PROGRAM SCHEMES WITH PUSHDOWN STORES

BY

Steven Brown

David Gries

Thomas Szymanski

TR 72 - 126

Department of Computer Science
Cornell University
Ithaca, New York 14850

PROGRAM SCHEMES WITH PUSHDOWN STORES

by

Steven Brown†

David Gries

Thomas Szymanski

ABSTRACT

We attempt to characterize classes of schemes allowing pushdown stores, building on an earlier work by Constable and Gries [1]. We study the effect (on the computational power) of allowing one, two, or more pushdown stores, both with and without the ability to detect when a pds is empty. A main result is that the use of using one pds is computationally equivalent to allowing recursive functions.

We also study the effect of adding the ability to do integer arithmetic, and multi-dimensional arrays.

KEYWORDS Program schemes, schemata, pushdown stores, stacks, recursion, programming languages

†This research was supported by the National Science Foundation under Grant GJ-28176.

§1. Introduction. In Constable and Gries [1] the following classes of schemes were defined:

P = class of schemes using simple variables, with assignment, conditional, goto and while statements.

P_A = class of schemes P, with the additional feature of arrays of subscripted variables.

P_{Ae} = class of schemes P_A, with the additional feature of an equality test on subscript values.

P_R = class of schemes P, with the additional feature of ALGOL-like recursive procedures.

P_M = class of schemes P, with the additional feature of a finite number of distinguishable markers, or constants, allowed as values. (There may appear arbitrarily many instances of a marker.)

P_{pds} = class of schemes P, with the additional feature of pushdown stores.

$P_{\aleph}$ = class of schemes P, with the additional feature of integer arithmetic.

One could then build other classes. For example, P_{AM} is the class of schemes allowing arrays and markers. In particular, $P_{(m,n)}$ refers to the class of schemes allowing m pushdown stores and n markers.

In a sense, a scheme is an abstraction of a program, and by studying these classes of schemes we gain more understanding of the computational power of the different data structures and control mechanisms used in programming languages. A large part of a recent paper by Constable and Gries [1] was devoted to showing the following inclusions and equivalences, where for example $P < P_R$ means that for every scheme in P there exists an equivalent scheme in P_R but not conversely; and $P_{Ae} \equiv P_{AM}$ means that for every scheme in P_{Ae} there exists an equivalent scheme in P_{AM}, and conversely:

$$P < P_R \leq P_{(1,0)} < P_A \equiv P_{Ae} \equiv P_{AM} \equiv P_{(2,1)} \equiv P_{(1,0)\aleph}$$

Hence you can "do more" with arrays than you can with recursive procedures. It was claimed that P_{Ae} and equivalent classes are "universal". All the above inclusions and equivalences are effective, except for $P_A \equiv P_{Ae}$; for any scheme $S \in P_{Ae}$ an equivalent scheme $S' \in P_A$ exists, but it can't in general be constructed! In [1] it is assumed that all the basic functions and predicates are total.

This paper resolves some questions left open in [1], and discusses some more inclusions and equivalences of classes of schemes, mostly having to do with pushdown stores. Our results can be best given by the inclusion diagram of Figure 1.

Two new classes of schemes appear in the Figure. P_{Rg} is the class of schemes P_R allowing the additional feature of global variables (as used in ALGOL). P_{pdsb} is the class of schemes P_{pds} with the additional feature of a test for the bottom of a pushdown store. (In P_{pds} execution of a pop instruction has absolutely no effect if the stack is empty.) Thus $P_{(2b,0)}$ allows 2 pushdown stores, tests on emptiness of these pds's, and no markers.

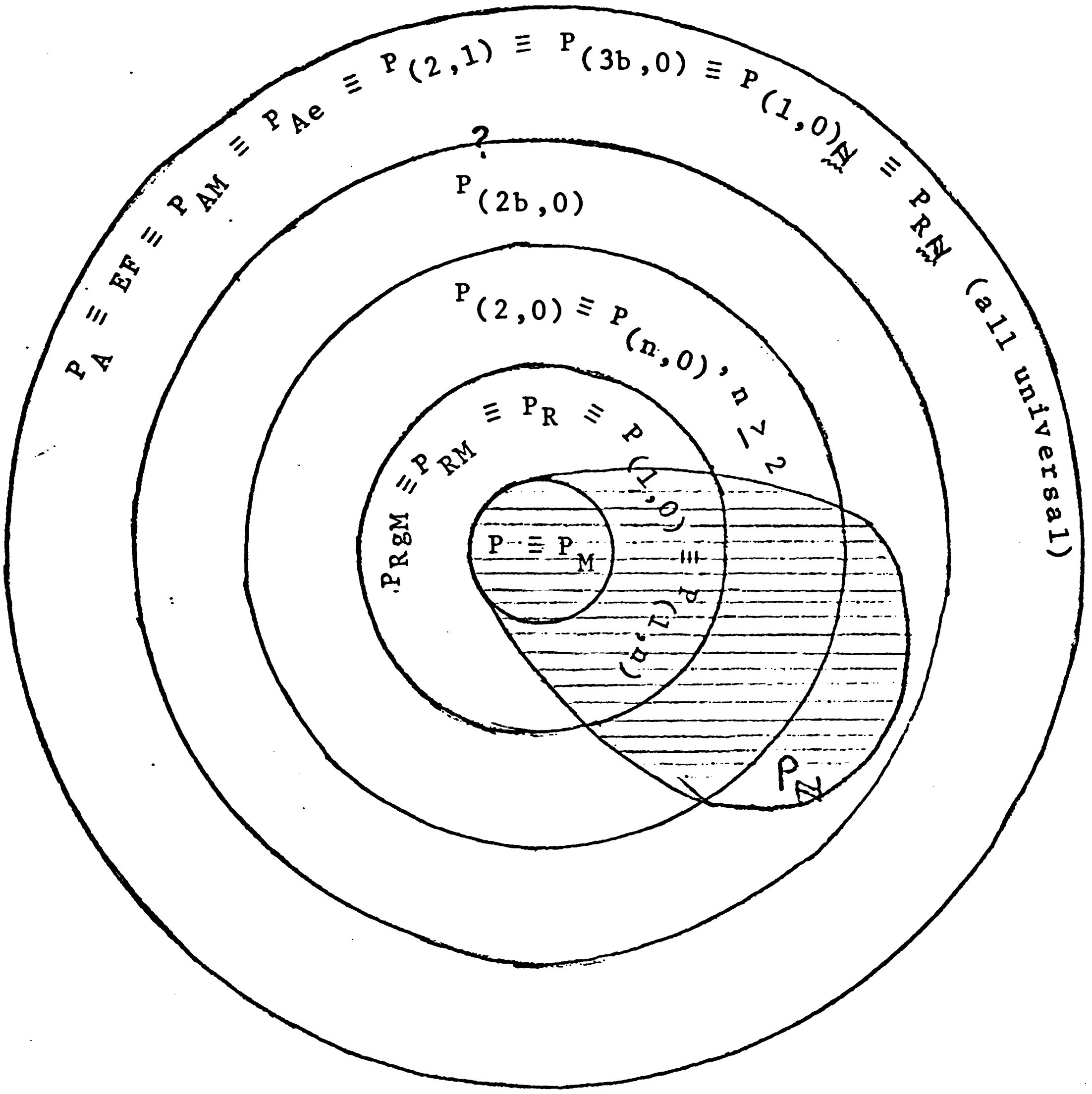


Figure 1. Inclusion diagram for classes of schemata.

The question mark on the line above $P_{(2b,0)}$ indicates an unsolved problem; we don't know whether

$$P_{(2b,0)} < P_{(3b,0)} \quad \text{or} \quad P_{(2b,0)} \equiv P_{(3b,0)} \quad .$$

The inclusion diagram brings out some interesting points. Oddly enough, adding the feature of markers adds nothing to the power of many classes; we have

$$P \equiv P_M, \; P_R \equiv P_{RM}, \; P_{(1,0)} \equiv P_{(1,n)} \quad \text{for} \quad n \geq 0, \quad \text{and} \quad P_A \equiv P_{AM} \; .$$

Only when adding markers to $P_{(2,0)}$ do we add computational power, and then only one marker is needed to achieve "universality".

Adding the ability to do integer arithmetic, however, has more of an effect on the computational power. Thus, adding integer arithmetic to P_R or $P_{(1,0)}$ yields the "universal" class of schemes $P_{R\underset{\sim}{N}}$ or $P_{(1,0)\underset{\sim}{N}}$. Of special interest in the diagram is $P_{\underset{\sim}{N}}$. Note how it "contains a piece" of each of the other classes. According to Corollary 10.9 of [1], the characteristic property of this class is the following: Let S be any scheme in any class. Then there exists an equivalent scheme $S' \in P_{\underset{\sim}{N}}$ if and only if there is a bound n and an equivalent effective functional in which each expression and proposition can be evaluated using at most n variables. Thus the characteristic property is that the scheme really needs only a fixed, bounded number of variables, if it can internally perform integer arithmetic.

In [1], the pushdown store in P_{pds} was formulated so that a pop is a null operation if the pds is empty. This was done solely because it was the "cleanest" and easiest definition to work with. It is interesting to note that being able to test for the bottom of a pds is computationally important. Thus we have $P_{(2,0)} < P_{(2b,0)}$. Of course $P_{(1,0)} \equiv P_{(1b,0)}$, since $P_{(1,0)} \equiv P_{(1,n)}$ and we can simulate the test for the bottom of a stack by using a marker. Note also that $P_{(3b,0)}$ is universal and thus equivalent to $P_{(2,1)}$.

This paper is organized as follows. We assume the reader is familiar with [1] and refer to all the definitions and results given there, without repeating them here. The rest of this section is devoted to a few other necessary definitions and comments.

Section 2 discusses the equivalence of $P_{(1,0)}$ with P_R . This means that the data structure of a single stack is equivalent to the control mechanism of recursive procedures. In Section 3 we relate $P_{(1,0)}$ to $P_{(2,0)}$ and $P_{(n,0)}$, and $P_{(n,0)}$ to P_{Ae} for $n > 2$. Section 4 discusses the use of the statement which tests for the emptiness of a pds, and relates classes $P_{(nb,0)}$ for $n \geq 3$, with $P_{(2b,0)}$ and $P_{(2,0)}$.

In Section 5 we show how $P_{\mathbf{N}}$ fits in. In the final section we solve another open problem of [1]; we show that adding multidimensional arrays to P_A adds no more computational power. All equivalences and inclusions shown in this paper are effective.

(1.1) Definition. A scheme in the class P_{Rg} (Recursive functions allowing global variables) is a scheme in P_R (see Definition 3.4 of [1]) with the following change: the function definition may also have the form

<function def> :: = $f(v_1,\ldots,v_{Rf})$ global $w\{,w\}$; <body>

The global variables w_i in the statement "global $w_1,\ldots,w_n$" may not appear in the formal parameter list $v_1,\ldots,v_{Rf}$. $w_1,\ldots,w_n$ refer to the variables with the same names (if any) used in the main <body> of the programs and they are not initialized to Ω upon invocation of the function <body> . Note that if two <function def>s declare the same name to be global, then the names refer to the same variable.

