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
ABSTRACT

§ Introduction ................................................. 1

§ 2 Programming Languages
    (2.1) notation........................................... 4
    (2.2) universal programming languages............. 4
    (2.3) subrecursive programming languages.......... 7
    (2.4) bounded universal formalisms............... 8

§ 3 Size of Programs
    (3.1) axioms and basic theorem.................... 14
    (3.2) presentation of relatively shortest programs... 17
    (3.3) size-efficiency exchange.................... 20
    (3.4) the size of constants....................... 23

REFERENCES
Subrecursive Programming Languages II
On Program Size

Robert L Constable
Cornell University

ABSTRACT

Programming languages which express programs for all computable (recursive) functions are called universal, those expressing programs only for a subset are called subrecursive programming languages, 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

---

1The 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 \( \phi: \mathbb{N}^m \rightarrow \mathbb{N} \) 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, \( \mathcal{J} \subseteq \mathcal{R} \). 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, \( L_{\mathcal{J}} \), can be designed which express algorithms only for the functions in a subrecursive class \( \mathcal{J} \). 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 \( \mathcal{C} \) of elementary, or even its subclass \( \mathcal{C}^2 \) of primary 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 \( \mathcal{C}^n \), the multiply recursive functions of Péter, based on the notion of a stack. In [9] Constable defined another class of language for \( \mathcal{C}^n \) 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 significant 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 infinitely often (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.

"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 formalisms, 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) notation

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ and let $\mathcal{J}_n$ 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 $\mathcal{J}_n$ and lower case Greek letters, $\alpha, \phi, \psi$, for names of elements of $\mathcal{P}_n$. Here we use expressions like $f(\ )$, $\alpha(\ )$, 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.

$\mathcal{J}_n \subseteq \mathcal{P}_n \subseteq \mathcal{PR}_n \subseteq \mathcal{R}_n$

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

(2.2) universal programming languages\(^5\)

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 formalism for $\mathcal{PR}$ will be a formal language, $L$, whose elements are partial function names, $\phi_i$, and a semantic mapping onto $\mathcal{PR}$, i.e.

$\langle \cdot \rangle : L \rightarrow \mathcal{PR}$

The map takes $\phi_i$ to $\phi_i(\ )$. The formalism is called acceptable iff there is a computable encoding of function names, $\phi_i$, 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 acceptable indexing 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 \(\phi\)-map, \(\phi(i)\) is often written \(\phi_i\), any many authors identify \(\phi_i\) and \(\phi_i();\) reserving the encoding \(i\) to denote the function name, \(\phi_i\).

High level programming languages (like Algol) with their semantics are formalisms. They are usually thought of in terms of a grammar 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 straightforward, and there is ample treatment in the literature.

Examples

(1) UPL, \(G_3\)

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

There are two types of statement schemes, \textit{assignments} and \textit{conditionals}

\[
\text{<assignment scheme>} = v + f(v) \quad \text{<conditional scheme>} = \text{if } p(v) \text{ then } \ell
\]

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

\(^6\)In a more constructive theory, the class \(\mathcal{P} \subseteq\) cannot be used without specifying a formalism for it. So constructively our definition requires a standard formalism for \(\mathcal{P} \subseteq\) 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 \begin{align*}
& V_1 + F_1(V_1) \\
& \text{if } P_1(V_1) \text{ then } 1 \\
& V_2 + F_2(V_2) \\
& \text{if } P_2(V_2) \text{ then } 3
\end{align*}

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 $f_i(\ ) : \mathbb{N} \rightarrow \mathbb{N}$, so $v + f(v)$ means "assign to variable $v$ the result of applying function $f(\ )$ to the contents of variable $v$", and $p$ denotes a predicate $p(\ ) : \mathbb{N} \rightarrow \{T,F\}$ and so if "$p(v)$ then $\ell_i$" means "if $p(v)$ is true then go to statement labelled $\ell_i$ as the next instruction, otherwise go to the immediately following statement."

