COMPUTATIONAL COMPLEXITY AND NONDETERMINISM IN FLOWCHART PROGRAMS*

Theodore Paul Baker

TR 73-185

September 1973

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University for the Degree of

Doctor of Philosophy

Department of Computer Science
Cornell University
Ithaca, N.Y. 14850

*This research was partially supported by a Ford Foundation 6-Year Ph.D. Program fellowship and by National Science Foundation Grant GJ-3317X.

Acknowledgements

I would like to express my deepest gratitude to Professor J. Hartmanis, chairman of my special committee, for the shared insights, guidance, and encouragement that made this thesis possible. I would also like to thank Professors R. Constable, J. Williams, and A. Nerode for serving on my special committee.

I want to thank J. Simon for his thoughtful comments and his help in proofreading, particularly for suggesting Corollary 4.3 and tying up several "loose ends" mentioned in Chapter 7. I also am indebted to K. Mehlhorn for Theorem 4.5 and M. Siegel for help in proofreading.

This work was partially supported by a Ford Foundation 6-Year Ph.D. Program fellowship and by National Science Foundation Grant GJ-3317X.

CONTENTS

It is clear that a viable theory of computation must deal realistically with the quantitative aspects of computing and must develop a general theory which studies the properties of possible measures of the difficulty of computing functions. Such a theory must go beyond the classification of functions as computable and noncomputable, or elementary and primitive recursive, etc. It must concern itself with computational complexity measures which are defined for all possible computations and which assign a complexity to each computation which terminates. Furthermore, this theory must eventually reflect some aspects of real computing to justify its existence by contributing to the general development of computer science.

J. Hartmanis and J.E. Hopcroft
"An Overview of the Theory of Computational Complexity"

1 Introduction

§1.1 Motivation

This thesis is an effort to contribute to the Theory of Computation by investigation of a special class of complexity measures, the step-counting functions associated with flowchart programs. It would perhaps be fitting to begin by describing the worthy contributions of all the others who have brought the theory to its present state, yet the field has grown to a point where such a review would far overshadow any new material we might present. In the discussion which follows we therefore attempt merely to outline the immediate context in which this work was done. Apologies are offered to those whose work may be relevant but whose work is not mentioned because it did not directly influence the thesis.

The work which did directly influence the thesis falls roughly into three areas: Program Schemata; the LBA Problem, NP=?P, and nondeterminism in general; Computational Complexity. Although bordering on the first two areas, our interest lies primarily in Computational Complexity.

An undesirable gap has been growing in Complexity Theory between the concrete an the abstract. Early work, beginning with Hartmanis and Stearns [14], dealt with complexity measures defined in terms of very specific computing devices, mostly Turing machines. With the introduction in 1967, by M. Blum [1] of an axiomatic characterization of "step-counting functions", the theory took a turn toward the abstract. Blum's axioms have been shown to have many deep consequences and have served as the roots of a rapidly-growing fruitful theory. Nevertheless, they are not wholly satisfactory in that they admit too many measures, some of which are intuitively unacceptable (see §1.5). Since the answers to a large number of questions one may ask about a complexity measure are independent of the Blum axioms, work has continued in the specific models of computation where such questions are answerable. The problem is that we frequently want to talk about measures that lie between these two extremes: on the one side the specific measures with their peculiar local properties, and on the other side the Blum measures, so general as to render many questions meaningless. The theory needs a notion of "natural" complexity measure to fill this void, as Hartmanis points out in [10].

We can find a case in point, and one motivation for the present work, in nondeterminism. Much attention has been drawn of late to questions which amount, in essence, to whether there is any computational power to be gained by allowing nondeterminism in certain specific models of resource-bounded computation [6,13,17]. Such questions are beyond the scope of general Blum complexity theory, which does not recognize a distinction between deterministic and nondeterministic measures. Nonetheless, this distinction seems to be an important abstract concept, transcending individual models.

Our first approach to more abstract nondeterminism (which resulted in theorems 4.1 and 4.2) made use of Turing machines with recursive oracles. Although the measures defined by these machines are quite varied in their behaviour, owing to the oracles, they still have many of the idiosyncracies of the Turing machine step measure and therefore are not exactly what we are looking for.

What we seek is either a narrower abstract definition of complexity measure, or, failing that, a class of specific measures which reflect a natural conception of computation and in which we can discuss notions like nondeterminism, but which are sufficiently general to allow us to overlook the peculiarities of the individual measures. Hartmanis [10] gives several good reasons why we are unlikely to succeed by piling on more and more exclusionary additions to the Blum axioms, suggesting that a bottom-up constructive approach is more promising. With this in mind, we have decided to put off attempts to improve upon the Blum axioms until we know more. In the mean time we hope to learn more by examining a class of specific natural Blum measures.

Research on reducibility and equivalence in program schemata by Ianov, Paterson, and others [16,21,24] has shown a way to view the structure of algorithms separate from the obscuring details of computational systems. We apply this same technique to complexity measures. A flowchart is a program schema, defining a program for each interpretation given to its operations. Likewise, a flowchart language is a complexity measure schema, defining a complexity measure for each (universal) interpretation given to its operations. Whereas the research in program schemata has been primarily motivated by an interest in the problems of formal simplification and, more recently, in questions about the computational generality of various program structures (as exemplified by [3]), we are presently interested in how the complexity classes defined by the step-counting functions are affected by changes of interpretation, which we consider superficial relative to the unchanging schematic framework.

§1.2 Outline of Thesis

In this chapter we intend to lay a basis in motivations and definitions for the results which we describe later. The chapters following this one each cover a group of properties of the flowchart step measures. Together, they represent an attempt to place the class of flowchart measures as a bridge between the specific natural measures with which we are familiar and the full generality of the Blum measures.

Chapter 2 deals with the problem of recursively enumerating complexity classes. It exhibits two large groups of bounds for which the flowchart complexity classes are provably r.e.

Chapter 3 approaches the flowchart measures and their associated Gödel numberings as objects of translations. The flowchart G.N.s are shown to be especially good targets for translation from other Gödel numberings. We show that a "simple" translation between flowchart interpretations exists and that under some circumatances it produces efficient programs.

Chapter 4 is concerned with how nondeterminism relates to determinism in flowchart programs. Two examples of flowchart interpretations are given, for one of which NP≠P, and for the other of which NP=P. These are presented so as to apply not only to the polynomial time bounds, but to any class of time bounds which share certain properties with them.

It is thus shown that the answer to Cook's NP=?P [6] and the answers to similar questions about nondeterminism and other classes of time bounds must be strongly dependent on the precise model of computation. Further results are given which indicate that if there is a "rule" it is for nondeterminism to allow greater computational speed than determinism.

Chapter 5 gives two examples of the manner in which well-known natural measures (Turing machine tape and multitape T.M. time) can be "simulated" by flowchart measures so as to make classes like the polynomially-bounded computable functions the same.

Chapter 6 offers two additional properties which we have observed in the flowchart and other natural measures that perhaps whould be considered requisite for "naturalness". Examples are then given of "natural" properties proposed by others which do not hold for all flowchart measures. Finally, some ancillary properties are presented which demonstrate how flexibly the flowchart interpretation may be used to tailor the measure, so as to uniformly speed up the flowchart "machine", to selectively reorder the complexities of certain functions, or to limit the computational advantage gained by having more than one register.

Chapter 7 concludes the thesis. The first section summarizes the main results. The second section catalogues some natural questions which we would like to answer but have not answered as yet. The last section proposes a tentative framework for an abstract axiomatization of step measures which would explicitly relate the "step" as a unit of cost to the step as part of a computation.

§1.3 Basic Notation and Definitions

We are forced to assume that the reader has prior knowledge of the most fundamental terms and concepts of Recursive Function Theory [25], Complexity Theory [12], and Graph Theory [8]. However, some elementary definitions are included below, because our versions are not standard or because we wish to fix notation.

A. General

N denotes the set of non-negative integers, $\{0,1,2,3, \ldots \}$. A <u>number</u> means an element of N. N is the same as $('1'\ '0'^*)^*$, a regular subset of the binary (bit-) strings, $\{0,1\}^*$. Bit-strings are differentiated from decimal numerals and from variables by enclosure with "'"s. Thus, if n = 5, $'1^n01'n'0'$ represents the bit-string '11111011010', which is 2010 in decimal notation. Zero is written ε (the null string) when treated as a bit-string. Multiplication $(x \cdot y)$ is distinguished from concatenation of bit-strings (xy).

Unless stated otherwise, a (partial) <u>function</u> means a (partial) computable function from (a subset of) N to N. R denotes the set of all total recursive functions. $f|S$ denotes the restriction of the (partial) function f to the domain S. Two (partial) functions f and g are <u>equivalent</u> $(f \equiv g)$ if for all x (such that f(x) or g(x) is defined) $f(x) = g(x)$. If for all x (such that f(x) or $g(x)$ is defined) $f(x) \leq g(x)$ then we write $f \leq g$. If a relation holds everywhere but on a finite set we say that it holds <u>almost everywhere</u> (a.e.); if it holds on an infinite set we say it holds <u>infinitely often</u> (i.o.). For example, $f \geq g$ i.o. means that there are infinitely many x such that $f(x) \geq g(x)$. If $f > k$ a.e. for every k in N then f is <u>ultimately increasing</u>. A function g is said to be $O(f)$ for some function f if there is a constant c such that $g \leq c \cdot f$.

A <u>predicate</u> is a function from N to $\{0,1\}$. If P is a predicate then $\overline{P}$ denotes the <u>complement</u> of P, the function which is 0 when P is 1 and v.v.

To define functions we will frequently use an informal LISP-like [22] notation, based on A. Church's [4] lambda calculus (e.g., $\lambda x,y((x>y) \longrightarrow (x-y);\ (y-x))$ for the integer norm). It follows standard conventions closely enough that the reader should have no difficulty in understanding.

G generally denotes an acceptable Gödel numbering of the partial recursive functions (e.g., a lexicographic enumeration of the programs for a universal binary single-tape Turing machine). $\underline{G}$ denotes a Blum complexity measure on G (e.g., the T.M. step count). G_i denotes the i^{th} partial recursive function in this G.N. and $\underline{G}_i$ denotes its complexity or step-counting function. For any recursive function t, $R_t^{\underline{G}}$ (abbreviated R_t) is the class of total recursive functions of complexity t or less almost everywhere, i.e.,

$$R_t^{\underline{G}} = \{f:N\to N \mid f\equiv G_i,\ \underline{G}_i\leq t \text{ a.e.}, f\in R\} .$$

Similarly $L_t^{\underline{G}}$ denotes the class of sets

$$\{S:\ G_i \text{ accepts } S,\ \underline{G}_i\leq t \text{ a.e.}, f\in R\} .$$

A function τ is a <u>translation</u> from Gödel numbering G to G.N. H if for every $i\in N$

$$G_i \equiv H_{\tau(i)} .$$

B. Flowcharts

We define here a specific type of program schema, called a flowchart. Similar definitions of schemata are found in [3,16,21,24].

A function name is a sequence of one or more "F"s followed by one or more "'"s. A predicate name is a sequence of one or more "P"s followed by one or more "'"s. $\overline{F}_i^{(j)}$ denotes "F...F'...'", with i "F"s and j "'"s, and $\overline{P}_i^{(j)}$ should be interpreted similarly.

A flowchart language L is a pair (I_P, I_F) of sets of numbers defining a set*

$$\overline{P} = \{\overline{P}_{m_i}^{(n_i)}: \ i \epsilon I_P\}$$

of n_i-ary predicate names and a set

$$\overline{F} = \{\overline{F}_{m_i}^{(n_i)}: \ i \epsilon I_F\}$$

of n_i-ary function names. Unless the contrary is stated, $\overline{P}$ and $\overline{F}$ are assumed to be recursively enumerable and of unary names. With few exceptions, the following definitions and proofs apply to the general n-ary case, but the notation required to carry this through will eventually be dropped, since it becomes nearly unreadable.

An L-predicate-term is a predicate name $\overline{P}_{m_i}^{(n_i)}$ followed by n_i positive integers, called register indices. An L-function-term is a positive integer, also called a register index, followed by a function name $\overline{F}_{m_i}^{(n_i)}$ and n_i register indices.

* [Note: $\{(m_i, n_i)\}_{i=1}^{\infty}$ is an enumeration of N×N.]

A nondeterministic L-flowchart is a labelled directed graph with nodes labelled with L-predicate-terms and L-function-terms and edges labelled with 0 and 1, the nodes being indexed by an initial segment of the positive integers. The node indexed with 1 is called the starting node. A deterministic L-flowchart in addition satisfies the requirement that here be at most one edge from each function-term node and at most one edge labelled 0 and at most one edge labelled 1 from each predicate-term node. Ordinarily, a flowchart is taken to mean a non-deterministic L-flowchart for some flowchart language (FCL) L. All flowcharts are assumed to be finite, although it does make sense to speak of infinite flowcharts.

In a flowchart, a terminal node is a node which has no edges from it, or which is a predicate-term node and has no 0-edges or no 1-edges from it.

Example of a flowchart:

```
┌──→1: 2 F 1
│        │0
│        ↓
│  2:  P 2 ──1──→ 5: 1 G 1
│        │0              │
│        ↓               │
│  3: 2 F 2 ←────0───────┘
│        │0
│        ↓
└──0──4:  Q 2
```

Here we have substituted shorter auxiliary names for the functions and predicates in order to enhance readability. This practice will be continued in subsequent proofs and examples. The canonical "FP'" notation is necessary only to insure the existence of an enumeration of all flowcharts.

If A_i is a flowchart, $A_{i<j>}$ denotes the node of A_i with index j, if such exists. Otherwise, $A_{i<j>}$ is defined to be the null set, $\emptyset$. A path in A_i is a sequence $A_{i<j_1>}$, $A_{i<j_2>}$, ... of nodes such that there is an edge from each node to the next one, repetitions being permitted. A cycle is a path which begins and ends with the same node. A flowchart which is a tree is called a tree-flowchart.

When we talk about the complexity of computations on flowcharts, we will need a simple canonical way of encoding the flowcharts as integers. For this reason only, the following detailed description is given.

flowchart ::=	node*
node ::=	\$ term 0-edges 1-edges
term ::=	index f-name index list
term ::=	p-name index list
index ::=	$(1\ 0^*)^*$
0 ::=	'00'
1 ::=	'01'
p-name ::=	# \$ name
# ::=	'10'
\$ ::=	'11'
name ::=	$'1'^+\ '0'^+$
index list ::=	\$ (# index)*
f-name ::=	\$ name
0-edges ::=	\$ (# edge)*
edge ::=	0 index
edge ::=	index
1-edges ::=	\$(# edge)*

Condensed slightly, the flowchart descriptions are:

(\$(index+ #)\$ name \$(# index)* \$(# edge)* \$(# edge)*)* .

Note: In the description above the notation of regular expressions is used. X* denotes all strings of the form X concatenated with each other in all possible ways, that is, all sequences of strings of the form X, including the null string. When we wish to exclude the null string we write X^+. X+Y denotes all strings of the form X or Y.

As this clearly demonstrates, the valid flowchart descriptions are a regular subset of $('1'('0'^*))^*$, which we have identified with N. The way in which they may be interpreted as describing flowcharts is suggested in part by the names of the nonterminals in the preceding grammar, and may be further explained as follows:

A node describes a node. The index of the node it describes is the order of the node in the F.C. description. An index is an encoded integer, possibly the index of a register.

An edge is an encoded integer, possibly preceded by a '00'. It represents the index of the node it is in less the index of the node to which it goes. A negative sign is represented by the initial '00' when it appears. If the edge is to an index beyond the range of the flowchart description it is interpreted as null (going nowhere). An f-name is the name of a function and a p-name is the name of a predicate. For n-ary operations there is a final string of n '0's in the name. The n argument register indices are taken to be the first n indices encoded following the name. If there are too many, the excess are ignored. If there are too few, the unspecified register indices are assumed to be 1. If the result register index of a function is missing, then it is also assumed to be 1.

[Local note on underlining: Here an index is not an index, but the nonterminal in the preceding grammar, or a string generated from that nonterminal. Similarly for other words.]

All invalid flowchart descriptions are defined to represent the null flowchart (0 or ε), so that every element of N can be thought of as denoting a unique flowchart, A_n. If we wish to enumerate some subset of the flowcharts, say the deterministic flowcharts, D, we define $D_n = A_n$ if $A_n \in D$ and $D_n = A_0$ (the null flowchart) if $A_n \notin D$.

C. Programs

An <u>interpretation</u> J is a pair (P,F) of a set

$$P = \{P_{m_i}^{(n_i)} : i \in I_P^J\}$$

of predicates and a set

$$F = \{F_{m_i}^{(n_i)} : i \in I_F^J\}$$

of functions. Together these predicates and functions are called the <u>operations</u> of J. Unless stated otherwise, P and F are always r.e. sets and the predicates and functions are always recursive. L_J denotes the language (I_P^J, I_F^J).

<u>J-flowchart</u> is an abbreviation for L_J-flowchart. A and D denote the canonical lexicographic enumerations of the (N,N) flowcharts (i.e., all the flowcharts) as described above. These may be also interpreted as enumerations of the J-flowcharts by letting all flowcharts using names of operations not in J represent the null flowchart.

A <u>flowchart program</u> A_i^J is a flowchart A_i together with an interpretation J such that A_i is an L_J-flowchart. A_i^J is called a <u>J-program</u>.

D. Computation

Each program A_i^J may be interpreted for each n as defining a unique partial n-ary function $A_i^J(x_1,\ldots x_n)$. We define this by first describing a more general action of A_i^J on sets of sequences of numbers, which might be called "instantaneous descriptions" (i.d.s). Let m be the maximum register index appearing in A_i. The finite sequences of one or more elements of N are denoted by N^+, and $^{N_2^+}$ denotes the finite sets of these. For each set S in $^{N_2^+}$ define $A_i^J[S]$ =

$\{ (s',x_1', \ldots x_m')$: $(s,x_1, \ldots x_n)$ is in S, $n \geq m$, $A_{i<s>} \neq \emptyset$, and

(1) If $A_{i<s>}$ is labelled $\overline{P}_j^{(k)} z_1 \ldots z_k$ then

(a) there is an edge labelled $P_j^J(x_{z_1},\ldots x_{z_k})$ from $A_{i<s>}$ to $A_{i<s'>}$ and $x_i' = x_i$ for i = 1 to m, or

(b) there is no edge so labelled, s' = 0, and $x_i' = x_i$ for i = 1 to m.