(1.2) Definition. A scheme in the class P_{pdsb} (or $P_{(1b,0)}$, $P_{(2b,0)},\ldots$) is a scheme in the class P_{pds} (or $P_{(1,0)}$, $P_{(2,0)},\ldots$) (see [1, Definition 4.7] and [1, Section 7]), with the following additional statement type allowed:

<S> :: = IF EMPTYPDS(s) THEN [ℓ:] $\langle S\rangle_1$ ELSE [ℓ:] $\langle S\rangle_2$

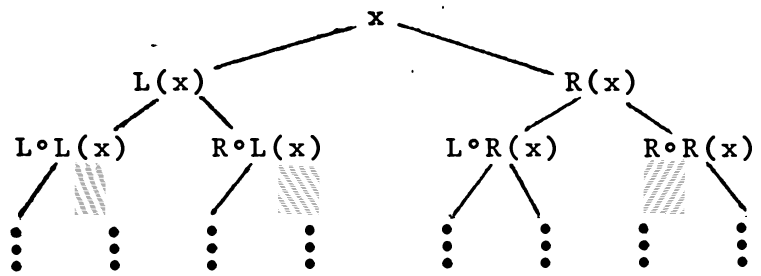
where s is a pushdown store.

We next define a functional which will be used frequently in this paper. Let

$$\text{(1.3)} \quad \text{Leaftest}(P,L,R,x) : \mathcal{P}(D) \times \mathcal{F}(D) \times \mathcal{F}(D) \times D \to D$$

$$\begin{aligned} &= x \text{ if there exists a sequence} \\ &\quad f_1, f_2, \ldots, f_n \text{ where each } f_i \\ &\quad \text{is either } L \text{ or } R \text{ and} \\ &\quad P(f_n \circ f_{n-1} \circ \cdots \circ f_1(x)) = \text{true}; \\ &= \text{undefined otherwise.} \end{aligned}$$

Informally, Leaftest is a search performed on the following binary tree:



Leaftest searches this tree in an attempt to find a node whose value makes the predicate P true. If such a node is found, Leaftest returns x as its output value; otherwise, the search continues forever.

Leaftest has been an important functional in the brief history of "comparative schematology". Paterson and Hewitt [3] first used it as a scheme which could not be performed in P_R . Gries and Constable [1] then gave a scheme in P_A for it, to help show that $P_R < P_A$. In this paper, Leaftest or variations of it are used to prove the inclusions

$P_{(1,0)} < P_{(n,0)}$ for $n > 1$, $P_{(n,0)} < P_A$, $P_{(n,0)} < P_{(2b,0)}$ ..

We shall also make use of "locators" in several proofs.

(1.4) <u>Definition</u>. Given a scheme S (in any class) a <u>locator</u> S' for S is a scheme with the following properties:

(1) S and S' use the same input variables, basic functions and predicates.

(2) When executing, S' attempts to find a predicate P_i of rank RP_i and two lists of argument values a_1 and a_2 such that $P_i(a_1)$ = true, $P_i(a_2)$ = false. If it finds them, S' puts the values of a_1 into variables $RT_1,\ldots,RT_{RP_i}$, puts the values of a_2 into variables $RF_1,\ldots,RF_{RP_i}$, and transfers control to a statement

$$BEGIN_i: \ HALT(OMEGA) \ .$$

(3) If S' does not find a predicate as in (2), then

a) if S executes infinitely long then so does S';

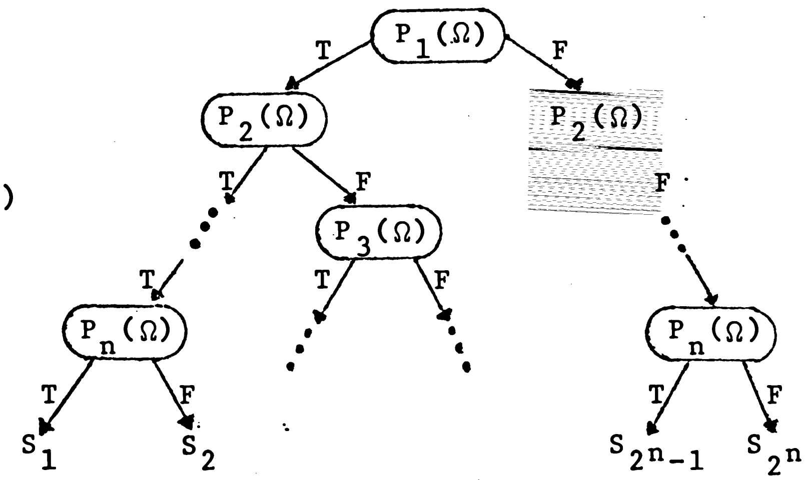
b) if S halts with value V then so does S'.

The chief use of a locator is in the construction of a scheme S' without markers equivalent to a scheme S which uses markers. Once the predicate is "located" as described in the definition above the markers of S can be "simulated" in S' using a sequence of the argument lists a_1, a_2 as bits (see Definition 5.1 of [1]). A main result which we shall use is the following rewording of Theorem 5.5 of [1].

(1.5) <u>Theorem</u>. <u>Let</u> S <u>be a scheme in some class</u>. <u>Let</u> S' <u>in class</u> P_2 <u>be a locator for S</u>. <u>Suppose there exist P-simulators</u> (see Definition 5.1 of [1]) <u>for</u> S <u>in</u> P_2. <u>The locator and P-simulators can be put together to form a scheme</u> S'' <u>in</u> P_2 <u>equivalent to S</u>.

<u>Proof</u> Assume without loss of generality that the predicates $P_1,\ldots,P_n$ of a scheme S all have rank 1. Then the locator we construct has the following form:

(1.6)



where the S_i are statements. S_1 for example must do the following:

(1.7) S_1 must "simulate" the v = (true,...,true)-autonomous behavior of S until either

(1) it halts and outputs the same result that S would, or

(2) a predicate P_i is evaluated with argument a_2 such that $P_i(a_2)$ = false . At this point RT is initialized to Ω , RF is set to a_2 , and control is transferred to $BEGIN_i$.

This is a <u>very</u> brief introduction to locators and simulators, and the reader is encouraged to review Sections 5 and 9 of [1].

Throughout the rest of this paper, all manipulations of pushdown stores will be written using the following notation:

PUSH(pd,V) — when executed, places the value currently stored in the variable V on the top of the stack pd.

POP(pd,V) — when executed, removes the top value from the stack pd and assigns it to the variable V. If the stack pd is empty when this statement is executed, then the operation is treated as a null operation.

§2. The Equivalence of P_R and $P_{(1,0)}$.

Theorem 7.5 of [1] showed that $P_R \leq P_{(1,0)}$. Here we prove that $P_{(1,0)} \leq P_R$, yielding the equivalence of P_R and $P_{(1,0)}$. Hence a single stack is just as computationally powerful as recursive procedures. The proof is a series of lemmas establishing the following inclusions, in order:

(2.1) $\quad P_{(1,n)} \leq P_{RgM} \leq P_{Rg} \leq P_R \leq P_{(1,0)} \quad$ for $\quad n \geq 0$

An obvious by-product is that neither global variables nor markers add anything to the power of recursive procedures (P_R). A look at the proof of $P_{RgM} \leq P_{Rg}$ (Theorem 2.3) will also convince the reader that $P_M \leq P$ and thus $P_M \equiv P$.

Suppose we have a scheme $S \in P_R$. We can translate S into an equivalent scheme $S1 \in P_{(1,0)}$, then translate S1 into $S2 \in P_{RgM}$, into $S3 \in P_{Rg}$, and finally into $S4 \in P_R$, again. You will note by the constructions of the lemmas that S4 uses only <u>one</u> recursive procedure definition. Hence, for any scheme $S \in P_R$ which uses $n > 1$ recursive procedures we can construct an equivalent scheme S4 in P_R which uses only <u>one</u> recursive procedure.

Another interesting point concerns the class $P_{(1,0)}$. Given any scheme $S \in P_{(1,0)}$ we can construct an equivalent scheme $S1 \in P_{(1,0)}$ such that if S1 halts, its pds is empty. This is quite remarkable since in $P_{(1,0)}$ one cannot test to see if the pds is empty. This fact comes out easily from the constructions in the lemmas involved.

(2.2) <u>Lemma</u>. $P_{(1,n)} \leq P_{RgM}$ <u>for</u> $n \geq 0$

<u>Proof</u> Given a scheme $S \in P_{(1,n)}$ which uses a single pds P, we construct an equivalent scheme $S3 \in P_{RgM}$. The basic idea is to define a function F which is essentially the same as the main scheme. The pds P becomes a simple variable P which is a formal parameter of F, and the pds is represented by the "stack" of invocations of F. Except for a second formal parameter, all other variables are global to F. This is illustrated in Figure 2.1. When $S \in P_{(1,n)}$ executes the statement PUSH(p,v) at $\langle S\rangle_1$, the scheme S3 executes a call of F, with the value of V as the argument. The main problem is that F should begin executing <u>not</u> at the first statement, but at statement $\langle S\rangle_2$ (see Figure 2.1). We do this by passing a marker as a second argument to F to indicate where it should begin executing.

Similarly, a pop instruction POP(p,w) is essentially a return instruction. Again, we must take sure that the calling invocation of F does not begin executing after its call (which was a push), but at the statement after this pop ($\langle S\rangle_5$ in Figure 2.1). To do this, the value returned by F is also a special marker.

Normal HALTs in F and pops in the main scheme must be handled similarly. We leave the details to the appendix.

