

21

EXTENDING AND REFINING HIERARCHIES
OF COMPUTABLE FUNCTIONS

by

Robert L. Constable

Computer Sciences Technical Report #25

June 1968

PREFACE

In part I we extend the Kleene hierarchy of primitive recursive functions into the transfinitely recursive functions. The extended hierarchy, E_α , is indexed by the ordinals $\alpha < \epsilon_0$. For \mathcal{R}^n Peter's n-fold recursive functions, $E_{\omega^n} = \mathcal{R}^n$. Emphasis is placed on finding a hierarchy which can be interpreted in terms of computing theoretic concepts.

In part II we examine the low end of the hierarchy using a new type of machine especially suited for such investigations. The machine is called a rewind automaton. We examine computational complexity classes based on the number of rewinds required to compute a function.

ACKNOWLEDGEMENTS

I would like to thank Professors Stephen C. Kleene and Larry Travis for their part in this work. Professor Kleene has seen to it that I have had the time for the research and has given of his valuable time and advice to help me along. Professor Travis has listened to me, guided me, taught me and counseled me for three years on this and other projects. His criticisms and suggestions have done much to shape this work.

Finally I give hearty thanks to my wife, Carol, for her prodigious efforts in typing the manuscript and for her many sacrifices while the dissertation was being written.

NOTATION

A function is denoted by $f()$ when the arguments are not essential. When they are it is denoted by $\lambda x_1 \dots x_n f(x_1, \dots, x_n)$.

The non-negative integers are denoted by N , and elements of N^n , called vectors, are usually denoted by X . The n is usually specified by writing $X \in N^n$.

The letter π will always denote a program and π_f a program which computes the function $f()$.

For sets, write $A \subset B$ iff $A \subseteq B$ and $A \neq B$.

\mathcal{R} denotes the set of (total) recursive functions, and \mathcal{R}' denotes the partial recursive functions. \mathcal{F} denotes the set of all number theoretic functions. \mathcal{E} denotes the set of elementary functions.

For K a class of functions, $K^{[n]}$ denotes the n argument members of the class. Also write $f(X) < K$ iff $\exists b() \in K^{[n]}$ such that $f(X) < b(X) \quad \forall X \in N^n$.

TABLE OF CONTENTS

PREFACE

ACKNOWLEDGEMENTS

NOTATION

PART I

Chapter 1	Introduction	1
Chapter 2	Random Access Stored Program Machines	16
Chapter 3	Grzegorzczuk, Ritchie and Cleave Hierarchies	30
Chapter 4	The Ritchie-Cleave Hierarchy	49
Chapter 5	A New Ritchie-Cleave Hierarchy	63
Chapter 6	The Extended Ritchie-Cleave Hierarchy	74
Chapter 7	Comparison of Hierarchies	105
Chapter 8	Algorithmic Complexity	114

PART II

Chapter 9	Finite Automata	122
Chapter 10	Rewind Automata	133
Chapter 11	Low Level Complexity Classes	146

APPENDIX

A	A Computing Procedure for $f_{\alpha}(\)$, $\alpha < \epsilon_0$	169
B	Estimating $of_{\alpha}(\)$	190
C	Arithmetizing the RASP	206

BIBLIOGRAPHY	213
--------------	-----

Chapter 1 Introduction

At present works on recursive function complexity fall into three major categories;

(A1) works by logicians on subrecursive hierarchies where extensive use is made of constructive ordinals, recursive function theory and the theory of recursive functionals,

(A2) work by logicians on sequences of formal theories and the functions provably recursive in them where extensive use is made of the constructive ordinals, various formalizations of parts of number theory and analysis, formalized transfinite induction, and Gödel's theorems on consistency and incompleteness,

(B) work by automata theorists on computational complexity where extensive use is made of automata, Turing machines and other mathematical machines such as iterative arrays, push-down store machines and generalized sequential machines,

(C) work by automata theorists on algorithmic complexity where extensive use is made of finite automata theory, semi-group theory and combinatorial analysis.

Work in category A is by far the oldest, dating back to Hilbert in 1925, while efforts in area B began in 1960 and have been flourishing vigorously ever since. Category C has its origins in 1956 but did not become directed sys-

tematically and explicitly to complexity problems until after 1963.

These categories have been for the most part unrelated, especially areas A and B with C. We are concerned with presenting all three areas from a unified viewpoint.

There are strong reasons for introducing a unified viewpoint at this time. First, once this viewpoint is stated, many implicit connections between the areas become apparent, and numerous interesting questions arise. Also each of these areas helps explain the phenomena of the other areas more deeply.

The viewpoint we adopt is computing theoretic. Rapid developments in the field of computers are causing the swift emergence of computer science. This new science is concerned with such activity as computer design, design and analysis of programming languages and programming systems, analysis of computations and algorithms, and construction of extremely complex programs. Study of these activities is leading to an abstract computer science already embracing such precise new notions as abstract machines and formal grammars. The concept of recursive function complexity seems destined to play a crucial role in the development of this abstract computer science, namely as part of a theory of computing.¹

¹We emphasize the use of functions as central to the theory of computing rather than that of recognition problem or set as is customarily done in areas B and C.

The precise goals of a computing theory are being formed from real computing experience, but there already exist vague yet clearly interesting questions for such a theory. Among them are these.

Q1. Can we discover a class of functions which represents the functions which are "actually or practically computable" more faithfully than \mathcal{R} does? (For example, \mathcal{R}^1 is a candidate for such a class.)

Q2. Can we discover intrinsic computational and algorithmic properties of functions? (For example, is multiplication intrinsically harder to perform than addition? Are the multiplication algorithms always more difficult than those for addition? Is there a natural computational hierarchy of recursive functions?)

Q3. Can we define a quantitative notion of the size of a computation perhaps analogous to Shannon's notion of information?

Q4. Can we discover trade off relationships to balance the computational parameters of programs such as time and memory, time and reliability and so forth?

Q5. What relationships exist between structural properties of programs and properties of their computations? (For example, are there functions for which all short programs are inefficient?)

We will be able to state these and other questions more precisely as technical concepts accumulate. We will remain

concerned to some extent with Q1 - Q5 throughout our work. But the results of the first part of the thesis are better motivated by more precise considerations from mathematical logic and recursive function theory. These considerations occur among the branches of logic and recursion theory, which are least blessed with notational simplicity, elegant proofs, deep philosophical content and popularity. To mitigate against these characteristics we offer in Chapter 3 an extensive introduction to the technical work. In the same spirit we continue this introduction with some historical background on area A.

The study of subrecursive hierarchies has its antecedents in the 1888 work of Dedekind. The notion of a primitive recursive function appeared in his Was sind und was sollen die Zahlen. The schema of primitive recursion, though not so named, was presented as a natural generalization of the schemata for recursive definitions of addition and multiplication. A modern version of that schema is R1 below.

$$\begin{aligned} \text{R1. (a)} \quad & f(X, 0) = g(X) \quad X \in N^n \\ \quad \quad \quad & \text{(b)} \quad f(X, n+1) = h(X, n, f(X, n)) \end{aligned}$$

Here $f()$ is defined from known functions $g()$ and $h()$ by primitive recursion. The primitive recursive functions were defined to be those obtained from 0, the successor function, and the projection functions ($U_i^n(x_1, \dots, x_n) = x_i$ for $i \leq n$), by composition and the schema of primitive recursion. We denote this class by \mathcal{R}^1 .

In 1925 the schema of primitive recursion found itself in the limelight of a sweeping proposal by one of the century's most eminent mathematicians.

Hilbert in "Über das Unendliche" outlined his proposal for solving the continuum problem, the problem he placed first on his famous list of unsolved mathematical problems.¹ He proposed a hierarchy of number theoretic functions \mathcal{F}_α such that

- (1) \mathcal{F}_α is countable for all $\alpha < \omega_1$ (actual) and
- (2) $\mathcal{F}_\alpha \subset \mathcal{F}_\beta$ if $\alpha < \beta < \omega_1$ and
- (3) $\bigcup_{\alpha < \omega_1} \mathcal{F}_\alpha = \mathcal{F} =$ all number theoretic functions .

Such a hierarchy would allow associating a unique countable ordinal with each $f() \in \mathcal{F}$ thus proving the continuum hypothesis. Further Hilbert claimed that the processes of substitution and recursion (transfinite included) were sufficient to obtain any function of \mathcal{F} . The claim was based on his proposed formalization of mathematics and his belief that all the recursions required could be subsumed under the ordinary schema R1 if functional variables were allowed. Thus a large part of the continuum problem was intuitively reduced to examining primitive recursions with respect to functionals.

Essential to Hilbert's construction was the notion of

¹In 1963 P. Cohen proved that the Generalized Continuum Hypothesis was independent of the usual axioms of set theory, ZF + C for example.

"type of a function". A function is of type 1 if it is a number theoretic function, it is of type 2 if it is a function of functions, i.e. assigns a number to a function. A function of type 3 assigns numbers to functions of functions and so on. A function is of limit type α if it is a function which assigns numbers to sequences of functions f_n where f_n is of type α_n for α_n a fundamental sequence to α .

Hilbert could now define a hierarchy of number theoretic functions using 0, the successor function, substitution, and the schema R1 with functional variables. A function is said to be of degree α if it can be defined using functionals of type α but can not be defined with functionals of lower type.

For Hilbert's plan to succeed the hierarchy must be proper, i.e. there must be functions of degree α which are not of degree $\beta < \alpha$ for any β . Ackermann produced an example in 1925 (published 1928) showing that at least for degrees 1 and 2 the hierarchy was proper. He indicated that his method would work to provide examples for all degrees $n < \omega$.

Hilbert's bold plan was fraught with difficulties, and although it received little attention, the discovery by Ackermann which it spawned stimulated further activity in the classification of recursive functions. Ackermann had exhibited a function $a(x, y, n)$ which was of 2nd degree but not of 1st. It also satisfied the following recursive equations

$$(a) \quad a(x, y, 0) = x + y$$

$$(b) \quad a(x, 0, n+1) = g(x, n)$$

$$(c) \quad a(x, y+1, n+1) = a(x, a(x, y, n+1), n)$$

$$\text{where } g(x, n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ x & \text{if } n>1. \end{cases}$$

These equations uniquely define a recursive function which is not primitive recursive yet the equations do not involve functionals. Also they do not follow schema R1 since the recursion proceeds on two variables simultaneously. Ackermann's result implies that there is no way to reduce these equations to the form R1 without introducing functional variables.

In the late 1930's R. Péter studied recursion schemata modeled after these equations. Such schemata were called multiply recursive or n -fold recursive since the recursion proceeds on more than one variable simultaneously. She called an n -fold recursion "nested" if the values of the function itself occurred for the recursion variables, e.g. as in Ackermann's function. She showed that nested $(n+1)$ -fold recursions could not be reduced to n -fold recursions. But unnested n -fold recursions could be reduced all the way down to 1-fold (primitive) recursion. Classes of functions defined similarly to \mathcal{R}^1 can be based on the notion of n -fold recursion. Let \mathcal{R}^n be the class of functions defined from 0, the projection function $(U_1^n(\))$, and the successor function by composition and nested n -fold recursion. Péter

proved $\mathcal{R}^n \subset \mathcal{R}^{n+1}$ $n < \omega$. We call this the Péter subrecursive hierarchy. Péter further demonstrated that her \mathcal{R}^n were precisely the \mathcal{F}_n of Hilbert for $n < \omega$.¹

In 1953 Grzegorzczuk constructed another hierarchy based on ideas in Ackermann's work. He used a sequence of functions $g_n(x, y)$ similar to Ackermann's $a(x, y)$ to stratify \mathcal{R}^1 into an ω -sequence of classes \mathcal{E}^n where \mathcal{E}^n contains $g_n(\)$ and is closed under certain elementary operations. We discuss this hierarchy fully in Chapter 3.

In 1940 Ackermann introduced a notion of transfinite recursion, and in 1950 Péter was able to show that her n -fold recursive functions were precisely the ω^n -transfinite recursive functions with respect to standard well orderings of order type ω^m .

The notion of ordinal recursion seemed like the ideal means of studying hierarchies of recursive functions. To go beyond Péter's \mathcal{R}^n into \mathcal{R}^α for $\alpha \leq \omega$ might require only an α -recursive ordinal schema. The recursive functions could then be classified according to the ordinal level at which they could first be defined. Furthermore since Hilbert's hierarchy also extended into the transfinite the possibility existed of comparing an ordinal recursion hierarchy and the functional hierarchy. An ideal situation might have been that \mathcal{R} is obtained at level ω_1 (constructive) in

¹See Lacklan[31] for a modern, thorough account of \mathcal{R}^n .

both hierarchies, i.e. $\mathcal{R}^{\omega_1} = \mathcal{R}_{\omega_1}$ and that the hierarchies \mathcal{R}^α and \mathcal{R}_α continue through \mathcal{F} for higher type recursion schemas and higher level functionals as envisioned by Hilbert. However, this plan ran into a "stumbling block" discovered by Routledge and Myhill.

The problem begins with the fact that to define the ordinal recursion hierarchy, the ordinals must be represented in the integers. Suppose then for the ordinal α , $<_\alpha$ is a well-ordering on $\mathbb{N} \times \mathbb{N}$ of order type α . Say that $f(\)$ is defined from $g(\)$ and $h(\)$ by unnested ordinal recursion of order type α iff

- (a) $f(X, 0) = g(X)$
- (b) $f(X, n) = h(X, n, f(X, t(X, n)))$
- (c) $t(X, n) <_\alpha n$.

What Routledge and Myhill discovered is that for every recursive function $g(\)$, there is a recursive well ordering of type ω , $<_{\omega, g}$ such that $g(\)$ can be defined by ω -recursion using $<_{\omega, g}$. (In fact $<_{\omega, g}$ can be elementary.) Thus all recursive functions turn out to be ordinal recursive of order type ω . The reason Péter's hierarchy escapes this calamity is that she uses only certain standard well orderings of types ω^n . Therefore in order to erect a proper ordinal recursive hierarchy, the notion of a standard well ordering must be clarified in general. This remains an open problem which has attracted many researchers yet has

rewarded none of them significantly.¹

In an attempt to circumvent this stumbling block, Kleene in 1958 proposed a new way of connecting ordinals with recursive functions. Of the two basic methods for extending classes of functions, diagonalization and majorization, Kleene chose diagonalization. First he fixed a standard enumeration procedure for the class of functions to be extended based on the fact that the class of functions from which the hierarchy begins is effectively generated by the application of certain operations to a finite set of initial functions. Thus given the class C , its single argument functions are enumerated in a standard manner by $e(x, y)$. Then $e()$ is adjoined to C as a new initial function to form a class C' . As a specific example, Kleene took C to be the class \mathcal{R}^1 .

The above method provides the mechanism for stepping from a class indexed by α to that indexed by $\alpha+1$. Now given classes indexed by $\alpha_0, \alpha_1, \alpha_2, \dots$ in a fundamental sequence to α , i.e. $\alpha = \lim \alpha_n$, the enumerating function $e_{\alpha_n}()$ can be combined into a new enumerating function $e_\alpha()$.

Kleene pointed out that this combining of enumerators must be done with care lest the stumbling block materialize here. Control of the enumerations is given by the Church-Kleene system of constructive ordinals. Recalling that in

¹See Tait[56] for recent work on this problem.

this system if the limit ordinal α is represented by the integer $\bar{\alpha} = 3 \cdot 5^{\bar{f}}$, for \bar{f} the Gödel number of $f(\)$, then the fundamental sequence for α is represented by the integers $f(n)$ for $n = 0, 1, 2, \dots$. The new initial function for $C_{\bar{\alpha}}$ is given by

$$e_{\bar{\alpha}}(x, y) = e_f((x)_1)((x)_0, y)$$

where the $(x)_1$ are the usual "exponent of i -th prime functions" (see A.20 here).

For a hierarchy beginning with the class \mathcal{R}^1 it was found necessary to restrict the ordinal notations so that fundamental sequences, e.g. $f(\)$ above, were only primitive recursive. It was shown by a student of Kleene's Paul Axt[1] that this subrecursive hierarchy was proper and was independent of ordinal notations for $\alpha < \omega^2$. He also showed that \mathcal{R}^n was contained in the Kleene subrecursive hierarchy below the level ω^{n-1} . In later work, [2], Axt showed that the Grzegorzczuk hierarchy for $n > 3$ could be obtained by the Kleene procedure starting with the class \mathcal{E}^4 .

Although Axt showed that the Kleene subrecursive hierarchy was proper and unique for $\alpha < \omega^2$, he discovered non-uniqueness at ω^2 . Feferman[12], in 1961 completed this negative result by showing that, alas, the hierarchy collapsed at ω^2 . That is for every recursive function $f(\)$ there is an ordinal notation W_f for ω^2 such that $f(\) \in C_{W_f}^{\omega^2}$.¹

¹In 1965 R. Fabian[11] attempted to stretch the Kleene hier-

Recently J. Robbin[44] building on work by W. W. Tait[56] has shown what would happen if the problem of standard notations should be solved. By limiting himself to certain ordinal notations which logicians believe to be "standard", he has compared the Kleene hierarchy, the ordinal recursion hierarchy, the Péter hierarchy and the extended Grzegorczyk hierarchy, which he constructed and which will be discussed in detail later. It turns out that

$$\mathcal{R}^n = \bigcup_{\alpha < \omega^{n-1}} C_\alpha = \bigcup_{\alpha < \omega^n} \mathcal{E}^\alpha = O(<_{\omega^{n-1}})$$

for C_α the Kleene hierarchy classes ($C_0 = \mathcal{R}^1$), \mathcal{E}^α the extended Grzegorczyk hierarchy classes and $O(<_\alpha)$ the functions ordinal recursive with respect to standard well orderings $<_\alpha$.

If our results are viewed in this context, then what we have done is produce a fifth hierarchy, the Extended Ritchie-Cleave hierarchy, E_α , which extends beyond the others up to ϵ_0 and relates to them by $\mathcal{R}^n = E_{\omega^n}$. Control of the hierarchy is basically in terms of computing theoretic concepts.

Thus after 43 years, the ideas in Hilbert's On the Infinite seem to be trapped. The method of functionals which

archy further by beginning lower down, with \mathcal{E} the elementary functions, and by further restricting the ordinal notation using elementary fundamental sequences. But all that is known is that Feferman's proof will not collapse the revised hierarchy.

he proposed has not developed beyond Péter's work. The multiple recursions which supercede them are limited by their syntactic dependancy to a hierarchy of type ω . The transfer of the burden to transfinite recursions runs into the stumbling block of standard notations for recursive ordinals. The attempt by Kleene to skirt this problem led to another form of it, this time appearing in his system of constructive ordinals. What remains is a whole crop of short hierarchies all closely related, each of which can probably be extended up to any well-understood ordinal $\alpha < \omega_1$ (constructive) by some tedious ad hoc method. Recent results by Moschovakis[36], Feferman and Spector[13] shed some light on this discouraging situation.

Suppose \mathcal{N} is a formalism for the partial recursive functions, \mathcal{R}' , and $g(\)$ is an arithmetization of \mathcal{N} . Let A be the set of integers corresponding to total functions, \mathcal{R} , under $g(\)$. Suppose further that A is hyperarithmetical.¹ Then for $a \in \mathcal{O}$ letting $|a|$ denote ordinal represented by a , there is no classification of \mathcal{R} into subsets \mathcal{R}_a such that

$$(a) \bigcup_{a \in \mathcal{O}} \mathcal{R}_a = \mathcal{R}$$

$$(b) \bigcup_{|a| < \beta} \mathcal{R}_a \subset \mathcal{R} \quad \text{if } \beta < \omega_1$$

$$(c) \text{ the predicate " } a \in \mathcal{O} \text{ and } x \in A_a \text{ " is } \Pi^1_1.$$

If \mathcal{O} is replaced by a path P through \mathcal{O} , then there is still no classification satisfying (a) - (c) with P for \mathcal{O} .

¹We know in fact that A is in Σ^0_3 but not in Π^0_3 by Shoenfield [51].

The hierarchies considered above all violate condition (b) (Kleene, ordinal recursion) or condition (a) (Péter, Extended Grzegorczyk and Extended Ritchie-Cleave). Furthermore these hierarchies will not violate condition (c) under any reasonable extensions. Thus it appears that a kind of incompleteness phenomenon arises for subrecursive hierarchies which pretend to constructivity. This situation would apparently also preclude a solution to the problem of standard well orderings which would apply to the above hierarchies.

In the face of this phenomenon, one can now expect the following possibilities for subrecursive hierarchies:

1. There is some ordinal $\beta < \omega_1$ which is a "natural" closure ordinal for \mathcal{R} i.e. such that the subrecursive hierarchies are proper and satisfy (a) - (c) above for $\alpha < \beta$.

2. For every $\alpha < \omega_1$ there is a proper subrecursive hierarchy up to α which does not reach \mathcal{R} but which may possess other interesting properties such as relating nicely to other subrecursive hierarchies up to α (as for $\beta < \omega^\omega$ above), or extending the hierarchies for $\beta < \alpha$ in a natural manner or relating to functions definable in certain formal theories.

