ON THE POWER OF ARRAYS IN UNIVERSAL LANGUAGES

by

Donald B. Johnson†

TR 73-155

January 1973

Computer Science Department
Cornell University
Ithaca, New York 14850

†This research was done while the author was an NSF predoctoral fellow and was partially supported by NSF grant GJ-28176

On the Power of Arrays in Universal Languages

by

Donald B. Johnson

Abstract

A language with arrays but with no conditional statement is shown to be universal under "simulation," a relation on programs frequently encountered in the practical computing world. Any r.e. set can be enumerated by a program (in this language) whose flow chart is a single loop which contains no alternate execution paths normally thought necessary for computation in general. A related result is shown for any general program, thus characterizing selection in arrays as at least as powerful as conditional branching in programs. These results are related to important results in schemata.

Introduction

Turing machines, recursive functions, and certain languages on program machines with a finite number of registers of infinite precision are equivalent in computing power, and are generally accepted as computationally "universal." In fact, so great is the power of simple program machines that extremely small universal languages exist. For example, the universal language named G_3 by Constable and Borodin [1] is comprised of only three operations, addition of one, proper subtraction of one, and a transfer of control conditional on a specified register being nonzero. The reader is referred to Minsky [6] for an extensive discussion of small universal languages, their equivalence to Turing machines, and the question of universality itself.

Not only are such small languages universal, they can be proved to be so with very few registers, as few as 4 in the case of G_3. While these results on small langaages are intriguing, they put all commonly used programming languages (if infinite precision variables are allowed) into the same class, and so are of no help in comparing the power of important language features such as pushdown stores, recursion, labels as variables, and so forth. Each of these features can be mimicked through an appropriate manipulation of a small number of machine registers.

Patterson and Hewitt [7], Strong [8], and Constable and Gries [2] have shown that differences in computing power appear between such language features mentioned when program schemes are considered. For instance, recursive program schemes have more power than schemes with a finite number of variables but are themselves inferior in power to schemes with arrays. It is the array result, due to Constable and Gries [2], which interests us here. Because of the equivalence in power of schemes with arrays to schemes with pushdown stores and to the effective functionals of Strong [8], Constable and Gries suggest that these schemes are "universal." No more powerful functionals are known, and the suggestion of universality is of the same form and perhaps as convincing (given time) as that for the universality of Turing machines and languages such as G_3.

To employ arrays in program schemes in which, of course, neither the functions nor the domain of computation can be fixed, subscripting must be a "special operation." Constable and Gries maintain the necessary scheme properties by requiring the subscript values themselves to be from a domain disjoint from the domains of all interpretations for the schemes, requiring arithmetic operations on subscripts to always yield a value from the subscript domain, and defining the subscripting or selection operation as a primitive of the class of schemes.

In the sections that follow, we consider the power of array mechanisms in programs rather than in schemes. Since very simple languages are already universal, it does not appear at first that there can be any important gain from being able to name an unbounded number of variables in an infinite array. In fact, any array or set of arrays can be modelled in a few registers by mapping an array reference with multiple indices into an index in a one-dimensional array packed into two registers by means of a pairing function. The array can be "shifted" left or right to make any desired position accessible to the pairing function inverse.

What can be gained from employing arrays is a simplification in the language and in the control structure of programs. If we allow array references as variables so that subscripting or selection is a primitive operation of the program machine, under appropriate conventions regarding termination a universal language exists which has no conditional transfer of control. The entire control structure of the program is contained in the data structure, and every program has a branchless flow chart. Another way of stating the result is that the conditionals needed to mimic selection in arrays are the only conditionals needed in a universal language. We conlcude that the power of the selection operation in arrays is equivalent to the power of conditional branching in programs.