(2) Otherwise, $A_{i<s>}$ is labelled $z_0 \overline{F}_j^{(k)} z_1 \ldots z_k$, and

(a) there is an edge from $A_{i<s>}$ to $A_{i<s'>}$ and $x_{z_0}' = F_j^J(x_{z_1},\ldots x_{z_k})$, $x_i' = x_i$ for i = 1 to m, $i \neq z_0$, or

(b) there is no such edge, s' = 0, and x $x_{z_0}' = F_j^J(x_{z_1},\ldots x_{z_k})$, $x_i' = x_i$ for i = 1 to m, $i \neq z_0$.$\}$

$$A_i^J[S]_0 = S.$$

$$A_i^J[S]_n = A_i^J[A_i^J[S]_{n-1}] \quad \text{for } n > 0.$$

$[A_i^J(y_1,\dots y_k)]_n = A_i^J[\{(1,y_1,\dots y_m): i>k \implies y_i=0\}]_n$.
This is called the n^{th} stage of the computation of $A_i^J(y_1,\dots y_k)$. The halting point is the first $n\geq 0$ for which the first, or node-, component of one element of the n^{th}stage of the computation is 0. If n is the halting point of $A_i^J(y_1,\dots y_k)$ and

$$\{x_1: (0,x_1,\dots x_m) \text{ is in } [A_i^J(y_1,\dots y_k)]_n\}$$

has a single element we say that this element is $A_i^J(y_1,\dots y_k)$ and that the computation converges in n steps. In this case n is called the running-time, complexity, or step count of $A_i^J(y_1,\dots y_k)$ and written as $\underline{A}_i^J(y_1,\dots y_k)$. Otherwise, $A_i^J(y_1,\dots y_k)$ and $\underline{A}_i^J(y_1,\dots y_k)$ are undefined. A_i^J computes the (partial) function

$$\lambda y_1,\dots y_k(A_i^J(y_1,\dots y_k)).$$

A_i^J strongly computes a partial function $f^{(n)}$ iff

$$A_i^J(x_1,\dots x_m) = f(x_1,\dots x_n)$$

for every $x_1,\dots x_m$ and $x_1,\dots x_n$, where m is the maximum register index of A_i, as before. (This means that the function computed by A_i^J is independent of the initial value of any but the input registers— that is, it does not matter whether the other registers are zero or not.)

It follows directly from the definitions that A_i^J defines a (partial) recursive function if A_i is finite and J recursive. Unless it is stated to the contrary, we will speak only of this type of program, with finite flowchart and recursive interpretation.

For any flowchart interpretation J, under the standard conventions we have stated, the J-programs are r.e. This follows directly from the J-flowcharts' being r.e. We denote the canonical enumerations of the nondeterministic and deterministic J-programs by $A_1^J, A_2^J, A_3^J, \ldots$ and $D_1^J, D_2^J, D_3^J, \ldots$, respectively.

A program <u>accepts</u> a set S if it computes a total function f such that

$$S = \{x: f(x) \neq 0\}.$$

We extend this definition to include partial functions, saying that f (partial) accepts S iff

$$S = \{x: f(x) \text{ defined } \& \ f(x) \neq 0\}.$$

Zero is thus interpreted as a "reject value", wheras all other values are interpreted as "accept values".

E. Universality

A flowchart interpretation J is universal iff, for any Gödel numbering of the partial recursive functions, G, there is a recursive map, h, such that for each i

$$G_i \equiv D^J_{h(i)} .$$

Thus, equivalently, J is universal iff $D^J_1, D^J_2, \ldots$ is a G.N., and J is universal iff for every partial recursive G_i there is a D^J_j such that $G_i \equiv D^J_j$. Say that J is nondeterministically universal (N-universal) iff $D^J_1, D^J_2, \ldots$ is a G.N.; we conjecture that N-universality does not imply universality.*

A flowchart interpretation J is strongly universal iff for every p.r. G_i there is a j such that D^J_j strongly computes G_i. We can easily see that this kind of universality is stronger than the first kind by looking at a universal interpretation J, modified so that each of its functions and predicates requires an additional argument which must be zero unless the function is to return the value one. Unless we are certain to always have a zero-valued register, this interpretation cannot compute anything consistently by the one-function, so it is not strongly universal; it is still universal, however.

---

*[As J. Simon has pointed out, there is a simple proof of this when we restrict ourselves to single-register programs.]

If D^J is a Gödel numbering then $\underline{D}^J$ is a Blum complexity measure on it, and similarly for the nondeterministic programs. When $\underline{D}^J$ is a Blum measure, it is a submeasure of $\underline{A}^J$. If $I \subseteq J$ then $\underline{D}^I$ is a submeasure of $\underline{D}^J$, and similarly with respect to $\underline{A}^I$ and $\underline{A}^J$.

If t is a recursive function, we denote by $R_{\overline{t}}^{A^J}$ the complexity class of functions

$$\{f:N\to N \mid f \equiv A_i^J,\ \underline{A}_i^J \leq t \text{ a.e., } f \in R\} .$$

By $R_{\overline{t}}^{D^J}$ we mean the same thing, but for deterministic programs only. We say that a set S of functions is r.e. iff there is an r.e. set of numbers W such that

$$S = \{f:N\to N \mid f \equiv G_i,\ i \in W\} ,$$

where G is a Gödel numbering.

$L_{\overline{t}}^{D^J}$ and $L_{\overline{t}}^{A^J}$ are the corresponding complexity classes of languages.

This notation is all extended to <u>sets</u> T of complexity bounds in the obvious way, e.g.,

$$R_{\overline{T}}^{A^J} = \bigcup_{t\in T} R_{\overline{t}}^{A^J} .$$

The superscript J is sometimes omitted when the meaning is clear from the context.

A <u>t-bounded</u> J-program is a program A_i^J such that

$$\underline{A}_i^J \leq t \quad \text{a.e.}$$

F. Abbreviations and Symbols

a.e.	almost everywhere
G.N.	Gödel numbering
F.C.	flowchart
iff	if and only if
i.o.	infinitely often
lng(n)	length of n as bit-string
min(S)	least element of the set S
p.r.	partial recursive
r.e.	recursively enumerable
recursive	total recursive
s.t.	such that
T.M.	Turing machine
v.v.	vice versa
w.r.t.	with respect to
w.l.o.g.	without loss of generality
$\forall$	for all
$\exists$	there exists
$\Longleftrightarrow$	if and only if
$\Longrightarrow$	implies
$\subset$	proper set inclusion
$\subseteq$	set inclusion
$\in$	is an element of
$\cup$	set union
$\cap$	set intersection
$\times$	Cartesian product
$\lvert S\rvert$	cardinality of the set S
$f\mid S$	function f restricted to set S
$\cdot$	multiplication
$\lfloor x\rfloor$	greatest integer $\leq$ x
$\lceil x\rceil$	least integer $\geq$ x
ε	the null string
$A_i^J[S]$	the set of instantaneous descriptions reachable in one step from the set of instantaneous descriptions S, by the program A_i^J

$A_{i<j>}$	the j^{th} node of A_i
i.d.	instantaneous description
i.d. function	the identity function, $\lambda x.x$
'0'	the bit (character) zero, as opposed to the integer zero, 0
'1'	the bit one
N^+	all the non-null sequences of non-negative integers
N_2	all the finite sets of non-negative integers
Q.E.D.	end of proof (quod erat demonstrandum)

§1.4 How the Blum Axioms are Weak

We have claimed that the Blum axioms are too liberal. Since the vigorous theory which has grown from them attests to their strength [12], we would like to illustrate here how they are weak.

The definition that Blum proposed in 1967 [1] was essentially as follows: If $G_1, G_2, G_3, \ldots$ is an acceptable Gödel numbering of the partial recursive functions and $\underline{G}_1, \underline{G}_2, \underline{G}_3, \ldots$ is a sequence of partial recursive functions such that:

(1) $G_i(x)$ is defined iff $\underline{G}_i(x)$ is defined.

(2) The function

$$M(i,x,m) = \begin{cases} 1 \text{ if } \underline{G}_i(x) = m \\ 0 \text{ otherwise} \end{cases}$$

is total recursive,

then the $\underline{G}_i$ are <u>step-counting functions</u> (a complexity measure) for G. What this means is, first, that the amount of resource used by an algorithm is known precisely when the algorithm defines a convergent computation and, second, that it is always possible to determine whether an algorithm exceeds any particular bound on its resource. One can see how it would go contrary to the intuition to admit anything as a complexity measure that failed either of these simple requirements.

Nevertheless, because the axioms do not mention anything about how the complexity of an algorithm is related to the value computed, they allow us to construct some very strange "complexity measures". Starting with an intuitively acceptable measure $\underline{G}$, the step-count for single-tape T.M.s, one may distort the measure in countless ways.

Example 1.1 Let $G_{i_1}, G_{i_2}, G_{i_3}, \ldots$ be a subenumeration of the Gödel numbering G which includes only total functions. Such a set of functions might include all the constant functions or all the polynomially time-bounded functions. Let

$$\underline{G}_i' = \lambda x((i \in \{i_1, i_2, \ldots\}) \longrightarrow 0; \ \underline{G}_i(x)) .$$

In this measure, an infinite number of nontrivial functions are computable "for free".

Example 1.2 Let G_c be an arbitrary T.M. Let

$$\underline{G}_i' = \lambda x((i \neq c \text{ and } G_i\text{'s control structure is isomorphic to } G_c\text{'s as a finite automaton}) \longrightarrow \underline{G}_i(x)!; \ \underline{G}_i(x)) .$$

Here, definitionally-equivalent algorithms may have exponentially different "step counts".

Example 1.3 Let f be a function such that f is not the identity and

$$f \circ f \notin R_t$$

for some nontrivial time bound t. Suppose G_j computes f. Define

$$\underline{G}_i' = \lambda x((i=j) \longrightarrow 0; \ \underline{G}_i(x)) .$$

Thus f is "free" but we cannot apply it more than one time to get f∘f for less than cost t.

To see that such an f does indeed exists and is total, define one: Start with the assumption that f is going to be monotonically increasing. Choose a total recursive t. Suppose that f(x) is already defined for x = 1,2, ... N_i. Let r be a recursive function such that r = k i.o. for every k in N. If

$$\underline{G}_{r(i)}(N_i) \leq t(f(N_i))+t(N_i)$$

let $$f(f(N_i)) = \min\{y: y>f(N_i),\ y\neq G_{r(i)}(N_i)\},$$

let $$N_{i+1} = f(f(N_i))+1,$$

and let f(x) = x+1 for all x s.t. $x\neq f(N_i)$ and $N_i<x\leq N_{i+1}$.

Otherwise, let $f(N_i+1) = N_i+2$ and $N_{i+1} = N_i+2$.

These examples demonstrate that, although functions satisfying the Blum axioms must at least bear a <u>remote</u> resemblance to what we would accept as step-counting functions, they need not satisfy some of our basic ideas about consistency and "fairness" in measuring computational cost. One believes that changing the name of an algorithm should not change the amount of computational resource it uses. One also believes that one should be able to compose the results of various computations without great additional cost. We have seen how well the Blum axioms reflect such beliefs. Doubtless, it is difficult to say exactly what relations should hold between "cost" and value computed, but it must be considered a weakness of the Blum axioms that they do not in any way explicitly relate the two.

A further example, which we borrow from [12], shows that there are Blum measures where the functions of complexity zero are not r.e.. This is widely held to be undesirable.

Example 1.4 Let $\{G_{j_\ell}\}_{\ell=1}^{\infty}$ be an enumeration of the constant functions such that

$$G_{j_\ell} = \lambda x.\ell \ .$$

Define

$$\underline{G}_i' = \lambda x((i=j_\ell) \longrightarrow ((\underline{G}_\ell(\ell)<x) \longrightarrow x;\ 0);\ x+1) \ .$$

Thus

$$R_0 = \{G_{j_\ell} : G_\ell(\ell) \text{ is not defined}\} \ .$$

If R_0 were r.e. then, since $G_{j_\ell} \equiv \ell$,

the set

$$\{\ell : G_\ell(\ell) \text{ is not defined}\}$$

would be r.e. This set is well known not to be r.e., a contradiction.

§1.5 Nondeterminism—Comments on Definitions

From §1.3, it should be evident that our definition of acceptance, particularly nondeterministic acceptance, is not standard. This is not by caprice, but is part of a deliberate attempt to correct disparities in the traditional definitions that prevent certain notions we feel are related from fitting into a unified definitional framework.

We are concerned that definitions dealing with the notions of set recognition, function computation, determinism, nondeterminism, and computational complexity should satisfy the following principles:

(1) The class of nondeterministic algorithms (acceptors) includes the deterministic algorithms (acceptors). Consequently:

- (a) All terms meaningful in the deterministic case should have meaningful generalizations to the nondeterministic case (e.g., rejecting should be distinguishable from not halting).
- (b) Definitions should consistently treat the deterministic algorithms as a restricted case of the nondeterministic algorithms, i.e., the set accepted (or function computed) and the running time of a deterministic algorithm should be independent of whether it is viewed as a nondeterministic algorithm or as a deterministic algorithm.

(2) Recognizing a set amounts to computing a (partial) function (of at least two values, by (1a) above). Consequently:

(a) Complexity measures for acceptors should satisfy the Blum axioms, when the acceptor is viewed as computing a (partial) function to

{accept, reject}.*

(b) The complexity of accepting a set should not be greater than that of computing its characteristic function.

(c) Recursive sets should be recognizable by algorithms with total complexity functions.

In the interest of brevity, we omit the extensive philosophical justifications, based on "intuition" and specific examples of "real" computation, that could be given for these principles, trusting that the reader will join us in accepting them as self-evident.

Every one of these principles is violated by what, up to now, has passed for the "standard" definition of nondeterministic acceptance [15, 17]. According to custom, a nondeterministic algorithm "accepts" an input iff one of the computations it defines accepts (by whatever definition of acceptance is used in the deterministic case). The "running time" is the running time of the shortest accept-

---

*[In the nondeterministic case it might be desirable to weaken the first Blum axiom to say that $G_i(x)$ is defined only if $\underline{G}_i(x)$ is defined. This may be justified informally by saying that the computing device may "jam", due to an inconsistency in the output.]

ing computation.* This fails (1a) because there is no way to reject an input. It fails (1b) and (2a) because a deterministic algorithm may halt and reject, having finite running time, wheras the "running time" of the same algorithm, viewed as a nondeterministic algorithm, is undefined. (2b) and (2c) are failed for the same reason.

We incorporate principle (2) into our definitions very simply, saying that an algorithm <u>accepts</u> a set S if it computes the (0-1) characteristic function of S, or by extension, if it computes a (partial) function f such that

$$S = \{x: f(x) \text{ defined and } \neq 0\}.$$

Any Blum measure on the complexity of computing functions is thus also a measure on the complexity of accepting sets. It should be clear that this is close to the usual definition of acceptance for deterministic "natural" models of computation, such as Turing machines, in the sense that an accepting algorithm according to the usual definition may be transformed recursively into an accepting algorithm according to our definition without more than a constant increase in the complexity. It remains for us to define how a nondeterministic algorithm computes a function in such a way that nondeterministic acceptance also has a meaning close to the usual one.

---

*[This definition <u>is</u> adequate for "real time" or other fixed-bound computations, where the running time is always defined and rejecting means halting without accepting.]

Define a _computation_ to be an _initial value_ and a sequence (possibly infinite) of functions, called _operations_. Thus each computation defines a unique sequence of (intermediate) _values_, starting from the initial value and applying the successive operations of the computation. An _algorithm_ is a (partial) function which gives a set (possibly empty) of operations for each value, thus defining a computation or set of computations for each initial value. The algorithm is _deterministic_ if there is at most one permissible operation for each value. A (nondeterministic) algorithm defines a tree of computations for each initial value. If the algorithm is deterministic then the tree is a simple sequence. Let there be a (partial) function from the values to the _outputs_, which gives the _result_ of a computation from its final value. Let there be another (1-1) function from the _inputs_ to the values, giving an initial value for each input. The _function computed by_ a deterministic algorithm A on input x is the result of the computation defined by A on the initial value for x if the computation is finite.

So far, we have merely fixed a terminology for notions about which there is widespread agreement. When we come to extend this notion of computing a function to all (nondeterministic) algorithms, we run into uncertainty. Clearly, a single nondeterministic algorithm and initial value may define an infinite number of terminating computations. How do we define an output from the results of all these computations? We start by making a choice:

First, we may allow all terminating computations to contribute to the output. We reject this because it implies that the "output" and "running time" may be infinite (not inherently a bad idea, but out of keeping with our desire that the output be a (partial) recursive function of the input). The alternative is to (explicitly or implicitly) define a "cut-off time" by which a computation must have terminated if it is to contribute to the output. Intuitively, suppose we view the nondeterministic computing device as an infinitely-expandable network of independent computing devices, each one adding a copy of itself to the network whenever it is faced with a choice and passing on its result to a central output device when it terminates a computation. Having a cut-off means that the output device listens to all the results it gets up to a certain point, whereupon it decides to put out something. Naturally, all results of computations terminating after that point cannot contribute to the output, and we might just as well assume that all computations are cut off at the time of output.

No matter how good a cut-off criterion we choose, we must still occasionally face the problem of defining an output when several computations terminate simultaneously and produce inconsistent results. We call this problem ambiguity. The simplest action we can take is to give up, saying that the output is undefined unless all computations terminating by the cut-off time produce the same result.

Because this is simplest, we believe it is preferable to any ad hoc rule for resolving ambiguity,* assuming that an unambiguous output is required‡.

Having chosen to leave the output undefined when the computations terminating by cutoff time are ambiguous, and faced with a multiplicity of possible cut-off rules, two extremes beckon to us.† On the one hand, we could strengthen the policy of exclusivity we have followed in ignoring ambiguous results, letting the "cut-off" criterion be that there be no unterminated computations remaining to cut off, thereby minimizing the number of algorithm/input combinations producing output. On the other hand, we could try to maximize this number, cutting off all computations when first some computation terminates. We choose the latter policy because the definition of nondeterministic acceptance it gives us is closest to the traditional one, even though we believe the stronger policy surpasses it in elegance.

---

*[For example, choosing the least of the results defined by the terminating computations.]

‡[Of course, it would be most natural to allow for ambiguous output, interpreting the algorithms as defining set-valued rather than integer-valued functions. However, the prospect of saying the deterministic algorithms define (partial) functions from N to $\{\{n\}: n \in N\}$ is not appealing. The partial recursive functions are traditionally from N to N, and identifying n with {n} is inconsistent.]