3. There is some subrecursive hierarchy which is proper, satisfies (a) and (b), fails at (c) (from above) but fails in an interesting manner, e.g. perhaps preserving constructivity for certain "small" ordinals.

Since the compelling hope of finding a proper subrecursive hierarchy satisfying (a) - (c) is thwarted, questions about the relationship among hierarchies and about their naturalness or their relationship to formal theories (e.g. via Z -provable recursion for axiomatic theory Z) or their applicability (e.g. to classifying decision problems) or their interpretation in terms of a theory of computing or a theory of constructivity become more attractive.

Therefore we can propose that the work we do here is also in pursuit of this brand of questions. Moreover, we claim that from the prospect of studying and managing very complex computations, a task often required in pursuing these questions, our development and use of the RASP computing system is advantageous. We turn next to a detailed consideration of the RASP machine.

Chapter 2 Random Access Stored Program Machines

A particular brand of RASP machine (Random Access Stored Program) is used to obtain the hierarchy results. The RASP class of machines was developed by Elgot & Robinson[45]. Such machines offer several advantages. They can easily simulate other types of machines, such as Turing machines, or register machines. They are among the best available abstract models of "actual" digital computers. Their organization allows for easy expression of complex computations.

2.1 A RASP is an ordered sextuple $\langle A, B, b_0, K, F_1(), F_2() \rangle$ where

- (a) A is a set called the address set
- (b) B is a set called the set of words
- (c) $b_0 \in B$ and is called the empty word
- (d) $K \subseteq B^A = \{f() \mid f() : A \rightarrow B\}$
 $k() \in B^A$ is called a content function and K is called the set of memory configurations.
- (e) $F_1() : K \times A \times B \rightarrow B^A$, F_1 is called the next content functional
- (f) $F_2() : K \times A \times B \rightarrow A$, F_2 is called the next control functional

Let $\Sigma = K \times A$, the pairs $\langle k(), a \rangle$ are called the states of the RASP. The functional $F(k(), x, y) = \langle F_1(), F_2() \rangle$ then maps states and content into states, i.e. $F() : \Sigma \times B \rightarrow \Sigma$. If $\sigma = \langle k(), a \rangle$ then $k()$ is content at σ and a is control

location at σ .

The intuitive interpretation is that A is a set of addresses of memory locations (registers) in the machine; each register can hold a word, i.e. element of B . (b_0 corresponds to a blank.) The function $k(\)$ gives an assignment of words to locations, thus $k(a)$ is the word stored in location a . Some words can be regarded as instructions to the machine. The location of the word controlling the machine when it is in state $\sigma = \langle k(\), a \rangle$ is given by a . The functional $F_1(k(\), x, y)$ takes the content function $k(\)$ and alters it depending on what that content is, on what the control location is and depending on some word, usually the word in $k(x)$, for x the control location. $F_2(\)$ determines a next control location depending on the memory, a control location and a word. These functionals contain the heart of the machine. They correspond roughly to transition tables for a Turing machine and circuitry of a digital computer. These functionals give "operational meaning" to the words of B .

2.2 We now list some of the more important properties of RASP machines. If the subset K_f of B^A has finite support i.e. $k(\) \in K_f \iff k(x) = b_0$ a.e. (on all but a finite subset of A), then we say that the RASP is finitely supported.

A sequence of states, $\sigma_i = \langle k_i(\), a_i \rangle \in \Sigma$, $i = 0, 1, 2, \dots$ is called a computation of the RASP $\langle A, B, b_0, K, F_1, F_2 \rangle$ iff

$\sigma_{i+1} = F(\sigma_i, k_i(a, i)) \quad \forall i$. If $E = \{e_1, \dots, e_n\} \subset A$ and if $\text{comp}(\sigma_0) = \sigma_0, \sigma_1, \sigma_2, \sigma_3, \dots$ then define $\text{comp}_E(\sigma_0) = \text{comp}(\sigma_0)$ if $a_i \notin E$ and $\text{comp}_E(\sigma_0) = \sigma_0, \dots, \sigma_n$ if $a_n \in E$.

Given a fixed $b \in B$, $F(k(\), x, b)$ defines a mapping from Σ into Σ . Putting $H_b(k(\), x) = F(k(\), x, b)$ we call $H_b(\)$ an atomic instruction of the RASP $\langle A, B, b_0, K, F_1, F_2 \rangle$. Any map (functional) $H: \Sigma \rightarrow \Sigma$ is called an instruction. A word $b \in B$ will be called active iff $\exists \sigma \in \Sigma$ such that $H_b(\sigma) \neq \sigma$ otherwise the word is called passive.

An instruction $H_b(\)$ will be called finitely determined iff

- (a) $\forall a \exists n \exists x_1, \dots, x_n$ such that $F_1(k(\), a, b) = F_1(k'(\), a, b)$ if $k(x_i) = k'(x_i) \quad i = 1, 2, 3, \dots, n$.
- (b) $\forall a \exists p \exists x_1, \dots, x_p$ if $F_2(k(\), a, b) = k'(\)$, then $k(y) = k'(y)$ if $y \neq x_i \quad i = 1, 2, 3, \dots, p$.

Condition (a) is just the usual statement of finite determination for $F(\)$ a functional with individuals as values; condition (b) covers the case of functions as values.

If $H_b(\)$ is finitely determined for every $b \in B$, then we say that the RASP is finitely determined.

2.3 An instruction schema for a RASP $P = \langle A, B, b_0, K, F_1, F_2 \rangle$ is a set of mappings from $A^q \times B^m$ into Σ^Σ , the set of instructions or equivalently it is a map from $\Sigma \times A^q \times B^m$ into Σ . We designate an instruction schema by $\rho(\sigma, x_1, \dots, x_q, y_1, \dots, y_m)$. Fixing the parameters x_i, y_i produces a member of Σ^Σ which may or may not be an atomic instruction, H_b .

To make the proper connection with atomic instructions we

define a designated instruction schema as a pair $\langle \rho, \mu \rangle$ where $\rho()$ is an instruction schema and μ is a 1-1 map from $A^q \times B^m$ into $B_{\langle q, m \rangle} \subseteq B$ such that $\rho(\sigma, a_1, \dots, a_q, b_1, \dots, b_m) = \mu(a_1, \dots, a_q, b_1, \dots, b_m)^H(\sigma)$.

The mapping μ is just a way of assigning computer words to instruction schemata, that is a way of representing the schema internally.

A RASP P will be said to be generated by the set of designated instruction schemata $R = \{s_i\}$ where $s_i = \langle \rho_i, \mu_i \rangle$, with parameters $y_1, \dots, y_{q_i}, z_1, \dots, z_{m_i}, q_i \geq 0, m_i \geq 0$, if exactly the active words of P belong to the union of all B_{s_i} where B_{s_i} is the range of μ_i .

2.4 We next consider the important notion of a program for a RASP. Intuitively a program is a finite sequence of instruction words stored in memory in machine language. The address of the instruction at which the program begins is called the entrance to the program and the addresses to which the terminal instructions send control are exits from the program. We also include in the program certain constants needed in instructions (e.g. $x+1$ requires a "1" or transfer "a" to ___ requires an a). Thus a fairly general formal definition of a program π is that $\pi = \langle p, a_0, e_0, \dots, e_m \rangle$ where p maps a finite subset of A , denoted D_π , into $\sum xB$ such that

$$(a) \quad a_0 \in D_\pi$$

- (b) $p(a_0) \in \Sigma^\Sigma$
- (c) $i \neq j \implies e_i \neq e_j$
- (d) $e_i \in A-D_\pi \quad i = 0, 1, \dots, m.$

The set D_π is called the domain of π . a_0 is called the entrance to π . e_i are called the exits of π . If $p(a) \in \Sigma^\Sigma$, then $p(a)$ is an instruction of π ; and if $p(a) \in B$, then $p(a)$ is called a parameter of π (pr constant of π). We say that $k(\)$ holds π iff

- (a) $\forall a \ a \in p^{-1}(\Sigma^\Sigma), H_{k(a)}(\) = p(a)$ and
- (b) $\forall a \ a \in p^{-1}(B), k(a) = p(a).$

If $k(\)$ holds $\pi = \langle p, a_0, e_0, \dots, e_m \rangle$, then $\text{comp}_E(k(\), a_0) = \langle k_1(\), a_1 \rangle$ is called a comp of π where $E = \{e_0, \dots, e_m\}$. A computation terminates if $a_i \in E$ and does not terminate otherwise.

2.5 Let $f(\): B^n \longrightarrow B^m$, we define the notion of a program π for a RASP, P , computing $f(\)$. First let x_1, \dots, x_n and y_1, \dots, y_m be addresses. A program $\pi = \langle p, a_0, e \rangle$ is said to compute f at x_i, y_j $i = 1, \dots, n, j = 1, \dots, m$ iff

- (a) $x_i \notin D_\pi$
- (b) If $k(\)$ holds π and $k(x_i) = c_i$, then letting $\sigma_1 = \langle k_1(\), a_1 \rangle \quad i = 0, 1, 2, \dots$, and $C = \langle c_1, \dots, c_n \rangle \in B^n$,
 - (i) if $f(C) = C'$ then $\text{comp}_e(\sigma_0)$ terminates for some n with $k_n(y_1) = c'_1, C' = \langle c'_1, \dots, c'_n \rangle \in B^m$,
 - (ii) if $f(C)$ is not defined, then $\text{comp}_e(\sigma_0)$ does not terminate.

Notice that if both $k_1(\)$ and $k_2(\)$ hold π and agree on x_i

and $\text{comp}_e(k_1(\), a)$ terminates but $\text{comp}_e(k_2(\), a)$ does not, then π does not compute any function at x_i and any value locations y_i .

2.6 By the range of influence of a program $\pi = \langle p, a_0, e_0, \dots, e_m \rangle$, $RI(\pi)$, we mean a set of addresses, $RI(\pi) \subseteq A$, such that $d \in RI(\pi)$ iff $\exists k(\)$ such that $k(\)$ holds π and $\text{comp}_E(k(\), a_0)$ terminates in $(k_n(\), a_n)$ with $k_n(d) \neq k(d)$ where $E = \{e_0, \dots, e_m\}$. In other words, the range of influence of π is the set of registers which do not have the same content at the end of the computation as they did at the beginning, no matter what happened in between. By range of action of program π in core $k(\)$, $RA(\pi, k(\))$, we mean that $RA(\pi, k(\)) \subseteq A$, such that $s \in RA(\pi, k(\))$, iff $\text{comp}_E(k(\), a_0) = \sigma_1, \dots, \sigma_n$ and $k(\)$ holds π , and $k_i(s) \neq k_j(s)$ for some $i, j \leq n$.

2.7 To say that a RASP P can compute a function, $f: B^n \rightarrow B^m$, means intuitively that there is a program π and addresses x_i, y_i such that π computes f at x_i, y_i . However we also want to express the notion that this program can be loaded anywhere in P and can be loaded together with other programs, that is by running one program π , some memory of the machine is not altered. To say this precisely we pick any finite subset A_1 of A and ask that D_π be outside of A_1 (thus π can be loaded anywhere) and we pick a subset $A_0 \subseteq A_1$ and ask that $RI(\pi)$ is disjoint from A_0 , thus execution of π does not alter A_0 . The formal definition follows.

2.8 A RASP P can compute a function $f: B^n \rightarrow B^m$ iff for every sequence $x_1, \dots, x_n, y_1, \dots, y_m, a_0$ of distinct elements of A and for all finite subsets A_0, A_1 of A such that $a_0 \notin A_1$ and $A_0 \subseteq A_1$, there exists a program $\pi = \langle p, a_0, e \rangle$ satisfying:

- (a) π computes f at $x_1, \dots, x_n, y_1, \dots, y_m$
- (b) D_π is disjoint from $A_1 \cup \{x_1, \dots, x_n, y_1, \dots, y_m\}$
- (c) if A_0 is disjoint from y_1, \dots, y_m , then $RI(\pi)$ is disjoint from A_1 .

2.9 Given an m -valued relation, $R(x_1, \dots, x_n)$, defined on a subset of B (a 2 valued relation is called a predicate), a program $\pi = \langle p, a_0, e_1, \dots, e_m \rangle$ is said to compute $R()$ at data locations d_1, \dots, d_n provided

- (a) D_π is disjoint from d_1, \dots, d_n
- (b) If $k()$ holds π , $k(d_i) = x_i$ and $\sigma_i = \langle k_i(), a_i \rangle$ $i = 0, 1, \dots$, then
 - (i) if $R(x_1, \dots, x_n)$ is defined and has value $l \leq r \leq m$, then $\text{comp}_E(\sigma_0)$ for $E = \{e_1, \dots, e_m\}$ terminates in e_r .
 - (ii) if $R(x_1, \dots, x_n)$ is undefined, then $\text{comp}_E(\sigma_0)$ does not terminate.

A RASP P computes an m -valued relation $R(x_1, \dots, x_n)$ iff there is a sequence $x_1, \dots, x_n, a_0, e_1, \dots, e_m$ of distinct elements of A and for all finite subsets A_1 of A such that $a_0 \notin A_1$ there exists a program $\pi = \langle p, a_0, e_1, \dots, e_m \rangle$ satisfying:

- (a) π computes $R()$ at d_1, \dots, d_n
- (b) $D_\pi \cap (A_1 \cup \{d_1, \dots, d_n\}) = \emptyset$
- (c) $RI(\pi) \cap A_1 = \emptyset$

2.10 A program π is called fixed if whenever $\langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$ is a terminating computation of π , then $k_1(), k_2(), \dots, k_n()$ all agree on D_π where $\sigma_i = \langle k_i(), a_i \rangle$. A program π is called self-restoring iff whenever $\langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$ is a terminating computation of π , then $k_1()$ and $k_n()$ agree on D_π . Equivalently π is self-restoring iff $D_\pi \cap RI(\pi) = \emptyset$.

2.11 It is a fairly easy exercise to show that the following facts hold (see Elgot & Robinson[45]). Let P be a RASP

- (a) If P computes f_1 and f_2 then P computes the composition of f_1 and f_2 .
- (b) If P computes the predicates R_1 and R_2 then it computes the predicates $R_1 \& R_2$, $R_1 \vee R_2$ and $\neg R_1$.
- (c) Let $ch_R()$ be the characteristic function of R , then

- (i) If P computes the identity function $i: B \rightarrow B$, and P computes predicate R , then P computes ch_R .
- (ii) If P computes $eq()$ predicate and the function $ch_R()$, then P computes R .

2.13 We are now interested in defining more specific RASP's to be used as basic machines in the work below. These machines will be able to compute all recursive functions and will employ many features of real digital computers. For

$M = \langle A, B, b_0, K, F_1, F_2 \rangle$ we will take

(a) $A = B = N = 0, 1, 2, \dots$

(b) $b_0 = 0$

(c) $K = (N^N)_f$, the set of functions $N \rightarrow N$ of finite support.

The functionals F_1, F_2 will be generated by certain designated instruction schemata. First consider the following informal instructions.

2.13 Let a be the control location at the time the instruction is reached and let a' be the next control location.

Symbols for mapping $N^3 \rightarrow$	Informal description of state transition mapping.
(1) $\text{ADD}(x, y, z); a' = a+1$	add $k(x)$ to $k(y)$ and store result in z , go to $a+1$ for next instruction
(2) $\text{SUB}(x, y, z); a' = a+1$	subtract $k(y)$ from $k(x)$ and store the result in z , go to $a+1$ for next instruction
(3) $\text{MULT}(x, y, z); a' = a+1$	multiply $k(x)$ by $k(y)$ and store result in z , go to $a+1$ for next instruction
(4) $\text{T}(x, y); a' = a+1$	transfer $k(x)$ to y , go to $a+1$ for next instruction
(5) $\text{C}(x, y, z); a' = a+1$	if $k(x) = k(y)$, then go to z for next instruction, otherwise go to $a+1$

We will abbreviate this set of instructions by $\{+, -, x, T, C\}$ or by Σ_0 . They each consist of two components, change of content component and change of control component. The change of content components of ADD, SUB, and MULT are

associated with certain functions $N^2 \rightarrow N$ in a natural manner. Conversely given any function $f: N^n \rightarrow N$, an instruction $F(x_1, \dots, x_n, y)$ can be defined which puts $f(k(x_1), \dots, k(x_n))$ into y and changes control according to some function $c(\)$ applied to a . Such an instruction will be called an arithmetic instruction, it will be denoted $\rho_f(x_1, \dots, x_n, y)$.

2.14 We will now provide a designation, μ , for the instructions above. We indicate how this could also be done for any set of recursive arithmetic instructions. Providing a designation can be thought of as defining a machine language (simultaneously with construction of the machine), the language is the set of words in the range of μ , i.e. the instruction names.

First consider the set $\Sigma_0 = \{+, -, x, T, C\}$. Define operation codes for the instructions by letting $\overline{ADD} = 2$, $\overline{SUB} = 3$, $\overline{MULT} = 5$, $\overline{T} = 7$, $\overline{C} = 11$. Then define $\#(X, x, y, z,) = 2^{\overline{X}} \cdot 3^x \cdot 5^y \cdot 7^z$ for $X = \overline{ADD}, \overline{SUB}, \overline{MULT}, \overline{T}, \overline{C}$. The range of $\#$ is precisely the set of active words. The designated instruction schemata are: $\langle X(x, y, z), \#(\overline{X}, x, y, z) \rangle$ for $X = \overline{ADD}, \overline{SUB}, \overline{MULT}, \overline{T}, \overline{C}$.

We now define $M_1(\Sigma_0)$ as the RASP generated by the above five designated instruction schemata. We use $M_0(\Sigma_0)$ to denote $M_1(\Sigma_0)$ restricted so that all programs are fixed.

Given a set S of recursive functions, let $\Sigma_S =$

$\{S \cup \{T(x,y,z), C(x,y,z)\}\}$, then a corresponding set of designated instructions can be defined and used to generate a RASP $M_1(\Sigma_S)$ as follows. Let \mathcal{N} be an effective notation system for S with $\alpha: \mathcal{N} \rightarrow S$, e.g., using a Kleene type equation calculus, the names $n \in \mathcal{N}$ will be equations. Let $\beta: \mathcal{N} \rightarrow N$ be an effective arithmetization of \mathcal{N} . Given $f() \in S$ and given a change of control function $c()$, define the instruction schema for $f()$ as above. Select a unique brancy of α inverse and denote α^{-1} , then $\langle \rho_f, \beta \alpha^{-1}(f) \rangle$ will be a designated instruction schema for $f()$. $M_1(\Sigma_S)$ is the RASP generated by $\langle \rho_f, \beta \alpha^{-1}(f) \rangle$ for $f() \in S$.

2.15 For a RASP of type $M_1(\Sigma_S)$ there is a natural way to present programs. Recall that a program π is an n -tuple whose first member, p , is a mapping of addresses to instructions and data. The map p can be presented as a table. Consider the case $M_1(\Sigma_0)$; a sample program is $\pi = \langle p(), a, e \rangle$ represented by

$p()$	<u>argument (location)</u>	<u>value</u>
	a	$C(0,0,m); a' = a+1$
	m	$T(d,v); a' = a+1$
	$m+1$	$C(0,0e); a' = a+1$
	e	-----

Various informal versions of this will be used such as

<u>location</u>	<u>statement</u>	<u>comment</u>
a	$C(0,0,m)$	if $k(0) = k(0)$, then go to m for next instruction otherwise go to $a+1$
m	$T(d,v)$	transfer $k(d)$ to v , go to $m+1$ for instruction

m+1	C(0,0,e)	if k(0) = k(0) then go to e for next instruction, otherwise go to m+2
-----	----------	---

We will also use some suggestive abbreviations such as

$Y \leftarrow X$ for $T(x,y)$

$X = Y \Rightarrow Z$ for $C(x,y,z)$

2.16 Given a function $f() : N^p \rightarrow N$ we denote by π_f some program for computing $f()$ on either $M_0(\Sigma_0)$ or $M_1(\Sigma_S)$ depending on the context (which will always be clear).

2.17 As we build up various programs we may want to use them as "subprograms" in the construction of new programs. Such a use of programs can be made very explicit by the introduction of the instruction prefixes SUB, RTN.

A program π_g is said to be used as an explicit subprogram of π_f iff $D_{\pi_g} \subset D_{\pi_f}$, every transfer of control from $D_{\pi_f} - D_{\pi_g}$ into D_{π_g} is to an instruction whose prefix is SUB and every return from π_g to π_f is a return to an instruction whose prefix is RTN. We can thus think intuitively that the subprogram π_g is bracketed in π_f by SUB and RTN. Also it is clear that given any program π_g it can be used in the role of an explicit subprogram. Since we shall only refer to subprograms when they are explicit, we drop the adjective. It is also clear that prefixed instructions can be given new operation codes and new designations in accordance with 2.14.

