STANFORD ARTIFICIAL INTELLIGENCE
MEMO AIM-185

STAN-CS-73-333

ON THE POWER OF PROGRAMMING FEATURES

BY

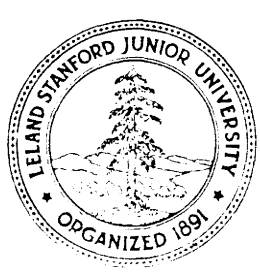
ASHOK K. CHANDRA

ZOHAR MANNA

SUPPORTED BY
NASA CONTRACT NSR 05-020-500
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

JANUARY 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-185

January 1973

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-73-333

ON THE POWER OF PROGRAMMING FEATURES

by

Ashok K. Chandra

Zohar Manna

ABSTRACT: We consider the power of several programming features such as counters, pushdown stacks, queues, arrays, recursion and equality. In this study program schemas are used as the model for computation. The relations between the powers of these features is completely described by a comparison diagram.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract No. SD-183, and by NASA contract NSR 05-020-500.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

Introduction

In this paper we consider the problem of comparing the power of several features used in programming languages. For example, it is intuitively obvious to any programmer that recursion cannot, in general, be replaced by iteration with variables alone, but recursion can always be replaced by a pushdown stack. This indicates that a pushdown stack is at least as powerful as recursion, and that recursion is more powerful than iteration. Thus, from the iteration-vs-recursion standpoint we would say that ALGOL and PL/1 are more powerful than FORTRAN. The question is whether an intuitive notion of this kind can be understood in a formal-way, and possibly elaborated upon to obtain a better understanding of programming features and to enable us to compare their power.

Unfortunately, the problem is not so simple. Consider, for example, the programming language of flowcharts, which contain ideal integer variables, i.e., their values can be arbitrarily large. .The operations allowed in the flowchart are incrementing and decrementing variables by one, and testing to see if the value of a variable is zero. Such a simple language with just three variables can calculate all the "computable" functions, that is, all the partial recursive functions over the natural numbers. Thus if we add recursion or a pushdown stack to such a language, the power of the language will not be increased.

This suggests that in order to carry out such a study, we must isolate in some way the effect of the programming features, whose power we wish to compare, from the values being computed by the program. For this purpose we consider for each programming language a class of program schemas; a program schema may use the control features of the language

but the basic operations (constants, functions, and predicates) are used only as symbols without being specified.

Related work has been done previously, among others, by Paterson and Hewitt [1970], Garland and Luckham [1971], Constable and Gries [1972], Plaisted [1972] and Chandra and Manna [1972]. The classes of schemas considered in these papers are not identical to ours, but the differences are not significant. Details of the results presented in this paper can be found in Chandra's thesis [1973].

Part I. The Class of Program Schemas

A program schema is a program in which the data domain is not specified. In addition, the constants are indicated simply by the symbols $a_1, a_2, \ldots$, the functions by $f_1, f_2, \ldots$, and the predicates by $p_1, p_2, \ldots$. Thus a program schema may be thought of as representing a family of real programs. A real program of the family is obtained by providing an interpretation for the symbols of the program schema, i.e., specifying a data domain and specifying data elements, functions and predicates over the domain for the symbols a_i, f_i and p_i, respectively.

In our program schemas we use two kinds of variables: data variables, denoted by $y_1, y_2, \ldots$, and boolean variables, denoted by $z_1, z_2, \ldots$. Boolean variables can have value either true or false. Data variables, on the other hand, have values from the data domain that is specified along with an interpretation for the schema. Correspondingly, we distinguish between two types of terms: data terms and boolean terms. A data term τ can be built up using the data variables y_i of the schema and the individual constants a_i, and applying the

function symbols f_i to them. The value of a data term for a given interpretation is always a data element. A boolean term α is an atomic formula or a negated atomic formula, where an atomic formula is a boolean value (true or false), a boolean variable z_i, or a predicate test of the form $p(\tau_1, \ldots, \tau_k)$. Under any interpretation, the value of α is a boolean value, true or false.

1. Simple Algol-like Schemas

The first class of schemas we consider is the class of Algol-like schemas which--can be constructed from statements of the following form (we use standard Algol-like notations):

(i) start statement: START(a)

(ii) halt statement: HALT(τ)

(iii) loop statement: LOOP

(iv) assignment statements: $y_i \leftarrow \tau$ or $z_i \leftarrow \alpha$

(v) test statement: if α then goto L_1 else goto L_2.

L_1 and L_2 here are labels. In addition we may use begin . . . end for grouping statements.

The start statement, START(a) , initializes all data variables y_i to the value a and all boolean variables to true . The halt statement, HALT(τ) , outputs the data value of the term τ . The loop statement, LOOP , causes the schema to loop forever.

We use $\mathcal{C}()$ to denote the class of all simple Algol-like schemas.

2. Augmented Algol-like Schemas

We will also consider Algol-like schemas augmented with features designed to make the schemas more powerful.

(a) Counters

A counter is a variable whose value is always a non-negative integer. Counters are denoted by $c_1, c_2, \ldots$. All counters used by a schema are initialized to zero by the start statement. The statements allowed on an arbitrary counter c are:

(1) $c \leftarrow c+1$

(2) if c = 0 then goto L_1 else begin c $\leftarrow$ c-1; goto L_2 end.

We use $\mathcal{C}(c)$ to denote the class of Algol-like schemas with counters (it includes the subclass of schemas with no counters), $\mathcal{C}(1c)$ to denote the class of schemas with at most 1 counter, and $\mathcal{C}(2c)$ to denote the class with at most 2 counters.

(b) Pushdown Stack

A pushdown stack is a last-in first-out store in which a pair of values of both types (data, boolean) can be stacked. Pushdown stacks are denoted by $s_1, s_2, \ldots$. All pushdown stacks used by a schema are initialized to be empty by the start statement. A schema with a stack

can "push" a data value and a boolean value into the stack, and it can "pop" them (if the stack is non-empty).

The statements allowed on an arbitrary pushdown stack s are:

(1) push(s,y, z)

(2) if s = A then goto L_1

else begin pop(s,y,z); goto L_2 end

Here, y denotes an arbitrary data variable, z a boolean variable, and A the empty stack. The statement " push(s,y,z) " adds the current values of the variables y,z on top of the stack s . The statement " pop(s,y,z) " does the opposite: the one data and one boolean value at the top-of the stack s are assigned to the variables y and z , respectively, and these two values are removed (popped) from the stack.