†[In certain cases, especially when the algorithms are specified to have built-in "clocks", intermediate cut-off policies may be preferable.]

We have thus made three decisions which, together with the two principles we stated earlier, completely shape our definitions:

(1) There will be a cut-off time for each algorithm/input combination that produces output.

(2) If any algorithm/input combination produces ambiguous results no output will be defined.*

(3) An algorithm will be cut off for a particular initial value when first one of its computations terminates.

Although some compromise has been made in (2) and (3) between elegance and compatibility with previous work, we feel that any framework in which we can simultaneously discuss accepting sets and computing functions, deterministically and nondeterministically, along with the associated complexities must be better than the jumble of disparate notions we started with.

---

*[This means the running time must also be undefined, by Blum's first axiom, if we wish to satisfy the Blum axioms fully.]

2 Enumerability of Complexity Classes

§2.1 General

We have shown, in Example 1.4, that certain Blum measures have non-r.e. complexity classes. This property may be deemed undesirable, since all complexity classes are r.e. in the natural measures we traditionally use as examples. It has long been known that non-r.e. complexity classes must be low in the hierarchy of classes when they do exist. All classes larger than the first class to include all finite variants of some function must be r.e., as the following lemma indicates.

Lemma 2.1 In any Blum measure $\underline{G}$ on Gödel numbering G, if there are total recursive functions f and t such that for every total recursive function g

$$g \equiv f \text{ a.e.} \implies g \in R_t$$

then R_r is r.e. for every recursive function r s.t.

$$r \geq t \text{ a.e.}$$

Proof For each (i,j,k) in NxNxN let

$$G_{g(i,j,k)}(x) = \begin{cases} f(x) & \text{if there is a y such that} \\ & \quad y \leq j \text{ and } \underline{G}_i(y) > k \\ & \text{or } j < y \leq x \text{ and } \underline{G}_i(y) > r(y); \\ G_i(x) & \text{otherwise.} \end{cases}$$

If $G_i \in R_r$ then there exist j and k such that

$$G_{g(i,j,k)} \equiv G_i .$$

Unless $G_{g(i,j,k)} \equiv f$ a.e.,

$$\underline{G}_i(y) \leq k \text{ for } y \leq j \quad \text{and} \quad \underline{G}_i(y) \leq r(y) \text{ for } y > j ,$$

and so

$$G_{g(i,j,k)} \in R_r .$$

Otherwise

$$G_{g(i,j,k)} \in R_t \subseteq R_r .$$

Thus if

$$\{(i_\ell, j_\ell, k_\ell)\}_{\ell=1}^{\infty}$$

enumerates NxNxN then R_r is enumerated by

$$\{G_{g(i_\ell, j_\ell, k_\ell)}\}_{\ell=1}^{\infty} .$$

Q.E.D.

Example 2.2 For any measure $\underline{G}$ on G there is a t such that the identity function and its finite variants are all in R_t. To see this, start with an encoding of the finite subsets of N×N as integers,

$$\kappa = \lambda S((S=\emptyset) \longrightarrow \text{'11'};$$
$$\text{'11'}\sigma(\min(S))\kappa(S-\{\min(S)\})),$$
$$\sigma = \lambda x((x=\varepsilon) \longrightarrow \varepsilon;$$
$$(x=y\text{'0'}) \longrightarrow \sigma(y)\text{'10'};$$
$$(x=y\text{'1'}) \longrightarrow \sigma(y)\text{'01'}),$$
$$<,> = \lambda x,y(x\text{'1}0^{y}\text{'}),$$

and use it to define an enumeration of the finite variants of the identity function,

$$G_{\gamma(i)} = \lambda x((i=\kappa(S)\ \&\ <x,y>\in S) \dashrightarrow y;\ x),$$

in terms of which we define

$$t = \lambda x(\sum_{i=1}^{x} \underline{G}_{\gamma(i)}(x)) .$$

§2.2 Constant Bounds

Theorem 2.3 For any (finite) clowchart interpretation J the constant complexity classes $R^{A^J}_{\lambda x.k}$ are r.e.

Proof If f is a function in $R^{A^J}_{\lambda x.k}$ then there is a finite tree-program A^J_j which computes f. Any change we may make in A^J_j which is more than k nodes away from the root cannot change whether the function computed by the program is in $R^{A^J}_{\lambda x.k}$, since it does not change whether any paths of length greater than k are followed infinitely often. Thus, starting with a k-height tree-program, we may add on whatever suprograms we wish to any of the nodes that are reached for only finitely many inputs, and still have a program computing a function in $R^{A^J}_{\lambda x.k}$. Conversely, any function in $R^{A^J}_{\lambda x.k}$ may be gotten by adding finite tree-programs to a k-height tree-program in the same manner. Therefore, the only problem we face in enumerating $R^{A^J}_{\lambda x.k}$ is knowing for each k-height J-program which nodes may be reached only finitely often. We solve this by observing that whether a node may be reached for infinitely many inputs depends only on the sequence of function and predicate steps on the path leading to it from the root. There is only a finite number of these sequences; let the (finite, although not necessarily r.e. w.r.t. k) set of k-operation sequences that are followed for only finitely many inputs be called E_k. Enumerating $R_{\lambda x.k}$ reduces to enumerating the permissible combinations of a k-height tree-program,

a finite set of paths with operation sequences in E_k, and

and a finite set of tree-programs,

which is clearly recursive. Q.E.D.

Corollary 2.4 For any F.C. interpretation J the (constant-bounded) complexity classes $R_{\frac{A}{t}}^{J}$ are r.e., for t recursive and bounded in magnitude by some constant k.

Proof As in Theorem 2.3, start with the set of tree-programs that are bounded by t everywhere and add on other tree-programs for paths in E_k. Q.E.D.

Example 2.5

If $k = 3$, $J = (\{\underline{z}\}, \{\underline{p}, \underline{m}\})$, $\underline{z} = \lambda x(x=0)$,

$\underline{p} = \lambda x(x+1)$,

and $\underline{m} = \lambda x(\ x \neq 0 \dashrightarrow x-1;\ \ 0)$

then the set of k-operation sequences which can be followed for only finitely many inputs, E_k, is

$$\left\{ \begin{array}{l} \underline{pmz},\ \underline{mpz},\ \underline{mmz},\ \underline{mzp},\ \underline{zpp},\ \underline{zmp},\ \underline{zzp}, \\ \qquad\qquad\qquad \underline{mzm},\ \underline{zpm},\ \underline{zmm},\ \underline{zzm}, \\ \qquad\qquad\qquad \underline{mzz},\ \underline{zp\bar{z}},\ \underline{zmz},\ \underline{zzz}, \\ \qquad\qquad\qquad\qquad\qquad\quad \underline{\bar{z}mz} \end{array} \right\}$$

This means that the 3-height tree program

```
1:     z 1
2: 1   p 1     3: 1 m 1     4: 1 m 1
5:     z 1     6:   z 1     7: 1 m 1
```

may have any finite tree-program added by a 0-edge from node 5 or a 1-edge from node 6 without changing the complexity class, but any other addition at level 4 will change it.

§2.3 Ultimately Increasing Bounds

We have just shown that the flowchart complexity classes are r.e. for constant-bounded complexity bounds. It is also possible to prove that they are r.e. for ultimately increasing bounds.

Theorem 2.6 If J is a universal flowchart interpretation and t is an ultimately increasing recursive function, then the complexity class $R_{\underline{t}}^{\underline{A}^J}$ is r.e.

Proof Since t is ultimately increasing, we know that R_t includes all the functions computed by finite tree-programs. We modify each J-program to compute a function computed by a finite tree-program if the time bound t is exceeded often enough.

Take $\{(j_i,k_i)\}_{i=1}^{\infty}$ to be an enumeration of N×N; define

$$M_i(x) = \begin{cases} A_{j_i}^J(x) & \text{if } \underline{A}_{j_i}^J(y) \le k_i \text{ or } \underline{A}_{j_i}(y) \le t(y) \\ & \text{for every } y \le x; \\ A_{\tau(j_1,n)}^J(x) & \text{otherwise, where} \end{cases}$$

$$n = \max\{\ \underline{A}_j^J(z):\ \underline{A}_{j_i}^J(y) \le k_i \text{ or } \underline{A}_{j_i}^J(y) \le t(y) \text{ if } y \le z\ \}$$

and $A_{\tau(j,n)}$ is the finite tree-program containing all paths in A_j of length n or less (see Example 2.7). M_i computes $A_{j_i}^J$ if

$$\underline{A}_{j_i}^J(x) > k_i \implies \underline{A}_{j_i}^J(x) \le t(x),$$

and computes a function of finite complexity otherwise. If f is in R_t then there is a j such that A_j^J computes f and for some k

$$x > k \implies \underline{A}_j^J(x) \le t(x).$$

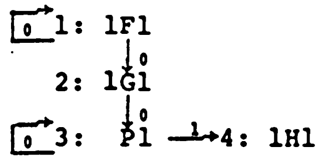
M_i computes f when j_i is j and

$$k_i = \max\{\underline{A}_j^J(x):\ x \le k\}.$$

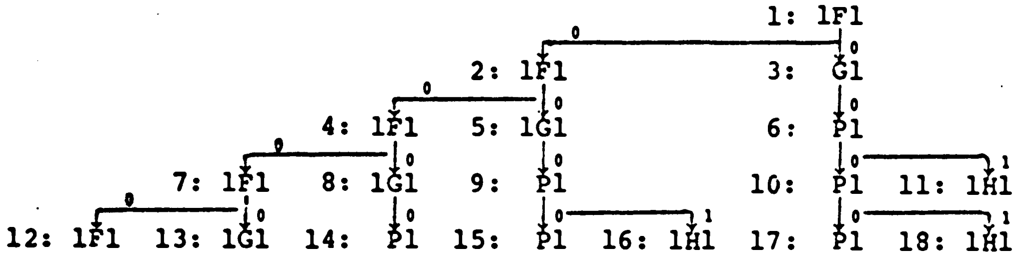
Thus $\{M_i\}_{i=1}^{\infty}$ enumerates $\mathcal{R}_t$. If J is universal this enumeration may be translated recursively into a subsequence of A^J. Q.E.D.

Example 2.7

If A_j is



then $A_{\tau(j,5)}$ is



The following theorem is a weaker version of Theorem 2.6, given because it uses Lemma 2.1 and because the technique is illustrative of the kind of manipulation possible on flowcharts.

Theorem 2.8 If J is a universal flowchart interpretation which includes the identity function, $\iota = \lambda x.x$, then the complexity classes $R_{t}^{A^J}$ are r.e. for all ultimately increasing recursive t.

Proof By Lemma 2.1 we need only show that the identity function and every finite variant of it are in $R_{\lambda x.k}^{A^J}$, each one for some constant k. Clearly the identity function is in $R_{\lambda x.0}^{A^J}$. It will be enough to show that every function f which is equivalent to the identity except at a single point is in some $R_{\lambda x.k}^{A^J}$, since any finite variant of the identity function is expressible as a finite composition of these.

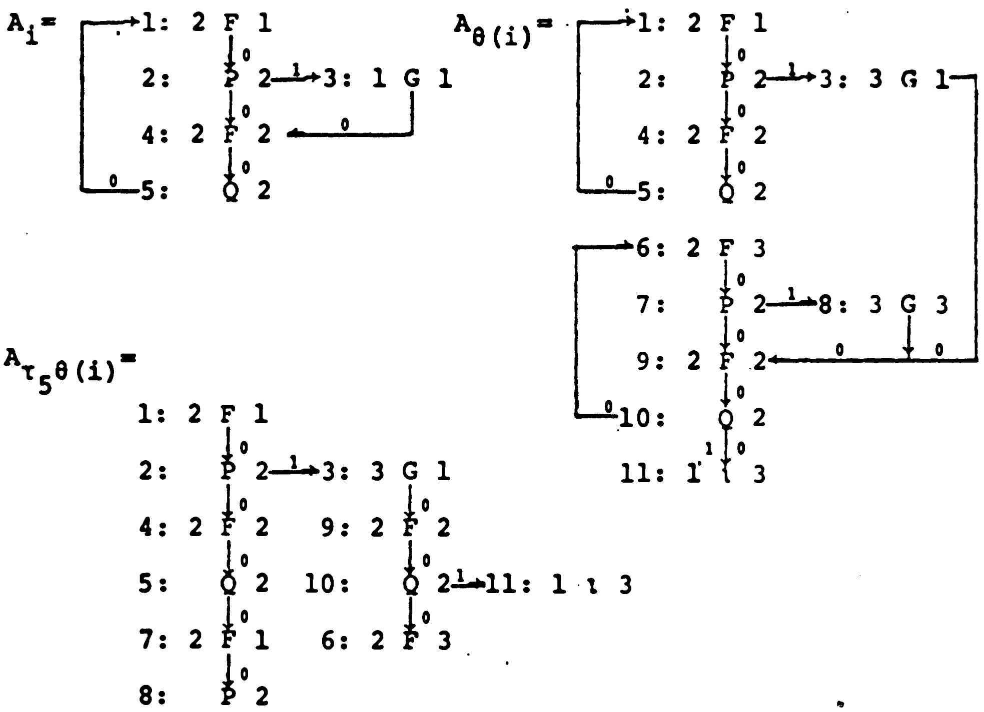
We will describe two transformations on flowcharts, θ and τ_k, which are recursive.

θ alters flowcharts so as to guarantee that, as programs, they do not destroy the contents of the input register until the final operation, yet still compute the same function. A flowchart A_i is transformed by first changing the result register index of every node with result register index 1 to the index n, where n-1 is the highest register index appearing in A_i. The edges leading from these altered nodes are then connected to the correspond ing next nodes in another copy of A_i in which all occurrences of 1 as a register index have been replaced by n. To each terminal node of the copy of A_i a link is added to a new node , $1 \iota n$. The final flowchart, $A_{\theta(i)}$, is functionally equivalent to A_i, but never alters the value of register 1 until the last step. All computational paths are at most one step longer than in A_i. (See Example 2.9.)

τ_k transforms a flowchart into a tree of height k, so that for all paths of length k or less the operations are the same, and no paths of length greater than k exist. (See Example 2.9.)

If A_i computes a p.r. function equivalent to the identity except at one point, y, then $A_j \equiv A_i$ for $j = \tau_{\underline{A}_i(y)+1}(\theta(i))$, and furthermore, $\underline{A}_j(x) \leq \underline{A}_i(y)+1$ for all x. Q.E.D.

<u>Example 2.9</u>



Theorem 2.6 and Corollary 2.4 leave an obvious gap in the classes of time bounds for which the flowchart complexity classes are known to be r.e. So far as we know, it is also an open question for Turing machine time whether the complexity classes bounded by non-constant-bounded non-ultimately-increasing functions are r.e. This is because the T.M. is generally assumed to read its input, which would require at least linear time. However, there is no solid reason for this, since certain regular sets, such as $1^*(0 + 1)^*$, are acceptable in time < lng(x) for many inputs x.

3 Translations

§3.1 Between Flowchart Interpretations

Diverse as the flowchart interpretations may be, it is always fairly simple to translate programs from one interpretation to another.

Theorem 3.1 For any two flowchart interpretations I and J, if J is universal and I finite, then there is a $\mathrm{lng}(\mathrm{lng}(x))$-tape-bounded Turing machine which translates the I-programs 1-1 into equivalent J-programs in time $O(\mathrm{lng}(x)\cdot\mathrm{lng}(\mathrm{lng}(x)))$, where x is the encoded I-program and lng is the bit-string length function.

Note: A Turing machine as used in tape-bounded computation has finite control, two-way read-only binary input tape, and read-write work tape with finite alphabet. In the present context we allow it to serve as a transducer by optionally putting a '0' or a '1' out onto an output-tape for each control-state transition.

Proof Intuitively, each node of an I-program is replaced by a J-subprogram. Since J is universal, there is a J-program for every function of I. Similarly, J must have a nontrivial predicate Q such that $Q(a)=0$ and $Q(b)=1$ for some a and b. For each predicate P of I there is a program A_i^J which computes $A_i^J(x)=$ a if $P(x)=0$ and $A_i^J(x)=$ b otherwise. The J-program represented by

$$\begin{array}{c} (\ A_i\ ) \\ {\scriptstyle 0}\big\downarrow{\scriptstyle 1} \\ \overset{0}{\text{——}}k:\ \ Q\ 1\overset{1}{\text{——}} \end{array} \quad ,$$

created by adding the predicate-node k: Q 1 to the flowchart A_i in terminal position, is operationally equivalent to the I-predicate P in the sense that the 0-edge will be followed iff P is 0 on the contents of register 1. By substituting register indices not used anywhere in the rest of the program for all the register indices local to A_i, this program segment, or subprogram, may be modified so as to have the same effect as P would when applied to register j. We thus have a J-program which is operationally equivalent for each function or predicate of I. These may be inserted in a J-program, with certain modifications of register index and the addition of entering and leaving edges, so as to have the same effect as if the corresponding I-operations were inserted.

We would like to say, without loss of generality, that these J-program counterparts to the operations of I (called translation patterns) are well behaved in certain ways:

(1) They do not alter the contents of the input register (which we shall still assume is 1) and they use some other register as output register. (If the programs we have are not already of this form, then they can be made so, without changing their complexity, by a transformation similar to

the θ described in the proof of Theorem 2.8, omitting the final transfer of the output back to register 1.)

(2) All register indices are of the form '10^n', for various n.

(3) The patterns all have the same number of nodes, this being 2^m for some fixed m. (Add inaccessible and useless nodes if necessary.)

With the flowcharts of the I-programs encoded as binary strings (see Chapter 1 for encoding), a Turing machine may translate them into flowcharts of equivalent J-programs node by node. A well-formed flowchart description is a sequence of node descriptions. Each node description consists of a function or predicate term, with register indices, followed by lists of the 0 and 1-edges from the node, given as displacements in the flowchart description. Because the well-formed descriptions are a regular set, a finite automaton may recognize the malformed descriptions, which may be translated trivially as themselves, since each malformed description represents the null flowchart. The transducer replaces each node of a well-formed flowchart description by a sequence of nodes generated from the appropriate pattern (which is stored in its finite control). Modifying the indices of the input register in the source node by suffixing a '1', it substitutes them for the indices of the input registers of the pattern, wherever they occur. In the case of the function nodes, it substitutes the similarly modified index of the result register of the source node for the output register index of the pattern. The other registers local to the pattern are given unique indices, obtained by suffixing the register indices of the pattern to the index of the source node being translated (i.e., '10^n' becomes k'10^n' in the translation of the k^{th} source node). The edges of the pattern nodes are suffixed with '0^m' to allow for the expansion which has taken place.

