ON THE EFFICIENCY OF PROGRAMS IN SUBRECURSIVE FORMALISMS

Robert L. Constable
Allan B. Borodin

Technical Report No. 70-54

April 1970

Department of Computer Science
Cornell University
Upson Hall
Ithaca, New York 14850

CONTENTS

ON THE EFFICIENCY OF PROGRAMS
IN SUBRECURSIVE FORMALISMS

by

Robert L. Constable
Cornell University*

Allan B. Borodin
University of Toronto

§1 Introduction

It is widely accepted that the theory of computing can be organized as are the natural sciences on the basis of conservation principles or trade-off relationships. Such relationships hold among quantities characterizing computation (such as information, logical complexity, structural complexity, resource expenditure, etc.). There are a number of important exchange relationships which are well-known. For instance the universal machine involves a trade-off between program structure versus size and computational complexity. Structural complexity in this example is a quantity like the "state-symbol product" for Turing machines.

An interesting measure of structural complexity of programming languages, therefore of programs, is the complexity of the subrecursive class of functions which characterizes the language. Using such a measure of structural complexity, we examine the trade-off relationship between structure and computational complexity.

---

* This research was supported in part by National Science Foundation Grant No. GJ-579.

Measures of structural complexity directly related to high level programming languages interest us most. We therefore use abstract languages which facilitate approximation of program complexity in a highly structured language like Algol.

Our attention to programming language models is also motivated by a concern for a thesis (somewhat like Church's or Turing's Thesis), implicitly known in the literature, that all functions actually used in computing are a subset of the primitive recursive functions. (Call it the Implied Thesis.) Acceptance of this thesis leads to the fact that the subrecursive programming languages considered here are adequate for "actual" computing. Furthermore, among the virtues these languages possess are that all programs halt, the run-time of any program can be estimated from its syntax, and the structure of the programs is simpler than that of programs in general recursive languages.

This situation explains our concern for structural trade-off. The natural question to ask about subrecursive languages is "at what cost do they buy their advantages?" Blum has shown that one cost is economy of program size. The subrecursive languages will always be very uneconomical in the sense that for every recursive function $f(\ )$ there will be functions $k(\ )$ whose shortest subrecursive programs, π, satisfies

$$f(|\pi|) < |\tau|$$

where $|\ |$ measures size and τ is a general recursive program for $k(\ )$.

It was conjectured that the same price was paid for run-time as well as size, at least for certain interesting subrecursive languages and formalisms such as [18], [14] or [7]. We show that the conjecture is false and that in fact these interesting subrecursive run-times are (given the right primitive arithmetics) within a linear factor of general recursive run-times.

The case for subrecursive languages is supported further by the observation in Constable [8] that the very uneconomically long subrecursive programs known from Blum must also be computationally very complex (at least on a finite set).

Owing to the advantages of these languages over general recursive languages,one surmises that they should be examined more carefully, expecially in regard to such problems as equivalence and correctness of programs.

The subrecursive languages presented here are all based on existing languages. They are selected with several criteria in mind. In one case (SR) it is to point out their expressive power as support for the "Implied Thesis." In another case (SA) it is to facilitate definitions of structural complexity. In the third case ("pure Loop") it is to relate our languages to the most elegant examples in the literature. In each case, one is able to acquire more of a "feel" for the primitive recursive functions and their apparent "naturalness". †

---

† Another implicit problem in the literature of recursive function theory and the theory of computing is to explain the apparent naturalness of $\mathcal{R}^1$. Some authors interpret their results as denying naturalness [19], other go to lengths to affirm it Constable [7].

In the last sections of this paper we explore the results and problems presented here (including "naturalness" of $\mathcal{R}^1$) in the context of abstract computational complexity on subrecursive classes. The exploration is in the form of a tentative step not a definitive step. We also offer two "applications" of the main result. The first concerns the speed-up theorem for Loop programs and the second concerns the "go to" controversy flourishing in programming circles.

§2 General Recursive Languages

One of the simplest universal programming languages is $G_3(\mathbb{N})$ defined by the instruction set

$$[v \leftarrow v + 1 \ ; \ v \leftarrow v \dot{-} 1 \ , \ \text{if } v \neq 0 \text{ then go to } \langle\text{label}\rangle]^{\dagger}$$

where v is any variable of the language. The variables "range over" the set $\mathbb{N} = \{0,1,2,\ldots\}$. The labels are usually positive integers which provide the instructions with unique identifiers. A program is a finite sequence of instructions with unique labels. The language $G_3(\mathbb{N})$ refers to the class of all programs over the instruction set.

As is shown in Shepherdson & Sturgis [24], $G_3(\mathbb{N})$ language can express programs for all partial recursive number theoretic functions, $\mathcal{P}$, consequently for all general recursive functions $\mathcal{R}$. We say that $\underline{\mathcal{R}\text{ characterizes } G_3(\mathbb{N})}$.

---

† The semantics of this language are obvious except for $v \leftarrow v \dot{-} 1$ which means v is assigned 0 if $v = 0$ otherwise $v - 1$.

To obtain a higher level language the features are slightly altered and expanded.

(1) iterative statements

```
DO v
 π
END
```

is allowed as a program (or a block) when π is a program and v any variable. The meaning of the DO-block is understood as

$$
\begin{array}{ll}
 & \bar{v} \leftarrow v \\
1 & \text{if } \bar{v} \neq 0 \text{ then } \_\_ \\
 & \pi \\
 & v \leftarrow \bar{v} \dot{-} 1 \\
 & \text{go to } 1
\end{array}
$$

where $\bar{v}$ does not occur in π. The variable $\bar{v}$ is called the loop control variable (or register).

(2) expanded conditional

"if $v \neq 0$ then s_1 else s_2"

where s_1 and s_2 are simple statements (go to's or assignments but not conditionals). These replace the simpler "if $v \neq 0$ then go to <label>" conditionals.

In order to isolate the universal power of the language, we distinguish between forward and backward go to's.

(3) directional go to's.

"go to + <label>"
"go to - <label>"

The meaning of these instructions is that in the + case the label *must* occur *below* (or *after*) the "go to" statement, in the - case it *must* occur above (or before) it, otherwise the program is not well-formed.†

We also allow

(4) computed directional go to's

"go to + <variable>"
"go to - <variable>"

The interpretation is clear, the value of the variable determines the label. One must assume that labels are integers. In fact we specify

<label> ::= 1|2|3|...

(5) expanded arithemtics

In addition to the simple arithmetic operations, + 1 and ∸ 1 it is customary to have binary addition, + , multiplication, * , exponentiation, ** , and integer division, ÷ , an identity and constants. These are syntactically encorporated by the following rules

<arithmetic operator> ::= +|*|÷|**
<term> ::= <variable>|<constant>|<variable><operator><variable>††
<assignment> ::= <variable> ← <term> .

---

† A semantically simpler statement would be "go to + c" or "go to - c" which means "go to + or - c statements from the statement." However, this use of relative addressing is not common in higher-level languages.

†† The clause <term> ::= <variable> provides for the identity and <term> ::= <constant> provides the constants

More general arithmetic operators for functions $f : \mathbb{N}^n \rightarrow \mathbb{N}$ can be added. The symbol $\perp$ will be used to indicate the addition of an arbitrary such operation,

<arithmetic operator>>::= <arithmetic operator>$|\perp$

The language having features (1) to (5) is denoted GR($\mathbb{N}$) or simply GR . Adding block structure to the language and allowing <u>nesting of conditionals</u>, e.g.

if ℓ_1 then if ℓ_2 then s_1 else s_2 else if ℓ_3 then s_3 else s_4 ,

represents an even more structured language. Since this feature is suggestive of Algol syntax, we designate the language with this feature as GRA , General Recursive Algol. The procedure feature of an actual Algol-like language is missing here, but it could easily be encorporated if it would serve our purposes. Stated precisely the nesting of conditionals becomes

(6) nested conditionals

<statement> ::= <assignment>|<go to>|<conditional>
<block> ::= <statement><iterative>| begin <program> end
<program> ::= <statement>|<block>|<program>;<program>
<conditional> ::= if <variable> $\neq$ 0 then <block> else <block> .

A complete BNF description of the languages is given by adding

<constant> ::= 0|1|2|...
<Letter> ::= A|B|C|,..|X|Y|Z
<variable> ::= <Letter>|<Letter><constant> .

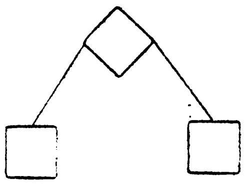
```
<iterative> ::= DO <variable>;<program>;END
<go to> ::= <go to +>|<go to ->
<go to +> ::= go to + <label>
<go to -> ::= go to - <label>
```

Simple examples of GR and GRA programs are given below.

```
GR:      if X ≠ 0 then go to +1 else go to +2
      1  DO Y
         Y ← Y + 1
         END
         go to 3
      2  DO X
         Y ← Y + 1
         END
         go to 3

GRA:     if X ≠ 0 then DO Y
                       Y ← Y + 1
                       END
                  else DO X
                       Y ← Y + 1
                       END
```

The logical structure of the programs is the same. It is represented by



where ◇ is a decision (conditional) node and □ are program blocks.

```
GR:      X2 ← X
      1  X ← X + 1
         X2 ← X2 - 1
         if X2 ≠ 0 then go to -1 else X2 ← X
```

The universality of this language can be isolated to the <u>negative go to</u> statement, either in the form "go to - <constant>", "go to - <variable>" . In other words, GR can express algorithms for all partial recursive functions, $\mathcal{R}$, because it can use the "go to" and the conditional to form uncontrolled loops. The conditional statement without the "dreaded negative go to's" is "harmless" in that it cannot produce programs that fail to halt.