In the development which follows, we confine our attention to programs (partial functions) on the natural numbers. As is well known, this is sufficient for universality. Furthermore, we draw subscripts from the same domain. The most important restriction has to do with program termination. In general, a program with no conditional branches will not terminate. Consequently we introduce the concept of simulation and are satisfied with programs of simple structure which simulate either terminating or nonterminating programs much in the same way a nonterminating computer system executes a user's (hopefully) terminating program. Also, our constructions require a language with an assignment statement.

Programs and Simulation

In later sections we give constructions which produce, from a given program, a more complex program which "does the same thing" in a subset of its variables. We say the second program simulates the first. It is the purpose of this section to develop some of the properties of simulations of this type. The definitions of Milner [5] and Goguen [3], for instance, are somewhat different from ours, as are the applications to which their theory is applied.

Every program names a set of variables. This set is finite unless there are one or more infinite collections of variables called arrays. Operations are always on individual variables, however, whether they are members of arrays or not. We use the word "variable" to mean what is often called in the literature "simple variable." As will appear, subscripting is an operation which yields a variable. We will have no need to speak of "array variables." The following definitions make these notions more precise.

A _program_ is a finite collection of statements, with rules for their execution and a rule for deciding the order in which the statements are executed. All programs we consider execute statements in the order given except when an explicit branch occurs. We exclude parallel and non-deterministic programs. Also, statements are restricted so that they change the value of at most one variable, that is, there are no array operations implying parallelism within statements.

A _variable_ is a unique name which may take on any value from the domain $\mathbb{N} \cup \{\text{undef}\}$. An _array name_ is a bijection from $\mathbb{N}^n$ onto a countably infinite set of variables not otherwise named in the program. An array name is thus not a variable but a name for a subset of the variables of a program. Array names appear in programs only in _array references_, names qualified by a list of arguments called subscripts. An array reference, therefore, always stands for some variable. Which one it stands for is determined when the array reference is evaluated.

While there is a distinction between a variable and an array reference (which returns a variable) we class both together with the name elementary variable because at execution time both refer to some member of the program's set of variables.

In the languages we will discuss later computation occurs in assignment statements, the right hand sides of which are elementary variables or certain (total) functions of elementary variables. It is convenient to allow as subscripts any expression which may appear on the right hand side of an assignment. Thus subscripts may be elementary variables, and so themselves may be subscripted subject only to the restriction that any array reference must be a finite string of symbols. There is always an "innermost" subscript which does not belong to an array.

A full state of a program is a function from the set of variables of the program into the domain of values. A partial state is a restriction of a full state to a subset of variables. The convention chosen for initialization of variables and the choice of variables for input will define a set of allowable initial full states.

A partial computation with respect to some initial full state p_0 is a sequence of partial states beginning with q_0, a restriction of p_0. The sequence may be finite or infinite. If it is finite, the partial computation is $q_0, q_1, \ldots, q_n$ such that for $0 \leq i \leq n$ each q_i is restricted to the same subset of variables Q and, for $0 \leq i < n$, q_{i+1} is the partial state following the first assignment to some variable in Q to occur after q_i when executing the program beginning at its first statement and in p_0. Of course, q_n is the partial state at termination if execution terminates or it may be the last partial state to ever occur in some infinite execution. If the sequence is infinite, execution must be infinite. The definition is similar, but for $0 \leq i$.

A <u>full computation</u> is defined in the obvious way over full states.

A partial computation with respect to a singleton {X} is equivalent to the output of a program where X is the "printer," a distinguished variable which "prints" its contents whenever it receives an assignment.

A partial computation $q_0, q_1, \ldots, q_n$ is a <u>simulation</u> of a full computation $r_0, r_1, \ldots, r_n$ if there is a bijection f from the variable set R of the full computation to Q , the variable set of the partial computation, and, for $0 \leq i \leq n$, $r_i = q_i \cdot f$.[1] The definition extends naturally to infinite computations where the same condition holds for $0 \leq i$. The range Q of f is called the <u>simulation set</u>. A <u>control state</u> c_i is the state of all variables not in the simulation set, that is, the complement of q_i in the full state in which q_i occurs.