$\pi_g = \langle p_g, a_g, E_g \rangle$ is an (explicit) subprogram of $\pi_f = \langle p_f, a_f, E_f \rangle$ iff

(a) $D_{\pi_g} \subset D_{\pi_f}$ and $E_g \subset D_{\pi_f} \cup E_f$ and if $e_i \in D_{\pi_f} \cap E_g$, then $k(e_i) = \text{RTNI}$ for I an instruction of π_f

(b) if $p(a) \in D_{\pi_f} - D_{\pi_g}$ and control can transfer into

$b \in D_{\pi_g}$ in D_{π_g} then that next control location b must

be a_g and $k(a_g) = \text{SUBI}$ for I an instruction of π_g .

Given any program $\pi_g = \langle p, a, E \rangle$ it is clear how to prepare it to be a subprogram of π_f . First replace the instruction $k(a)$ by the instruction $\text{SUB}k(a)$ and then prefix all instructions of π_f occurring in E by RTN .

2.18 A program π_f can generate its own subprograms by setting up the appropriate code in memory during execution. We say that program π_f generates a subprogram π_g at input X iff given $\sigma_0(X), \sigma_1(X), \dots, \sigma_n(X), \dots$ a computation of π_f at $X \exists k_n()$ such that $k_n()$ holds π_g but $k_0()$ does not, and π_g is a subprogram of π_f .

2.19 Before closing this account of RASP machines let us point out some facts which help in lending perspective to the hierarchy results. First notice that a RASP serves as a model of a very wide range of computational devices. Besides being a good abstract model for the behavior of a conventional digital computer, a RASP can model computational processes described by "problem oriented languages" or computational processes carried out by hand. A RASP can also be used to represent computations on classical Turing machines, multi-tape Turing machines and various general-

izations of Turing machines (such as Cook's Bounded Activity Machine). Furthermore, there is a precise sense in which a RASP such as $M_1(\Sigma_0)$ can simulate machines similar to it such as the URM of Shepherdson and Sturgis. Moreover if the elements of A are taken to be ordinals, then various computation processes on the ordinals can be represented on a suitable RASP.

One of the primary purposes for which Elgot & Robinson introduced the RASP was to study the relationship between machine oriented languages (MOL) and problem oriented languages (POL). Since POL's are designed for the convenience of the human user whereas MOL's are severely constrained by machine requirements, one might regard POL as abbreviating "people oriented language". In their study Elgot & Robinson pose the fundamental problem, called by them the compiler problem, in terms of extending the machine language of a RASP P by activating certain passive words. They seek an algorithm which will operate on any program in an extension of the machine language for P and produce an equivalent program in machine language itself. We will see below that this notion of language (or machine) extension plays a significant role in complexity theory as well. Perhaps theoretical insights gained in the study of computer based complexity theory will be valuable in studies of programming languages.

Chapter 3 The Grzegorzczuk, Ritchie and Cleave Hierarchies

In this chapter we present a proof-free account of the Grzegorzczuk, Ritchie and Cleave subrecursive hierarchies. The chapter is intended to help motivate work in the rest of this paper. Before getting into these three subrecursive hierarchies, we define some characteristics of RASP computations and programs which are important for a complexity theory. These definitions will fix the relationship between the three subrecursive hierarchies and the RASP even before we get down to details.

For every partial recursive function $\phi: N \supset D \rightarrow N$, there is a program π_ϕ such that π_ϕ computes ϕ on $M_1(\Sigma_0)$. This fact is proved in Elgot & Robinson for their machine P_0 which is easily seen to be a submachine of $M_1(\Sigma_0)$ in the sense that all programs of P_0 are programs of $M_1(\Sigma_0)$. The converse, that every program π on $M_1(\Sigma_0)$ computes a partial recursive function, will be established in this paper (Appendix C).

We are interested only in total recursive functions. For every $f(\) \in \mathcal{R}$ there are infinitely many programs, $\pi_{f,1}$, $\pi_{f,2}$, $\pi_{f,3}$, ... which compute $f(\)$ on $M_1(\Sigma_0)$. Certain properties of these programs are basic to complexity theory. Consider the following properties.

3.1 Computing parameters

- (a) $\sigma\pi_f(X)$ = number of atomic steps in computation

- of π_f with input $X \in N^n$, i.e. if $\sigma_0(X), \sigma_1(X), \dots, \sigma_n(X)$ is a computation of π_f at X then $\sigma\pi_f(X) = n+1$.
- (b) $\delta\pi_f(X)$ = number of times the instruction $C(\)$ is executed in computation of π_f with input $X \in N^n$.
- (c) $\tau\pi_f(X)$ = number of registers changed in computation of π_f with input $X \in N^n$.

We say that $\sigma(\)$ measures computing time, that $\delta(\)$ measures number of decisions and that $\tau(\)$ measures working space. In general, functions which measure properties of a computation of π are called measures on computation or computing parameters of π . An abstract definition of these measures is given in Blum[3]. According to that definition $\sigma(\)$, $\delta(\)$ and $\tau(\)$ are measures on computation.

We write $\alpha f(\) < b(\)$ for any computing parameter $\alpha(\)$ iff there exists a program π_f such that $\alpha\pi_f(X) < b(X)$ $X \in N^n$. In this case $\alpha(\)$ is also called a computing parameter of $f(\)$.

There are other properties of programs π_f for $f(\)$ which are relevant to complexity theory and which are of an entirely different character than computing parameters. Informally these are

3.2 Algorithmic parameters

- (a) $\ell(\pi_f)$ = number of primitive symbols in an encoding of π_f .
- (b) $i(\pi_f)$ = number of $C(\)$ instructions in π_f .

(c) $d(\pi_f)$ = depth of "nesting" of $C()$ instructions in π_f .

To define and develop properties of algorithmic parameters requires a formalized theory of programs. To move in this direction we have to describe our machine in terms of concrete objects, integers and finite sets of integers, instead of abstract objects, function spaces and functionals. This will be done in a later section. The classical development of Turing machines proceeds in terms of concrete objects and affords examples of algorithmic properties. In particular the state-symbol product is an example of an algorithmic property of machines (see Shannon[]). A basic result in this area is the existence of a universal Turing machine. This fact can be interpreted to mean that there is an upper bound to certain measures of algorithmic complexity, such as the number of $C()$ instructions in a program.

Although there are some specific results on algorithmic complexity, there is yet no general theory for algorithmic complexity of recursive functions. For the special case of finite automata there is a new and developing theory of algorithmic complexity. This theory may suggest generalizations applicable to a wide class of recursive functions. We will consider this problem in more detail in Part II. Now we turn our attention to computational complexity and

present an informal account of the Grzegorzczuk, Ritchie and Cleave subrecursive hierarchies.¹

Grzegorzczuk in [15] presented a subrecursive hierarchy, \mathcal{E}^a , for which $\bigcup_{a=0}^{\infty} \mathcal{E}^a = \mathcal{R}^1 =$ primitive recursive functions. His hierarchy classes were developed in terms of elementary operations applied to a sequence $\lambda xy g_n(x, y)$ of functions.

3.3 The elementary operations are given below.

(1) Operations of substitution

(a) If $h(x_1, \dots, x_{k-1}, y_1, \dots, y_m, x_{k+1}, \dots, x_n) = f(x_1, \dots, x_{k-1}, g(y_1, \dots, y_m), x_{k+1}, \dots, x_n)$, then $h()$ is said to be obtained from $f()$ and $g()$ by substitution of $g()$ in $f()$.

(b) If $h(x_1, \dots, x_j, y, x_k, \dots, x_n) = f(x_1, \dots, x_j, y, y, \dots, y, x_k, \dots, x_n)$, then $h()$ is said to be obtained from $f()$ by indentification of variables.

(c) If $h(x_1, \dots, x_{k-1}, x_{k+1}, \dots, x_n) = f(x_1, \dots, x_{k-1}, c, x_{k+1}, \dots, x_n)$, then $h()$ is said to be obtained from $f()$ by substitution of a constant.

Denote the operations of substitution by O_s .

(2) Operation of limited recursion.

If $h(X, 0) = g(X)$, $h(X, y+1) = f(X, y, h(X, y))$ and $h(X, y) \leq j(X, y)$, then $h(X, y)$ is said to be defined from $g()$, $f()$ and $j()$ by limited recursion.

Denote the operations of limited recursion by $\lim_{\leq j()}$.

¹For recent literature on the topic of algorithmic complexity see Chaitin[5], Engler[10], Krohn & Rhodes[30], Hartmanis & Stearns[19], Ritchie & Meyer[34].

The functions $g_n()$ were defined by the equations

$$\begin{aligned} g_0(x, y) &= y+1 \\ g_1(x, y) &= x+y \\ g_2(x, y) &= (x+1) \cdot (y+1) \end{aligned}$$

\vdots

$$\begin{cases} g_{n+1}(0, y) = g_n(y+1, y+1) \\ g_{n+1}(x+1, y) = g_{n+1}(x, g_{n+1}(x, y)), \end{cases}$$

where $g_0()$ and $g_1()$ are special cases.

The class \mathcal{E}^n was defined to be the least class containing $x+1$, $U_1(x, y) = x$, $U_2(x, y) = y$, $g_n()$ as initial functions and closed under the operations of substitution and limited recursion. Grzegorzczuk then showed that

$$\mathcal{E}^n \subset \mathcal{E}^{n+1}, \bigcup_{n=0}^{\infty} \mathcal{E}^n = \mathcal{R}^1 \text{ and } \mathcal{E}^3 = \mathcal{E} \text{ the class of elementary functions of Kalmar-Csillag.}$$

tary functions of Kalmar-Csillag.

The class \mathcal{E}^0 of this hierarchy has the property that it is a basis for the class of recursively enumerable sets, that is if S is an r.e. set enumerated by the recursive function $f()$, ($S = \{ f(x) \mid x \in \mathbb{N} \}$), then there is a recursive function $g() \in \mathcal{E}^0$ such that $g()$ enumerates S . The property of being a basis for the r.e. sets is an important property for a class of functions to possess. It is a measure of the richness of the class (see Smullyan[54], Kreider & Ritchie[27] for further information).

The class \mathcal{E}^3 of Kalmar-Csillag elementary functions (denoted often by K , here by \mathcal{E}) is quite noteworthy since

it is so basic to the theory of subrecursive structures. In his paper [15] Grzegorzcyk offered several alternative definitions of \mathcal{E} . They are worth listing. First define

(3) Limited summation

If $h(X, y) = \sum_{i \leq y} f(X, i)$, then $h()$ is said to be defined from $f()$ by limited summation.

Denote the operation by $\sum_{\leq y}$.

(4) Limited multiplication

If $h(X, y) = \prod_{i \leq y} f(X, i)$, then $h()$ is said to be defined from $f()$ by limited multiplication.

Denote the operation by $\prod_{\leq y}$.

(5) Minimum operations

(a) Limited minimum. If

$$h(X, y) = \begin{cases} \mu z < y \text{ for which } f(X, z) = 0 & \text{if such a } z \text{ exists} \\ 0 & \text{otherwise,} \end{cases}$$

then $h()$ is said to be defined from $f()$ by the operation of limited minimum.

Denote this operation by $\min_{\leq y}$.

(b) Minimum value. If $g(X, y) = \min_{x \leq y} f(X, y)$, then $h()$ is said to be defined from $f()$ by the operation of limited minimum value.

Denote this operation by $\text{Min}_{\leq y}$.

(6) Maximum operations

(a) Limited maximum. If

$$h(X, y) = \begin{cases} \text{largest } z \leq y \text{ such that } f(X, y) = 0 \\ 0 \text{ if no such } z \text{ exists} \end{cases}$$

then $h()$ is said to be defined by limited maximum from $f()$.

Denote this operation by $\max_{\leq y}$.

(b) Maximum value. If $h(X, y) = \max_{i \leq y} f(X, i)$ ("max"

here is applied to sets) then $h()$ is said to be defined by limited maximum value.

Denote this by $\text{Max}_{i \leq y}$.

He then proves that the following definitions of \mathcal{E} are equivalent. Let $[f_1(), \dots, f_n(); O_1, \dots, O_p]$ denote the least class containing $f_i()$ $i = 1, \dots, n$ as initial functions and closed under the operations O_i $i = 1, \dots, p$.

Definitions of \mathcal{E} :

- (a) $[x+1, x+y, x \div y; O_s, \Sigma, \Pi]_{\leq y, \leq y}$
- (b) $[x+1, x \div y, x^y; O_s, \min]_{\leq y}$
- (c) $[x+1, x^y; O_s, \lim_{\leq j()} (\text{limited recursion})]$
- (d) $[x+1, x \div y, x \cdot y, x^y; O_s, \Sigma]_{\leq y}$
- (e) $[x+1, x \div y, x \cdot y, x^y; O_s, \Pi]_{\leq y}$
- (f) $[x+1, x \div y, x \cdot y, x^y; O_s, \max, \text{Max}]_{\leq y, \leq y}$
- (g) $[x+1, x \div y, x \cdot y, x^y; O_s, \text{Min}]_{\leq y}$

Kleene in IM p. 285 and Péter in [39] defined \mathcal{E} in a way equivalent to

$$(h) \quad [x+1, x+y, x \cdot y, [x/y]; 0_s, \sum_{\leq y}, \prod_{\leq y}]$$

which is essentially the way Kalmar originally defined the class.

In the familiar way a class of predicates can be associated with \mathcal{E} . A predicate $P(X)$ $X \in N^n$ is said to be elementary iff $\exists f(\) \in \mathcal{E}$ such that $P(X) \iff f(X) = 0 \ \forall X \in N^n$. Grzegorzcyk and Kleene show that the class of relations of \mathcal{E} is closed under

- (i) operations of the propositional calculus ($\&$, \vee , $-$)
- (ii) limited quantification ($\forall x \leq y$, $\exists x \leq y$).

The scope of the elementary functions and predicates becomes clear from the fact that all the examples of primitive recursive functions ##1-21 of Kleene's IM (pp. 222-230) and all predicates used in arithmetizing his function calculus are elementary. Likewise in Davis[9] (pp. 58-62) all the functions and predicates used to arithmetize the theory of Turing machines are elementary. Thus the function $U(\)$ and predicates $T_n(\)$ of each book are elementary.

The intuitive conclusion is that almost all functions of practical use in logic are elementary. There are numerous other ways of lending intuitive significance to \mathcal{E} . Since

\mathcal{E} is closed under the usual operations of arithmetic, it is customary to think of \mathcal{E} as representing the class of

functions which arise naturally in elementary number theory. However, to obtain functions beyond \mathcal{E} it is necessary only to allow iteration or primitive recursion as a new operation. In fact Péter shows (p. 84) that

$$[x=0, x+1, x \dot{-} y, x \cdot y, sq(x), \overline{sq}(x), [\sqrt{x}]; 0_s, IT_1] = \mathcal{Q}^1$$

where IT_1 is the operation of iteration of unary functions,¹ e.g.

(7) Iteration

If $h(0) = 0$, $h(x+1) = f(f(x))$, then $h()$ is said to be defined by iteration₁ from $f()$.

The Grzegorzcyk hierarchy itself provides another example of how to move from \mathcal{E} to \mathcal{Q}^1 . We now return to continue a discussion of this hierarchy as preparation for the Ritchie and Cleave hierarchies.

In subsequent accounts of the Grzegorzcyk hierarchy (e.g. Ritchie) the operations of substitution have been replaced by

(8) Composition

If $h(X) = f(g_1(X), \dots, g_p(X))$ $X \in N^n$, then $h()$ is said to be obtained from $f()$ and $g_i()$ $i = 1, \dots, p$ by composition.

(9) Explicit transformation

If $h(x_1, \dots, x_n) = f(\alpha_1, \dots, \alpha_k)$ $k \leq n$ where for each α_i $i = 1, \dots, k$ $\alpha_i = x_j$ $j = 1, \dots, n$ or $\alpha_i = a$ constant,

¹ $sq(0) = 0$; $sq(n+1) = 1$ and $\overline{sq}(0) = 1$, $\overline{sq}(n+1) = 0$.

then $h()$ is said to be obtained from $f()$ by explicit transformation.¹

Thus \mathcal{E}^n can be defined as the least class containing the aforementioned initial functions and closed under composition, explicit transformation and limited recursion.

In general for $n \geq 3$ \mathcal{E}^n is the set of all functions elementary in $g_n()$ where "elementary in" is defined as follows.

A function $h()$ is said to be elementary in the functions $f_1(), \dots, f_n()$ iff $h()$ can be defined by a finite number of applications of composition, explicit transformation and limited recursion beginning with functions from

- (a) $f_1(), \dots, f_n()$
- (b) $x+1$
- (c) $U_1^m(x_1, \dots, x_m) = x_1 \quad 1 \leq i \leq m.$ ²

If $h()$ is elementary in $f()$, we write $h() \leq_e f()$. By the elementary degree containing $f()$ we mean $\{h() \mid h() \leq_e f() \text{ and } f() \leq_e h()\}$. Denoting the elementary degree of $f()$ by $\{f()\}_e$ and denoting the class of all functions elementary in $f()$ by $\mathcal{E}(f())$ we can then state

$$\mathcal{E}^n = \mathcal{E}(g_n()) \quad n \geq 3 \text{ and if } f() \in \mathcal{E}^n, \text{ then } \{f()\}_e \subset \mathcal{E}^n \quad n \geq 3.$$

Grzegorzczuk's functions $g_n()$ can be clearly understood

¹This definition is due to Smullyan[54] p. 21.

²See Kleene IM p. 286 for an equivalent definition.

in terms of rate of growth and relative "size" in the structure $\langle \mathcal{R}, \leq \rangle$ where \leq is a partial order relation defined as

$$f(\) \leq g(\) \text{ iff } f(X) \leq g(X) \text{ a.e. } X \in N^n$$

where a.e. (almost everywhere) means for all but finitely many $X \in N^n$.

Grzegorzczuk shows that his $g_n(\)$ satisfy:

- (1) $g_n(x, y) > y \quad \forall n > 1 \quad \forall x$
- (2) $g_{n+1}(x+y, y) > g_{n+1}(x, y) \quad \forall n \quad \forall x \quad \forall y$
- (3) $g_{n+1}(x, y+1) > g_{n+1}(x, y) \quad \forall n \quad \forall x \quad \forall y$
- (4) $0 < i \leq n \implies g_i(x, y) < g_{n+1}(x, y) \quad \forall x \quad \forall y$

So the $g_n(\)$ form a strictly ascending sequence of strictly increasing functions in $\langle \mathcal{R}, \leq \rangle$. We call the $g_n(\)$ backbone functions for the hierarchy, \mathcal{E}^n . This terminology suggests not only the picture in terms of $\langle \mathcal{R}, \leq \rangle$ but also the fundamental role that these functions play in the construction of the hierarchy. Most of the essential properties of \mathcal{E}^n are proved using rate of growth arguments on the $g_n(\)$.

The functions $g_n(\)$ are closely related to the celebrated Ackermann function (see Kleene[24] p. 271 or Péter[39] p. 106) which Ackermann produced as one of the first examples of an effectively calculable function which was not primitive recursive. Ritchie in [43] has shown that functions $\lambda xyf_n(x, y)$ which are essentially like Ackermann's suffice to obtain the Grzegorzczuk hierarchy.

For the purposes of a computational complexity theory the Grzegorzcyk hierarchy suffers from the defect of arbitrariness. There seems to be no good intuitive reason for selecting the backbone $g_n()$. Perhaps because of this the Grzegorzcyk hierarchy (which we denote hereafter as GH) did not serve as a catalyst to the now burgeoning theory of computational complexity. A hierarchy which did help spark the activity in this area was the Ritchie hierarchy developed in [42].

Ritchie addressed himself to one of the fundamental problems in the foundations of recursion theory, namely to describe a natural subclass of recursive functions which was more constructive than \mathcal{R} itself. As we mentioned in the introduction, the difficulty with \mathcal{R} is that although every element in \mathcal{R} is a constructive object, the class \mathcal{R} itself is not. Thus while we may prove that given a program $\pi_f()$, for every input X there exists a number y such that π_f computes $f(X)$ in less than y steps, the proof may not tell us how actually to find such a y . If the method of proof is open to doubt, say by an intuitionist or a finitist, then there is doubt about whether $f()$ is in \mathcal{R} .¹ Without having a bound on y there is no way directly to test whether $f(X)$ is indeed defined at X . To some logicians with constructivist tendencies this situation is troublesome. They

¹See Heyting[23] and Kreisel[29] for a more detailed discussion.

want a "safer" class of recursive functions. It is interesting that such a philosophical demand has now become associated with a more practical demand for a narrower class of recursive functions. In the expository articles of Cobham[7], McCarthy[32], and others, it is shown that a good theory of computation is required to support and advance the exploding field of computer science. Basic to such a theory will be a class or classes of recursive functions which correspond in various ways to the vaguely conceived class of "practically" computable functions. Thus from this computer science direction came a group of researchers led by Hartmanis and Stearns to join with the group of logicians who had been working on such problems as ordinal recursion, provably recursive functions, independence results in recursive arithmetic, transfinite progression of theories, the theory of constructive ordinals and the theory of sub-recursive hierarchies. Ritchie's work is ideal middle ground. Its technical features relate directly to computing theory via automata and Turing machines whereas its content ties in directly with the philosophical problem of constructivity.

