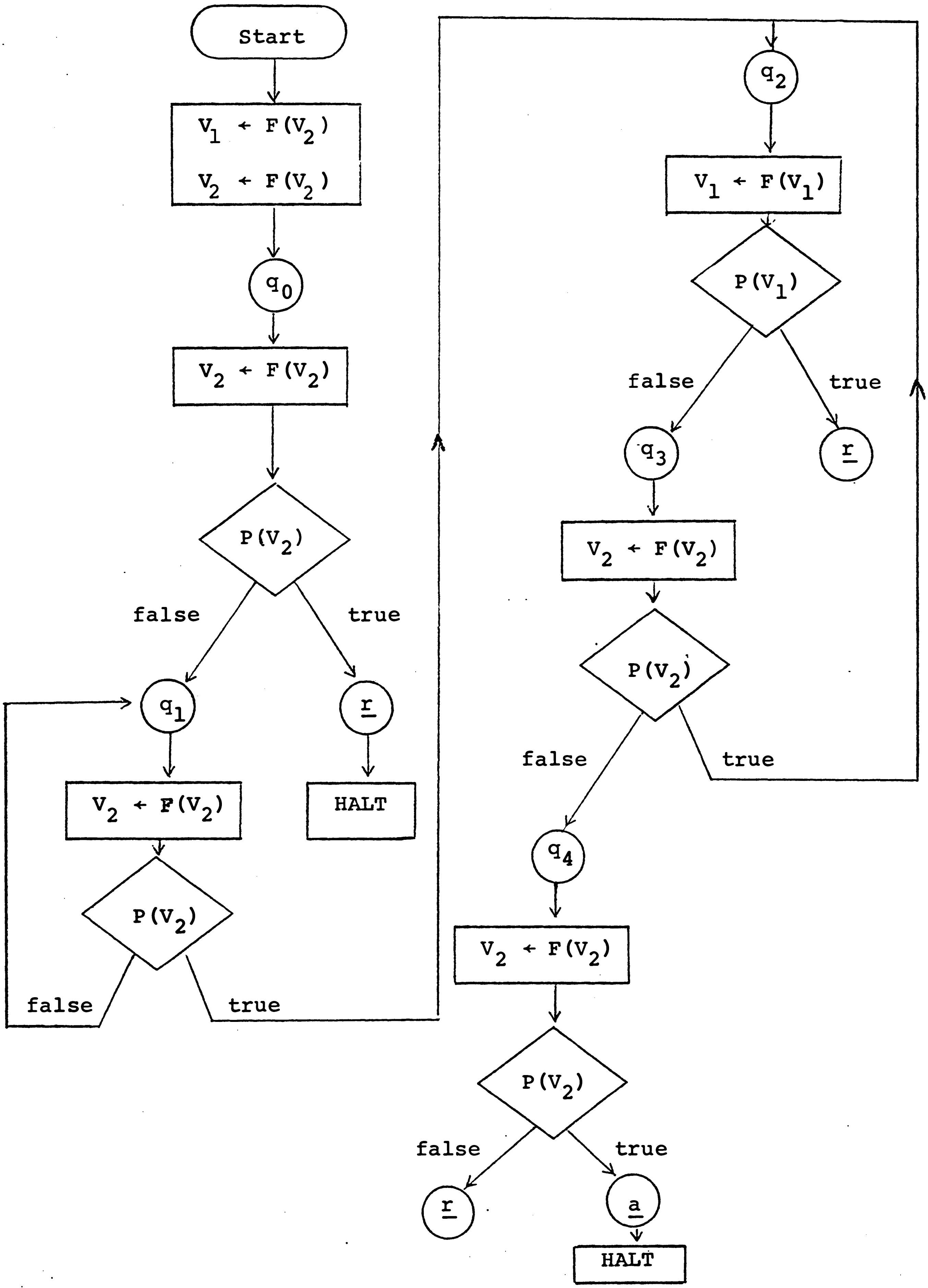


3. For each dead state $\underline{q}$, construct the scheme



4. Connect the above n+2 schemes by identifying labels, thus obtaining P(B).

As an example of a simulator we offer the following flowchart scheme which simulates the two-headed automaton given above to recognize $\{0^n 1^n 0(0 \vee 1)^* | n \geq 1\}$.


Start
$V_1 \leftarrow F(V_2)$
$V_2 \leftarrow F(V_2)$
q_0
$V_2 \leftarrow F(V_2)$
$P(V_2)$
false
true
q_1
r
$V_2 \leftarrow F(V_2)$
HALT
$P(V_2)$
false
true
q_2
$V_1 \leftarrow F(V_1)$
$P(V_1)$
false
true
q_3
r
$V_2 \leftarrow F(V_2)$
$P(V_2)$
false
true
q_4
$V_2 \leftarrow F(V_2)$
$P(V_2)$
false
true
r
a
HALT

write

```
ℓ:  IF P(V_i)  THEN ℓ';
    GO TO m;
ℓ': GO TO n
```

Now for each upward transfer, replace

GO TO $-R_i$

by

$S_1 \leftarrow X_2$;

...

$S_t \leftarrow X_2$;

$S_i \leftarrow X_1$;

GO TO end

and surround the translated code with

$S_1 \leftarrow X_2$;

...

$S_t \leftarrow X_2$;

DO X;

IF $P(S_1)$ THEN R_1;

IF $P(S_2)$ THEN R_2;

...

IF $P(S_{t-1})$ THEN R_{t-1};

IF $P(S_t)$ THEN R_t ELSE Start;

translated code described above

```
end:  END;
      HALT
```

to create the $L_\Diamond$ simulator S' .

Thus at each place in which S would have executed an upward transfer, say to R_i , we now set S_i to X_1 and all other S_j to X_2 and transfer to the END statement of the containing DO X loop. If X iterations of this loop have not yet been exhausted, we then evaluate $P(S_1)$, $P(S_2)$,... finding them all <u>false</u> until we reach $P(S_i)$ = <u>true</u> and transfer to R_i as intended. This $L_\Diamond$ scheme thus executes X upward transfers and hence at least X execution steps of the flowchart scheme S .

Q.E.D.

We now apply lemma 1 in proving that equivalence of $L_\Diamond$ schemes is recursively undecidable by reducing to the problem the known undecidable problem of testing emptiness of the set of tapes accepted by a two-headed one-way finite automaton.

<u>Theorem 3</u>: The $L_\Diamond$ equivalence problem (with pure data transfers) is recursively unsolvable.

<u>Proof</u>: Suppose the contrary. We will then show that the emptiness problem for two-headed finite automata is solvable. We are given a one-way two-headed finite automaton B , for which we construct the flowchart scheme $P(B)$ using predicate P and function F .

Consider the following scheme SP(B) , where we place the instruction

OUT ← F_2(IN)

before the HALT representing the accept state of the finite automaton and OUT ← IN before all the other halts:

```
        OUT ← IN;
        V2 ← V1;
        X1 ← V1;
        X2 ← V1;
        DO   IN;
             IF P(V1) THEN L1 ELSE L2;
L1:          X1 ← V1;
             GO TO +L3;
L2:          X2 ← V1;
L3:          V1 ← F(V1);
             IF P(X1) THEN L4 ELSE L5;
L4:          IF P(X2) THEN L5 ELSE L6;
L5:     END;
        OUT ← F1(IN);
        HALT;
```

Part 1: initialize variables

Part 2: locate i, j such that $F^i(0)$ = <u>true</u> and $F^j(0)$ = <u>false</u>, if they exist

```
L6: S1 ← X2;
      ...
    St ← X2;
    DO   X;
         IF  P(S1) THEN R1;
         IF  P(S2) THEN R2;
               ...
         IF  P(St-1) THEN Rt-1;
         IF  P(St) THEN  Rt ELSE Start;
       ┌──────────────────────────────┐
       │ translated code              │
       │ from Lemma 1 with            │
       │ OUT ← F2(IN)                 │
       │ before the HALT repre-       │
       │ senting the accept           │
       │ state of the automaton       │
       │ and OUT ← IN  before all     │
       │ the other HALT's             │
       └──────────────────────────────┘
end: END;
     OUT ← IN;
     HALT
```

Part 3: two-headed one-way finite automaton simulator

This scheme SP(B) first attempts to locate $X_1 = F^i(0)$, $X_2 = F^j(0)$ with $P(X_1) =$ true and $P(X_2) =$ false and halts with $OUT = F_1(IN)$ if it does not find them in $0, F(0), \ldots, F^{IN}(0)$.*

* This method of using a locator program to find a binary valued predicate is used extensively in scheme theory. See [3] for a general account of it.

(3.4) Unsolvability of the Equivalence Problem (with pure data transfers)

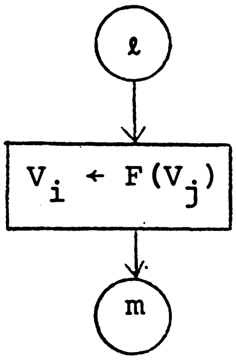
To prove the equivalence problem for $L_\Diamond$ schemata with pure data transfers unsolvable an indirect simulation of a two-headed automaton is used. We first simulate the automaton B with a flowchart scheme P(B) and then simulate P(B) with an $L_\Diamond$ scheme SP(B) which, on input X , simulates at least X execution steps of P(B) . The following lemma provides the core of the simulator SP(B) .