We use $\mathcal{C}(s)$ to denote the class of Algol-like schemas with pushdown stacks, and similarly for $\mathcal{C}(1s)$ and $\mathcal{C}(2s)$.

(c) Queues

A queue is a first-in first-out store. Queues are denoted by $q_1, q_2, \cdots$. All queues used by a schema are initialized to be empty by the start statement. A schema with a queue can "add" values at one end, and "remove" them from the other. The statements allowed on an arbitrary queue q are:

(1) add(q,y, z)

(2) if q = A then goto L_1

else begin remove(q,y,z); goto L_2 end .

The statement " add(q,y,z) " adds the current values of the variable y, z at one end of the queue. The statement " remove(q,y,z) " does the

following: the one data and one boolean value at the end of the queue are assigned to the variables y and z , respectively, and these two values are removed from the queue.

We use $\mathcal{C}(q)$ to denote the class of Algol-like schemas with queues.

(d) Arrays

An array is a semi-infinite sequence of "locations" (numbered 0,1,2, . . .), each of which can take on a pair of values: one data value and one boolean value. Arrays are denoted by $A_1, A_2, \ldots$. The start statement, START(a) , initializes all locations in arrays to the data value a and the boolean value true . A location can be accessed by subscripting the array with a counter. The statements allowed on an arbitrary array A are:

(1) $A[c] \leftarrow (y, z)$

(2) $(y,z) \leftarrow A[c]$.

We use @(A) to denote the class of schemas with arrays. Note that the use of an array implies the use of counters, that is, schemas in @(A) do have an arbitrary number of counters.

The class of Algol-like schemas with any or all these features (counters, stacks, queues, arrays) is denoted by $\mathcal{C}(s,q,A)$.

3. Recursive Schemas

A recursive schema consists of a set of recursive definitions of the following form:

$F_1(a,a,\ldots,\underline{\text{true}},\underline{\text{true}},\ldots)$ where

$F_1(\bar{y}_1,\bar{z}_1) \Leftarrow \text{if } \alpha_1(\bar{y}_1, \bar{z}_1,\bar{F}) \text{ then } \tau_1(\bar{y}_1,\bar{z}_1,\bar{F}) \text{ else } \tau_1'(\bar{y}_1, \bar{z}_1,\bar{F})$

$\vdots$

$F_n(\bar{y}_n,\bar{z}_n) \Leftarrow \text{if } \alpha_n(\bar{y}_n,\bar{z}_n,\bar{F}) \text{ then } \tau_n(\bar{y}_n,\bar{z}_n,\bar{F}) \text{ else } \tau_n'(\bar{y}_n,\bar{z}_n,\bar{F})$,

where $\bar{y}_i$ represents a vector of data variables, $\bar{z}_j$ a vector of boolean variables, and $\bar{F} = (F_1,\ldots,F_n)$ is a vector of "defined function::". Each defined function F_i may take both data values and boolean values as arguments but, for simplicity, we assume that it always returns just one data value. $\alpha_i(\bar{y}_i, \bar{z}_i,\bar{F})$ is a boolean term and $\tau_i(\bar{y}_i,\bar{z}_i,\bar{F})$ and $\tau_i'(\bar{y}_i,\bar{z}_i,\bar{F})$ are data terms that may use the variables in $\bar{y}_i$ and $\bar{z}_i$, and the defined functions $\bar{F}$ along with the constant symbols $a_1,a_2,\ldots$, the function symbols $f_1,f_2,\ldots$, and the predicate symbols $p_1,p_2,\ldots$.

The value of the schema for any given interpretation is the value of F_1 withall its data arguments set to the value of the individual constant a , and all its boolean arguments set to <u>true</u>. During computation, all arguments are passed by value, i.e., the innermost function calls are evaluated first. Note that there are no "global" variables, and function calls cannot have any side effects, they simply return values.

We use $\mathcal{C}(R)$ to denote the class of all recursive schemas.

4. Equality

We also consider schemas in which every boolean term α may have the form $\tau_1 = \tau_2$ or $\tau_1 \neq \tau_2$ in addition to the earlier possibilities.

When equality is allowed in a class $\mathcal{C}(\ldots)$, we denote the augmented class by $\mathcal{C}(\ldots,=)$. Thus, we use $\mathcal{C}(=)$ to denote the class of Algol-like schemas with equality, $\mathcal{C}(c,=)$ to denote the class of Algol-like schemas with counters and equality, $\mathcal{C}(R,=)$ to denote the class of recursive schemas with equality, etc.

5. Example

Any two schemas S and S' are said to be equivalent if for every interpretation of S and S' ,[*] either both schemas diverge (i.e., loop forever), or both halt with the same output.

Consider the following recursive schema

S_0: F(a) where

F(y) <= if P(Y) then Y else f(y,F(g(y))) .

Note that if we have an interpretation of S_0 for which

$p(g^n(a))$ = true for some $n \geq 0$, and

$p(g^i(a))$ = false for all $i < n$,

then

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

Below we exhibit some Algol-like schemas that are equivalent to S_0 . To simplify the programs we use an extended Algol-like language, using regular while . ..do . . . statements, goto statements and if . . . then . . . else . . . statements. All these statements can be expressed easily in terms of our primitive statements. We allow also the statement $c_0 \leftarrow c_1$ which can be replaced by legal statements for counters by adding one additional counter.

For clarity, we add a few comments in the schemas below. Since boolean variables play no role in this example, we ignore their presence in the comments.

---

[*] i.e., the interpretation includes an assignment to all constant, function and predicate symbols occurring in S or in S' .

(a) A simple schema

S_1: START(a);

while $\neg\, p(y_1)$ do $y_1 \leftarrow g(y_1)$;

[comment: $y_1 = g^n(a)$ }

L: if $p(y_4)$ then HALT(y_1)

else begin $y_2 \leftarrow a$; $y_4 \leftarrow g(y_4)$; $y_3 \leftarrow y_4$ end;

{comment: in the i-th loop ($1 \leq i \leq n$)

$y_2 = g^0(a)$, $y_3 = g^i(a)$, $y_4 = g^i(a)$}

while $\neg\, p(y_3)$ do begin $y_2 \leftarrow g(y_2)$; $y_3 \leftarrow g(y_3)$ end;

{comment: in the i-th loop ($1 \leq i \leq n$)

$y_2 = g^{n-i}(a)$, $y_3 = g^n(a)$, $y_4 = g^i(a)$}

$y_1 \leftarrow f(y_2, y_1)$;

got0 L .