What Ritchie did is this. He took as his basic class, F , those functions computable by a finite automaton. For every $f \in F$, $f(X)$ can be computed with time bound $y = \max X + c$. From the viewpoint of computing theory the

choice of F is also judicious because the finite automaton is a concept basic to study of computer components.

To generate his hierarchy Ritchie uses a Turing machine with a single tape infinite in one direction. He defines $a(M)_f(X)$ as the amount of tape used in the computation of $f(\)$ at X on a Turing machine M . The $a(M)_f(X)$ can be thought of as the maximum length of the tapes appearing in the instantaneous descriptions of the computation. Writing $f(\) \leq S$ iff $\exists b(\) \in S$ such that $f(X) \leq b(X) \ \forall X \in N^n$, the Ritchie hierarchy is now defined as follows:

$$(1) \ F_0 = F$$

$$(2) \ F_{n+1} = \{f(\) \mid \exists M \ a(M)_f(\) < F_n\}$$

Ritchie calls the functions in $F_\omega = \bigcup_{i=0}^{\infty} F_i$ the predictably computable functions. The terminology captures the idea that for $f(\) \in F_\omega$ the amount of tape used to compute $f(\)$ can be predicted from functions of a lower level (thus of lower complexity). The class F_ω has a very appealing constructive structure. F_ω are "secure" because at each stage they can be secured in terms of the previous stage. That is, if a person believes that all functions of F_n are "constructive", then he ought to believe equally well that those of the class F_{n+1} are.

Ritchie proves that $F_\omega = \mathcal{E}$. This is a new and interesting characterization of the class of Kalmar-Csillag elementary functions. However, this result immediately

raises the question of extending the hierarchy. It is known that \mathcal{E} is only \mathcal{E}^3 in the GH. Why then does F_ω stop at \mathcal{E}^3 ? What feature of \mathcal{E}^4 keeps it from being included among the predictably computable functions? Can the hierarchy be extended while preserving most of its constructive features? Can the hierarchy be extended while preserving its high relevance for computability theory, in particular toward computational complexity theory? Cleave has given interesting answers to these questions.

To extend the Ritchie hierarchy (RH), Cleave first changes to a more sophisticated machine, the URM of Shepherdson & Sturgis. Instead of measuring "space used" in the computation, he measures the number of binary decisions made during a computation of program π_f .

Let C be the class of constant functions and let $E(\Sigma)$ be the set of URM-computable functions with set of basic instructions Σ ; this is essentially the same as $M_1(\Sigma)$. Now define

$$\Sigma = \{+, x, \delta\}$$

where

$$\delta(X, Y) = \begin{cases} 1 & \text{if } X = Y \\ 0 & \text{otherwise.} \end{cases}$$

(This set is essentially like our Σ_0 .) Then a Ritchie type hierarchy is defined

$$(1) \quad E_{-1} = E(\Sigma)_{-1} = C$$

$$(2) \quad E_{i+1} = E(\Sigma)_{i+1} = \{f(\) \mid \delta f(\) \leq E(\Sigma)_i\}$$

$$(3) \quad E_{\omega} = \bigcup_{i=0}^{\infty} E_i$$

Cleave shows that $E_{\omega} = F_{\omega}$ and that the classes E_i have properties similar to those of F_i . Cleave now extends the Ritchie hierarchy by taking

$$(4) \quad E_{\omega \cdot a + 1} = E(E_{\omega \cdot a})_1$$

That is, at the limit steps, $\omega \cdot a$ he takes all previously obtained functions as new primitive instructions for his computer, and then he continues with the Ritchie process. This technique relies on the use of a machine which can accept infinitely many primitive instructions. Such a device is a kind of super computer. However, we can interpret Cleave's results in another way. At the limit step, $\omega \cdot a$, we can imagine that the method of "telling time" (measuring computation steps) is changed so that whenever a program π_f computing a function $f(\) \in E_{\omega \cdot a}$ appears as a subprogram in a larger program, then the steps used in computing π_f are counted as only a single step. This interpretation might be called the primitive subprogram interpretation as opposed to the super computer interpretation because programs of $E_{\omega \cdot a}$ are regarded as primitive instructions of the machine as far as counting running time of programs.

The Cleave hierarchy generating process works well as an extension of the Ritchie process. Cleave shows the following

$$(1) \quad E_{\omega \cdot a} = \mathcal{E}^{a+2}, \quad \text{thus } E_{\omega} = F_{\omega} = \mathcal{E} = \mathcal{E}^3$$

$$(2) \quad \alpha < \beta < \omega^2 \implies E_{\alpha} \subset E_{\beta}$$

(3) $E_{\omega_2} = \mathcal{R}^1 =$ primitive recursive functions.

So not only does Cleave extend RH using computing theoretic methods, but he obtains a refined GH as a byproduct. The classes $E_{\omega.a}$ preserve some of the constructive character of F_{ω} , and the theory is quite relevant to questions of computational complexity. Furthermore, Cleave has provided intuitive content to the GH and has revealed certain properties of \mathcal{R}^1 that help explain its fundamental role in recursion theory. But again, the question of extension arises.

Péter has constructed a very rich and powerful hierarchy of recursive functions, $\mathcal{R}^1 \subset \mathcal{R}^2 \subset \mathcal{R}^3 \subset \dots \subset \mathcal{R}^n \dots$ called the multiply recursive functions. This hierarchy is based on a syntactical description of recursion equations, and it is not immediately clear how such a hierarchy is related to those based on computational complexity.¹ Since $E_{\omega_2} = \mathcal{R}^1$ we can ask questions about the Cleave hierarchy and the Péter hierarchy analogous to those asked about the Ritchie and the Grzegorczyk hierarchies. In fact, the work of Robbin shows the Péter hierarchy to be an extension of the GH (called Extended Grzegorczyk Hierarchy, EGH), thus the analogy is quite exact. Can the Cleave hierarchy be extended to include the EGH? In what sense are the elements

¹Robbin has recently clarified this relationship in terms of Hartmanis & Stearns complexity classes, and at this writing it seems that Dennis Ritchie is also working on this relationship (expected Ph.D. thesis).

of \mathcal{Q}^n $n > 1$ less constructive than those of E_{ω^2} ? Can the Cleave hierarchy be extended while preserving most of its constructive features? Can the hierarchy be extended in terms of purely computer theoretic concepts? What is the next natural stopping point (closure ordinal) for this Ritchie-Cleave type hierarchy?

As a starting point to the investigation of these questions let us examine why the Cleave hierarchy closes off at ω^2 . What we show is that $E_{\omega^2+1} = E_{\omega^2}$. By definition $E_{\omega^2+1} = E(E_{\omega^2})_0$ and $E_{\omega^2} = \bigcup_{\alpha < \omega^2} E_{\alpha}$. If π is a program with instructions from E_{ω^2} , then since programs contain only a finite number of instructions, say $h_1(\), \dots, h_p(\)$, we can find an n such that $h_i(\) \in E_{\omega \cdot n}$ $i = 1, \dots, p$. Now to say $f(\) \in E(E_{\omega^2})_1$ means there is an E_{ω^2} program, π_1 , computing $f(\)$ with a bound $g(\) \in E(E_{\omega^2})_0$. Since $g(\) \in E(E_{\omega^2})_0$, there is a program π_2 computing $g(\)$ whose instructions all come from $E_{\omega \cdot n_2}$ for some n_2 , so $g(\) \in E(E_{\omega \cdot n_2})$. Choose n_1 so that the instructions of π_1 are in $E_{\omega \cdot n_1}$. Then if $m = \max\{n_1, n_2\}$, $f(\) \in E_{\omega \cdot m} \subset E_{\omega^2}$.

From the above argument it is apparent that an essential reason for the hierarchy's collapse at ω^2 is that programs for URM contain only a fixed and finite number of instructions. What happens if programs can reflexively change

their own instructions or contain infinitely many instructions? The stored programs of actual computers can alter themselves in the course of their own computations. For programs with such a capability it is clear that the above argument will not work to collapse the hierarchy. Will other arguments work or will self-modifying programs extend the hierarchy?

One of the features of RASPs (in contrast to URM's) is their capability of using self-modifying programs. We propose to study the Cleave hierarchy on such machines. First we will present the Cleave hierarchy in detail on a RASP using only fixed programs. Our approach is different from Cleave's. We give the proofs in terms of $\sigma(\)$ rather than $\delta(\)$ and we proceed directly to the hierarchy results whereas Cleave also presents results relating his machine to simultaneous recursions. It is not possible to simply carry over Cleave's results on simultaneous recursions to RASPs. Approximating such results would require developing some elaborate applications of recursive functionals. In place of this we use an arithmetization of $M_1(\Sigma_0)$.

Chapter 4 The Ritchie-Cleave Hierarchy

The subrecursive hierarchy which is presented below will be called the Ritchie-Cleave Hierarchy (RCH) rather than the Cleave Hierarchy because the role of Ritchie's ideas is so central. (From a technical point of view it might be called the Ritchie-Grzegorzczuk-Cleave Hierarchy since many of the proof techniques come from Grzegorzczuk as well as Ritchie. However, to keep the computing theoretic character of the hierarchy distinguished we stick with the term Ritchie-Cleave Hierarchy.)

4.1 Basic to the technical results about the RCH are the notions of a normal set of functions and of a normal element for a set of functions.

1. A function $f()$ is said to majorize a class of functions S iff $\forall g() \in S \exists p \in \mathbb{N}$ such that $g(X) < f^{(p)}(\max X)$ $X \in \mathbb{N}^n$.

For S a set of functions let $S_u = S \cup \{U_1^n()\}$ where for $1 \leq i \leq n$ $U_1^n(x_1, \dots, x_n) = x_i$ and let S_u^* be the closure of S_u under composition. Then

2. S is a normal set of functions and $f()$ a normal element for S iff
 - (a) $f() \in S_u^*$
 - (b) $f()$ is strictly increasing
 - (c) $f()$ majorizes S_u^* .
3. A function $\lambda x f(x)$ dominates the one argument

functions of a class S iff $\forall h() \in S \exists m$ such that
 $h(x) < f(x) \quad x > m$.

4. A set of instructions $\Sigma = A \cup \{T(), C()\}$ where A is a set of arithmetic instructions is called normal iff the associated set of arithmetic functions is normal. A set of programs, P , is called normal iff the set of functions computed by members of P is normal. For example the basic instruction set $\Sigma = \{+, -, \times, T(), C()\}$ is normal with normal element $f_0(x) = (x+1)^2$.

The usefulness of the notion of normality derives from the fact that most of the arguments which establish the basic hierarchy properties are rate of growth type arguments. It is convenient to handle monotonic functions in such arguments, and it is necessary to have closure under composition. Moreover it is useful to be able to treat a single function, the normal element, when examining the growth behavior of a set of instructions. Thus a normal set of instructions is roughly one which has a largest instruction in some sense.

4.2 Before defining the RCH we need the notion of σ_P -computing time for P a set of programs.

1. $\sigma_P \pi_f(X)$ = number of atomic steps relative to P in the computation of π_f on $M_0(\Sigma_0)$ where a single step relative to P is either the execution of a program of P occurring as a subprogram in π_f or the execution of a single instruction of Σ_0 not occurring in such a subprogram.

2. A subprogram π_g of P occurring in a program π is called primitive with respect to P . Such terminology indicates that π_g acts as an instruction when counting σ_P -computing time.

3. For convenience in defining the RCH we first define the map $*a$. If P is a set of programs, then

$P^{*a} = \{f() \mid \pi_f \in P\}$. Given a set $\mathcal{R} \supset S$ we also define the inverse type of map, a^* . Let $S^{a^*} = \{\pi_f \mid f() \in S\}$.

4.3 For P a set of programs put $L(P)_{-1} = C = \{\text{constants}\}$ and put $L(P)_1 = \{\pi_f \mid \sigma_P \pi_f() < (L(P)_{1-1})^{*a}\}$. Then define

$$(a) L_0 = \Sigma_0$$

$$(b) L_{b,1} = L(L_b)_1$$

$$(c) L_{b+1} = \bigcup_{i=0}^{\infty} L_{b,i}$$

Now set $E_{b,1} = (L_{b,1})^{*a}$, $E_b = (L_b)^{*a}$. The classes $E_{b,1}$ correspond to $E_{\omega.b+1}$ of Cleave. For the RCH it turns out that $E_{b,0} = E_b$, but this is not the case for the new hierarchy, \bar{E}_b , defined in Chapter 6. The classes E_b with the single index are the classes of primary interest since they are the most invariant classes of the theory. The refined classes $E_{b,1}$ are quite dependent on the exact computation parameters chosen (e.g., δ vs σ) and on the initial set Σ_0 of instructions.

We will now prove the following theorems about RCH. For convenience in these theorems we shall use the notation

σ_a in place of σ_{L_a} .

4.7 Hierarchy Theorem

- (a) $E_{a,1} \subset E_{a,1+1}$
- (b) E_a is normal with a normal element $f_a(\)$ for $a > 0$
- (c) $a < b < \omega$ implies $E_a \subset E_b$.

4.12 Actual Time Theorem

If $a < \omega$ and $g(\) \in E_{a,1}$, then $\sigma g(\) < E_{a,1}$.

4.13 Comparison of Hierarchies Theorem

$$E_a = \varepsilon^{a+2}.$$

These theorems are proved with the help of the following.

4.5 Main Lemma

Let $f_a(\)$ be a normal element for E_a and define

$$F_{a,0}(n,y) = f_a^{(n)}(y)$$

$$F_{a,M+1}(n,y) = f_a^{(F_{a,M}(n,y))}(y).$$

Then (a) for each $M, n \ \lambda y F_{a,M}(n,y) \in E_{a,M}$

(b) for each $M \ \lambda n y F_{a,M}(n,y) \in E_{a,M+1}$

(c) for all $g(\) \in E_{a,M} \ \exists m$ such that

$$g(X) < F_{a,M}(m, \max X)$$

(d) for each $M \ \lambda x x F_{a,M}(x,x) \in E_{a,M+1}$ and dominates the one argument functions of $E_{a,M}$.

Proof. Part (a) By definition $F_{a,0}(n,y) = f_a^{(n)}(y)$, and $f_a(\)$ is normal for E_a . Since $f_a(\) \in (E_a)_u^*$ it follows that $\sigma_a f_a(y) = \text{constant}$. To compute $f_a^{(n)}(y)$ a standard iteration block (see A.5) is set up with control parameter n which is fixed and may be stored as a program constant. Thus computing $\lambda y F_{a,0}(n,y)$ requires only $c \cdot n - 1$ steps for fixed n which means $\lambda y F_{a,0}(n,y) \in E_{a,0}$. For purposes of the inductive argument we also notice $\forall d \ \exists p$ such that

$$c(\sigma_a \lambda y F_{a,0}(n,y)) < \lambda y F_{a,0}(n,y) \quad y > p.$$

Continuing now by induction assume the result for M and for simplicity use $F_M()$ for $F_{a,M}()$. Then by definition $\lambda y F_{M+1}(n,y) = \lambda y f_a^{(F_M(n,Y))}(y)$. To compute $\lambda y F_{M+1}(n,Y)$ a standard iteration block is set up with $F_M(n,y)$ as controlling parameter. By the induction hypothesis this parameter can be computed in less than $\lambda y F_M(n,y)$ steps for $y > p$. Using the methods of the case $M = 0$ it is known that

$$\sigma_a \lambda y F_{M+1}(n,y) < (c+1) \cdot F_M(n,y) \quad y < p.$$

Letting S_n be the number of steps needed to compute for $y \leq p$ we have $\sigma_a \lambda y F_{M+1}(n,y) < (c+1) \cdot F_{M+1}(n,y) + S_n \forall y$. Now noticing that S_n and $F_M(n,y)$ belong to $D_{a,M}$ and using the lemma 4.6 below it follows that $\lambda y F_{M+1}(n,y) \in E_{a,M+1}$.

q.e.d. part (a)

4.6 Lemma $\forall a \forall M \quad E_{a,M}$ is closed under addition and multiplication.

The proof of 4.6 is by induction on M . Returning to 4.5 we consider the next part.

Part (b) Again $F_0(n,y)$ is computed by setting up a standard iteration block. But now the control parameter n is input. As in part (a) we have $\sigma_a F_0(n,y) < c \cdot n \forall y$. Putting $b(n,y) = c \cdot n$ we clearly have $b() \in E_{a,0}$ so by definition $F_0() \in E_{a,1}$. For future induction notice that $\exists c \quad \sigma_a F_0(n,y) < c \cdot F_0(n,y)$ since $f_a^{(n)}(y) > n$ by definition of normality in 4.1.

For induction assume $\lambda y F_M(n,y) \in E_{a,M}$. Arguing as above $\exists d \quad \sigma_a F_M(n,y) < d \cdot F_M(n,y)$ and $\sigma_a F_{M+1}(n,y) <$

$< (d+c) \cdot F_M(n, y) \forall n \forall y$ so that by 4.6 and the induction hypothesis $F_{M+1}(\) \in E_{a, M+2}$. To verify the remaining condition notice $F_M(n, y) < F_{M+1}(n, y)$ since $F_{M+1}(n, y) = f_a^{(F_M(n, y))}(y) > F_M(n, y)$ by 4.1, 2. (b).

q.e.d. part (b)

Part (c) To say $g(\) \in E_{a, 0}$ implies that $\exists \pi_g$ in which the instructions or primitive subprograms, say $h_1(\), \dots, h_2(\)$ (using function notation for the subprograms as well as the instructions) all come from E_a and for which the σ_a -computing time is a constant, c . Since $f_a(\)$ is normal for $E_a, \exists m_i \ h_1(Y) < f_a^{(m_i)}(\max Y) \ Y \in N^{n_i} \ i = 1, \dots, p$. Let $m = \max m_i$ and let the input to π_g be $X \in N^r$. Suppose the maximum constant of π_g is d . Then on the first step of the computation the maximum value accessible to π_g will be $f_a^{(m)}(\max\{X, d\})$. Therefore after s steps the maximum value accessible to π_g will be $f_a^{(m \cdot s)}(\max\{X, d\})$.

If $\sigma_a \pi_g(X) = c \forall X$, then

$$** \quad g(X) < f_a^{(m \cdot c)}(\max X, d) \quad X \in N^r.$$

Let q be the largest value of $f_a^{(m \cdot c)}(\max\{X, d\})$ for all X such that $\max\{X, d\} = d$. Then since $f_a^{(n)}(y) > n$, an e can be chosen such that $f_a^{(e)}(y) > q$ for all y . Thus putting $b = \max\{m \cdot c\} + e$ it follows that

$$g(X) < f_a^{(m \cdot c + e)}(\max X) = F_0(b, \max X) \forall X.$$

For induction assume the result for $D_{a, M}$, take $g(\) \in E_{a, M+1}$. Proceeding exactly as above to line **, notice that $\sigma_a \pi_g(X) < b(X) \in E_{a, M} \forall X \in N^r$.

Applying the induction hypothesis to $b(\)$ it follows that

$$\exists \bar{m} \quad \sigma_a \pi_g(X) < F_M(\bar{m}, \max X).$$

Using the inequality in line ** yields

$$g(X) < f_a^{(m \cdot F_M(\bar{m}, \max X))}(\max\{X, d\})$$

and as before

$$\exists e \quad g(X) < f_a^{(m \cdot F_M(\bar{m}, \max X) + e)}(\max X)$$

Since $E_{a,M}$ is closed under summation, $m \cdot F_M(\bar{m}, \max X) + e \in E_M$ and so using the induction hypothesis again we have

$$m \cdot F_M(\bar{m}, \max X) + e < F_M(b, \max X).$$

Thus since $\lambda n f_a^{(n)}(y)$ is increasing in both arguments,

$$g(X) < f_a^{(F_M(b, \max X))}(\max X) = F_{M+1}(b, \max X) \quad X \in N^T.$$

q.e.d. part (c)

Part (d) For each $M \lambda x x F_M(x, x) \in E_{a, M+1}$ and dominates the one argument functions of $E_{a, M}$. To compute $\lambda x x F_M(x, x)$ use the program for $\lambda n f_a^{(n)}(y)$ plus an instruction to use the input as iteration parameter. Now use part (b). The fact that $\lambda x x F_M(x, x)$ dominates $E_{a, M}$ follows immediately from part (c).

q.e.d. Main Lemma

4.7 Hierarchy theorem

$$(a) \quad E_{a, i} \subset E_{a, i+1}$$

$$(b) \quad E_a \text{ is normal with a normal element}$$

$$f_a(\quad) \in E_a \text{ for } a > 0$$

$$(c) \quad a < b \text{ implies } E_a \subset E_b.$$

Proof. Part (a) We show $F_1(x, x)$ is contained in E_{1+1} but not in E_1 . Containment is just 4.5 part (d), and non-containment follows immediately from the definition of dominance.