Lemma 1: We can construct for any flowchart scheme S an $L_\Diamond$ scheme S' with the following property:

> if P, X_1, X_2 satisfy $P(X_1)$ = true and $P(X_2)$ = false then, on input X, S' simulates at least X execution steps of S .

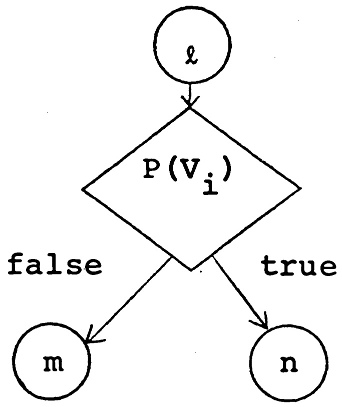
Proof: For each block of the flowchart S write a segment of $L_\Diamond$ code as follows:

(i) for



write ℓ: $V_i \leftarrow F(V_j)$;
GO TO m

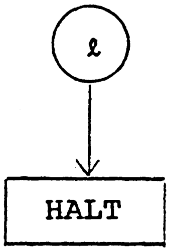
(ii) for



write

ℓ: IF $P(V_i)$ THEN n ELSE m

(iii) for



write

ℓ: HALT

Arrange these segments in any order. Some of the GO TO's will refer to statements which follow them ("downward") and others to ones which precede them ("upward"). Leave the downward transfers as they are. Suppose the scheme contains t upward transfers with targets $R_1, R_2, \ldots, R_t$. If some of these transfers are in IF statements rewrite them as follows:

for

ℓ: IF $P(V_i)$ THEN n ELSE m

If it does find X_1, X_2 with this property it then simulates the two-headed finite automaton B for at least IN execution steps. If the automaton has halted and accepted in the process it sets OUT = F_2(IN), otherwise OUT = IN .

Note that the sequence $0, F(0), F^2(0), \ldots$ corresponds to the contents of the tape cells, and P corresponds to the test to determine whether the contents of a tape cell is 0 or 1, so that Part 2 of SP(B) corresponds to checking the contents of the first IN tape cells to determine whether they all contain the same value or not (and noting at least one pair of cells which contain different values if any exists).

Let S_1(B) denote a scheme identical to SP(B) except that the one assignment

OUT $\leftarrow$ F_2(IN)

is replaced by

OUT $\leftarrow$ IN .

Now suppose that equivalence of $L_\diamond$ schemata were decidable. If SP(B) $\not\equiv$ S_1(B), then B must accept some tape. If SP(B) $\equiv$ S_1(B), then B does not accept any tapes, except perhaps those which consist entirely of 0's or entirely of 1's. These we may check for separately since in this case B can only make a number of moves equal to the number of its state transitions before it must either halt or enter a nonterminating loop. Thus the emptiness problem for one-way two-headed finite automata is reducible to the $L_\diamond$ equivalence problem and since the former problem is undecidable, so is the latter.

Q.E.D.

(3.5) Unsolvability of the equivalence problem (without pure data transfers)

Unsolvability of equivalence of $L_{\Diamond}$ schemata without pure data transfers is proved by simple modifications of Lemma 1 and Theorem 3, as follows:

Instead of the code substituted for

GO TO $-R_i$

in Lemma 1, use

$S_1 \leftarrow F(X_2);$

...

$S_t \leftarrow F(X_2);$

$S_i \leftarrow F(X_1)$;

GO TO end

We then have

<u>Theorem 4</u>: The $L_\Diamond$ equivalence problem (without pure data transfers) is recursively unsolvable.

<u>Proof</u>: Replace the scheme SP(B) of Theorem 3 with SP'(B) given by

$OUT \leftarrow F_1(IN)$;

$V_2 \leftarrow F(V_1)$;

$X_1 \leftarrow F(V_1)$;

$X_2 \leftarrow F(V_1)$;

DO IN;

  $V_3 \leftarrow F(V_1)$;

  $V_3 \leftarrow F(V_3)$;

  IF $P(V_3)$ THEN L_1 ELSE L_2;

L_1: $X_1 \leftarrow F(V_1)$;

  GO TO $+L_3$;

L_2: $X_2 \leftarrow F(V_1)$;

L_3: $V_1 \leftarrow F(V_1)$;

  $V_3 \leftarrow F(X_1)$;

  IF $P(V_3)$ THEN L_4 ELSE L_5;

L_4: $V_3 \leftarrow F(X_2)$;

  IF $P(V_3)$ THEN L_5 ELSE L_6;

```
L5:       END;
          OUT ← F1(IN);
          OUT ← F1(OUT);
          HALT;



L6:       S1 ← F(X2);
            ...
          St ← F(X2);
          DO   X;
               IF  P(S1) THEN R1;
               IF  P(S2) THEN R2;
                 ...
               IF  P(St-1) THEN Rt-1;
               IF  P(St) THEN Rt ELSE Start;

             +------------+
             | translated |
             | simulator  |
             | code       |
             +------------+

  end:    END;
          OUT ← F1(IN);
          HALT
```

The remainder of the proof is analogous to that of Theorem 3, except that we miss not only the tapes with all 0's or all 1's, but also those with a single 1 followed by all 0's and a single 0 followed by all 1's.

Q.E.D.

(3.6) Unsolvability of the Equivalence Problem for $PL_\lozenge$

Techniques similar to those used in (3.4) and (3.5) establish the unsolvability of the equivalence problem for PL . However, we cannot use the code DO V_2; OUT ← F_1(IN); HALT; END which causes OUT ← F_1(IN) to be executed only when $V_2 \neq 0$ to insure that each simulation of automaton B starts at the same tape cell. Instead, we use another device which causes OUT ← F_1(IN) to be executed only when $V_2 \geq 2$. The mechanism for this is given in (4.4) and results in our simulating the automata beginning at either tape cell 0 or 1. The reader will see that essentially the same proofs as those given for Theorems 3 and 4 will establish

Theorem 5. The equivalence problem for $PL_\lozenge$ with or without pure data transfers is unsolvable.

§4. Loop Schemata Equivalence

(4.1) Translation of $L_\lozenge$ into L :

To show that equivalence of Loop schemata is undecidable we exhibit a translation from $L_\lozenge$ to L , i.e. an effective procedure which constructs an equivalent L scheme for a given one in $L_\lozenge$. Our method will be to reduce all conditional, halt, and transfer schemes to the form

IF P(X) THEN L

and then to transform these simple conditional schemes into iterative schemes which control sequencing through the entire scheme in the same way.

To simplify the conditional schemata, notice that

(i) IF P(X) THEN L_1 ELSE L_2

is equivalent to

IF P(X) THEN L_1;

GO TO L_2

(ii) if $P^*(X^*)$ is guaranteed to have the value <u>true</u>, then

GO TO L_1

is equivalent to

IF $P^*(X^*)$ THEN L_1,

and (iii) if again $P^*(X^*)$ is <u>true</u> then HALT is equivalent to

IF $P^*(X^*)$ THEN L^*

where L^* labels a dummy statement appended to the end of the scheme.

(4.2) <u>Elimination of Conditionals</u>:

Suppose $\Pi \in L_\Diamond$ and let Π_0 be the result of modifying Π by the procedures given in (4.1), so that it contains no two-branched conditionals, HALT's, or GO TO's. Note that if $P^*(X^*)$ = <u>true</u> then Π and Π_0 are equivalent. Suppose further that Π_0 has m simple conditionals

L_i: IF P(V) THEN M_i

ordered as $L_1, L_2, \ldots, L_m$ in the scheme. The following procedure applied successively m times produces programs $\Pi_1, \Pi_2, \ldots, \Pi_m$ which (we claim) are all equivalent to Π_0 (if only $Z = 0$ and $IN = 0$) and such that $\Pi_m \in L$.

1. Replace the statement at L_i by

   ```
   V′ ← χP(V);
   DO V′;
      Hi ← Z;
   END
   ```

where χ_P is a characteristic function for the predicate P, i.e.

$$P(V) = \underline{\text{false}} \text{ iff } \chi_P(V) = 0$$

2. If the statement at M_i is

   M_i: S_i

   then replace it by

   M_i: $H_i \leftarrow IN$;
   S_i

3. Replace every statement between L_i and M_i (in the following section, where we use this translation process, L_i will always precede M_i)

as follows:

(i) if the statement is N_i: S_i where S_i is

```
    U ← W
 or U ← F(W)
 or IF P(U) THEN L
 or HALT
```

where U and W are different variables, then replace it by

```
N_i:  DO H_i;
          S_i;
          END
```

(ii) if the statement is N_i: $U \leftarrow F(U)$ (statements of the form $U \leftarrow U$ can obviously be deleted) then replace it by

```
N_i:  U' ← U;
      DO H_i;
      U ← F(U');
      END
```

(iii) leave all other statements as they are.

4. Place the statement $H_i \leftarrow IN$ at the beginning of the program.

This generates from Π_{i-1} a new $L_{\Diamond}$ scheme Π_i with m-i conditionals, so that ultimately $\Pi_m \in L$.