Those readers familiar with Ritchie's Loop Language [18] or Minsky's "repeat" language [20] will recognize that GR without the negative "go to's" has the same power as Loop. Therefore, it will allow computations for exactly the primitive recursive functions. We will sketch an argument for this below, but first we must be precise about how programs compute functions.

<u>Computing functions in GR</u> .

To use a program π in GR to compute a function, certain variables must be specified as inputs and outputs. This will be done by writing "IN" followed by a list of variables from π and "out" followed by a single variable of π (multivariables if π is to compute $f : \mathbb{N}^n \rightarrow \mathbb{N}^p$ a vector valued function). The variables of π which are not input variables are called <u>work</u> variables. If π is to be well-defined as a function computing program, then the initial value of its work variables should not effect the output of the computation.

Def: An occurrence of a variable V in π is a *left occurrence* iff it appears on the lhs (left hand side) of an assignment but not on the rhs (right hand side). All other occurrences are *right*.

Def: Call a variable of π *left variable* iff every possible first occurrence in π (during execution) is a left occurrence. Call all other variables apparent *right* variables.

Prop: If V is a left variable of π, then the initial value of a V does not affect the output of the computation.

Pf: trivial.

Remark. The condition is clearly not necessary because if V appears first in 1 $V \leftarrow V \dot{-} 1$
if $V \neq 0$ then 1 then its initial value is irrelevant.

Def: The program π with *inputs* $x_1, \ldots, x_n$, *output* Y *computes the function* $F : S \to \mathbb{N}$ for $S \subset \mathbb{N}^n$ iff the work variables of π are all left variables and if X_i has initial value x_i, then π halts with $f(x_1, \ldots, x_n) = y$ in Y.

The program π computes a partial function $\phi : \mathbb{N}^n \to \mathbb{N}$ iff on $S = \text{domain } \phi$, π computes ϕ restructed to S and if $\langle x_1, \ldots, x_n \rangle \notin S$ then π does not halt when X_i begins with value x_i.

To be precise about this definition we should define what we mean by a *computation* of π and a *terminating computation* of π. However, this matter is treated extensively in the literature ([10], [24], etc.) and we refer the reader to these sources for a precise definition. Suffice it to say that a computation of π is simply a sequence $s_0, s_1, s_2, \ldots$ each of whose elements is an instantaneous description (id.). Each id, $\langle M_1, N_1 \rangle$, is a list M of the <u>values</u> of the variables π and the <u>number</u> N_i of the statement which was executed in the last discrete interval of time to produce those values. The first id, s_0, also contains a listing of the program, and $i = 1$ the initial instruction of π. The sequence of instruction numbers $(1, n_1, n_2, \ldots)$ associated with $s_0, s_1, s_2, \ldots$ is the <u>flow of control</u> and the sequence of lists of variable values $M_0, M_1, M_2, \ldots$ is the sequence of <u>memory configurations</u>. The id's thus contain information about all quantities which change from dit to dit.

Non-terminating computations $s_0, s_1, s_2, \ldots$ can be represented as ω-length sequences and terminating computations as finite sequence $s_0, s_1, \ldots, s_n$. Thus a computation is a sequence $s_0, s_1, \ldots, s_\sigma$ where $\sigma \leq \omega$. The value σ is the <u>number of steps</u> (number of dits) taken by π on input x, written $\sigma(x)$. We can thus represent the number of steps taken by π on input x as $\sigma\pi(x) = \sigma(x)$. The statement $\sigma\pi(x) = \omega$ is abbreviated $\pi(x)\uparrow$, read "π diverges at x", and $\sigma\pi(x) < \omega$ is abbreviated $\pi(x)\downarrow$, read "π converges at x".

Characterizing the language GR

We can now speak precisely about the expressive power of a programming language.

Def: A programming language $\mathcal{L}$ is capable of computing $\phi : \mathbb{N}^n \to \mathbb{N}$ iff there is a program π of $\mathcal{L}$ with inputs $X_1, \ldots, X_n$ and output Y which computes ϕ .

FACT: GR is capable of computing $\mathcal{P}$.

See Shepherdson & Sturgis [24] for a proof technique which applies directly.

The programming language $\mathcal{L}$ is characterized by the class H of number-theoretic (or partial number-theoretic) functions iff $\mathcal{L}$ computes precisely H .

Basic recursive function theory applied to GR

For the purposes of §3 we will need to mention certain elementary facts about GR stated in the vocabulary of recursive function theory. Such discussions usually begin with a list (indexing) of all function computing programs of the language (in general of the formalism for expressing algorithms). Therefore let $\phi_o, \phi_1, \phi_2, \ldots$ be an effective enumeration of all function computing programs.† The basic theorems needed about the list are the "universal machine theorem" and the "s-m-n" theorem (so called for Kleene's original formulation). We state these theorems for the simple case of one argument functions.

† We think of lists as including functions of any finite number of inputs, but we usually want only the one argument functions, (i.e. the ϕ_i have only one input variable specified, usually x). Therefore, we think of the list as containing n-argument function for all n from which the sublist of n argument functions for fixed n can be effectively extracted, and we use the same notation for both lists unless this will be confusing, in which case we write $\phi_{i_n}^n$ indicating n arguments.

Theorem 1: (Universal machine for one argument functions)

There is an ϕ_j^2 such that $\phi_j^2(i,x) = \phi_i(x)$ for all i and all x.[†]

Theorem 2: There is a function $\sigma(\ )$ such that $\phi_j(i,x) = \phi_{\sigma(j,i)}(x)$ for all x, i.

§3 Subrecursive Programming Language, SR, SA.

The language SR is defined to be the language obtained from GR by deleting all "go to's" except the positive "go to's" "go to + <constant>" and "go to + <variable>".

The language SA (Subrecursive Algol) is obtained by deleting all go to's from the language GRA .

Consider the simple subrecursive language "Loop" (or pure Loop) defined by

```
<constant> ::= 0
<label>::=  1|2|3|...
<Letter>::= A|B|C|...|X|Y|Z
<variable>::= <Letter>|<Letter><label>
<assignment>::= <variable>←<variable> + 1|*<variable>←<constant>|
                          <variable>←<variable>
```

*The restriction in the first two clauses is that the variable be the same on the right and left.

```
<iterative> ::= DO <variable>;<program>; END
<program> ::= <assignment>|<iterative>|<program>;<program>
```

† $\phi_i(x) = \phi_j^2(i,x)$ means $\phi_i(x)\downarrow$ iff $\phi_j^2(i,x)\downarrow$ and if $\phi_i(x)\downarrow$ then $\phi_i(x) = \phi_j^2(i,x)$.

The language is more simply specified by giving the instruction set as [$v \leftarrow v + 1$, $v \leftarrow 0$, $v \leftarrow w$, DO , END] where the semantics are as in GR . The language without $v \leftarrow w$ is called "minimal" Loop. The basic design of this language is due independently to D. Ritchie [18] (pure Loop) and M. Minsky [20] (min Loop). It was based on ideas developed by the logician R.M. Robinson [22]. It is thus easy to prove (see [18], [20] or [22]) that

Theorem 3.1: The Loop languages are characterized by $\mathcal{R}^1$.

To show that SR and SA are characterized by $\mathcal{R}^1$, the primitive recursive functions, we describe a translator from SR and SA into Loop. The ideas are very simple so we shall only sketch the description for SR .

Theorem 3.2: The language SR is characterized by $\mathcal{R}^1$.

Pf: (1) We construct a translator $T : SR \rightarrow$ min Loop by considering all statement types. First the assignment statements. To obtain the assignments of the type $X \leftarrow Y$, $X \leftarrow n$ do the following:

$X \leftarrow Y$ becomes

```
X ← 0
DO Y
X ← X + 1
END
```

and $X \leftarrow n$ becomes

$$\begin{array}{l} X \leftarrow 0 \\ \left.\begin{array}{l} X \leftarrow X + 1 \\ \quad\vdots \\ X \leftarrow X + 1 \end{array}\right\} \text{n-times} . \end{array}$$

To do the basic arithmetic, like $X \leftarrow Y \dot{-} 1$ perform

$$
\begin{array}{l}
S \leftarrow 0 \\
\mathrm{DO}\ Y \\
X \leftarrow S \\
S \leftarrow S + 1 \\
\mathrm{END}
\end{array}
$$

The other arithmetic operations, $X + Y$, $X \dot{-} Y$, $X \cdot Y$, $X \div Y$ can be done in a routine manner. Assignments involving complex terms of the form $((X + (Y \cdot Z)) \div X) + 1$ are broken down into simple assignments

$$
\begin{array}{l}
W_1 \leftarrow Y \cdot Z \\
W_2 \leftarrow X + W_1 \\
W_3 \leftarrow W_2 \div X
\end{array}
$$
using extra work variables W_i .

(2) Next consider the iterative statement DO <V>; π; END . Once π is translated, this statement translates directly. Thus one can simply proceed by induction on the depth of nesting of iteratives to prove that all can be translated.

(3) The "go to's" provide a more interesting situation. A go to of the form "go to + c" is of interest only when it occurs in a conditional, otherwise simply take the statement referred to and insert in place of the "go to + C" the entire program from that statement to the end. The case of a go to in a conditional will be covered in step (4).

In the case of a computed go to , "go to + <variable>", we show how to replace it with a sequence of conditionals and go to's . First notice that only finitely many values of variable can make sense, say $1, \ldots, m$. Therefore, to translate "go to + variable " simply replace it by m lines

$$\begin{cases} \text{if } V = 1 \text{ then go to } + (1 + m) \text{ else go to next statement} \\ \text{if } V = 2 \text{ then go to } + (2 + m) \text{ else go to next statement} \\ \vdots \\ \text{if } V = m \text{ then go to } + (m + m) \text{ else go to next statement .} \end{cases}$$

The logical expressioms " $V = i$ " are of course by an appropriate $\overline{V} = 0$.