Part (b) E_a is normal with normal element $f_a(\)$ and for $a > 0$ $f_a(\) \in E_a$. Recall that $f_0(\)$ is normal for Σ_0 . Define $f_{a+1}(\)$ from $f_a(\)$ by the condition $f_{a+1}(x) = f_a^{(x)}(x)$. We have produced normal elements for Σ_0 (see 4.1). These cover the above result for the case $a = 0$. Continue by induction and assume $f_a(\)$ is normal for E_a . Then notice

$$f_{a+1}(x) = f_a^{(x)}(x) = F_{a,0}(x,x) \in E_{a,1}.$$

So $f_{a+1}(\) \in E_{a+1} = \bigcup_{i=0}^{\infty} E_{a,i}$. To complete the definition of normality $f_{a+1}(\)$ must be strictly increasing and must majorize E_{a+1} , since $E_{a+1} = (E_{a+1})_u^*$ for $a > 0$ by 4.8 below. That $f_{a+1}(\)$ is strictly increasing is proved in 6.23. To show that $f_{a+1}(\)$ majorizes E_{a+1} we must show that

$$g(\) \in E_{a,i} \text{ implies } \exists n \ g(\) < f_a^{(n)}(\max X) \quad X \in N^{\mathbb{F}}.$$

But by 4.6 part (b) it is sufficient to show

$$\forall M \forall n \exists m \quad F_{a,M}(n,x) < f_{a+1}^{(m)}(x) \quad \forall x.$$

Since $f_a^{(n)}(x) > n \quad \forall x \quad \forall n$ it follows that

$$F_{a,0}(n,x) \leq F_{a,n+1}(1,x) \quad \text{that is}$$

$$F_{a,0}(n,x) = f_a^{(n,x)}(x) \leq f_a^{(f_a^{(\dots f_a(x))}(x))}_{f_a^{(\dots f_a(x))}(x)} \}^{n+1}$$

So by induction on M

$$F_{a,M}(n,x) \leq F_{a,M+m+1}(1,x).$$

Also $F_{a,m}(1,x) < f_{a+1}^{(m+1)}(x)$ by induction on m , e.g.,

$$F_{a,0}(1,x) = f_a(x) \quad \text{and}$$

$$\begin{aligned} F_1(1,x) &= f_a^{(f_a(x))}(x) < f_a^{(f_a(x))}(f_a(x)) = \\ &= f_{a+1}(f_a(x)) < f_{a+1}^{(2)}(x), \text{ etc.} \end{aligned}$$

Hence $F_{a,M}^{(n,x)} < F_{a,M+m+1}^{(1,x)} < f_{a+1}^{(M+m+2)}(x)$.

q.e.d. part (b)

Part (c) $a < b$ implies $E_a \subset E_b$. Consider $b = a+1$ and let $f_{a+1}(\)$ be a normal element for E_{a+1} . then $f_{a+1}(\)$ belongs to E_{a+1} . Suppose $f_{a+1}(\) \in E_a$, then there is a d such that $f_{a+1}(x) < f_a^{(d)}(x)$ and for $x = d$ a contradiction results.

q.e.d. part (c)

4.8 Composition Theorem

Let $f(\) \in E_{a,M}^{[n]}$, and $g_i(\) \in E_{a,J}^{[m]}$ for $1 \leq i \leq n$. Then $h(X) = f(g_1(X), \dots, g_n(X))$ implies $h(\) \in E_{a,M+J}^{[m]}$.

Proof. To compute $h(\)$ first compute each of the $g_i(\)$ at X . This requires $S_1(X) = \sum_{i=1}^n \sigma_a g_i(X)$ steps. If $J = 0$, then $\sigma_a g_i(\) = c_i$ so $S_1(X) = s_1 = \sum_{i=1}^n c_i$. If $J > 0$, then $\sigma_a g_i(X) \in E_{a,J-1}$ and $S_1(\) \in E_{a,J-1}$ (lemma 4.6). To finish the computation requires $\sigma_a f(g_1(X), \dots, g_n(X)) = S_2(\)$ steps. If $M = 0$, then $S_2(X) = s_2$. So either $\sigma_a h(\) = s_1 + s_2$ in which case $h(\) \in E_{a,0}$ or $\sigma_a h(\) \in E_{a,J}$ giving the result for $M = 0, J > 0$.

If $M > 0$, then $\exists b \ \sigma_a f(g_1(X), \dots, g_n(X)) < F_{a,M-1}(b, \max g_i(X))$ and $g_i(X) < F_{a,J}(m_i, \max X)$ so taking $m = \max m_i$ it follows that

$$\sigma_a f(g_1(X), \dots, g_n(X)) < F_{a,M-1}(b, F_{a,J}(m, \max X)).$$

Now by the lemma 4.9 below we conclude

$$\sigma_a f(g_1(X), \dots, g_n(X)) < F_{a,M+J-1}(d, \max X) \text{ for some } d.$$

Thus $h(\) \in E_{a,M+J}$ by definition. q.e.d.

4.9 Lemma. $\forall M \forall J F_M(n, F_J(m, x)) < F_{M+J}(p, x) \quad \forall x$

Proof. We can either prove this directly using a tedious inductive and combinatorial argument or we can appeal to Cleave's paper in the following manner. Since $\lambda x F_M(n, x) \in E_M^e$ and $\lambda x F_J(m, x) \in E_J^e$ where E_J^e denotes Cleave's hierarchy, then using his proposition on composition (Corollary 4 p. 342), $\lambda x F_M(n, F_J(m, x)) \in E_{M+J}^e$. So by his Lemma 4 p. 342

$$\exists p F_N(n, F_J(m, x)) < F_{M+J}(p, x) \quad \forall x$$

q.e.d.

4.10 Limited Recursion Theorem.

If $a < \omega$, $h() \in E_{a, s+1}^{[n+2]}$, $g() \in E_{a, r}^{[n]}$ and there exists a function $k() \in E_{a, t}^{[n+1]}$ such that

$$f(X, 0) = g(X)$$

$$f(X, y+1) = h(X, y, f(., y))$$

$$f(X, y) < k(X, y) \quad X \in N^n \quad y \in N.$$

Then

$$f() \in E_{a, s+\max(t, r)+1}^{[n+1]}$$

Proof. To compute $f(X, y)$ start with $f(X, 0)$ and compute $f(X, 1), f(X, 2), \dots, f(X, y)$. This involves computing $g(X)$ and computing $h(X, 0, g(X)), h(X, 1, g(X, 1)), \dots, h(X, y, f(X, y))$. The total number of σ_a steps is thus

$$S(X, y) = \sum_{i=0}^y h(X, i, f(X, i)) + \sigma f(X)$$

Since $h() \in E_{a,s}$, $k() \in E_{a,t}$ and $g() \in E_{a,r}$ this can be estimated by

$$\sum_{i=0}^y F_s(b_1, \max\{X, i\}, F_{\max\{t,r\}}(b_2, \max\{X, i\})) + F_{r-1}(b_3, X)$$

and since $\max\{X, i\} < F_{\max\{t,r\}}(d, \max\{X, i\})$ and $\max\{X, i\} \leq \max\{X, y\}$

$$S(X, y) < y \cdot F_s(b_1, F_{\max\{t,r\}}(b_2, \max\{X, y\})) + F_{r-1}(b_3, X).$$

Now to handle $y \cdot F()$ consider the following argument.

$$y \cdot z < f_a^{(c)}(\max\{y, z\})$$

since x is in Σ_0 . So that for $z = F_a(m, q(y))$ where $q(y) > y$ it follows that for some e

$$y \cdot F_{a,i}(m, q(y)) < f_a^{(c)}(F_{a,i}(m, q(y))) \leq F_{a,i}(m+e, q(y))$$

Applying this to *,

$$S(X, y) < F_{s+\max\{t,r\}}(d, \max\{X, y\}) + F_{r-1}(b, X).$$

So

$$f() \in E_{a, s+\max\{t,r\}+1}$$

q.e.d.

4.11 Notice, the methods of this proof also give us that for $r > 0$, $E_{a,r}$ is closed under limited summation and limited multiplication, i.e. if $f(X, i) \in E_{a,r}$ and $S(X, y) = \sum_{i < y} f(X, i)$ and $P(X, y) = \prod_{i < y} f(X, i)$, then $S()$, $P() \in E_{a,r}$.

4.12 Actual Time Theorem.

If $a < \omega$ and if $g() \in E_{a,i}$ then $\sigma g() < E_{a,i}$.

Proof. We proceed by induction on a and i .

Consider the case $a=0$ for all i . If $a=0$, then $\sigma_a = \sigma$ so the result is in fact stronger than stated, namely $g(\) \in E_{0,i-1}$.

For induction assume the result for a . For induction on i , consider $g(\) \in E_{a,0}$, then there is a π_g so that $\sigma_a g(X) < c \quad X \in N^n$. Suppose π_g uses instructions $h_1(\), \dots, h_p(\) \in E_a$, then by the induction hypothesis $\sigma h_1(\) < E_a$ so

$$\sigma h_1(Y) < f_a^{(s_1)}(\max Y) \quad Y \in N^{n_1}$$

and also

$$h_1(Y) < f_a^{(v_1)}(\max Y) \quad Y \in N^{n_1}.$$

Putting $m = \max\{s_1, v_1\}$, then after $c \sigma_a$ -steps,

$$\begin{aligned} \sigma g(X) &\leq f_a^{(m)}(\max X) + f_a^{(m)}(f_a^{(m)}(\max X)) \\ &\quad + \dots \\ &\quad + f_a^{(m)}(f_a^{(m \cdot c)}(\max X)) \quad X \in N^n. \end{aligned}$$

Applying a simple estimate to this sum yields

$$* \quad \sigma g(X) \leq c \cdot f_a^{(m)}(f_a^{(m \cdot c)}(\max X)) \quad X \in N^n.$$

With these facts the next line follows immediately from line * and from closure of $E_{a,0}$ under addition (using 4.5 part (iii)).

$$\sigma g(X) < c \cdot f_a^{(m \cdot c + n)}(\max X) < f_a^{(\bar{m})}(\max X) \quad X \in N^n$$

for $\bar{m} > m$. So $g(\) < E_{a,0}$.

Continuing the induction on i , assume the result for $i = M$ and suppose $g(\) \in E_{a,M+1}$. Thus $\sigma_a g(X) < b(X) \in E_{a,M} \quad \forall X \in N^n$. Proceeding as in $i=0$ case to line * and putting

$b(X)$ in for c we get

$$\sigma g(X) < b(X) \cdot f_a^{(m)}(f_a^{(m \cdot b(X))}(\max X)) \quad X \in N^n.$$

Now noticing that $\exists e \ b(X) < F_M(e, \max X)$ (by 4.5 part (iii)) and applying the methods of $i=0$ case again, we have

$$\sigma g(X) < F_M(e, \max X) \cdot f_a^{(m \cdot F_M(e, \max X) + d)}(\max X).$$

Picking \bar{e} such that $m \cdot F_M(e, \max X) + d < F_M(\bar{e}, \max X)$ and noting that

$$f_a^{(F_M(\bar{e}, \max X))}(\max X) = F_{M+1}(\bar{e}, \max X),$$

it follows that

$$\sigma g(X) < F_M(e, \max X) \cdot F_{M+1}(\bar{e}, \max X).$$

So since $E_{a, M+1}$ is closed under multiplication there exists a d such that

$$\sigma g(X) = F_{M+1}(d, \max X) \quad X \in N^n.$$

So $g(X) < E_{a, M+1}$ which completes the induction on i , and

since $E_{a+1} = \bigcup_{i=0}^{\infty} E_{a, i}$ the result holds for E_{a+1} .

q.e.d.

4.13 Comparison of Hierarchies

(a) Let $E_{a, M}^e$ denote Cleave's hierarchy classes (in terms of δ). Then $\forall a, M < \omega \ E_{a, M} = E_{a, M}^e$.

(b) Let \mathcal{E}^n be Grzegorzcyk's hierarchy classes. Then $E_a = E_a^e = \mathcal{E}^{a+2} \quad a < \omega$.

Proof. First $\delta_a g(X) \leq \sigma_a g(X) \quad \forall X$. Since a RASP, M_0 , can fully simulate a URM (as discussed in Chapter 2), it follows that $E_{a, M}^e \subseteq E_{a, M}$. On the other hand, given a fixed

program for computing $g(\)$, $\exists n \sigma_a g(X) \leq n \cdot \delta_a g(X) \ \forall X$, and since $E_{a,M}$ is closed under addition for all a , $M \leq \omega$, the containment, $E_{a,M}^e \supseteq E_{a,M}$, holds.

Finally (b) follows from (a) and the results of Cleave.

q.e.d.

Note: We will demonstrate these results in an alternative manner in Chapter 6.

Chapter 5 A New Ritchie-Cleave Hierarchy

5.1 Our extension of the Ritchie-Cleave Hierarchy relies on a definition of program modification over an infinite set of instructions, or alternatively (and equivalently) on the notion of regarding subprograms of a program introduced by the program itself as primitive subprograms, those whose execution time is counted as a single step regardless of the actual number of steps required. The question confronting us is how to define these notions so that the hierarchy generated using them is interesting. There are several applicable criteria determining interest:

- Q1. How far into the class \mathcal{R} does the hierarchy extend?
- Q2. How does the hierarchy relate to other known hierarchies (in particular the Péter hierarchy and the Kleene subrecursive hierarchy)?
- Q3. How natural is the hierarchy (how independent of arbitrary choices, whether of ordinal notations, sequences of functions or operations on functions)?
- Q4. Can the hierarchy be described easily in terms of computing theoretic concepts or concepts from logic?

Suppose we allow a main program to generate any subprogram of level E_α as primitive at input X . Then it is possible to construct a program π which generates primitive subprograms only over E_1 and yet can compute any recursive

function within an elementary bound on E_1 -computing time, i.e. on the time measured by counting execution of primitive subprograms over E_1 as single steps. Thus E_2 would already be \mathcal{R} .

5.2 Considering the above argument more precisely define the notion $\bar{\sigma}_S$ -computing time for S a set of programs.

$\sigma_S \pi_f(X)$ = number of steps relative to S used in computing π_f at input X where execution of any of the following is regarded as a single modified step:

- program of S occurring as a subprogram in π_f ,
- program of S generated as a primitive subprogram in π_f ,
- an instruction of Σ_0 not occurring in either of the above.

5.3 Suppose that every $\pi_g \in L_1$ is allowed as a possible primitive subprogram at input X . Then consider any arbitrary $f(\) \in \mathcal{R}$ and π_f computing $f(\)$. Define $g_n(X) = f(x)$ if $x \leq n$, 0 otherwise and define the sequence of programs $\pi_{g_n}(\) \in L_1$ by the program schema for $N = n$ where output of π_f is placed in Y .

- (1) $X < N \implies$ (2)
- $Y \leftarrow 0$
- $C(X, X, 3)$
- (2)

π_f
- (3) exit

Thus π_{g_n} with entrance 1 and exit 3 computes $g_n(\)$ at X

with value at Y . Each program π_{g_n} has the same segment π_f which does most of the work. It is thus easily possible to construct a main program π which given input x first generates π_{g_x} and then executes it for the value x . All that is required is that π contain some program constant which is an encoding of the schema described above. It then decodes and loads the program inserting the input value x in N . Such a program can generate π_{g_n} in a constant number of steps, say c , and execute it as primitive in one step. Thus in $c + 1$ steps relative to E_1 the function $f()$ can be computed. So clearly $f() \in E_{1,0} \subseteq E_2$.

It is not surprising that the above unrestricted method of selecting primitive subroutines breaks down because it is not in the spirit of the previous hierarchy processes, Ritchie and Cleave. Those processes were carefully controlled from below whereas generation of subprograms as used above is not. What is needed is a stronger connection between the hierarchical structure on E_ω and the use of program modification over E_ω to generate primitive subprograms. In looking for this stronger connection, the example given above can be looked at from two viewpoints.

From the vantage point of computational complexity the difficulty is that we have allowed the main program π to list members of L_1 "too fast". From the vantage point of algorithmic complexity the difficulty is that the π_{g_n} are

"improper programs". They are not purely L_1 programs because they involve subprograms like π_f which may come from a "higher language".

In Chapter 8 we will consider the algorithmic complexity viewpoint. We will see there that by selecting a subset of L_1 and restricting ourselves to programs from this subset, it becomes possible to define parameters of algorithmic complexity, $l()$ for "length" and $d()$ for "depth" and to define the class of primitive programs allowed at input X to be those whose algorithmic parameters are bounded by functions in E_1 . With such a restriction the hierarchy does not collapse at E_2 . However, a more general restriction can be described in terms of computational complexity.

5.4 Intuitively an interesting restriction on the generation of primitive subprograms would be that the speed at which primitive subprograms are generated is bounded by a previously obtained function. In other words, a function $t()$ already obtained in the hierarchy is used to predict the index i of $E_{a,i}$ to which the subprogram π_g being generated belongs.

To be precise about the position of a subprogram π_g in L_a we can use the normal elements. We know that

* if $g() \in E_a$, then $\exists \pi_g$ such that $\sigma\pi_g(X) < f_a^{(p)}(\max X)$.
The constant p has the property that $\pi_g() \in L_{a-1,i}$ for some $i \leq p$.

Thus one precise way of controlling the position of subprograms in E_a is to control the p in line *. The speed at which primitive subprograms are generated can be taken as the rate of growth of p with respect to input.

5.5 Let π_f be a program which generates subprograms π_g and suppose at input X to π the subprograms generated are denoted by π_{i_x} $i_x = 1, 2, 3, \dots, n_x$. Also let Y denote the input to any subprogram, thus for each i_x , Y may be different, but for notational simplicity we avoid writing Y_{i_x} . We then say that a subprogram π_{i_x} can be generated as primitive in π_f over L_a at input X iff $\pi_{i_x} \in L_a$ and $\exists t() \forall X \forall Y$
 $\sigma\pi_{i_x}(Y) < f_a^{(t(X))}(\max Y) \& t() \in E_a$.

It is significant that this condition must be stated in terms of the $f_a()$ rather than solely in terms of the hierarchy class indices, a, i . We will see in Chapter 8 how this condition relates to algorithmic complexity. We will discuss its intuitive significance more fully in Chapter 6. At this point we wish to examine how the RCH behaves when primitive subprograms are generated as above.

5.6 Recalling definition 5.2 of $\bar{\sigma}_p$ -computing time and definition 4.2 of the $*_a$ map we define for a set of programs P

$$\begin{aligned} \bar{L}(P)_{-1} &= C = \text{constants}, \text{ and} \\ \bar{L}(P)_i &= \{ \pi_f \mid \bar{\sigma}_P \pi_f() < (\bar{L}(P)_{i-1})^{*a} \}. \end{aligned}$$

Also define

- (i) $\bar{L}_0 = \Sigma_0$
- (ii) $\bar{L}_{b,i} = \bar{L}(\bar{L}_b)_i$
- (iii) $\bar{L}_{b+1} = \bigcup_{i=0}^{\infty} \bar{L}_{b,i}$

Put $\bar{E}_{b,i} = (\bar{L}_{b,i})^{*a}$, $\bar{E}_b = (\bar{L}_b)^{*a}$. The class \bar{E}_b form the new Ritchie-Cleave Hierarchy. Notice, we will often write σ_a -computing time for σ_{L_a} -computing time.

In this chapter we wish to compare the $\bar{E}_{b,i}$ with the $E_{b,i}$ and in the next chapter we wish to extend the \bar{E}_a beyond ω . Comparison of the hierarchy classes is easy once we have the main lemma for the $\bar{E}_{b,i}$. We turn to this lemma after the preliminary lemma 5.7.

We shall establish the convention that the notation \bar{E}_a , \bar{L}_a , $\bar{\sigma}_a$, will be used for the new RCH only when there is some possibility of confusing the new RCH with the old, i.e. with E_a , L_a , σ_a . Most often we shall simply use E_a , L_a , σ_a for the new RCH. The remaining theorems all apply to the new RCH as defined above.

5.7 Lemma. If $\pi_h()$ can be generated as primitive over E_a at X , then $\exists s() \in E_a$ such that

$$h(Y) < f_a^{(s(X))}(\max Y) \quad X \in N^n, \quad Y \in N^m.$$

Proof. By definition 5.5

$$\sigma\pi_h(Y) < f_a^{(t(X))}(\max Y).$$

Since the instructions for computing $\pi_h()$ come from Σ_0

which has normal element $f_0()$, the methods of 4.5 Main Lemma part (c) can be applied to conclude that for S computing steps needed to determine $h(Y)$

$$h(Y) < f_0^{(S + d)}(\max Y).$$

If $a = 0$, then $h()$ is a basic instruction, and the result is immediate. For $a > 1$ $f_0^{(y)}(y) < f_a(y) \quad \forall y$ so that

$$h(Y) < f_a(f_a^{(t(X) + e)}(\max Y)) = f_a^{(s(X))}(\max Y)$$

where $s(X) = t(X) + e + 1 \in E_a$ (because $t() \in E_a$ by definition).

q.e.d.