(b) A schema with counters

S_2: START(a);

while $\neg\, p(y_1)$ do begin $y_1 \leftarrow g(y_1)$; $c_1 \leftarrow c_1+1$ end;

(comment: $y_1 = g^n(a)$, $c_1 = n$}

L: if $c_1 = 0$ then HALT(y_1)

else begin $y_2 \leftarrow a$; $c_1 \leftarrow c_1-1$; $c_2 \leftarrow c_1$ end.;

while $c_2 \neq 0$ do begin $y_2 \leftarrow g(y_2)$; $c_2 \leftarrow c_2-1$ end;

{comment: in the i-th loop ($1 \leq i \leq n$)

$y_2 = g^{n-i}(a)$, $c_1 = n-i$]

$y_1 \leftarrow f(y_2, y_1)$;

got0 L .

(c) A schema with a pushdown stack

S_3: START (a) ;

while $\neg\, p(y_1)$ do begin push(s,y_1,z); $y1 \leftarrow g(y_1)$ end;

{comment: $y_1 = g^n(a)$, $s = (a,g(a),\ldots,g^{n-1}(a))$}

L: if $s = A$ then HALT(y_1) else pop(s,y_2,z);

[comment: in the i-th loop $(1 \leq i \leq n)$

$y_2 = g^{n-i}(a)$, $s = (a,g(a), \ldots,g^{n-i-1}(a))$}

$y_1 \leftarrow f(y_2,y_1)$;

goto L .

(d) A schema with an array

S_4: START(a);

while $\neg\, p(y_1)$ do begin $A[c\ 1 \leftarrow (y_1,z)$; $c \leftarrow c+1$; $y_1 \leftarrow g(y_1)$ end;

[comment: $y_1 = g^n(a)$, AIO] =a, A[1] $= g(a)$, . . .

$A[n-1] = g^{n-1}(a)$, $c = n$}

L: if $c = 0$ then HALT(y_1)

else begin $c \leftarrow c-1$; $(y_2,z) \leftarrow A[c]$ end;

(comment: in the i-th loop $(1 \leq i \leq n)$

$y_2 = g^{n-i}(a)$, $c = n-i$}

$y_1 \leftarrow f(y_2,y_1)$;

got0 L .

(e) A schema with equality

S_5: START(a) ;

while $\neg p(y_1)$ do $y_1 \leftarrow g(y_1)$,

$y_2 \leftarrow y_1$;

[comment: $y_1 = y_2 = g^n(a)$ }

L: if $y_2 = a$ then HALT(y_1) else $y_3 \leftarrow a$;

while $g(y_3) \neq y_2$ do $y_3 \leftarrow g(y_3)$;

$y_2 \leftarrow y_3$;

(comment: in the i-th loop $(1 \leq i \leq n)$

$y_2 = y_3 = g^{n-i}(a)$ }

-- $y_1 \leftarrow f(y_2, y_1)$;

goto L

Part II. On the Power of Classes of Schemas

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two classes of schemas. We say that

(a) $\mathcal{C}_1$ is more powerful than $\mathcal{C}_2$ (notation* $\mathcal{C}_1 \geq \mathcal{C}_2$) if for every schema in $\mathcal{C}_2$ there is an equivalent s c h e m a$_1$,

(b) $\mathcal{C}_1$ and $\mathcal{C}_2$ are equally powerful (notation: $\mathcal{C}_1 \equiv \mathcal{C}_2$) if $\mathcal{C}_1 \geq \mathcal{C}_2$ and $\mathcal{C}_2 \geq \mathcal{C}_1$, and

(c) $\mathcal{C}_1$ yis strict1 more powerful than $\mathcal{C}_2$ (notation $\mathcal{C}_1 > \mathcal{C}_2$) , if $\mathcal{C}_1 \geq \mathcal{C}_2$ but $\mathcal{C}_1 \not\equiv \mathcal{C}_2$.

1. The Comparison Diagram

We now consider the interrelations between the classes of schemas we have defined.

Intuitively, anything that can be done iteratively can also be done recursively. In other words, we would expect that $\mathcal{C}(R) \geq \mathcal{C}()$, and $\mathcal{C}(R,=) \geq \mathcal{C}(=)$ .That these are indeed true was shown by McCarthy [1962]. Also, as mentioned earlier, one expects that recursion is strictly more powerful than iteration. Paterson and Hewitt [1970] showed that there are certain recursive schemas for which there are no equivalent simple Algal-like schemas, i.e., $\mathcal{C}(R) > \mathcal{C}()$, and also $\mathcal{C}(R,=) > \mathcal{C}(=)$.

Another intuitive notion is that recursion can always be replaced by a pushdown stack. Thus, if our schemas in G(R) and $\mathcal{C}(ls)$ do capture the intuitive power of recursion and of a pushdown stack, we would expect that $\mathcal{C}(R) \leq \mathcal{C}(ls)$, and similarly, $\mathcal{C}(R,=) < \mathcal{C}(ls,=)$. These were shown to be true by Hewitt [1970] and by Constable and Cries [1972]. One should also ask whether a pushdown stack has power strictly greater than recursion, or whether they are equally powerful. To state this in another way, we observe that recursion involves the use of an implicit stacking mechanism. The question is whether or not this implicit stack really utilizes the full power of a pushdown stack. Chandra [1973] answered this by showing that $\mathcal{C}(R) \equiv \mathcal{C}(ls)$, and that $\mathcal{C}(R,=) \equiv \mathcal{C}(ls,=)$.*/