Intuitively, Π_{i-1} and Π_i are equivalent because H_i functions as a switch to determine whether instructions should be executed or skipped over. The code which replaces a branch sets the switch to zero if the branch were to be executed and non-zero otherwise. The code surrounding each statement from L_i to M_i causes it not to be executed if the branch was to be executed and to be executed if the branch was not to be executed. The code at M_i then turns the switch on to make sure that the preceding statements will be executed on later passes if, e.g., they are in the range of a DO.

We will not give a formal proof of the equivalence here, but refer the reader to [2], where an analogous translation is used to show that the subrecursive language SR is equivalent in expressive power to the Loop language and is proved valid.

(4.3) <u>Unsolvability of Loop and Linear Loop Schemata Equivalence:</u>

The critical element in our proof that the Loop Schemata equivalence problem is unsolvable will be to locate a variable with the value zero and another which is non-zero.

The second variable is easily obtained (we simply use an input, which will be non-zero except in one case which we will be able to ignore), but the first is not so simple. We can write a scheme which will determine whether a function ever assumes the value zero (assuming the equivalence problem solvable), but cannot actually localize that value. For example,

```
Z ← IN;
DO IN;
      Z′ ← Z;
      DO Z;
            Z ← F₁(Z′);
      END;
END;
Y ← IN;
Y′ ← IN;
DO Z;
     Y ← F₂(Y′);
END
```

is equivalent to the scheme

```
Z ← IN;
DO IN;
      Z′ ← Z;
      DO Z;
            Z ← F₁(Z′);
```

```
        END;
    END;
    Y ← IN;
    Y′ ← IN
```

iff for all values of IN , $F_1(IN) \neq 0$. But, we cannot guarantee that we can get a zero into Z for all values of IN.

Instead we shall construct a scheme with the general form S[IN, Z] given below

```
[initialize two-headed finite automaton simulator]
DO IN;
[simulate two-headed finite automaton operating for
 IN steps and assuming  Z = 0                      ]
END;
DO Z;
["kill off" the results of the previous loop if
 Z ≠ 0                                          ]
END
```

Thus S[IN, Z] will be trivial for $Z \neq 0$ and a two-headed finite automaton simulator for $Z = 0$.

Using the above ideas we now produce an L scheme LP(B) which behaves like the scheme SP(B) used in the proof of Theorem 3, and thus obtain recursive unsolvability for the L equivalence problem.

<u>Theorem 6</u>: The L equivalence problem (with pure data transfers) is recursively undecidable.

<u>Proof</u>: Suppose the contrary. We are given a one-way two-headed finite automaton B , for which we construct the flowchart scheme P(B) using predicate P and function F .

Construct an L scheme LP(B) as follows:

OUT $\leftarrow$ IN;
$H_4 \leftarrow$ IN;
$V_2 \leftarrow V_1$;
$V_3 \leftarrow V_1$;
$X_1 \leftarrow V_1$;
$X_2 \leftarrow V_1$;

Part 1: initialize variables

DO IN;
    $H_1 \leftarrow$ IN;
    $H_2 \leftarrow G(V_1)$;
    DO H_2;
        $X_1 \leftarrow V_1$;
        $H_1 \leftarrow V_2$;
    END;
    DO H_1;
        $X_2 \leftarrow V_1$;
    END;

Part 2: locate i, j such that $F^i(0)$ = <u>true</u> and $F^j(0)$ = <u>false</u>, if they exist

```
(**)        V1 ← F(V1);
            H3 ← G(X1);
            H4 ← V2;
            DO H3;
               H4 ← IN;
               H5 ← G(X2);
               DO H5;
                  H4 ← V2;
               END;
         END;
         OUT ← F1(IN);
         DO H4;
            OUT ← IN;
         END;

      END;
      DO H4;
```

Part 3 of SP(B) from the proof of Theorem 3 translated into L by the procedure of sections (4.1) and (4.2)

```
      END;
```

Part 3: two-headed one-way finite automaton simulator

```
DO V3;
   OUT ← F1(IN);
END
```

Part 4: "kill off" simulation results if original $V_1 \neq 0$

This scheme LP(B) is designed to operate just like SP(B) in Theorem 3. The function G used here is assumed to be a characteristic function for the predicate P, i.e.

$$P(X) = \underline{\text{false}} \text{ iff } G(X) = 0 \ .$$

Let $S_2(B)$ be the equivalent of $S_1(B)$ for LP(B). Suppose that equivalence of L schemata were decidable. If $LP(B) \not\equiv S_2(B)$, then B must accept some tape. If $LP(B) \equiv S_2(B)$, then B does not accept any tapes, except perhaps those which contain only all 0's or all 1's. We proceeed as in the proof for $L_{\diamond}$ equivalence.

Q.E.D.

A careful inspection of the code generated for Part 3 of the scheme LP(B) in the proof of Theorem 6 will show it to be simultaneously linear in V_1 and V_2. The code in Part 2 can be made (strictly) linear in V_1 simply by moving the assignment

(**) $V_1 \leftarrow F(V_1)$

upward to appear between the lines

```
DO IN;
  [V1 ← F(V1);]
  H1 ← IN;
```

With this change we have

Theorem 7: The simultaneously linear Loop schemata equivalence problem (with pure data transfers) is recursively unsolvable.

(4.4) Unsolvability of Pre-Loop Schemata Equivalence (without pure data transfers)

By considerably extending the locator and "killing off" techniques presented in the preceding sections and by modifying the procedure for translating from $L_{\diamond}$ into L, we may prove L equivalence without pure data transfers unsolvable.

The necessary changes to the translation procedure (see (4.2)) are

1. Replace the statement at L_i by

   ```
   V′ ← χP(V);
   DO V′;
      Hi ← G(V3);
   END
   ```

2. If the statement at M_i is M_i: S_i then replace it by

M_i: $H_i \leftarrow G(W)$;

S_i

3\. (ii) If the statement at N_i is N_i: $U \leftarrow F(U)$ then replace it by

N_i: DO H_i;

$U \leftarrow F(U)$;

END

4\. Place the statement $H_i \leftarrow G(W)$ at the Start statement of the simulator.

Now we may establish

<u>Theorem 8:</u> The L equivalence problem (without pure data transfers) is recursively unsolvable.

<u>Proof</u> We construct the scheme WP(B)

$\overline{OUT} \leftarrow F_1(IN);$ $OUT \leftarrow F_1(IN);$ $OUT \leftarrow F_1(OUT);$	Part 1: initialize OUT & $\overline{OUT}$

$V_3 \leftarrow F(Z);$
$V_2 \leftarrow F(Z);$
$V_1 \leftarrow G(V_3);$
DO IN;
   DO V_1;
      $V_3 \leftarrow F(V_2);$
   END;
   $V_2 \leftarrow F(V_2);$
   DO V_1;
      $V_1 \leftarrow G(V_2);$
   END;
END;
	Part 2:
	locate $i \geq 1 \ni$
	$\lvert GF^i(Z) \rvert = 0$
	and set
	$V_3 = F^i(Z)$

$D \leftarrow F(Z);$
$C \leftarrow F(Z);$
$A \leftarrow G(D);$
DO IN;
   DO A;
      $IN_2 \leftarrow G(V_3);$
   END;
   DO IN_2;
      $D \leftarrow F(C);$
   END;
	Part 3:
	locate $j \ni$
	$\lvert GF^j(Z) \rvert > 0$
	and set
	$D = F^j(Z)$

```
   C ← F(C);
   DO IN₂;
      A ← G(D);
   END;
END;
W  ← F(Z);
W3 ← F(Z);
W2 ← G(W3);
DO IN;
    W̄ ← G(V3);
    DO W2;
       W̄ ← G(D);
       DO V1;
          W̄ ← G(V3);
       END;
       V1 ← G(D);
    END;
    W̿ ← G(D);
    DO W̄;
       W̿ ← G(V3);
    END;
    DO W̿;
       W ← F(W3);
    END;
```

Part 4:
locate $k \geq 1 \ni$
$|GF^k(Z)| = 1$
and set
$W = F^k(Z)$

```
        W₃ ← F(W₃);
    DO  W̄;
        W₂ ← G(W);
    END;
    DO  V₁;
        V₁ ← G(V₃);
    END;
END;
```

```
DO  IN;
```

Part 3 of SP(B) from the proof of Theorem 3 translated into L by the procedures of sections (4.1) (4.2), and this section and with OUT ← F_2(IN) replaced by $\overline{\text{OUT}}$ ← F_2(IN)

```
END;
```

Part 5: one-way two-headed finite automaton simulator

```
OUT ← F₁(IN);
OUT ← F₁(OUT);
DO  A;
    E ← G(V₃);
    DO W₂;
        OUT ← F₁(OUT̄);
        DO E;
            OUT ← F₁(IN);
            OUT ← F₁(OUT);
        END;
        E ← G(D);
    END;
END;
```

Part 6:

"kill off"

if $|G(W)| \neq 1$

or $|G(D)| = 0$

```
V1 ← G(V3);                          Part 7:
DO   V1;                             "kill off"
     OUT ← F1(IN);                   if |G(V3)| ≠ 0
     OUT ← F1(OUT);
END;
DO   Z;                              Part 8:
     OUT ← F1(IN);                   "kill off"
     OUT ← F1(OUT);                  if |Z| ≠ 0
END
```

and a corresponding scheme $S_3(B)$ with $\overline{OUT} \leftarrow F_1(IN)$ for $\overline{OUT} \leftarrow F_2(IN)$ and proceed as in the proof of Theorem 6.