5.8 Main Lemma.

Let $f_a()$ be normal for E_a and define

$$F_{a,0}(n,y) = f_a^{(n)}(y)$$

$$F_{a,M+1}(n,y) = f_a^{(F_M(n,y))}(y)$$

Then (a) for each $M, n \quad \lambda y F_{a,M+1}(n,y) \in E_{a,M}$

(b) for each $M \quad \lambda n y F_{a,M}(n,y) \in E_{a,M}$

(c) for all $g() \in E_{a,M} \quad \exists m$ such that $X \in N^n$
 $g(X) < F_{a,M+1}(m, \max X)$

(d) for each $M, \quad \lambda x x F_{a,M}(x,x) \in E_{a,M}$ and dominates $E_{a,M-1}$.

Proof. (For simplicity we use $F_N()$ for $F_{a,N}()$ when no confusion is possible.)

Part (a) In this case $\lambda y F_1(n, y) = f_a^{(f_a^{(n)}(y))}(y)$. A subprogram $h_1()$ can be generated as primitive if

$$h_1(X) < f_a^{(f_a^{(d)}(\max X))}(\max X).$$

One subprogram which satisfies this condition is the standard program for $\lambda y F_1(n, y)$. This fact is verified in Appendix B. Thus a program for computing $\lambda y F_{a,1}(n, y)$ would first generate the program for $\lambda y F_{a,1}(n, y)$ and then execute it. To generate such a subprogram only requires determining the iteration parameter for $f_a()$. That parameter is $f_a^{(d)}(\max X)$ which can be computed using $f_a^{(d)}()$ and $\max()$ as fixed primitive subprograms (both are in E_a , $a > 0$; if $a = 0$, both can be computed in a constant number of steps). Thus total σ_a -computing time for $\lambda y F_1(n, y)$ is a constant.

For induction assume the result for M . Recall

$$F_{M+1}(n, y) = f_a^{(F_M(n, y))}(y).$$

Notice that $\lambda y F_{M+1}(n, y)$ could be computed in $F_M(n, y)$ steps from a fixed program if the value $F_M(n, y)$ were known. By using $f_a()$ as a primitive subprogram. But $\lambda y F_M(n, y)$ can be computed with a bound in $E_{a, M-2}$, so since $E_{a, M-2} \subseteq E_{a, M-1}$, $\lambda y F_M(n, y) \in E_{a, M-1}$ and $E_{a, M}$ is closed under addition

$$\sigma_a \lambda y F_{M+1}(n, y) < E_{a, M-1}.$$

thus $\lambda y F_{M+1}(n, y) \in E_{a, M}$.

q.e.d.

Part (b) $\lambda n y F_0(n, y) = f_a^{(n)}(x)$ can be computed in one step using $\lambda n y f_a^{(n)}(y)$ as an instruction. Since by Appendix B $\exists \pi_{f_a}(\)$ such that

$$\sigma_a^{(n)}(x) < f_a^{(f_a^{(d)}(\max\{n, x\}))}(\max\{n, x\}) \quad \forall n \forall x,$$

$f_a^{(n)}(\)$ can be generated as primitive. This can be done in a constant number of steps (store program schema for iteration as a constant, insert n as iteration parameter). Thus

$$\sigma_a \lambda n y F_0(n, y) = c$$

so $\lambda n y F_{a,0}(n, y) \in E_{a,0}$.

For induction assume $\lambda n y F_M(n, y) \in E_{a,M}$ and consider

$$F_{M+1}(n, y) = f_a^{(F_M(n, y))}(y).$$

Knowing $F_M(n, y)$, $F_{M+1}(n, y)$ can be computed with a fixed program in $c \cdot F_M(n, y)$ steps. Thus by the induction hypothesis

$$\sigma_a^{F_{M+1}}(n, y) = \sigma_a^{F_M}(n, y) + c \cdot F_M(n, y) < E_{a,M}.$$

So $\lambda n y F_{M+1}(n, y) \in E_{a,M+1}$.

q.e.d.

Part (c) Let π_g compute g so that $\sigma_a g(\) = c$ and suppose $h_1(\), \dots, h_p(\)$ are the instructions and fixed primitive subprograms of π_g . Let p be the maximum program constant. In addition to the fixed subprograms $h_i(\)$, π_g may generate other primitive subprograms $k_i(\)$, but by lemma 5.7 they must satisfy

$$k_i(Y) < f_a^{(f_a^{(d)}(\max X))}(\max Y) \quad X \in N^{r_i}.$$

Since $f_a()$ is normal for E_a we also have

$$h_1(Y) < f_a^{(m_1)}(\max Y) \quad Y \in N^{n_1}.$$

Let $n = \max m_1$, then by methods of 4.5 part (c) after s steps

$$* \quad g(X) < f_a^{((f_a^{(d)}(\max X) + n) \cdot s)}(\max\{X, p\}).$$

For $\sigma_a g(X) = c$ this can be reduced by methods of 4.5 to

$$g(X) < f_a^{(f_a^{(m)}(\max X))}(\max X) = F_1(m, \max X).$$

For induction we assume $\sigma_a g(X) < F_M(e, \max X)$. Proceeding exactly as above to line * and taking $s = F_M(d, \max X)$ the result is

$$g(X) < f_a^{((f_a^{(d)}(\max X) + n) \cdot F_M(d, \max X))}(\max\{X, c\}).$$

Now since $E_{a,M}$ is closed under addition and multiplication the case $M = 0$ methods yield the result

$$g(X) < f_a^{(F_M(m, \max X))}(\max X) = F_{M+1}(m, \max X) \quad X \in N^n.$$

q.e.d.

Part (d) This follows immediately from (b) and (c).

q.e.d. Main Lemma

Having established the Main Lemma in terms of the same functions, $F_{a,M}()$, as used in the Main Lemma 4.5 we can carry over the results 4.7 - 4.10 with no effort. We thus have:

5.9 (New) Hierarchy Theorem

$$(a) \quad E_{a+1} \subseteq E_{a+1,0}$$

- (b) $E_{a,i} \subset E_{a,i+1}$
- (c) $f_a(\)$ are normal for E_a
- (d) $a < b \Rightarrow E_a \subset E_b$

5.10 Composition Theorem

Let $F(\) \in E_{a,M}^{[n]}$, $g_i(\) \in E_{a,J}^{[m]}$ for $1 \leq i \leq n$. Then

$h(X) = f(g_1(X), \dots, g_n(X))$ implies

$h(\) \in E_{a,N+J}^{[m,1]}$ $X \in N^m$.

5.11 Limited Recursion Theorem

If $h(\) \in E_{a,s+1}^{[n+2]}$ and $g(\) \in E_{a,r}^{[n]}$ and there exists a

function $k(\) \in E_{a,t}^{[n+1]}$ such that

$$f(X, 0) = g(X)$$

$$f(X, y+1) = g(X, y, f(X, y))$$

$$f(X, y) \leq k(X, y) \quad X \in N^n \quad y \in N.$$

Then $f(\) \in E_{a,s+\max t,r+1}^{[n+1]}$.

We can also prove an actual time theorem for the new RCH.

5.12 Actual Time Theorem

If $g(\) \in E_{a,i}$, then $\sigma g(\) \in E_{a,i}$.

The details of this proof are given for a more general case in Chapter 6, namely in theorem 6.35.

Chapter 6 The Extended Ritchie-Cleave Hierarchy

Following our plan to investigate the possibility of basing an extension of the Ritchie-Cleave hierarchy on the concept of program self-modification, we turn now to finding adequate restrictions on the generation of primitive subprograms over E_ω in particular and over E_α in general. Example 5.3 shows the difficulties that must be avoided, but the solution of Chapter 5 is not available since E_ω does not contain a normal element (if it did, the hierarchy would not collapse at E_ω , c.f. Main Lemma 5.8). However, we can attempt to implement the idea which motivated our previous restrictions. That is, we want to use E_a to control which programs of L_a will be generated as primitive subprograms in programs constructed at level $a+1$.

Pursuing the analogy with Ritchie, we want a function $b(\) \in E_\omega$ to control the hierarchy class $E_{b(X)}$ from which primitive subprograms of a main program can be chosen for input X . But we require more than restriction to E_ω as we have already seen. Thus by analogy with what we did before, we want to select a normal element from E_a , say $f_a(\)$ $a < \omega$, and we want a $t(\)$ to control iteration of $f_a(\)$. We could then allow π_g to be generated as a primitive subprogram in π at input X if

$$6.1 \quad \sigma\pi_g(Y) < f_{b(X)}^{(t(X))}(\max Y).$$

Such a method depends on being able to choose $f_a(\)$ from E_a

in a manner which is both uniform and controlled by E_ω .

One way to make the notion of a "uniform E_a selection procedure" precise would be to ask that the procedure be a functional, $F(\)$, associated with E_ω . $F(\)$ would be required to map normal elements of E_n into normal elements of E_{n+1} , and it would be associated with E_ω in the sense that $F(g(\), X)$ would be " E_ω computable in $g(\)$ ".¹ We would then be required to show that such functionals existed. (They do.)

A simpler technique which avoids explicit use of functionals would be to first use a standard functional of the above type to select a standard sequence of normal elements

$$6.2 \quad f_{n+1}(x) = F(f_n(\), x)$$

starting with $f_0(\)$ normal for E_0 . Then define a class of admissible transformations of the standard sequence $f_n(\)$ where the transformations were controlled by E_ω .

This approach has its analogy in other branches of mathematics. An example from geometry would be the definition of a tensor product or a differential form. One can either give the definition invariantly (independent of coordinate systems) or one can pick a standard coordinate system and a group of transformations and then give the definition on

¹Relative computability with respect to the hierarchy classes is considered in Chapter 7. A stronger condition on $F(\)$ which is more in line with what we actually do is that $F(\)$ also be E_ω bounded, that is the number of steps used in computing ${}^\omega F(\), \sigma F(\),$ satisfies

$$\exists t(\) \quad \forall g(\) \quad \sigma F(g(\), X) < t(\max X) \in E_\omega.$$

the particular system, showing it is preserved under the transformations.

A standard functional is suggested by the Hierarchy Theorem 4.7. It is shown that if $f_n()$ is normal for E_n , then $f_{n+1}()$ is normal for E_{n+1} where

$$6.3 \quad f_{n+1}(x) = f_n^{(x)}(x).$$

Thus define $F(g(), x) = g^{(x)}(x)$ so that choosing $f_0() \in (\Sigma_0)_u^*$, we can generate a

6.4 standard sequence of normal elements (s.s.n.e.)

$$f_0(), f_1(), \dots, f_n(), \dots$$

To determine an appropriate class of transformations associated with E_ω and an s.s.n.e. for E_ω , we observe that given any normal element for E_a , say $h_a()$, there is a constant c_a such that

$$6.5 \quad h_a(x) < f_a^{(c_a)}(x).$$

Thus since $f_a()$ is used in 6.1 only to provide a bound, we can represent the effect of choosing any arbitrary sequence $h_a()$ of normal elements by finding a function $s(a)$ such that

$$6.6 \quad h_a(x) < f_a^{(s(a))}(x).$$

The degree of arbitrariness of $h_a()$ is reflected in $\lambda a s(a)$. Thus to control the choice of $h_a()$ we need only control $s()$. The obvious requirement is that $s() \in E_a$. Thus given an s.s.n.e. for E_ω , say $f_n()$, the suggested condition 6.1 becomes

$$6.7 \quad \sigma \pi_g(Y) < f_{b(X)}^{(s(b(X)) + t(X))}(\max Y).$$

Since E_ω is closed under composition and summation, it is

sufficient to require only $\exists t(), b() \in E_\omega$ such that

$$6.8 \quad \sigma\pi_g(Y) < f_{b(X)}^{(t(X))}(\max Y).$$

In terms of the analogy with coordinate systems and transformations, 6.1 is a reasonably natural statement of our intuitive condition on primitive subprograms. Pursuing this analogy, the question arises of whether a completely coordinate free statement of condition 6.1 is possible, i.e. whether we can avoid any reference to standard sequences of normal elements. The import of this question can be seen by recalling the relationship between the Cleave and the Grzegorzcyk hierarchies. As we discussed in Chapter 3, the Cleave hierarchy provides an invariant significance to the Grzegorzcyk hierarchy in terms of computing theoretic concepts. The question now is whether there is a similar way to avoid explicit use of the $f_n()$ in defining the ERCH. We shall see below that this question has two parts. Can we eliminate reference to the normal elements? Can we eliminate reference to standard sequences? We answer both questions affirmatively, the first in Chapter 7 and the second in Chapter 8.

Let us now consider extending 6.1 to E_α for $\alpha > \omega$. The problem is to define the notion of an s.s.n.e. for E_α . By analogy with the ω case, the obvious definition is that we take a fundamental sequence β_n for α , i.e. write $\beta_n \rightarrow \alpha$ if β_n is a sequence of ordinals whose limit is α . Define a β_n -sequence of normal elements for E_α as that sequence

generated by some normal element $f_0()$ applying iteration and diagonalization. That is,

$$6.10 \quad (a) \quad f_{\alpha+1}(x) = f_{\alpha}^{(x)}(x) \text{ and}$$

$$(b) \quad f_{\alpha}(x) = f_{\beta_x}(x) \text{ for } \alpha \text{ a limit ordinal.}$$

The generalization of 6.1 then becomes

$$6.11 \quad \sigma\pi_g(Y) < f_{\beta_{b(X)}}^{(t(X))}(\max Y) \quad \text{for } t(), b() \in E_{\alpha}$$

where $\beta_x \rightarrow \alpha$.

However, if the concept of an s.s.n.e. is to retain its intuitive significance, we must restrict α to the constructive ordinals and must require the fundamental sequences to be effective. The following example illustrates what can happen if the fundamental sequences are not controlled.

6.12 Let $h()$ be an arbitrary element of \mathcal{R} and let $g()$ be a strictly increasing member of \mathcal{R} such that for some π_h

$$\sigma\pi_h(x) < g(x) \quad \forall x.$$

Then $g(n) \rightarrow \omega$ so $f_{g(n)}()$ is a sequence of normal elements, and since $f_n(x) > n$ for all normal elements (see 6.17) it follows that

$$f_{g(x)}(x) > g(x),$$

thus if $f_{g()}()$ were allowed in condition 6.11, then taking $t(x) = 1$ and $b(x) = x$, $\pi_h()$ could be introduced as a primitive subprogram.

In light of the above example, we must introduce some further standardization. In particular we must define the notion of a standard fundamental sequence in a manner con-

sistent with the motivation behind condition 6.1. That is, how can we give invariant significance to such a definition?

We recall from Chapter 1 that the matter of standard fundamental sequences was crucial to the basic subrecursive hierarchies, e.g. ordinal recursion, Extended Grzegorzczuk, and the Kleene subrecursive hierarchy. Cleave avoids the problem in his hierarchy because he does not use the ordinals in an essential way. They serve only to index his generation process. We shall proceed in a like manner, but for several reasons which will become clear later, we present the extension first in terms of ordinals and a standard fundamental sequence. It is to this task that we now turn. Then in Chapter 7 we consider a justification for these sequences.

6.13 We define fundamental sequences for $\alpha < \epsilon_0$; this can be done in a very straightforward manner. Define $\alpha_n =$ fundamental sequence for α as follows. First put α in its normal form to the base ω (see Sierpinski[53] p. 323) so $\alpha = \omega^{\beta_1} \cdot a_1 + \dots + \omega^{\beta_n} \cdot a_n$ where $\beta_1 > \beta_2 > \dots > \beta_n$, $a_i \in \mathbb{N} - 0$. This normal form is unique.

- (a) if $\alpha = \beta + \gamma$, then $\alpha_n = \beta + \gamma_n$
- (b) if $\alpha = \beta \cdot n$, then $\alpha_n = \beta \cdot (n-1) + \beta_n$
- (c) if $\alpha = \omega^\beta$, then we consider two subcases:
 - (i) if β is a limit ordinal, then $\alpha_n = \omega^{\beta_n}$
 - (ii) if β is a successor ordinal, then $\alpha_n = \omega^{(\beta-1)} \cdot n$ (recall $\omega^0 = 1$).

We will usually write $\alpha_n \rightarrow \alpha$. Thus, for example, $n \rightarrow \omega$, $\omega^2 + n \rightarrow \omega^2 + \omega$, $\omega \cdot n \rightarrow \omega^2$, $\omega^n \rightarrow \omega^\omega$, $\omega^{\omega \cdot n} \rightarrow \omega^{\omega^2}$, etc.

6.14 Given any function $\lambda x h(x)$ we define the standard sequence of functions generated from $h()$ as follows. Let $\alpha < \epsilon_0$

- (a) $h_0(x) = h(x)$
- (b) $h_{\alpha+1}(x) = h_{\alpha}^{(x)}(x)$
- (c) $h_{\alpha}(x) = h_{\alpha_x}(x)$ if α is a limit ordinal.

We can extend this sequence of functions beyond ϵ_0 if we have a method of choosing standard fundamental sequences. Whenever we have something we want to call a standard fundamental sequence for all ordinals $\alpha < \beta$, then we can define s.s. derived from $h()$ for $\alpha < \beta$.

6.15 Recall the definition of a normal element for Σ in the Cleave hierarchy. We will say that $f_n()$ is normal for E_n . If $f()$ is a normal function for $\Sigma_0 = \{ +, -, \times, TC \}$ we call any standard sequence generated from $f()$ a standard sequence of normal elements (s.s.n.e.). The reason for this wording is that $f_{\alpha}()$ are normal elements for the E_{α} of the Extended Cleave-Ritchie Hierarchy (ERCH). We show this fact in 6.31.

6.16 Given ordinals α and β in base ω representation we say that $\alpha + \beta$ is canonical if $\alpha + \beta$ is the base ω representation of the sum. For example $\omega^2 + \omega$ is canonical but $\omega + \omega^2$ is not. We now state and prove some basic facts about the $f_{\alpha}()$.

6.17 $f_\alpha(x) > x \quad \forall x \quad \forall \alpha.$

6.18 If $0 < \beta < \alpha$ and $\alpha + \beta$ is canonical, then $f_\alpha(x) < f_{\alpha+\beta}(x)$
 $x > 0$ and $f_\alpha(x) \leq f_{\alpha+\beta}(x) \quad \forall x.$

6.19 If $\gamma_1 + \beta$ and $\gamma_2 + \beta$ are canonical and $f_{\gamma_1}(x) < f_{\gamma_2}(x)$
 $x > M$, then $f_{\gamma_1+\beta}(x) < f_{\gamma_2+\beta}(x) \quad x > M.$

6.20 $\alpha > 0$ implies $f_0(x) < f_\alpha(x) \quad x > 0$ and $f_0(x) < f_\alpha(x)$
 $\forall x.$

6.21 If $\alpha + \beta$ is canonical, then $f_\beta(x) < f_{\alpha+\beta}(x) \quad x > 0.$

6.22 If α is a limit ordinal $< \epsilon_0$ and $\alpha_n \rightarrow \alpha$, then
 $f_{\alpha_n}(x) < f_{\alpha_{n+1}}(x) \quad x > 1.$

6.23 $f_\alpha(x)$ is strictly increasing for $\alpha < \epsilon_0 \quad x > 1.$

6.24 If $m(\alpha)$ = the maximum of the integers in the base
 ω representation of α , then $\alpha < \beta$ implies $f_\alpha(x) < f_\beta(x)$
 $\forall x > m(\alpha).$

6.25 For $\alpha < \beta < \epsilon_0$ let $mc(\alpha)$ = the maximum coefficient in
the base ω representation of α , then $f_\alpha(x) < f_\beta(x)$
 $\forall x > mc(\alpha).$

We now turn to the proofs of these statements.

6.17 $f_\alpha(x) > x \quad \forall x \quad \forall \alpha.$

Proof. To say $f_0(\)$ is normal means that it is strictly increasing. Clearly no strictly increasing function can satisfy $f(x) < x$ for any x , thus we already know $f_0(x) \geq x \quad \forall x$. Suppose $f_0(x_0) = x_0$. then $f_0(y) = y \quad \forall y \leq x_0$ and $f_0^{(m)}(y) = y \quad \forall m \quad \forall y \leq x_0$, but this is impossible since $f_0(\)$ is normal for $\{+, -, x, \}$ and taking $\lambda x g(x) = 2x$ there must

exist m such that $2x < f_0^{(m)}(x) \quad \forall x$. Thus $f_0(x) > x$ for all x . We now show that $f_\alpha(x) > x \quad \forall x \quad \forall \alpha$. We proceed by induction having established the proposition for $\alpha = 0$.