[1] Composed functions, denoted $f \cdot g$, are applied from the right so that $f \cdot g(x) = f(g(x))$.

Program π_2 simulates program π_1 if there exists an f and an initial control state c_0 such that for all allowable initial full states s_0 of π_1 there is a simulation on π_2 when begun in full state $c_0 \cup (s_0 \cdot f^{-1})$.

Computations and simulations have several interesting properties. Computations over sets of variables embed computations over subsets of these variable sets. Also, computations are unique. The following lemmas give these results as well as a transitivity result for simulation.

<u>Lemma 1</u> (embedding): Let $p_0, p_1, \ldots, p_n$ be a partial computation with states restricted to P , a subset of the variables of some program. If $q_0 = p_0|Q$ [1] where $Q \subseteq P$ then $q_0, q_1, \ldots, q_m$ is a partial computation with states restricted to Q if and only if $(q_0,(q_0,)^* q_1,(q_1,)^* \ldots (q_m,)^* q_m) = (p_0|Q, p_1|Q, \ldots, p_n|Q)$ where q_{i+1} differs from q_i in at most one variable for $0 \leq i < m$.

<u>Proof</u>: The proof is by induction over the length of $p_0, p_1, \ldots, p_n$. We have given that $q_0 = p_0|Q$. By definition of a computation and the fact that but one assignment can occur at a time, p_{j+1} follows p_j because of exactly one variable for each $0 \leq j < n$. It is also true that q_{i+1} can differ from q_i in at most one variable for $0 \leq i < m$; for the same reasons if

---

[1] $f|X$ denotes the restriction of the function f to the domain X .

$q_0, q_1, \ldots, q_m$ is a computation (in which case one variable causes q_{i+1} to follow q_i) and by a given condition in the case of the reverse implication.

Assume that $q_0, q_1, \ldots, q_i$ is a partial computation if and only if $(q_0, (q_0,)^* \, q_1, (q_1,)^* \ldots q_i(, q_i)^*) = (p_0|Q, p_1|Q, \ldots, p_j|Q)$. If this is true, both computations will have been generated by the same execution sequence. There will then occur a next assignment to a variable in P . This will generate p_{j+1} .

There are two cases. If the variable which generates p_{j+1} is in Q , then the definition of a computation requires there to be a q_{i+1} following q_i such that $q_{i+1} = p_{j+1}|Q$. So for this case we prove $q_0, q_1, \ldots, q_i, q_{i+1}$ is a partial computation if and only if $(q_0, (q_0,)^* \ldots q_i (q_i,)^* \, q_{i+1}) = (p_0|Q, \ldots, p_j|Q, p_{j+1}|Q)$.

If the variable which generates p_{j+1} is in $P - Q$, then $p_{j+1}|Q = p_j|Q = q_i$ and there is no q_{i+1} . So for this second case we have $q_0, q_1, \ldots, q_i$ is a partial computation if and only if $(q_0, (q_0,)^* \ldots q_i(, q_i)^*) = (p_0|Q, \ldots, p_j|Q, p_{j+1}|Q)$.

The lemma is proved by induction on j , as these are the only cases that can occur. □

Corollary 2: The result of Lemma 1 is true for $p_0, p_1, \ldots$ infinite and $q_0, q_1, \ldots, q_m$ finite or $q_0, q_1, \ldots$ infinite. □

It is important to note that an infinite computation may embed a finite computation.

<u>Corollary 3</u>: Any partial computation relative to some fixed initial full state is unique.

<u>Proof</u>: Let $q_0, q_1, \ldots, q_m$ and $p_0, p_1, \ldots, p_n$ be computations over Q and P respectively such that $Q = P$, and with respect to initial full state r_0. By Lemma 1 they must be identical since if $Q = P$ then $p_i | Q = p_i$ for $0 \leq i \leq n$. The result is clearly true for infinite computations as well. □

We use these results to show transitivity of simulation.