Q.E.D.

(4.5) Unsolvability of Non-linear Post-Loop Equivalence (with pure data transfers)

The methods of (4.2) and (4.3) apply to non-linear Post-loop because we can simulate branching with the following device. Let

$$\chi_P^1(X) = \begin{cases} 0 \text{ or } 1 & \text{if } P(X) \\ 2 & \text{if } \sim P(X) \end{cases}$$

be a modified characteristic function for P. Then to simulate the branch

IF P(V) THEN M

use the switching technique of (4.2) with the following code to set the switch, where $|Z_1| = 1$ and $|Z_2| = 2$.

$V \leftarrow \chi_P^1\ (V);$

DO;

$H \leftarrow Z_1;$

$Z_1 \leftarrow Z_2;$

TEST V

The initial values Z_1, Z_2 are taken as inputs, and the same "killing off" trick is used as that explained at the beginning of (4.3). The reader will note that $|H|$ takes the value 1 or 2 depending on whether $P(V)$ is <u>true</u> or <u>false</u>. Then a switch is constructed using H as before. For example, a switch after $U \leftarrow F(W)$ is

$U_1 \leftarrow U;$

$U \leftarrow F(W);$

$U_2 \leftarrow U;$

DO;

$U \leftarrow U_2;$

$U_2 \leftarrow U_1;$

TEST H

The rest of the construction is similar enough to that given in (4.3) that the reader can fill in the details, noting only that $|Z_1|$ and $|Z_2|$ must be arranged to contain 1 and 2 respectively by the "killing off" technique.

Theorem 9: The equivalence problem for PL (with pure transfers) is undecidable.

§5. Assigning Mathematical Notation to Schemes

(5.1) Function Names in Mathematics

The notation described in this section is equally valid for Pre- and Post-Loop programs and schemes, though, as the reader will note, most of the examples refer to Pre-Loop, since the following section provides numerous Post-Loop examples in our equivalence algorithm for Post-Loop schemata.

The standard mathematical notation for the Loop program DO X: X ← X+1; END is $2 \cdot x$, for DO X; DO X; X ← X+1; END; END is $2^x \cdot x$, and DO; X ← X+1; TEST X is $2 \cdot x + (1 \dot{-} x)$. If one is careful to distinguish function names (functors) from integer names, then the usual convention has $\lambda x[2 \cdot x]$ or $f_1(\ )$ with $f_1(x) = 2 \cdot x$ and $\lambda x[2^x \cdot x]$ or $f_2(\ )$ with $f_2(x) = 2^x \cdot x$ for the function names.

If the two Loop programs, α_1 and α_2, compute functions $\mathbb{N} \to \mathbb{N}$; and the output variable of α_1 is identical to the input variable of α_2, then the program $\bar{\alpha}_1$; $\bar{\alpha}_2$ is usually denoted by $\bar{\alpha}_2(\bar{\alpha}_1(\ ))$ where $\bar{\alpha}_1$, $\bar{\alpha}_2$ are the <u>functors</u> for α_1 and α_2. Thus DO X; X ← X+1; END; DO X; DO X; X ← X+1; END; END is denoted by $2^{(2x)} \cdot (2x)$ or by $f_2(f_1(\ ))$. We see that conjunction of programs corresponds to composition of functions.

The application of DO v; ——; END to a program α, resulting in DO v; α; 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 <u>iteration</u> by

$$\text{(i)}\quad f^{(0)}(x) = x$$
$$\text{(ii)}\quad f^{(n+1)}(x) = f(f^{(n)}(x))$$

Now it is easy to see that DO v; α; END is represented by $\alpha^{(x)}(\ )$ where $x = |v|$. Similarly, we define the <u>post-iteration</u> of $f(\ )$ by

$$\text{(i)}\quad f^{[0]}(x) = f^{[1]}(x) = f(x)$$
$$\text{(ii)}\quad f^{[n+1]}(x) = f(f^{[n]}(x)) \quad \text{for} \quad n \geq 1$$

so that DO; α; TEST v is represented by $\alpha^{[x]}(\ )$ for $x = |v|$. Note that $f^{(n)}(\ ) = f^{[n]}(\ )$ for $n \geq 1$.

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 an adequate notation for expressing the iteration of vector functions.

There is also a small matter of terminology. If programs compute <u>functions</u> which are denoted mathematically by <u>functors</u>, then what do we call the mathematical names for functionals, which program schemes compute? We shall call them functional expressions. An example of one is $H[f,x,y] = f^{(x)}(Y)$, the mathematical notation for

```
DO X; Y ← f(Y); END
```

Notation: Capital Latin letters, F_i, G_i, H_i denote functional expressions and $F_i[\ ]$, $G_i[\ ]$, $H_i[\ ]$ denote the corresponding functionals. To simplify the connection between schemes and their expressions we use variable and function names (x_i, f_i) for variables and function variables (v_i, h_i) in schemes.

(5.2) Iteration of Vector Functions:

A key piece of notation needed to describe non-linear schemes is that for the iteration of vector-valued functions and functionals. We need iteration only into individual arguments, so it suffices to treat only the case of functions.[4]

Notation for vectors is critical, so we consider it with care. A vector $\langle x_1,\ldots,x_n\rangle$ will be denoted by $\langle x\rangle$, or by X when no confusion results. The number of elements is indicated by writing $\langle x\rangle\varepsilon D^n$ or $X\varepsilon D^n$ in the context.

[4]Notation for the iteration of functions $f(\ ) : D^n \to D$ is difficult so one might expect that for $f(\ ) : D^n \to D^p$ it would be terrible. Happily, it is not. It is easier than the single output case, and in fact, clarifies that case.

The i^{th} component is denoted $<x>_i$ or X_i .

The 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 of individual-valued functions, $f_1(\ )$, with vector-valued functions, $<f_2(\ )>$, we can write $f_1(X)$ versus $<f_2(X)>$ or even $<f_2(<x>)>$. Now for iteration.

We are concerned only with iterations which occur in the form DO X; S; 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 the feedback built-in. This allows for a simple notation as we shall see below.

Given $f(\ ) : D^n \rightarrow D^p$, the required idea of iteration, $f^{(x)}(\ )$, is DO X; f; END where f has n inputs and p outputs.

We notice that unless some output is also an input, the above expression is just $f(\ )$ itself. In fact, the only outputs of consequence are those which are also inputs (<u>feed-back</u> 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 <u>implicit</u> output).

The definition follows:

(i) $f^{(0)}(X)_j = \begin{cases} X_j \text{ if } j \leq n \\ 0 \text{ ow} \end{cases}$

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

(iii) $f^{[0]}(X)_j = f^{(1)}(X)_j$

(iv) $f^{[n]}(X)_j = f^{(n)}(X)_j$ for $n \geq 1$

An example showing the connection between these iteration functional expressions and iterative schemes should be helpful.

<u>Example</u>: Consider the following scheme

```
DO X1;
Y ← F1(X1);      }
X1 ← F2(X2);     }  S
X2 ← F3(Y);      }
END
```

The scheme S inside the iterative 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 S 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

$$S^{(0)}(x_1,x_2) = \langle x_1, x_2, 0\rangle$$

$$S^{(n+1)}(x_1,x_2) =$$

$$\langle S(S^{(n)}(x_1,x_2)_1,S^{(n)}(x_1,x_2)_2)_1,S(S^{(n)}(x_1,x_2)_1,S^{(n)}(x_1,x_2)_2)_2, S(S^{(n)}(x_1,x_2)_1,S^{(n)}(x_1,x_2)_2)_3\rangle$$

The scheme $S^{(x_1)}(x_1,x_2)$ will be denoted

$$\langle f_2(x_2), f_3(f_1(x_1)), f_1(x_1)\rangle^{(x_1)}[x_1, x_2]$$

so that the < > part replaces the letter S and becomes the expression's main body. The ordering of the input variables along with the ordering of the output components determines the feedback for iteration. Therefore, this ordering information must be preserved when the functional expression is combined with others. For example, if X_1 and X_2 receive values $X_1 \leftarrow F_0(X_1)$ and $X_2 \leftarrow F_0(X_1)$ before the iterative, then the corresponding functional expression is $S^{(f_0(x_1))}(f_0(x_1), f_0(x_1))$ which will be denoted

$$\langle f_2(x_2),\ f_3(f_1(x_1)),\ f_1(x_1)\rangle^{(f_0(x_1))}_{\langle x_1,x_2\rangle}[f_0(x_1),\ f_0(x_1)]$$

We can avoid the subscripting n-tuple, in this example the $\langle x_1,x_2\rangle$, if we agree to number the input variables in order, $x_1,x_2,\ldots,x_n$ in the expression for the function body. This will be our most frequent convention. We will next discuss a uniform way of making these assignments.

(5.3) <u>Assigning Functional Expressions to Schemes</u>:

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 variables. A simple informal routine for this is given below. We continue to use the correspondence between F_i and f_i, X_i and x_i, Y_i and y_i

to separate the mathematical expressions from the schemes.

