SUBRECURSIVE PROGRAMMING LANGUAGES II

On Program Size

Robert L Constable
Cornell University

71 - 96
Technical Report

Robert L Constable
Department of Computer Science
Cornell University
Ithaca, New York 14850

March, 1971

Subrecursive Programming Languages II
On Program Size[1]

Robert L Constable
Cornell University

ABSTRACT

Programming languages which express programs for all computable (recursive) functions are called <u>universal</u>, those expressing programs only for a subset are called <u>subrecursive programming languages</u>, SPL's. M. Blum has shown that for certain SPL's any universal programming language (UPL) contains programs which are arbitrarily shorter and nearly as efficient as the shortest SPL program for the same function. We offer new proofs of this theorem to make the relationship between size and efficiency more revealing and to show that finitely often efficiency is the price of economy of size. From the new proof we derive refinements of the basic theorem. In particular we consider the size-efficiency exchange for the task of computing constants, and derive a measure of the relative expressive power of SPL's. The results are illustrated with some new programming language models.

KEY WORDS AND PHRASES

programming language, computable (recursive) function, primitive recursive function, program size, program run-time, computational complexity, complexity class, r.e. set, recursive set, presentation, complexity of constants, ordinals, subrecursive hierarchies.

CR CATEGORIES

4.20, 4.22, 5.22, 5.24.

---

[1]The material in this paper is a portion of that presented in "On the Size of Programs in Subrecursive Formalisms" which appears in the Conference Record of the 2nd Annual ACM Symposium on Theory of Computing. Northampton, Mass., 1970, 1-9. The research was supported in part by NSF Grant GJ-579.

§ Introduction

Consider a programming language L such as reference Algol or LISP capable of expressing algorithms for all partial recursive functions $\phi: \mathbb{N}^m \to \mathbb{N}$ where $\mathbb{N} = \{0,1,2,\ldots\}$. It is well known that such languages have the capacity to express algorithms which produce astronomically large computations. Letting $\mathcal{R}$ denote the class of all (total) recursive functions, this fact means that the functions "actually computed" belong to subrecursive classes, $\mathcal{S} \subset \mathcal{R}$. For instance, there is reason to believe that all functions actually used in computing belong to $\mathcal{R}^1$, the class of primitive recursive functions.[2]

Natural programming languages, $L_{\mathcal{S}}$, can be designed which express algorithms only for the functions in a subrecursive class $\mathcal{S}$. In this paper these are called subrecursive programming languages, SPL's. Examples of them have been based on the logician's formalisms for special classes like the primitive recursive function.[3]

All programs in SPL's terminate so there is no "halting problem". A bound for the running time of a program can be determined from the input and syntax. The conceptual structure of programs is simpler

---

[2] One can argue that only finite functions are "actually computed". However, for reasons of mathematical application a first approximation to actual computing should allow for the computability of infinite functions such as $x+y$, $x \cdot y$, etc. See Elgot and Robinson [12] and McCarthy [18] for a discussion of this point. It is in fact one of the tasks of computing theory to discover a class (or classes) of functions which adequately represents the functions actually computed. The class $\mathcal{E}$ of elementary, or even its subclass $\mathcal{E}^2$ of <u>primary</u> functions, may be more reasonable a candidate than $\mathcal{R}^1$. However, subrecursive classes are not primarily of interest because the functions are used in computing, but because they serve to measure the capabilities of languages and computing systems.

---

[3] In 1965 Cleave [6], designed a language for $\mathcal{R}^1$ based on ideas in Grzegorczyk [13] while Ritchie [24] and Minsky [22] designed another such language based on the ideas of R M Robinson [25]. Ritchie [24] (and later Constable [9]), designed languages for $\mathcal{R}^n$, the multiply recursive functions of Péter, based on the notion of a stack. In [9] Constable defined another class of language for $\mathcal{R}^n$ based on restricted program modification.

than that in universal programming languages.

Computational efficiency is not sacrificed for these advantages. In a forthcoming article, the author and Borodin [10] show there is no significiant loss of computational efficiency caused by computing with certain subrecursive languages.

Blum [3] shows that program compactness is sacrificed for these advantages. He defines the notion of program size axiomatically. An example of size is the length of a program (number of cards in the deck). For a program i, let $|i|$ be its size. Blum shows that if f is a recursive function, there is a primitive recursive function f_i whose minimum subrecursive program, say i_0, satisfies

$$f(|j|) < |i_0|$$

for j some general recursive program for f_i. So for $f(x)=100\cdot x$, there is some primitive recursive function whose shortest subrecursive program is 100 times longer than one of its general recursive programs. Furthermore, Blum shows that the computational complexity, say run-time of j is nearly the same as that for i except on a finite set.

Blum's result seems to indicate that universal programming languages have a decided advantage over subrecursive languages. He says, "in order for programs to be of economical size, the programming language must be powerful enough to compute arbitrary general recursive functions".

In this paper Blum's results are given more transparent proofs and examined further. It is shown that for a wide class of formalisms, a program which can be significantly compressed without degrading efficiency <u>infinitely often</u> (i.o.), is difficult on a finite set. Thus, any significant improvement in size comes at a cost of efficiency, either i.o. or on a finite set.

---

[4]With modifications of the "Loop" language of Minsky and Ritchie the loss is at most a linear factor c, and for their original language the loss is at most a square factor.

The technical results here both new and old are not difficult. But when formulated in terms of programming languages they are provocative, and they capture fundamental facts in a way which should be meaningful to the computer scientist who is not a specialist in theory. In particular, they serve to place the halting problem in perspective and focus attention on SPL's.

This viewpoint has influenced the format of the paper. We begin with a substantial section defining programming language models, motivating bounded formalsims, and relating each to a basic theoretical structure, the acceptable indexing.

In §3 we present the theorems and their bearing both on evaluating and understanding SPL's. We conclude with a suggested measure of an SPL's expressive power.

§2 Programming Languages

(2.1) <u>notation</u>

Let $\mathbb{N}=\{0,1,2,\ldots\}$ and let $\mathcal{F}_n$ and $\mathcal{P}_n$ denote respectively the classes of total and partial number theoretic functions from $\mathbb{N}^n$ into $\mathbb{N}$. Following Roger's [25] use lower case Latin letters, f,g,h, for names of elements of $\mathcal{F}_n$ and lower case Greek letters, α,ϕ,ψ, for names of elements of $\mathcal{P}_n$. Here we use expressions like f( ), α( ), etc., to distinguish the function name from the function. (Note function computing programs are names for functions.) Let $\mathcal{R}_n$ and $\mathcal{PR}_n$ be the total and partial recursive (computable) functions respectively. The following diagram indicates the containment relations.

$$\begin{tikzcd}[arrows=dash] & \mathcal{P}_n \arrow[dl] \arrow[dr] & \\ \mathcal{F}_n \arrow[dr] & & \mathcal{PR}_n \arrow[dl] \\ & \mathcal{R}_n & \end{tikzcd}$$

The set symbol without the subscript denotes the union of overall n. Thus, e.g. $\mathcal{F}=\bigcup_{i=0}^{\infty}\mathcal{F}_i$. The sequence $\{\phi_i^n\}$ is a listing of $\mathcal{PR}_n$ and $\{\phi_i\}$ a listing of $\mathcal{PR}$.