<u>Lemma 4</u>(transitivity): If a program π_3 simulates a program π_2, and π_2 simulates a program π_1, then π_3 simulates π_1.

<u>Proof</u>: By definition of simulation there exists an initial control state c_0 and a simulation set selector f by which π_2 simulates π_1. Similarly d_0 and g exist by which π_3 simulates π_2. If P and V are the variable sets of π_1 and π_2 respectively, then $f(P)$ and $g(V)$ are the simulation sets, under f and g respectively, in π_2 and π_3.

If $p_0, p_1, \ldots, p_m$ is a full computation on π_1, then by definition of simulation there exists a partial computation $q_0, q_1, \ldots, q_m$ on π_2 begin in $q_0 \cup c_0$ such that $p_i = q_i \cdot f$ for $0 \leq i \leq m$.

For π_2 to begin in $q_0 \cup c_0$ implies existence of a full computation $(q_0 \cup c_0) = r_0, r_1, \ldots, r_n$ on π_2 . From the definition of simulation, then, there exists a partial computation $s_0, s_1, \ldots, s_n$ on π_3 begun in $d_0 \cup s_0$ such that $r_i = s_i \cdot g$ for $0 \leq i \leq n$.

By Lemma 1 there exists on π_3 begun in $d_0 \cup s_0$ a computation $u_0, u_1, \ldots, u_k$ such that $u_0 = s_0 | g \cdot f(P)$. But from this it follows that $u_0 \cdot g \cdot f = s_0 \cdot g \cdot f = r_0 \cdot f = p_0$. By uniqueness (Corollary 3), $p_i = u_i \cdot g \cdot f$ for $0 \leq i \leq m$ from which we conclude that π_1 is simulated by π_3 with initial control state $d_0 \cup c_0 \cdot g^{-1}$ and simulation set selector $g \cdot f$.

The proof is presented for the case in which all computations are finite. It may be that $r_0, r_1, \ldots$ and $s_0, s_1, \ldots$ are infinite or that all computations $p_0, p_1, \ldots$; $q_0, q_1, \ldots$; $r_0, r_1, \ldots$; $s_0, s_1, \ldots$; and $u_0, u_1, \ldots$ are infinite. From Corollary 2 we may insert language into the proof which allows all possible cases, thus completing the proof. □

Simulation between programs is not an equivalence relation since, although simulation is transitive and reflexive, it is not in general symmetric.

That the decision problem for simulation is undecidable is shown in the next section. Time complexity for conditional-free simulations is discussed later.

Decision Problem for Simulation

The way we have defined simulation, it is possible for a terminating program and a nonterminating program to "do the same thing," to produce identical (finite) computations. So the "halting problem" becomes the "finite computation problem." We claim this is a natural recasting of the problem from its usual terms.

Certainly computers produce output before programs run to completion. Even in the functional model, we find ourselves speaking not quite precisely of programs enumerating a set when we really mean the set would be enumerated if we ran our program on all arguments in succession.

It is obvious that the finite computation problem is undecidable. We would also expect the decision problem for simulation to be undecidable. This is shown in the following theorem:

Theorem 5: It is undecidable whether general program π_2 simulates program π_1.

Proof: We reduce the strong halting problem to the decision problem for simulation, using G_3 augmented with arrays and any total functions F.

Given some π_0 with input I , construct π_3 by replacing each assignment $X[Y] \leftarrow F(Z_1, \ldots, Z_n)$ with

$$\begin{aligned}
&T_1 \leftarrow Y \\
&T_2 \leftarrow F(Z_1, \ldots, Z_n) \\
&X[T_1] \leftarrow T_2 \\
&X[T_1] \leftarrow T_2+1 \\
&X[T_1] \leftarrow T_2 \dot{-} 1
\end{aligned}$$ [1]

and preceding the program with the statements

$$\begin{aligned}
&T_1 \leftarrow T_1+1 \qquad && T_2 \leftarrow T_2+1 \\
&T_1 \leftarrow T_1 \dot{-} 1 \qquad && T_2 \leftarrow T_2 \dot{-} 1
\end{aligned}$$

where T_1, T_2 do not appear in π_0 . [2]

Then construct π_2 as follows:

$$\begin{aligned}
&\pi_3 \\
&H \leftarrow H+1
\end{aligned}$$

where H does not appear in π_3 . Notice that every variable in π_2 , except H , changes at least twice if it changes at all. H changes no more than once.