To assign a program $\phi_i$ to a partial function $\phi_i(\ )$ we specify input variables $v_1, \ldots, v_n$ and an output variable $v$. Then $\phi_i$ computes $\phi_i(x_1, \ldots, x_n)$ iff when $\phi_i$ is run with input variables $v_i$ containing $x_i$, it halts with value $\phi_i(x_1, \ldots, x_n)$ in $v$ iff $\phi_i(x_1, \ldots, x_n)$ 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 $f(x) = x \cdot 2$, and $P_1$ as $p(x)$ iff $x \neq 0$. These are abbreviated $+1$, $\cdot 2$, $\neq 0$, respectively, so $G_3$ is denoted by the base $[+1, \cdot 2, \neq 0]$. 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 $<\text{assignment}> = v + f(v_1, \ldots, v_n)$. The specific functions and predicates used are $f_1(x,y) = x + y$, $f_2(x,y) = x \cdot y$, $f_3(x,y) = x \cdot y$, $p_1(x,y)$ iff $x = y$, $p_2(x,y)$ iff $x \neq y$.

We abbreviate by $+, \cdot, \ldots, =, \neq$. 
We also separate the go to's from the conditional.

<go to +>  go to +\ell  abbreviated \dagger
<go to ->  go to -\ell  abbreviated \dagger

The \dagger signs tell the direction of the label \ell in relation to the go to itself. Thus go to +\ell means "find \ell below this location" and go to -\ell means "find \ell 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 \(\triangleleft\).

Finally, we add an iterative type statement.

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

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

\[\tilde{v} + v\]
1 if \(\tilde{v} \neq 0\) then 2
\[\tilde{v} + \tilde{v}/= 1\]
go to 1

2

where \(\tilde{v}\) does not appear in either \(\pi\) or \(\hat{\pi}\). We abbreviate the iterative by DO.

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

\([+1, -1, +, -, \cdot, \neq 0, =, \neq, \dagger, \triangleleft, \text{DO}].\]

(2.3) subrecursive programming languages

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

\((\_): L \rightarrow \mathcal{F} \subseteq \mathcal{R}\)

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 "SubRecursive", in the next part a different type of formalism called bounded universal languages.
Examples

The language SR is GR without the negative go to statement. Thus by abbreviation SR is \([+1, -1, +, -, \cdot, \neq 0, =, \neq, \downarrow, \Box, \text{DO}]\).

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

(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 \textit{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 \(\phi_i\) 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 \(R_1\) define an \textit{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) = \text{"number of steps in the computation } \phi_i(x) \text{ 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, \(\phi_s\), can be made precise as follows. The programmer "submits" the triple \(<\phi_i, x, y>\) and the supervisor system computes \(\phi_s(i, x, y) = (\text{if } m_{\phi_i}(x) \leq y \text{ then } \phi_i(x) \text{ else } e)\) where \(\phi_i : \mathbb{N} \rightarrow \mathbb{N} \cup \{e\}\), and where "e" is the error message.\(^7\)

\(^7\)It is mathematically cleaner to compute a vector function \(\phi^b_{\phi_i} : \mathbb{N} \rightarrow \mathbb{N}^2\) where the first component is 0 if \(\phi_i(x) \leq y\) and 1 otherwise.
The supervisor system is simply the function \( \phi_s \). It essentially monitors the computation of \( \phi_i \) and terminates it abnormally if the resource is exceeded.

In the real situation, the bound \( y \) is a function of \( \phi_i \) 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 \( b_i, \phi_i, \) 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 \( \phi_i \) on \( x \) (see Cleave [6] for details), or it computes \( b \) and \( \phi_i \) in parallel, (i.e. simulates parallel processing by multiprogramming the two computations), and allows \( \phi_i \) 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 \( \phi_i \) to determine its minimum possible run-time (see [21] and [10]).

The parallel strategy also requires a certain supervisor system overhead (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 \( \phi_j \) as a clock, i.e. \( \phi_i \) runs as long as \( \phi_j \) is running.

---

\(^8\)Precisely, \( \phi_i \) is \( h(\_\_\_\_\_\_) \) honest wrt \( m_{\phi} \) at \( x \) iff \( m_{\phi}(x) \leq h(\phi \_\_\_\_\_\_, x) \). We require honesty for all \( x \). In the literature, \( \phi_i \) is \( h(\_\_\_\_\_\_) \) honest iff it is \( h(\_\_\_\_\_\_\_\_) \)-honest except on a finite set.
A clock with respect to wrt the operating system $\phi_s$ described above is a program $\phi_j$ such that $t\phi_j = b_j$. (Unless $t\phi_j = \phi_j$ the set of time bounds and the set of clocks wrt $\phi_s$ are not necessarily the same.)