Given a scheme $S = s_1; s_2;\ldots;s_\ell$, the last occurrence of a variable v in S is its occurrence in s_m where $m = \max\{i | v \text{ occurs in } s_i\}$. Note that an iterative is a single statement in the BNF definitions in (2.1) - (2.3) and in this definition also.

We now give an algorithm for translating a scheme to a functional expression.

Routine A:

(1) Locate all output variables of S. 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) $Y_i \leftarrow F_j(W_i)$ outside the scope of all iteratives

(b) Y_i occurs on the lhs in the scope of an iterative, say in DO V; H; END.

Follow a separate procedure for each case:

(a)-procedure: Write $f_j(w_i)$ for y_i in the output vector.

(b)-procedure: Let s_m be the statement of last occurrence.

Select an H_ℓ, locate inputs and outputs to s_m , order them and write

$$H_{\ell<V_{i_1},\ldots,V_{i_q}>}^{(V)}(V_{i_1},\ldots,V_{i_q})_k$$

for y_i in the output vector (where the subscript k on the functional expression 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 (2). Continue in this way until only input variables, X_i to S 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 ℓ steps $(|S| = |s_1;\ldots;s_\ell| = \ell)$.

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

<u>Example 5.1</u>:

Scheme S

$V_1 \leftarrow F_1(V_1);$

DO $V_1;$

$V_1 \leftarrow F_2(V_2);$

$V_2 \leftarrow F_1(V_1);$

END;

$V_3 \leftarrow F_2(V_2);$

DO $V_1;$

$V_2 \leftarrow F_1(V_3);$

    DO $V_2;$

    $V_2 \leftarrow F_1(V_2);$

    $V_1 \leftarrow F_1(V_2);$

```
12          END;
13       V1 ← F2(V1);
14       END;
15       V3 ← F1(V1);
```

We begin with the output vector

$$<V_1, V_2, V_3>$$

and, on applying Routine A once, we obtain

$$<H_1{}^{(V_1)}[V_1, V_2, V_3]_1, H_1{}^{(V_1)}[V_1, V_2, V_3]_2, F_1(V_1)>$$

where H_1 denotes the iterative in lines 7-14. Expanding H_1 we find that

$$H_1 = <F_2(V_1), H_2{}^{(F_1(V_3))}[F_1(V_3)]_2>$$

where H_2 denotes the iterative in lines 9-12 and the $F_1(V_3)$ is from line 8. Similarly, on expanding H_2, we find

$$H_2 = <F_1(F_1(V_2)), F_1(V_2)> \quad .$$

Now, substituting H_2 into H_1, we have

$$H_1 = <F_2(V_1),<F_1(F_1(V_2)), F_1(V_2)>^{(F_1(V_3))}[F_1(V_3)]_2> \cdot$$

Substituting this into the original expression we obtain

$$<<F_2(V_1),<F_1(F_1(V_2), F_1(V_2)>^{(F_1(V_3))}[F_1(V_3)]_2>^{(V_1)}$$

$$[V_1,V_2,V_3]_1,\langle F_2(V_1),\langle F_1(F_1(V_2)),F_1(V_2)\rangle^{(F_1(V_3))}$$

$$[F_1(V_3)]_2\rangle^{(V_1)}[V_1,V_2,V_3]_1,F_1(V_1)\rangle.$$

We have now expanded lines 7 - 15. We leave it to the reader as an exercise to expand lines 1 - 6 and incorporate them into the above.

§6. Equivalence and Normal Form Algorithms for Linear Post-Loop Schemata

(6.1) Loop Functional Expressions (lfs) for Linear Schemes

From §5 we know that the form of a functional expression is either

(a) $\langle F_1,\ldots,F_n\rangle(E_1,\ldots,E_n)$

or

(b) $\langle F_1,\ldots,F_n\rangle^{[E]}_{\langle X_1,\ldots,X_n\rangle}(E_1,\ldots,E_n)$

where each F_i, E_i, or E is itself a functional expression.

In the case of monadic linear schemes we can show that the form is much simpler than this. First we do not need vector iteration, since there is only one feedback variable (at most) in each loop. Therefore we can write the expression as

(a) $\langle F_1\rangle(E_1,\ldots,E_n)$

or

(b) $\langle F_1\rangle_{\langle X_1\rangle}(E_1,\ldots,E_n)$.

But we can further simplify the expression in an important way. Since all the functions are monadic, the output of a scheme is simply a composition sequence

$$f_{i_1} \circ f_{i_2} \circ \dots \circ f_{i_p}$$

where each f_{i_j} is one of the function inputs to the scheme. We want our functional expression for the scheme to more closely resemble a composition sequence. We can accomplish this by singling out the one variable in any expression which is the input to the composition sequence; we call it the <u>primary variable</u>. We then notice that because the scheme is linear, this variable must also be the feedback variable (if any). We single out the primary variable in our notation by writing it on the same line as the function expression and moving all the other variables up to the exponent line. Thus the expression

$$\langle F_1 \rangle^{[E]}_{\langle X_1 \rangle}(E_1,\dots,E_n)$$

is now written as

$$\langle F_1 \rangle^{[E]}(E_1)^{(E_2,\dots,E_n)}$$

because E_1 is the primary input (the X_1 is no longer written because we know that E_1 is substituted for the primary variable).

For example, in the scheme

```
DO;
     DO;
          Z ← F1(Z);
     TEST X;
TEST Y
```

the primary input is Z and the functional form is

$$<<f_1>^{[x]}(z)>^{[y]}(z)$$

or, alternatively, by omitting the primary variable until the end,

$$<<f_1>^{[x]}>^{[y]}(z) \; .$$

We can write a fairly simple syntactic definition of these loop functional expressions for monadic linear Post-Loop schemes. Using a modified (but transparent) BNF notation we we get

<u>loop functional expression (lf)</u>

$L \to f \mid L\ L \mid <L>^{[V]}(x)^{(V,\ldots,V)}$

$V \to x \mid L$

where f is a function variable and x is an individual variable. Often we omit the composition sign, , between function letters, e.g. we write $f_1 f_2$ for $f_1\ f_2$. Sometimes we omit the primary variable, especially when listing a sequence of compositions. Finally, the reader should recall that the correspondence between input expressions, say $E_1,\ldots,E_n$ in $<F>^{[E](E_1,\ldots,E_n)}$ and variables in F, say $x_1,\ldots,x_n$, is given by the numerical ordering of the subscripts. In this case E_i corresponds to x_i. A typical lf is given below.

<u>Example</u>

$$f_0<<f_3<f_2>^{[x_1]}(z)>^{[x_2]}(z)^{(x_1)}>^{[<f_1>^{[x_2]}(x_1)]}(f_1(x_1),x_2)f_0(z)$$

which we could also write as

$$f_0 << f_3 < f_2 >^{[y_1]} >>^{[y_2]} (y_1)_{>} [< f_1 >^{[x_2]} (x_1)] (f_1(x_1), x_2)\, f_0$$

i.e., leaving out the primary variable and relabelling the variables inside various functional expressions.

There is one case in which the primary variable must always be written. That is when the iteration variable is also the primary variable, as in the example below.

<u>Example</u>

```
DO;
     DO;
          X ← F(X);
     TEST X;
TEST X
```

The functional expression is $<< f >^{[x]} (x) >^{[x]} (x)$. (The reader can check that we never need to express an identity between the primary variable and any other inputs because all such relationships are instances of feedback of the primary variable into the iteration variable of the functional expression being iterated.)

We call this type of functional expression an atom and we deal with it below in the more general setting of extended loop functional expressions (elfs).

We summarize the above discussion as a theorem which the reader can easily prove (by induction on length and depth of nesting).

Theorem 10: If S is a monadic linear PL scheme, then its functional expression can be written in the form

$$<F_1>^{[E_1]}(x_1)^{(I_1^1,\ldots,I_{n_1}^1)} \cdots <F_m>^{[E_m]}(x_1)^{(I_1^m,\ldots,I_{n_m}^m)}$$

where E_i, F_i, and I_j^i are loop functional expressions and x_1 is the primary input variable.

The equivalence algorithm in (6.5) applies to any pair of PL schemes represented by loop functional expressions of the above form. The theorem shows that this class includes (in fact properly) the linear schemes.

(6.2) Extended Loop Functional Expressions (elfs)

In the normal form and equivalence algorithms developed here it will be necessary to consider the equivalence of extended loop functional expressions. These arise because we want the normal forms for

```
DO;                           and   DO;
     DO;                                 DO;
          Z ← F(Z);                           Z ← F(Z);
     TEST X1;                            TEST X2;
TEST X2                             TEST X1
```

to be the same and to be as if both were from the scheme

```
DO;
     Z ← F(Z);
TEST X1·X2 ,
```

where we use the multiplication to indicate that the iteration is to run $|X_1|\cdot|X_2|$ steps. To do this we need "$X_1 \cdot X_2$" to be a legitimate subexpression even when we substitute arbitrary loop functional expressions for the variables.*

The general idea for obtaining extended expressions is to allow arithmetic expressions with loop functional expressions (and, recursively, extended loop functional expressions) as operands and to allow these elfs to be substituted for iteration variables. Using the same modified BNF notation as before we have

<u>extended loop functional expressions (elfs)</u>

$$E \rightarrow L \mid E\ E \mid (E)+(E) \mid (E)\cdot(E) \mid (E)^{c} \mid \langle L\rangle^{[W]}{}_{(x)}{}^{(V,\ldots,V)}$$

with a proviso

$$W \rightarrow E \mid y$$

where the proviso is that x does not occur in W. The meanings of L, V, x, and y are as before, and c is a positive integer.