Suppose the result for α , we then show that $f_\alpha^{(m)}(x) > x \quad \forall x \quad \forall m$. To this end, assume $f_\alpha^{(m)}(x) > x \quad \forall x$. Then

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

by induction hypothesis on m and on α , So

$$f_{\alpha+1}(x) = f_\alpha^{(x)}(x) > x \quad \forall x.$$

Now suppose α is a limit ordinal with $\alpha_n \rightarrow \alpha$, then since $\alpha_n < \alpha$ we have by the induction hypothesis that $f_{\alpha_n}(x) > x$. So

$$f_\alpha(x) = f_{\alpha_n}(x) > x \quad \forall x.$$

q.e.d.

Notice that this result does not depend on any particular choice of fundamental sequence nor on any bound on α .

6.18 If $0 < \beta < \alpha$ and if $\alpha + \beta$ is canonical, then $f_\alpha(x) < f_{\alpha+\beta}(x) \quad x > 0$ and $f_\alpha(x) \leq f_{\alpha+\beta}(x) \quad \forall x$.

Proof. By induction on α and β .

1. Suppose α is a successor ordinal, then for $\alpha + \beta$ to be canonical, β must be an integer. Consider $\beta = 1$, by definition

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

and we notice since $f_\alpha(x) > x$

$$f_\alpha^{(m)}(x) > f_\alpha(x) \quad \forall x$$

$$(f_\alpha(f_\alpha(x))) > f_\alpha(x)$$

so

$$f_{\alpha}^{(m)}(x) > f_{\alpha}^{(m-1)}(x) > \dots > f_{\alpha}(x) \quad \forall x.$$

Thus

$$f_{\alpha}(x) < f_{\alpha+1}(x) \quad \forall x.$$

Now taking $\alpha+1$ for α

$$f_{\alpha}(x) < f_{\alpha+1}(x) < f_{\alpha+2}(x) < \dots < f_{\alpha+\beta}(x) \quad \forall x.$$

So we have the result for β .

2. Suppose α is a limit ordinal, for induction on β assume result for β then we show result for $\beta+1$ and for $\beta+1$ exactly as above taking α and $\alpha+\beta$ respectively as α in 1. Assume now that β is a limit ordinal, $\beta_n \rightarrow \beta$. By induction hypothesis

$$f_{\alpha}(x) < f_{\alpha+\beta_n}(x) \quad \forall x \text{ if } \beta_n > 0.$$

So

$$f_{\alpha}(x) < f_{\alpha+\beta_x}(x) = f_{\alpha+\beta}(x) \text{ if } \beta_x > 0.$$

We need that $\alpha+\beta$ is canonical to conclude that $\alpha+\beta_x \rightarrow \alpha+\beta$.

We observe that if $x > 0$ then $\beta_x > 0$ for all fundamental sequences for $\alpha < \epsilon_0$. If $\beta=0$, then we get the \leq result.

q.e.d.

6.19 If $\gamma_1+\beta$ and $\gamma_2+\beta$ are canonical and $f_{\gamma_1}(x) < f_{\gamma_2}(x)$ $\forall x > M$, then $f_{\gamma_1+\beta}(x) < f_{\gamma_2+\beta}(x)$ $x > M \geq 0$.

Proof. We are merely starting the standard sequences with two different bases. The proof is by induction on β .

Suppose

$$f_{\gamma_1+\beta}(x) < f_{\gamma_2+\beta}(x) \quad \forall x > M$$

then we show

$$f_{\gamma_1+\beta}^{(m)}(x) < f_{\gamma_2+\beta}^{(m)}(x) \quad \forall m \quad \forall x > M.$$

For $m = 1$ the result is clear. But also

$$f_{\gamma_1+\beta}(f_{\gamma_1+\beta}^{(m)}(x)) < f_{\gamma_2+\beta}(f_{\gamma_1+\beta}^{(m)}(x)) < f_{\gamma_2+\beta}(f_{\gamma_2+\beta}^{(m)}(x)) \\ \forall x > M$$

which follows by induction hypothesis on β , assuming $f_{\gamma_2+\beta}(\)$ is strictly increasing, and by induction hypothesis on m (since $f_{\gamma_1}(x) > x \quad \forall x$ we can assume strict increasing for $x > 1$ as we have in 6.21). Thus the result.

Suppose now that β is a limit ordinal, $\beta_n \rightarrow \beta$. Then

$$f_{\gamma_1+\beta_n}(x) < f_{\gamma_1+\beta_n}(x) \quad x > M.$$

But

$$f_{\gamma_1+\beta}(x) = f_{\gamma_1+\beta_n}(x) < f_{\gamma_2+\beta_n}(x) = f_{\gamma_2+\beta}(x) \quad x > M.$$

We use that $\gamma_1+\beta$ and $\gamma_2+\beta$ are canonical in this last step to obtain the proper fundamental sequences.

q.e.d.

Notice we have proved this statement under the assumption that 6.21 holds for γ_1, γ_2 . This causes a technical difficulty with 0 which accounts for the hypothesis $x > M$. It also causes us to be careful in the proof of 6.21 that we do not use this statement for ordinals γ_1, γ_2 for which $f_{\gamma_1}(\)$, $f_{\gamma_2}(\)$ have not already been shown to be increasing.

q.e.d.

6.20 If $\alpha > 0$ then $f_0(x) < f_\alpha(x)$ $x > 0$ and $f_0(x) \leq f_\alpha(x) \forall x$.

Proof. First consider $\alpha=1$. Notice by induction

$$f_0(x) < f_0^{(m)}(x).$$

In particular $m=1$ is clear and

$$f_0(x) < f_0(f_0^{(m)}(x))$$

since

$$f_0^{(m)}(x) > x \quad \forall x \quad \forall m \text{ (see 3.2)}$$

and $f_0(\cdot)$ is strictly increasing.

Thus

$$f_0(x) < f_0^{(m+1)}(x)$$

hence

$$f_0(x) < f_0^{(x)}(x) = f_1(x) \quad x > 0.$$

Now suppose result holds for α , to know

$$f_0(x) < f_{\alpha+1}(x)$$

we notice from 6.18

$$f_0(x) < f_\alpha(x) < f_{\alpha+1}(x) \quad x > 0.$$

Suppose α is a limit ordinal, $\alpha_n \rightarrow \alpha$, then again as in above

$$f_0(x) < f_{\alpha_n}(x) \quad x > 0$$

$$f_0(x) < f_{\alpha_x}(x) = f_\alpha(x) \quad x > 0.$$

q.e.d.

6.21 Corollary. If $\alpha+\beta$ is canonical, then $f_\beta(x) < f_{\alpha+\beta}(x)$ $x > 0$.

Proof. Take $\gamma_1=0$, $\gamma_2=\alpha$, $M=0$ in 6.19.

6.22 If α is a limit ordinal and $\alpha_n \rightarrow \alpha$, then $f_{\alpha_n}(x) < f_{\alpha_{n+1}}(x)$ $x > 1$, $\alpha < \epsilon_0$.

Proof. To prove this we proceed by offering a tedious analysis of the form of the terms α_n in a fundamental sequence. We see from our definition of fundamental sequence that every term α_n is of the form

$$\alpha_n = \gamma_1 + \omega^{\gamma_2 + \omega^{\gamma_3 + \omega, \dots, \gamma_t + \omega^n}}$$

or

$$\alpha_n = \gamma_1 + \omega^{\gamma_2 + \omega^{\gamma_3 + \omega, \dots, \gamma_t + \omega^p \cdot n}}$$

For example, in $\omega^2 \cdot n$ we have $\gamma_1 = \gamma_2 = 0$ and $p = 0$; in $\omega^{\omega \cdot 2} + \omega^2 \cdot n$, $\gamma_1 = \gamma_2 = \omega^{\omega \cdot 2}$, $p = 2$; in $\omega^{\omega^{\omega} + \omega^{\omega^4 \cdot 2} + \omega^3 \cdot n}$, $\gamma_1 = \omega^{\omega^{\omega}}$, $\gamma_2 = \gamma_t = \omega^{\omega^4 \cdot 2}$, $p = 3$. First we analyse terms where $\gamma_1 = \gamma_t$, e.g. $\gamma + \omega^p \cdot n$. We wish to show

$$f_{\gamma + \omega^p \cdot n}(x) < f_{\gamma + \omega^p \cdot (n+1)}(x) \quad x > 1.$$

By definition

$$\begin{aligned} f_{\gamma + \omega^p \cdot (n+1)}(x) &= f_{\gamma + \omega^p \cdot n + \omega^{p-1} \cdot x - 1 + \dots + \omega \cdot x}(x) \\ &= f_{\gamma + \omega^p \cdot n + \omega^{p-1} \cdot x - 1 + \dots + x}(x). \end{aligned}$$

We notice that for $x > 0$

$$\gamma + \omega^p \cdot n < \gamma + \omega^p \cdot n + \omega^{p-1} \cdot x - 1 + \dots + x.$$

So we can apply 6.18 to conclude

$$f_{\gamma+\omega^p \cdot n}(x) < f_{\gamma+\omega^p \cdot n+1}(x) \quad x > 0.$$

We must now examine the more complex case where $t > 1$.

First consider a simple example, $\alpha = \omega^{\omega^2}$, $\alpha_n = \omega^{\omega \cdot n}$.

To show

$$f_{\omega^{\omega \cdot n}}(x) < f_{\omega^{\omega \cdot n+1}}(x) \quad x > 1$$

consider

$$f_{\omega^{\omega \cdot n+1}}(x) = f_{\omega^{\omega \cdot n+x}}(x) = f_{\omega^{\omega \cdot n+x-1} \cdot x}(x)$$

which in turn is equal to $f_{\theta}(x)$ where $\theta =$

$$\omega^{\omega \cdot n+x-1} \cdot x-1+\omega^{\omega \cdot n+x-2} \cdot x-1+\dots+\omega^{\omega \cdot n+1} \cdot x-1+\omega^{\omega \cdot n} \cdot x-1+\omega$$

We observe that for $x > 0$ we have

$$f_{\omega^{\omega \cdot n+1}}(x) = f_{\beta_x + \omega^{\omega \cdot n}}(x) \quad \beta_x > 0 \text{ if } x > 1$$

and furthermore $\beta_x > \omega^{\omega \cdot n}$ for $x > 1$. Thus by 6.18

$$f_{\omega^{\omega \cdot n}}(x) < f_{\beta_x + \omega^{\omega \cdot n}}(x) = f_{\omega^{\omega \cdot n+1}}(x) \quad x > 1.$$

We want to show for $t > 1$ and $x > 1 \exists \beta_x$ such that if

$$\delta_{n+1} = \gamma_1 + \omega \gamma_2 + \omega \dots \gamma_t + \omega^{\omega \cdot n+1}$$

and

$$\delta_n = \gamma_1 + \omega \gamma_2 + \omega \dots \gamma_t + \omega^{\omega \cdot n}$$

then

$$f_{\delta_{n+1}}(x) < f_{\beta_x + \delta_n}(x)$$

for $\beta_x > \delta_n$.

First we examine the case

$$f_{\gamma_2 + \omega^{\gamma_2 + \omega^{p \cdot n + 1}}}(x).$$

Observe that

$$f_{\gamma_1 + \omega^{\gamma_2 + \omega^{p \cdot n + \omega^{p-1} \cdot x}}}(x)$$

is equal to

$$f_{\gamma_1 + \omega^{\gamma_2 + \omega^{p \cdot n + \omega^{p-1} \cdot x - 1 + \dots + \omega \cdot x - 1 + x}}}(x)$$

which is equal to

$$f_{\gamma_1 + \omega^{\gamma_2 + \omega^{p \cdot n + \omega^{p-1} \cdot x - 1 + \dots + \omega \cdot x - 1 + x - 1} \cdot x}}(x)$$

which for

$$\delta_1 = \omega^{\gamma_2 + \omega^{p \cdot n + \dots + x - 1}}$$

$$\delta_2 = \omega^{\gamma_2 + \omega^{p \cdot n + \dots + x - 2}}$$

$$\lambda = \omega^{\gamma_2 + \omega^{p \cdot n + \dots + \omega \cdot x - 1}}$$

is equal to

$$f_{\gamma_1 + \delta_1 \cdot x - 1 + \delta_2 \cdot x - 1 + \dots + \lambda \cdot x}(x)$$

which is equal to

$$f_{\gamma_1 + \beta_x + \omega^{\gamma_2 + \omega^{p \cdot n}}}(x)$$

where

$$\beta_x = \omega^{\gamma_2 + \omega^{p \cdot n + \dots + x - 1} \cdot x - 1 + \omega^{\gamma_2 + \dots + \omega} \cdot x - 1 + \dots + \omega^{\gamma_2 + \omega^{p \cdot n}} \cdot x - 1}$$

At each step we apply the rule for obtaining the fundamental sequence, and we observe that the term $\omega^{\gamma_2 + \omega \cdot n}$ appears as a summand. So we have

$$f_{\gamma_1 + \omega^{\gamma_2 + \omega^{p \cdot n}}}(x) < f_{\gamma_1 + \beta_x + \omega^{\gamma_2 + \omega^{p \cdot n}}}(x) \quad x > 1$$

by the principles of 6.18. So finally the left hand side of the above is less than

$$f_{\gamma_1 + \omega^{\gamma_2 + \omega^{p \cdot n+1}}}(x) \quad x > 1$$

as was to be shown.

We must now observe that these same principles apply to give the result for any t . If we put

$$\theta_n = \gamma_1 + \omega \cdots \gamma_{t-2} + \omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n+1}}}$$

and

$$\theta'_n = \gamma_1 + \omega \cdots \gamma_{t-2} + \omega^{\gamma_{t-1} + \beta_x + \omega^{\gamma_t + \omega^{p \cdot n}}}$$

and if $f_{\theta_n}(x) = f_{\theta'_n}(x)$, then β_x contains a term of the form

$$\omega^{\gamma_t + \omega^{p \cdot n + \dots + x-1}} \cdot x-1 \text{ as a summand. The reduction of}$$

$f_\alpha(x)$ at the γ_{t-1} level will result in a term of the form

$$\gamma_{t-2} + \beta_x^2 + \omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n}}}$$

where β_x^2 contains a summand of the form

$$\omega^{\gamma_{t-1} + \beta_x + \gamma_t + \omega^{p \cdot n + 1 + \dots + x - 1} \cdot x - 1 + \dots + x - 1} \cdot x - 1.$$

This will happen because in reducing

$$\omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n}}} \text{ terms of the form } \omega^{\gamma_{t-1} + \alpha} \cdot x \text{ will}$$

appear for each term α in the exponent of ω and for each term obtainable by a decrement of the integers in the proper order. Since β_x contains a term of the form

$$\omega^{\gamma_t + \omega^{p \cdot n + \dots + x - 1}} \text{ the term } \omega^{\gamma_t + \omega^{p \cdot n}} \text{ will occur as a}$$

result of decrement of the integers (first the coefficient gets reduced to 1, then the summands get reduced to 0). Thus

$$\omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n}}} \text{ will appear. Noting that}$$

$$\omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n}}} \cdot x - 1 + \omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n}}} = \omega^{\gamma_{t-1} + \omega^{\gamma_t + \omega^{p \cdot n}}} \cdot x$$

gives the result as claimed.

We can now obtain this result for all t by an induction on t and the forms presented. This leaves the case with terms of the form

$$\gamma_1 + \omega^{\gamma_2 \dots \gamma_{t-1} + \omega^{\gamma_t + \omega^n}}.$$

This case proceeds in an entirely similar manner to the last

6.23 Corollary. If $1 < x < y$, then $f_\alpha(x) < f_\alpha(y)$. Also

if $x < y$, then $f_\alpha(x) < f_\alpha(y)$ $x < y$. In each case $\alpha < \epsilon_0$.

Proof. By induction on α . Suppose the result holds for α . Recall that $f_{\alpha+1}(x) = f_\alpha^{(x)}(x)$ and let $x < y$. The induction hypothesis gives $f_\alpha(x) < f_\alpha(y)$. We get $f_\alpha^{(m)}(x) < f_\alpha^{(m)}(y)$ for all m by induction on m . Using the induction hypothesis and the fact that $f_\alpha(\)$ is strictly increasing it follows that

$$f_\alpha^{(m+1)}(x) = f_\alpha(f_\alpha^{(m)}(x)) < f_\alpha(f_\alpha^{(m)}(y)) = f_\alpha^{(m+1)}(y).$$

Applying this to the definition of $f_{\alpha+1}(\)$ we get

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

For the limit case, $f_\alpha(\)$ with $\alpha_n \rightarrow \alpha$ we must show

$$f_{\alpha_0}(0) < f_{\alpha_1}(1) < f_{\alpha_2}(2) < \dots < f_{\alpha_n}(n) < \dots$$

But this follows directly from 6.22.

q.e.d.

6.24 Let $m(\alpha)$ = maximum integer appearing in the base ω representation of α . If $\alpha < \beta$ and $m(\alpha) = m$, then

$$f_\alpha(x) < f_\beta(x) \text{ for all } x \geq m+1.$$

Proof. By induction on β . For $\beta = 0$ there is nothing to prove. Suppose the result holds for β , then since

$$f_\beta(x) < f_{\beta+1}(x) \text{ } x > 0, \text{ the result holds for } \beta+1.$$

Suppose β is a limit ordinal, $\beta_n \rightarrow \beta$. If $\alpha < \beta$, then $\alpha < \beta_n$ for some n . So $f_\alpha(x) < f_{\beta_n}(x)$ for $x > m+1$.

If we now had $n \leq m+1$ we could say

$$f_\alpha(x) < f_{\beta_x}(x) \quad x > m+1$$

and we would be finished. All we need is that for each $\alpha < \beta$ we can find n such that $\alpha < \beta_n$ for $n \leq m(\alpha)+1$. Call such fundamental sequences admissible for the representation defining $m(\cdot)$. We now show that all fundamental sequences for $\beta < \epsilon_0$ are admissible. Consider $\beta < \omega^\omega$. Say

$$\beta = \omega^{n_1} \cdot b_1 + \omega^{n_2} \cdot b_2 + \dots + \omega^{n_s} \cdot b_s$$

and given $\alpha < \beta$ ($b_s \neq 0$) where

$$\alpha = \omega^{m_1} \cdot a_1 + \dots + \omega^{m_p} \cdot a_p$$

let $m(\alpha) = m$ for α and β in unique base ω representation.

Let $\omega^{n_1} \cdot b_1$ be the first term in β such that

$$\omega^{m_i} \cdot a_i < \omega^{n_i} \cdot b_i.$$

Then if $i \neq s$, clearly $\alpha < \beta_{n_x} \forall x$. Thus we need only consider the case of $\beta = \omega^n \cdot b$ and $\alpha = \omega^m \cdot a$ with $\alpha < \beta$. If $m < n$ and $b > 1$ then clearly $\alpha < \beta_x \forall x$. Thus we need only consider $\omega^m \cdot a < \omega^n$. If $m+1 < n$, then again $\alpha < \beta_x \forall x$. So consider $\omega^m \cdot a < \omega^{m+1}$. Then $\beta_x = \omega^m \cdot x$ so that $\omega^m \cdot a < \beta_x$ if $x > m(\alpha) \geq a$. So if $x \geq m(\alpha)+1$ the result follows.

We handle $\beta \geq \omega^\omega$ in levels according to the power of ω . ω^n for $n < \omega$ is of level 0, ω^ω of level 1, and if β is of level n then ω^β is of level $n+1$. At level $n+1$ are those β of the form $\omega^{\beta_1} \cdot b_1 + \dots + \omega^{\beta_s} \cdot b_s$ where β_i are of level n . Since level 0 ordinals are the β 's $< \omega^\omega$, we have proved the result for level 0. Now assume it for level n . Let

$$\exists t() \forall X \forall Y \sigma_{1_x}(Y) < f_{\alpha_{b(X)}}^{(t(X))}(\max Y)$$

and

$$t(), b() \in E_{\alpha}.$$

6.27 Definition 5.2 can now be extended to $\alpha \geq \omega$. We write σ_{α} for $\bar{\sigma}_{L_{\alpha}}$ of 5.2.

A hierarchy can now be defined as in 5.6 by taking α for b in conditions (b) and (c) and adding

$$(d) \quad L_{\alpha} = \bigcup_{\beta < \alpha} L_{\beta} \text{ if } \alpha \text{ is a limit ordinal.}$$

However in this hierarchy \bar{L}_{ω} is not normal so that the main lemma does not hold. What we need at limit ordinals is the following modification.

$$6.28 \quad \mathcal{D}(\bar{L}_{\alpha}) = \begin{cases} \bar{L}(\bar{L}_{\alpha})_0 & \text{if } \alpha \text{ is a limit ordinal} \\ \bar{L}_{\alpha} & \text{otherwise} \end{cases}$$

6.29 We present the ERCH for $\alpha < \epsilon_0$.

$$(a) \quad \bar{L}_0 = \Sigma_0$$

$$(b) \quad \bar{L}_{\alpha, i} = \bar{L}(\mathcal{D}(\bar{L}_{\alpha}))_i$$

$$(c) \quad \bar{L}_{\alpha+1} = \bigcup_{i=0}^{\infty} \bar{L}_{\alpha, i}$$

$$(d) \quad \bar{L}_{\alpha} = \bigcup_{\beta < \alpha} \bar{L}_{\beta} \text{ if } \alpha \text{ is a limit ordinal}$$

Also put $\bar{E}_{\alpha, i} = (\bar{L}_{\alpha, i})^{*a}$, $\bar{E}_{\alpha} = (L_{\alpha})^{*a}$ and $\mathcal{D}(\bar{E}_{\alpha}) = \mathcal{D}(\bar{L}_{\alpha})^{*a}$.