If an algorithm for $b$ has the property that the output variable, $Y$, increases every $d$ steps, $Y_0 < Y_d < Y_{2d} < \ldots < Y_{nd}$, a supervisory system can give $b$ a slice of $d$ steps during multiprogramming and then run $\phi_i$ for $Y_{nd}$ steps. Programs which have this internal monotonicity 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 $\phi_s$; (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

<table>
<thead>
<tr>
<th>function</th>
<th>canonical program</th>
<th>program name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0(x) = x + 1$</td>
<td>[x+x+1]</td>
<td>$f_0$</td>
</tr>
<tr>
<td>$f_1(x) = 2 \cdot x$</td>
<td>[DO \ x\br X+x+1\br END]</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$f_{n+1}(x) = f_n(x)$</td>
<td>[DO \ x\br f_n\br END]</td>
<td>$f_{n+1}$</td>
</tr>
</tbody>
</table>

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 $\mathcal{AG} = \{f(p) \mid n, p = 0, 1, 2, \ldots\}$ be the set of Ackermann-Grzegorczyk 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

---

9Generally given a supervisor system $\phi_s$ with parallel bounding and a set of time bounds $B = \{b_0, b_1, \ldots\}$, a program $\phi_i$ is a clock for $b_k$ if $\phi_i(j, i, x) = (\text{if } t\phi_i(x) \leq b_k(x) \text{ then } \phi_j(x) \text{ else } e)$ and $t\phi_i(j, i, x) \leq c \cdot \min\{b_k(x), t\phi_i(x)\}$. 
shown in [10] among other places.

Given a system function $\phi_s$ and a set of time bound programs, $B$, the class of all pairs $(b, \phi_i)$ for $b \in B$ is called a resource bounded formalism. The class of functions computed is denoted $R_\beta$ where $\beta$ is the set of functions computed by $B$.

If $\phi_s$ is sequential rather than parallel, then any $t \in R$ can serve as a bound. The class $\{\phi_k(\cdot) | t\phi_k(x) \leq t(x) \text{ for all } x\}$ is denoted $R_t$.

The classes $R_t$ are called (strict) time complexity classes. They are generalized in §3 to any measure.

If $T = \max\{n, t(\cdot) | n \in \mathbb{N}\}$ then $R_T$ is the time complexity class.

The formalisms (sequential or parallel when they exist) for $R_\beta$, $R_T$ will be especially important in what follows.

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

A presentation for a class of $\mathcal{J} \subseteq R$ is a set of programs $S = \{\phi_s(i)\}$ such that

(i) $\phi_s(i)(\cdot) \in \mathcal{J}$

(ii) if $f(\cdot) \in \mathcal{J}$ then $\exists i$ such that $\phi_s(i) = f(\cdot)$.

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

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

Examples

(1) We define a formalism for $R_{-T}$. Let $P$ be an infinite subset of $\mathbb{N}$. 
and $S$ a presentation of $P$. (Thus $S \subseteq \{\phi_1^0\}$). Let $T(S) = \{ \langle c_i, t^i_1 \rangle | c_i \in S \}$. 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,

$$<\text{bounded program}> \quad \text{is} \quad <\langle \text{clock} \rangle : <\text{program}>>$$

and $<\text{clock}>$ 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, $\{f_n^{(p)}\}$

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

$$<\text{bounded program}> \text{ is } <\langle \text{clock} \rangle : <\text{program}>>$$

where a $<\text{program}>$ is a GR program. A parallel type supervisor is used to interpret the bounded programs.

Let this language be denoted $GR_\omega$ and the language having only the clocks $f_n^{(p)}$ be denoted $GR_n$. It is well known (Grzegorczyk [13]) that $GR_\omega$ is a recursive presentation for $\mathcal{R}_1$, (and for $n \geq 2$, $GR_n$ presents the $n+1$st at Grzegorczyk's class $\mathcal{C}_{n+1}$, see [13] or [10]).

(3) Extensions of the class of AG bounds 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 spines here. A spine $\{f_\alpha(\ )\}$ is called an iteration spine iff

$$f_{\alpha+1}(x) = f_\alpha(x)$$
for all $x$ and all $\alpha$. If at limit ordinals $\alpha$, the definition is $f_\alpha(x)=f_{\alpha_n}(x)$ where $\alpha_n \rightarrow \alpha$ (so that \( \{\alpha_n\} \) is a fundamental sequence for $\alpha$), then when standard fundamental sequences are known up to $\beta$ and used in defining $f_\alpha$ at limit ordinals, the sequence $\{f_\alpha\}$ for $\alpha < \beta$ is called a standard spine.\(^1\)

It is known that there are standard iteration spines up to large ordinals which extend the AG spine. In Robbin [25] a $\omega^\omega$-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 $\epsilon_0$-standard iteration spine is used to extend the $\mathcal{R}^n$ hierarchy using resource bounded RASP's.