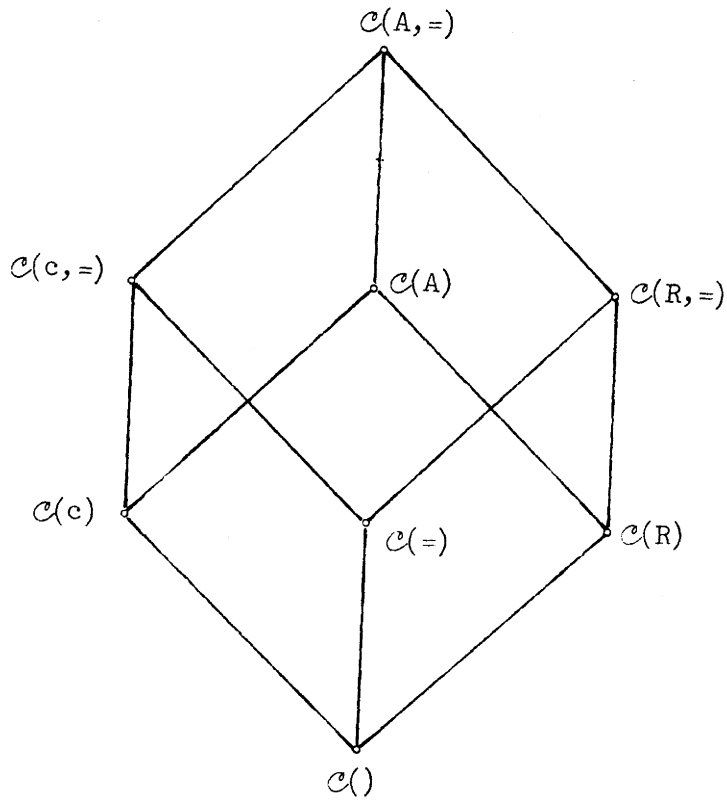
Paterson [unpublished memorandum] and Garland and Luckham [1971] showed that $\mathcal{C}(c) > \mathcal{C}(1c)$. Plaisted [1972] proved the surprising result that the addition of just one counter to simple Algol-like schemas adds no power, i.e., $\mathcal{C}(1c) \equiv \mathcal{C}()$. However, the addition of a second counter adds power, i.e., $\mathcal{C}(2c) > \mathcal{C}(1c)$; and after that, the addition of a third, fourth, fifth counter, etc., does not increase the power.

*/ It can be shown that the power of recursive schemas is not affected by the addition of features such as: (a) recursive definitions which consist of simple Algol-like programs with global variables and local variables as well as recursive calls, or (b) defined functions which return-not just one data value, but a vector of data and boolean values.

Constable and Gries [1972] introduced schemas with arrays and used a problem suggested by Paterson and Hewitt to show that $@(A) > \mathcal{C}(R)$. Chandra and Manna [1972] observed that the use of equality increases the power of schemas.

The interrelationships between the various classes of schemas is shown in Figure 1. In the figure (and all following figures), if there is an ascending arc (or a chain of such arcs) leading from a class $\mathcal{C}_1$ to a class $\mathcal{C}_2$, and $\mathcal{C}_2$ is above $\mathcal{C}_1$ in the figure, it means that "$\mathcal{C}_2$ is a strictly more powerful class than $\mathcal{C}_1$". If two classes, $\mathcal{C}_1$ and $\mathcal{C}_2$, nre not linked by an ascending chain of arcs, then the classes are unrelated, i.e., $\mathcal{C}_1 \not\leq \mathcal{C}_2$ and $\mathcal{C}_2 \not\leq \mathcal{C}_1$. For example, $\mathcal{C}(=) \not\leq \mathcal{C}(A)$, and $@(A) \not\leq \mathcal{C}(=)$. In other words, there is at least one schema in $\mathcal{C}(=)$ for which there is no equivalent schema in $@(A)$, and vice versa. Details of all the results suggested by Figure 1 can be found in Chandra's thesis [1973].

From Figure 1 it is apparent that schemas with arrays and equality act as a "maximal" class. In fact, any arbitrary schema with equality, counters, stacks, queues and arrays can be effectively translated into an equivalent schema With equality and one array. Also, one pushdown stack has the same power as recursion, but two stacks are strictly more powerful -- they are together as powerful as arrays. Even the seemingly "weaker" class with one pushdown stack and one counter has the same power as arrays. Observe that a queue is a more powerful feature than a stack; actually, a queue is as powerful as two stacks (addition of more stacks or queues adds no power).



$\mathcal{C}() \equiv \mathcal{C}(1c)$

$\mathcal{C}(c) = \mathcal{C}(2c)$

$\mathcal{C}(R) \equiv \mathcal{C}(1s)$

$\mathcal{C}(A) \quad \mathcal{C}(1s, 1c) \equiv \mathcal{C}(2s) \quad \mathcal{C}(1q) \quad \mathcal{C}(1A) \quad \mathcal{C}(s,q,A)$

and similarly, when we add equality to each class

$\mathcal{C}(=) \equiv \mathcal{C}(1c,=)$

$\mathcal{C}(c,=) \equiv \mathcal{C}(2c,=)$

$\mathcal{C}(R,=) \equiv \mathcal{C}(1s,=)$

$\mathcal{C}(A,=) \equiv \mathcal{C}(1s,1c,=) \equiv \mathcal{C}(2s,=) \equiv \mathcal{C}(1q,=) \equiv \mathcal{C}(1A,=) \equiv \mathcal{C}(s,q,A,=)$.

Figure 1

It is interesting to label the vertices of Figure 1 in another way, as shown in Figure 2. (Note that Figures 1 and 2 are isomorphic; that, is, they represent the same relationships). This figure can be treated as a unit cube where the axes are labeled:

x-axis: "add a stack and delete a counter",

y-axis: "add a counter", and

z-axis: "add equality tests".

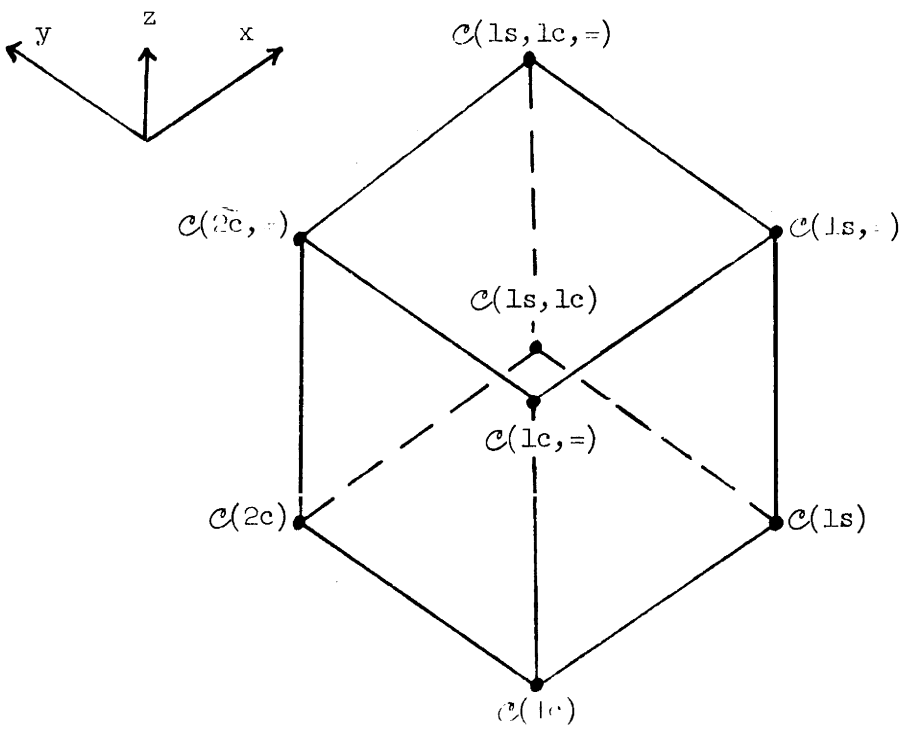


Figure 2

2. Some Proofs

To illustrate how the results of Figure 1 are proved, we give an intuitive idea of the proofs for the results indicated in Figure 3.