Remark: Up to this point the <u>translation process</u> can work by simply making one pass over the "source program" in SR for each step (1) - (3) and making the appropriate replacements. The last phase, replacing the conditionals, requires more finesse.

(4) Preparatory to the last phase it must be observed that π can be arranged to have <u>no computed "go to's" within conditionals</u>. This is done by replacing any such go to with "go to + c" and relocating at c the computed go to .

(5) The conditional statements present more of a problem. Suppose "if $V \neq 0$ then s_1 else s_2" appears for s_1 and s_2 assignment statements. Then set $D \leftarrow 1 \dot{-} V$ so that $D = 0$ iff $V \neq 0$ and $D = 1$ iff $V = 0$. Now translate to

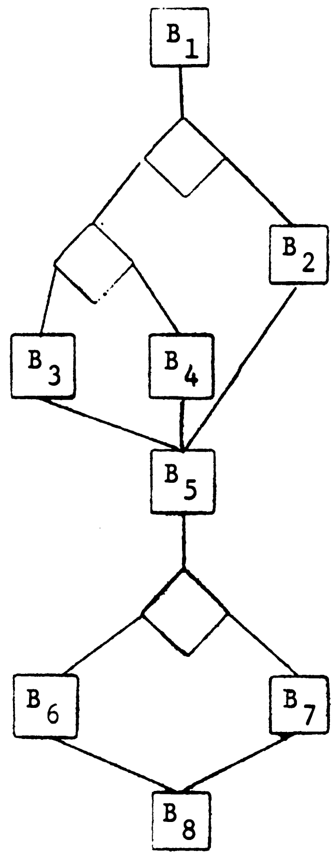
*scheme
$$\begin{cases} V \leftarrow 1 \dot{-} D \\ \text{DO } D \\ \quad s_1 \\ \text{END} \\ \text{DO } \overline{D} \\ \quad s_2 \\ \text{END} \end{cases}$$

where D and $\bar{D}$ do not occur in s_1 or s_2. If the statements s_1 or s_2 are "go to's" then we must be more careful. Suppose s_1 is go to + c , then take the statement, call it $\bar{s}_1$, referred to and put $\bar{s}_1$ and the entire SR program following $\bar{s}_1$ into the position of s_1 in the above expression. Likewise for s_2 .

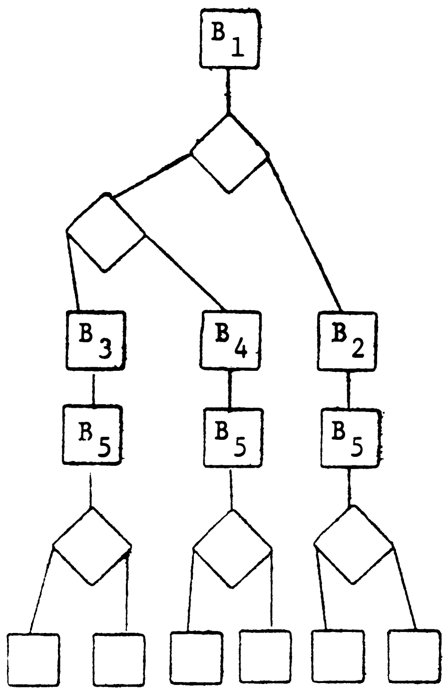
Letting $[\bar{s},\text{end}]$ denote the program from $\bar{s}$ to the last statement, we say succintly that $[\bar{s}_i,\text{end}]$ goes in for s_i .

At this point we must be careful about the translation process. When the first "go to" is encountered within a conditional, the above translation takes place and then further translation take place <u>within</u> the scope of the DO-loops defined above. (Therefore, it is critical that the variables D , $\bar{D}$ not occur elsewhere on the <u>source SR program</u> , because they should not be modified while $[\bar{s}_1,\text{end}]$ is being executed otherwise $[\bar{s}_2,\text{end}]$ might be executed also.) In summary, the source program is translated from top down according to rules (1) - (3) until a conditional containing a "go to" is encountered. Then the translation takes place within the scope of the DO loops of the * scheme, still top-down.

The translation in step (5) is logically clear because it displays the tree structure of the program. It is, however, wasteful because the programs actually have the structure of a graph. For instance, the flow diagram



Becomes the tree



As the diagram clearly shows, the possibility for duplication of code is unlimited. A more economical method of translating is given below. It does not duplicate code, instead it expands the number of DO-statements.

Method 2:

Given the conditional "if $v \neq 0$ then s_1 else s_2", call v the <u>control variable</u>. Call the statements $\pi \in SR$ <u>entry points</u> if their labels are mentioned in conditionals.

(A) Assume program π has no "go to exits" from DO-loops. Let $V_1, \ldots, V_p$ be the control variables associated with the conditional go to's of π.

Insert a "dummy location" after each transfer and before each "entry point" in π. Call these locations $T_1, \ldots, T_q$. Whether the instructions between T_i and T_{i+1} are executed depends on some binary valued expression of the V_i, say $B_i(\vec{V})$.

Then π can be translated to:

```
      ___
      ___
      ___
T1    evaluate B1
      DO B1
      ___
      ___
      ___
T2    evaluate B2
       :
Tq-1  evaluate Bq-1
      DO Bq-1
      ___
      ___ <- - - - last statement of π
      END
Tq
```

The evaluation of the B_i is straightforward.

	Pure Loop
$B_i(\vec{V}) = 1 \dot{-} V_j$	$B_i \leftarrow 1$ DO V_j $B_j \leftarrow 0$ END
$B_i(\vec{V}) = MAX(1, V_j)$	$B_i \leftarrow 0$ DO V_j $B_i \leftarrow 1$ END
$B_i(\vec{V}) = L_j \vee L_K$	$B_i \leftarrow 0$ $W = MAX(1, K_j)$ DO W $B_i \leftarrow B_i + 1$ END $W = MAX(1, L_K)$ $B_i \leftarrow B_i + 1$ END

More complicated Boolean expressions can be realized in the obvious way.

(B) It remains to consider the more complex case when the program π has nested DO-loops and there is a "go to" causing exit from the middle of a loop. The general principle is the same but now the "by-pass" loops must be placed around all code which is in the scope of the loop exited. The same scheme of Boolean expressions can be used. For example, given

```
DO v1              use the code          DO v1
___                                      DO B i1
___                                      ___
___                                      ___
DO v2                                    END
___                                      DO v2
___                                      DO B i2
___                                      ___
.                                        ___
___                                      ___
___                                      END
___                by-pass               DO B i3
END                loops                 ___
END                                      ___
___                                      ___
___                                      END
___                                      END
                                         END
                                         DO B i4
                                         ___
                                         ___
                                         ___
                                         END
```

The details are similar to case A. They are not critical to the main theorems, so we omit them here.

§4 Preliminary Theory

Before the main theorem on program efficiency can be proved, it is advantageous to organize certain scattered facts about recursive functions, subrecursive hierarchies and computational complexity. There are old results but we present them in a new light.

To begin we define a class of functions mentioned frequently in the literature of both recursion theory and complexity theory, the class of <u>elementary functions</u>, $\mathcal{E}$. At present there are two basic ways to define $\mathcal{E}$, one algebraic, the other computational. We present the latter here and refer the reader to [11] for the former.

Def: Let L_n be the class of Loop programs such that the DO-loops are <u>nested to a depth of at most n</u> . More precisely, L_o programs have no DO-instructions, L_1 programs allow DO instructions but no nested instructions and L_2 allows sub-programs of the form

```
DO X1
 DO X2
  π
 END
END
```

for $\pi \in L_o$.

In general, a Loop π is in L_{n+1} iff $\pi \in L_n$ or the only programs in the scope of a DO instruction of π are programs in L_n . The same rules define SR_n .

Def: The class of elementary functions, $\in$, is the class of functions computed by programs of L_2 .† (or of SR_2 if the arithmetics are only +1, $\dot{-}1$).

Augmenting the Loop language or the SR language by allowing computable functions f as <u>basic instructions</u> allows us to talk about relative computability. Even for noncomputable f this

---

† The definition was given first by the logician Kalmer in a more algebraic form, and proved equivalent to this definition in [ ].

concept is well defined; we imagine an "oracle" which produces the values $f(x)$ for us on demand.

Let Loop(f) be the language based on the instruction set [$v \leftarrow 0$, $v \leftarrow v + 1$, $v \leftarrow f(v)$, DO , END] and let SR(f) be the language SR augmented by the clause

<oracle assignment> ::= <variable> ← f(<variable>) .†

Defining the class of functions <u>elementary in f</u> computationally using L_2 is unnatural, and for this reason the above definition of $\mathcal{E}$ is not completely satisfactory.

Def: $\mathcal{E}(f)$, the class of functions elementary in f is the class computed by $L_2(f)$ where <u>no f assignment is allowed in the scope of a DO</u>.

This definition works because it is known from the algebraic approach that $g \in \mathcal{E}(f)$ iff $\exists\, i \;\; g(x) = T(i,x,f(x))$, for multiple argument, say $\bar{X} \in \mathbb{N}^n$, $g(\bar{X}) = T_n(i,x,f(\max\{\bar{X}\})$ where $T_n \in \mathcal{E}$. The functions T_n are obtained from the well-known Kleene "T-predicate" using the bounded <u>least number operator</u>, $\mu \leq$.†† Summarizing the relevant facts (these are not essential to the main results of the paper, but they add deeper insight).