Let $\text{GR}_\beta$ for $\beta < \epsilon_0$ be the bounded GR language with clocks coming from $\{f^{(p)}_{\alpha+1}\} \alpha < \beta$ as defined in [8]. No one yet has discovered a natural class of SPL's for $\text{GR}_\alpha$ when $\alpha > \omega$. But for $\alpha = \omega^n$ the Loopstack languages of Contable [9] and the Loop languages of Ritchie [24] are equivalent to $\text{GR}_\alpha$ which presents $\mathcal{R}^n$.

---

\(^1\) 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 \( \phi_i \) 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 \( \text{Var}(\pi) = \{ v_1, \ldots, v_p \} \) be the variables occurring in \( \pi \) and let \( \text{Lab}(\pi) = \{ \ell_1, \ldots, \ell_q \} \) be the labels in \( \pi \). The simple form results from requiring that if there are \( p \) members of \( \text{Var}(\pi) \) and \( q \) members of \( \text{Lab}(\pi) \), 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 \( \phi_s \) 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) $\phi^1_i(x) = \phi^1_{s(j)}(x)$ for all $x$
(ii) $f(|i|) < |s(j)|$

(iii) if $\phi^1_{s(j)}(x)$ is defined, then $t\phi^1_i(x) \leq h(x,t\phi^1_{s(j)}(x))$ e.f.s.

Discussion of proof: Let $\phi^2_u$ be the universal program for the one argument sublist $\{\phi^1_i\}$. Then a uniform way to compute $\phi^2_{s(i)}(x)$ 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 $\phi^1_{\beta(i)}(x) = \phi^2_u(s^1(i),x)$ ($\beta$ 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 $|\phi^1_{\beta(i)}| = |u| + |s^1| + i \overset{\text{def}}{=} c^1 + i$, and since $f(2 \cdot c^1) < |s^1(c^1)|$ it follows that

(i) $\phi^1_{\beta(i)}(x) = \phi^1_{s^1(c^1)}(x)$ 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 $\phi_{B(i)}$ has nearly the same run time as $\phi_{s^{-1}(i)}$ e.f.s., because the program $\phi_{B(i)}$ simply computes $s^{-1}(i)$ first, at some fixed cost, say $d$, and then uses $\phi_u$ 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 $S$'s a size class for $S$ iff

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

(ii) $S'$ 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) $S'$ contains a universal function, $\phi_s$. 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 $S'$ of any size class $S'$ of $S$ there is an $h(\ ) \in S'$ such that all $f(\ )$ in $S'$ 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
\[ s'(i) = s(\mu z | s(z) \geq f(2 \cdot i)) . \]
Notice that \( s(z) \leq |f(2 \cdot i)|^{-1} \) since \( |\cdot|^{-1} \) is monotone over \( \{\phi_i\} \).
Also \( s \leq s^{-1}(|f(2 \cdot i)|^{-1}) \)
because \( s^{-1}(|f(2 \cdot i)|^{-1}) = \text{least } y \text{ such that } s(y) \geq |f(2 \cdot i)|^{-1} \).
Thus \( |s(y)| \geq |f(2 \cdot i)|^{-1} \geq f(2 \cdot i) \).

So the function \( t(i) = \mu z \leq s^{-1}(|f(2 \cdot i)|^{-1})(|s(y)| \geq f(2 \cdot i)) \) 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 \( \mathcal{R} \{ f^p \} \) (abbreviated \( \mathcal{R}_\alpha \)) are closed under composition
and bounded \( \mu \) (see [8] or [25]) and \( \mathcal{R}_{\alpha+1} \) contains the universal function
for \( \mathcal{R}_\alpha \) and \( |\cdot|^{-1} \) and \( h(\_ ) \) are primitive recursive for the indexing of
\( \text{GR} \), we can state a simple corollary for the classes \( \mathcal{R}_\alpha \).