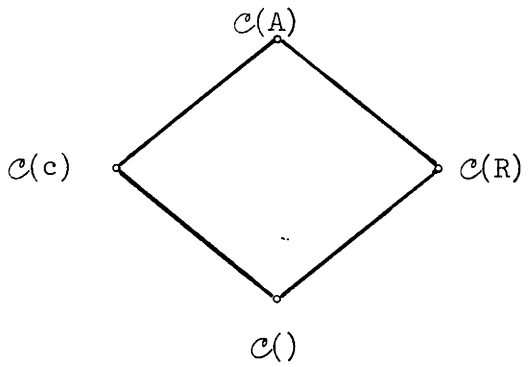


Figure 3

In the following we use the result that for any classes $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ of schemas, if $\mathcal{C}_1 \leq \mathcal{C}_2 \not\leq \mathcal{C}_3$ and $\mathcal{C}_1 \leq \mathcal{C}_3$ then $\mathcal{C}_1 < \mathcal{C}_3$. This follows from the fact that if $\mathcal{C}_1 \leq \mathcal{C}_2 \not\leq \mathcal{C}_3$ then there is a schema S in $\mathcal{C}_3$ for which there is no equivalent schema in $\mathcal{C}_2$, and hence no equivalent schema in $\mathcal{C}_1$. This implies that $\mathcal{C}_3 \not\leq \mathcal{C}_1$. Since $\mathcal{C}_1 \leq \mathcal{C}_3$, it follows that $\mathcal{C}_1 < \mathcal{C}_3$. Similarly we have that if $\mathcal{C}_1 \geq \mathcal{C}_2 \not\leq \mathcal{C}_3$ and $\mathcal{C}_1 \geq \mathcal{C}_3$ then $\mathcal{C}_1 > \mathcal{C}_3$. Thus, to show that $@(A) > \mathcal{C}(R) > \mathcal{C}()$, $@(A) > \mathcal{C}(c) > \mathcal{C}()$, and that $\mathcal{C}(R)$ and $\mathcal{C}(c)$ are unrelated, it suffices to prove that $\mathcal{C}(A) \geq \mathcal{C}(R) \geq \mathcal{C}()$, $@(A) \geq \mathcal{C}(c) \geq \mathcal{C}()$, and that $\mathcal{C}(R)$ and $\mathcal{C}(c)$ are unrelated, i.e., $(O,(R) \not\leq \mathcal{C}(c)$ and $\mathcal{C}(R) \not\leq \mathcal{C}(c)$. This follows because

$\mathcal{C}() \leq \mathcal{C}(c) \not\geq \mathcal{C}(R)$ and $\mathcal{C}() \leq \mathcal{C}(R)$ imply $\mathcal{C}() < \mathcal{C}(R)$,

$\mathcal{C}() \leq \mathcal{C}(R) \not\geq \mathcal{C}(c)$ and $\mathcal{C}() \leq \mathcal{C}(c)$ imply $\mathcal{C}() < \mathcal{C}(c)$,

$\mathcal{C}(A) \geq \mathcal{C}(c) \not\leq \mathcal{C}(R)$ and $\mathcal{C}(R) \leq @(A)$ imply $\mathcal{C}(R) < @(A)$, and

$\mathcal{C}(A) \geq \mathcal{C}(R) \not\leq \mathcal{C}(c)$ and $\mathcal{C}(c) \leq \mathcal{C}(A)$ imply $\mathcal{C}(c) < @(A)$.

It is trivial that $@(A) \geq \mathcal{C}(c) \geq \mathcal{C}()$ since every schema in $\mathcal{C}()$ is in $\mathcal{C}(c)$, and every schema in $\mathcal{C}(c)$ is in $@(A)$. We also have

$\mathcal{C}(R) \geq \mathcal{C}()$ since every simple Algol-like schema can be translated into an equivalent recursive schema by associating a defined function with each statement in the Algol-like schema. $\mathcal{C}(A) \geq \mathcal{C}(R)$ can be shown by simulating a pushdown stack with arrays using standard call-by-value ALGOL compilation (booleans are used to represent the return address).

The interesting part is to show that $\mathcal{C}(R)$ and $\mathcal{C}(c)$ are unrelated, i.e., to exhibit a schema S_1 in $\mathcal{C}(R)$ for which there is no equivalent schema in $\mathcal{C}(c)$, and a schema S_2 in $\mathcal{C}(c)$ for which there is no equivalent schema in $\mathcal{C}(R)$.

(a) Consider the following recursive schema (in $\mathcal{C}(R)$):

S_1: F(a) where

F(y) <= if p(y) then y else f(F(g(y)),F(h(y)))

There is no schema in $\mathcal{C}(c)$ equivalent to this. The reason is that the computation requires storing an arbitrarily large number of temporary data values, whereas every schema in $\mathcal{C}(c)$ has a fixed number of data variables.

Consider a class of interpretations $\{I_n\}$ having the following property: for every I_n, $n \geq 0$,

(i) distinct terms yield distinct data elements under I_n, and

(ii) p is true only for the terms that contain n occurrences of the functions g and h applied to a.

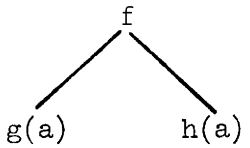
The schema S_1 on the interpretation I_n computes the term z,(a) where

$\tau_0(y) = Y$, and

$\tau_{i+1}(y) = f(\tau_i(g(y)),\tau_i(h(y)))$.