Clearly $\pi_3(i)$ halts if and only if $\pi_0(i)$ halts. So any computation on π_2 over a variable set of a single variable can have exactly two states only if the variable set contains H and π_0 halts.

The only way π_2 can simulate a program π_1 with the single statement

---

[1] Proper subtraction is defined $x \dot{-} 1 =$ <u>if</u> $x=0$ <u>then</u> 0 <u>else</u> $x-1$.

[2] Generalization to multiply subscripted references is obvious.

$H \leftarrow H+1$

is for π_0 to halt on all inputs. Thus if we could solve the decision problem for simulation we could solve the strong halting problem, known to be unsolvable. □

<u>Conditional-free</u> <u>Simulations</u>

In the usual theory, in which programs are functions which return a value (give output) only upon termination, conditional statements in some form are required for sufficient power to compute all recursive functions. These programs in general have flow charts with one or more branches.

In this section we show that, if one is content to <u>simulate</u> computing a function rather than computing it with a terminating program, then there exists a program without conditionals (and thus with a branch-free flow chart) which does so. The constructions make use of array references which are, of course, selectors of a sort. Selection is performed on the variables of the program, however, and not over the program structure as is the case when conditionals are employed.

For purposes of exposition, a result is first given for simulating nonterminating programs with a single conditional. The result is then generalized to simulation of nonterminating programs with an arbitrary number of conditionals. This result is used to show that any r.e. set can be enumerated with a conditional-free program, and finally we show any general program may be simulated by a conditional-free program.

Single-conditional Nonterminating Programs

A single-conditional nonterminating program has the form

$$
\begin{array}{rl}
 & T_1 \\
 & T_2 \\
 & \vdots \\
 & T_m \\
L_1: & S_1 \\
L_2: & S_2 \\
 & \vdots \\
L_n: & S_n \\
L_{n+1}: & \underline{\text{go}}\ \underline{\text{to}}\ L_1
\end{array}
$$

where the T_i and S_i are of the form

$$X \leftarrow F(Y_1, \ldots ,Y_k)$$

for an elementary variable X and for any k-adic total function of the elementray variables $Y_1, \ldots ,Y_k$ except that there is one statement S_j of the form

$$\underline{\text{if}}\ X \neq 0\ \underline{\text{then}}\ \underline{\text{go}}\ \underline{\text{to}}\ L_p\ .$$

The range of any function F is a subset of the natural numbers. Since our definition allows any constant function, for notational convenience we allow constants in place of any $Y_1, \ldots, Y_k$ or as subscripts in any array reference.

Lemma 6: For any single-conditional nonterminating program there exists a nonterminating program without conditionals which simulates it.

Proof: Construct a conditional-free program from the given program as follows:

(1) Increase the number of subscripts of all elementary variables by one, using a variable I_j not in the given program. Variables become monadic array references and the dimensionality of each array name is increased by one;

(2) Precede the labelled statement $L_p: S_p$ with the statement $I_j \leftarrow 0$;

(3) Replace the labelled conditional $L_j: S_j$ with

$$L_j: I_j \leftarrow X[\ldots ,0]$$

where $X[\ldots]$ is the original test variable of S_j ;

(4) Precede the entire program with the statement

$$I_j \leftarrow 0 .$$

We now show that the transformed program simulates the original under the obvious simulation set selector from the variables of the original program into the set

$\{x \mid x$ is an array reference and its last subscript is zero$\}$.