Given \( \{ \phi_s(i) \} \subseteq \text{GR}_\alpha \) 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 \( \{ \phi_s(i) \} \subseteq \text{GR}_\alpha \) and for all \( f(\_ ) \in \mathcal{R}_{\hat{\alpha}} \) there are \( i, j \) such that

(i) \( \phi_i(\_ ) = \phi_s(j)(\_ ) \) and \( \phi_i(\_ ) \in \text{GR}_{\hat{\alpha}} \)
(ii) \( f(|\phi_i|) \leq |\phi_s(j)| \)
(iii) \( t_{\phi_i}(x) \leq h(x, t_{\phi_s(j)}(x)) \) e.f.s.

(3.2) presentation of relatively shortest programs

Let \( S = \{ g_0, g_1, \ldots \} \) be a presentation of \( \mathcal{S} = \mathcal{S}_\alpha \), then \( \hat{S} = \{ \hat{g}_0, \hat{g}_1, \ldots \} \) is
the presentation of shortest \( S \)-programs 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 \( S \subseteq \mathcal{R} \), 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 nth 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 $\hat{S}$.

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

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

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

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

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) $\phi_i(\ ) = \hat{S}_j(\ )$
(ii) $2 |\phi_i| + 1000 < |p_j|$  

(iii) $t\phi_i(x) \leq (t\hat{\phi}_j(x))^2$ 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$^w$ exhibit the same contrast (in this case $\hat{w}=w+1$). This suggests that for certain practical limitations on $f()$, GR$^w$ might already allow nearly the shortest possible expression of SR programs.

Indeed, we claim that the shortest reasonably efficient program for a feasible $R^1$ function is about the same size as the shortest SR program for it. A similar claim can be made for complexity classes $R_t$.

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 $f_\alpha()$ 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 $X^{+1}+1$ (after $X^{+0}$). This program has length $n+1$. But if $n=2^x$ a shorter program would generate $x$ and calculate $2^x$. If $n=2^2^x$, then it is even more worth while to compute $x$, then $2^x$, then $2^2^x$. If the formalism for $\hat{w}$ does not allow $2^x$ nor any function growing that fast, then some of its program, namely those for $n=2^2^x$, will be shorter than programs in a formalism allowing $2^x$.

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) **size-efficiency exchange**

It is easy to show that the shortest GR program for a function $f()$ is sometimes drastically inefficient. Say that $\phi_i$ is (absolutely) **shortest** for $\phi_i()$ iff for all $\phi_j()=\phi_i()$, $|\phi_j| \leq |\phi_i|$.
Theorem 6. For all \(a( )\), \(r( )\) \(\in\mathcal{R}\) there is a \(f( )\) \(\in\mathcal{R}\) such that if \(\phi_i\) 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 \(\phi_i\) shrinks \(\phi_j\) by \(s\) within \(h( )\) efficiency 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 \(\text{GR}_n\) because they are convenient ways to present the fundamental ideas, which carry over to other bounded formalisms. Recall \(\text{GR}_n\) is a presentation of \(\mathcal{R}\{f_n(p)\}\).

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

We prove the size-efficiency exchange for the case of \(\text{GR}_2\). Notice that \(|f_2( )|=|\text{DO V; DO V; V V+1; END; END}|=5\).

Theorem 7. Let \(\text{GR}_2=\{\hat{\phi}_j\}\), where \(\hat{\phi}_j=<f(p_j), \phi_j>\). Then if \(\phi_i \in \text{GR}\) shrinks \(\hat{\phi}_j\) by \(s\) and has finite \(\text{GR}_2\) complexity, then

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

Corollary 1: If \(\phi_i\) shrinks \(\hat{\phi}_j\) by \(s\) within \(f_2\) efficiency, then

\[p_j \geq s/5 - 1\]

\[\text{For technical reasons it is better to use } f_n(p)(x+1) \text{ than } f_n(p)(x).\]
Proof: Notice that if \( t\phi_j(x) \leq f(p_j)^{(p)}(x) \) for all \( x \), then \( c(\phi_j) = p_j \) for otherwise \( \hat{\phi}_j \) would not be shortest. For such functions we can replace \( p_j \) by \( c(\phi_j) \) in the theorem.

Let \( \hat{\phi}_j \) be \(<f_2(p_j), \phi_j>\). If \( \phi_i \) shrinks \( \hat{\phi}_j \) within \( f_2^{(p)} \) efficiency, for some \( p \), then \(<f_2(p_i), \phi_i>\) is in the GR_2 formalism where \( c(\phi_i) = p_i \). So

\[ |<f_2(p_i), \phi_i>| \geq |<f_2(p_j), \phi_j>| = |m_j| \] since \( \hat{\phi}_j \) is shortest. This implies

\[ |f_2| \cdot |p_i| + |\phi_i| \geq |f_2| \cdot |p_j| + |\phi_j| \]. Therefore, \( \phi_i \)'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 \( \phi_i \) saves \( s \) statements,

\[ |f_2| \cdot |p_i| + |\phi_i| > |f_2| \cdot |p_j| + |\phi_j| \].