(2.2) <u>universal programming languages</u>[5]

The results of this paper hold for a wide class of formalisms for $\mathcal{PR}$. Universal programming languages (UPL's) are only one example. (Turing machines or Herbrand-Gödel equations are other examples.) Here a <u>formalism</u> for $\mathcal{PR}$ will be a formal language, L, whose elements are partial function names, ϕ_i, and a <u>semantic</u> <u>mapping</u> onto $\mathcal{PR}$, i.e.

$$(\ ): L \rightarrow \mathcal{PR}$$

The map takes ϕ_i to $\phi_i(\ )$. The formalism is called <u>acceptable</u> iff there is a computable <u>encoding</u> of function names, ϕ_i, one-one onto $\mathbb{N}$, say $\phi_i \rightarrow i$, such that the induced mapping $\phi:\mathbb{N}\rightarrow\mathcal{PR}$, defined by $\phi(i)=\phi_i(\ )$ is

---

[5]As is customary in the theory we restrict our attention to the simple task of computing functions from $\mathbb{N}^n$ into $\mathbb{N}$.

an <u>acceptable indexing</u> in the sense of Roger's [27], i.e. the universal machine theorem and S-m-n theorem hold for it (see Roger's [27], exercise 2.10).[6]

The image of the ϕ-map, $\phi(i)$ is often written ϕ_i any many authors identify ϕ_i and $\phi_i(\ )$ reserving the encoding i to denote the function name, ϕ_i.

High level programming languages (like Algol) with their semantics are formalisms. They are usually thought of in terms of a grammer to describe the language proper and a translator (compiler) to specify the semantics. The same is true of the abstract models of high level programming languages which we present. But the semantics will be handled very informally because it is straight forward, and there is ample treatment in the literature.

<u>Examples</u>

(1) UPL, G_3

We introduce the notion of a <u>G-type language</u> via a class of program schemata. The schemata have (individual) variables, V, VV, VVV,... . For abbreviation, let V...V n-times be V_n and use v_i as variables over these variables. The positive integers, $\mathbb{N}^+=\{1,2,3,...\}$ are statement labels, denoted ℓ_i. Schemes also used function variables, F, FF, ... abbreviated F_i, and f_i are used as variables over them. Relation variables, P, PP, ... are abbreviated P_i, and p_i range over them.

There are two types of statement schemes, <u>assignments</u> and <u>conditionals</u>

<assignment scheme>=$v \leftarrow f(v)$ <conditional scheme>=if $p(v)$ then ℓ

A program scheme is a finite sequence of uniquely labelled statement schemes. For example

[6] In a more constructive theory, the class $\mathcal{PR}$ cannot be used without specifying a formalism for it. So constructively our definition requires a standard formalism for $\mathcal{PR}$ and a translation from L into that formalism. The standard formalism is usually Turing machines but that is a matter of what you learn first.

1 $V_1 \leftarrow F_1(V_1)$

2 if $P_1(V_1)$ then 1

3 $V_2 \leftarrow F_1(V_2)$

4 if $P_1(V_2)$ then 3

The G-type semantics should be obvious and we have chosen this example primarily as a basis for the more interesting examples GR and SR. Briefly, f denotes a function $f_i(\ ): \mathbb{N} \to \mathbb{N}$, so $v \leftarrow f(v)$ means "assign to variable v the result of applying function f( ) to the contents of variable v", and p denotes a predicate $p(\ ): \mathbb{N} \to \{T,F\}$ and so if "p(v) then ℓ_i" means "if p(v) is true then go to statement labelled ℓ_i as the next instruction, otherwise go to the immediately following statement." To assign a program ϕ_i to a partial function $\phi_i(\ )$ we specify input variables $v_1,\dots,v_n$ and an output variable v. Then ϕ_i computes $\phi_i(x_1,\dots,x_n)$ iff when ϕ_i is run with input variables v_i containing x_i, it halts with value $\phi_i(x_1,\dots,x_n)$ in v iff $\phi_i(x_1,\dots,x_n)$ is defined. A precise semantics is conveniently given in terms of a Register machine, see Shepherdson & Sturgis [28] or Minsky [22], or in terms of a RASP, see Elgot and Robinson [12].

A particular G-type language results from specifying values for a finite number of functions and predicate variables (and removing all other such variables). For example, the specific language G_3 results from taking F_1 as $f(x)=x+1$, F_2 as $f(x)=x\dot{-}1$, and P_1 as $p(x)$ iff $x\neq 0$. These are abbreviated $+1$, $\dot{-}1$, $\neq 0$, respectively, so G_3 is denoted by the base $[+1, \dot{-}1, \neq 0]$. Minsky [22] shows that G_3 is universal and provides semantics.

(2) UPL, type GR

Our second UPL example is an extension of the G-type. First we allow the operations and predicates to be n-ary. Thus <assignment>$=v \leftarrow f(v_1,\dots,v_n)$. The specific functions and predicates used are

$f_1(x,y)=x+y$, $f_2(x,y)=x\dot{-}y$, $f_3(x,y)=x\cdot y$, $p_1(x,y)$ iff $x=y$, $p_2(x,y)$ iff $x\neq y$.

We abbreviate by $+$, $\dot{-}$, $.$, $=$, $\neq$.

We also separate the go to's from the conditional.

<go to +> go to $+\ell$ abbreviated $\downarrow$

<go to -> go to $-\ell$ abbreviated $\uparrow$

The $\pm$ signs tell the direction of the label ℓ in relation to the go to itself. Thus go to $+\ell$ means "find ℓ below this location" and go to $-\ell$ means "find ℓ at or above this location".

We now liberalize the conditional <GR-conditional>="if p(v) then s" where s is either <go to +>,<go to -> or <assignment>. We abbreviate this conditional as $\diamond$.

Finally, we add an iterative type statement.

<iterative>::=DO v;<program>;END.

The G-type meaning of the iterative DO v; π; END appearing in the program $\hat{\pi}$ is

$$
\begin{array}{ll}
 & \bar{v} \leftarrow v \\
1 & \text{if } \bar{v} \neq 0 \ \text{ then } 2 \\
 & \bar{v} \leftarrow \bar{v} \dot{-} 1 \\
 & \text{go to } 1 \\
2 & \underline{\qquad\qquad}
\end{array}
$$

where $\bar{v}$ does not appear in either π or $\hat{\pi}$. We abbreviate the iterative by DO.

In summary, using the abbreviations, the GR-type language is

$$[+1,\ \dot{-}1,\ +,\ \dot{-},\ \cdot,\ \neq 0,\ =,\ \neq,\ \uparrow,\ \downarrow,\ \diamond,\ \text{DO}].$$

(2.3) <u>subrecursive programming languages</u>

As for UPL's our results apply to a wide class of subrecursive formalisms (such as primitive recursive equations) as well as SPL's. Again a formalism is a formal language L and a semantic map

$$(\ ) : L \to \mathcal{S} \subset \mathcal{R}$$

The concept of an acceptable subrecursive formalism is not clear. Problems with it are discussed in [10]. Here we are content with two interesting examples of subrecursive formalisms. In this part, an SPL is called SR for "<u>S</u>ub<u>R</u>ecursive", in the next part a different type of formalism called <u>bounded</u> <u>universal</u> <u>languages</u>.