In doing all this the Turing machine would be performing nothing more than a finite-state transduction, were it not for the problem of local registers (which disappears if J is strongly universal— see following corollary). Keeping local register indices unique makes us count the number of nodes translated. This requires at most lng(lng(x)) tape for input x. To put this out, when we need to, as part of a register index takes about lng(lng(x)) steps. Although this happens at most a constant number of times per node, there may be about lng(x) nodes in the input, so the running time is pushed up from O(lng(x)) to O(lng(x)·lng(lng(x))) for input x.

As to the translation being 1-1, that should be clear; if it is not, then it is simpler to make it clear by encoding the original program in the form of an extra node than by going over the translation process again in detail. Q.E.D.

<u>Corollary 3.2</u> If I and J are two flowchart interpretations such that J is universal and

(1) I is finite and, for every recursive t, $R_{t}^{\underline{A}^{I}} \leq R_{t}^{\underline{A}^{J}}$, and

(2) each constant function $\lambda x.k$ is included in $R_{\ell}^{\underline{A}^{I}}$ for some other constant function ℓ,

then there is a translation σ, computable by a $\mathrm{lng}(\mathrm{lng}(x))$-tape-bounded T.M. in time $O(\mathrm{lng}(x)\cdot\mathrm{lng}(\mathrm{lng}(x)))$ (where x is the input), and a constant c such that for every j

$$A_{\sigma(j)}^{J} \equiv A_{j}^{I} \qquad \text{and} \qquad \underline{A}_{\sigma(j)}^{J} \leq c\cdot\underline{A}_{j}^{I} \quad .$$

<u>Proof</u> From (1) we know that we can get translation patterns for the functions of I which are of complexity 1 a.e. The predicates, however, do not compute functions in themselves. Thus, in order to apply (1) to show that the translation patterns for the predicates of I are also of some bounded complexity, we use (2) to convert them into programs which compute functions. For each I-predicate P and some constants a and b, there is an I-program

$$\begin{tikzcd} 1:\ P\ 1 \arrow[r, "1"] \arrow[d, "0"] & (\ B\ ) \\ (\ A\ ) & \end{tikzcd}$$

(where A and B compute the constant functions which are a and b respectively, computing a when P is 0 on the input and computing b otherwise. By (2), this program is bounded in complexity by a constant dependent only on a and b. This function must be computable by a J-program of lesser or equal complexity a.e. Since there are a finite number of these programs for the various predicates of I and they are all bounded in complexity by a constant a.e., we can remove the a.e. condition and make the constant independent of the predicate as well. The same thing can be done for the function patterns.

Following the translation procedure of Theorem 3.1 we see that the complexity of the translated J-program is bounded by a constant times the complexity of the original I-program. Q.E.D.

If J were strongly universal we could require that the pattern programs compute the same functions regardless of the contents of the non-input registers which they use. Thus the need for unique local register indices vanishes, and with it the need for any nontrivial storage capacity. The following stronger version of Corollary 3.2. then holds.

Corollary 3.3 For any flowchart interpretations I and J, if J is strongly universal and conditions (1) and (2) of Corollary 3.2 hold, then there is a translation σ, computable by two-way (read-only) finite-state transducer, and a constant c such that for every j

$$A^J_{\sigma(j)} \equiv A^I_j \qquad \text{and} \qquad \underline{A}^J_{\sigma(j)} \leq c \cdot \underline{A}^J_j .$$

Notes:

(1) A finite state transducer here is a finite automaton that can optionally put out a '0' or a '1' with each state-transition. By two-way it is meant that the automaton may

back up on its input (and thus is not operating in "real time").

(2) Having a zero-function implies that a flowchart interpretation is also strongly universal.

Theorem 3.4 If I and J are two infinite flowchart interpretations it is possible that there is no recursive translation taking I-programs to J-programs of lesser or equal complexity, even if an equivalent J-program of the same complexity exists for every I-program and I and J are universal.

Proof We construct I and J to force such a translation to solve the halting problem. Let I and J have the same functions and predicates, but let certain ones of them have different names. Let both include the zero-function under one name. Let I also include, for each i, a function

$$f_i = \lambda x((\underline{G}_i(1) > x) \longrightarrow 0;\ 1)\ .$$

Let J also include these functions, but named differently, as

$$g_i = \lambda x((i > x) \longrightarrow 0;\ 1)\ .$$

It follows that if we can translate every program

$$1:\ 1\ f_i\ 1$$

into a 1-step J-program (i.e., one of the g_i or the zero-function) then we can decide the halting problem fro all G_i (on input 1), which is well known to be undecidable.

Q.E.D.

<u>Remark</u>: If I and J are further restricted, allowing them to have only successor, predecessor, and zero-test (or some other "well-behaved" finite universal set of functions and predicates) in addition to the zero-function and the f_is or g_is, then one can say even more about the difficulty of translations from I to J. If there were a translation which were both recursive and took I-programs into equivalent J-programs of complexities <u>linearly</u> related to the complexities of the original I-programs then, the programs

$$1{:}\ \ 1\ f_i\ 1$$

would all be taken into J programs of some fixed-constant-bounded complexity. It would then be possible to transform them into finite tree-programs of constant height.

With a simple enough interpretation J, one should be able to decide whether such a program ever computes a non-zero value by examining the finite number of possible sequences of k operations allowed by the program. This is possible for the interpretation we have given and many others. It follows that for such interpretations there is no translation computable which does not slow down some algorithms by more than a linear amount.

§3.2 To Flowchart Interpretations

Theorem 3.5 For any Gödel numbering G and universal flowchart interpretation J there is a 1-1 translation from G to D^J computable by a lng(lng(x))-tape-bounded T.M. in time O(lng(x).lng(lng(x))).

Proof Intuitively, an algorithm G_i is translated into a sequence of operations which encode i, followed by a universal program. Since J is universal there are J-programs $D^J_{i_0}$, $D^J_{i_2}$, and $D^J_{i_3}$ to compute the functions

$$\lambda x((x \neq \varepsilon) \longrightarrow x'00'; \ \varepsilon) \ ,$$
$$\lambda x((x \neq \varepsilon) \longrightarrow x'10'; \ \varepsilon) \ , \text{ and}$$
$$\lambda x((x \neq \varepsilon) \longrightarrow x'11'; \ \varepsilon) \ .$$

There is also a "universal" program D^J_u to compute the partial function

$$\lambda x((x=\langle i,j\rangle) \longrightarrow G_j(i)) \ ,$$

where $\langle i,j\rangle = i'11'\sigma(j)$

and $\sigma = \lambda x((x=\varepsilon) \longrightarrow \varepsilon;$
$(x=y'1') \longrightarrow \sigma(y)'10';$
$(x=y'0') \longrightarrow \sigma(y)'00' \ ) \ .$

If $j=j_1j_2\ldots j_k$ (as a bit-string) let $D^J_{\tau(j)}$ be the composition

$$D^J_{i_3}(D^J_{i_{\sigma(j_1)}}(\ldots D^J_{i_{\sigma(j_k)}}(D^J_u) \ldots)),$$

which computes G_j by encoding $\langle i,j\rangle$ from input i and applying D^J_u. Producing $D_{\tau(j)}$ from j involves simply writing down a copy of D_{i_3} followed by a copy of D_{i_2} or D_{i_0} for each bit of j and, finally, a copy of D_u, making sure that the registers of each subprogram have unique names. We may assume without loss of generality:

(1) all the subprograms use the same register(s) for input and output;

(2) each subprogram has a single terminal node with 0- and 1-edges to a (hypothetical) node following the last node of the subprogram, which is treated as a null edge in the absence of such a node;

(3) the sets of register indices of the subprograms D_{i_3}, D_{i_2}, D_{i_0}, and D_u are disjoint, except for the input-output register(s);

(4) the register indices of D_{i_2} and D_{i_0} are of the form '10^n' for various n and the register indices of D_{i_3} and D_u are of the form x'1' for various x.

If the flowcharts do not satisfy these conditions we can change them so that they do, without changing the function computed.

Translating j to $\tau(j)$ would be even simpler than the translation process of Theorem 3.1 (since we do not have to back up to copy out register indices), were it not that the various copies of D_{i_0} and D_{i_2} might interfere with each other (by leaving non-zero values) if they had any but input-output registers in common. Keeping these unique, by counting bits of j and substituting ℓx for every register index x of $D_{i_{\sigma(j_\ell)}}$, requires lng(lng(j)) tape and boosts the time by a factor of lng(lng(j)) over lng(j), the length of the input.

Because of the direct way in which j is encoded, it is obvious that the translation is 1-1. Q.E.D.

<u>Corollary 3.6</u> For any Gödel numbering G and strongly universal flowchart interpretation J there is a 1-1 translation from G to D^J computable by (one-way) finite-state transducer.

<u>Proof</u> Knowing that J is strongly universal allows us to require that the D_{i_3}, D_{i_2}, D_{i_0}, and D_u be insensitive to changes in the initial values of any but their input registers. The assignment of unique register indices is no longer necessary and the translation technique of Theorem 3.5 gives the stronger result. Q.E.D.

§3.3 Between Gödel Numberings

Loosely speaking, the preceding two results say that the flowcharts make a good universal target language for translations (supporting our assertion that they are a natural class of Gödel numberings on which to base complexity measures, and indeed worthy of study). The next theorem shows that many Gödel numberings are worse-behaved as targets than the flowchart G.N.s and that we cannot say much good about any Gödel numbering as a universal translation source. It is a weaker version of a theorem which we prove in [11] about "optimal" G.N.s.

Theorem 3.7 For any Gödel numbering G, complexity measure $\underline{G}$, and recursive function t, there is another Gödel numbering H such that any translation from G to H must be of complexity greater than t.

Proof The desired H is defined as a recursive permutation of G by an inductive diagonalization over the possible translations G_k. Let $\{G_{j_i}\}_{i=1}^{\infty}$ be an r.e. sequence of constant functions such that for every ℓ

$$G_{j_\ell} = \lambda x(\ell) \qquad \text{and} \qquad j_\ell > \ell \ .$$

We know that such an enumeration exists by the S_n^m theorem. Let $\{G_{k_i}\}_{i=1}^{\infty}$ be a recursive enumeration of $R_{t'}^{\underline{G}}$, where $t' > t$ and $R_{t'}^{\underline{G}}$ includes all finite-variants of the zero-function (see Lemma 2.1 that $R_{t'}^{\underline{G}}$ is r.e.). By the i^{th} stage $H_1, \ldots H_{N_i}$ are assumed to have been defined. We define

$H_{N_i+1},\ldots H_{N_{i+1}}$ as follows: Compute $G_{k_i}(j_\ell)$ for $\ell=N_i+1, N_i+2,\ldots$ until one of the following cases holds:

[1] $(G_{k_i}(j_\ell) > N_i)$: Let $n = \{n: n>\ell,\ j_n > G_{k_i}(j_\ell)\}$,

$$N_{i+1} = j_n,$$

$$H_{G_{k_i}(j_\ell)} = G_{j_n},$$

$$H_{j_n} = G_{G_{k_i}(j_\ell)}, \quad \text{and}$$

for $N_i < j \leq N_{i+1}$, $j \neq G_{k_i}(j_\ell)$ and $j \neq j_n$,

$$H_j = G_j \ .$$

$$H_{G_{k_i}(j_\ell)} = G_{j_n} \equiv \lambda x.n \not\equiv \lambda x.\ell \equiv G_{j_\ell} = H_j \ .$$

[2] $(\ell > 2 \cdot N_i)$: Let $N_{i+1} = j_\ell$

and $H_j = G_j$ for $N_i < j \leq N_{i+1}$.

Then for some m and $n \leq \ell$, and such that $m \neq n$,

$$G_{k_i}(j_m) \equiv G_{k_i}(j_n),$$

$$H_{G_{k_i}(j_m)} \equiv H_{G_{k_i}(j_n)},$$

and so if $H_{G_{k_i}} \equiv G$ (as G.N.s) we have

$$\lambda x.m \equiv G_{j_m} \equiv H_{G_{k_i}(j_m)} \equiv H_{G_{k_i}(j_n)} \equiv \lambda x.n \ .$$

Clearly this halts, at the latest when ℓ reaches $2 \cdot N_i + 1$, and it always produces a contradiction from $H_{G_{k_i}} \equiv G$. Q.E.D.

Lemma 3.8 For any measure $\underline{G}$ on Gödel numbering G there is a recursive f such that for any recursive g and t,

$$g \in R_t \Longrightarrow g^{-1} \in R_{f\circ t}$$

where g^{-1} denotes

$$\lambda x(\min\{y: g(y) = x\}).$$

Proof We proceed by a series of "claims" which are "obviously" true, but some justification is sketched for each.

claim 1 There is a recursive α such that R_α includes the identity function and all its finite variants. (This is shown in Example 2.2.)

claim 2 There is a recursive γ such that for every total G_i

$$\{G_{\gamma(i,j)}\}_{j=1}^{\infty} = R_{G_i} .$$

(Let $\{(m_i,n_i)\}_{i=1}^{\infty}$ enumerate N×N s.t. $(m_i,n_i)=(k,\ell)$ i.o. for every (k,ℓ) in N×N. Define

$$G_{\gamma(i,j)} = \lambda x((\underline{G}_{m_j}(y) \le \alpha(y) \text{ or } \underline{G}_{m_j}(y) \le G_i(y) \text{ for every } y \text{ such that } n_j < y \le x) \longrightarrow G_{m_j}(x);\ x) .$$

Thus $G_{\gamma(i,j)}$ computes a finite variant of the i.d. function unless G_i is not total (in which case $G_{\gamma(i,j)}$ is a.e. undefined), unless

$$\underline{G}_{m_j} \le G_{\beta(i)} \text{ a.e., where}$$

$$G_{\beta(i)} = \lambda x(\max(\{\alpha(x), G_i(x)\})) \ . \qquad)$$

<u>claim 3</u> There is a recursive δ such that

$$G_{\delta(j)} = G_j^{-1}$$

for every j.

(Define $\qquad G_{\delta(j)} = \lambda x(\min\{y: G_j(y) = x\})$.)

We can now define

$$f_{G_i} = \lambda x(\max_{j=1}^{x}(\underline{G}_{\delta(\gamma(i,j))}(x))) \ ,$$

so that

$$G_j \in R_{G_i} \Longrightarrow G_{\delta(j)} \in R_{f_{G_i}} \ . \qquad \text{Q.E.D.}$$

Corollary 3.9 For any complexity measure G on Gödel numbering G and any recursive t there is a G.N. H such that all isomorphisms from H to G must be of complexity greater than t.

Proof Suppose no such H exists, i.e., for every H there is a t-bounded isomorphism from H to G. Applying the preceding lemma we see that there is a translation from G to H of complexity f(t). This contradicts Theorem 3.7. Q.E.D.

This is in contrast to:

Corollary 3.10 For any two universal F.C. interpretations I and J there is an isomorphism from A^I to A^J computable by a T.M. in at most $O(2^n)$ steps on input n.

Proof This follows immediately from a coarse analysis of the time required to compute an isomorphism by the usual "back and forth" method from the two 1-1 translations guaranteed by Theorem 3.1. Q.E.D.

Remark The time analysis here is very coarse. We conjecture that the actual bound should be polynomial.

4 Nondeterminism

§4.1 Discussion

Questions about the precise complexity relations between nondeterministic and deterministic models of computation have bedeviled computer theorists for a long time. This family of as-yet-unsolved problems traces its origin back to the question whether nondeterministic and deterministic linearly-bounded automata accept the same family of languages—a subject of conjecture since 1964, when Kuroda [18] showed that the family of languages accepted by the nondeterministic linearly-bounded automata is exactly the family of context-sensitive languages. Hartmanis and Hunt [13] explain the importance of this question and other problems related to it, leading up to the more recent question whether the families of languages accepted by the nondeterministic and deterministic T.M.s in time polynomially bounded w.r.t. the length of the input (frequently denoted by NP and P) are the same.

As Cook [6] and, later, Karp [17] have shown quite well, there is a large class of problems of practical interest which are solvable by nondeterministic Turing Machine in polynomially-bounded time. These problems include satisfiability for Boolean formulae in conjunctive normal form and several problems dealing with graphs, such as finding the chromatic number and detecting a Hamiltonian circuit. The surprising thing is that these problems are all interreducible by deterministic Turing machine in polynomial time and can be shown to have deterministic polynomially-time-bounded solutions only if NP=P.

Questions such as these and others relating to nondeterminism have taken on a new interest with recent efforts toward parallelism in computing hardware. Nondeterminism in computation formally expresses a notion that can be viewed intuitively as infinite parallelism. Although it is not always true that what holds for the infinite must also hold for the finite, there is at least a conceivable real model for such parallelism in an indefinitely-expanding network of tape-equipped minicomputers. Even if there is not a precise real-world counterpart to nondeterministic computation, it might be wise to ask whether anything can be gained by nondeterminism before we ask the same about finite parallelism, since theory often is more effective in dealing with the infinite than with the finite.

The flowchart measures lend themselves well to discussion of nondeterminism, the distinction between deterministic and nondeterministic programs being a simple difference in the structure of the flowcharts, and thus independent of the interpretation. In this context we may generalize Cook's [6] question, NP=?P, to NDP=?DP, where NDP and DP are the families of languages acceptable nondeterministically and deterministically by computations bounded polynomially with respect to the length of the input.

As the following theorems show, there is a large class of sets T of time-bound functions (including the set Π of polynomials applied to the input length) where whether

$$R_{\overline{T}}^{A^J} = R_{\overline{T}}^{D^J}$$

is provably dependent on the flowchart interpretation J. That is, there are some flowchart interpretations where NDP≠DP and there are others where NDP=DP. Interestingly enough, the latter interpretations are not recursively enumerable as a class of sets of functions and predicates. In fact, it is a consequence of Mehlhorn's work [23] that the set of predicates {P: an interpretation possessing as much computational speed as a single-tape T.M. may include P and still have NDP=DP} is "meagre" in the space of recursive functions.

Based on the observation that whether NDP=DP for flowcharts is dependent on the interpretation, we remark that an answer to this question for a specific natural measure, such as Turing machine time, will have to rely heavily on the peculiar local properties of the measure. The fact that the class of interpretations where NDP=DP is non-r.e. and "meagre", together with the work of Cook [6] and Karp [14], leads us to conjecture that for interpretations of practical interest NDP≠DP.