We give below some simple examples of elfs and a more extensive example is given in (6.3) after the discussion of atoms. The method for assigning elfs to schemes is part of the normal form algorithm of (6.5).

---

* Another way to accomplish this would be to extend the Loop schemata language by allowing arithmetic expressions as iteration variables. We could then show that the two languages were equivalent and that the functional form of a scheme in the extended language is an elf.

Examples

(1) $<<f_1 \circ f_2>^{[x_1+x_2\cdot x_3+2]}>^{[x]} \circ f_2 \circ f_1(x)$

(2) $<<f_1 \circ f_2>^{[x_1]}(x_1)>^{[x_2]}$

(3) $<<f_1>^{[<<f_2>^{[x_3]}(x_3)>_{(x_1)} \circ < f_1>^{[x_1]}(x_3)]}>^{[x_2]}(x_1)^{(x_3)}$

(4) $<f_1>^{[(<<<f_2>^{[x_3]}(x_3)> \circ <f_1>^{[x_2]}(x_3)>^{[x_2]}(x_3)) \cdot (x_2)]}(x_1)$

(6.3) Atoms

Definition: An atom is a functional expression

$<L>^{(L)}(v)$ where v occurs in L .

The following are examples of atoms

(1) $<f_1 \circ f_2>^{[x]}(x)$

(2) $<f_1 \circ f_2>^{[<f_3>_{(x_1)}^{[x_2]}(x_1)]}$

Atoms are important because they behave somewhat analogously to function letters in the way they combine with other expressions, especially in the procedure for contracting loops (given below).

The role of elfs and atoms together is illustrated in the following example:

S_1	and	S_2
DO;		DO;
Z ← F(Z);		Z ← F(Z);
TEST X_2;		TEST X_1;
DO;		DO;
Z ← F(Z);		Z ← F(Z);
TEST X_1		TEST X_2

We want to reduce these to the same canonical form,

$<f(z)>^{[x_1+x_2]}$. Products enter when we consider equivalent schemes like

```
DO;                         DO;
  DO;                         DO;
  Z ← F(Z);                   Z ← F(Z);
  TEST X1;                    TEST X1;
  DO;                       TEST X3;
  Z ← F(Z);                 DO;
  TEST X2;                    DO;
TEST X3                       Z ← F(Z);
                              TEST X2;
                            TEST X3
```

which are written as $<f(z)>^{[(x_1+x_2)\cdot x_3]}$ or in canonical form as $<f(z)>^{[x_1\cdot x_3+\ x_2\cdot x_3]}$.

We do not allow expressions like $<L>^{[E]}(v)$ where v occurs in E because in an expression like

$$<f_1\circ f_2>^{[<f_1>^{[x_2]}(x_1) + x_2]}(x_1)$$

the terms $f_1^{[x_2]}(x_1)$ and x_2 do not commute under +, i.e. the two schemes below are not equivalent.

DO;	and	DO;
$X_1 \leftarrow F_2(X_1)$;		$X_1 \leftarrow F_1(X_1)$;
$X_1 \leftarrow F_1(X_1)$;		TEST X_2;
TEST X_2;		DO;
DO;		$X_1 \leftarrow F_2(X_1)$;
$X_1 \leftarrow F_1(X_1)$;		$X_1 \leftarrow F_1(X_1)$;
TEST X_2;		TEST X_2;
DO;		DO;
$X_1 \leftarrow F_2(X_1)$;		$X_1 \leftarrow F_2(X_1)$;
$X_1 \leftarrow F_1(X_1)$;		$X_1 \leftarrow F_1(X_1)$;
TEST X_1		TEST X_1

(6.4) <u>Discussion of the Normal Form and Equivalence Algorithms</u>:

The equivalence test will simply be a check for identical normal forms. This discussion is to motivate the construction of this form and to indicate why equivalent schemes have the same normal form.

The linearity of the schemes and the use of primary variables to further "linearize" the form of a scheme causes the external form of a linear scheme to be

$$* \quad A_1^{[E_1]}(I_1^1,\ldots,I_{n_1}^1) \circ A_2^{[E_2]}(I_1^2,\ldots,I_{n_2}^2) \circ \ldots \circ$$

$$A_m^{[E_m]}(I_1^m,\ldots,I_{n_m}^m)$$

where A_i, I_k^j, and E_i are elfs.

The primary input is x, and all other inputs occur in various of the expressions I_k^j. (For many purposes it is sufficient to view the expression as $A_1^{[E_1]} \circ \ldots \circ A_m^{[E_m]}$.)

Any such expression is put into normal form in the following steps: (i) exponents are moved into atoms (we call this "contracting loops" and it is a delicate point); this may result in terms with products as exponents, (ii) atoms are put into normal form, (iii) equivalent adjacent bases are collected, i.e. if $B_i \equiv B_{i+1}$, then $B_i^{[E_i]} B_{i+1}^{[E_{i+1}]}$ becomes $B_i^{[E_i+E_{i+1}]}$, (iv) broken groupings are collected, e.g. $f_1 \circ f_2 \circ (f_3 \circ f_1 \circ f_2)^n \circ f_3$ and $f_1 \circ (f_2 \circ f_3 \circ f_1)^n \circ f_2 \circ f_3$ are both put in the form $(f_1 \circ f_2 \circ f_3)^{n+1}$, (v) as the process proceeds, variables are renamed in order.

Two schemes S_1, S_2 will be equivalent with respect to Y iff their normal form expressions in Y are identical. The reasons for this will be sketched briefly below before the precise normal form algorithm is given.

The most basic underlying idea is that on input x two schemes produce sequences of function calls

$$S_1: \ f_{i_1} \circ f_{i_2} \circ \ldots \circ f_{i_n}$$

$$S_2: \ f_{j_1} \circ f_{j_2} \circ \ldots \circ f_{j_p}$$

where f_{i_k} , f_{j_k} are input functions. The schemes are equivalent iff $n = p$ and $j_k = i_k$, $k = 1,\ldots,n$. We show that this does not happen unless the normal forms are identical.

Clearly the result is true in expressions with no exponents because they are of the form $f_{i_1} \circ \ldots \circ f_{i_p}(x)$.

Indeed, if the expressions are not equivalent, they differ for infinitely many inputs (in fact for all $x > x_0$ for some x_0) . Suppose this is true for expressions with exponents of depth n (i.e. for schemes with DO-loops nested to depth n). Then we must show it for expressions with maximum nesting n+1 (of arbitrary length).

All loop functional expressions have the external form

$$S_1 \equiv A_1^{[E_1](I_1^1,\ldots,I_{n_1}^1)} \circ A_2^{[E_2](I_1^2,\ldots,I_{n_2}^2)} \circ \ldots \circ A_m^{[E_m](I_1^m,\ldots,I_{n_m}^m)} .$$

Let S_2 have this form with $\bar{A}$, $\bar{E}$, $\bar{I}$ replacing A, E, I and with $\bar{m}$ replacing m . Assume these expressions are in normal form, but not identical. We then show they are not equivalent.

Let $A_{i_1}, \overline{A}_{i_1}$ be the first location from the left where the expressions are not identical. We concentrate our attention on

$$A_{i_1}^{[E_{i_1}]}(I_1^{i_1},\ldots,I_{n_1}^{1})\circ\ldots\circ A_m^{[E_m]}(I_1^m,\ldots,I_{n_m}^m) \quad \text{and}$$

$$\overline{A}_{i_1}^{[\overline{E}_{i_1}]}(\overline{I}_1^{i_1},\ldots,\overline{I}_{n_{i_1}}^{i_1})\circ\ldots\circ\overline{A}^{[\overline{E}_{\overline{m}}]}(\overline{I}_1^{\overline{m}},\ldots,\overline{I}_{n_{\overline{m}}}^{\overline{m}})$$

Notice that $A_{i_1}, \overline{A}_{i_1}$ are atoms. We want to check whether $A_{i_1} \equiv \overline{A}_{i_1}$. If either exponent, E_{i_1} or $\overline{E}_{i_1}$ is constant, the corresponding atom could be nested to depth $n+1$. If both exponents are variables, then the atoms are equivalent iff they are identical. So we first assume an exponent is constant.

Assume A_{i_1} is nested to depth $n+1$. Notice that its form is $A^{[E(v,v_1,\ldots,v_n)]}(v)$ and suppose $\overline{A}_{i_1}$, is $\overline{A}^{[\overline{E}]}(\overline{v})$ where $\overline{v}$ may or may not be in $\overline{E}$. Since the atom itself is in normal form, A is compared to $\overline{A}$ and, because the depth of nesting is less, they are equivalent iff they are identical.