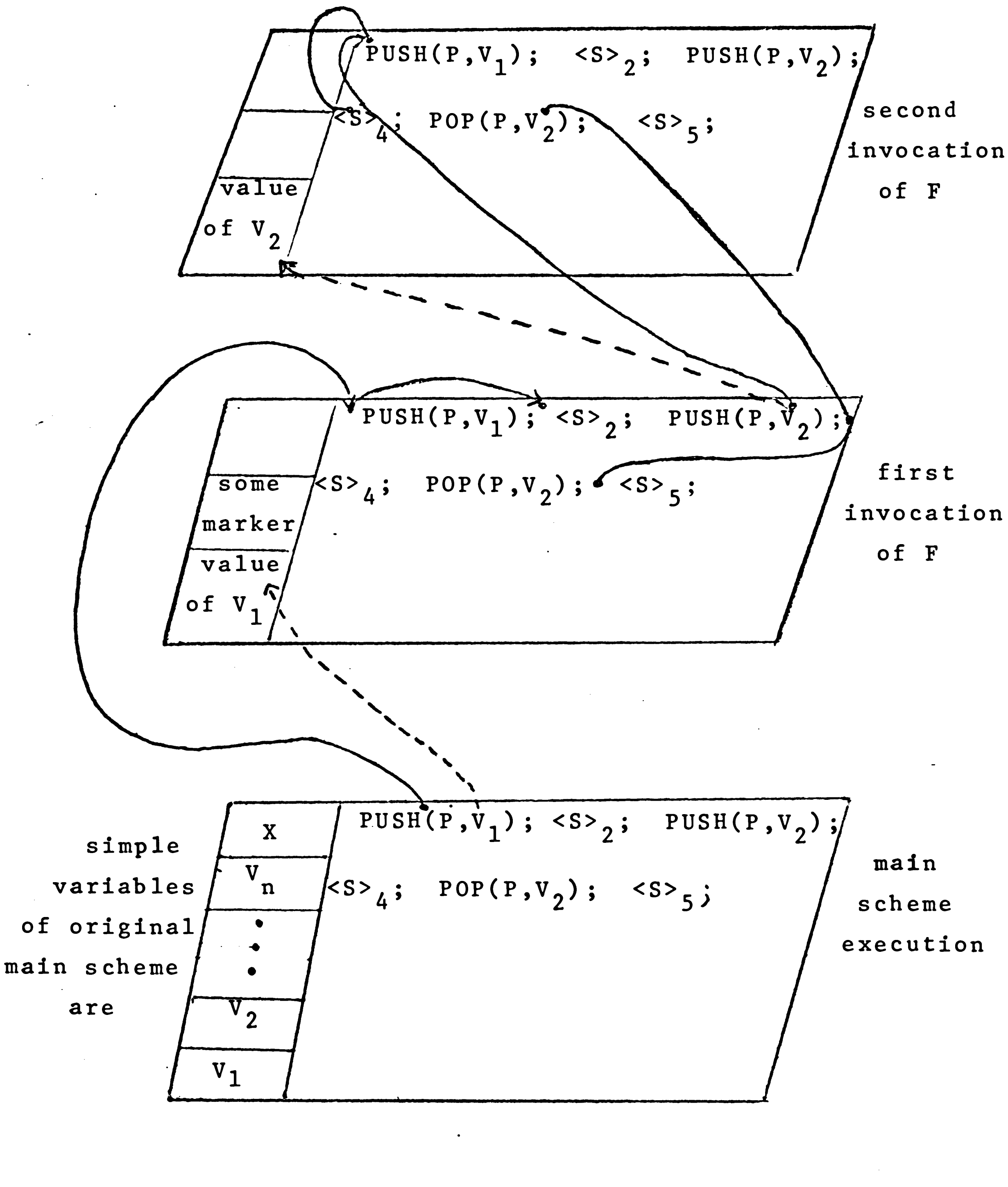


Figure 2.1 Representing a pds by function calls.

(2.3) Theorem. $P_{RgM} \leq P_{Rg}$

Proof We first show in Lemma 2.4 that we can construct P-simulators $\in P_{Rg}$ for any scheme $\in P_{RgM}$ (see Definition 5.1 of [1]). According to Theorem 1.5, we then need only show that for $S \in P_{RgM}$ we can construct a locator $\in P_{Rg}$. In Lemma 2.6 we establish the decidability of the finiteness of the v-autonomous behavior of any scheme $\in P_{RgM}$ (see Section 9 of [1]). This important fact, and the method of the decision, are used in Lemma 2.7 to build a locator $\in P$ (and thus $\in P_{Rg}$) for any scheme $\in P_{RgM}$.

(2.4) Lemma. Let $S \in P_{RgM}$ use a predicate P. Then we can construct a P-simulator $S1 \in P_{Rg}$.

Proof We proceed essentially as in the proof of Theorem 5.3 of [1]. Assume without loss of generality that P has rank 1, and that $P(RT) = T$, $P(RF) = F$, where RT contains the value rt and RT contains rf.

Suppose S uses markers $M_1, M_2, \ldots, M_k$. Each variable v of S is represented in S1 by variables $v, v^1, \ldots, v^k$. The following table indicates the correspondence between values stored in v during execution of S, and in $v, v^1, \ldots, v^k$ during execution of S1,

variable v in S	variables $v, v^1, \ldots, v^k$ in S1
$\bar{v} \in D$	$\bar{v}, rf, \ldots, rf$
M_1	$\Omega, rt, rf, \ldots, rf$
$\vdots$	$\vdots$
M_k	$\Omega, rf, \ldots, rf, rt$

We leave it to the reader to show how to translate the statements $v \leftarrow f(\ldots)$ (where f is a basic function), $v \leftarrow w$, $v \leftarrow M_i$, IF $p(\ldots)$ THEN ..., and IF $v = M_i$ THEN ..., of S into equivalent statements for S1 . The main problem is with calls and returns of recursive functions.

Each function definition $f(v_1,\ldots,v_n)$: ... is transformed into $f(v_1,v_1^1,\ldots,v_1^k,\ldots,v_n,v_n^1,\ldots,v_n^k)$: ..., so that the parameters get passed properly. For a call of a recursive function

$$w \leftarrow f(v_1,\ldots,v_n) \tag{2.5}$$

in S, however, we must return values not only for w, but also for $w^1,\ldots,w^k$. These will be returned in new global variables $x^1,\ldots,x^k$. Add them to the list of global variables in each function definition. Now change each call (2.5) to

BEGIN $w \leftarrow f(v_1,v_1^1,\ldots,v_1^k,\ldots,v_n,v_n^1,\ldots,v_n^k)$;
$w^1 \leftarrow x^1;\ldots;w^k \leftarrow x^k$;
END

and change each HALT(v) within a function definition to

BEGIN $x^1 \leftarrow v^1;\ldots;x^k \leftarrow v^k$; HALT(v) END

Q.E.D.

(2.6) <u>Lemma</u>. <u>It is decidable whether the v-autonomous behavior of a scheme in</u> P_{RgM} <u>is finite or infinite</u>.

<u>Proof</u> We can assume that the global variables of S are $V_1,\ldots,V_g$ and that by suitable renaming of variables, they are <u>not</u> used as local variables or formal parameters. Assume that S is completely labeled. Let r be the largest of the ranks of the recursive functions of S, and let S use markers $M_1,\ldots,M_{m-1}$. Let S use predicates $P_1,\ldots,P_n$.

Consider the v-autonomous behavior of S, as described in 9.9 of [1]. This behavior does not depend on the input values, or on which value of the domain D is in any variable at any point. Using $\bar{v}$ to denote <u>any</u> value in D, the m possible values that can affect the behavior at some point are $\bar{v},M_1,\ldots,M_{m-1}$.

If the v-autonomous behavior is infinite, then one of the two following things must happen: (1) the level of nesting of function invocations is infinite; or (2) within the execution of a function (or main program), there must be an infinite loop. We now derive bounds on the nesting of function invocations and the number of statements executed within a function which, if executed, indicate there is infinite behavior.

Suppose there is a call $v \leftarrow f(\ldots)$ of a recursive function. The behavior of the scheme while f is executing depends only on the <u>values</u> of the actual parameters and of the global variables $V_1,\ldots,V_g$. Hence there are at most

$$m(r + g) \text{ possible different behaviors .}$$

Thus, if a recursive function f is called recursively $m(r + g) + 1$ times (without returning), two calls on f have already occurred with the same actual parameter and global variable values $(\bar{v}, M_1, \ldots, M_{m-1})$. Neither of these two calls will finish and the scheme is in an infinite loop.

Secondly, consider the v-autonomous behavior within a recursive function f (or the main scheme). Suppose f has s statements and ℓ local variables (including the formal parameters). Then we know the recursive function has infinite v-autonomous behavior if the behavior has as many as $s \cdot r \cdot (\ell + m) + 1$ labels in it.

Q.E.D.

(2.7) Lemma. Every scheme S in P_{RgM} has a locator S' in P.

Proof The locator S' for S has the form shown in (1.6) we need only show how to construct the statements $S_1, \ldots, S_{2^n}$ described there. We outline in the appendix the construction of S_1 only which simulates the v-autonomous behavior of S where $v = (\text{true}, \ldots, \text{true})$, as described in 1.7. The construction of the other S_i is similar. The important point to note is that we can effectively decide whether the v-autonomous behavior of S is finite or infinite (Lemma 2.6).

Q.E.D.

(2.8) Lemma. $P_{Rg} \leq P_R$.

Proof Suppose scheme $S \in P_{Rg}$ has function definitions for functions $F_1,\dots,F_n$, and suppose that the variables used globally are $V_1,\dots,V_m$. By suitably renaming the local variables we can make sure that $V_1,\dots,V_m$ are used only as global variables, and we can assume S has the form

$$
(2.9)\quad
\begin{array}{ll}
(v,\dots,v): & \text{<S>; } \dots \text{; <S>} \\
F_1(v,\dots,v): & \underline{\text{global}}\ V_1,\dots,V_m\text{; <S>; } \dots \text{; <S>} \\
\vdots & \qquad\quad \vdots \\
F_n(v,\dots,v): & \underline{\text{global}}\ V_1,\dots,V_m\text{; <S>; } \dots \text{; <S>}
\end{array}
$$

We give in the appendix a construction which reduces by one the number of global variables. By executing this construction m times, we arrive at an equivalent scheme in P_{RM} . What this construction does is make V_1 a parameter of each function. This creates the problem that we cannot return the value of V_1 , so what we do first is call F_1 (say) to get the function value back, and then call a similar routine F_1' which returns the value for V_1 .

Q.E.D.

§3. Markerless Pds Schemes.

In this section we show that

$$P_A > P_{(n,0)} \equiv P_{(2,0)} > P_{(1,0)} \quad \text{for} \quad n \geq 2 .$$

The proper inclusions are both proved using the Leaftest scheme or a variation of it.

A second important idea is proved in Lemma 3.2; for any scheme $S \in P_{(n,0)}$ we can construct a locator in P. We use this to show that

(3.1) Theorem. $P_{(n,0)} \equiv P_{(2,0)}$ for $n \geq 2$.

Proof Lemma 3.2 shows how to construct a locator in P for S $S \in P_{(n,0)}$; because of Theorem 1.5 we need only show how to construct P-simulators in $P_{(2,0)}$ for S. Consider S to be in P_{pdsM} rather than P_{pds} and use Theorem 7.3 of [1] to construct $S1 \in P_{(2,1)}$ equivalent to S.

We construct a simulator $S2 \in P_{(2,0)}$ for S_1 (and thus for S) by simulating the single marker. We represent each simple variable V of S1 by variables V and V' and initially set each V' to rf . If V contains a value $\bar{V} \in D$, $V' = rt$ (in S2).

To produce the P-simulator we make a copy S' of S1 change it as follows (we assume without loss of generality that all predicates have rank 1):

(a) At the beginning of S' insert for every simple variable V the statement $V' \leftarrow RF$;

(b) For the pds's PD1 and PD2 add at the beginning of S'

```
PUSH(PD1,RF);  PUSH(PD2,RF);
```

to indicate they are empty.

(c) Change each PUSH(PDj,V) (except those inserted in (b)) to

```
BEGIN PUSH(PDj,V);  PUSH(PDj,V');  PUSH(PDj,RT) END
```

(d) Change each POP(PDj,V) to

```
BEGIN POP(PDj,X);
      IF P(X) THEN
         BEGIN POP(PDj,V');  POP(PDj,V) END
      ELSE PUSH(PDj,X)
END
```

where X is a new temporary variable. This construction allows a pop of an empty pds to be treated as a null operation.

(e) Change each assignment V ← W to

```
BEGIN V ← W;  V' ← W'  END
```

(f) Change each assignment V ← M to

```
BEGIN V ← OMEGA;  V' ← RT  END
```

(g) Change each assignment V ← f(...) to

```
BEGIN V ← f(...);  V' ← RF  END
```

(h) Change each test

$$\text{IF } V = M \text{ THEN } <S_1> \text{ ELSE } <S_2>$$

to $$\text{IF } P(V') \text{ THEN } <S_1> \text{ ELSE } <S_2>$$

It should be clear from the construction that the modified S' runs in $P_{(2,0)}$ and simulates the behavior of S exactly.