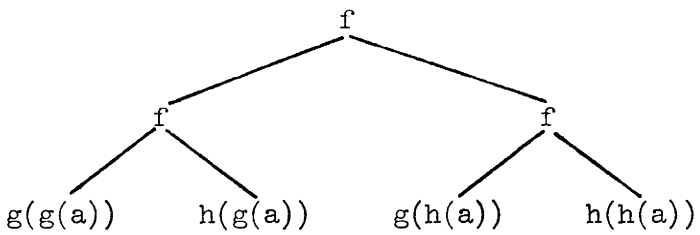
For example, S_1 under I_1 and I_2 computes the terms f(g(a),h(a)) and f(f(g(g(a)),h(g(a))),f(g(h(a)),h(h(a)))) , respectively. These terms can also be represented as binary trees as shown below:

$\tau_1(a)$: f(g(a),h(a)) is



and

$\tau_2(a)$: f(f(g(g(a)),h(g(a))),f(g(h(a)),h(h(a)))) is



Suppose there is a schema S from $\mathcal{C}(c)$ that is equivalent to S_1 . Without loss of generality we assume that S has no symbols other than a , f , g , h and p , that the only assignments that use f have the form $y_i \leftarrow f(y_j,y_k)$, and that halt statements have the form HALT(y_i) . Consider the computation of S under the interpretation I_n . Since S is assumed to be equivalent to S_1 it computes the term $z_1(a)$ which can be represented as a perfectly balanced binary tree of height n . Now we consider the computation of arbitrary binary trees in which each node corresponds to a distinct value and where in a single step at most one binary function can be applied. It is well known, and can be proved readily by induction, that the number of variables #(T) required to

compute the term corresponding to such a binary tree T is given by

$\#(\circ) = 1$, and

$\#\left(\begin{array}{c}\circ\\ T_1 \quad T_2\end{array}\right) =$ if $(\#(T_1) = \#(T_2))$ then $\#(T_1)+1$

else $\max(\#(T_1),\#(T_2))$.

This tells us that $n+1$ variables are required for computing the term $\tau_n(a)$. For example, three variables are required to compute $z_1(a)$:

$y_1 \leftarrow g(g(a))$; $Y2 \leftarrow h(g(a))$; $y_1 \leftarrow f(y_1,y_2)$;

$y_2 \leftarrow g(h(a))$; $y_3 \leftarrow h(h(a))$; $y_2 \leftarrow f(y_2,y_3)$;

$y_1 \leftarrow f(y_1,y_2)$.

Now, if the schema S has, say, m data variables, then for the computation of τ_m under I_m, S must have at least $m+1$ data variables -- a contradiction. Thus no schema in $\mathcal{C}(c)$ is equivalent to S_1.

(b) Consider the following problem: "given a constant a , unary functions f,g , and a predicate p , find an element x of the form $f^i(g^j(a))$, $i,j > 0$, such that p(x) is false. If no such x exists then the schema loops forever". In the following we refer to this problem as the witch-hunt problem.

It is easy to see that schemas in $\mathcal{C}(c)$ can solve this problem. The following is one such schema:

S_2: START(a);

L_1: $c_2 \leftarrow c_1$; $y_1 \leftarrow a$;

L_2: $c_3 \leftarrow c_2$; $y_2 \leftarrow y_1$;

while $c_3 \neq 0$ do begin $c_3 \leftarrow c_3 - 1$; $y_2 \leftarrow f(y_2)$ end;

if $p(y_2)$ then HALT(y_2);

$y_1 \leftarrow g(y_1)$;

if $c_2 \neq 0$ then begin $c_2 \leftarrow c_2 - 1$; goto L2 end;

$c_1 \leftarrow c_1 + 1$;

goto L_1.

The idea is that for a given c_1, $c_1 = 0,1,2,3,\ldots(L_1\text{-loop})$, we check the value of p for all possible terms of the form $Y2 = f^{c_2}(g^{c_1-c_2}(a))$ in the following order: $c_2 = c_1, c_1-1, \ldots, 1, 0$ (L2-loop).

However, no schema in $\mathcal{C}(R)$ can solve the witch-hunt problem. Intuitively, the reason is that no schema in $\mathcal{C}(R)$ can compute all terms of the form $f^i(g^j(a))$, in any order. For suppose there is a schema S in $\mathcal{C}(R)$ that solves the witch-hunt problem. Then, without loss of generality we can assume that S has no predicate other than p, and that defined functions in S have no boolean arguments. Let n be the largest number of arguments of any defined function in S. Consider an interpretation I_{true} for which the predicate p is true for all terms. We also require that distinct terms yield distinct data elements under I_{true}, and we claim that S cannot generate all the terms on the n+1 columns described in Figure 4.

The j-th column, $0 \leq j \leq n$, consists of all terms $f^i(g^j(a))$ for all $i > 0$. To show this, we divide all terms into 2n+3 sets A_j, B_j, C

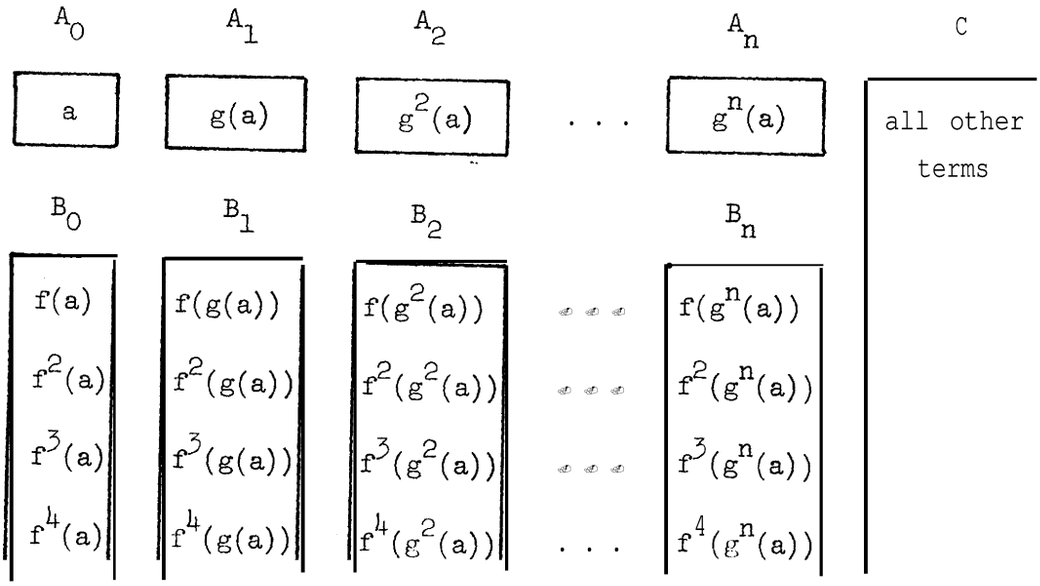


Figure 4

for $0<j<n$. The set A_j consists of the single term gj(a), the se-t B_j consists of the entire column of terms $f^i(g^j(a))$ for $i > 0$, and the set C is the "catch all" consisting of all other terms. Now, as the schema S must loop on the interpretation I_{true}, and there are only finitely many sets, there must be some defined function Fk that calls itself recursively such that each one of its argmments is in the same set as in the earlier call. Then, as the predicate tests are always true, the defined functions called between such two calls of Fk are repeated in the same order, and with the arguments from the same sets as before. Hence, there is at least one column, say j_1, such that no argument of these calls of F_k is from it. Therefore only finitely many terms from column j_1 can be