Examples

The language SR is GR without the negative go to statement. Thus by abbreviation SR is [+1, -1, +, -, $\cdot$, $\neq 0$, =, $\neq$, $\dot{-}$, $\diamond$, DO].

In Constable & Borodin [10], it is proved that SR is an SPL for the class $\mathcal{R}^1$ of primitive recursive functions, i.e., there is an SR program α_i for the function f( ) iff f( )$\varepsilon \mathcal{R}^1$.

(2.4) bounded universal programming languages

Another class of subrecursive formalisms arises naturally both in theory and practice. When computer jobs are submitted for execution on a real machine, they are usually accompanied by a time limit and often by limits on other resources such as memory size and printer lines as well. These limits are provided by the programmer who presumably estimates them on the basis of his knowledge of the program ϕ_i and the input x.

To capture this situation mathematically one introduces the concept of a computing resource, such as time or space. This is most conveniently done axiomatically in the manner of Blum [4]. Given an acceptable indexing $\phi=\{\phi_i(\ )\}$ of $\mathcal{R}_1$ define an abstract computational complexity measure as a listing of partial functions, $m\phi=\{m\phi_i(\ )\}$ such that

A1. $\phi_i(x)$ is defined iff $m\phi_i(x)$ is defined

A2. there is a recursive predicate M such that M(i,x,y) iff $m\phi_i(x)=y$

A measure of particular interest is the time measure, $t\phi$ defined as $t\phi_i(x)$="number of steps in the computation $\phi_i(x)$ if that computation halts, otherwise undefined". See [10] or [6] for a precise account of this measure.

Now the concept of a resource bounded program running on a supervisor system, ϕ_s, can be made precise as follows. The programmer "submits" the triple $<\phi_i,x,y>$ and the supervisor system computes $\phi_s(i,x,y)=$ (if $m\phi_i(x) \leq y$ then $\phi_i(x)$ else e)$\stackrel{def}{=}\phi_i^b(x)$ where $\phi_i^b:\mathbb{N}\to\mathbb{N}\cup\{e\}$, and where "e" is the error message.[7]

[7]It is mathematically cleaner to compute a vector function $\phi_i^b:\mathbb{N}\to\mathbb{N}^2$ where the first component is 0 if $\phi_i(x)\leq y$ and 1 otherwise.

The supervisor system is simply the function ϕ_s. It essentially monitors the computation of ϕ_i and terminates it abnormally if the resource is exceeded.

In the real situation, the bound y is a function of ϕ_i and the input x, say $y=b_i(x)$. Since the bound is determined by the programmer, we can think of $b_i(\ )$ as a recursive function (if we accept Church's Thesis). It now requires no great leap of imagination to let the machine compute b(x) as well as $\phi_i(x)$ when supplied with b_i, ϕ_i, and x.

If the system must calculate b(x) as well as $\phi_i(x)$, it can choose two strategies, sequential or parallel. Either it computes b(x) first and then monitors the running of ϕ_i on x (see Cleave [6] for details), or it computes b and ϕ_i in parallel, (i.e. simulates parallel processing by multiprogramming the two computations), and allows ϕ_i to run as long as b is still running.

The sequential strategy may be very inefficient if b(x) should take more time than $\phi_i(x)$, and unfortunately there is no way to uniformly analyze ϕ_i to determine its minimum possible run-time (see [21] and [10]).

The parallel strategy also requires a certain <u>supervisor system overhead</u> (multiprogramming overhead) and requires a special type of b, but its inefficiency is guaranteed to be bounded, i.e.

$$t\phi_i^b(x) \leq c \cdot \min\{t\phi_i(x), b(x)\}$$

for c the overhead.

The special property of b required is that its run time is close to the value, so that while b is still running, $t\phi_i(x)$ has not exceeded b(x). Such functions are called honest.[8] This is arranged by having b be a run-time itself, say $b=t\phi_j$ and using program ϕ_j as a <u>clock</u>, i.e. ϕ_i runs as long as ϕ_j is running.

---

[8]Precisely, ϕ_i is <u>h( ) honest</u> wrt $m\phi$ at x iff $m\phi_i(x) \leq h(\phi_i(x), x)$. We require honesty for all x. In the literature, ϕ_i is h( )-honest iff it is h( )-honest except on a finite set.

A clock with respect to wrt the operating system ϕ_s described above is a program ϕ_j such that $t\phi_j=b$.[9] (Unless $t\phi_j=\phi_j$ the set of time bounds and the set of clocks wrt ϕ_s are not necessarily the same.)

If an algorithm for b has the property that the output variable, Y, inceases every d steps, $Y_0<Y_d<Y_{2\cdot d}<\ldots<Y_{n\cdot d}$, a supervisory system can give b a slice of d steps during multiprogramming and then run ϕ_i for $Y_{n\cdot d}$ steps. Programs which have this <u>internal</u> <u>monotonicity</u> can serve as their own clocks on the right supervisory system. (Note, d may depend on i.)

The discussion suggests that the time bounds, $\mathcal{B}$, for a parallel system should satisfy; (a) each b_i is known to be total, i.e. $\mathcal{B}=\{b_0,b_1,\ldots\}$ is an r.e. subset of $\mathcal{R}$; (b) each b_i possesses a clock wrt ϕ_s; (c) $\mathcal{B}$ is large enough to bound the run times of the class of programs of interest. A particularly simple set of bounds for $\mathcal{R}^1$ is the following

function	canonical program in SR	program name
$f_0(x) = x+1$	$\boxed{X \leftarrow X+1}$	f_0
$f_1(x) = 2\cdot x$	DO X X←X+1 END	f_1
$f_{n+1}(x)=f_n^{(x)}(x)$	DO X f_n END	f_{n+1}

Where for any $f:\mathbb{N}\to\mathbb{N}$ the iterates of f are defined by $f^{(0)}(x)=x$, $f^{(p+1)}(x)=f(f^{(p)}(x))$.

Let $AG=\{f_n^{(p)}\mid n,p=0,1,2,\ldots\}$ be the set of <u>Ackermann-Grzegorczyk</u> bounds. They satisfy (a) & (b) because they are internally monotone and can be used as their own clocks. The fact that they satisfy (c) for $\mathcal{R}^1$ is

---

[9] Generally given a supervisor system ϕ_s with parallel bounding and a set of time bounds $B=\{b_0,b_1,\ldots\}$, a program ϕ_j is a clock for b_k if $\phi_s(j,i,x)=(\text{if } t\phi_i(x)\le b_k(x) \text{ then } \phi_i(x) \text{ else } e)$ and $t\phi_s(j,i,x)\le c\cdot\min\{b_k(x),t\phi_i(x)\}$.

shown in [10] among other places.

Given a system function ϕ_s and a set of time bound programs, B, the class of all pairs (b,ϕ_i) for $b\varepsilon B$ is called a <u>resource bounded formalism</u>. The class of functions computed is denoted $\mathcal{R}_{\mathcal{B}}$ where $\mathcal{B}$ is the set of functions computed by B.

If ϕ_s is sequential rather than parallel, then any $t\varepsilon\mathcal{R}$ can serve as a bound. The class $\{\phi_k(\ )|t\phi_k(x)\leq t(x)$ for all $x\}$ is denoted $\mathcal{R}_t$.

The classes $\mathcal{R}_t$ are called (strict) <u>time complexity classes.</u> They are generalized in §3 to any measure.