§4.2 NDP and DP for Flowchart Measures

The following theorems and corollaries deal with the relations between nondeterministic and deterministic complexity classes for flowchart measures. Theorems 4.1 and 4.2 begin by showing that, for large enough time-bound classes T, it is dependent on the flowchart interpretation whether nondeterminism permits anything to be computed that is not also computable deterministically.

With regard to Cook's [6] NP=?P question, it should be noted that the classes NP and P were defined in terms of the traditional notions of acceptance, and in the context of Turing machine and random access machine measures. On the other hand, NDP and DP, our corresponding generalized classes of nondeterministic and deterministic polynomially-acceptable languages, are defined in terms of our own version of acceptance. Seemingly as if to obfuscate relationships further for those interested in the NP=?P question, some of the theorems are stated in terms of complexity classes of functions, R_T. The latter really should present no difficulty, since the recognition problem is a special case of computing a (characteristic) function. However, the former confusion is more subtle and might be misleading did not the ability of the Turing machine and like models to adapt from one definition of acceptance to another without more than polynomial change in complexity permit us to treat NDP and DP as consistent generalizations of NP and P.

<u>Theorem 4.1</u> For any flowchart interpretation J and any r.e. class $T=\{t_j: j\epsilon I_T\}$ of recursive complexity bounds containing a t_i s.t.

[H1] for each $j\epsilon I_T$, $2^{t_i(x)}>t_j(x)$ a.e.,

[H2] there is a $j\epsilon I_T$ s.t. $t_j(x)>5\cdot t_i(x)$ a.e., and

[H3] t_i is a.e. increasing,

there is a flowchart interpretation J', gotten from J by adding a finite number of unary predicates and functions, for which

$$L_{\overline{T}}^{A^{J'}} \neq L_{\overline{T}}^{D^{J'}} .$$

<u>Proof</u> We will exhibit a set X that is accepted by a nondeterministic T-bounded J'-program, but not by any deterministic T-bounded J'-program. Intuitively, X is a diagonal set over the sets accepted by deterministic T-bounded J'-programs. We want to be able to accept X with a nondeterministic T-bounded J'-program. Assuming J' allows the necessary arithmetic operations, we know that if we have a predicate $\underline{e}$ in J' we can use $\underline{e}$ on all numbers less than or equal to $2^{t_i(x)}$ in approximately $t_i(x)$ steps nondeterministically. We will define $\underline{e}$ so that it tells us whether a given point is in X, but only by examining $\underline{e}$ on all points in the interval $S_n=\{2^{t_i(x-1)}+1,\ldots 2^{t_i(x)}\}$ will we be able to get this information, since $\underline{e}$ "hides" the point in this interval

where $\underline{e}$ is 1 so well that no T-bounded deterministic J'-program can find it.

More formally, define X to be the set

$$\{n: S_n \cap E = \emptyset,\ n \geq 1\}$$

where E is the set accepted by the predicate $\underline{e}$, which we shall define later. (The letter "E" is used for this predicate-set since it might be thought of as an exponential <u>expansion</u> of X.)

To be certain that we can write a nondeterministic T-bounded J'-program to accept X, we let J' contain the following functions and predicates, as well as any others included in J:

$\underline{\text{zero}} = \lambda x.0;$

$\underline{\text{ad1}} = \lambda x(x+1);$

$\underline{\text{suf0}} = \lambda x(2\cdot x);$

$\underline{\text{cod1}} = \lambda x((t_i(x-1)+2)\text{'}10^{x}\text{'});$

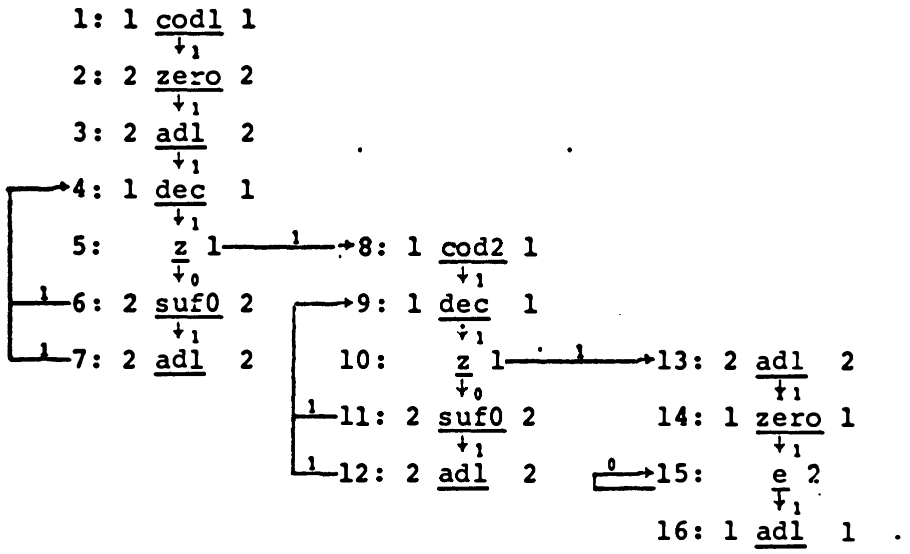
$\underline{\text{cod2}} = \lambda x((x=y\text{'}10^{z}\text{'}\ \&\ y\neq 0) \longrightarrow ((t_i(z)-t_i(z-1))\text{'}10^{z}\text{'});$
$\quad 0);$

$\underline{\text{dec}} = \lambda x((x=y\text{'}10^{z}\text{'}\ \&\ y\neq 0) \longrightarrow ((y-1)\text{'}10^{z}\text{'});\ \ 0);$

$\underline{z} = \lambda x(x=\text{'}10^{z}\text{'}\ \&\ z\in N);$

$\underline{e}$ = a predicate to be defined later.

The following J'-program accepts X in $4\cdot(t_i(x)+1)$ steps or less, and thus is T-bounded, by H2.



In the program above a number with between $t_i(n-1)+2$ and $t_i(n)+2$ binary digits is nondeterministically generated in register 2 (where n is the initial content of register 1). Such a number is between $2^{t_i(n-1)+2}$ and $2^{t_i(n)+2}-1$. One is added to the content of register 2 and the predicate $\underline{e}$ is applied to the result. This accepts X.

We now define the predicate $\underline{e}$ by induction. Let $\{(D_{i_k}, t_{j_k})\}_{k=1}^{\infty}$ be an enumeration of all pairs of a deterministic J'-flowchart and a function in T, each pair appearing infinitely often. It is clear that such an enumeration exists. Let $\underline{e}(x)=0$ for $x \leq 2^{t_i(c)+2}$, where c is chosen such that $t_i(x)>t_i(x-1)$ for $x \geq c$. For n>c assume the first n-1-c pairs have been looked at and $\underline{e}(x)$ is defined for all $x \leq 2^{t_i(n-1)+2}$. Let Z_{n-1} be the finite set of numbers greater than $2^{t_i(n-1)+2}$ for which $\underline{e}$ has already been defined to be zero. Naturally, Z_c is $\emptyset$. Now for the n^{th} pair, (D,t), the following procedure is performed:

[1] If $t(n) \geq 2^{t_i(n)}$ let $\underline{e}(x)=0$ for all $x \in S_n$ and let $Z_n = Z_{n-1} - S_{n-1}$.

[2] Otherwise compute $D^{J'}(n)$ for up to t(n) steps, recording all arguments $x > 2^{t_i(n-1)+2}$ for which $D^{J'}$ uses the predicate $\underline{e}(x)$. For all these x, $\underline{e}(x)=0$. Let $Z_n = (Z_{n-1} \cup \{x\colon D^{J'}(n) \text{ used } \underline{e}(x)\}) - S_{n-1}$.

[a] If $\underline{D}^{J'}(n) > t(n)$ then let $\underline{e}(x)=0$ for all $x \in S_n$.

[b] Otherwise

[i] If $D^{J'}(n) \neq 0$ then let $\underline{e}(x)=0$ for all $x \in S_n$.

[ii] Otherwise let $\underline{e}(x)=0$ for all $x \in S_n$ <u>except</u> the least element not in Z_n. $\underline{e}$ is 1 for this element.

From H1 we know that $t(n) < 2^{t_i(n)}$ a.e. Since the enumeration of pairs is infinitely repetitive, the pair (D,t) comes up for infinitely many n such that $t(n) < 2^{t_i(n)}$, and [2] is followed.

In [2bii] we know that the number of elements in Z_n is less than or equal to

$$\sum_{k=1}^{n} \underline{D}^{J'}_{i_k}(k) \leq \sum_{k=1}^{n} t_{j_k}(k) < \sum_{k=1}^{n} 2^{t_i(k)} \leq \sum_{k=1}^{t_i(n)} 2^k$$

which is less than $2^{t_i(n)+1}$. At any stage we have left in $S_n - Z_n$ at least

$$2^{t_i(n)+2} - 2^{t_i(n-1)+2} - 2^{t_i(n)+1} = 2^{t_i(n-1)} \cdot (2^{t_i(n)-t_i(n-1)} - 1)$$

elements, which is more than one, by H3.

$\underline{e}$ is thus well-defined, and by construction the set X is not acceptable by any T-bounded deterministic J-program, as was claimed. Q.E.D.

Theorem 4.2 For any flowchart interpretation J and any r.e. class $T=\{t_j: j \in I_T\}$ of recursive complexity bounds such that

[H4] for each $j \in I_T$ there is an $i \in I_T$ such that

$$t_i(x) \geq 2\cdot t_j(x) \text{ a.e.,}$$

there is a flowchart interpretation J', gotten from J by adding a finite number of unary functions and predicates, for which

$$R_{\overline{T}}^{A^{J'}} = R_{\overline{T}}^{D^{J'}} \quad .$$

Proof Intuitively, we add a "universal operation" which "operates" on instantaneous descriptions of nondeterministic programs in a manner similar to the way the operations of J' operate on the registers of a deterministic J'-program.

Let h be a recursive 1-1 function from the finite sets of finite sequences of non-negative integers (written N_2^+) to the non-negative integers (N) whose range is a recursive set. It is clear that such an encoding exists. Let J' contain all the functions and predicates of J plus all of the following: cod3, suf0, suf1, step, out, halt. Let $A_1, A_2, A_3, \ldots$ be an enumeration of the J'-flowcharts. Such an enumeration clearly exists. Define the new functions and predicate as follows:

$\underline{\text{cod3}} = \lambda x('10^{h(\{(1,x)\})}1')$;

$\underline{\text{suf0}} = \lambda x(x'0')$;

$\underline{\text{suf1}} = \lambda x(x'1')$;

$\underline{\text{step}} = \lambda x(((x='10^{k'}10^{j}1')\&(X\epsilon N_2^+)\&(k=h(X))\&(k'=h(A_j^{J'}[X])))$
$\longrightarrow '10^{k}10^{j}1' \; ; \; 0));$

$\underline{\text{out}} = \lambda x(((x='10^{k}10^{j}1')\&(k=h(X))\&(X\epsilon N_2^+))$
$\longrightarrow \min\{\ell_1 : (0,\ell_1,\ell_2,\ldots\ell_m)\epsilon X\}; \quad 0);$

$\underline{\text{halt}} = \lambda x((x='10^{k}10^{j}1')\&(k=h(X))\&(X\epsilon N_2^+)\&$
(X includes a sequence starting with 0)).

explanations: cod3, suf0, suf1 will be used to encode '10^k10^i1'. step is the "universal operation" we mentioned, performing one step of the program $A_i^{J'}$ on the partial results encoded in k. halt is a convergence test. out decodes the output from the encoded partial results.

We will now exhibit a program $D_{\sigma(i)}^{J'}$ which deterministically computes the same function as $A_i^{J'}$, for each i. Furthermore, this program will use

$$2\cdot \underline{A}_i^{J'}(x)+i+4$$

steps for input x, thus being T-bounded whenever $A_i^{J'}$ is, by H4.

$D_{\sigma(i)}^{J'}$:

$$
\begin{array}{rl}
1: & 1\ \underline{\text{cod3}}\ 1 \\
 & \quad\downarrow_1 \\
2: & 1\ \underline{\text{suf0}}\ 1 \\
 & \quad\downarrow_1 \\
 & \vdots \\
i+1: & 1\ \underline{\text{suf0}}\ 1 \\
 & \quad\downarrow_1 \\
i+2: & 1\ \underline{\text{suf1}}\ 1 \\
 & \quad\downarrow_1 \\
i+3: & 1\ \underline{\text{suf1}}\ 1 \\
 & \quad\downarrow_1 \\
\rightarrow i+4: & 1\ \underline{\text{step}}\ 1 \\
 & \quad\downarrow_1 \\
\ell\ \ i+5: & \quad \underline{\text{halt}}\ 1 \xrightarrow{1} i+6:\ 1\ \underline{\text{out}}\ 1 \quad .
\end{array}
$$

It should be pointed out that, although step is defined recursively, it is always convergent. The case in doubt would be when computing $A_j^{J'}[X]$ involves computing step(y) for some y. In this case, y must be smaller than the x which encodes X. For successive recursions each argument of step must be smaller. Since the initial argument is finite, step always converges. The program $D_{\sigma(i)}^{J'}$ does what we claimed and so we are done. Q.E.D.

Janos Simon [26] observed that the constructions of Theorems 4.1 and 4.2 allow us to prove the following corollary.

Corollary 4.3 It is recursively undecidable whether

$$R_{T}^{A^J} = R_{T}^{D^J}$$

for arbitrary flowchart interpretations J, where T is a class of complexity bounds satisfying H1-H4 of the preceding two theorems.

Proof We combine the previous two constructions, so that one is effective if a given Turing machine halts and the other is effective if the Turing machine does not halt.

Let J be a flowchart interpretation. Let J_p include the functions and predicates of J plus the functions zero, ad1, suf0, cod1, cod2, dec, cod3, suf1, step, out and predicates z, e, halt as in the proofs of theorems 4.1 and 4.2, except that e and step are to be defined slightly differently.

<u>step</u> is defined as before, but over the language of J_p and with the modification that <u>step</u> computes $A_i^{J'}[X]$, where J' denotes the interpretation identical to J_p except that <u>e</u> is zero everywhere. <u>step</u> thus fulfills its role as "universal operation" for J_p only if <u>e</u> is identically zero in J_p.

<u>e</u> is defined as before for arguments x such that $\underline{G}_p(0) \leq x$. For other arguments, $\underline{e}(x)=0$. Thus <u>e</u> serves its role as expansion of a diagonal set only if $G_p(0)$ converges. When $G_p(0)$ does converge, <u>e</u> gives us a set in $R_{\overline{T}}^{\underline{A}^{J}p}$ but not in $R_{\overline{T}}^{\underline{D}^{J}p}$. Otherwise, $\underline{e}\equiv 0$ and <u>step</u> "works", making

$$R_{\overline{T}}^{\underline{A}^{J}p} = R_{\overline{T}}^{\underline{D}^{J}p} .$$

Clearly a decision procedure for whether $R_{\overline{T}}^{\underline{A}^{J}p} = R_{\overline{T}}^{\underline{D}^{J}p}$ would solve the halting problem, which is widely known to be undecidable. Q.E.D.

Since the non-halting Turing machines are not r.e., we also get a stronger consequence.

<u>Corollary 4.4</u> The flowchart interpretations where

$$R_{\overline{T}}^{\underline{A}} = R_{\overline{T}}^{\underline{D}}$$

are not r.e.

The preceding results remain valid when the complexity classes L_T^A and L_T^D of languages are interchanged with the complexity classes R_T^A and R_T^D of functions. In fact,

$$L_T^A \neq L_T^D \Longrightarrow R_T^A \neq R_T^D \quad ,$$

and contrapositively. Except for minor changes in encodings and in the hypotheses H1-H4, the proofs are also independent of the precise model of flowchart computation. That is, it does not matter: whether cross-register assignments ($i \; F \; j$, $i \neq j$) are allowed; whether a nondeterministic algorithm converges when the first computation terminates or when the last one does; whether only one register, some fixed number of registers, or arbitrarily many registers are allowed.

Furthermore, although the flowchart programs defined compute functions on N (the bit strings without leading zeros), "acceptance" for general character strings may be defined in terms of a simple isomorphic representation of the character strings in N (e.g., $w \longrightarrow$ '1'w). It follows that the statements made above about complexity classes of functions and sets of integers might as well have been made about complexity classes of functions and sets over the character strings, specifically, the classes of languages NDP and DP.

K. Mehlhorn [23] has recently proposed interpretations in the domain of computable functions for some of the elementary notions of general topology. Based on our earlier formulation of Theorem 4.1 (in terms of T.M.s with "oracles") he observed that the construction gives a set of oracle functions which, by his definition, are "co-meagre". In terms of flowchart interpretations, this means that, for sufficiently powerful interpretations J, the set of predicates which J may include without causing $R_{\overline{\Pi}}^{A^J} \neq R_{\overline{\Pi}}^{D^J}$ is "meagre". It implies that in some sense "most" flowchart interpretations J satisfy $R_{\overline{\Pi}}^{A^J} \neq R_{\overline{\Pi}}^{D^J}$. Since the construction is essentially the same as that for Theorem 4.1, we give it here in brief form, along with the needed definitions.

The following are Mehlhorn's definitions:
Let T be the set of (finite) functions from an initial segment of N to N. For s to <u>extend</u> t means domain(t) is a subset of domain(s) and $s(x)=t(x)$ for all $x \in$ domain(t). Length(t)= max(domain(t)) for $t \in T$. The <u>basic open neighborhoods</u> are the sets $U_t = \{f \mid t \in T, f \in R, f \text{ extends } t\}$. A set S of recursive functions is (effectively) <u>nowhere dense</u> if there is a recursive <u>verifying function</u> $v: T \to T$ s.t.

(1) for all $t \in T$, v(t) extends t, and

(2) there is an $n \in N$ s.t. for all $t \in T$,

$$\text{length}(t) > n \Longrightarrow U_{v(t)} \cap S = \emptyset .$$

A set M of recursive functions is (effectively) meagre iff there is a sequence

$$\{(M_i, v_i)\}_{i=1}^{\infty}$$

of nowhere dense sets with verifying functions v_i such that

$$M = \bigcup_{i=1}^{\infty} M_i$$

and

$$v = \lambda i,t(v_i(t)) \text{ is total recursive.}$$

<u>Theorem 4.5</u> If J is a flowchart interpretation such that

$$R_{\Pi}^{\underline{G}} \subseteq R_{\Pi}^{\underline{D}^J}$$

then the set Q of recursive predicates which J may not include unless

$$R_{\Pi}^{\underline{A}^J} \neq R_{\Pi}^{\underline{D}^J}$$

is meagre in R.