Fact (Kleene Normal Form): There are U , $T_n \in \mathcal{E}$ such that for all $\phi_i^n : \mathbb{N}^n \rightarrow \mathbb{N}$

---

† More generally allow Loop($f_1,\ldots,f_n$) and SR($f_1,\ldots,f_n$) by adding the clause
<oracle name> ::= $f \mid f_{\text{<label>}}$
<oracle assignment> ::= <variable>←<oracle name>(<variable>) .

†† The least number operator $\mu\ldots$ reads "the least n such that ..., the operator $\mu \leq x\ldots$ reads "the least $n \leq x$ such that ...".

$$\phi_i^n(\vec{X}) = U(\mu y \; T_n(i,\vec{X},y) = 1) \; .$$

Fact: $G_n(i,\overline{X},z) = U(\mu y \leq z \; T_n(i,X,z) = 1)$ is in $\in$.

Another normal form theorem is of interest here.

Def: Let $F : \mathbb{N} \to \mathbb{N}$, the __iterates__ of f are defined as $f^{(0)}(x) = x$, $f^{(p+1)}(x) = f(f^{(p)}(x))$.

Def: Let $f_o(x) = x + 1$, $f_{n+1}(x) = f_n^{(x)}(x)$.

Theorem (Grzegorczyk Hierarchy):

(a) $\mathcal{E}(f_n) \subset \mathcal{E}(f_{n+1})$ for $n \geq 3$

(b) $\bigcup_{n=0}^{\infty} \mathcal{E}(f_n) = \mathcal{R}^1$.

This theorem is proved in [11] and [5]. The theorem becomes immediately a normal form for primitive recursive functions.

Theorem (Grzegorczyk Normal Form, GNF):

If $g : \mathbb{N} \to \mathbb{N}$ and $g \in \mathcal{E}(f_n)$, then $\exists i \; \exists p$

$$g(x) = G_1(i,x,f_n^{(p)}(x)) \text{ for all } x \; .$$

The theorem holds for m arguments using $G_m(\;)$, It applies to a wide class of functions, namely all those of the form $\bigcup_{\alpha<\beta} \mathcal{E}(f_\alpha)$ for f_α a transfinite sequence of recursive functions satisfying certain conditions. An account of this is given in Constable [7].

These results are applied to computational complexity via the following two important theorems.

Def: For C a class of functions, write $g < C$ iff $\exists$ $h \in C \ni g(x_1,\ldots,x_n) < h(x_1,\ldots,x_n)$ for all $\langle x_1,\ldots,x_n \rangle$. If C contains only one argument function, then $g(x_1,\ldots,x_n) < h(\max\{x_1,\ldots,x_n\})$

Theorem (Ritchie-Cobham)†: For any GR program ϕ_i $(\exists\ \phi_2(\ ) = \phi_i(\ )\ \&\ \sigma\phi_2(\ ) < \mathcal{E}(f))$ iff $\phi_i(\ ) \varepsilon\ \mathcal{E}(f)$.

Pf: (1) If $\sigma\phi_i < b \in \mathcal{E}(f)$ then looking at the definition of a computation and the definition of the T-predicate, one can find a function $\bar{b} \in \mathcal{E}(f)$ such that

$$\phi_i(x) = U(\mu y \leq \bar{b}(x)\ [T(i,x,y) = 1])$$

and since all operations are elementary in f, $\phi_i \in \mathcal{E}(f)$.

q.e.d.

Def: Any class K satisfying the condition $\sigma\phi_i < K \Rightarrow \phi_i \in K$ is called a full class (wrt σ).

Thus for the step counting measure on GR, $\mathcal{E}(f)$ are full classes.

(2) To show:

then $\exists_j\ \phi_j(\ ) = \phi_i(\ )$ and $\sigma\phi_j\ \varepsilon\ \mathcal{E}\ (\sigma\phi_f)$.

where ϕ_f is a program for computing $f(\ )$.

---

† The historical origins of this theorem can be found in Kleene's treatment of primitive recursive functions. Routledge [ ] called the theorem "Kleene's principle". The exact version given here was first due to Ritchie [21] and later explicitly discussed by Cobham[6].

Pf: If $\phi_i \in \mathcal{E}(f)$, then $\phi_i(\ ) = E[f(\ )]$ for $E[\ ]$ an elementary operator. But $L_2(f)$ can express the elementary operator in such a way that the number of program steps of $L_2(f)$ is elementary, $\overline{E}[\ ]$, in f . Then given a program ϕ_f for doing $f(\ )$, the total computation in $\overline{E}[\sigma\phi_f]$. These facts are tedious to verify in detail. These details can be found in Ritchie [2]. q.e.d.

Def: Any class of functions K satisfying the condition: that $\phi_i \in K \Rightarrow \sigma\phi_i \in K$ is called <u>closed</u> (wrt σ).

Def: A function f is called <u>h-honest</u> iff there is a program ϕ_f for f such that $\sigma\phi_f(x) \leq h(f(x),x)$ for all x.. Call f $\mathcal{E}$-honest iff $h \in \mathcal{E}$.

Prop.: The functions $\sigma\phi_i(\ )$ are $\mathcal{E}$-honest.

Pf: They can be computed by running ϕ_i and keeping track of the number of steps. This tallying of steps is an elementary task. q.e.d.

Theorem (Emc-theorem)[†]:

(a) if f is $\mathcal{E}$-honest, then for any recursive h , $h \in \mathcal{E}(f)$ iff $\sigma\phi_h \in \mathcal{E}(f)$ in particular;

(b) $h \in \mathcal{E}(\sigma\phi_i)$ iff $\sigma\phi_h \in \mathcal{E}(\sigma\phi_i)$.

Open Problem: What is the least closed and full class K for G or GR ?

[†] "Emc" stands for "elementary machine class" which describes a wide class of machines for which the theorem is true, see Constable [8].

§5 Relative Efficiency of GR and SR Programs for $\mathcal{R}^1$.

We know from the work of Blum that the availability of the "negative go to" allows a programmer to "compress his code". That is, GR programs can be much shorter than the shortest SR programs for some $\mathcal{R}^1$ functions. How does the "negative go to" effect computational efficiency measured in terms of running time?

The best result previously known, Meyer & Ritchie [ ], is that if ϕ_i denote GR programs and α_i denote SR programs, then if $\sigma\phi_i < f_n^{(p)}$ there is an $\alpha_i(\ ) = \phi_i(\ )$ and $\sigma\alpha_i < f_n^{(p)}$. Thus the Emc theorem for $\mathcal{E}(f_n)$ remains the same whether SR or GR is used.

There is, however, considerable latitude among run times in $\mathcal{E}(f_n)$, and the above results left open many interesting questions, particularly questions about the existence of speed-ups for primitive recursive programs. In fact, it has been conjectured (see §1 Introduction) that arbitrarily large primitive recursive gaps exist between the reasonable run-times for some SR programs and the reasonable GR programs for the same function. We show that there are no such gaps. Precisely

Relative Efficiency Theorem 5.1: For all $\phi_i(\ ) = f(\ ) \in \mathcal{R}^1$ such that $\sigma\phi(x) \geq x$ all x, there is an $\alpha_j(\ ) = f(\ )$ and a $C_i \in \mathbb{N}$ such that

$$\sigma\alpha_i(x) \leq C_i \cdot \sigma\phi_i(x) \quad \text{for all } x .$$

Pf: The theorem is proved with the restriction that SR programs do not allow exits from the middle of a DO-loop. This makes the result easily applicable to SA and Loop (pure and minimal).

The method: The motivating idea is that given a ϕ_i for f with $\sigma\phi_i < \mathcal{E}^1$ (other ϕ_i for f are clearly not of interest) there is a least n and given n a least p such that $\sigma\phi_i(x) < f_n^{(p)}(x)$ for the f_n of the Grzegorczyk hierarchy. The function f can be computed by an SR program in KNF, Kleene Normal Form, using the bound $f_n^{(p)}$. This form suggests an even better SR way to compute f( ). Namely, attempt to "simulate" ϕ_i using $f_n^{(p)}$ as a <u>clock</u> to shut off the <u>simulation</u>. If the clock can be computed in parallel and shut off when ϕ_i halts, and if the simulation cost if a fixed function of $\sigma\phi_i$, then this normal form computation will not cost much more than $\sigma\phi_i$.

The simulation cannot be the usual variety used with Turing machines because SR programs cannot perform negative go to's. But a reasonable simulation concept can be defined. The proof below consists of two parts. In part (A) we describe an SR program $\overline{\phi}_i$ which is used for simulation, and we prove that it does in fact simulate ϕ_i.

In part (B) we show how to compute $f_n^{(p)}$ in parallel with $\overline{\phi}_i$ and most critically, we show how to shut the $f_n^{(p)}$

computation off without much "over run". In this regard we notice that if there were a command to allow control to leave loops before they finish, then the over run would be minimal. We show that even without this "exit loop" option, the clock can be shut off within $c \cdot \sigma\phi_i(x)$.

It is interesting to observe that these "clock vs. simulation" arguments and "clock shut off" arguments are characteristic of many of the results in computational complexity, namely in those using diagonalization constructions. The clock arguments are especially critical in techniques which attempt <u>downward diagonalization results</u>. See [13], [4] for a discussion of these phenomena.