If $A \equiv \bar{A}$, then we claim $A_{i_1} \equiv \bar{A}_{i_1}$, iff $E \equiv \bar{E}$ which happens iff E and $\bar{E}$ are identical because again the degree of nesting is less (here we actually need the induction hypothesis on elfs). The claim holds because if $E \not\equiv \bar{E}$, then one scheme can produce more A-terms in the final evaluation sequence than the other can and because of the normal form, no adjacent terms A_{i_1-1}, A_{i_1+1} can make up the difference.

If $A \not\equiv \bar{A}$, then regardless of the exponents, the contribution of A_{i_1} and $\bar{A}_{i_1}$ terms in the final evaluation sequence is different. So, we have that $A_{i_1} \equiv \bar{A}_{i_1}$ iff they are identical.

Returning to the main expression, if $A_{i_1} \equiv \bar{A}_{i_1}$, then $E_{i_1} \not\equiv \bar{E}_{i_1}$ otherwise the expressions would be identical. Now we argue as above to conclude that more A_{i_1} terms appear in the final evaluation sequence, so the expressions are not equivalent.

IF $A_{i_1} \not\equiv \bar{A}_{i_1}$, then we consider whether the E_{i_1}, or $\bar{E}_{i_1}$ terms contain variables or not. If one does, then we can argue as before that inequivalent A_{i_1}, $\bar{A}_1$ terms occur in the final evaluation sequence. If one or both terms contain constants, then by condition (iv) of the normal form, the next block containing a variable exponent is not able to combine with these A_{i_1} terms to form equivalent groupings in the final evaluation, so the schemes are inequivalent.

This brief and incomplete sketch will be filled in in Section 6.6 where we prove the correctness of the precise normal form algorithm given in the next section.

(6.5) Normal Form and Equivalence Algorithms for elfs.

I. Normal Form Algorithm for elfs (NFA)

(1) (a) Put elf into dnf (disjunctive normal form),

$$T_1 + T_2 + \ldots + T_n \,,$$

by distributing all products over sums. Regard each loop expression, L , as a new variable (indeterminate) in this process. Make a list of variables and corresponding expressions

$$\begin{array}{l} X_1 - L_1 \\ X_2 - L_2 \\ \quad\vdots \\ X_n - L_n \end{array}$$

Notice each L_i is an lf (rather than elf).

(b) Compare each L_i, L_j; ask $L_i \equiv L_j$, i, j=1,...,n . List the equivalences in a table, and use the first variable to name the expression.

(c) Collect terms (retaining the normal form).

Examples:

$((<f_1f_2>^{[x]}(x)) + (<f_2f_1>^{[x]}(x))^2)(<f_1f_2f_1>^{[x]}\circ<f_1>^{[x]}(y))$

becomes $(x_1+x_1^2)x_2$ which in dnf is $x_1x_2+x_1^2x_2$ which is

$(<f_1f_2>^{[x]}(x))\ (<f_1f_2f_1>^{[x]}\circ<f_1>^{[x]}(y))$

$+\ (<f_2f_1>^{[x]}(x))^2\ (<f_1f_2f_1>^{[x]}\circ<f_1>^{[x]}(y))$.

Notice that we can parse these E-expressions in a simple manner:

with parentheses	with precedence
$E \to (E) + (E)$	$E \to A + E \mid A$
$E \to (E) \cdot (E)$	$A \to L \cdot A \mid L$

(2) Contracting loops

Given an iteration block $(A_i)^{[E_i]}(I^i)$ where $I^i = (I_1^i,\ldots,I_{n_i}^i)$, the exponent E_i may be applicable to a smaller base than A_i . For an instance, in

```
DO;
 DO;
  X3 ← F(X3);
 TEST X2;
TEST X1
```

the form is $<<f>^{[x_2]}>^{[x_1]}(x_3)$ and the outer exponent, x_1, is applicable to the base $<f>$ which is smaller than the base $\ll f>^{[x_2]}>$. So we will "contract the outer loop" to obtain the equivalent expression $<f>^{[x_1 \cdot x_2]}(x_3)$.

In general if A_i has the form $<\bar{A}_i>^{[v]}(I^i)$, then contraction is possible (I^i is the same inside because no new variables can be introduced in passing to the outer loop), and we form

$$<\bar{A}_i>^{[v \cdot E_i]}(I^i) \quad .$$

This step relies on the fact that loop invariant code has already been moved outof the loop expressions (when they were formed), thus a scheme like

```
DO;
 X1 ← F1(X3);
 DO;
 X2 ← F2(X2);
 TEST X1;
TEST W
```

appears as

```
X1 ← F1(X3);
DO;
 DO;
  X2 ← F2(X2);
  TEST X1;
TEST W
```

which has the lf $\langle\langle f_2\rangle^{[x_1]}(x_2)\rangle^{[w]}(f_1(x_3))$.

The contraction step is performed on all loop expressions, working inward until no further contractions are possible. Then the next loop is contracted.

(3)

Notice that the loop functional expression (lf) is now given to us in the form

$$S = (A_1)^{[E_1]}\circ(I^1)\,(A_2)^{[E_2]}(I^2)\circ\ldots\circ(A_n)^{[E_n]}(I^n)$$

where $I^i = (I^i_1,\ldots,I^i_{n_i})$. We shall sometimes drop the I^i terms for notational simplicity.

(a) Our first task is to incorporate constant exponents. An exponent E_i is constant iff it is an integer.

Search S from left to right until the

first term $(A_{i_q})^{[E_{i_q}]}$ surrounded by

constant terms, is found, say

$$A_{i_1}A_{i_2}\ldots A_{i_{q-1}}(A_{i_q})^{[E_{i_q}]}A_{i_{q+1}}\ldots A_{i_{q+r}}$$

Suppose $A_{i_q} \equiv \overline{A}_1\overline{A}_2\ldots\overline{A}_{p_q}$, then see if there is a match such that

$$A_{i_j}\ldots A_{i_{q-1}}\overline{A}_1\ldots\overline{A}_t \equiv \overline{A}_{t+1}\ldots\overline{A}_{p_q}A_{i_{q+1}}\ldots A_{i_{q+s}} .$$

If there is, then form a new term

$\hat{A}_{i_q} = A_{i_j}\ldots\overline{A}_t$ and rewrite S as

$$A_{i_1}\ldots A_{i_{j-1}}(\hat{A}_{i_q})^{[E_{i_q}+1]}A_{i_{q+s+1}}\ldots A_{i_{q+r}} .$$

Continue the process moving right across S . (This process will be applied recursively to the A_i blocks when the NFA is called on them.)

The process is illustrated by the simple case

$S = (abc)^{[E_1]}a(bca)^{[E_3]}bc$. The result is

$(abc)^{[E_1]}(abc)^{[E_3+1]}$.

(b) The next task is to determine which A_i blocks are equivalent. We first apply the NFA to each A_i . Then we can assume that A_i appears in the form $B_1^{p_1} \circ \ldots \circ B_m^{p_m}$ where each $p_i \in \mathbb{N}$ and where no adjacent loop expressions B_i are equivalent. We say A_i is <u>prime</u> iff it has the form B^p .

Next compare adjacent blocks, A_i, A_{i+1} . If both are prime say

$A_i = B_i^{p_i}$ and $A_{i+1} = B_{i+1}^{p_{i+1}}$, then check whether $B_i \equiv B_{i+1}$. If so, replace

$$(A_i)^{[E_i]}(A_{i+1})^{[E_{i+1}]} \text{ by } (B_i)^{[p_i \cdot E_i + p_{i+1} \cdot E_{i+1}]} .$$

If one is not prime, then see if the expressions for A_i and A_{i+1} are identical, if so form

$$(A_i)^{[E_i + E_{i+1}]} \text{ for } (A_i)^{[E_i]} \circ (A_i)^{[E_{i+1}]} .$$

If not, then go to the next pair.

A key point here is that if A_{i+1} is equivalent to B^p, then it will appear as B^p because the NFA has been applied to each block A_i.

Examples:

(i) $(ab)^{x_1}(ababab)^{x_2}$ becomes $(ab)^{x_1+3\cdot x_2}$

(ii) $(ab)^{x_1}(ab)^{x_2}(ab)^{x_3}(ab)^{x_3x_5}(ab)^{x_4x_5}$ becomes

$(ab)^{x_1+x_2+x_3+(x_3+x_4)\cdot x_5}$

(iii) $(abbcd)^{x_1}(a)(bbcda)^{x_2}(bbcd)(abbcdabbcd)^{x_3}$

becomes $(abbcd)^{x_1+(x_2+1)+2\cdot x_3}$

(iv) $(abc)^{x_1}a(bca)^{x_2}bc(adbc)^{x_3}ad(bcad)^{x_4}$

becomes $(abc)^{x_1+x_2+1}(adbc)^{x_3+x_4}ad$

Simple example for common base:

$$(aa)^{E_1}(aaaa)^{E_2}(aaa)^{E_3}(aaaaaa)^{E_4}(aaa)^{E_5}$$

becomes

$$(a)^{2E_1+4E_2+3E_3+6E_4+3E_5}$$

(4) Now put exponents into normal form.