If $T=\max\{n,t(\ )\}|n\varepsilon\mathbb{N}\}$ then $\mathcal{R}_T$ is the <u>time complexity class</u>.

The formalisms (sequential or parallel when they exist) for $\mathcal{R}_{\mathcal{B}}$, $\mathcal{R}_T$ will be especially important in what follows.

From the viewpoint of computational complexity theory the resource bounded formalisms are simply <u>presentations</u> of complexity classes (see Lewis [17] and Landweber & Robinson [16]).

A <u>presentation</u> for a class of $\mathcal{S}\subseteq\mathcal{R}$ is a set of programs $S=\{\phi_{s(i)}\}$ such that

(i) $\phi_{s(i)}(\ )\varepsilon\mathcal{S}$

(ii) if $f(\ )\varepsilon\mathcal{S}$ then $\exists i$ such that $\phi_{s(i)}=f(\ )$.

That is, S contains names for all and only members of $\mathcal{S}$. The programs $\phi_{s(i)}$ are called S-programs for $\mathcal{S}$. A presentation is recursively enumerable (r.e.) iff S is r.e. and is recursive iff S is recursive. It is <u>unbounded</u> iff s( ) has infinite range.

Notice that ST is a recursive presentation of $\mathcal{R}^1$. Below we give examples of bounded UPL's. Some of these are also examples of recursive presentations of classes, $\mathcal{R}_{\mathcal{B}}$.

<u>Examples</u>

(1) We define a formalism for $\mathcal{R}_T$. Let P be an infinite subset of $\mathbb{N}$

and S a presentation of P. (Thus $S \subseteq \{\phi_i^0\}$). Let $T(S) = \{<c_i, t\phi_i> | c_i \in S\}$. Notice that $\mathcal{R}_{T(S)} = \mathcal{R}_{T(S')}$ for S, S' presenting any infinite subsets of $\mathbb{N}$. Let $GR_{T(S)}$ denote the set of GR programs bounded by clocks from T(S). More precisely,

<bounded program> is <<clock>:<program>>

and <clock> is any member of T(S).

The purely numerical functions defined by the formalism $GR_{T(S)}$ are $\mathcal{R}_{T(S)}$. But $GR_{T(S)}$ is not an r.e. presentation of $\mathcal{R}_{T(S)}$ because it contains programs for functions not in $\mathcal{R}_t$, namely those with error messages. It is not possible to recursively extract those functions with error messages, but it is possible to give a recursive presentation of $\mathcal{R}_{T(S)}$ (see [14] and [16]). For some complexity measures, like space, the bounded formalism is a presentation (given the right t). For the purposes of this example, we do not need the intricacies of presentations, and we leave the matter to the articles mentioned.

(2) Our next example of a bounded UPL uses the Ackermann-Grzegorczyk (AG) bounds, $\{f_n^{(p)}\}$

<clock> is any one of $f_n^{(p)}$

<bounded program> is <<clock>:<program>>

where a <program> is a GR program. A parallel type supervisor is used to interpret the bounded programs.

Let this language be denoted GR_ω and the language having only the clocks $f_n^{(p)}$ be denoted GR_n. It is well known (Grzegorczyk [13]) that GR_ω is a recursive presentation for $\mathcal{R}^1$, (and for $n \geq 2$, GR_n presents the n+1st at Grzegorczyk's class $\mathcal{E}^{n+1}$, see [13] or [10]).

(3) Extensions of the class of AG boundes have been studied extensively and used to define subrecursive classes and bounded universal formalisms (see Robbin [25], Constable [9] and Ritchie [24]). These extensions are usually based on ordinal indexings of honest strictly monotone functions. Such sequences are called <u>spines</u> here. A spine $\{f_\alpha(\ )\}$ is called an <u>iteration spine</u> iff

$$f_{\alpha+1}(x) = f_\alpha^{(x)}(x)$$

for all x and all α. If at limit ordinals α, the definition is $f_\alpha(x)=f_{\alpha_x}(x)$ where $\alpha_n \to \alpha$ (so that $\{\alpha_n\}$ is a fundamental sequence for α), then when standard fundamental sequences are known up to β and used in defining f_α at limit ordinals, the sequence $\{f_\alpha\}$ for $\alpha < \beta$ is called a standard spine.[10]

It is known that there are standard iteration spines up to large ordinals which extend the AG spine. In Robbin [25] a ω^ω-standard iteration spine is used to obtain the classes of multiple recursive functions $\mathcal{R}^n$ (discussed by Péter) from resource bounded Turing machines. In Constable [8], an ε_0-standard iteration spine is used to extend the $\mathcal{R}^n$ hierarchy using resource bounded RASP's.

Let GR_β for $\beta<\varepsilon_0$ be the bounded GR language with clocks coming from $\{f_{\alpha+1}^{(p)}\}$ $\alpha<\beta$ as defined in [8]. No one yet has discovered a natural class of SPL's for GR_α when $\alpha>\omega^\omega$. But for $\alpha=\omega^n$ the Loopstack languages of Contable [9] and the $Loop_n$ languages of Ritchie [24] are equivalent to GR_α which presents $\mathcal{R}^n$.

---

10 In the Conference record of the Second Annual ACM Symposium on Theory of Computing, this author and Bass & Young [2] independently reported results on a new type of subrecursive hierarchy based on the union and honesty theorems of [19]. This hierarchy suggests a new and interesting type of spine called a minimal spine.

§3 Size of Programs

(3.1) axioms and basic theorem

According to Blum [3] the notion of program size can be axiomatically defined over any acceptable indexing $\{\phi_i(\ )\}$ by specifying a size function $|\,|: \mathbb{N} \to \mathbb{N}$ which satisfies[11]

SA1. $|\,|: \mathbb{N} \to \mathbb{N}$ is recursive

SA2. There is a recursive function $b(\ )$ such that $b(y)$ is the cardinality of $|y|^{-1}$.

The size of program ϕ_i is denoted $|i|'$ or $|\phi_i|$. It is the value of the size function. Notice that axiom SA2 implies there are only finitely many programs of any given size y.

As an example of a size function, consider the length (number of statements) of a GR program. To force axiom SA2 programs are put in canonical form. Let Var $(\pi)=\{v_1,\ldots,v_p\}$ be the variables occuring in π and let Lab $(\pi)=\{\ell_1,\ldots,\ell_q\}$ be the labels in π. The simple form results from requiring that if there are p members of Var(π) and q members of Lab(π), then $v_i=V_i$ and $\ell_i=j$, $i=1,\ldots,p$, $j=1,\ldots,q$. For a program in simple form, its length is the number of statements in the program.

The canonical form programs of GR can be listed in order of length (within a fixed length they are ordered lexicographically). Assume $\{\phi_i\}$ is listed in such order. The length measure for GR applies directly to SR. The corresponding definition of the length of a program $<t,\phi_i>$ in a resource bounded formalism with system function ϕ_s is $|t|+|\phi_i|+|\phi_s|$. This can be written $|<t,\phi_i>|$, and we shall drop the term $|\phi_s|$ since it is only a small additive constant (about 100).

The Blum size result is now given a new proof. First consider length and time.