[As stated before, $\underline{G}$ denotes the Turing machine step-counting measure for single-tape machines; Π denotes the class of functions $p_i = \lambda x(q_i(\text{lng}(x)))$, where q_i ranges over all the polynomials with integer coefficients and lng is the length function; R is the class of total recursive functions.]

<u>Proof</u> Because of the great similarity to the proof of Theorem 4.1, some details of the argument have been omitted, but the construction is given in full.

The interval $2^{\text{lng}(n)-1}+1, \ldots 2^{\text{lng}(n)}$ is denoted by S_n. If $\underline{e}$ is a predicate the set $X_{\underline{e}}$ is defined:

$$X_{\underline{e}} = \{n: \exists x(x \in S_n \ \&\ \underline{e}(x)=1)\} .$$

Let $\{(D_{m_i}, p_{n_i})\}_{i=1}^{\infty}$ be an enumeration of all pairs of a deterministic "J+$\underline{e}$"-program and an element of Π.

$$B_i = \{\underline{e} \mid X_{\underline{e}} \text{ is accepted by (deterministic "J+e"-program) } D_{m_i}^{\text{"J+}\underline{e}\text{"}} \text{ and } \underline{D}_{m_i}^{\text{"J+}\underline{e}\text{"}} \leq p_{n_i} \text{ (everywhere)}\} ,$$

where "J+$\underline{e}$" denotes J if J includes the predicate $\underline{e}$ and denotes J augmented to include $\underline{e}$ otherwise.

$$B = \bigcup_{i=1}^{\infty} B_i .$$

To satisfy the definition of meagre we must exhibit a computable verifying function $v: N \times T \to T$ such that B_i is nowhere dense with verifying function $v_i = \lambda t(v(i,t))$ for each i.

$v_i(t)$ is defined in a fashion similar to $\underline{e}$ of Theorem 4.1: Let

$$m = \min\{x: p_{n_i}(x) \leq 2^{\lg(x)-2} \;\&\; x > \text{length}(t)\} .$$

The domain of $v_i(t)$ is

$$\{x: 1 \leq x \leq 2^{\lg(x)-1}\} .$$

Cases: (A) $(x \leq m)$: let $v_i(t)(x) = t(x)$.

(B) $(x > m)$:

(1) $(\underline{D}_{m_i}(m) > p_{n_i}(m)$ or $(\underline{D}_{m_i}(m) \leq p_{n_i}(m)$ & $D_{m_i}(m) \neq 0))$: let $v_i(t)(x) = 0$.

(2) $(\underline{D}_{m_i}(m) \leq p_{n_i}(m)$ and $D_{m_i}(m) = 0)$: let $v_i(t)(x) = 0$ <u>unless</u>

$x = \min\{y: y \in S_m$ and $D_{m_i}(m)$ does <u>not</u> use $\underline{e}(y)$ in its computation$\}$;

otherwise, let $v_i(t)(x) = 1$.

Clearly $v_i(t)$ extends t. To show that B_i is nowhere dense with verifying function v_i we need only show that for all $t \in T$

$$\text{length}(t) > m \implies U_{v(t)} \cap B_i = \emptyset .$$

Suppose the contrary, that there is an $\underline{e}$ in B_i which extends $v_i(t)$ for some t such that length(t)>m. This means that the "J+$\underline{e}$"-program D_{m_i} accepts $X_{\underline{e}}$ in p_{n_i} steps, i.e.,

$\underline{D}_{m_i} \leq p_{n_i}$. Since $\underline{D}_{m_i}(m) \leq p_{n_i}(m)$ we have either $D_{m_i}(m) \neq 0$ and $\underline{e}(x) = v_i(t)(x) = 0$ for all x in S_m, or else we have $D_{m_i}(m) = 0$ and $\underline{e}(x) = v_i(t)(x) = 1$ for some x in S_m. In both cases we have a contradiction.

It follows that B is meagre. Since $R_{\Pi}^{D^J} \supseteq R_{\Pi}^{G}$ we know $X_{\underline{e}}$ is acceptable in linear time with respect to the length of the input by a nondeterministic "J+$\underline{e}$"-program, simply by "guessing" a number in S_n and applying the predicate $\underline{e}$. If $\underline{e}$ is in Q we must have

$$R_{\Pi}^{A^{\text{"}J+\underline{e}\text{"}}} = R_{\Pi}^{D^{\text{"}J+\underline{e}\text{"}}} ;$$

which means $\underline{e}$ is in B_i for some i. Thus Q=B and we are done. Q.E.D.

§4.3 Nondeterminism at Other Complexity Levels

It is natural to ask whether nondeterminism may gain anything in simpler interpretations and for tighter time bounds. It may.

Theorem 4.6 For any monotone non-decreasing recursive function t and any flowchart interpretation J which includes the function "+1" = $\lambda x(x+1)$, there is a recursive predicate P such that if J includes P then

$$R_{\underline{t}}^{A^J} \neq R_{\underline{t}}^{D^J} .$$

Proof Although the statement may appear similar to Theorem 4.1, the method of proof is quite different. The critical observation here is that a nondeterministic program may loop without executing a test operation. The predicate P is defined so as to allow a nondeterministic program to make use of this advantage to diagonalize over the deterministic programs of similar complexity.

Let $\{D_i^J\}_{i=1}^{\infty}$ be an enumeration of the J-programs with infinite repetition; that is, for every i and j there is a k>i such that $D_j^J = D_k^J$. Suppose J includes a predicate named P and that it is the same as the P defined as follows:

Stage 0: $N_0 = 0$. $P(0) = 0$.

Stage i: Assume P(y) is defined for $y \leq N_i$. Compute $D_i^J(N_i+1)$ for up to $t(N_i+1)$ steps. Whenever a value for P(y) which is not yet defined is needed in the computation of $D_i^J(N_i+1)$, assume that it is zero and add y to the set U_i (which starts empty for each i). For each new argument added to U_i (after the first one) a new function step must have been performed. Adding the function steps needed to generate the arguments together with the predicate steps needed to apply P, we get

$$\underline{D}_i^J(N_i+1) \geq 2\cdot|U_i|-1 .$$

If $D_i^J(N_i+1)$ doesn't converge within $t(N_i+1)$ steps then we give it up and go on to step i+1, letting

$$N_{i+1} = \max(U_i \cup \{N_i+1\}) \qquad \text{and}$$

$$P(y) = \begin{cases} 0 \text{ if } y \in U_i \\ 1 \text{ if } y \notin U_i \text{ and } N_i < y \leq N_{i+1} . \end{cases}$$

If, on the other hand, $D_i^J(N_i+1)$ does converge within the time bound t, then let

$$N_{i+1} = \max(U_i \cup \{D_i^J(N_i+1)\})+1 \qquad \text{and}$$

$$P(y) = \begin{cases} 0 \text{ if } y \in U_i \text{ or } y = D_i^J(N_i+1) \;\&\; y > N_i \\ 1 \text{ if } y \notin U_i \cup \{D_i^J(N_i+1)\} \;\&\; N_i < y \leq N_{i+1} . \end{cases}$$

The nondeterministic program

```
┌─0→1:    P    1
│         ↓0
└─0→2: 1 "+1"  1
```

computes the function

$$f = \lambda x(\min(\{y: y \geq x \;\&\; P(y)=1\})$$

in $2+f(x)-x$ steps or less. If $f(x)>x$ we know that $x \neq N_i$ for every i, since otherwise $f(x)=x$. Thus there is an i such that $N_i+1 \leq x < N_{i+1}$ if $f(x) > x$.

In this case we also know

$$x \in U_i \qquad \text{or} \qquad x = D_i^J(N_i+1) ,$$

and $$|U_i| \leq (t(N_i+1)+1)/2 .$$

For every y between f(x) and x, $P(y) = 0$ (including x but not f(x)), so

$$f(x)-x \leq |U_i|+1 \leq t(N_i+1)/2+1$$

and $$2+f(x)-x \leq t(N_i+1)/2+3 < t(x)/2+4$$

if t is nondecreasing. That is,

$$f \in R^{A^J}_{\lambda x(t(x)/2+4)} .$$

Since f was defined so that

$$f \in R^{D^J}_{t} ,$$

we have $$R^{A^J}_{t} \neq R^{D^J}_{t} . \qquad \text{Q.E.D.}$$

5 Relations to Other Natural Measures

The flowchart measures are a natural class of complexity measures in the sense that they do count recognizable "steps" in a specific model of computation. They do not properly include all of the commonly accepted Natural* measures, which have come to mean the single and multitape Turing machine time and tape measures and the various random-access machine step measures. There is, however, a limited sense in which these can be "mimicked" by flowchart measures. If we consider a large enough class, such as the polynomially-time-bounded functions, it is the same for some flowchart measures as it is for certain Natural measures. Although we show this only for Turing machine time and tape, Karp [17] has claimed that the classes NP and P are the same for all Natural time (as opposed to storage) measures, which would mean that the flowchart measures extend all the Natural time measures, at least in so far as the polynomially-time-bounded computations are concerned. Of course Karp gives no proof of this, but our experience with the two examples given leads us to believe it is true.

---

*[Note distinction between "Natural" and "natural".]

§5.1 Turing Machine Time

Theorem 5.1 There is no flowchart interpretation J s.t.

$$R_{\overline{t}}^{A^J} = R_{\overline{t}}^{G}$$

for every recursive t, where $R_{\overline{t}}^{G}$ is the class of t-bounded functions with respect to the T.M. step measure.

Proof We show that the tape of the T.M. forces it to compute things differently from the way a F.C. program does.

Consider a function which forces the T.M. to move its head along the entire length of the tape. Assuming a T.M. starts with its head on the leftmost bit of an input string, such a function would be:

$$f = \lambda x(x=x_1 \ldots x_n) \longrightarrow x_1 \ldots x_{n-1}\overline{x}_n; \quad 0),$$

where we denote by $\overline{x}_n$ a bit that is different from x_n. f is in $R_{\overline{\lambda x(\mathrm{lng}(x))}}^{G}$ but no function equivalent i.o. to f is in $R_{\overline{t}}^{G}$ for any $t \le \lambda x(\mathrm{lng}(x))$ i.o. Suppose there is a flowchart program A_i^J which computes this function in $\lambda x(\mathrm{lng}(x))$ steps; we can show there is another program which uses fewer steps and still computes the same function infinitely often.

A_i^J may be altered to skip the first predicate step (if there is any). This is done by making a tree-program of all the paths from the starting node of A_i^J which consist of only function nodes up to the final node, which is a predicate or terminal node. A copy of A_i^J is then made. For each predicate-node of the tree-program, consider the predicate either it is 1 i.o. or it is 0 i.o. First suppose it is 1. If the 1-edge from the corresponding predicate-node in A_i^J is null, we simply delete the node. If the edge is not null, we delete the predicate-node and replace all edges leading to it by edges leading to the node (in the copy of A_i^J) corresponding to the node (in A_i^J) to which the 1-link of the predicate-node goes. If the predicate is not 1 i.o. then we perform a similar procedure for the 0-edge. The resulting program is called A_j^J.

Since A_i^J computes a total function, every path containing a loop must include a predicate-node. Thus, unless $\underline{A}_i^J$ is bounded by a constant, the modified program we have described, A_j^J, will use one fewer step , while still performing the same sequence of function operations, for an infinite number of inputs. It thus computes a function in $R_{\underline{A}_j^J}^{\underline{A}^J}$ but not in $R_{\underline{A}_j^J}^{\underline{G}}$. Q.E.D.

Theorem 5.2 If T is an r.e. class of recursive complexity bounds s.t.

[H5] there is a t in T s.t. $t \geq \log_2$ a.e. and

[H6] for every t and t' in T and every polynomial P(x,y) with coefficients in N there is a t" in T such that

$$t'' \geq P(t,t')$$

a.e., then there is a flowchart interpretation J such that

$$R_{\underline{T}}^{\underline{A}^J} = R_{\underline{T}}^{\underline{G}} ,$$

where $\underline{G}$ is the step-counting measure for (nondeterministic) multitape Turing machines with binary alphabet.

Proof The model of T.M. which we use has a binary alphabet for input and computation, with an additional "blank" character which can be read but not written. The portion of the tape which is not occupied by input and which has not yet been read is filled with this character, which is found nowhere else. For consistency with our input-output conventions for flowchart programs the input is assumed to be only on the first tape and the output is the nonblank portion of the input tape to the right of the head at the time the machine halts. The operations for an ℓ-tape T.M. consist of an ℓ-tuple of operations from the set: print-0; print-1; print-0-move-right; print-1-move-right; print-0-move-left; print-1-move-left—— one operation per tape.

the input in a k-bit code (e.g., '0' → '10^{k-1}', '1' → '11^{k-1}') so that the T.M. will have some extra "characters" to use as markers. (The precise number k is irrelevant, since it is constant.) The initial expansion will take about $c_1 \cdot m$ steps, where c_1 is a constant and m is k times the length of the input to be encoded. It is seen that the operations and tests of J can all be computed by the T.M. in time bounded by $c_2 \cdot m$, where again c_2 is a constant and m is k times the length of the simulated register which is operated on. If the result of such an operation is stored in a register other than that which contained the argument there is the additional work for the T.M. of copying the result, but this is again linear with respect to the length of the copied simulated register. We can thus bound the number of T.M. steps for an operation of the J-program on a register of length n by $c_3 \cdot (k \cdot n)$, for some constant c_3. Since the J-program can increase the size of a register by at most two bits per operation, by nature of the operations we have allowed, the cost to simulate t steps on input of length n is at most

$$\sum_{i=1}^{t} c_3 \cdot (n+2 \cdot i) = c_3 \cdot n \cdot t + \sum_{i=1}^{t} 2 \cdot c_3 \cdot n \cdot i$$

$$\leq c_3 \cdot n^2 \cdot t + c_4 \cdot n \cdot t^2$$

which is bounded by some t", according to H5 and H6.

The flowchart interpretation J has predicates <u>zero</u> and <u>one</u> and functions <u>code</u>, <u>decode</u>, <u>p-0</u>, <u>p-1</u>, <u>p-0-r</u>, <u>p-1-r</u>, <u>p-0-l</u>, and <u>p-1-ℓ</u>. The latter functions will be seen to correspond in an obvious way to the T.M. single-tape operations.

$$\underline{\text{zero}} = \lambda x((x='1^i 0 y_1 \ldots y_n') \;\&\; (y_i=0) \;\&\; (i \le n))$$

$$\underline{\text{one}} = \lambda x((x='1^i 0 y_1 \ldots y_n') \;\&\; (y_i=1) \;\&\; (i \le n))$$

$$\underline{\text{code}} = \lambda x('10'x)$$

$$\underline{\text{decode}} = \lambda x(\, (x='1^i 0 z_1 \ldots z_i 'y) \longrightarrow y;\; x)$$

$$\begin{aligned}\underline{\text{p-0-r}} = \lambda x(((x='1^i 0 y_1 \ldots y_n') \;\&\; (i \le n)) \\ \longrightarrow '1^{i+1} 0 y_1 \ldots y_{i-1} 0 y_{i+1} \ldots y_n'; \\ ((x='1^i 0 y_1 \ldots y_n') \;\&\; (n=i+1)) \\ \longrightarrow '1^{i+1} 0 y_1 \ldots y_n 0';\; x)\end{aligned}$$

<u>p-0</u>, <u>p-1</u>, <u>p-1-r</u>, <u>p-0-l</u>, and <u>p-1-ℓ</u> are defined similarly, to mimic the function of the T.M. operations.

Clearly, a J-program can simulate an ℓ-tape T.M. which uses t(x) steps on input x, itself using $2+2\cdot\ell\cdot t(x)$ steps, simply by including, for each state of the T.M., a test tree of depth up to ℓ and at most ℓ operation nodes for each leaf of this tree. (The register naturally corresponds to a T.M. tape, with $'1^i 0'x$ encoding a tape with contents x and head in position i.)

An ℓ-tape T.M. can simulate an ℓ-register J-program with a little more difficulty. The first step is to encode

§5.2 Turing Machine Tape

Theorem 5.3 For any r.e. class T of recursive complexity bounds s.t.

[H7] there is a t in T such that $t(x) \geq x$ a.e. and

[H8] for every t in T there is a t' in T s.t. $t' \geq 2 \cdot t$

there is a flowchart interpretation J such that

$$R_{\overline{T}}^{D^J} = R_{\overline{T}}^{G'}$$

where G' is the Turing machine tape measure.

Proof J includes a "big step" function step which works on encoded instantaneous descriptions of Turing machines (i.d.s). step computes the next i.d. for which a new square of work tape would be used by a T.M. started from step's argument. By iterating step until a terminal i.d. is reached, we simulate the T.M. computation.

Define J to include:

(1) encoding functions to encode $<\xi_i,0,0,0,0,y>$ from input y, where $<\xi_i,0,0,0,0,y>$ is the i.d. of the i^{th} T.M. on input y in starting state and starting head positions.

(2) "universal big step function" step($<\xi_i,s,p,x,q,y>$) which computes what the i^{th} T.M. G_i' would do if started from i.d. $<\xi_i,s,p,x,q,y>$ and run until either the work head moves off the described work area, the T.M. halts, or the T.M. cycles, whichever comes first. ξ_i encodes a complete description of the i^{th} T.M. s is the index of

Thus, if T is as hypothesized, the functions of complexity T are the same in both measures. Q.E.D.

The preceding theorem is true also if restated in terms of deterministic flowchart programs and deterministic T.M.s. The proof is essentially the same.

the current state as it appears in ξ_i. x is the work tape, with head position p. y is the input tape, with head position q. If T.M. G_i' halts or expands its work area, <u>step</u>($\langle \xi_i,s,p,x,q,y\rangle$) is the T.M.'s i.d. at the point where it first expands its work area or halts. If G_i' cycles, without expanding its work area, then <u>step</u>($\langle \xi_i,s,p,x,q,y\rangle$) is $\langle \xi_i,s,p,x,q,y\rangle$.

(3) decoding function <u>decode</u>($\langle \xi_i,s,p,x,q,y\rangle$) = $x_q \ldots x_n$, where $x = x_1 \ldots x_n$ (leading zeros assumed dropped).