The proof: (1) The clock-functions $f_n^{(p)}$ can be computed by the following sequence of programs.

f_o is
```
X ← X + 1
```

f_{n+1} is
```
DO X
 f_n
END
```

$f_n^{(p)}$ is $f_n; f_n; \ldots; f_n$ p-times. For example,

$f_1^{(2)}$ is

$$f_1 \left\{ \begin{array}{l} \text{DO X} \\ \text{X} \leftarrow \text{X} + 1 \\ \text{END} \end{array} \right.$$

$$f_1 \left\{ \begin{array}{l} \text{DO X} \\ \text{X} \leftarrow \text{X} + 1 \\ \text{END .} \end{array} \right.$$

(2) Given $\phi_i(\ ) = f(\ ) \in \mathcal{R}^1$ and $\sigma\phi_i < \mathcal{R}^1$, let n_i be the least $n > 1$ such that $\exists\, p$ and $\sigma\phi(x) < f_n^{(p)}(x)$ for all x ; n_i is known to exist by [ ] . Given n_i let p_i be the least p as above. That is $\sigma\phi_i(x) < f_{n_i}^{(p_i)}(x)$ for all x . (It is shown later that there is noeeffective procedure to determine n_i, p_i given ϕ_i) .

(3) The goal of this step is to describe an effective procedure translating ϕ_i into Φ_i as the SR program used to simulate ϕ_i .

Let ϕ_i be $s_1;s_2;\ldots;s_m$. Let $s_{m+1}, s_{m+2}, s_{m+3}$ be vacant slots into which we will later insert statements. Observe that all statements s_i of ϕ_i which are not negative go to's can be translated directly into SR (if ϕ has none, then $\phi_i \in$ SR and there is nothing to prove.) Three modifications must be made.

(i) for computed go to's : For simplicity, label all statements of ϕ_i . Say s_k is labeled k . Replace all computed negative go to's by "go to $-c$" for the appropriate c as was done in Theorem 3.2 for SR programs Assume now without loss of generality that ϕ_i has this form.

(ii) Let $t_1,\ldots,t_p$ be the negative go to's among the s_i . Suppose that the variables G , S , do not occur in

ϕ_i . Then replace each t_i by the pair of instructions

G ← "label of the instruction referred to by t_j"
go to + d_j

where d_j is the distance (number of statements) to the slot s_{m+3} . Now put Z ← Z + 1 into the slot labeled s_{m+3} .

(iii) for halting: The program ϕ_i halts either by trying to execute a statement after s_m or by branching to a non-existant statement. To cover the first case, insert into the slots s_{m+1} and s_{m+2} the statements "S ← 1" and "G ← m + 1" respectively.† The variable S is used to sense the stop condition. In the case that ϕ_1 halts by branching, replace each such branch statement by "go to + d_i" where d_i is the distance to s_{m+1} .

Denote the program obtained from (i) - (iii) excluding s_{m+3} by $\overline{\phi}_i$. Note that $\overline{\phi}_i$ mimics ϕ_i until a negative go to is encountered. At that point $\overline{\phi}_i$ grinds to a halt after setting G and increasing Z . To keep the program running, put a DO-loop around it. Thus

```
DO X
go to + G
 φi
END
```

will keep $\overline{\phi}_i$ running for a least x steps. Each passage through $\overline{\phi}_i$ results in executing at least one more step of ϕ_i . To keep

---

† To perform "S ← 1" and "G ← m + 1" we have variables pre-set to these values available.

$\overline{\phi}_i$ running long enough requires building more loops around it. This can be done while computing the clock $f_{n_i}^{(p_i)}$, as is shown next.

(4) The goal of this step is to describe a way to compute the clock in parallel with $\overline{\phi}_i$ and shut it off (without much "over run") when ϕ_i halts. The asterisk will indicate the critical statement needed.

$$^{*}\text{if } S \neq 0 \text{ then } X \leftarrow 0 \text{ else } X \leftarrow Z$$

Now form the program β_f

```
DO X                    )
 .                      |
  .                     |
   DO X                 |  n_i
      DO X              |
         DO X           )
         go to + G
         φ̄_i
         Z ← Z + 1
         END            )
          *             |
      END               |
        *               |  n_i
   END                  |
      *                 |
     .                  |
    .                   |
END                     )
```

Looking at the inner most loops we see the mechanism in more detail.

```
DO X
DO X
go to + G
Φ̄_i
Z ← Z + 1
END
if S ≠ 0 then X ← 0 else  X ← Z
END
```

Observe that as long as $S = 0$ this program will compute $f_n(x)$ in variable Z, since the program is essentially

$$
\left.\begin{array}{l}\text{DO X}\\ \vdots\\ \text{DO X}\end{array}\right\} n_i\text{-times}
$$
$$
\begin{array}{l}\text{DO X}\\ Z \leftarrow Z + 1\\ \text{END}\end{array}
$$
$$
\left.\begin{array}{l}X \leftarrow Z\\ \vdots\\ \text{END}\end{array}\right\} n_i\text{-times}
$$

Furthermore, while $S = 0$ the program is computing at least one half step of ϕ_i every time $Z \leftarrow Z + 1$ is executed. Thus while $S = 0$ the value of Z indicates a lower bound on the number of steps of ϕ_i which $\bar{\phi}_i$ has "simulated".

To compute the final result, $\phi_i(x)$ form $\alpha_f = \beta_f;*;\beta_f;*;\ldots;\beta_f$ p-times. Now Z will potentially have the value $f_{n_i}^{(p_i)}(x)$. Its actual value will depend on the value it has when S becomes non-zero, i.e. when $\overline{\phi}_i$ shuts off, i.e. when $\phi_i(x)$ halts. In the next step we determine how long α_f will run compared to $\sigma\phi_i$.

(5) To calculate $\sigma\alpha_f(x)$, four facts are needed about α_f and β_f .

Let $D_1,\ldots,D_{n_i}$ be the loop control registers (see §2 Semantics) in β_f (listed in order with the inner most loop first). The following hold for all inputs x .

Lemma 1: After executing the inner most DO, at every step of the computation, $Z \geq D_i$ for $i = 1,\ldots,n_i$.

Lemma 2: After the first execution of the inner most loop, $D_1 \leq Z \leq$ number of times instructions of $\overline{\phi}_i$ have been executed. Also, $2 \cdot Z \leq$ number of steps of ϕ_i already simulated.

Lemma 3: If $S \neq 0$, then at most $3(D_1 + D_2 + \ldots + D_{n_i}) + n_i + D_i$ steps can be executed before β_f halts.

Lemma 4: If $S \neq 0$, then the maximum value of X is Z .

Using these lemmata it is easily proved that there is a c such that $\sigma\alpha_f(x) \leq c \cdot \sigma\phi_i(x)$ for all x . When $\overline{\phi}_i$ halts ,

$\overline{\phi}_i$ has been executed no more than $2 \cdot \sigma\phi_i(x)$ steps ("on the average" $\overline{\phi}_i$ is probably executing nearly one for one). Thus when $\overline{\phi}_i$ halts, $S \neq 0$, and $D_1 \leq X \leq Z \leq 2 \cdot \sigma\phi_i(x)$, by lemmas 1, 2, and 4. The total number of steps taken outside $\overline{\phi}_i$ is no more than $(4n_i)$ Z, thus no more than

$4 \cdot n_i \cdot 2\sigma\phi_i(x)$. When $\overline{\phi}_i$ halts, control is in some β_f and will **not go into another** β_f. By lemma 3, β_f can execute at most $4 \cdot \sum_{j=1}^{n_i} D_j + n_i$ more steps. So that by lemmas 1 and 2, at most $4 \cdot Z + n_i \leq 4 \cdot (2\sigma\phi_i(x) + n_i)$ more steps. Therefore to complete the program we add at most $2 \cdot p_i$ more steps in slipping over unused β_f's to complete α_f. Hence at least

$$\sigma\alpha_f(x) \leq (8) \cdot (n_i + 1) \cdot \sigma\phi_i(x) .$$

To complete the proof we need only prove the lemmata.

(6) Proofs of lemmata:

Proof of lemma 1:

1. After the first time through inner most loop, $Z = X$ and no D_i has been increased.

2. Assume the result true after m steps, to prove that it is true after $m + 1$ steps. At each step only three instruction types can change values. They are
   (i) $D_i \leftarrow X$
   (ii) $Z \leftarrow Z + 1$
   (iii) $D_i \leftarrow D_i - 1$
   (iv) $X \leftarrow Z$

Therefore, if $D_i \leq Z$ at m, then (i) can at worst bring some $D_i = X$ which by (iv) is $\leq Z$. The other two instructions can only cause $D_j < Z$ for some j. q.e.d.

Proof of lemma 2:

1. Z cannot be increased unless an instruction of $\overline{\phi}_i$ is executed. Therefore Z number of steps taken in in $\overline{\phi}_i$.

2. Every step of $\overline{\phi}_i$ either directly carries out a step of ϕ_i or else carries out the step of ϕ_i after one loop, thus after increasing Z. Thus $2 \cdot Z \leq$ number of steps of ϕ_i already simulated.

q.e.d.

Proof of lemma 3:

1. If $S \neq 0$, then by * statement the only value that can be assigned to D_i is 0. Also when $D_i = 0$ then the only statements executed in the D_{i+1} loop are "$D_{i+1} \leftarrow D_{i+1} \dot{-} 1$", "go to __" and "if $D_i \neq 0$ then __" so that after $3 \cdot D_{i+1}$ steps, $D_{i+1} = 0$. (See §2 Semantics.)

2. After $D_1 = 0$, then $3D_2 + 1 + 3D_3 + 1 + \ldots + 3D_n$ steps are executed. D_1 may execute $4D_1$ steps before being set to 0 (the "go to G" is also executed). q.e.d.