As long as X[...] = 0 in the original program, the simulating program computes identically in its simulation set. All statements in the loop L_1 to L_{n+1} execute in sequence since no branch occurs in the original program. If, however, X[...] $\neq 0$ when L_j is encountered, the original program branches to L_p . No new state can occur in the computation of the original program until statement L_p is executed.

But this is exactly what the simulation does. All statements from L_j to L_p in the forward direction over the loop are executed by the simulator, but in a set disjoint from the simulation set, that is, with $I_j \neq 0$. The variable I_j becomes 0 exactly at label L_p so a new state in the simulation is then possible. □

In [2], Constable and Gries give two programs to compute F(0,0) and F applied to all combinations of previously computed function values for an arbitrary total function F . The first is a conventional program with a conditional, and the second a program without conditionals ([2], p 97 et fol). We present a similar program for the same sequence, first with and then without conditionals. In our case, the preceding lemma gives an immediate proof that the second program computes the same sequence as the first.

```
        I ← 0
        J ← 0
        P ← 0
        A[0] ← 0
    L1: P ← P+1
        A[P] ← F(A[I],A[J])
        J ← J+1
        if I ≠ 0 then go to L2
        I ← J+1
        J ← 0
    L2: I ← I∸1
        go to L1
```

This program is a single-conditional nonterminating program. Thus the following program simulates it, computing in A[•,0] the same sequence the above program computes in A[•] . The reader may verify that the construction is in accord with Lemma 6.

```
        K ← 0
        I[K] ← 0
        J[K] ← 0
        P[K] ← 0
        A[0,K] ← 0
    L1: P[K] ← P[K]+1
        A[P,K] ← F(A[I[K],K],A[J[K],K])
        J[K] ← J[K]+1
        K ← I[K]
        I[K] ← J[K]+1
        J[K] ← 0
        K ← 0
    L2: I[K] ← I[K]∸1
        go to L1
```

Of course, label L_2 now serves no purpose except to make comparison of the programs easier.

General Nonterminating Programs

Lemma 6 illustrates our method of simulating branches by taking execution "out of the simulation set" until the destination label is reached. With this idea understood, it is now easy to generalize the method to general nonterminating programs which may have an arbitrary number of conditional branches.

A general nonterminating program has the form

$$\begin{array}{rl} & T_1 \\ & T_2 \\ & \vdots \\ & T_m \\ L_1: & S_1 \\ L_2: & S_2 \\ & \vdots \\ L_n: & S_n \\ & \underline{\text{go}}\ \underline{\text{to}}\ L_1 \end{array}$$

where the T_i are of the form

$$X \leftarrow F(Y_1, \ldots ,Y_k)$$

and the S_i may have either the form

(i) $X \leftarrow F(Y_1, \ldots ,Y_k)$ or

(ii) <u>if</u> $X \neq 0$ <u>then</u> <u>go</u> <u>to</u> L_{p_i} .

Except for allowing an arbitrary number of conditionals, the definition is the same as before.

<u>Theorem 7</u>: For any general nonterminating program there exists a nonterminating program without conditional which simulates it.

<u>Proof</u>: Let there be q conditional branch statements (of type (ii)) in a given general nonterminating program. Define a simulating program by the following transformation:

(1) Create q new variables, each with $q-1$ subscripts:

$$\begin{array}{l} I_1[\cdot,\cdot, \ldots ,\cdot] \\ I_2[\cdot,\cdot, \ldots ,\cdot] \\ \quad\vdots \\ I_q[\cdot,\cdot, \ldots ,\cdot] \quad ; \end{array}$$

(2) Precede the program with q statements to initialize the origin index location of the new variables to zero:

$$\begin{array}{l} I_1[0,0, \ldots ,0] \leftarrow 0 \\ I_2[0,0, \ldots ,0] \leftarrow 0 \\ \quad\vdots \\ I_q[0,0, \ldots ,0] \leftarrow 0 \quad ; \end{array}$$