(4) halting-state-test predicate <u>halt</u>($\langle \xi_i,s,p,x,q,y\rangle$) = 1 iff x is a halting state of G_i'.

With these functions and this predicate we can write a J-program

$$\boxed{\text{encode } \langle \xi_i,0,0,0,0,y\rangle}$$

$$\downarrow$$

$$j_1\text{:} \quad r_1 \ \underline{\text{step}} \ r_1$$

$$\downarrow$$

$$j_1+1\text{:} \quad \underline{\text{halt}} \ r_1 \xrightarrow{1} j_1+2\text{:} \quad r_1 \ \underline{\text{decode}} \ r_1$$

which computes G_i' in $\underline{G}_i' \cdot 2 + c_i$ steps, if the encoding of $\langle \xi_i,0,0,0,0,y\rangle$ is done in c_i steps. In that case,

$$R_{\frac{D}{T}}^{J} \supseteq R_{\frac{G'}{T}} \quad ,$$

by H7 and H8.

$\langle \xi_i,s,p,x,q,y\rangle$ is encoded so as to make the initial encoding easy. The usual encoded form is

$$'1'\#\sigma(\xi_i)\#\sigma(s)\#\sigma(x')\#\sigma(x'')\#\sigma(y')\#\sigma(y'') \quad ,$$

where $\sigma = \lambda x\big((x{=}y'1') \longrightarrow \sigma(y)'11';$
$(x{=}y'0') \longrightarrow \sigma(y)'10'; \;\; 0\big)$,

$\# = '00'$, $\$ = '01'$, $x' = x_1 \ldots x_{p-1}$ and $x'' = x_p \ldots x_n$, and y' and y'' are defined similarly for the input tape, y. The exception to this is when y has head position 1, as in starting. Then the form is

$$'1'\#\sigma(\xi_i)\#\sigma(s)\#\sigma(x')\#\sigma(x'')\$y \; .$$

This allows us to encode $<\xi_i,0,0,0,0,y>$ in a constant number of steps independent of y, using the encoding functions

$$\underline{\text{prel}} = \lambda x('1'x) \quad \text{and}$$
$$\underline{\text{shift}} = \lambda x((x{=}'1'y) \longrightarrow '10'y; \;\; 0) \; .$$

It remains to be shown that

$$R_{\overline{T}}^{D^J} \subseteq R_{\overline{T}}^{G'} \; ,$$

i.e., that a T.M. can compute any function computed by a T-bounded deterministic J-program without using more tape squares than can be bounded by some function also in T. In simulating a J-program, a T.M. will require many steps for each operation of the J-program. We must merely assure ourselves that each operation of the J-program which the T.M. simulates will require at most a constant increase in the size of the T.M.'s work area. Initially let the T.M. copy the input onto its work tape in an expanded form, so as to get the effect of a very large character set. Conceptually, we view this as a five-track work tape,

e.g., 1 register contents: $r_1, r_2, r_3, \ldots r_n$
2 working copy of register contents
3 delimiters of registers
4 working markers for comparisons, moves, etc.
5 counting space to detect cycling

This initial encoding will require a linear increase in work area over the size of the input, but it is only done one time. The T.M. makes use of a few markers to help it simulate the various steps of the J-program on the encoded registers, which are recorded in sequence on the first track of the tape. When the program terminates, the T.M. decodes the output into single-track form and halts with its work head at the beginning This again only happens once and in fact need not increase the work area at all.

If the operations of J can be simulated in this environment, each costing at most a constant increase in work space, we can conclude that

$$R_{\overline{T}}^{D^J} \subseteq R_{\overline{T}}^{G'} ,$$

by H7 and H8. Considering these operations by groups, we see that the encoding functions, decoding functions, and predicate can easily be computed within the constant-tape-increase constraint. The only apparent difficulty is the function <u>step</u>. Since we have encoded everything in a rather simple fashion, there should be no difficulty in following the steps of the T.M. computation simulated by <u>step</u>. The counting space in the fifth track enables us to halt the simulation at a point where we can be certain it has begun to cycle. This is done by counting each step of the simulation until the space is full. This takes Q^{ℓ} steps for a fifth-track alphabet of size Q and track length ℓ. The lengths of the tapes plus the number of states of the T.M. G_i' simulated by <u>step</u> must be less than the length of the tapes of the T.M. simulating <u>step</u>, since the argument to <u>step</u> encodes the entire description ξ_i along with the tapes of G_i'. With read and write tapes of lengths n and m, and q_i states, G_i' can go at most $2^m \cdot q_i \cdot n \cdot m$ steps without cycling. This is less than Q^{q_i+n+m}, which is less than Q^{ℓ} for Q greater than 2. Since <u>step</u> was the only operation of J that could not obviously be done without more than a constant amount of additional tape, we are done. Q.E.D.

is a G_j such that $G_i|S \equiv G_j|S$ and $G_j \in R^{G}_{\lambda x(k)}$ for some constant k. This property was used to prove that the flowchart complexity classes R_t are r.e. for ultimately increasing t.

Examples have been given, in §1.4, to show that these properties do not hold for all Blum measures.

§6.2 Some "Unnatural" Failures

Many desirable properties have been proposed for "natural" measures [10]. We show that a few of these do not necessarily hold for the flowchart measures, at least in their strong forms.

6 Other Properties of the Flowchart Measures

§6.1 Two "Natural" Properties

We will attempt to show, by further examples, that the flowchart measures are better behaved than many Blum measures, while at the same time they are sufficiently diverse to allow us to speak in some generality about the characteristics of natural measures. Previous chapters have already given some properties which support this thesis. Chapters 2 and 3 illustrate two ways in which the flowchart measures are particularly well behaved. Two more "natural" properties may be recognized in the flowchart measures:

Composition

Intuitively, if the function f costs $\underline{f}$ to compute and the function g costs $\underline{g}$ to compute, then it should not cost more than $\underline{f}\circ g+\underline{g}$ to compute $f\circ g$. This property is clearly true of flowchart measures as well as the Natural measures we have discussed, such as Turing machine time and tape. We call it the composition property.

Finite Freedom

Another property which is true of the flowchart measures and also of the Natural measures (assuming that Turing machines need not read all of their input) we call "finite freedom". A Blum measure $\underline{G}$ on G has finite freedom if for every finite set S in the domain of G_i there with the predicate P, halting immediately if P is 1 and computing f otherwise. Look at $R_{\underline{A}_j^J}$. It includes infinitely many functions, among them all the functions computed by the finite tree-programs corresponding to A_j^J's computations on finite sets of inputs. What is worse, $R_{\underline{A}_j^J}$ includes the identity function, but not all of its finite variants. For example, there is a k such that the function

$$\underline{i0}_k = \lambda x((x=k) \longrightarrow 0; \; x)$$

is not in $R_{\underline{A}_i^J}$, as the following reasoning shows: Suppose $A_{i_k}^J$ computes $\underline{i0}_k$ in time $\underline{A}_j^J$ a.e. Since

$$P(x) = 1 \Longrightarrow \underline{A}_i^J(x) = 1,$$

$A_{i_k}^J$ must begin with a predicate node which terminates the program almost every time that P is 1 on the input. There is a finite number of predicates in J, so that for some k

$$P \equiv Q \text{ a.e.} \Longrightarrow P(k) = Q(k) = 1$$

$$\text{and} \quad P \equiv \overline{Q} \text{ a.e.} \Longrightarrow P(k) = \overline{Q}(k) = 1$$

for every Q in J. For such a k every predicate which could halt $A_{i_k}^J$ in $\underline{A}_j^J$ steps a.e. would also force it to halt immediately on input k, so

$$A_{i_k}^J(k) = k. \qquad \text{Q.E.D.}$$

Finite Invariance

A Blum measure $\underline{G}$ on Gödel numbering G is <u>finitely invariant</u> iff for all recursive G_i and G_j

$$G_i \equiv G_j \text{ a.e.} \Longrightarrow (\forall t \in R)(G_i \in R_t \Longleftrightarrow G_j \in R_t) ,$$

i.e., if G_i is in R_t then all finite variants of it are in R_t also, for every recursive t. Intuitively, this says that a small change in a function should only cause a small change in its complexity. The Turing machine tape measure is clearly finitely invariant. It is unfortunate, but the flowchart measures do not all share this elegant property, as proven in the following lemmas.

<u>Lemma 6.1</u> For any Blum measure $\underline{G}$ on G and any recursive function t there is a recursive f such that

$$G_i \equiv f \text{ i.o.} \Longrightarrow \underline{G}_i > t \text{ i.o.}$$

<u>Proof</u> (trivial) Let

$$f = \lambda x\left(\sum_{i=1}^{x} ((\underline{G}_i(x) \leq t(x)) \longrightarrow G_i(x);\ 0)\right) .$$

If $\underline{G}_i \leq t$ a.e. then, clearly, $f > G_i$ a.e. Q.E.D.

<u>Lemma 6.2</u> For any universal flowchart interpretation J with a predicate P that is 0 i.o. and 1 i.o. $\underline{A}^J$, is <u>not</u> finitely invariant.

<u>Proof</u> The f of Lemma 6.1 is not computable i.o. in less than t steps. Take t > 1. Since J is universal there is a program $A^J_{i_f}$ which computes f. Let A^J_j be the program gotten by preceding A_{i_f} by a node which tests the input

Properness

A Blum measure $\underline{G}$ on G.N. G is proper iff for every i

$$\underline{G}_i \in R^{\underline{G}}_{\underline{G}_i} .$$

This means intuitively that the running time of an algorithm should be at least as easy to compute as it is to run the algorithm. Although this seems a good idea, it may be a little too strongly stated. We shall see that many flowchart measures come close to satisfying it without satisfying it precisely.

Lemma 6.3 If flowchart interpretation J includes the function $\lambda x(x+1)$ then for every i

$$\underline{A}_i^J \in R^{\underline{A}^J}_{\lambda x(2\cdot \underline{A}_i^J(x))} .$$

Proof (trivial) Alter the program A_i by putting an addition operation between every two steps, doing the final addition into the output register at termination. The program counts the steps of A_i. Q.E.D.

This idea can be taken a step further, expanding the program to make all cycles of lengths divisible by k and no cycle beginning closer than k nodes to the initial node. The steps are counted by adding one for the first k steps, adding k every k steps thereafter, and adding 1 up to k times at termination. This yields a stonger version of the lemma.

<u>Lemma 6.4</u> If J includes the functions $\lambda x(x+1)$ and $\lambda x,y(x+y)$ then for every i and k

$$\underline{A}_i^J \in R^{A^J}_{\lambda x(\underline{A}_i^J(x)\cdot(1+\frac{1}{k})+2\cdot k)} .$$

(Proof has been sketched above.)

This means that for reasonably powerful interpretations there is a t arbitrarily close to each $\underline{A}_i$ such that $\underline{A}_i \in R_t$. Now we look at a specific one of these interpretations to see that we can do no better than this.

Let $J = (\{\underline{P},\underline{Z}\},\{\underline{plus}, \underline{plus1}, \underline{minus1}\})$ where

$\underline{minus1} = \lambda x(x-1)$,

$\underline{plus1} = \lambda x(x+1)$,

$\underline{plus} = \lambda x,y(x+y)$,

$\underline{Z} = \lambda x(x=0)$,

$\underline{P} = \lambda x((x\neq y'10^{k}') \longrightarrow 0;$
$\quad (x\neq N_i'10^{k}') \longrightarrow 1;$
$\quad (x=N_i'10^{k}i') \longrightarrow 0;$
$\quad (x=N_i'10^{(\zeta(N_i)-3)/2}) \longrightarrow 0;\ 1)$,

$\zeta = \lambda x(x'10^9 10^9 1')$,

$k_i = \min\{k$: $(A^J_{r(i)}(N_i)$ does not use $P(N_i'10^{k}')$ in the first $\zeta(N_i)$ steps, $A^J_{r(i)}(N_i)\neq 3+2\cdot k$, and $k<(\zeta(N_i)-3)/2$ or $k=(\zeta(N_i)-3)/2\}$,

$N_1 = 1$

$N_{i+1} = N_i\cdot 2^{\zeta(N_i)+2}$, and

r recursive s.t. $r(i)=k$ i.o. for every k in N.

Let $f(x)$ be the running time on input x of the following program:

```
    1: 1 plus  1
    2: 1 plus1 1
 -->3:     P   1
 |__4: 1 plus  1
   0
```

which is 3 for all inputs but N_i, and which is $3+2\cdot k_i$ for N_i. Suppose A_j^J is a program which computes f in f steps a.e. Then $j=r(i)$ and

$$\zeta(N_i) = 3+2\cdot k_i = A_j^J(N_i) \geq \underline{A}_j^J(N_i) \quad \text{i.o.}$$

To compute n values of $N_i{}'10^{k}{}'$ and apply P to each of them requires at least $n+(n-1)$ steps. Thus, since $A_j^J(N_i)$ must use $P(N_i{}'10^{k}{}')$ for every $k < (\zeta(N_i)-3)/2$ such that

$$A_i^J(N_i) \neq 3+2\cdot k$$

at least $\zeta(N_i)-6$ steps are needed for this purpose. We know that

$$\underline{A}_j^J(N_i) \leq \zeta(N_i)$$

so there are only 6 steps left for the rest of $A_j^J(N_i)$'s computation. If $\zeta(N_i) = N_i{}'10^9 10^9 1{}'$ is computed, then for infinitely many i it must be computable by some 6-step process from N_i or from $N_i{}'10^{k}{}'$ for some k. It is not. We have proved the following lemma.

Lemma 6.5 There is a flowchart interpretation J such that $\underline{A}^J$ is not proper but such that for every i and k there is a recursive t such that

$$t - \underline{A}_j^J \leq \frac{A_i^J}{k} + 2 \cdot k$$

and

$$\underline{A}_j^J \in R_{\underline{t}}^{\underline{A}^J} .$$

We comment that the intent of the notion of properness, that it should be no harder to compute the running time of an algorithm than it is to run the algorithm, is honored by measures such as those described in Lemmas 6.3 and 6.4. Lemma 6.5 really only shows that "properness" should not be stated in quite so strong a form. Perhaps

$$\underline{G}_i \in R_{L(G_i)}$$

(where L is linear or "close" in some sense to the identity) is good enough.

§6.3 Further Dimensions of Flexibility

The results in chapters 4 and five show to some extent the generality of the flowchart measures, demonstrating how they may easily be tailored to possess certain properties or not. The results which follow explore other dimensions of flexibility exhibited by these measures.

Theorem 6.6 For any flowchart interpretation J there is another interpretation J' such that, for every recursive t,

$$R_t^{A^J} \subseteq R_{\log_2 t}^{A^{J'}} .$$

Proof J' will include encoding functions, a "universal" function, a halting test, and a decoding function. cod3, suf0, and suf1 are defined as in the proof of Theorem 4.2.

$$\underline{\text{jump}} = \lambda x(((x='10^k10^i10^j') \; \& \; (k=h(X)) \; \& \; (X\epsilon N_2^+) \; \& \; (k'=h(A_i^{J\,(2^{i+6+3\cdot j})}[X])))$$
$$\longrightarrow \; '10^k10^i10^j'; \quad 0)$$

$$\underline{\text{out}} = \lambda x(((x='10^k10^i10^j') \; \& \; (k=h(X)) \; \& \; (X\epsilon N_2^+))$$
$$\longrightarrow \; \min\{\ell_1: (0,\ell_1,\ell_2,\ell_3,\ldots \ell_m)\epsilon X\}; \quad 0)$$

$$\underline{\text{halt}} = \lambda x((x='10^k10^i10^j') \; \& \; (k=h(X)) \; \& \; (X\epsilon N_2^+) \; \& \; (X \text{ includes a sequence starting with } 0))$$

where $A_i^{J\,(n)}[X]$ denotes the result of iterating A_i^J n times on the set X of i.d.s, and h is the encoding function of Theorem 4.2.

For each program A_i^J we have a program $A_{\tau(i)}^{J'}$ which is equivalent to it, but faster by a logarithm.

```
          1:  1 cod3 1
                 ↓0
          2:  1 suf0 1
                 ↓0
          ⋮
        i+1:  1 suf0 1
                 ↓0
        i+2:  1 suf1 1
                 ↓0
  ┌────→i+3:  1 suf0 1
  │              ↓0
  │     i+4:  2 jump 1
  │              ↓0
  └──0── i+5:    halt 2 ───1───→ i+6:  1 out  2
```

This program computes

$$((2^{h(\{(i,\text{input})\})+1}+1)\,2^{i+1}+1)\,2^{j}$$

for successive j, starting with 1, applying jump and halt to each, until A_i^J would have halted in $i+6+3\cdot j$ steps, at which point out decodes the output and the program halts. The program uses $i+2+3\cdot j+1$ steps to compute $A_i^J(x)$ if

$$\underline{A}_i^J(x) \leq 2^{i+6+3\cdot j} \quad .$$

Thus, if

$$\underline{A}_i^J(x) > 2^{i+6+3(j-1)}$$

then

$$\log\,(\underline{A}_i^J(x)) > i+6+3\cdot j-3 = i+3\cdot j+3$$

and so

$$\underline{A}_i^J \in R^{A^J}_{\log\,\underline{A}_i^J} \quad . \qquad\qquad \text{Q.E.D.}$$

This says we can always make all computations easier by making the interpretations more powerful, an obvious truth. The log function used here is for convenience of proof only. Other speed increases are provable. In effect, we can always speed up the flowchart "machine".

Every complexity measure defines an "order" relation on the computable functions: f may be said to be <u>no more complex than</u> g iff for each algorithm which computes g there is an algorithm for f which is of equal or lower complexity, i.e.,

$$\forall\, t \in R,\quad g \in R_t \Longrightarrow f \in R_t \,.$$

This relation is clearly reflexive and transitive. It lacks the property of antisymmetry of being a partial order, but

$$f \sim g \Longleftrightarrow (\forall\, t \in R,\ g \in R_t \Longleftrightarrow f \in R_t)$$

is an equivalence relation. Looking at the functions, modulo this relation, we have an alternative concept of "complexity class", these equivalence classes being partially ordered. It is natural to ask whether these orderings have any properties in common and to what extent the ordering may vary with the complexity measure. In the case of the flowchart measures we make a start by showing that the relation between two functions may be quite dependent on the interpretation.