Example

We shall put the following scheme into normal form as an example:

```
DO;
    X3 ← F1(X3);
    X2 ← F2(X2);
TEST X3;
DO;
    DO;
        Z ← F1(Z);
        DO;
            Z ← F2(Z);
            Z ← F1(Z);
        TEST X1;
        Z ← F2(Z);
    TEST X2;
TEST X3;
DO;
    DO;
        Z ← F1(Z);
        Z ← F2(Z);
        DO;
            Z ← F1(Z);
            Z ← F2(Z);
        TEST X1;
    TEST X1;
TEST X3
```

The functional expression for z (the primary variable) is

$$<<<f_2f_1>^{[x_1]}f_2f_1>^{[x_1]}>[<f_1>^{[x_3]}(x_3)](x_1)$$

$$<<f_2<f_1f_2>^{[x_1]}f_1>[<f_2>^{[x_3]}(x_2)](x_1)>[<f_1>^{[x_3]}(x_3)](x_1,x_2) \quad (z)$$

The external form is

$$A_1[<f_1>^{[x_3]}(x_3)](x_1) \quad A_2[<f_1>^{[x_3]}(x_3)](x_1,x_2)$$

where

$$A_1 = <<f_2f_1>^{[x_1]}f_2f_1>^{[x_1]}$$

and

$$A_2 = <f_2<f_1f_2>^{[x_1]}f_1>[<f_2>^{[x_3]}(x_2)](x_1).$$

The algorithm will now cause a contraction of A_1 to the form

$$A_1 = <<f_2f_1>^{[x_1]}f_2f_1>^{[x_1 \cdot <f_1>^{[x_3]}(x_3)]}$$

and a similar one for A_2, obtaining the elf

$$A_2 = <f_2<f_1f_2>^{[x_1]}f_1>[<f_2>^{[x_3]}(x_2) \cdot <f_1>^{[x_3]}(x_3)].$$

Now the normal form algorithm applied to each base block puts both of them into the form

$$<f_2f_1>^{[x_1+1]}$$

Thus, after collecting over common bases, we get the normal form

$$<<f_2f_1>^{[x_1+1]}>^{[x_1 \cdot <f_1>^{[x_3]}(x_3)+<f_2>^{[x_3]}(x_2) \cdot <f_1>^{[x_3]}(x_3)]}(z).$$

II. Equivalence algorithm for elfs.

Given the NFA, the algorithm for equivalence is simple. Let S and $\overline{S}$ be two linear Post-Loop schemes.

(a) Select a correspondence α between input variables of S and $\overline{S}$, output variables and function variables. Use this correspondence to rename the variables of S and $\overline{S}$ so that corresponding variables and only such variables receive the same names. Apply the NFA to the renamed schemes and denote the results by $S(\alpha)$, $\overline{S}(\alpha)$.*

(b) $S \equiv \overline{S}$ iff $\exists \alpha S(\alpha) = \overline{S}(\alpha)$, i.e. S and $\overline{S}$ are equivalent iff the renamed normal form schemes are identical under some α.

We shall treat the correctness of this algorithm in the next section.

(6.6) Correctness proof for monadic Linear Post-Loop equivalence: Outline

Theorem 11: The monadic Linear Post-Loop Equivalence problem is recursively decidable.

Given schemes S, $\overline{S}$ with variable assignment α,

---

* Notice that all individual variables except input variables occur in function bodies, and we use the convention that they are labeled $x_1, x_2, \ldots, x_n, \ldots$ taking variables as ordered in the expression for arguments, as discussed at the end of (5.2).

suppose the renamed forms $S(\alpha)$, $\overline{S}(\alpha)$ are not identical. Then we look from the left for the first place at which they differ and we will show that they are not equivalent near that point. We proceed by induction on depth + length so that we can assume the theorem for any pieces of the scheme. Indeed we assume that the schemes differ infinitely often if they are not equivalent.

The key point in the analysis arises as follows. Suppose the first point of difference occurs at $A_{i_o}, \overline{A}_{i_o}$ (when $S = (A_1)^{E_1} \ldots (A_n)^{E_n}$, $\overline{S} = (\overline{A}_1)^{\overline{E}_1} \ldots (\overline{A}_m)^{\overline{E}_m}$. If the difference occurs because $A_{i_o} \not\equiv \overline{A}_{i_o}$, then the result follows easily since we can lay down enough of A_{i_o} and $\overline{A}_{i_o}$ to produce an irrevocable difference in the final calling sequence (from the left).

However, if $A_{i_o} \equiv \overline{A}_{i_o}$, then the schemes differ only in the number of A_{i_o} terms appearing in the calling sequence and it might be possible that this difference in the number of terms is compensated by adjacent expressions.

To assure that adjacent expressions cannot help, we rely on step 3 of NFA. To apply this we must show that the only way for adjacent terms to help is if they are in a form which would have caused incorporation at step 3 of NFA.

A good way to accomplish this is to first recognize that the adjacent term, A_{i_o+1}, must have the same <u>base</u>

as A_{i_o} , i.e. $A_{i_o} \equiv A_{i_o+1}$. But we must be careful with the term base, here we must allow the base of atoms, i.e. we might have

$A_{i_o}^{E_{i_o}}$

```
DO;
 DO;
 Y ← F(Y);
 TEST X;
TEST X
```

---

and

$A_{i_o+1}^{E_{i_o}}$

```
DO;
 DO;
 X ← F(X);
 TEST X;
TEST X
```

We then must show that the base cannot occur in an atom. Once that happens, then we are in the situation where the common base would be incorporated at step (3) of NFA and we get a contradiction to the normal form.

To show that the base cannot occur in an atom which is different than A_{i_o} we proceed by induction on the depth of nesting. The idea is that if the adjacent atom is higher (has more nestings), then we can easily use our freedom to select values for the f_i to spoil any chance that this term compensates for terms in $A_{i_o}^{(E_{i_o})}$ or $A_{i_1}^{(E_{i_1})}$.

If the atoms have the same depth, then we look at their internal structure and use the equivalence algorithm to guarantee a difference infinitely often between feedback terms inside a loop. We then use our freedom to choose f to convert this infinitely often occurring difference into an arbitrary difference and argue as before.

The analysis required in this last step is essentially implicit in the NFA. We want to examine all possible ways in which $(A_{i_o})^{E_{i_o}}$ $(\overline{A}_{i_o})^{\overline{E}_{i_o}}$ can differ. We then see that each difference which still leads to equivalence is covered by NFA. In the case of atoms we use step 2, otherwise step 3.

Q. E. D.

This decision procedure is basically a more detailed formulation of the one given in [1] which was thought to apply to monadic Pre-Loop.

(6.7) Conclusion

We hope to present a more transparent decision procedure as we continue our investigations into the equivalence problem over algebraically restricted domains, work suggested in [1] and begun by the second author, and into polyadic functions.

References

[1] Constable, R. L., Loop Schemata, Third Annual ACM Symposium on Theory of Computing, 1971, pp. 24-39.

[2] Constable, R. L. and A. B. Borodin, On the Efficiency of Programs in Subrecursive Formalisms, Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory, 1970, pp. 60-67 (to appear in JACM).

[3] Constable, R. L. and David Gries, On Classes of Program Schemata, SIAM Journal on Computing, vol. 1, no. 1, 1972.

[4] Engler, E., Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, vol. 188, 1971, p. 372.

[5] Ershov, A. P., Theory of Program Schemata, Proceedings of IFIP Congress 71, vol. 1, 1971, pp. 144-163.

[6] Hopcroft, J. E. and R. Tarjan, Planarity Testing in V log V Steps: Extended Abstract, Computer Science Technical Report STAN-CS-71-201, Stanford University, 1971.

[7] Ianov, Iu. A., The Logical Schemes of Algorithms, Problems of Cybernetics (USSR), vol. 1, 1960, pp. 82-140.

[8] Luckham, D. C., D. M. R. Park, and M. S. Paterson, On Formalised Computer Programs, Journal of Computer and System Sciences, vol. 4, 1970, pp. 220-249.

[9] Manna, Z., Properties of Programs and the First-Order Predicate Calculus, JACM, vol. 16, no. 2, April 1969, pp. 244-255.

[10] Meyer, A. R. and D. Ritchie, Computational Complexity and Program Structure, IBM Research Paper RC-1817, May 15, 1967.

[11] Minsky, Marvin L., Computation: Finite and Infinite Machines, Prentice-Hall, 1967, p. 212.

[12] Paterson, M. S., Equivalence Problems in a Model of Computation, Ph. D. Dissertation, Cambridge University, 1967; also published as Artificial Intelligence Technical Memo No. 1, M.I.T., 1970.

[13] Péter, Rōsza, Recursive Functions, Third edition, Academic Press, 1967.

[14] Rogers, Hartley, Jr., Theory of Recursive Functions and Effective Computability, McGraw-Hill, 1967.

[15] Rosenberg, A., On Multi-Head Finite Automata, Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design, 1963, pp. 221-228.

[16] Sheperdson, J. C. and H. E. Sturgis, Computability of Recursive Functions, JACM, vol. 10, 1963, pp. 217-255.

[17] Strong, H. R., Jr., Translating Recursion Equations into Flow Charts, Journal of Computer and System Sciences, vol. 5, June 1971, pp. 254-285.