(3) Replace each conditional branch statement. If the r^{th} such statement is

$$L_i\text{: } \underline{\text{if}}\ X \neq 0\ \underline{\text{then}}\ \underline{\text{go}}\ \underline{\text{to}}\ L_r$$

replace it with a control variable "set" statement

$$L_i\text{: } I_r[I_1[0,\ldots,0],\ldots,I_{r-1}[0,\ldots,],\ I_{r+1}[0,\ldots,0],\ldots,I_q[0,\ldots,0]] \leftarrow X$$

and precede each label L_r with a "reset" statement

$$I_r[I_1[0,\ldots,0],\ldots,I_{r-1}[0,\ldots,0],\ I_{r+1}[0,\ldots,0],\ldots,I_q[0,\ldots,0]] \leftarrow 0 \quad ;$$

(4) Augment each elementary variable of the original program with q subscripts which are origin-indexed control variables, $I_r[0,\ldots,0]$. For instance, for an array reference $X[Y_1, \ldots ,Y_k]$ write

$$X[Y_1,\ldots,Y_k,I_1[0,\ldots,0],I_2[0,\ldots,0],\ \ldots,\ I_q[0,\ldots,0]] \quad .$$

The selector from the variables of the original program into the simulation set $\{x \mid x$ is an array reference with at least q subscripts and the last q subscripts are zero$\}$ is defined in the obvious way.

Simulation is shown in the same way as before. As long as no test variable is nonzero at the time it is tested, no branch occurs in the original program and no control variable gets changed in the simulator.

When some test variable is nonzero when tested, the same arguments again apply since control variables only enter array references in the origin-indexed position. Thus, once one control variable is nonzero in its origin-indexed position, neither a control variable in this position nor any variable in the simulation set can be affected until the "reset" statement for the appropriate control variable is reached. We conclude that simulation is correct. □

The reader is invited to compare our construction with that given by Gries [4].

<u>General Programs</u>

It is well known that for every general recursive function there exists a <u>general program</u> of the form

$$
\begin{array}{l}
L_1\colon S_1 \\
L_2\colon S_2 \\
\quad\vdots \\
L_n\colon S_n \\
L_{n+1}\colon
\end{array}
$$

where the S_i are of the form

(i) $X \leftarrow X+1$,

(ii) $X \leftarrow X \dot{-} 1$,

(iii) $X \leftarrow Y$, or

(iv) <u>if</u> $X \neq 0$ <u>then</u> <u>go</u> <u>to</u> L_p ,

where X, Y are elementary variables. This language is simply

G_3 augmented with an assignment statement, (iii). A general program terminates by a branch or by natural sequencing to label L_{n+1}. We include assignment statements in order to allow conditional-free simulations and make output conventions more natural. Array references are allowed in order to state an equivalence result.

For such a program to compute a function there must be a selected output variable, say Z. When the program is run on some input p_0 it generates a computation $p_0, p_1, \ldots, p_m$, which is finite if the program computes a function. The value of Z in p_m is the value of the function.

Take a general program π_0 with input variable I and output variable Z. Construct π_1 with input variable J and output variable W, neither of which appear in π_0, as follows:

L_0: I ← J
L_1: π_0
L_{n+1}: W ← Z
J ← J+1
<u>if</u> J ≠ 0 <u>then</u> <u>go</u> <u>to</u> L_0

A computation of π_1 over variable set W will enumerate the r.e. set for the function computed by π_0, beginning of course with the $j+1^{st}$ value, where j is the input to J. We may say that the r.e. set is "printed in W."

Since all we need to do to make π_1 a general non-terminating program is to change the last statement to <u>go</u> <u>to</u> L_0 , we can construct a conditional-free simulator. This gives us the theorem:

<u>Theorem 8</u>: For any r.e. set there exists a general nonterminating program without conditionals which enumerates the set. □

There is a stronger result than this, however.