<u>Theorem 6.7</u> For any total recursive function t there exist flowchart interpretations I and J and total recursive functions f and g such that

$$f \in R^{A^I}_{\lambda x.1} \quad \text{and} \quad g \notin R^{A^I}_{\lambda x.1} \qquad (g \in R^{A^I}_{t})$$

but

$$g \in R^{A^J}_{\lambda x.1} \quad \text{and} \quad f \notin R^{A^J}_{t} .$$

<u>Proof</u> I and J have a set of common functions and predicates, which may be chosen to make them universal. In addition, each has one other function; I has f and J has g. f and g are defined by a double diagonalization so that g is not computable by any t-bounded I-program and so that f is not computable by any t-bounded J-program. (The details are similar to previous diagonal constructions we have done.)

<u>Theorem 6.8</u> For any universal flowchart interpretation J and any recursive functions r and t there exists an extension J', gotten by adding a recursive function g to J, such that, for some recursive function f and some recursive function s > r, if t $\geq$ s (> r) then

$$f \notin R^{A^I}_{r}, \quad f \in R^{A^I}_{t}, \quad g \notin R^{A^I}_{t},$$
$$g \in R^{A^J}_{r}, \quad f \in R^{A^J}_{t}.$$

<u>Proof</u> f is gotten by diagonalizing over all r-bounded I-programs. s is any function greater than or equal to the time needed to compute f by an I-program. g is gotten by diagonalizing over all t-bounded I-programs. (Again the details are similar to previous proofs.)

Theorem 6.9 For any flowchart interpretation I there is another flowchart interpretation J such that every I-program has an equivalent J-program which uses a single register and has complexity linearly related to the I-program. Furthermore, for every J-program there is another J-program for which the same is true.

Proof It is shown that arbitrarily many registers may be encoded into a single register and the functions of I modified to work on the encoded registers without more than linear loss of time.

The functions and predicates of I are carried over into J, but modified so as to treat their arguments as encoding several "registers" plus information about which "register" is to be treated as argument and, in the case of functions, which "register" is to receive the result. To make the "furthermore" clause of the theorem true, J must also include functions and predicates which bear the same relation to the modified functions and predicates of I just described as those do to the unmodified functions and predicates of I. Since the number of these may be infinite and we wish to keep J finite, we so this by means of a universal function, F_0', and a universal predicate, P_0'.

The encoding and interpretation J are formally defined as follows: Let $F_1, \dots F_m$ be the functions of I and $P_1, \dots P_n$ be the predicates. Let $\langle i,j,k,r_\ell,\dots r_1\rangle$ denote

$$'1'\#\sigma(i)\#\sigma(j)\#\sigma(k)\#\sigma(r_\ell)\cdots\#\sigma(r_2)\dots\$r_1\ ,$$

where

$$\sigma = \lambda x(\ x=y'0' \longrightarrow \ \sigma(y)'10';$$

$$x=y'1' \longrightarrow \ \sigma(y)'11';\ \ 0\ ),$$

$\# =$ '00', and

$\$ =$ '01'.

This encodes the indices of the result and argument "registers", the operation code for the universal operations, and the "registers".

$F'_p(<i,j,k,r_\ell,\ldots r_1>) = <0,0,0,r'_m,\ldots r'_1>,$
for F_p in I, where $r_n=0$ for $n\geq\ell$, $m=\max(\{\ell,i\})$, $r'_n=r_n$ for $n\leq\ell,\neq i$, $r'_n=0$ for $n>\ell,\neq i$, and $r'_i=F_p(r_j)$.

$P'_p(<i,j,k,r_\ell,\ldots r_1>) = P_p(r_j)$
for P_p in I and $\ell\geq j$.

$P'_p(<i,j,k,r_\ell,\ldots r_1>) = P_p(0)$
for P_p in I and $\ell<j$.

$F'_{m+1}(x) = \underline{\text{sufl}}(x) = 2^{\lceil\log_2 x\rceil+1}+x.$ (encoding)

$F'_{m+2}(x) = \underline{\text{shift}}(x) = \underline{\text{sufl}}(x) - 2^{\lceil\log_2 x\rceil}.$ (encoding)

$F'_{m+3}(<i,j,k,r_\ell,\ldots r_1>) = r_1.$ (decoding)

$F'_{m+4}(<i,j,k,r_\ell,\ldots r_1>) = \sigma(r_\ell)\ldots\#\sigma(r_2)\$r_1.$ (clean-up)

$F'_0(<i,j,k,r_\ell,\ldots r_1>) = <0,0,0,r'_m,\ldots r'_1>,$
for $k\leq m+4$, where $r_n=0$ for $n\geq\ell$, $m=\max(\{\ell,i\})$, $r'_n=r_n$ for $n\leq\ell,\neq i$, $r'_n=0$ for $n>\ell,\neq i$, and $r'_i=F'_k(r_j)$.

$P_0'(<i,j,k,r_\ell,\ldots r_1>) = P_k'(r_j)$

for $k \leq n$ and $\ell \geq j$.

$P_0'(<i,j,k,r_\ell,\ldots r_1>) = P_k'(0)$

for $k \leq n$ and $\ell < j$.

For all cases not yet defined, the functions are defined to yield the identity and the predicates to yield 1. Thus in the last case given for P_0' the value will always be 1, since 0 is not decodable into $<i,j,k,r_\ell,\ldots r_1>$.

A simple inductive argument shows that F_0' and P_0' are total, even though they are defined recursively: If $k \neq 0$ there is no problem. For $k=0$, the only question is whether $F_0'(r_j)$ and $P_0'(r_j)$ are defined. If r_j is not decodable as $<i',j',k',r_\ell',\ldots r_1'>$ then $F_0'(r_j)=r_j$ and $P_0'(r_j)=1$ (by convention above). If r_j can be decoded then the question is reduced to one about $F_0'(r_{j'}')$ and $P_0'(r_{j'}')$. By nature of the encoding, $r_{j'}' < r_j$. Since r_j smaller than '1'###\$ are clearly undecodable, the initial step of the induction must hold, and we are done.

Given an I-program A_p^I, we can translate it to an equivalent J-program as follows. For each node i F_k j in A_p we substitute a sequence of nodes which compute $<i,j,0,r_\ell,\ldots r_1>$ from every string of the form $<x,y,z,r_\ell,\ldots r_1>$, followed by a node 1 F_k' 1. (The sequence which computes $<i,j,0,r\ ,\ldots r_1>$ uses F_{m+1}', F_{m+2}', and F_{m+4}' and is dependent only on i and j.) For each node P_k j in A_p we similarly substitute a sequence which computes $\langle 0,j,0,r_\ell,\ldots r_1\rangle$ in register 1, followed by P_0' 1. Initially, we add a sequence of nodes to compute $\langle 0,0,0,r_1\rangle$ from r_1 and, finally, we add a node 1 F_{m+3}' 1 following each of the previous terminal nodes.

For a J-program what we do is analogous. Each node i F_k' j is expanded to a computation of $\langle i,j,k,r_\ell,\ldots r_1\rangle$ followed by a node 1 F_0' 1, and similarly for predicates.

The complexity of the resulting program bears a simple relationship to the complexity of the original program. A constant number of steps are added at the beginning and at the end. Additionally, each step of the original program is expanded into an encoding sequence and a "computational" step. The length of the encoding sequence is bounded by a constant for each program, since the program and interpretation are finite. The complexity of the resulting single-register program can thus be bounded by c_1 plus c_2 times the complexity of the original program, where c_1=10 and c_2 is dependent on the size of the interpretation and the number of registers used in the program. Q.E.D.

Although the previous theorem is stated in terms of 1-argument interpretations, by the convention we made in §1.3, it could also be proved for the n-ary interpretations by more complex encoding.

Others have defined notions of program which deal with what are supposedly more general domains. The idea is that for full generality one would like to be able to think of the programs as operating on objects which might be any one of the numerous data structure conceivable— arrays, stacks, queues, lists, ... The "power" of a class of uninterpreted flowcharts as functionals may depend on the domain. In terms of interpreted flowcharts, however, there is no reason why the domain cannot be restricted to the integers. It is well known that all partial recursive functions can be computed by programs with a single variable over the integers and a proper finite interpetation. In effect, the one variable becomes the entire data space of the computation. By suitable encodings the operations of the program may work on various portions of it in any way that is desired.

<u>Remark</u> The preceding theorem is particularly significant in view of Ianov's work [16] on schemata. Despite the fact that schematic equivalence (whether two schemata define the same program under all interpretations) is undecidable for schemata with more than one free variable [21], Ianov has shown that equivalence <u>is</u> decidable for schemata with a single free variable, and that for such schemata each may be reduced to a simple canonical form. Since we cannot increase the complexity of a program by more than a constant factor in going to an interpretation where only a single variable is necessary, we can translate any program (in linear time if we have a zero-function) to a single-variable program, perform any sort of optimization we wish, using Ianov's equivalence procedure, and translate it back to the original interpretation, again without any more than a linear increase in complexity. Thus, if there is anything of higher than linear order to be gained by optimization on the schema level we can achieve it regardless of how many free variables we allow.

A further consequence of this theorem is that all previous proofs of theorems about classes of complexity bounds which are closed under multiplication by a constant also hold for the same theorems if we restrict ourselves to single-register programs.

7 Conclusion

§7.1 Summary of Results

We have shown that the flowchart complexity measures form a diverse class, useful in demonstrating that certain properties proposed as requirements for "natural" measures need not be true for all natural measures. Finite invariance and properness are examples of such properties. We believe that there are others.

We have shown that nondeterminism is meaningful in a very general framework, but that its effect on the complexity of functions is dependent on the specific model of computation.

In general, we have taken a step toward isolating some of the nonessential properties of natural measures from the more essential ones. Two properties which seem essential, at least so far, we have called "composition" and "finite freedom".

We have shown that the flowchart measures are well-behaved in two senses. Many of the complexity classes are provably r.e. This is because we may perform certain transformations on the flowcharts, knowing the effect they will have on the programs without knowing exactly what the programs do. These schematic manipulations also allow us to do simple translations, which in some cases produce translated programs that are reasonably efficient.

§7.2 Possibilities for Further Research

A. Loose Ends

In the course of this research several questions arose which were not answered. We have broken these into two groups. The "loose ends" are simple questions which represent gaps in our results. Those marked by * we believe we see how to answer, although time has prevented us from including the finished proofs here.

(1) Are the flowchart complexity classes r.e. for non-ultimately-increasing and nonconstant time bounds?

(2) Can better bounds on translation cost for "worst case" programs be found for translations:

- (a) from one flowchart interpretation to another?
- (b) from a Gödel numbering to a flowchart Gödel numbering?
- (c) yielding an isomorphism between flowchart interpretations?

(3) Can the complexity of the resulting program be simply bounded for translations between flowchart interpretations with equivalent complexity measures under weaker conditions than those of Corollary 3.2?

(4) Are the interpretations where DP=NDP r.e.?

(5) Are there a t' and a t such that

$$R_{\overline{t'}}^{D^J} = R_{\overline{t'}}^{A^J} = R_{\overline{t}}^{A^J} \neq R_{\overline{t}}^{D^J} \quad ?$$

That is, are there "gaps" in the nondeterministic classes where there are none in the deterministic classes?

(6)* We know we can always make a function "easy" by adding to the interpretation. Can we <u>always</u> change the interpretation to make a specific function <u>harder</u> to compute?

(7) Can some well-known "Natural" measure be simulated more precisely by a flowchart measure than we have done? That is, so that the complexity classes are the same for all bounds?

(8)* Can we give examples of universal flowchart interpretations which give finitely-invariant or proper measures?

(9)* Are there any other "natural" properties [10] which either hold for all flowchart measures or are interpretation-dependent (e.g., conformity, parallelism, downward diagonalization, denseness)?

(10)* Can recursive oracles be found for which the determinism-nondeterminism question for linearly-bounded automata with oracles, NDL?=DL, may be answered "yes"? (We know there are oracles for which the answer is "no".)

(11) May a flowchart interpretation be N-universal without being universal? (See p. 20.)

B. Open Areas

The remaining questions are much broader in scope, and cannot be asked quite so simply.

(1) [General Nondeterminism] What kinds of functions can be computed nondeterministically if we say that a nondeterministic algorithm computes a set-valued function, converging only when all computations have terminated? How much more powerful is this model than the one where all computations are halted with the first to terminate? Are there other interesting notions of nondeterministic algorithm or of parallel computation with which to compare these?*

(2) [Complexity of Flowcharts] How does the complexity of a flowchart as a graph relate to its complexity as an algorithm? For example, are planar graphs less powerful as flowcharts than nonplanar graphs? This area might border on work which has been done in program schemata.

(3) [General Step Measures] We regard the flowchart measures as "true" step measures because there is a recognizable computational operation for each "step" counted in the measure. Is there a nice abstract characterization of measures which have this "property"? Can we go to an abstract notion of "true" step measure without losing all of the nice "natural" properties that we have not lost already by accepting the flowchart measures as natural?

---

*[We have some conjectures and preliminary results in this area, hinted at in §1.5, which do not properly fall within the range of this thesis.]

§7.3 Step Measures

K. Weihrauch [27] has shown that *any* Blum measure may be realized as a submeasure of a flowchart measure. This means, as Hartmanis [10] has explained, that we cannot accept as an "improvement" on the Blum axiomatization any set of properties which are transitive with respect to submeasures. The example he gives of a property which may hold for a measure without holding for all of its submeasures is finite invariance, which we have shown to be excessively restrictive.

Although we have stated two properties which are characteristic of flowchart measures, and natural at least in that sense, we do not believe that they are sufficient to define a "natural" measure. The naturalness of a measure seems unavoidably connected to the existence of a corresponding model of computation, with recognizable steps and intermediate results related in some way to the results of shorter computations. We believe that the existence of a partial result (produced by a recognizable operation from a previous result) for each "step" counted should be a basic requirement for all step measures. The flowcharts do this, but have a good deal more structure than is needed. It should be possible to define a structure more abstract than the flowcharts which still has the properties we admire in them.

A possible start would be to hypothesize a single data space (represented by an integer variable, in the manner of Ianov [16]), with a set of unit-cost operations (a recursive, or finite, set of recursive functions), a pair of input-output encoding-decoding functions, and a set of algorithms (next-operation choice functions dependent only on the result of the previous step). The operations would determine a natural congruence relation and a metric on the space of possible partial results (the orbit of the range of the input encoding under the action of the free group generated by the operations). The space of partial results would admit a natural partition under the obvious equivalence relation defined by the algorithms. The equivalence classes so defined could naturally be called "states". One might even go so far as to require that the partial results decompose into a Cartesian product of the "states" and another ("data") space, although this seems unnecessary. The axioms would basically be closures on the set of algorithms, to assure the possibility of "programming" in some natural manner. It is hoped that a few such closures would be sufficient to get some interesting properties about the step measures (operation counts).

References

[1] Blum, M. "A Machine Independent Theory of the Complexity of Recursive Functions", JACM, Vol. 15, No. 2, 1967, pp. 322-336.

[2] Borodin, A., R.L. Constable and J.E. Hopcroft. "Dense and Non-Dense Families of Complexity Classes", IEEE Conference Record of 1969 Symposium on Switching and Automata Theory, 1969, pp. 7-19.

[3] Chandra, Ashok K. "On the Properties and Applications of Program Schemas", Stanford Computer Science Department Technical Report 73-336, 1973.

[4] Church, A., The Calculi of Lambda Conversion, Princeton University Press, 1941.

[5] Constable, R.L. "Type Two Computational Complexity", Procedings of Fifth Annual ACM Symposium on Theory of Computing, 1973, pp. 108-121.

[6] Cook, S.A. "The Complexity of Theorem-Proving Procedures", Third Annual ACM Symposium on Theory of Computing, 1971, pp. 151-158.

[7] Cook, S.A. "Linear Time Simulation of Deterministic Two-Way Pushdown Automata", Information Processing 71, North-Holland, 1972.

[8] Harary, F., Graph Theory, Addison-Wesley, 1969.

[9] Hartmanis, J. "Computational Complexity of Random Access Stored Program Machines", Mathematical Systems Theory, Vol. 5, No. 3, 1971, pp. 232-245.

[10] Hartmanis, J. "On the Problem of Finding Natural Computational Complexity Measures", Technical Report 73-175, Department of Computer Science, Cornell University, 1973.

[11] Hartmanis, J. and T.P. Baker "On Simple Gödel Numberings and Translations", Technical Report 73-179, Department of Computer Science, Cornell University, 1973.

[12] Hartmanis, J. and J.E. Hopcroft "An Overview of the Theory of Computational Complexity", JACM, Vol. 18, No. 3, 1971, pp. 444-475.

[13] Hartmanis, J. and H.B. Hunt III "The LBA Problem and its Importance in the Theory of Computing", Technical Report 73-171, Department of Computer Science, Cornell University, 1973.

[14] Hartmanis, J. and R.E. Stearns "On the Computational Complexity of Algorithms", Transactions of the AMS, Vol. 117, 1965, pp. 285-306.

[15] Hopcroft, J.E. and J.D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, 1969, p. 96.

[16] Ianov, Iu. I., "The Logical Schemes of Algorithms", Problems of Cybernetics (USSR) Vol. 1, 1960, pp. 82-140.

[17] Karp, R.M. "Reducibilities Among Combinatorial Problems", in R. Miller and J. Thatcher (eds.), Complexity of Computer Computations, Plenum, 1972, pp. 85-104.

[18] Kuroda, S.Y. "Classes of Languages and Linearly-Bounded Automata", Information and Control, Vol. 2, 1964, pp. 207-223.

[19] Landweber, L.H. and E.L. Robertson "Recursive Properties of Abstract Complexity Classes", JACM, Vol. 19, No. 2, 1972, pp. 296-308.

[20] Lewis, F.D. "The Enumerability and Invariance of Complexity Classes", JCSS, Vol. 5, No. 3, 1971, pp. 286-303.

[21] Luckham, D.C., D.M.R. Park and M.S. Paterson "On Formalised Computer Programs", JCSS, Vol.4, 1970, pp. 220-249.

[22] McCarthy, J. et al, The LISP 1.5 Programmer's Manual, MIT Press, 1968.

[23] Mehlhorn, K. "On the Size of Sets of Computable Functions", Technical Report 73-164, Department of Computer Science, Cornell University, 1973.

[24] Paterson, "Equivalence Problems in a Model of Computation", Doctoral Dissertation, Cambridge University, 1967.

[25] Rogers, H., Jr., Theory of Recursive Functions and Effective Computability, McGraw-Hill, 1967.

[26] Simon, J.: personal communication.

[27] Weihrauch, K.: personal communication.