Proof of lemma 4: Trivial by examining *.

q.e.d. Theorem

Discussion: The estimate produced in the proof is very crude. There are two basic factors influencing the cost of α_f : (A) simulation time, the cost of DO V

$\overline{\Phi}_i$

END , and (B) clock time. The clock time, (B), has two subcosts: (i) computation time while clock is still needed (ii) "over-run" time, the time the clock keeps running after it is no longer needed (after $\overline{\Phi}_i$ halts). We could eliminate the over run time (the factor $3(\Sigma D_i) + n_i + D_1$) if there were a "go to" instruction allowing control to leave the scope of a DO-loop before its termination. The cost of (i) is inescapable but is minimized by computing it in parallel. This cost is reflected in the factor $4n_i \circ Z$, the <u>time spent outside of $\overline{\Phi}_i$</u> . Notice that the (B) cost depends on n_i , an index reflecting the complexity of the clock.

The simulation cost (A) depends on the "structural complexity" of ϕ_i measured in terms of the number and distribution of negative go to's . The value of Z , which determines the non-direct simulation cost as well as the clock cost, actually measures the number of times that negative go to's are executed. Thus if there are few negative go to's , then $\sigma\overline{\phi}_i(\ )$ and $\sigma\phi_i(\ )$ may be very close. The topic of structural complexity and efficiency will be discussed further in §6 .

The main theorem was proved with the restriction that $\sigma\phi_i(x) \geq x$ for all x. This restriction is necessary because SR programs constructed as in the above proof cannot run in less than x steps. Moreover, in the modified Loop language, Loop + , (Loop plus positive go to's and conditionals) to be considered later, no function $\alpha(x)$ is computable in less than x steps. For GR , functions with running times below x are possible. However, all languages mentioned, G_3 , GR , SR , SA , Loop , have a <u>strong minimum growth rate</u> in the following sense: there is a recursive monotonic increasing function $\lambda(\ )$ such that if $\liminf_{x\to\omega} \phi_i(x) = \omega$, then $\sigma\phi_i(x) \geq \lambda(x)$ a.e.x . That is, if the run time grows, it must grow at least at the rate $\lambda(\ )$. Given a strong min growth rate λ for the general recursive language GR and the time measure, we know trivially

Cor.: There is a function λ^{-1} such that for all $\phi_i(\ ) = f(\ ) \in \mathcal{R}^1$ there is an $\alpha_j(\ ) = f(\ )$ such that $\sigma\alpha_j(x) \leq \lambda^{-1}(\max\{x, \sigma\phi_i(x)\})$ for all x .

Theorem 5.2: GR and G_3 have strong minimum growth rates.

Pf: We prove firectly that G_3 has strong min growth rate $\lambda_3(\ )$ and then show that growth rate in GR can be bounded in terms of λ_3 .

1. The strong minimum growth rate for G_3 is $\lambda_3(x) = x$. Given G_3 program ϕ_i and given a constant k it is possible to produce a finite tree of all possible paths of execution of length k (if there are no conditionals, the tree has only one main branch).

On the edges after each decision node, the condition on x which causes this branch is expressed. For single inputs the expression is always an algebraic expression of x. In the case of G_3 the conditions must always be $X - n = 0$.

Slowest growth rate is obtained by finding the largest possible input x which can lead to a terminating path in the k-length execution tree. If the last decision leading to this path was $X - n = 0$, then $n \leq k$ and n is the largest input. So for such paths the growth rate is $\lambda_3(x) = x$.

If the last decision along the terminating path was $x - n > 0$, then for all $x > n$ the program terminates in n steps (note again $n \leq k$). So $\liminf \sigma\phi_i(x) = \omega$ is impossible. Thus no such path exists.

2. To establish the growth rate for GR, notice that since GR can be translated uniformly into G_3, there is for each GR arithmetic function (say $x \dot{-} y$) a cost $s_i(x,y)$ in terms of G_3. If $\sigma\phi_i(x) = y$ then the simulation cost using G_3 can be determined. Let $S(x,y) = \sum_{i=1}^{p} s_i(x,y)$ for p the number of arithmetic instructions of GR. Thus since $S(x,y)$ is

monotone in x, y, the simulation cost will be $S(v_1, v_1) + S(v_2, v_2) + \ldots + S(v_y, v_y)$ where v_i is the maximum value in any variable at step i.

This maximum value v_i can be determined as a function of v_0, the maximum initial value, and y the number of steps. The time measure σ has a speed-limit, sl, that is, in y steps a program with maximum initial value v_0 cannot produce a value larger than $sl(v_0, y)$. Thus after y steps, $v_y \le sl(v_0, y)$. Since $s(\ )$ is monotone the value $y \cdot S(sl(v_0, y), sl(v_0, y)) \overset{def}{=} t(v_0, y)$ will be the maximum number of simulation steps required. The function $t(\ )$ is increasing in v and y and because of the a.e. conditions on min growth rate we need only consider $T(y) = t(y, y)$. Since T is increasing, T^{-1} is defined.

The minimum growth rate in GR, say λ, must satisfy $\lambda(x) > T^{-1}(\lambda_3(x))$ a.e.x.

q.e.d.

The idea of a speed-limit which appears in this proof will be of interest to us in § on abstract subrecursive complexity measures.

To finish this section we note that the Efficiency Theorem is not constructive in the sense that given ϕ_i we can not determine n_i and p_i effectively.

Theorem 5.3: (a) There is no algorithm to determine for any for any GR program ϕ_i whether $\phi_i(\ ) \in \mathcal{R}^1$. If $\phi_i(\ ) \in \mathcal{R}^1$, then there is an n such that $\exists$ p and

* $\sigma\phi_i(x) \leq f_n^{(p)}(x)$ for all x. (b) However, given the information that $\phi_i(\ ) \in \mathcal{R}^1$, there is no algorithm to determine an n satisfying *. (c) Moreover, given the information that $\phi_i(\ ) \in \mathcal{L}_n$, there is no algorithm to find the least p satisfying *.

Proof:

1. Case a: This is a well-known fact. It is proved by embedding the halting problem in the decision. Namely design $\phi_{\sigma(i,n)}$ such that on input x it runs $\phi_n(n)$ for x steps. If this halts, it then computes a non-primitive recursive function. If it does not halt, it computes the successor function $X \leftarrow X + 1$. Knowing whether $\phi_{\sigma(i_o,n)} \in \mathcal{R}^1$ is equivalent to knowing whether $\phi_n(n)\downarrow$.

Cases (b) and (c) are similar.

§6 Structural Complexity

Strong normal forms and program structure

The Kleene Normal Form theorem and Universal Machine Theorem have been interpreted as results about program structure. They say that the structural complexity of a program can be traded for size. The computational cost of the structural simplicity is efficiency of operation. The main theorem of the last section determines a constant bound on the loss of efficiency.

The GNF can be used in its strong form to obtain $\alpha_i \in \mathcal{R}^1$ as

$$(1) \quad \alpha_i(x) = U(\mu y \le f_n^{(p)}(x)\ T(i,x,y)) = G_1(i,x,f_n^{(p)}(x))$$

for all x. (This means that in the GNF an elementary operator $\mu \le$, need only be applied once.)

In [18] this result is interpreted for the Loop language by saying that

Theorem 6.1: If $\alpha_i \in$ Loop and $\sigma\alpha_i(x) \le f_n^{(p)}(x)$ for all x and $n > 1$, then there is an $\alpha_j(\ ) = \alpha_i(\ )$ and $\alpha_j \in L_n$.

The direct proof is to compute T of (1) in L_2 and sequentially adjoin the computation of $f_n^{(p)}$. Thus the normal form is

$\boxed{f_n}$ clock bound

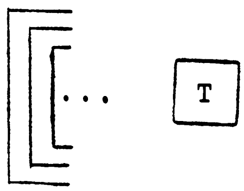
$\boxed{T}$ simulation.

Meyer & Ritchie [18] actually use a direct simulation of α_i rather than the T-predicate.

The advantage of this normal form is that structural complexity measured in terms of depth of nesting (level in L_n

hierarchy) is minimized, but at considerable cost of run time.

By translating SR (or SA) into pure Loop, the Efficiency Theorem, 5.1, immediately shows that there need be no significant loss of run time in achieving <u>nesting complexity</u> within 1 of the minimum for $n > 1$ because the clock can be computed in parallel and over-run is no more than a quadratic factor. The diagram in this case is



The L_2 complexity of T is added directly to f_n.

Precisely

Theorem 6.2: If $\beta_i \in$ Loop and $\sigma\beta_i(x) \leq f_n^{(p)}(x)$ for all x, then there is an $\beta_j(\ ) = \beta_i(\ )$, $\beta_j \in L_{n+1}$ and $\sigma\beta_j(x) \leq c \cdot (\sigma\beta_i(x))^2$ for all x.

Proof: (1) Given $\beta_i \in$ pure Loop, translate it into GR using the sematnics of §2, call the result ϕ_i. Then using $\overline{\phi}_i$ of Theorem 5.1, there is an SR program α_i such that $\alpha_i(\ ) = \beta_i(\ )$ and $\sigma\alpha_i(x) < c \cdot \sigma\beta_i(x)$ for all x, Note, the complexity of α_i is at most n.

(2) Now translate $\overline{\phi}_i$ into pure Loop using the procedure in §2, method 2 for step 5 (only part (A) is needed). Note, for pure Loop the arithmetic operations $x \dot{-} 1$ and $1 \dot{-} x$

must be simulated. The translation requires loops, but no nested loops. Therefore, the loop complexity of $T(\overline{\phi}_i)$, $T : SR \rightarrow$ pure Loop , is at most one. Thus the complexity of α_i is $n + 1$.