Theorem 1 (Blum)

---

[11] Let $|y|^{-1}=\{x|\,|x|=y\}$

There exists a recursive function h( ) such that if

1. s( ) is a recursive function with infinite range and
2. f( ) is any recursive function,

then there are i,j (uniform in f,s) such that

(i) $\phi_i(x)=\phi_{s(j)}(x)$ for all x

(ii) $f(|i|) < |s(j)|$

(iii) if $\phi_{s(j)}(x)$ is defined, then $t\phi_i(x) \leq h(x,t\phi_{s(j)}(x))$ e.f.s.

Discussion of proof: Let ϕ^2_u be the universal program for the one argument sublist $\{\phi^1_i\}$. Then a uniform way to compute $\phi_{s(i)}(x)$ is $\phi^2_u(s(i),x)$. Notice $|\phi^2_u(s(i),x)| = |u|+|s|+|i|$ which is simply c+i. As i increases linearly, the increase in $|s(i)|$ depends on the growth of s, which can be large. This suggests that (ii) can be made to hold by replacing s(i) with fast growing subsequence of $|s(i)|$.[12] We make this idea precise and get a proof.

Proof:

(1) Let $s^1(i)$ satisfy

$$|s^1(i)|>f(2\cdot i)$$

such an s^1 can be defined from s by

$$s^1(i)=s(\mu z\{|s(z)|>f(2\cdot i)\})$$ [13]

because s has infinite range and (by axiom 2) there are only finitely many programs of size y for any y.

(2) Now let $\phi^1_{\beta(i)}(x) = \phi^2_u(s^1(i),x)$ (β results from the S-m-n theorem), then by definition of u, $\phi^1_{\beta(i)}(x) = \phi^2_{s^1(i)}(x)$ for all x, and $|\phi_{\beta(i)}|= |u|+|s^1|+i \overset{def}{=} c^1+i$, and since $f(2\cdot c^1)<|s^1(c^1)|$ it follows that

(i) $\phi_{\beta(i)}(x) = \phi_{s^1(c^1)}(x)$ for all x.

(ii) $f(|\beta(c^1)|) = f(2\cdot c^1)<|s^1(c^1)|$.

---

[12]This subsequence improvement in the author's original idea is due to John Hopcroft.

---

[13]$\mu z\{P(z)\}$ means "the least z such that P(z)" for any predicate P( ).

(3) From this construction it is easy to see that $\phi_{\beta(i)}$ has nearly the same run time as $\phi_{s^1(i)}$ e.f.s., because the program $\phi_{\beta(i)}$ simply computes $s^1(i)$ first, at some fixed cost, say d, and then uses ϕ_u to simulate $\phi_{s^1(i)}$. The simulation cost, $\hat{h}(i,x,y)$ is fixed for the formalism and $h(x,y) = \max_{i \leq x} \hat{h}(i,x,y)$. Q.E.D.

Remark: For GR the simulation cost is e.f.s. bounded by x^2.

The result follows for all size measures, $|\,|$, by using the recursive relationship among them (see Blum [3]) and for all complexity measures by using the recursive relationship among them (see Blum [4]).

This proof suggests a refined theorem. The idea was basically that GR could use its universal function to shorten programs. So any formalism expressing a universal function for the sequence $\phi_{s(i)}$ should also allow shorter programs. We formalize this idea below.

Given a presentation $S=\{\phi_{s(i)}\}$ as above, call $\mathcal{S}'$s a <u>size class</u> for S iff

(i) $\mathcal{S}'$ is closed under composition and the bounded least number operation.

(ii) $\mathcal{S}'$ contains the functions $|\,|$, $s(\,)$ and $|\,|^{-1}$, $s^{-1}(\,)$ where
$|y^{-1}| = \max\{x \mid |x|=y\}$
$s^{-1}(y) = \min\{x \mid s(x) \geq y\}$

(iii) $\mathcal{S}'$ contains a universal function, ϕ_S. for S, i.e.
$\phi_S(i,x) = \phi_{s(i)}(x)$ for all x, all i.

We then have

Theorem 2 Given any unbounded presentation $S=\{\phi_{s(i)}\}$, then for all presentations $\hat{S}'$ of any size class $\mathcal{S}'$ of S there is an $h(\,) \in \mathcal{R}_2$ such that all $f(\,)$ in $\mathcal{S}'$ there are i,j (uniform in f) satisfying

(i) $\phi_i(\,) = \phi_{s(j)}(\,)$, $\phi_i \in S'$

(ii) $f(|\phi_i|) < |\phi_{s(j)}|$

(iii) $t\phi_i(x) \leq h(x, t\phi_{s(j)}(x))$ e.f.s.

Proof: As before define

$$s'(i) = s(\mu z\{|s(z)|>f(2\cdot i)\}).$$

Notice that $s(z) \leq |f(2\cdot i)|^{-1}$ since $||^{-1}$ is monotone over $\{\phi_i\}$.

Also $z \leq s^{-1}(|f(2\cdot i)|^{-1})$

because $s^{-1}(|f(2\cdot i)|^{-1})$ = least y such that $s(y) \geq |f(2\cdot i)|^{-1}$.

Thus $|s(y)| \geq ||f(2\cdot i)|^{-1}| \geq f(2\cdot i)$.

So the function $t(i)=\mu z \leq s^{-1}(|f(2\cdot i)|^{-1})\{|s(y)|>f(2\cdot i)\}$ belongs to $\mathcal{S}'$. Thus, $s(t(i)) = s'(i)$ belongs as well as $\phi_S(s'(i),\_)$. Given any presentation of $\mathcal{S}'$, say S', it has a name for $\phi_S(s'(\ ),\_)$. Call it $\hat{\phi}$. Now proceed as in the proof of Theorem 1 with $\hat{\phi}$ for $\phi(s'(\ ),\_)$. Q.E.D.

Since classes $\mathcal{R}_{\{f_\alpha(p)\}}$ (abbreviated $\mathcal{R}_\alpha$) are closed under composition and bounded μ (see [8] or [25]) and $\mathcal{R}_{\alpha+1}$ contains the universal function for $\mathcal{R}_\alpha$, and $||^{-1}$ and $h(\ )$ are primitive recursive for the indexing of GR, we can state a simple corollary for the classes $\mathcal{R}_\alpha$.

Given $\{\phi_{s(i)}\} \subseteq GR_\alpha$ let $\hat{\alpha}$ be the least ordinal such that $\mathcal{R}_{\hat{\alpha}}$ contains $s^{-1}(\ )$.

Theorem 3 There is a primitive recursive $h(\ )$ such that for any unbounded $\{\phi_{s(i)}\} \subseteq GR_\alpha$ and for all $f(\ ) \in \mathcal{R}_{\hat{\alpha}}$ there are i,j such that

(i) $\phi_i(\ ) = \phi_{s(j)}(\ )$ and $\phi_i(\ ) \in GR_{\hat{\alpha}}$

(ii) $f(|\phi_i|) < |\phi_{s(j)}|$

(iii) $t\phi_i(x) \leq h(x, t\phi_{s(j)}(x))$ e.f.s.

(3.2) <u>presentation of relatively shortest programs</u>

Let $S=\{g_0, g_1, \ldots\}$ be a presentation of $\mathcal{S} \subseteq \mathcal{R}$, then $\hat{S}=\{\hat{g}_0, \hat{g}_1, \ldots\}$ is the <u>presentation of shortest S-programs</u> iff for any g_i there is a $\hat{g}_j$ such that

(i) $g_i(\ ) = \hat{g}_j(\ )$

(ii) $|\hat{g}_j| \leq |g_i|$.