Q.E.D.

(3.2) Lemma. <u>For any scheme</u> $S \in P_{(n,0)}$ <u>we can construct a locator</u> $S' \in P_\ell$.

<u>Proof</u> The locator has the form given in (1.6). We show how to construct only statement S_1 of (1.6) as described in (1.7).

Assume that S has $|S|$ statements. We first show that under autonomous behavior the scheme references at most the top $|S|$ locations of any pds. With constant predicates, S executes ℓ (say) statements, $\ell \leq |S|$, and then halts (hence at most ℓ locations of any pds can be referenced), or executes ℓ different statements and then enters an infinite loop, where the loop consists of $r \leq |S|$ statements.

If a pds has a net growth during execution of the r statements of the loop, then no element lower than $|S|/2$ from the top can be referenced. On the other hand, if a pds shrinks in size or remains the same during one execution of the loop, then the stack size is at most $\ell + r/2 \leq |S|$.

We now show how to construct S_1 . We generate $(|S| + 1)^n$ different copies of S (changing the labels so the copies are independent). Let the copies be denoted by $S'_{i_1 i_2 \ldots i_n}$ where each i_j denotes the number of occupied positions in simulated stack j. Clearly the initial "state" is $S'_{000\ldots0}$. We will assign new labels to every statement in every copy; the labels will be $\ell^{j}_{i_1 i_2 i_3 \ldots i_n}$, where $j = 1,\ldots,|S|$, and the i_m are keyed to the copy.

The copies are then altered and connected in the following way. Consider the pushdown stack m.

a) In all copies $S'_{i_1 \ldots i_{m-1}, 0, i_{m+1} \ldots i_n}$, all statements popping stack m are replaced by the null statement.

b) In all copies $S'_{i_1 \ldots i_m \ldots i_n}$, where $i_m < |S|$, after each PUSH statement labeled $\ell^{j}_{i_1 \ldots i_m \ldots i_n}$ for stack m we insert

$$\text{GO TO } \ell^{j+1}_{i_1 \ldots i_{m+1} \ldots i_n}$$

c) In all copies $S'_{i_1 \ldots i_m \ldots i_n}$, $i_m > 0$ after each stack m POP statement labeled $\ell^{j}_{i_1 \ldots i_m \ldots i_n}$ we insert

$$\text{GO TO } \ell^{j+1}_{i_1 \ldots i_{m-1} \ldots i_n}$$

Most of this complexity is to guarantee that a null operation is performed if an empty stack is popped.

Assume now without loss of generality that all predicates are monadic and that we have $P_i(RT) = \text{true}$ for each predicate P_i. (We are creating S_1 of (1.6) only, now.) We represent each pds p by new simple variable $V_{p,1},\ldots V_{p,|S|}$. We modify all PUSH(p,w) statements and all POP(p,w) statements (in all copies of S) as follows:

a) Change PUSH(p,w) to

BEGIN $V_{p,|S|} \leftarrow V_{p,|S|-1}$; ...; $V_{p,2} \leftarrow V_{p,1}$; $V_{p,1} \leftarrow W$ END

b) Change POP(p,w) to

BEGIN $W \leftarrow V_{p,1}$; $V_{p,1} \leftarrow V_{p,2}$; ...; $V_{p,|S|-1} \leftarrow V_{p,|S|}$ END

We also replace each statement

IF $P_i(X)$ THEN <S_1> ELSE <S_2>

by

IF $P_i(X)$ THEN <S_1>
ELSE BEGIN RF ← X; GO TO $BEGIN_i$ END

and add statements

$BEGIN_i$: HALT (OMEGA);

at the end of the scheme.

The result of these transformations is statement S_1 .

Q.E.D.

(3.3) <u>Lemma</u>. <u>Leaftest cannot be computed in</u> $P_{(n,0)}$.

<u>Proof</u> Suppose $S \in P_{(n,0)}$ computes Leaftest (P,L,R,X) . Now consider the following scheme S':

```
S'(P,L,R,X):  V ← X;
              IF P(X) THEN GO TO BEGIN1;
              Locator (S);
      BEGIN1: HALT(V);
```

The notation "Locator (S)" refers to the body of the locator scheme for S constructed according to Lemma 3.2. Control is passed to the label BEGIN1 by the locator only if Locator(S) has generated some value for which P is true, since P is the only predicate in S which can potentially take on both true and false values. By the construction of S' the only new values which S' can generate are concatenated applications of the functions L and R applied to the initial value X. By definition these are just node values in the binary tree generated by X, L and R. Hence, control is passed to BEGIN1 only if a value is found for which P is true. It should be equally clear that if there is any value in the tree which P is true. It should be equally clear that if there is any value in the tree which makes P true, Locator(S) by hypothesis will eventually find it and will transfer control to BEGIN1.

We must also consider the possibility that Locator(S) will stop on the value Ω, which can arise in several situations, according to the definitions of schemata behavior (see Constable and Gries [1]). We can eliminate this case by observing that the value of the Leaftest functional is by definition independent of the truth value of $P(\Omega)$. Since Locator(S) is a P-scheme (Lemma 2.1), it has only a finite number of variables, and we can modify Locator(S) so as to keep track of which locations contain the value Ω. This is done by keeping many copies of the scheme, such that each copy corresponds to particular variables $V_1,\ldots,V_t$ containing Ω and all other variables containing computed values. By this means, therefore, we can force a false branch whenever $P(\Omega)$ is tested. Such a locator clearly performs the same locator tasks as the originial one.

After having taken care of the Ω problem as above, we see that S' is equivalent to S. Referring once again to Lemma 2.1, we note that since the modified Locator (S) is a P-scheme, S' is also a P-scheme. But S must still be able to compute Leaftest in its full generality, and we therefore would have a P-scheme (S') which computes Leaftest. But this contradicts the result of [4] in which it it shown that Leaftest cannot be computed in P_R (and hence not in P). Thus S could not have existed and Leaftest is not computable in $P_{(n,0)}$

Q.E.D.

(3.4) Theorem. $P_{(n,0)} < P_A$

Proof Consider $S \in P_{(n,0)}$ to be in P_{pdsM}. By Theorem 8.2 of [1] we can construct an equivalent scheme $S1 \in P_{AM}$, and by Theorem 5.4 of [1] we can construct P-simulators for it in P_A. Secondly, by Lemma 3.2, we can construct a Locator in P (and hence in P_A) for S. We then apply Theorem 1.5.

Theorem 6.6 of [1] and Lemma 3.3 show that the containment is proper.

Q.E.D.

(3.5) Theorem. $P_{(1,0)} < P_{(2,0)}$

Proof. Clearly $P_{(1,0)} \leq P_{(2,0)}$; to show that the containment is proper we exhibit a function computable in $P_{(2,0)}$ but not in $P_{(1,0)}$. Consider the functional $f(P,L,R,X,Y,Z)$:

IF P(Y) <u>and</u> $\neg$P(Z) THEN Leaftest(P,L,R,X) ELSE X

First we show how to compute the above functional in $P_{(2,0)}$. Clearly we can write Leaftest(P,L,R,X) as a scheme in $P_{(2,1)}$ since we can do it in P_A and $P_A \equiv P_{(2,1)}$. Lemma 3.1 shows how to construct a P-simulator for Leaftest in $P_{(2,0)}$. The following scheme in $P_{(2,0)}$ then computes the above functional:

```
(X,Y,Z):  IF P(Y) and ¬P(Z) THEN
               BEGIN RT ← Y; RF ← Z;
                     {P-simulator in P(2,0) for Leaftest}
               END
          ELSE HALT(X);
```

Suppose now we have a scheme $S(P,L,R,X,Y,Z) \in P_{(1,0)}$ which computes the above functional f. From it we construct a scheme $S'(P,L,R,X) \in P_{(1,m)}$ which computes Leaftest(P,L,R,X). Since $S' \in P_{(1,m)} \equiv P_{(1,0)} \equiv P_R$ and Leaftest(P,L,R,X) cannot be performed in P_R (Theorem 6.6 of [1]) we have a contradiction to the fact that a scheme to compute f existed in $P_{(1,0)}$.

To construct $S'(P,L,R,X)$ perform the following. Let M_1, M_2 be two markers, and let W be a new variable. Insert at the beginning of S the statements

$$Y \leftarrow M_1; \quad Z \leftarrow M_2;$$

§4. Bottom Markers and Pds's.

A major drawback to programming in $P_{(n,0)}$ is the inability to locate the bottom of a pushdown store. This makes it impossible to perform such useful tasks as transfering the contents of one pds into another while perhaps performing some action on each value as it goes by. However, every "real" programming language incorporating stacks or pds's also contains primitives which allow either the trapping of an interrupt on pds underflow or else explicit testing for empty pds's.

Accordingly, we extend the class $P_{(n,0)}$ by adding to the language the construct

$$\text{IF EMPTYPDS(pd) THEN } \langle S_1 \rangle \text{ ELSE } \langle S_2 \rangle$$

The semantics of this statement should be obvious. This new class will be called $P_{(nb,0)}$ where the b is intended to remind the reader that we now have the ability to find the <u>bottom</u> of the pds.

Intuitively, the ability to test for the bottom of a pds is less powerful than the ability to place markers in it. Classes utilizing markers are allowed an unbounded number of copies of the markers which can occur anywhere, whereas marking the bottom of each pds is equivalent to using only a fixed number of copies of each marker and requiring that the markers always appear in a certain relative position.

Then change each conditional

$$\text{IF } P(V) \text{ THEN } S_1 \text{ ELSE } S_2$$

of S to

$$\begin{array}{l} \text{IF } V = M_1 \text{ THEN } S_1 \\ \text{ELSE IF } V = M_2 \text{ THEN } S_2 \\ \qquad \text{ELSE IF } P(V) \text{ THEN } S_1 \text{ ELSE } S_2 \end{array}$$

We must show that $S'(P,L,R,X) = \text{Leaftest}(P,L,R,X)$ for all domains D and all interpretations of P, L, R, and X. For any interpretation, consider the domain $D' = D \cup \{M_1, M_2\}$ (where $D \cap \{M_1, M_2\} = \phi$), predicate P', and functions F', L' where

$$\begin{array}{lllll} P'(d) = P(d) & \text{for} & d \in D, & P'(M_1) = \text{true}, & P'(M_2) = \text{false} \\ L'(d) = L(d) & \text{for} & d \in D, & L'(M_1) = M_1, & L'(M_2) = M_2 \\ R'(d) = R(d) & \text{for} & d \in D, & R'(M_1) = M_1, & R'(M_2) = M_2 \end{array}$$

A look at S and S' will show that

$$S(P',L',R',X,M_1,M_2) = S'(P,L,R,X) \quad \text{for} \quad X \in D$$

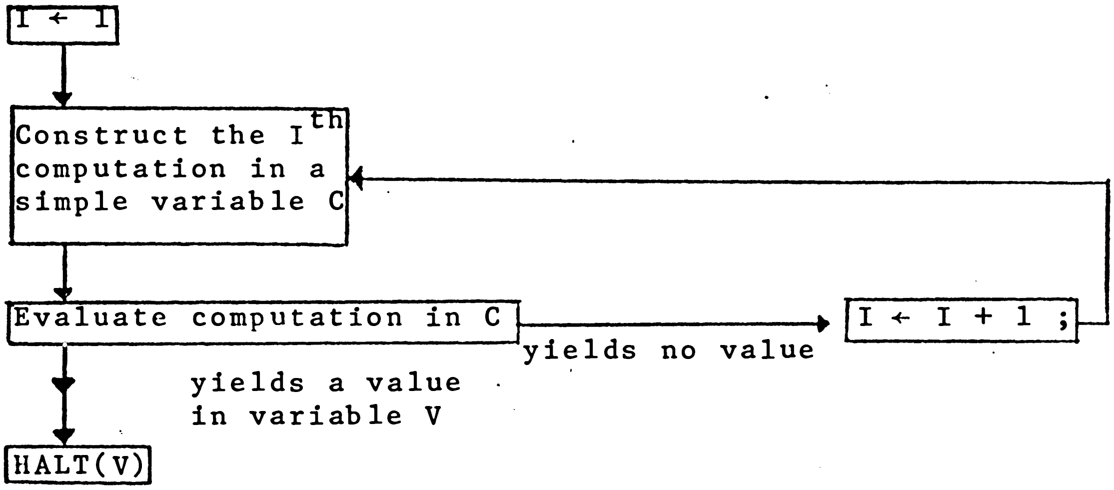
But, by definition of f we have $S(P',L',R',X) = S'(P',L',R',X)$ $= \text{Leaftest}(P',L',R',X) = \text{Leaftest}(P,L,R,X)$ for $X \in D$.