<u>Theorem 9</u>: For any general program there exists a general nonterminating program without conditionals which simulates it.

<u>Proof</u>: Let π_0 be an arbitrary general program. Construct from π_0 a program π_1 in which every elementary variable is increased in dimensionality by one, using a new variable H . Clearly π_1 simulates π_0 when $H = 0$ in the initial state and the simulation set $\{x \mid x$ is an array reference in π_1 and $H = 0\}$ is related to the variables of π_0 in the obvious manner.

Then construct a general nonterminating program π_2 from π_1 :

$$
\begin{array}{rl}
 & H \leftarrow 0 \\
L_1: & \pi_1 \\
L_{n+1}: & H \leftarrow H+1 \\
 & \underline{\text{go}}\ \underline{\text{to}}\ L_1
\end{array}
$$

It is clear that π_2 simulates π_0 on simulation set $\{x \mid x$ is an array reference in π_2 and $H = 0\}$, for if label L_{n+1} is reached, execution will never enter the simulation set again, and a simulation exists. Or if L_{n+1} is never reached, execution will remain in the simulation set, and a simulation exists within π_1.

Finally, apply the construction of Theorem 7 to produce a program π_3, free of conditionals, which simulates π_2 on the simulation set $\{x \mid x$ is in the simulation set of π_3 and $H[0,\ldots,0] = 0\}$. By Lemma 1, π_3 simulates π_0. □

Let equivalence of program classes exist whenever there are constructions which, given a program in each class, give simulating programs in the other class.

Theorem 10: The class of general programs is equivalent to the class of conditional-free nonterminating programs.

Proof: Theorem 9 gives us the construction from the class of general programs to the class of conditional-free nonterminating programs. The reverse construction is trivial since every conditional-free nonterminating program becomes a general program when the final go to is replaced by a conditional branch on an always nonzero variable. □

Obviously it is not true that conditional-free programs with arrays can be simulated by general programs without arrays, since this would require a bijection from an infinite to a finite set.

Time Complexity

In all the constructions of the preceding theorems, the nonterminating simulators give a simulation in time related linearly to the product of the size and execution time of the original program provided only the time to produce the respective computations is counted. It may indeed be reasonable to measure execution times in this way since it is always possible to arrange for a simulator of a terminating program to enter a final "computation done" state in its simulation, possibly printing a message to that effect. Otherwise, of course, we must observe that our simulating programs do not terminate in general.

References

[1] Constable, R. L. and Borodin, A. B., "Subrecursive Programming Languages, Part I: Efficiency and Program Structure," J.ACM, 19, 3, 526-568 (July 1972).

[2] Constable, R. L. and Gries, D., "On Classes of Program Schemata," SIAM J. Comput., 1, 1, 66-118 (March 1972).

[3] Goguen, J. A. Jr., "On Homomorphisms, Simulations, Correctness and Subroutines for Programs and Program Schemes," Proc. IEEE Symposium on Switching and Automata Theory, 1972, 52-60.

[4] Gries, D., "Programming by Induction," Information Processing Letters, 2 (1972).

[5] Milner, R., "An Algebraic Definition of Simulation between Programs," Stanford Artificial Intelligence Memo AIM-142, Computer Science Department, Stanford Univ. (also in Proc. I.J.C.A.I. Conf., London, September 1971).

[6] Minsky, M., Computation, Finite and Infinite, Prentice-Hall, Engelwood Cliffs, N.J., 1967.

[7] Patterson, M. S. and Hewitt, C. E., "Comparative Schematology," Conf. Record of Project MAC Conference on Concurrent Systems and Parallel Computation, Accoc. for Comput. Machinery, New York, 1970, 119-128.

[8] Strong, H. R., "High Level Languages of Maximum Power," Proc. IEEE Conf. on Switching and Automata Theory, 1971, 1-4.