reached during the computation, i.e., there is at least one term, say $f^{i_1}(g^{j_1}(a))$, that is never tested.

Now we change the interpretation I_{true} slightly to $I_{not\ so\ true}$ in which p applied to all terms is true except that $p(f^{i_1}(g^{j_1}(a)))$ is false. Then the computation of S on the interpretation $I_{not\ so\ true}$ is the same as the computation on I_{true}, i.e., S will loop on $I_{not\ so\ true}$. But as S is assumed to solve the witch-hunt problem, it must halt with output $f^{i_1}(g^{j_1}(a))$ -- a contradiction. This proves that no schema in $\mathcal{C}(R)$ can solve the witch-hunt problem.

It is interesting to note, however, that the witch-hunt problem can indeed by solved by some Algal-like schemas with equality and no counters, i.e., by schemas in $\mathcal{C}(=)$ (see Chandra [1973]).

3. Number of Variables and Depth of Data Terms

One can investigate further the effect of the number of data variables on the power of schemas. It can be shown, for example, that for every n, $n > 0$:*/

(a) $\mathcal{C}(R, n\ var) \geq \mathcal{C}(n\ var)$

(b) $\mathcal{C}(R, 1\ var) \not\leq \mathcal{C}(n\ var)$

(c) $\mathcal{C}(R, n\ var) \not\geq \mathcal{C}(n+1\ var)$.

This implies the relations shown in Figure 5. Recall that if there is an ascending arc leading from any class $\mathcal{C}_1$ to another class $\mathcal{C}_2$ it means that $\mathcal{C}_1 < \mathcal{C}_2$.

*/ Here, " n var " indicates that the schema has at most n data variables (in Algol-like schemas) or at most n data arguments for defined functions (in, recursive schemas).

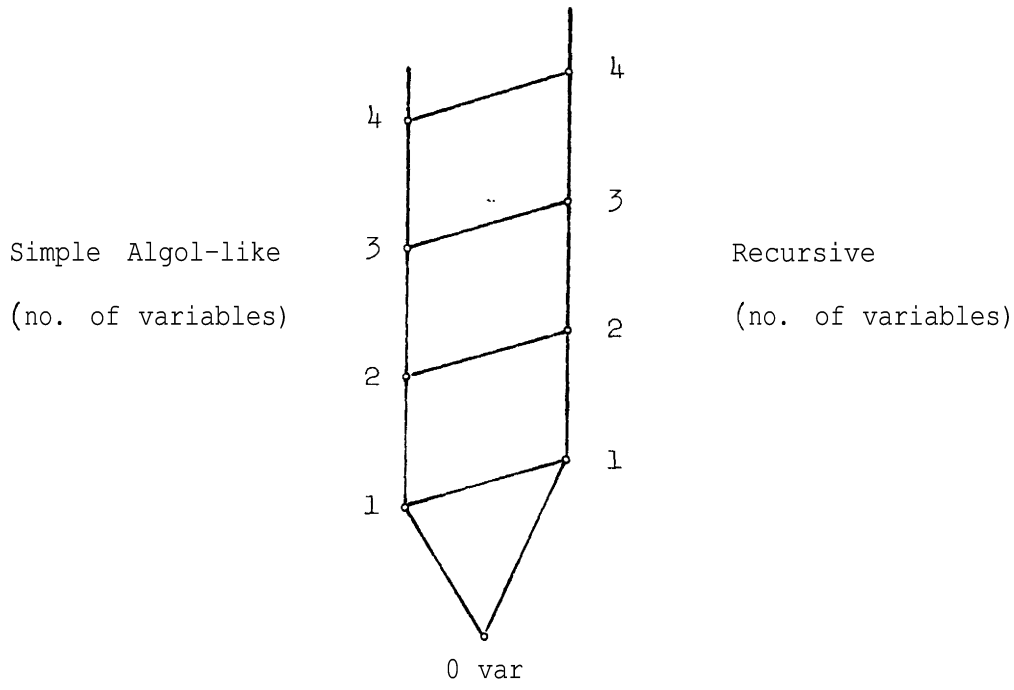


Figure 5

(a) The result that $\mathcal{C}(R, n \text{ var}) \geq \mathcal{C}(n \text{ var})$ follows by the standard process of translating a simple Algol-like schema into an equivalent recursive schema. (b) The recursive schema S_1 above is in $\mathcal{C}(R, 1 \text{ var})$, but there is no schema in $\mathcal{C}(n \text{ var})$, for any $n > 0$, which is equivalent to S. (c) To show that there is a schema in $\mathcal{C}(n+1 \text{ var})$ which is not equivalent to any schema in $\mathcal{C}(R, n \text{ var})$ we consider the following problem.

"Find an element x of the form $f^i(g^j(x))$, $i \geq 0$ and $j < n$, such that $p(x)$ is false." We refer to this problem as <u>the restricted witch-hunt problem</u>. The following schema S_3 in $\mathcal{C}(n+1 \text{ var})$ solves the problem.

S_3: START(a);

$y_2 \leftarrow g(y_1)$; $y_3 \leftarrow g(y_2)$; . . .; $y_{n+1} \leftarrow g(y_n)$;

L: if $\neg p(y_1)$ then HALT(y_1) else $y_1 \leftarrow f(y_1)$;

if $\neg p(y_2)$ then HALT(y_2) else $y_2 \leftarrow f(y_2)$;

if $\neg p(y_{n+1})$ then HALT(y_{n+1}) else $y_{n+1} \leftarrow f(y_{n+1})$;

got0 L .