Q.E.D.

We are now ready to discuss the "universality" of $P_{(3b,0)}$. We do this in two parts. First of all, we show that for any scheme S in P_{Ae} there exists a scheme S' in $P_{(1,0)\mathbb{N}}$ which does *not* store integer values on the stack. This result arises easily from some results in [1] concerning effective functionals and program schemes. Secondly, Lemma 4.5 will show how to construct a scheme in $P_{(3b,0)}$ equivalent to S'.

(4.4) *Lemma. For any scheme S in P_{Ae} there exists an equivalent scheme S' in $P_{(1,0)\mathbb{N}}$ which does not store integer values on its pds.*

Proof Assertion 10.6 of [1] says that there exists an *effective functional* F equivalent to S (see Definition 10.3 of [1]); Assertion 10.8 of [1] then states that there exists a scheme S' in $P_{(1,0)\mathbb{N}}$ equivalent to F and thus equivalent to S. In the constructions in Assertions 10.6 and 10.8 of [1], S' has the form



We prove in this section the expected result that $P_{(1b,0)} \equiv P_{(1,0)}$. We also show that $P_{(3b,0)}$ is effectively equivalent to the "universal" classes P_{AM} , P_{Ae} and $P_{(2,1)}$. As far as $P_{(2b,0)}$ is concerned, we show that

$$P_{(2b,0)} > P_{(2,0)} > P_{(1b,0)} \equiv P_{(1,0)}$$

However, we don't know whether $P_{(2b,0)}$ is equivalent to $P_{(3b,0)}$ or not. This open problem will be discussed at the end of the section.

(4.1) Lemma. $P_{(nb,0)} \leq P_{(n,1)}$ for ~~an~~ $n \geq 1$.

Proof Given a scheme S in $P_{(nb,0)}$, we must construct an equivalent scheme S' in $P_{(n,1)}$. S' uses a marker M. We insert at the beginning of S statements to push M onto each pds. Next we replace all tests for an empty pds by tests for M at the top of the pds. This also requires changes in POP statements. We leave the details to the reader.

Q.E.D.

(4.2) Theorem. $P_{(1b,0)} \equiv P_{(1,1)} \equiv P_{(1,0)}$.

Proof Clearly $P_{(1,0)} \leq P_{(1b,0)}$. By Lemma 4.1 we know that $P_{(1b,0)} \leq P_{(1,1)}$. By Section 2 $P_{(1,1)} \leq P_{(1,0)}$.

Q.E.D.

(4.3) Theorem. $P_{(nb,0)} \leq P_{AM}$.

Proof By Lemma 4.1 we have $P_{(nb,0)} \leq P_{(n,1)}$ and by Theorem 8.2 of [1], $P_{(n,1)} \leq P_{AM}$.

Q.E.D.

This scheme S' satisfies the desired property; the pds is used only to hold temporary values in D occurring during the evaluation of computation C. Each computation is constructed in Polish postfix form enocded as an integer in a single variable, and uses only a finite number of simple variables.

Q.E.D.

(4.5) Lemma. Let S be a scheme in $P_{(1,0)\aleph}$ which never stores an integer value on its pds PD. Then we can find an equivalent scheme S' in $P_{(3b,0)}$.

Proof. In addition to pds PD, S' uses two pds's PD1 and PD2 as counters to simulate the contents of the integer variables and arithmetic in the finite control of S. We first modify the scheme S so that its set of variables can be partitioned into a set $\{X_1,\ldots,X_k\}$ which is used only for manipulating domain values and a set $\{V_1,\ldots,V_m\}$ which is used only for holding integer values. This modification can easily be made by "splitting" each variable of the original scheme into two copies and adding some states to the finite control of S. The X_i and V_i work together in that whenever the simulated variable to which an (X_i,V_i) pair corresponds contains a domain element, X_i contains the element and $V_i = 0$; whenever the simulated variable contain an integer, $X_i = \Omega$ and V_i contains the integer.

At any point in simulated time, the height of pds PD1 of S' will be

$$p_1^{c(v_1)} \cdot p_2^{c(v_2)} \cdot \ldots \cdot p_m^{c(v_m)}$$

where the p_i's are distinct prime numbers and $C(V_i)$ represents the contents of variable V_i. We retain the simple variables X_i for holding domain values. Since all V_i contain 0 initially, we initialize PD1 to a height of 1 by pushing Ω into the stack. We must now show how to simulate the primitive arithmetic operations of $V \leftarrow V + 1$, $V \leftarrow V \dot{-} 1$ and $V \circleddash 0$.

(1) Replace each statement $V_i \leftarrow V_i + 1$ by a compound statement which "pours" the contents of pds PD1 into pds PD2, inserting $p_i - 1$ new elements into PD2 with each element that is transferred from PD1 to PD2 . This multiplies the stack height by p_i . We can then restore the canonical state by pouring the elements back into PD1 from PD2 .

(2) Replace each test for $V_i \ominus 0$ by a compound statement which pours PD1 into PD2 , computing $Z = |PD1| \bmod p_i$. Thus $V_i = 0$ iff $Z = 0$. We then restore the canonical state.

(3) Replace each statement $V_i \leftarrow V_i \dot{-} 1$ by a compound statement which first tests for $V_i \ominus 0$ and does nothing further if true. Otherwise, we pour PD1 into PD2 , pushing onto PD2 only one element for every p_i that are popped from PD1 . We then restore the canonical state.

Q.E.D.

(4.6) <u>Theorem</u>. $P_{Ae} \equiv P_{(3b,0)}$.

<u>Proof</u> Apply Lemmas 4.4 and 4.5 to get $P_{Ae} \leq P_{(3b,0)}$. Apply Theorem 4.3 and the fact that $P_{AM} \equiv P_{Ae}$ (Theorem 8.8 of [1]) to get $P_{(3b,0)} \leq P_{Ae}$.

Q.E.D.

(4.7) <u>Theorem</u>. $P_{(2b,0)} > P_{(n,0)}$ for $n \geq 1$.

<u>Proof</u> The relations $P_{(2b,0)} \geq P_{(2,0)} \equiv P_{(n,0)}$ from left to right are (1) obvious, and (2) proved in Theorem 3.1. We need

only find a functional which is $P_{(2b,0)}$ computable but not $P_{(n,0)}$ computable. Leaftest (see introduction) is not $P_{(n,0)}$ computable by Lemma 3.4. We show it can be performed in $P_{(2b,0)}$ by the following algorithm (using pds's PD1 and PD2):

<u>Step 1</u>: Initialize PD1 to contain a copy of the input X.

<u>Step 2</u>: Transfer the contents of PD1 to PD2 , applying the predicate P to each value moved. Halt if any value yields true.

<u>Step 3</u>: Compute L(V) and R(V) for each value V in PD2 , storing the results in PD1 as they are computed. When PD2 becomes empty, return to Step 2.

The ability to test for an empty stack is crucial here, because it allows us to tell when all values have been transferred.

Q.E.D.

We have carefully avoided discussing the class $P_{(2b,0)}$ in this section because this class has resisted our best attempts at characterization. Intuitively, two pds's seem to be adequate for control purposes, because such a configuration is essentially a two counter machine [3] and has sufficient power to simulate any Turing machine. Moreover, even with one pushdown store available we have as much room for intermediate results as is necessary. Thus at first glance, it would seem likely that the operation of the two control stacks could be merged with that of the work stack and hence we could prove that $P_{(2b,0)}$ is also universal. However, none of our attempts to do this have been successful.

We will now introduce a functional which is a generalization of Leaftest and which is pertinent to the discussion of the power of $P_{(2b,0)}$. Suppose we are given a set of functions $\{F_1,\ldots,F_k\}$, a set of predicates $\{P_1,\ldots,P_m\}$ and a set of values $\{x_1,\ldots,x_n\}$. The class of all "arithmetic" expressions generable from these objects may be represented by the following context-free grammar:

$$
\begin{aligned}
E &\to x_1 \;\ldots\; x_n \\
E &\to F_1(\underbrace{E,\ldots,E}_{RF_1 \text{ times}}) \\
&\;\vdots \\
E &\to F_k(\underbrace{E,\ldots,E}_{RF_k \text{ times}})
\end{aligned}
$$

We now define

(4.8) $\text{Husearch}\,(F_1,\ldots,F_k,P_1,\ldots,P_m,x_1,\ldots,x_n)$

$= x_1$ if $\exists$ an integer i and expressions E_1 thru E_{RP_i} (as defined by E above) $\ni P_i(E_1,\ldots,E_{RP_i}) = \text{true}$

$=$ undefined otherwise.

Thus, Husearch searches the Herbrand Universe generated by the F_i's and x_j's.

Constable and Gries (Construction 9.11 of [1]) showed how to perform this search in P_A . A thorough discussion of the search program in P_A is given by Gries [2].

(4.9) <u>Theorem</u>. $P_{(2b,0)} \equiv P_{(3b,0)}$ <u>iff the Husearch functional can be computed in</u> $P_{(2b,0)}$.

<u>Proof</u> The "only if" portion follows immediately from Construction 7.11 of [1] and Theorem 4.6. To prove the "if" part, we sketch how the Husearch computation can be used to construct a locator in $P_{(2b,0)}$ for a scheme $S \in P_{(3b,0)}$. Once we have such a locator we can use the values it generates to simulate markers and thus have universal power (Theorem 8.4 of [1]).

Therefore, suppose we are given a scheme $S \in P_{(3b,0)}$. Let us assume autonomous behavior for S. Using the same approach as in the proof of Lemma 2.7, we first ascertain whether the autonomous behavior is finite or infinite. If it is finite we can clearly construct a locator in P. If the autonomous behavior is infinite, we launch into the Husearch computation. If Husearch never halts then S cannot halt (though the converse is clearly not true). If Husearch does halt, we can then simulate S directly with the values it returns.