(3) The run-time of $T(\overline{\phi}_i)$ may not be proportional to $\sigma\overline{\phi}_i$ because $x \dot{-} 1$ and $1 \dot{-} x$ must be simulated. This simulation requires x steps. Since the speed limit in Loop is 1, in $\sigma\overline{\phi}_i(x)$ steps the largest value is $\sigma\overline{\phi}_i(x)$. Therefore, the simulation costs at most $(\sigma\overline{\phi}_i(x))^2$. The constant c appears as in Theorem 5.1.

q.e.d.

Remark: If Loop had the same arithmetics as SR , then the simulation would be a linear factor again.

Cor. 6.2: If $\phi_i(\ ) \in \mathcal{R}^1$ and $\sigma\phi_i(\ ) < \mathcal{R}^1$ for $\phi_i \in GR$, then there is a $\beta_j \in$ pure Loop , $\beta_j(\ ) = \phi_i(\ )$ and

$$\sigma\beta_j(x) \leq c \cdot \sigma\phi_i(x)^2 \quad \text{for all } x .$$

Remark: The same results hold for min Loop where v w must be simulated. Generally the result holds for a wide class of languages with general recursive bases like GR allowing arithmetics $f_1, \ldots, f_n$ and subrecursive bases like SR , SA , and Loop allowing arithmetics $g_1, \ldots, g_p$. When the arithmetics are the same, then the methods of analysis in Theorem 5.1 establish the relative efficiency. When they are different, the efficiency depends also on simulating the arithmetics. We discuss the generality of this type of result in §7.

Nesting complexity and SR_n are defined in the SR language just as in Loop, but because the additional logical structure of the conditionals and forward go to's is available, a good structure vs. efficiency result holds. Namely

Theorem 6.2: If $\alpha_i \in SR$ and $\sigma\alpha_i(x) \leq f_n^{(p)}(x)$ for all x, $n > 1$, then there is an $\alpha_j(\ ) = \alpha_i(\ )$ and $\alpha_j \in SR_n$ and $\exists\ c$ such that

$$\sigma\alpha_j(x) \leq c \cdot \sigma\alpha_i(x) \quad \text{for all } x .$$

Proof: Immediate from 5.1 by translating α_i into GR, say ϕ_i and applying the result to ϕ_i. q.e.d.

Can a similar result be proved in <u>pure Loop</u>? The answer is yes.

Theorem 6.4: If $\beta_i \in$ Loop and $\sigma\beta_i(x) < f_n^{(p)}(x)$ for all x,, $n > 1$, then there is an $\beta_j \in L_n$ and $\beta_j(\ ) = \beta_i(\ )$ and $\sigma\beta_j(x) \leq s(\max\{x,\sigma\beta_i(x)\})$ for all x where $s(x) = 2^x \cdot x$.

Proof: Translate Theorem 5.1 into pure Loop language. First notice that the * statement can be replaced by a pair of unnested DO's as shown above.

Now use the simulation in [18], given β_i this procedure produces a

```
DO V
  π̄
END
```

where π has only unnested DO's if any.

Now use

```
┌─DO X
│   *
│ ┌─DO Z
│ │ Z ← Z + 1
│ └─END
│   π
│   *
└──END
```

to replace the inner most nested loops of the program β_f in the main theorem. This program has nesting n as desired, and its run time behaves as claimed for reasons similar to those detailed in the main theorem.

q.e.d.

§7 Applications

Speed-up

One of the most interesting theorems in the theory of computational complexity is Blum's "speed-up theorem".

Fact 7.1: For all $r \in \mathcal{R}$ there is an $f \in \mathcal{R}$ such that for all $\phi_i(\ ) = f(\ )$ there is a $\phi_j(\ ) = f(\ )$ such that

$$r(\sigma\phi_j(x)) < \sigma\phi_i(x) \quad \text{a.e.}\, x\ .$$

This says that there are peculiar functions around whose computation time can be "sped up" by an arbitrary amount $r(\ )$ almost everywhere. However, Blum has shown that the speed up cannot be effective in the following sense

Fact 7.2: Let $r \in \mathcal{R}$ be any sufficiently large function. Let $f \in \mathcal{R}$, then there does not exist a program π such that if $\phi_i(\ ) = f(\ )$ then $\pi(i)\downarrow$ and $r(\sigma\phi_{\pi(i)}(x)) < \sigma\phi_i(x)$ a.e.x .

In the case of GR programs and the time measure σ "sufficiently large r" means $r(x) > x^2$ a.e.x .

The non-effectiveness of the speed-up means that it is impossible to exhibit examples of square speed up in GR . For the purpose of teaching the speed-up theorem this is disappointing. One might thus ask whether square speed ups could be illustrated in the Loop language or some subrecursive language where the structure is simple. This question has occurred to several people. The first step in answering it is to prove an $\mathcal{R}^1$-speed-up theorem in $\mathcal{R}^1$ using a simple language like Loop. One would aim to prove

Theorem 7.1: For all $r \in \mathcal{R}^1$ there is an $f \in \mathcal{R}^1$ such that for all $\alpha_i(\ ) = f(\ )$ there is an $\alpha_j(\ ) = f(\ )$ such that

$$r(\sigma\alpha_j(x)) < \sigma\alpha_i(x) \quad \text{a.e.x} .$$

This theorem cannot be proved by carrying out the Blum [1] proof directly to $\mathcal{R}^1$. It can, however, be proved using different methods, for example those in [13] and [10]. However, it has not been shown that this $\mathcal{R}^1$-speed-up is non-effective.

From the main theorem of §5 it is possible to easily prove the above Theorem 7.1 and to prove directly that for sufficiently large r the speed-up cannot be effective. Namely, the proof is to apply Blum's proof for a given r to yield an $\mathcal{R}^1$ function with r speed-up in GR. Then by Theorem 5.1 the SR programs also have r speed up for $r(x) \geq x^2$ a.e.x. Finally the speed up cannot be effective in SR because it would lead to an effective GR speed-up by the following argument.

Suppose π speeds up SR programs in the sense that if $\alpha_i(\ ) = f(\ )$ then $\pi(i)\downarrow$ and $r \circ (\sigma\alpha_{\pi(i)}(x)) < \sigma\alpha_i(x)$ a.e.x. Then define a program $\bar{\pi}$ in GR which uses a fixed SR way to compute $f(\ )$, say α_f, and given ϕ_i it assumes that $\phi_i(\ ) = f(\ )$ and that $\sigma\phi_i(x) < \sigma\alpha_f(x)$ a.e.x. Therefore, using a bound $f_n^{(p)}$ such that $\sigma\alpha_f(x) < f_n^{(p)}(x)$ for all x, it produces the image program $\bar{\Phi}_i$ and the simulation program $\alpha_k(i)$ according to the method of Theorem 5.1. Now if $\phi_i(\ ) = f(\ )$, and ϕ_i is reasonably fast, i.e. $\sigma\phi_i(x) < f_n^{(p)}(x)$, then $r(\sigma\alpha_{\pi(k(a))})$ a.e.x. To handle the case when $\phi_i(\ ) = f(\ )$ but $\sigma\phi_i(\ )$ is slow, (large), we modify $\alpha_k(i)$ so that if time $f_n^{(p)}$ is exceeded, then $\alpha_{\pi(f)}(x)$

is computed. Call the new image $\alpha_{h'(i)}$. Now the program $\alpha_{\pi(k'(i))}$ is an r speed up of any $\phi_i(\ ) = f(\ )$.

The same arguments will work for pure Loop, but now the "sufficiently large r" must be increased to compensate for the simulation of $x \dot{-} 1$.

$$\begin{tikzcd} SR \arrow[d, "\pi"'] & GR \arrow[l, "k"'] \arrow[d, dashed, "\bar{\pi}"] \\ o \arrow[r, "id"'] & o \end{tikzcd}$$

diag. commutes

Abstract approach

The Blum speed up theorem was originally proved for <u>abstract</u> (or <u>Blum</u>) measures of computational complexity. The time measure σ on GR programs is only one concrete instance of an abstract (computational complexity) measure.

The theory begins with an acceptable indexing $\phi : \mathbb{N} \to \mathcal{P}$ of all partial recursive functions and two axioms characterizing a measure $\Phi = \{\Phi_i\}$. The indexing generalizes the notion of a formalism and the measure generalizes the set $\{\sigma\phi_i\}$ of run-times. The axioms are simply that there exists a 0,1 valued recursive function $M(\ )$ and a list $\Phi = \{\Phi_i\}$ such that

Axiom 1 $\quad \Phi_i(x)\downarrow$ iff $\phi_i(x)\downarrow$

Axiom 2 $\quad \Phi_i(x) = y$ iff $M(i,x,y) = 1$.

The $K(\ )$ function is a generalization of the F predicate of KNF. From these two intentionally weak axioms a surprising number of the theorems and concepts about specific measures carry over in a reavealing machine (or programming language) independent form. Moreover, in this setting several new and important theorems were proved such as the speed-up theorem (Blum [1]), the gap theorem (Borodin [3]) and the honesty theory (McCreight & Meyer [17]), along with generalizations to operators of the first two (Fischer & Meyer [ ], Constable [ ]).

An abstract treatment of subrecursive formalisms and measures might be equally beneficial. At least it would help isolate the critical features of the arguments. We would thus propose an abstract treatment of certain aspects of the subrecursive complexity theory (restricted indexings, hierarchies, relative efficiency, relative size, etc.). This topic will be treated in more detail in Borodin & Constable [?]. Here we shall indicate a general approach to the area.

We suppose that $\mathcal{L}$ is an r.e. subset of $\mathcal{R}$, and we begin with $\alpha = \mathbb{N} \to \mathcal{L} \subset \mathcal{R}$ as an indexing of $\mathcal{L}$ obtained by selecting a subset of $\{\phi_i\}$ by the function τ.