We let the notation $\hat{S}$ remain standard hereafter.

Theorem 4 If S is an ordered r.e. presentation of $\mathcal{S} \subseteq \mathcal{R}$, then $\hat{S}$ is r.e.

Proof: Let $S=\{g_0, g_1, \ldots\}$. A procedure is described enumerating $\hat{S}=\{\hat{g}_0, \hat{g}_1, \ldots\}$, (but not in order of size). Then we prove that the procedure works correctly. First the procedure:

On the 0th step put g_0 in $\hat{S}$, i.e. $\hat{g}_0=g_0$. Notice g_0 is shortest for $g_0(\ )$ because S is an ordered presentation.

On the n-th step compute all of $g_i(x)$ for all $i,x \leq n$. Put in $\hat{S}$ any g_i such that

(i) g_i is not already in $\hat{S}$ and

(ii) for all $j \leq n$, $j \neq i$, $\exists y \leq n$ $g_i(y) \neq g_j(y)$

These two conditions simply state that g_i is a new function not equal to any of $g_0, \ldots, g_n$. Because $\hat{S}$ is listed in order of size, it should be obvious that if g_i goes into $\hat{S}$, it is a shortest program for the function $g_i(\ )$.

More precisely, for any $f(\ ) \in \mathcal{S}$, let g_f be its first occurrence in S. Because S is ordered, g_f is a shortest program for $f(\ )$, i.e. if $g_j(\ )=f(\ )$ then $|g_f| \leq |g_j|$. So we need only show that g_f goes into $\hat{S}$. Since g_f is the first occurrence of $f(\ )$ on S, for all $j \leq f$ $\exists$ y $g_f(y_j) \neq g_f(y_j)$. Let $m=\max\{y_j\}$ for $j=0,\ldots,f$. Then at stage m conditions (i) and (ii) are met and g_f goes into $\hat{S}$, i.e. $\exists k$ so that $\hat{g}_k=g_f$. Q.E.D.

Before applying this theorem to SR we prove the interesting fact that $\hat{S}$ cannot be ordered unless equivalence is decidable in S. This fact is important for SR.

Theorem 5. Let S be an ordered r.e. presentation of $\mathcal{S}$, then the following are equivalent.

(a) $\hat{S}$ is ordered

(b) $\hat{S}$ is recursive

(c) equivalence in S is decidable.

Proof: We show (a) and (b) are equivalent and then that (b) and (c) are.

(1) By Theorem 2 $\hat{S}$ is r.e. so to show it is recursive we need only show that it can be listed in order of size. Thus (a) and (b) are equivalent. Notice also that S is a recursive presentation of $\mathcal{S}$.

(2) (if part) If equivalence is decidable, then given any $\phi_i \varepsilon S$ to decide whether $\phi_i \varepsilon \hat{S}$, simply check whether $\phi_i(\ )=\phi_j(\ )$ for all ϕ_j which are shorter.

(Only if part) If $\hat{S}$ is recursive, then we claim we can find the shortest S program equivalent ot any $\phi_i \varepsilon S$. Proof: given ϕ_i, first determine whether it belongs to $\hat{S}$. If so we are done. If not, then let $\phi_{i_1}, \phi_{i_2}, \ldots, \phi_{i_n}$ be a list of all $\phi_{i_j} \varepsilon \hat{S}$ which are shorter than ϕ_i. Since $\phi_{i_j} \varepsilon \hat{S}$, at most one of them is a program for ϕ_i. Since $\phi_i \not\varepsilon S$, exactly one must be. Thus keep checking $\phi_{i_j}(x)=\phi_i(x)$ for $x=0,1,2,\ldots$ until only one program remains. This must be the shortest S program for ϕ_i.

Now determine equivalence in S, take any two programs $\phi_i, \phi_j \varepsilon S$ and find their shortest S program, say $\hat{\phi}_i, \hat{\phi}_j$. Then $\phi_i(\ )=\phi_j(\ )$ iff $\hat{\phi}_i=\hat{\phi}_j$. Q.E.D.

To apply these theorems to SPL's, notice that SR is an ordered r.e. presentation of $\mathcal{R}^1$. Also equivalence in SR is not decidable. Likewise the bounded UPL's for $\mathcal{R}_\alpha$ have ordered r.e. presentations. The GR_α are specific examples. Equivalence is not decidable in GR_α for any α.

We now apply Theorems 1-4 to SR with a specific f( ) for dramatic effect. Let the SR indexing of $\{\phi_i\}$ be denoted $\{p_i\}$. Then for application take $S=\{p_i\}$ and denote $\hat{S}$ by $\{\hat{p}_i\}$. Choose $f(x)=2^{x+1000}$ for Theorem 1, and assume that GR simulation of SR on a universal machine for SR costs at most a square. Then there are i,j such that

(i) $\phi_i(\ ) = \hat{p}_j(\ )$

(ii) $2^{|\phi_i|+1000} < |p_\daleth|$

(iii) $t\phi_i(x) \le (t\hat{p}_\daleth(x))^\daleth$ e.f.s.

Such an example makes it appear that SR is grossly "long winded" for expressing primitive recursive functions compared to GR (partly because SR cannot express its own universal function). However notice that SR and $GR_{\hat{\omega}}$ exhibit the same contrast (in this case $\hat{\omega}=\omega+1$). This suggests that for certain practical limitations on f( ), $GR_{\hat{\omega}}$ might already allow nearly the shortest possible expression of SR programs.

Indeed, we claim that the shortest reasonably efficient program for a feasible $\mathcal{R}^1$ function is about the same size as the shortest SR program for it. A similar claim can be made for complexity classes $\mathcal{R}_t$. We make this precise and prove it in the next section. Moreover we show that when examining feasible functions, those whose run times are bounded by some f_α( ) e.f.s., the question of program size reduces to a matter of getting short expressions for large constants. In fact, the idea behind most of these results is nicely illustrated by the constants. One program for the constant n is simply n copies of $X \leftarrow X+1$ (after $X \leftarrow 0$). This program has length n+1. But if $n=2^x$ a shorter program would generate x and calculate 2^x. If $n=2^{\daleth^x}$, then it is even more worth while to compute x, then 2^x, then $2^{\daleth^x}$. If the formalism for $\aleph$ does not allow $2^{\daleth^x}$ nor any function growing that fast, then some of its program, namely those for $n=2^{\daleth^x}$, will be shorter than programs in a formalism allowing $2^{\daleth^x}$.

Since any subrecursive formalism has a bounded growth rate and a universal formalism does not, there will always be constants which are uneconomical to express in that formalism. In a sense, to be made precise below, the expressive power of the formalism with a sufficient basis is related to the growth rate of its functions. We shall examine these phenomena briefly in section (3.4).

(3.3) <u>size-efficiency exchange</u>

It is easy to show that the shortest GR program for a function f( ) is sometimes drastically inefficient. Say that ϕ_i is (absolutely) <u>shortest</u> for ϕ_i( ) iff for all $\phi_\daleth( )=\phi_i( )$, $|\phi_\daleth| \le |\phi_i|$.