Q.E.D.

Notice that the construction outlined in this theorem is non-effective, because it is recursively unsolvable whether the autonomous behavior of an arbitrary $P_{(3b,0)}$ scheme is finite or infinite. Even if we could "program" the Husearch functional in $P_{(3b,0)}$ we still would have left as an open problem whether or not the two classes $P_{(2b,0)}$ and $P_{(3b,0)}$ are effectively equivalent.

We may note that in the simple case in which S is a scheme using only monadic functions and predicates of any rank then Husearch can be computed in $P_{(2b,0)}$ and there does exist $S' \in P_{(2b,0)} \ni S' = S$. However, all attempts to program the general Husearch in $P_{(2b,0)}$ have so far failed, leading us to the

(4.10) <u>Conjecture</u>. $P_{(2b,0)} < P_{(3b,0)}$.

In some sense $P_{(2b,0)}$ is very "close" to the universal power of $P_{(3b,0)}$, because the slightest additional power given to $P_{(2b,0)}$ makes it universal. In particular let us give $P_{(2b,0)}$ one "chip" which it can place anywhere in its stacks and for which it can test. Note that there is only one copy of this chip C, so if we execute the statements

```
V ← C;
PUSH(PD1,V);
IF  V = C  THEN ...
```

the predicate must be false. We assume that only the latest copy of C exists, and other instances (such as in V above) are replaced in V above) are replaced by Ω. For convenience, we assume that as long as the chip is in a simple variable it is "moved around" by assignments; that is, the sequence

```
V ← C;
W ← C;
X ← C;
```

results in V and W having the value Ω and X containing the chip. However, when the chip enters a data structure (such as a pushdown stack) it becomes inaccessible until it is later fetched by the data structure accessing primitives. Thus,

```
X ← C;
PUSH(PD1,X);
Y ← C;
```

while syntactically valid, results in both X and Y containing the value Ω and the chip being on the top of the pds. If a POP(PD1,W) is executed, W then contains the chip. The concept of a chip is difficult to express clearly because it is antithetical to the usual notion of the contents of a variable.

(4.11) <u>Theorem</u>. $P_{(2b,0)C} \equiv P_{(3b,0)}$, <u>where the equivalence is effective</u>.

<u>Proof</u> 1) $P_{(3b,0)} \geq P_{(2b,0)C}$.

We know from Theorem 3.5 that $P_{(3b,0)}$ is universal, and therefore by Theorems 8.4 and 7.3 of [1],

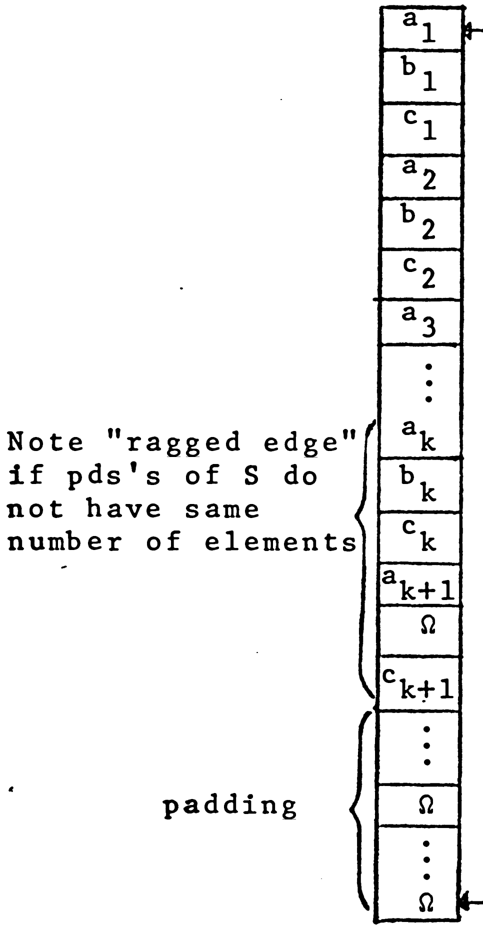
$$P_{(3b,0)} \overset{eff}{\equiv} P_{(2,1)} \overset{eff}{\equiv} P_{(2,3)}$$

(two stacks and three markers) .

Containment is obvious between $P_{(2,3)}$ and $P_{(2b,0)C}$.

2) $P_{(3b,0)} \leq P_{(2b,0)C}$

The argument in this direction depends on a Gödelization of the three pds's of an arbitrary scheme S in $P_{(3b,0)}$, so that these pds's can be represented in a new scheme S' in $P_{(2b,0)C}$. The method of pds storage where PD1 contains values $a_1, a_2, \ldots,$ PD2 contains $b_1, b_2, \ldots$ and PD3 contains $c_1, c_2, \ldots,$ is as follows:



The total height of the S' pds PD1 is

$$|PD1| = p_1^{|a|} p_2^{|b|} p_3^{|c|}$$

where $|a|$, $|b|$, $|c|$ are the lengths of the S pds's and the p_i are distinct primes. We obtain this height by inserting the proper padding at the bottom, as shown. Note that there will always be some padding if there are any pds elements.

In the simulation given below we assume that we initialize PD1 by PUSH(PD1,OMEGA) ; and that the resting configuration (between pds activity) is for the entire pds to be in PD1 and for PD2 to be empty.

Now we will show how to push, pop, and test for emptiness any of the three pds's:

<u>Push onto j^{th} pds</u>. We need to multiply the pds length in S' by p_j , then move every element in the j^{th} pds of S down 3 positions, and finally insert the new element in the j^{th} position.

We do this by

```
UNTIL EMPTYPDS(PD1) DO
      BEGIN POP(PD1,V); PUSH(PD2,V) END;
(thus moving the pds to  PD2 )
```

(The section below uses the chip to multiply the pds size by p_j , inserting the padding at the bottom of the pds.)

```
UNTIL EMPTYPDS(PD2) DO
      BEGIN
         POP(PD2,V); PUSH(PD2,C); PUSH(PD2,V);
         UNTIL EMPTYPDS(PD1) DO
            BEGIN POP(PD1,V); PUSH(PD2,V) END;
         PUSH(PD1,OMEGA); (repeated  pj-1  times)
               .
               .
               .
         POP(PD2,V);
         UNTIL  V = C  DO
            BEGIN PUSH(PD1,V); POP(PD2,V) END
      END
```

(Now we insert the new element and shift other elements of pds j down 3 positions)

```
POP(PD1,V); PUSH(PD2,V);  (repeated  j-1  times)
V1 ← new item;

UNTIL EMPTYPDS(PD1) DO
      BEGIN
         PUSH(PD2,V1); POP(PD1,V1);
         IF ¬EMPTYPDS(PD1) DO
             BEGIN POP(PD1,V); PUSH(PD2,V) END;
         IF ¬EMPTYPDS(PD1) DO
             BEGIN POP(PD1,V); PUSH(PD2,V) END;
      END;
```

(The element shifted off the bottom will be just padding.)

```
UNTIL EMPTYPDS(PD2) DO BEGIN POP(PD2,V); PUSH(PD1,V) END
```

(Thus restoring PD1.)

<u>Test for emptiness of j^{th} pds</u>: To simulate the statement

$$\text{IF EMPTYPDS}(\text{PDS}_j)\ \text{THEN}\ \langle S_1\rangle\ \text{ELSE}\ \langle S_2\rangle$$

we simply pour from PD1 to PD2, computing $Z = |\text{PD1}| \bmod p_j$. The j^{th} pds is empty iff $Z \neq 0$:

```
LOOP:        POP(PD1,V); PUSH(PD2,V);
             IF EMPTYPDS(PD1) THEN GO TO EMPTYJ;    repeated p_j-1
             POP(PD1,V); PUSH(PD2,V);                       times

                .
                .
                .

             IF EMPTYPDS(PD1) THEN GO TO NONEMPTYJ
               ELSE GO TO LOOP;
EMPTYJ:        UNTIL EMPTYPDS(PD2) DO
                   BEGIN POP(PD2,V); PUSH(PD1,V) END;
               (Thus restoring PD1.)
               <S_1>;
               GO TO OUT;
NONEMPTYJ:     UNTIL EMPTYPDS(PD2) DO
                   BEGIN POP(PD2,V); PUSH(PD1,V) END;
               (Thus restoring PD1.)
               <S_2>;
OUT:
                .  .
                .  .
                .  .
```

<u>Pop from j^{th} pds</u>: We first test the j^{th} pds for emptiness and do nothing if it is empty. Otherwise, the behavior is analogous to that for the push; we take the j^{th} element of PD1 as the one desired, percolate the $(3+j)^{th}$ element to the j^{th} position, etc., and divide $|PD1|$ by p_j. The division is accomplished by starting C from the top of PD1 and moving it downward. At each move we cut off p_j-1 elements from the bottom of PD1 by appropriate pouring manipulation. We terminate when C reaches the bottom of the shrinking stack.

To construct S' given S, we simply duplicate the body of S and substitute for each PUSH, POP and bottom test the code described above. That the scheme so created mimics S should be clear from the construction.

Hence, since we have shown $P_{(3b,0)} \leq P_{(2b,0)C}$ and $P_{(3b,0)} \leq P_{(2b,0)C}$ effectively, the theorem is established.

Q.E.D.

§5. Schemes and Integer Arithmetic.

In this section we will investigate the power of some classes of schemes whose control structures have been augmented by the ability to do integer arithmetic. Accordingly, we allow the statements $V \leftarrow 0$, $V \leftarrow V + 1$, $V \leftarrow V \dot{-} 1$ and the conditional statement IF $V = 0$ THEN $<S_1>$ ELSE $<S_2>$. We leave it to the reader to show how more complicated statements, such as $V_i \leftarrow V_j$ or $V_i \leftarrow V_j \times V_k$ or indeed any computable function over the integers can be built up from these primitive statements. The formal definition of this new class $P_{\mathcal{N}}$ is given in (4.9) of [1].

It has already been shown in Theorem 10.10 of [1] that the class $P_{(1,0)\mathcal{N}}$ is universal in the sense of being effectively equivalent to the classes P_{Ae}, $P_{(3b,0)}$, etc. In the remaining portions of this section we characterize the class $P_{\mathcal{N}}$ and find that it partially overlaps the classes P_R and $P_{(2,0)}$ but is properly contained in $P_{(2b,0)}$. The reason for this rather unusual property is the immense power of the control structure of a $P_{\mathcal{N}}$ scheme (indeed, we have enough power to simulate an arbitrary Turing machine) coupled with the restriction of a fixed number of locations for computing results over the output domain.

Our basic tool here involves the functional Evalcutset first described in [4]:

Evalcutset (P,L,R,H,x) = <u>if</u> P(x) <u>then</u> x <u>else</u>
H(Evalcutset(L(x)), Evalcutset(R(x))) .

Intuitively, Evalcutset does the following:

1) Examines the infinite binary tree formed by the monadic functions L and R operating on the value input in x;

2) Finds the (unique) minimal cutset of this tree such that all nodes in the cutset make P true;

3) Treats the portion of the tree above and including this cutset as a description of an arithmetic expression in H, L, R and x;

4) Evaluates the expression so defined.