$\alpha_n = \alpha(n) = \phi(\tau(n)) = \phi_{\tau(n)}$.* Let the measure $A = \{m\alpha_i\}$ be defined by $m\alpha_i(x) = \Phi_{\tau(i)}(x)$ all x.

There are certain obvious restrictions which must be placed on α for it to qualify as an $\mathcal{L}$-measure. Among the desirable attribures would be

---

* An acceptable $\mathcal{L}$-indexing cannot be defined simply by carrying over Roger's [23] definition of an acceptable $\mathcal{L}$-indexing because $\mathcal{L}$ cannot have a Universal Machine Theorem

(a) $\lambda i,x,y\ M(\tau(i),x,y) \in \mathcal{F}$

(b) each $m\alpha_i \in \mathcal{F}$

(c) the measure has a <u>speed limit</u>, $s \in \mathcal{F}$, i.e. $\alpha_i(x) \leq s(m\alpha_i(x))$ for all x.

These properties are analogies of the Blum axioms. Blum axiom 1 forces the measure to have arbitrarily large complexity functions, e.g. it prevents $\Phi = \{\Phi_i(x)\} = 0$ for all x from being a measure. This is accomplished here by (c). Blum axiom 2 prevents $\Phi = \{\phi_i(\ )\}$ from being a measure. This is accomplished here by (a).

Among the consequences desired for the subrecursive measures are those theorems of the general theory which hold in the class $\mathcal{F}$. For example, when $\mathcal{F} = \mathcal{R}^1$ we want

(1) speed-up theorem

(2) compression theorem (upward diagonalization theorem or jump theorem when stated in terms of classes)

(3) gap theorem

(4) honesty theorem

(5) union theorem

Many important abstract properties can be established using the recursive relationship theorem 2 in [1] in the following manner. Prove the result for T a specific measure like time, then show that the result is measure independent, and finally use the recursive relationship to carry over the result to any other measure, i.e. speed-up theorem [13] and [10].

Using the same technique with abstract subrecursive measures requires an $\mathcal{L}$-recursive relationship.

$\mathcal{L}$-(recursive) relationship: If $A = \{m\alpha_i\}$ and $B = \{m\alpha_i\}$ are $\mathcal{L}$-measures, then there is an r in $\mathcal{L}$ such that

(i) $m\alpha_i(x) \leq r(m\beta_i(x),x)$ a.e.x .

(ii) $m\beta_i(x) \leq r(m\alpha_i(x),x)$ a.e.x .

This attribute does not follow from (a) - (c) because it involves two formalisms while the others are all "internal" or "co-ordinate free" properties. In the Blum case, recursive relationship holds because the indexings are acceptable. The satisfying fact is that acceptable indexings are given an intrinsic or co-ordinate free definition. A satisfactory definition of $\mathcal{L}$-acceptable indexing would presumably lead to the $\mathcal{L}$-relationship among measures.

Some interesting observations can already be made about (a), (b), (c) as possible axioms. First, they are independent but insufficient to guarantee either the compression theorem or a recursive relationship between measures. Even a, b, c plus compression do not guarantee a recursive relationship. However, if $\mathcal{L}$ is closed under $\mu \leq$ and iteration, then (a) above implies the gap theorem. In [16] Lewis shows that (a), (b), (c) allow no r.e. complexity classes.

Results like Theorem 5.1 would follow from the existence of the function $G_n(i,x,y)$ of §4 in $\mathcal{L}$ and from a parallel cost axiom of the form

(d) $\exists\, p \in \;\; \forall i \, \forall j$

$$\sigma\alpha_i(\sigma_j) \leq p(\sigma\alpha_i, \sigma\alpha_j) \ .$$

The function p( ) represents the cost of parallelism in the formalism. For general recursive formalisms and measures such a p( ) always exists because of a recursive relationship with models like multi-tape Turing machines. However, there are subrecursive formalisms without that property (at first sight Loop might appear to be one).

In Borodin & Constable [3] these topics are treated in detail. Various consequences of possible axiom systems for subrecursive measures are examined. In [16] Lewis has considered the effect of requiring that the measures be finitely invariant.

Applications to the "go to" controversy

We have studies certain facits of program structure found in high-level languages like FORTRAN, Algol and PL/I. The use of the more sophisticated languages like Algol and PL/I has caused a certain controversy over the need for "go to's". The motivation for the controversial discussions is the fact that the use of "go to's" in Algol destroys the logical simplicity of programs and makes description of the computation difficult (see Dykstra [15]). Therefore, it is desirable to minimize their use. The question arises of whether they can be eliminated entirely without unbearable sacrifice.

The answer to the simple question of whether they can be eliminated at all is a trivial yes. Using the Kleene Normal Form we can express every number theoretic computation ϕ_i as $\phi_{k(i)}$ =

```
DO WHILE S = 0
Y ← Y + 1
S ← T(I,X,Y)
END
OUT ← U(Y)
```

The T-predicate can be computed in Loop, and we know that Loop does not need any conditionals.

Furthermore, we know that in the presence of an instruction of the form

$*$ if $S \neq 0$ then exit

which means "leave the loop immqdiately" (go to the statement immediately following the END of the inner most loop in which

$*$ occurs), the efficiency of a program similar to $\phi_{k(i)}$ (using the $\overline{\phi}_i$ simulation of §5) is within $3 \circ \sigma\phi_i(x)$.[†]

Another interpretation of the fact that "go to's" are unnecessary is that all functions actually used in computing belong to $\mathcal{R}^1$ and therefore can be computed in a restricted language like SR or SA . The main result of §5 shows that the loss of efficiency caused by using the restricted language is small. The loss of size is discussed in [2] and [8].

The interesting question about "go to's" is whether they can be eliminated in any "practical sense". To analyze this question thoroughly we should have precise measures of structural complexity and perhaps a measure of "conceptual" complexity. It will also be necessary to consider more carefully the problem of translating the high level language and executing the translated code. All of these problems appear to produce interesting mathematical questions.

† The convenience of $*$ suggests that one might want such an instruction in an actual programming language.

REFERENCES

[1] Blum, M. "Machine-Independent Theory of the Complexity of Recursive Functions," JACM 14 (1967), 322-336.

[1a] Blum, M. "On Effective Procedures for Speeding up Algorithms" ACM Symposium on Theory of Computing Marina del Rey (1969)

[2] Blum, M. "On the Size of Machines," Information and Control II (1967), 257-265.

[3] Borodin, A.B., and Constable, R.L. "Subrecursive Abstract Measures," Computer Science Tech. Report, Cornell Univ. to appear July 1970.

[4] Borodin, A.B. "Computational Complexity and the Existence of Complexity Gaps," Ph.D. Thesis, Cornell University, 1969.

[5] Cleave, John, Pl "A Heirarchy of Primitive Recursive Functions," Zerischr. F. Math. Logik and Grund D. Math. 9 (1963), 331-345.

[6] Cobham, Alan, "The Intrinsic Computational Difficulty of Functions," Logic Methodology and Philosophy of Science, Amsterdam, 1965.

[7] Constable, Robert L. "Extending and Refining Hierarchies of Computable Functions," Computer Science Tech. Report #25, University of Wixconsin, 1968.

[8] Constable, R.L. "On the Size of Programs in Subrecursive Formalisms," ACM Symposium on the Theory of Computing, 1970, 1-9.

[9] Elgot, C.C., and Robinson, A. "Random-Access Stored Program Machines, An Approach to Programming Languages," JACM 11 (1964), 365-399.

[10] Fischer, P.C., and Meyer, A.R. "On the Computational Speed-Up," IEEE Conf. Record, 9th Annual SWAT, 1968, 351-355.

[11] Grzegorczyk, A. "Some Classes of Recursive Functions," Rozprawy Mathematcyzne (1953), 1-45.

[12] Hartmanis, J., and Stearns, R.E. "On the Computational Complexity of Algorithms," Trans. AMC 117, 5 (1965) 285-306.

[13] Hartmanis, J., and Hopcroft, J.E. "An Overview of the Theory of Computational Complexity," Computer Science Tech. Report 70-59, April 1970, Cornell University.

[14] Kleene, S.C. Introduction to Metamathematics, Princeton, 1952

[15] Knuth, D., and Floyd, R. "Note on Avoiding 'go to' Statements," Computer Science Tech. Report, Stanford, 1970.

REFERENCES(cont'd.)

[16] Lewis, F.D. "Decision Problems for Complexity Classes of Recursive Functions," Proc. 2nd Ann. ACM Symp. on Theory of Computing, Northampton, 1970.

[17] McCreight, E.M., and Meyer, A.R. "Classes of Computable Functions Defined by Bounds on Computations," ACM Symp. on Theory of Computing, 1969, 79-88.

[18] Meyer, A.R., and Ritchie, D.M. "The Complexity of Loop Programs," Proc. 22nd National ACM Conf., 1967, 465-470.

[19] Meyer, A.R., and Ritchie, D.M. "A Classification of Functions by Computational Complexity," Proc. Hawaii Inter. Conf. on System Sciences, University of Hawaii Press, 1968, 17-19.

[20] Minsky, M. Computation, Finite and Infinite, Prentice-Hall, 1967.

[21] Ritchie, Robert W. "Classes of Predictably Computable Functions," Trans. AMS 106 (1963), 139-173.

[22] Robinson, R.M. "Primitive Recursive Functions," Bull. ACM 53 (1947), 915-942.

[23] Rogers, Hartley, Jr. Theory of Recursive Functions and Effective Computability, New York, 1967.

[24] Shepherdson, J.C., and Sturgis, H.E. "Computability of Recursive Functions," JACM 10 (1963), 217-255.