Theorem 6. For all $a(\ ), r(\ ) \varepsilon \mathcal{R}$ there is a $f(\ ) \varepsilon \mathcal{R}$ such that if ϕ_i is the shortest GR program for $f(\ )$, then there is a $\phi_j(\ )=f(\ )$ and $r(t\phi_j(x)) \leq t\phi_i(x)$ e.f.s., and for any $\phi_k(\ )=f(\ )$, $t\phi_k(x) \geq a(x)$ e.f.s.

Proof: Take $f(\ )$ to be a function with $r(\ )$ speed up known to exist from Blum [4] then apply the speed up to the shortest program. Q.E.D.

Thus there are arbitrarily complex functions for which the shortest program is arbitrarily inefficient. We are interested in the shortest programs only when they are reasonably efficient. To focus on this condition we say that ϕ_i <u>shrinks</u> ϕ_j by s <u>within h( )</u> <u>efficiency</u> iff $|\phi_i|+s=|\phi_j|$ and $t\phi_i(x) \leq h(x,t\phi_j(x))$ for all x.

We shall examine efficient shrinking for bounded UPL's first where the ideas are clear and then carry them over to SPL's. We use the GR_n because they are convenient ways to present the fundamental ideas, which carry over to other bounded formalisms. Recall GR_n is a presentation of $\mathcal{R}_{\{f_n^{(p)}\}}$.

Given a GR_n program it has the form $<f_n^{(p)},\phi_i>$ where $f_n^{(p)}$ is the clock and $\phi_i \varepsilon GR$. The finite GR_n-complexity, $c(\phi_j)$, of $\phi_j \varepsilon GR$ is the least p such that $t\phi_j(x) \leq f_n^{(p)}(x+1)$ for all x.[14] For programs in GR_n the finite complexity will be the parameter p of the clock.

We prove the size-efficiency exchange for the case of GR_2. Notice that $|f_2|=|$DO V; DO V; V V+1; END; END$|=5$.

Theorem 7. Let $\hat{G}R_2=\{\hat{g}_j\}$, where $\hat{g}_j=<f^{(p_j)},\phi_j>$. Then if $\phi_i \varepsilon GR$ shrinks $\hat{g}_j$ by s and has finite GR_2 complexity, then

$$5\cdot(c(\phi_i) - p_j) \geq s-5\cdot p_j.$$

Corollary 1: If ϕ_i shrinks $\hat{g}_j$ by s within f_2 efficiency, then

$$p_j \geq s/5 - 1$$

---

[14] For technical reasons it is better to use $f_n^{(p)}(x+1)$ than $f_n^{(p)}(x)$.

Proof: Notice that if $t\phi_j(x) \leq f_2^{(p_j)}(x)$ for all x, then $c(\phi_j)=p_j$ for otherwise $\hat{g}_j$ would not be shortest. For such functions we can replace p_j by $c(\phi_j)$ in the theorem.

Let $\hat{g}_j$ be $\langle f_2^{(p_j)},\phi_j\rangle$. If ϕ_i shrinks $\hat{g}_j$ within $f_2^{(p)}$ efficiency, for some p, then $\langle f_2^{(p_i)},\phi_i\rangle$ is in the GR_2 formalism where $c(\phi_i)=p_i$. So $|\langle f_2^{(p_i)},\phi_i\rangle| \geq |\langle f_2^{(p_j)},\phi_j\rangle| = |m_j|$ since $\hat{g}_j$ is shortest. This implies $|f_2|\cdot|p_i|+|\phi_i| \geq |f_2|\cdot p_j+|\phi_j|$. Therefore, ϕ_i's advantage in size must be accounted for by the fact that its clock is "built in" and $|f_2|\cdot|p_i|$ is unnecessary in GR. Therefore if ϕ_i saves s statements,

$$|f_2|\cdot|p_i|+|\phi_i| \geq s+|\phi_i| = |f_2|\cdot p_j+|\phi_j|.$$

So since $|f_2|=5$,

$$5\cdot p_i \geq s$$

Thus $5\cdot(p_i-p_j) \geq s-5\cdot p_j$, and $d=p_i-p_j$ represents the difference in finite efficiency of ϕ_i and ϕ_j. This establishes the theorem.

The efficiency of ϕ_i can improve that of ϕ_j iff $s \leq 5\cdot p_j$ and by the theorem hypothesis, $p_i \leq p_j+1$ so in the worst case $5+5\cdot p_j \geq s$ and $p_j \geq (s/5)-1$, which establishes the Corollary. Q.E.D.

According to this theorem, in order to save even 30 statements without degrading efficiency, the uneconomical program must run in time $f_2^{(5)}(x)$ for some $x \geq 1$ in some finite set S. We can calculate (from Theorem 1) an upper bound on x.

The same type of result holds for SR because the clocks, $f_n^{(p)}$, can be used in SR for a parallel simulation of GR functions. For details, see Constable & Borodin [10] Theorem 5.1, but the idea is essentially that SR contains an isomorphic image of GR_n for all n.

In particular, it is shown that GR programs ϕ_i can be put in the normal form

```
DO WHILE H ≠ 0
φ*_i
END
```

where ϕ_i^* is an SR program without DO-loops (called an SR_0 or Loop-free SR program). Bounded formalisms can be built around this normal form by computing a clock ϕ_f with output S, either sequentially or in parallel with DO S; ϕ_i^*; END. Since ϕ_i and ϕ_i^* are close in size, the size of the bounded program depends on the size of the clock. For feasible functions this clock size is small except for the parameter needed for the e.f.s. condition. This parameter (p in $f_2^{(p)}$, c in $\langle c,t\rangle$) is a constant, so the size of the bounded programs depend on the size of constants in the bounded formalism. We shall therefore examine constants in more detail in the next section. First another illustration of their importance.

In the $GR_{T(S)}$ formalism, if $\phi_i \varepsilon GR$ shrinks a $\langle c,t\rangle$ bounded program ϕ_j without infinite loss of efficiency (i.e. $t\phi_j(x) \leq t(x)$ e.f.s.) then there is a least $\overline{c}$ such that

$$t\phi_j(x) \leq \min\{\overline{c}, t(x)\} \quad \text{for all } x.$$

The finite loss of efficiency between ϕ_i and ϕ_j is $\overline{c}-c$.

Theorem 8: Let $\hat{G}R_{T(S)}=\{\hat{g}_i\}$, suppose ϕ_i shrinks $\hat{g}_j$ by s without infinite loss of efficiency. Then if $|\hat{g}_j|=|\phi_j|+|t|+|c|$, the finite loss of efficiency d satisfies $d \geq s-(|c|+|t|)$.

Proof: Similar to Theorem 7.

Thus, the more efficient ϕ_i is, the more complex $\hat{g}_j$ must be. The feasibility of $\hat{g}_j$ i.e. the size of S is a function of the presentation of constants S. The more compact the presentation the more complex $\hat{g}_j$ must be if it can be shrunk at all.

(3.4) the size of constants

For reasons indicated in (3.3) we are interested in the set of GR programs computing constants (no inputs), $\{\phi_1^0\}$. We summarize Theorem [illegible] for them as follows. An infinite set S is immune iff it has no infinite r.e. subsets.

Theorem 9: The set of absolutely shortest GR programs for constants, A, is immune.

Proof: If there were an r.e. subset, then some of its elements could be shrunk (using Theorem 1). Q.E.D.