So since \(|f_2| = 5\),

\[ 5 \cdot p_i > s \]

Thus \( 5 \cdot (p_i - p_j) > s - 5 \cdot p_j \), and \( d = p_i - p_j \) represents the difference in finite efficiency of \( \phi_i \) and \( \phi_j \). This establishes the theorem.

The efficiency of \( \phi_i \) can improve that of \( \phi_j \) iff \( s \leq 5 \cdot p_j \) and by the theorem hypothesis, \( p_i \leq p_j + 1 \) so in the worst case \( 5 + 5 \cdot p_j > s \) and \( p_j > (s/5) - 1 \), 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 \( x > 1 \) 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 \( \phi_i \) can be put in the normal form

\[
\text{DO WHILE } H \neq 0 \\
\phi_i^* \\
\text{END}
\]
where \( \phi^* \) is an SR program without DO-loops (called an SR or Loop-free SR program). Bounded formalisms can be built around this normal form by computing a clock \( \phi_f \) with output \( S \), either sequentially or in parallel with DO \( S \); \( \phi^* \); END. Since \( \phi_i \) and \( \phi^* \) 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 \( f^{(p)}_2 \), \( c \) in \( <c,t> \)) 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 \( \phi_i \in \text{GR} \) shrinks a \( <c,t> \) bounded program \( \phi_j \) 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 \( \phi_i \) and \( \phi_j \) is \( \overline{c} - c \).

Theorem 8: Let \( \hat{\text{GR}}_{T(S)} = \{\hat{\phi}_i\} \), suppose \( \phi_i \) shrinks \( \hat{\phi}_j \) by \( s \) without infinite loss of efficiency. Then if \( |\hat{\phi}_j| = |\phi_j| + |t| + |c| \), the finite loss of efficiency \( d \) satisfies \( d \geq s - (|c| + |t|) \).

Proof: Similar to Theorem 7.

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

\[
(3.4) \quad \text{the size of constants}
\]

For reasons indicated in (3.3) we are interested in the set of GR programs computing constants (no inputs), \( \{\phi^0_i\} \). We summarize Theorem 7 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_\( \alpha \)', could contain the absolutely shortest programs for all functions in GR_\( \beta \) for \( \beta < \alpha \). It also says that the absolutely shortest programs are arbitrarily complex.

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

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 \( \Sigma_S(n) = \max\{\phi_i^0( ) | \phi_i^0 < n \wedge \phi_i \in S \} \). This is the Rado function in \( S \) for constants. Call \( n \in \mathbb{N} \) \( S \)-maximal iff \( n \in M_S = \{x \in \mathbb{N} | x \in \mathbb{N} \} \).

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 \( \Sigma_S(2n) \), \( \Sigma_S(2n+1) \).

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 $|\eta_i| > 2.1$, 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 $\eta_\beta', \ldots, \eta_\beta^p$ and $\eta_\alpha', \ldots, \eta_\alpha^q$ be the absolutely shortest maximal constants for $R_\alpha$ and $R_\beta$ respectively, $\alpha < \beta$, listed in order of value. The ratio $\eta_\alpha^p / \eta_\beta^q < 1$ represents the relative expressive power of $\beta$ over $\alpha$. Notice that if $\alpha$ remains fixed and $\beta$ increased, the ratio $\eta_\alpha^p / \eta_\beta^q$ approaches 0 but "reaches 0" only when $R_\beta = R$ for some nonconstructive $\beta$.

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 $f(p)^2$ let the complexity of $\phi_i^0, c\phi_i^0$, be the least $j$ such that $t\phi_i^0(\ ) \leq f(j)^2$. Let $S = \{\phi_i^0 | t\phi_i^0(\ ) < f(p)^2 \text{ p} \in \mathbb{N}\}$. Then $S$ is a recursive presentation. A program $\hat{\phi}_j$ in $\hat{S}$ can be shrunk by $s$ statements using $\phi_1$ iff the difference in complexity, $d = c\phi_1 - c\hat{\phi}_j$, satisfies $d \geq s/5 - c\hat{\phi}_j$. 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