In [4] it was shown that Evalcutset cannot be computed using a fixed number of variables because an unbounded number of temporary results will in general be necessary for this computation. This then implies that Evalcutset cannot be computed in $P_{\aleph}$ and furthermore implies that any functional which requires an evaluation of Evalcutset independent of the other inputs to the functional cannot be computed in $P_{\aleph}$ either.

We will find the following fact concerning monadic functions useful.

(5.1) <u>Theorem</u>. <u>Consider the restriction of schemes to monadic functions only (predicates may have any rank)</u>. <u>Then</u> $P_{\aleph} \equiv P_{Ae}$.

<u>Proof</u> Clearly $P_{\aleph} \leq P_{Ae}$. Consider a scheme S in P_{Ae} , and construct an equivalent effective functional F (Theorem 10.6 of [1]). Since all functions are monadic, all expressions in

the computations of F have the form $f_1(f_2(\ldots(f_j(x)\ldots)$ where the f_i are function names and x is an input variable. Any expression can thus be evaluated using one variable. Any proposition $P(e_1,\ldots,e_n)$ can hence be evaluated using n variables. By Corollary 10.9 of [1] we can construct an equivalent scheme in $P_{\aleph}$.

Q.E.D.

In order to characterize $P_{\aleph}$, we now introduce 6 functionals, each using the monadic predicate P, the monadic functions L and R, the dyadic function H and the input variables w,x,y,z .

$$f_1 = \begin{cases} w & \text{if Evalcutset(P,L,R,H,w) is defined,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_2 = \begin{cases} \text{Evalcutset(P,L,R,H,w)} & \text{if Evalcutset(P,L,R,H,w) is defined,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_3 = \begin{cases} w & \text{if } P(x) \wedge \neg P(y) \wedge \text{Leaftest(P,L,R,z) is defined,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_4 = \begin{cases} \text{Evalcutset(P,L,R,H,w)} & \text{if } P(x) \wedge \neg P(y) \wedge \text{Leaftest(P,L,R,z) is defined,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_5 = \begin{cases} w & \text{if Leaftest(P,L,R,z) is defined,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

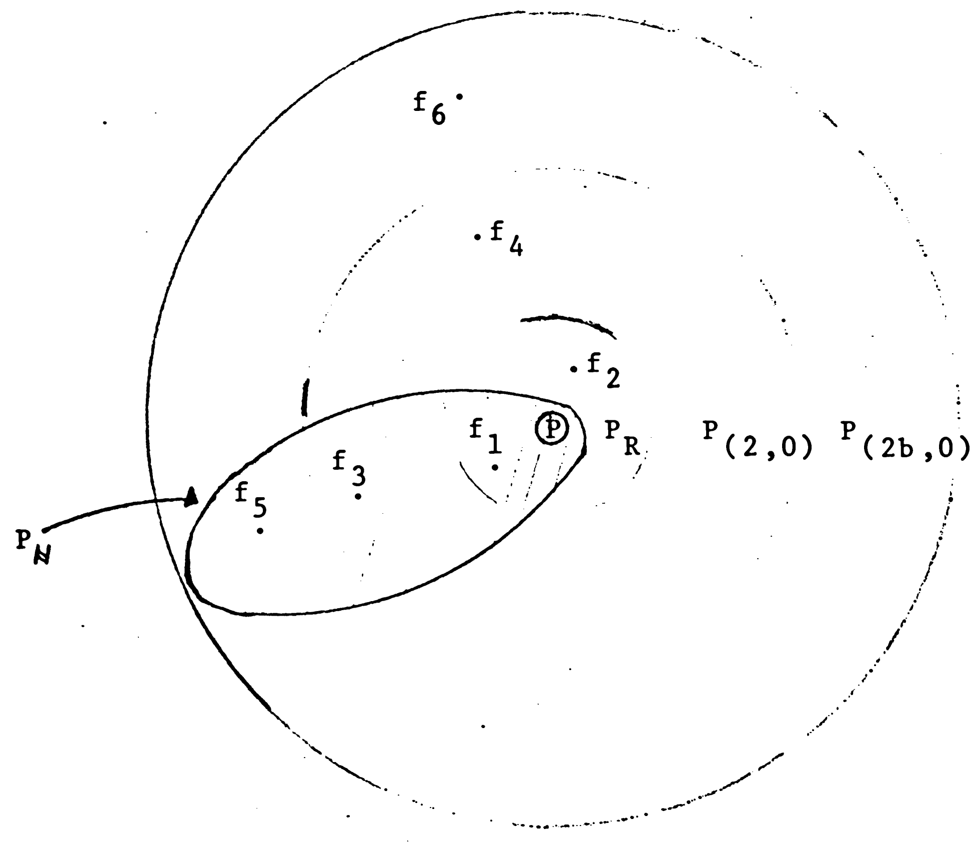
$$f_6 = \begin{cases} \text{Evalcutset(P,L,R,H,w)} & \text{if Leaftest(P,L,R,x) is defined,} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Clearly, f_2, f_4 and f_6 cannot be computed in $P_{\aleph}$ since each of them must conditionally evaluate Evalcutset which needs an unbounded number of variables. On the other hand, consider functionals f_1, f_3, and f_5. We don't have to evaluate Evalcutset; we just have to know whether it is <u>defined</u>, and we can tell this by the following functional:

Evalcutsetdef = <u>if</u> P(x) <u>then</u> true
<u>else</u> Evalcutsetdef(L(x)) <u>and</u> Evalcutset(R(x))

Evalcutsetdef and Leaftest can both be programmed in P_{Ae} using only monadic functions, and hence, by Theorem 5.1 can be computed in $P_{\aleph}$. Hence, f_1, f_3, and f_5 are computable in $P_{\aleph}$.

We now exhibit a Venn diagram of the classes P, P_R, $P_{(2,0)}$, $P_{(2b,0)}$, $P_{\aleph}$ and place each of the functionals f_i in the most restrictive class which permits its computation:



(5.2) Lemma. f_1 and f_2 can be computed in P_R .

Proof Obvious programming exercise.

(5.3) Lemma. f_3 and f_4 can be computed in $P_{(2,0)}$ but not in P_R .

Proof The test $P(x) \wedge \neg P(y)$ gives us the necessary values with which to simulate markers. Once we have markers we essentially have a universal scheme. Finally, Theorem 3.5 shows why neither f_3 or f_4 can be computed in P_R .

(5.4) Lemma. f_5 and f_6 can be computed in $P_{(2b,0)}$ but not in $P_{(2,0)}$.

Proof It should be clear that given a $P_{(2,0)}$ scheme to compute either f_5 or f_6 we can modify it to produce a $P_{(2,0)}$ scheme which computes Leaftest. However, this is impossible by Lemma 3.3. Hence, neither f_5 nor f_6 is $P_{(2,0)}$ computable. On the other hand, since Leaftest is $P_{(2b,0)}$ computable and thus furnishes us with any necessary markers, we conclude that both f_5 and f_6 are $P_{(2b,0)}$ computable.

The final result needed to complete the above Venn diagram is

Theorem. $P_{\aleph} < P_{(2b,0)}$

Proof The inclusion follows as a corollary to the proof of Lemma 4.5. The two pds's are used as counters holding the

Gödelized contents of the variables of the $P_{\mathcal{N}}$ scheme.

The fact that the inclusion is proper follows from the fact that f_6 is not $P_{\mathcal{N}}$ computable.

Q.E.D.

One final comment is in order here, namely that the Husearch functional of Section 4 is not $P_{\mathcal{N}}$ computable. The proof is left to the reader.

§6. Multi-dimensional Arrays.

Let us allow the use of an n-dimensional array A, $n > 1$. We use the obvious interpretation that $A[w_1,\ldots,w_n]$ is the same variable as $A[v_1,\ldots,v_n]$ if and only if $w_i \ominus v_i$ for $i = 1,\ldots,n$ (see Definition 4.5 of [1]). The main result of this section is that adding n dimensional arrays in P_A does not change the power of the class:

(6.1) Theorem. *Let S be a scheme in P_A, with the addition of n-dimensional arrays. We can construct an equivalent scheme S' in P_A.*

Proof We show how to construct in P_A both a locator and a P-simulator for S. We combine these as described in the proof of Theorem 1.5 to form S' in P_A equivalent to S.

The locator is constructed as was the locator for a P_{AM} scheme in showing that $P_A \equiv P_{AM}$ (Theorem 9.14 of [1]). The locator is shown in (1.6), where the statements $S_1,\ldots,S_{2^n}$ have to be constructed.

If the v_i-autonomous behavior of S is finite, we construct S_i as in 9.9 of [1]. Note that since the behavior of S is finite, S_i will reference only a finite set of variables, and we can change all the referenced variables to simple variables. Thus S_i will clearly be in P_A.

If the v_i-autonomous behavior of S is infinite, we construct S_i as described in 9.11 of [1], and S_i is a statement of P_A.

Now not. that, given S in P_A using multi-dimensional arrays, we can <u>effectively</u> decide whether the v-autonomous behavior of S is finite or infinite. Suppose S contains p statements. Begin recording the v-autonomous behavior; if p + 1 labels are recorded, there is a loop and the behavior is infinite. (This is the same process as deciding whether a scheme in P_A has finite or infinite v-autonomous behavior — Theorem 9.5 of [1]). Hence, we can <u>effectively</u> construct the locator.

To construct the P-simulators, consider scheme S to be in P_{Ae} with multi-dimensional arrays. Lemma 6.2 below shows how to construct an equivalent scheme S' in P_{Ae} using only one-dimensional arrays. Using Theorem 8.8 of [1] we can construct an equivalent scheme S'' in P_{AM} . Finally we use Theorem 5.4 of [1] to construct P-simulators in P_A for S'' and hence for S.

Q.E.D.

(6.2) <u>Lemma</u>. <u>Let</u> S <u>be a scheme in</u> P_{Ae} <u>which in addition uses an n-dimensional array</u> A, $n > 1$. <u>There exists an equivalent scheme</u> S' <u>in</u> P_{Ae} <u>which uses</u> $n + 1$ <u>one-dimensional arrays</u> $B_0, B_1, \ldots, B_n$ <u>in place of A</u>.

<u>Proof</u> In addition to the arrays, S' uses an additional variable I, whose purpose is to indicate how many different elements A[...] (in S) have been assigned values. If in S, $A[w_1, \ldots, w_n]$ has been assigned a value v, then in S' for some j we have

$$B_1[j] = w_1, \ldots, B_n[j] = w_n, B_0[j] = v .$$

Let J be a new variable, and let COPY[J,w_1,...,w_n] stand for the statement

```
BEGIN J ← 0;
      UNTIL I ⊜ J DO
            BEGIN IF B1[J] ⊜ w1  and ... and  Bn[J] ⊜ wn
                  THEN GOTO FOUND;
                  J ← J + 1
            END;
      B1[J] ← w1; ...; Bn[J] ← wn;
      B0[J] ← OMEGA; I ← I + 1;
   FOUND;
END;
```

This statement performs a linear search for an index J such that $B_1[J] = w_1, \ldots, B_n[J] = w_n$. If found then $A[w_1,\ldots,w_n]$ in S is the location $B_0[j]$ in S'. If not found, it is added.