Corollary 1. No r.e. presentation of $\mathbb{N}$ contains the absolutely shortest programs for all constants.

This offers a simple answer to the question of whether any subrecursive formalism, say GR_α, could contain the absolutely shortest programs for all functions in GR_β for $\beta < \alpha$. It also says that the absolutely shortest programs are arbitrarily complex.

Corollary 2. For any $f(\ ) \varepsilon \mathcal{R}$ there are $n \varepsilon \mathbb{N}$ such that if ϕ_i is the shortest GR program for n, then $t\phi_i(\ ) \geq f(n)$.

Let S be an r.e. presentation of $\mathbb{N}$, then $\hat{S}$ may or may not contain infinitely many of the absolutely shortest programs (members of A). But

Corollary 3. If S is an r.e. presentation of $\mathbb{N}$, then $\hat{S}_\cap A$ is either finite or immune.

It is more difficult to settle the matter. We can show that $\hat{S}_\cap A$ is immune using techniques from the theory of random sequences. But the work is too involved to include here.

In any r.e. presentation of $\mathbb{N}$ there are only a finite number of a certain simple type of constants. This fact is useful in determining the expressive power of a formalism.

Let S be an r.e. presentation of $\mathbb{N}$ and let $\Sigma_S(n) = \max\{\phi_i^0(\ ) \mid |\phi_i^0| < n \ \& \ \phi_i^0 \varepsilon S\}$. This is the Rado function in S for constants. Call $n \varepsilon \mathbb{N}$ S-maximal iff $n \varepsilon M_S = \{\Sigma_S(x) \mid x \varepsilon \mathbb{N}\}$.

Theorem 10: If S is an r.e. presentation of $\mathbb{N}$, then $\hat{S}_\cap A_\cap M_S$ is finite.

Proof: Assume the contrary and consider the two sets of maximal constants $\Sigma_S(2n)$, $\Sigma_S(2 \cdot n+1)$.

One of these sets must contain infinitely many of the absolutely shortest programs. Let $\eta_0, \eta_1, \ldots$ be an enumeration, $|\eta_i| < |\eta_{i+1}|$ of the set that does. Notice that $|\eta_i|>2\cdot i$, therefore by the method of Theorem 1 all programs of length $>c$ can be shrunk by some amount of s which contradicts the assumption. Q.E.D.

This fact can be used to describe the relative expressive power of formalisms. For example, let $\eta^{\beta},\ldots,\eta_p^{\beta}$ and $\eta^{\alpha},\ldots,\eta_q^{\alpha}$ be the absolutely shortest maximal constants for $\mathcal{R}_{\alpha}$ and $\mathcal{R}_{\beta}$ respectively, $\alpha<\beta$, listed in order of value. The ratio $\eta_p^{\alpha}/\eta_q^{\beta}<1$ represents the relative expressive power of β over α. Notice that if α remains fixed and β increased, the ratio $\eta^{\alpha}/\eta^{\beta}$ approaches 0 but "reaches 0" only when $\mathcal{R}_{\beta}=\mathcal{R}$ for some non-constructive β.

The results of (3.3) on size vs efficiency are especially sharp for constants since the only possible exchange is finite. For example, using clocks of the form $f_2^{(p)}(2)$ let the complexity of ϕ_i^0, $c\phi_i^0$, be the least j such that $t\phi^0(\ )\le f_2^{(j)}(2)$. Let $S=\{\phi_i^0 \mid t\phi_i^0(\ )<f_2^{(p)}(2)\ p\in\mathbb{N}\}$. Then S is a recursive presentation. A program $\hat{\phi}_j$ in $\hat{S}$ can be shrunk by s statements using ϕ_i iff the difference in complexity, $d=c\phi_i-c\hat{\phi}_j$ satisfies $d\ge s/5-c\hat{\phi}_j$. Thus again if $\hat{\phi}_j$ can be compressed without loss of efficiency it is difficult, and if it is simple ($c\hat{\phi}_j$ small) and is compressed, the shorter program is much more complex.

ACKNOWLEDGEMENTS

The author would like to thank Professor Allan Borodin for his helpful discussions about the material in this paper and Miss Diane Goolsby for her fine typing.

REFERENCES

[1] Axt, Paul Iteration of Primitive Recursion, Zeitschar. f. Math. Logik u. Grund der Math., II, (1965), 253-55.

[2] Bass, L J and P R Young "Hierarchies based on computational complexity and irregularities of class determining measured sets", Conference Record of Second Annual ACM Symposium on Theory of Computing, Northampton, Mass., 1970, p.37-40.

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

[4] Blum, M "Machine-Independent Theory of the Complexity of Recursive Functions", JACM, 14, 1967, 322-36.

[5] Chaitin, G "On the length of programs for computing finite binary sequences", JACM, 14, 1966, 547-569.

[6] Cleave, John P "A Hierarchy of Primitive Recursive Functions", Zeitschar f. Math Logik and Grund. der Math., 9, 1963, 331-345.

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

[8] Constable, Robert L Extending and Refining Hierarchies of Computable Functions. Computer Science Technical Report 25, University of Wisconsin, 1968.

[9] Constable, Robert L "Subrecursive Programming Languages for $\mathcal{R}^n$" Computer Science Technical Report 70-53, Cornell University, 1970. (To appear in Proceedings, Symposium on Computers and Automata, Vol XXI, Polytech. Institute of Brooklyn, MRI Symposia Series.)

[10] Constable, Robert L & Allan Borodin "On the Efficiency of Programs in Subrecursive Formalisms, Conference Record of Tenth Annual Symposium on Switching & Automata Theory, 1970, 60-67.

[11] Davis, M Computability and Unsolvability, New York, 1958.

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

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

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

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

[16] Landweber, L H & E L Robertson Recursive Properties of Abstract Complexity Classes, Conference Record Second Annual ACM Symposium of Theory of Computing, Northampton, Mass., 1970, 31-36.

[17] Lewis, F D Decision Problems for Complexity Classes of Computable Functions, Conference Record of Second Annual ACM Symposium on Theory of Computing, Northampton, Mass, 1970, 22-30.

[18] McCarthy, John "A Basis for A Mathematical Theory of Computation," Computer Programming and Formal Systems, Amsterdam, 1963, 30-70.

[19] McCreight, E M and A R Meyer "Classes of computable functions defined by bounds on computation", ACM Symposium on Theory of Computing, 1969, 79-88.

[20] Meyer, A and P C Fischer "On computational speed-up" IEEE Conference Record, 9th Annual SWAT, 1968, 351-355.

[21] Meyer, A R and D M Ritchie "The complexity of Loop programs", Proceedings 22 National ACM Conference, 1967, 465-470.

[22] Minsky, M Computation, Finite and Infinite, Prentice-Hall, Englewood Cliffs, 1967.

[23] Péter, Roza Recursive Functions, 3rd ed., New York, 1967.

[24] Ritchie, D M Program structure and computational complexity, PhD Thesis, Harvard University, 1968.

[25] Robbin, J W Subrecursive Hierarchies, PhD Thesis, Princeton University, 1965.

[26] Robinson, R M "Primitive Recursive Functions", Bull. AMS 53 1947, 915-942.

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

[28] Scott, Dana "Some Definitional Suggestions for Automata Theory", JCSS, 1, 1967, 187-212.

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