Our earlier proof shows, however, that there is no schema in $\mathcal{C}$(R, n var) which solves the problem, and therefore there is no schema in $\mathcal{C}$(R, n var) which is equivalent to S_j.

There is no need to investigate how the number of boolean variables affects the power of the schemas, since it can be shown that boolean variables do not add any inherent power to Algol-like schemas or to recursive schemas (with or without equality).*/

We can further consider how the depth of data terms affects the power of schemas. The depth $|\tau|$ of a data term τ is defined as follows: $|a_i| = 0$, $|y_i| = 0$, and $|f_i(\tau_1, \ldots, \tau_n)| = 1 + \max\{|\tau_1|, \ldots, |\tau_n|\}$. Trivially,**/ $\mathcal{C}$(0 var, 0 depth) $\equiv$ $\mathcal{C}$(n var, 0 depth) $<$ $\mathcal{C}$(0 var, 1 depth) for all n. It can be shown that for every $n > 0$ and $d > 0$, we have:

(a) $\mathcal{C}$(n var, d+1 depth) $\not\leq$ $\mathcal{C}$(n+1 var, 1 depth), and

(b) $\mathcal{C}$(n+1 var, d depth) $\not\leq$ $\mathcal{C}$(0 var, d+1 depth).

*/ Note, however, that owing to the particular way we introduce pushdown stacks, queues and arrays, at least one boolean variable is required to make use of these features.

**/ Here " d depth " indicates that the schemas use data terms of depth at most d.

These results imply the relations described in Figure 6. Note that the figure indicates, for example, that $\mathcal{C}$(3 var, 2 depth) and $\mathcal{C}$(2 var, 3 depth) are unrelated.

(a) The first result can be proved by using the restricted witch-hunt problem.

(b) The second result can be proved by observing that the following schema S_4 in $\mathcal{C}$(0 var, d+1 depth) is not equivalent to any schema in $\mathcal{C}$(n+1 var, d depth) :

S_4: START(a);

HALT($f(f_1^d(a), f_2^d(a), \ldots, f_{n+2}^d(a))$) ,

where $f_i^d(a)$ means f_i applied d times to the constant a .

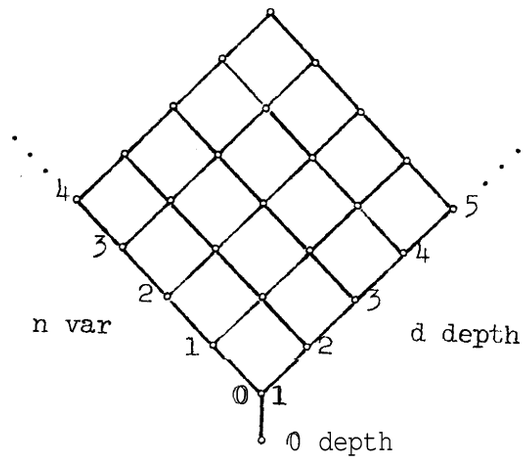


Figure 6

4. Discussion

It is reasonable to ask what it is about the various features we have discussed that makes one class of schemas more powerful than another. An observation of the arguments involved in proving the interrelationships shown in Figures 1 and 2 suggest three intuitive factors that determine the power of the various features.

(a) The amount of data space (x-axis of Figure 2 -- "add a stack and delete a counter"). Simple Algol-like schemas, and even those with counters and equality, have a fixed amount of data space. This limitation is shown by the fact that these schemas just cannot compute certain terms which are too large. The additions of a data variable to simple Algol-like schemas increases the power, as may be expected. Recursive schemas act as if they had an unbounded amount of data space available to them, as do schemas with stacks, queues or arrays.

(b) The control capability (y-axis of Figure 2 -- "add a counter"). The control capability of a schema signifies the ability of the schema to decide what to do next. Boolean variables and counters are examples of features that help in making such decisions. Boolean variables however add no inherent power, while two counters add as much control power as one might want. A pushdown stack provides, in addition to an unlimited amount of data space, some control capability because a stack can simulate a counter, but it does not have as much control capability as two counters. A queue, on the other hand, provides in addition to unlimited data space, as much control capability as two counters.

One can also consider other programming features that provide control capability. One such example is the boolean stack */ which is a pushdown stack consisting entirely of boolean values (see also Green, Elspas and Levitt [1971]).

(c) The structure of terms (z-axis of Figure 2 - "add equality"). In our discussion we observed that the addition of terms containing equality increases the power of schemas. This illustrates that if we enrich the structure of terms allowed we may increase the power of schemas. On the other hand, if we restrict the structure of terms, such as by limiting the depth of data terms, we may decrease the power.

*/ A boolean stack is strictly more powerful than one counter but strictly less powerful than a pushdown stack or two counters. Two boolean stacks, however, are just as powerful as two counters (as is also one boolean queue).

References

CHANDRA [1973]. A. K. Chandra, "On the properties and applications of program schemas," Ph.D. Thesis, Computer Science Dept., Stanford University, Report No. CS-336, AI-188 (February 1373).

CHANDRA and MANNA [1972]. A. K. Chandra and Z. Manna, "Program schemas with equality," in Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing, Denver, Colorado, (May 1972), pp. 52-64.

CONSTABLE and GRIES [1972]. R. L. Constable and D. Gries, "On classes of program schemata," SIAM Journal on Computing, Vol. 1, No. 1 (March 1972), pp. 66-118.

GARLAND and LUCKHAM [1971]. S. J. Garland and D. C. Luckham, "Program schemes, recursion schemes, and formal languages," UCLA report, No. ENG-7154, (June 1971).

GREEN, ELSPAS and LEVITT [1971]. M. W. Green, B. Elspas and K. N. Levitt, "Translation of recursive schemas into label-stack flowchart schemas," preliminary draft, Stanford Research Institute, Menlo Park, California, (June 1971).

HEWITT [1970]. C. Hewitt, "More comparative schematology," Artificial Intelligence Memo No. 207, Project Mac, M.I.T., Cambridge, Mass., (August 1970).

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

McCARTHY [1962]. J .McCarthy, "Towards a mathematical science of computation," Proc. IFIP, 1962, pp. 21-34.

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

PLAISTED [1972]. D. Plaisted, "Program schemas with counters," Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing, Denver, Colorado (May 1972), pp. 44-51.

STRONG [1971]. H. R. Strong, Jr., "Translating recursion equations into flowcharts," Journal of Computer and System Sciences, Vol. 5, No. 3, (June 1971), pp. 254-285.