Now, to translate S into S', we

(1) Add the following statement to the beginning of S: I ← 0;

(2) Transform S so that the only reference to the array A are in statements

$$A[w_1,\ldots,w_n] \leftarrow v \quad \text{and} \quad v \leftarrow A[w_1,\ldots,w_n]$$

where w_i and v are simple variables.

(3) Change each statement $A[w_1,\ldots,w_n] \leftarrow v$ to

BEGIN COPY[J,w_1,...,w_n]; $B_0[J] \leftarrow v$ END

(4) Change each statement $v \leftarrow A[w_1,\ldots,w_n]$ to

BEGIN COPY[J,w_1,...,w_n]; $v \leftarrow B_0[J]$ END

Q.E.D.

References

[1] Constable, Robert L. and David Gries. On Classes of Program Schemata, SIAM Journal on Computing, 1, 1 (1972).

[2] Gries, David J. Programming by Induction, Information Processing Letters, 1 (1972), p. 100-107.

[3] Hopcroft , John E. and J.D. Ullman. Formal Languages and Their Relation to Automata. Addison-Wesley, London 1969.

[4] Paterson, M.S. and C.E. Hewitt. Comparative Schematology, Conference Record of Project MAC Conference on Concurrent Systems & Parallel Computation, ACM, New York, 1970, p. 119-128.

Appendix

We give here the details of the proofs and constructions in some of the lemmas and theorems. Refer to the proper lemma in the paper for discussion.

(2.2) Lemma. $P_{(1,n)} \leq P_{RgM}$ for $n \geq 0$.

Step 1. Given S in $P_{(1,n)}$, create S1 equivalent to S as follows. For each statement PUSH(P,v) or POP(P,v) in S, generate a new label L and replace the statement by

```
      BEGIN PUSH(P,v); L: END
or    BEGIN POP(P,v);  L: END
```

For each statement HALT(v) , generate a new label L and replace the statement by L: HALT(v) . Hence, each PUSH and POP is followed by a labeled null statement, and each HALT is labeled. Let these new unique labels be called $L_1, L_2, \ldots, L_\ell$.

Step 2. Create S2 $\equiv$ S1 as follows. Let S1 have the form

$$(v,\ldots,v):\ S_1;\ S_2;\ \ldots;\ S_n$$

and suppose it uses simple variables $V_1,\ldots,V_k$, then S2 is the scheme

$$\begin{array}{rl} (v,\ldots,v): & S_1;\ S_2;\ \ldots;\ S_n \\ F(P,X): & \underline{\text{global}}\ V_1,\ldots,V_k; \\ & S_1;\ S_2;\ \ldots;\ S_n \end{array}$$

Note that the same labels are used in both the main scheme and in F. However, labels are local to the function in which they are used, and jumps out of functions are not allowed.

Step 3. Create $S3 \in P_{RgM}$, $S3 \equiv S2$. In S3, PUSH statements in S2 are replaced by calls on F; POPs and HALTs within F are replaced by returns; and POPs in the main scheme are deleted. The pds P is now a parameter variable of F.

(a) Let $\underline{L}_1,\ldots,\underline{L}_\ell$ be new unique markers corresponding to the labels $L_1,\ldots,L_\ell$ introduced in Step 1. Insert just before statement S_1 of F the sequence

$$\text{IF } X = \underline{L}_1 \text{ THEN GO TO } L_1;$$
$$\vdots$$
$$\text{IF } X = \underline{L}_\ell \text{ THEN GO TO } L_\ell;$$

(b) Change each PUSH statement BEGIN PUSH(P,v); L_i: END to BEGIN $X \leftarrow F(v,\underline{L}_i)$;

$$\text{IF } X = \underline{L}_1 \text{ THEN GO TO } L_1;$$
$$\vdots$$
$$\text{IF } X = \underline{L}_\ell \text{ THEN GO TO } L_\ell;$$

L_i: END

(c) Change each statement

L_i: HALT(V) within F to L_i: HALT($\underline{L}_i$) .

Note that this construction causes all HALTs of the scheme to result in an empty pds at termination.

(d) Change each POP statement BEGIN POP(P,v); L_i: END within F to

BEGIN v ← P; HALT($\underline{L}_i$); L_i: END

(e) Replace each POP statement BEGIN POP(P,v); L_i: END within the main scheme by BEGIN L_i: END . (Within the main scheme the pds is empty, and POP is a null instruction.)

Q.E.D.

(2.7) Lemma. Every scheme S in P_{RgM} has a locator S' in P.

Proof The locator S' for S has the form (1.6) and we must only show how to construct the statements $S_1, \ldots, S_{2^n}$ described there. We outline only the construction of S_1 , which simulates the v-autonomous behavior of S where v = (true,...,true) , as described by (1.7). We rely on the notation and results of Lemma 2.6.

The first phase is to construct the v-autonomous behavior of S as described in (9.10) of [1], with the following changes and additions:

(1) With each label L_i of the behavior, keep (a) the statement it labels; (b) an indication of which function execution it occurs in (not only the function, but which call of the function it is); (c) the current values $(\bar{v}, M_1, \ldots, M_{m-1})$ of the global variables; and (d) the current values of the local variables of the function (or main program).

(2) If a label is added which already occurs in the behavior for this particular function execution (say at position j), and if the values of the

global and local variables are the same, then the behavior is infinite. Stop building the behavior and record with this last label the position j.

(3) After a label L_i: $v \leftarrow f(\ldots)$ (where f is a recursive function) is added, perform the following. Check back to see if a call of f with the same argument values (not variables) and global values has occurred and is not yet finished (say at position j of the behavior). If so, the behavior is infinite. Stop building the behavior and record with this last label the position j.

If no such previous call has occurred, then before proceeding expand the call $v \leftarrow f(\ldots)$ in S as described in (3.7) of [1]. This of course makes the call $v \leftarrow f(\ldots)$ superfluous, and it will be deleted later.

The second phase is to construct the statement S_1 from the behavior constructed in Step 1. This behavior is of course just the partial behavior if we stopped building the behavior via Steps 2 or 3 above. (Because of Lemma 2.6 the construction must stop eventually, either with a HALT or by Steps 2 or 3.)

Construct S_1 from the behavior with the following changes:

(1) Replace statements L_i: $v \leftarrow f(\ldots)$, where f is a recursive procedure, by null statements.

(2) If the construction of the behavior was stopped by (2) of phase 1, then generate a new label L, prefix it to the j^{th} substatement within S_1

(using the j of (2) of phase 1), and replace the last label of the behavior by GO TO L .

(3) If the construction of the behavior was stopped by (3) of phase 1, then generate a new label L and prefix it to the j^{th} substatement of S_1 (using the j of (3) of phase 1). Suppose this call at the j^{th} position was originally $v \leftarrow f(v_1,\ldots,v_n)$ and suppose the last labeled statement of the behavior is $w \leftarrow f(w_1,\ldots,w_n)$. Then generate new variables $V_1,\ldots,V_n$ and add to S_1 the statements

$$V_1 \leftarrow w_1; \ \ldots; \ V_n \leftarrow w_n;$$
$$v_1 \leftarrow V_1; \ \ldots; \ v_n \leftarrow V_n; \quad \text{GO TO L;}$$

Q.E.D.

(2.8) <u>Lemma</u>. $P_{RgM} \leq P_{RM}$, $P_{Rg} \leq P_R$.

<u>Proof</u> Suppose scheme $S \in P_{RgM}$ has function definitions for functions $F_1,\ldots,F_n$, and suppose that the variables used globally are $V_1,\ldots,V_m$. By suitably renaming the local variables we can make sure that $V_1,\ldots,V_m$ are used only as global variables, and hence we can assume S has the form

$$(2.9)\quad \begin{array}{ll} (v,\ldots,v): & \text{<S>}; \ \ldots; \ \text{<S>} \\ F_1(v,\ldots,v): & \underline{\text{global}}\ V_1,\ldots,V_m; \ \text{<S>}; \ \ldots; \ \text{<S>} \\ \vdots & \qquad\qquad\qquad\qquad\qquad \vdots \\ F_n(v,\ldots,v): & \underline{\text{global}}\ V_1,\ldots,V_m; \ \text{<S>}; \ \ldots; \ \text{<S>} \end{array}$$

We give a construction which reduces by one the number of global variables. If the construction is executed m times we arrive at an equivalent scheme in P_{RM} . Let us now show how to eliminate the need for V_1 to be global.

Note that execution of a function F_i may change the value of V_1. We must therefore find a way of transmitting this change back to the calling function or main program. To do this, we replace each call of F_i by two calls; one call to F_i returns the normal value, and the second call to a new function $\underline{F}_i$. $\underline{F}_i$ executes exactly the way F_i does, but just returns a different value.

Step 1. For each function definition F_i in (2.9) insert a new function definition

$$\underline{F}_i(v,\ldots,v);\ \ \underline{\text{global}}\ \ V_1,\ldots,V_m;\ \ \langle S\rangle;\ \ldots;\ \langle S\rangle$$

where $\underline{F}_i$ looks exactly like F_i except that each HALT(w) has been replaced by HALT(V_1) . The resulting scheme S1 is equivalent to S, since all we have done is add function definitions.

Step 2. Replace S1 by the following scheme S2 :

$$\begin{array}{rlll}
(v,\ldots,v): & \langle S\rangle;\ \ldots;\ \langle S\rangle & & \\
F_1(v,\ldots,v,V_1): & \underline{\text{global}} & V_2,\ldots,V_m; & \langle S\rangle;\ \ldots;\ \langle S\rangle \\
\vdots & & \vdots & \vdots \\
F_n(v,\ldots,v,V_1): & \underline{\text{global}} & V_2,\ldots,V_m; & \langle S\rangle;\ \ldots;\ \langle S\rangle \\
\underline{F}_1(v,\ldots,v,V_1): & \underline{\text{global}} & V_2,\ldots,V_m; & \langle S\rangle;\ \ldots;\ \langle S\rangle \\
\vdots & & \vdots & \vdots \\
\underline{F}_n(v,\ldots,v,V_1): & \underline{\text{global}} & V_2,\ldots,V_m; & \langle S\rangle;\ \ldots;\ \langle S\rangle
\end{array}$$

S2 executes as S1 does, except for the fact that if during execution of a function V_1 is changed, this change is not transmitted back to the variable V_1 local to the point of call. The final Step 3 translates S2 into S3 where S3 is equivalent to S1 and thus S.

Step 3. Each of the functions F_i and $\underline{F}_i$ and the main program uses new variables $V_0, \underline{V}_2, \ldots, \underline{V}_n$ which are <u>local</u> to the function or main program. Replace each call

$$w \leftarrow F_i(v_1, \ldots, v_n, V_1)$$

by

BEGIN	$\underline{V}_2 \leftarrow V_2; \ \ldots; \ \underline{V}_n \leftarrow V_n;$	(Save global values)
	$V_0 \leftarrow F_i(v_1, \ldots, v_n, V_1);$	(Call F_i to get normal result into V_0)
	$V_2 \leftarrow \underline{V}_2; \ \ldots; \ V_n \leftarrow \underline{V}_n;$	(Restore global values)
	$V_1 \leftarrow \underline{F}_i(v_1, \ldots, v_n, V_1);$	(Call $\underline{F}_i$ to execute as F_i did but return the value of V_1)
	$w \leftarrow V_0;$	(Put result into variable w)
END		

Q.E.D.