As in Chapter 5 we will delete the bar and use E_{α} , L_{α} to denote the hierarchy classes when no confusion with RCH is possible. Before we can assert that the hierarchy is well defined we must show that $\mathcal{D}(E_{\alpha})$ is normal for all $\alpha < \epsilon_0$.

Relying heavily on the properties 6.13 - 6.25 of the $f_{\alpha}()$ we show that the ERCH is well defined (6.31) and we

$$\beta = \omega^{\beta_1} \cdot b_1 + \omega^{\beta_2} \cdot b_2 + \dots + \omega^{\beta_s} \cdot b_s$$

$$\alpha = \omega^{\alpha_1} \cdot a_1 + \omega^{\alpha_2} \cdot a_2 + \dots + \omega^{\alpha_p} \cdot a_p$$

and again check for the first terms which determine the ordering. Given

$$\omega^{\alpha_i} \cdot a_i < \omega^{\beta_i} \cdot b_i,$$

if $\alpha_i + 1 < \beta_i$ and $b_i > 1$ then $\alpha < \beta_n \quad \forall n$.

If $\alpha_i + 1 < \beta_i$ and $b_i = 1$ then, if β_i is a successor ordinal we have $\alpha < \beta_n \quad \forall n$. If β_i is a limit ordinal, then using the induction hypothesis on levels, $\alpha_i < \beta_i$, we know

$\beta_n = \omega^{\beta_{i_n}}$. So $\alpha_i < \beta_{i_n}$ for $n > m(\alpha_i)$. Thus since $m(\alpha) \geq m(\alpha_i)$, we have $\alpha < \beta_n \quad n > m(\alpha)$. If $\beta_i = \alpha_i + 1$, then $b_i > a_i$ implies $\alpha < \beta_n \quad \forall n$. If $b_i = 1$, then $\beta_n = \omega^{\alpha_i} \cdot n$, and for $n > m(\alpha) > a_i$ we have $\alpha < \beta_n$.

q.e.d.

6.25 The proof occurred as a special case of the proof of 6.24.

6.26 We now present the precise definitions for the ERCH. Let π_f be a program which generates subprograms π_g and suppose at input X to π_f the subprograms generated are denoted by $\pi_{i_x} \quad i_x = 1, 2, \dots, n_x$. Also let Y denote the input to any subprogram, thus for each i_x Y may be different but for notational simplicity we avoid writing Y_{i_x} . We then say that a subprogram π_{i_x} can be generated as primitive in π over L at X iff

exhibit its main features in parallel to those of RCH.

First appears a lemma giving us control over the value of functions in $\mathcal{D}(E_\alpha)$.

6.30 Lemma. If $g(\) \in \mathcal{D}(E_\alpha)$, then $\exists t_2(\), b_2(\) \in E_\alpha$ such that

$$g(X) < f_{\alpha_{b_2}}^{(t_2(X))}(\max X)$$

To say that $g(\) \in \mathcal{D}(E_\alpha)$ is to say there is a program such that $\sigma_\alpha \pi_g(X) < c$. Let $h_1(\), \dots, h_p(\)$ be the fixed subprograms and instructions of π_g . They must come from L_α , thus there exist v_i and n_i such that

$$h_1(Y) < f_{\alpha_{n_1}}^{(v_1)}(\max Y) \quad Y \in N^{m_1}$$

If $h_j(\)$ is generated as primitive, then $\exists t(\), b(\)$ such that

$$h_j(Y) < f_{\alpha_{b(X)}}^{(t(X))}(Y) = S(X, Y) \quad Y \in N^{m_j}.$$

The subprogram $h_j(\)$ is composed of instructions from Σ_0 so that after S steps at input Y

$$h_j(Y) < f_0^{(S)}(\max Y).$$

Thus from the above

$$h_j(Y) < f_0^{(S(X, Y))}(\max Y),$$

and since $\alpha_{b(X)+1} < 0$

$$f_0^{(f_{\alpha_{b(X)}}^{(t(X))}(\max Y))}(\max Y) < f_{\alpha_{b(X)+1}}^{(f_{\alpha_{b(X)}}^{(t(X))}(\max Y))}$$

$$< f_{\alpha(b(X)+1)}^{(t(X)+1)}(\max Y).$$

Now take $M = \max n_i$ and $V = \max v_i$, then

$$h_1(Y) < f_{\alpha(b(X)+M+1)}^{(t(X)+V+1)}(\max Y) \quad Y \in N_1^m, \quad X \in N^n.$$

Thus after $c \cdot \bar{\sigma}_\alpha$ -steps the maximum value which can be produced is

$$f_{\alpha(b(X)+M+1)}^{(d \cdot (t(X)+V+1))}(\max X) \quad \text{for } d > c \text{ depending on the program constants.}$$

So taking $t_2(X) = d \cdot (t(X)+V+1)$ and $b_2(X) = b(X)+M+1$ we have the result.

q.e.d. lemma

6.31 Theorem. $\mathcal{D}(E_\alpha)$ is a normal set of functions and if $f_\alpha(\)$ is an s.s.n.e. for E_α , then $f_\alpha(\)$ is a normal element for $\mathcal{D}(E_\alpha)$.

Proof. We must show the following:

- (1) any s.s.n.e., $f_\alpha(\)$, belongs to $\mathcal{D}(E_\alpha)_u^*$
- (2) $f_\alpha(\)$ is strictly increasing
- (3) any s.s.n.e., $f_\alpha(\)$, majorizes $\mathcal{D}(E_\alpha)_u^*$.

(1) We in fact show that $f_\alpha(\) \in \mathcal{D}(E_\alpha)$. This work is quite involved and has consequently been relegated to Appendices A and B. It enters the proof here as

6.32 Lemma. $f_\alpha(\) \in \mathcal{D}(E_\alpha) \quad \forall \alpha \quad 0 < \alpha < \varepsilon_0$.

For α a non-limit ordinal greater than 0, the proof will be covered by the Main Lemma as it is for $\alpha < \omega$. We are interested only in the case that α is a limit ordinal. Then

$\mathcal{D}(E_\alpha) = E(E_\alpha)_0$. To show $f_\alpha(\) \in E(E_\alpha)_0$ we must show that there exists a program π_{f_α} such that

$$\sigma_{\alpha} \pi_{f_\alpha}(x) < c \quad x \in N.$$

By definition $f_\alpha(x) = f_{\alpha_x}(x)$ for $\alpha_x \rightarrow \alpha$ so an obvious way to compute $f_\alpha(\)$ would be to introduce $\pi_{f_{\alpha_x}}(\) \in L_{\alpha_x}$ as a primitive subroutine. This could only be done if $\exists t(\), b(\) \in E_\alpha$ such that

$$* \quad \sigma \pi_{f_{\alpha_x}}(y) < f_{\alpha_{b(x)}}^{(t(x))}(y) \quad \forall y \quad \forall x \in N$$

and if there were a program π_M which satisfied

$$(a) \quad \bar{\sigma}_\alpha \pi_M(x) < c$$

$$(b) \quad \pi_M(x) \text{ generates } \pi_{f_\alpha} \text{ as a subprogram.}$$

In Appendix A we present a computing procedure for $f_\alpha(\)$ and we exhibit a π_M , which satisfies (a) and (b). It in fact satisfies (a') $\pi_M \in L_2$. We also prove *.

Returning to the proof of 6.31 we consider case (2). $f_\alpha(\)$ is strictly increasing is known from 6.23.

(3) We consider the following subcases:

$$(a) \quad g(\) \in E_\alpha$$

$$(b) \quad g(\) \in \mathcal{D}(E_\alpha)$$

$$(c) \quad g(\) \in \mathcal{D}(E_\alpha)_u^*$$

(a) If $g(\) \in E_\alpha$ consider the case for α a successor, then $g(\) \in \bigcup_{i=0} E_{\alpha-1,i}$ by induction hypothesis, $f_{\alpha-1}(\)$ is normal for $E_{\alpha-1}$ and by the Main Lemma $\exists M$

$$g(X) < F_{\alpha-1, M}(e, \max X) \quad X \in N^n.$$

Then by the arguments of the Hierarchy Theorem 4.7

$$g(X) < F_{\alpha-1, M}(e, X) < f_{\alpha}^{(M+R)}(\max X).$$

So $f_{\alpha}(\)$ majorizes E_{α} . If $g(\) \in E_{\alpha}$ and $\alpha_n \rightarrow \alpha$, then by induction hypothesis $f_{\alpha_n}(\)$ majorizes E_{α_n} . If $g(\) \in E_{\alpha} =$

$\bigcup E_{\alpha_n}$, then for some n $g(\) \in E_{\alpha_n}$. So $g(X) < f_{\alpha_n}^{(d)}(\max X) <$

$f_{\alpha_{n+1}}(\max X)$ if $\max X > d$. Thus if $\max X > \max\{n+1, d\}$, then

$$g(X) < f_{\alpha_{n+1}}(\max X) < f_{\alpha}(\max X).$$

So $\exists V$ such that $g(X) < f_{\alpha}^{(V)}(\max X)$.

(b) To show $g(\) \in \mathcal{D}(E_{\alpha})$ notice first $E_{\alpha} \subseteq \mathcal{D}(E_{\alpha})$. We have considered $g(\) \in E_{\alpha}$, thus we are only interested in those $g(\)$'s defined by programs using program modification over E_{α} . For this result we need only estimate the maximum value of functions computed by such programs with a constant bound on σ_{α} .

First determine a bound on $g(\)$. If $g(\) \in \mathcal{D}(E_{\alpha})$ then by 6.30 there is an $f_{\alpha}(\)$ such that

$$g(X) < f_{\alpha_{b(X)}}^{(t(X))}(\max X) \quad X \in N^n,$$

with $t(\), b(\) \in E_{\alpha}$. So since $E_{\alpha} = \bigcup E_{\alpha_n}$, for some n_2, n_1

$t(\) \in E_{\alpha_{n_2}}, b(\) \in E_{\alpha_{n_1}}$ and $t(X) < f_{\alpha_{n_1}}^{(e_1)}(\max X), b(X) <$

$f_{\alpha_{n_2}}^{(e_2)}(\max X)$. Now taking $e = \max\{e_1, e_2\}, m = \max\{n_1, n_2\}$ we

get

$$g(X) < f_{\alpha_m}^{(e)}(\max X) \cdot c \quad (\max X) \quad X \in N^n.$$

And since $f_{\alpha_m}^{(e)}(y) \cdot c < f_{\alpha_m}^{(\bar{e})}(y)$ $\bar{e} > e$,

we have

$$\begin{aligned}
g(X) &< f_{\alpha_m}^{(\bar{e})}(\max X) \\
&< f_{\alpha_m}^{(\bar{e})}(\max X)_{+1}(\max X) \\
&< f_{\alpha_m}^{(\bar{e})}(\max X)_{+1}(\max X) \\
&< f_{\alpha_m}^{(\bar{e})}(\max X) \\
&< f_{\alpha_m}^{(c)}(\max X) \quad X \in N^n.
\end{aligned}$$

We now want to show that for any $f'_\alpha(\cdot)$, not just $f_\alpha(\cdot)$, we can bound $g(\cdot)$. To do this we need only compare $f_\alpha(\cdot)$ and $f'_\alpha(\cdot)$. The following lemma handles this task.

6.32 Lemma. If $f_\alpha(\cdot)$ and $f'_\alpha(\cdot)$ are two s.s.n.e.'s for E_α , $\alpha < \epsilon_0$, then

$$\exists m \quad \forall x \quad \forall \alpha \quad 0 < \alpha < \varepsilon_0 \quad f'_\alpha(x) < f'_\alpha(x+m).$$

Proof. For $\alpha=0$, by definition of normality

$$f_0'(x) < f_0^{(m)}(x) \quad \forall x$$

so that

$$f'_1(x) = f_0^{(x)}(x) < f_0^{(x+m)}(x) \ll f_0^{(x+m)}(x+m).$$

So

$$f'_1(x) < f_1(x+m).$$

For transfinite induction assume

$$f'_\alpha(x) < f_\alpha(x+m)$$

Then simply as above

$$\begin{aligned} f'_{\alpha+1}(x) &= f'_\alpha(x)(x) < f_\alpha^{(x)}(x+m) \ll f_\alpha^{(x+m)}(x+m) \\ &= f_{\alpha+1}(x+m) \end{aligned}$$

For the limit case assume

$$f'_{\alpha_n}(x) < f_{\alpha_n}(x+m).$$

Then since

$$f_{\alpha_n}(x+m) < f_{\alpha_{n+p}}(x+m) \quad p > 0 \text{ (by 6.22),}$$

we have

$$f'_\alpha(x) = f'_{\alpha_x}(x) < f_{\alpha_x}(x+m) \ll f_{\alpha_{x+m}}(x+m).$$

Thus

$$f'_\alpha(x) < f_\alpha(x+m).$$

q.e.d. Lemma

Having established that $\mathcal{V}(E_\alpha)$ is normal with normal elements $f_\alpha(\)$, we can apply the Main Lemma 5.8 to conclude the following for all $\alpha < \epsilon_0$.

6.33

- (a) $E_\alpha \subseteq E_{\alpha,0}$
- (b) $E_{\alpha+1} \subseteq E_{\alpha+1,0}$
- (c) $E_{\alpha,i} \subseteq E_{\alpha,i+1}$

$$(d) \quad E_\alpha \subset E_{\alpha+1}$$

(e) E_α is closed under composition

(f) E_α is closed under limited recursion

Notice, we can also establish that $E_\alpha \subset \mathcal{D}(E_\alpha) \subseteq E_{\alpha,0}$ since $f_\alpha(\)$ majorizes E_α .

Using the above we then easily conclude

$$6.34 \quad \alpha < \beta < \epsilon_0 \implies E_\alpha \subset E_\beta$$

Proof. For transfinite induction on β , suppose $\alpha < \beta \implies E_\alpha \subset E_\beta$ then since $E_\beta \subset E_{\beta+1}$ by (d) above the result holds for $\beta+1$. If $\beta_n \rightarrow \beta$ and the result holds for all $\beta' < \beta$, then $\alpha < \beta \implies \exists n \alpha < \beta_n$ and so $E_\alpha \subset E_{\beta_n}$ and clearly $E_{\beta_n} \subset E_\beta$ since $E_\beta \subset \bigcup E_{\beta_n}$, so $E_\alpha \subset E_\beta$.

q.e.d.

To finish this section we want to prove the actual time theorem for $\alpha < \epsilon_0$, thus complete the parallel treatment of RCH and ERCH.

6.35 Actual Time Theorem

If $g(\) \in E_{\alpha,i}$ then $\sigma g(\) < E_{\alpha,i}$.

Proof.

(1) Case $\alpha=0$ for all i . Here $g(\) \in E_{\alpha,i}$ implies $g(\) < E_{\alpha,i-1} \forall i$ by definition.

(2) Case for successor ordinals. For induction assume the result for $E_{\alpha,i} \forall i$. Let $g(\) \in E_{\alpha+1,0}$, then by definition $\pi_{\alpha+1} g(\) < c$ which means $\exists \pi_g$ such that $\sigma_{\alpha+1} \pi_g(\) < c$. Let $h_1(\), \dots, h_p(\)$ be the fixed instructions in π_g .

Since $h_i() \in E_{\alpha+1} = \bigcup_{i=0} E_{\alpha,i}$ we have by induction hypothesis

$$h_i(Y) < E_{\alpha,j}.$$

Thus by definition of normality

$$h_i(Y) < f_{\alpha+1}^{(s_i)}(\max Y) \quad Y \in N^{m_i} \quad i = 1, \dots, p,$$

and

$$h_i(Y) < f_{\alpha+1}^{(v_i)}(\max Y) \quad Y \in N^{m_i} \quad i = 1, \dots, p.$$

If $h()$ is generated as primitive in π_g at x then by definition $\exists t()$ such that

$$\sigma h(Y) < f_{\alpha+1}^{(t_1(X))}(\max Y) \quad X \in N^n \quad Y \in N^{m_1} \quad t_1() \in E_{\alpha+1}$$

and by lemma 6.30 $\exists t_2()$ such that

$$h(X) < f_{\alpha+1}^{(t_2(X))}(\max Y).$$

Since $t_1() \in E_{\alpha+1}$, it follows by Main Lemma, that $\exists e_1, e_2$

$$t_1(X) < f_{\alpha+1}^{(e_1)}(\max X)$$

$$t_2(X) < f_{\alpha+1}^{(e_2)}(\max X).$$

So after S steps in the execution of $\pi_g()$, putting $m = \max\{e_1, e_2, v_i, s_i\}$ the maximum number of actual steps is given by

$$\sigma \pi_g(X) < f_{\alpha+1}^{(f_{\alpha+1}^{(m)}(\max X))}(\max X) +$$

$$\begin{aligned}
& + f_{\alpha+1}^{(f_{\alpha+1}^{(m)}(\max X))} (f_{\alpha+1}^{(m)}(\max X)) \\
& + \vdots \\
& + f_{\alpha+1}^{(f_{\alpha+1}^{(m)}(\max X) \cdot S)} (\max X)
\end{aligned}$$

and as before

$$* \quad \sigma \pi_g(X) < S \cdot f_{\alpha+1}^{(f_{\alpha+1}^{(m)}(\max X) \cdot S)} (\max X).$$

For $S = c$ a constant the right hand side is in $E_{\alpha,0}$. Now assume $g(\) \in E_{\alpha,i}$, then by definition

$$\sigma_{\alpha} \pi_g(\) < E_{\alpha,i-1}.$$

So by the Main Lemma $\exists e$ such that

$$\sigma_{\alpha} \pi_g(X) < F_i(e, \max X) \quad X \in N^n.$$

Applying the same analysis as above up to line * and putting $S = F_i(e, \max X)$ yields

$$\sigma \pi_g(X) < F_i(e, \max X) \cdot f_{\alpha+1}^{(f_{\alpha+1}^{(m)}(\max X) \cdot F_i(e, \max X))} (\max X)$$

which is

$$< F_{i+1}(p, \max X) \in E_{\alpha,i}.$$

(3) Case for limit ordinals. Assume for induction that the result holds for all $\beta < \alpha$. Let $\alpha_n \rightarrow \alpha$ with $f_{\alpha_n}(\)$ s.s.n.e. for E_{α} . Let $g(\) \in E_{\alpha,0}$ with π_g computing $g(\)$ such that $\sigma_{\alpha} \pi_g(\) < \text{constant}, c$. Suppose as above that $h_i(\)$ $i = 1, \dots, p$ are fixed instructions. Then by definition $h_i(\) \in \mathcal{D}(E_{\alpha})$ so that by definition of $\mathcal{D}(E_{\alpha})$ and by lemma 6.30 respectively $\exists s_i, v_i$

$$\sigma h_1(Y) < f_{\alpha}^{(s_1)}(\max Y) \quad Y \in N^{n_1}$$

$$h_1(Y) < f_{\alpha}^{(v_1)}(\max Y) \quad Y \in N^{n_1}.$$

Furthermore, if $h_j(\)$ is generated as primitive over $\mathcal{D}(E_{\alpha})$, then by definition

$$\sigma h_j(Y) < f_{\alpha}^{(t_1(X))}(\max Y) \quad t_1(\) \in E_{\alpha}$$

and by lemma 6.30

$$h_j(Y) < f_{\alpha}^{(t_2(X))}(\max Y) \quad X \in N^n \quad Y \in N^{n_1} \quad t_2(\) \in E_{\alpha}.$$

We are now in the situation for case (2) and can proceed exactly as before.

Chapter 7 Comparison of Hierarchies

In the first part of this chapter we consider the relationship between the ERCH and the other subrecursive hierarchies. In the second part we show how the choice of the s.s.n.e.'s can be made to depend rather naturally on a notion of relative computability with respect to the hierarchy classes.

The main theorem for the first part is:

7.1 Theorem.

$$(a) \quad \mathcal{E}^{\alpha+1} = E_{(1+\alpha)+1} \quad -1 \leq \alpha < \epsilon_0$$

$$(b) \quad \bigcup_{\alpha < \omega^{n-1}} C_\alpha = O(<_{\omega^n}) = \bigcup_{\alpha < \omega^n} \mathcal{E}^\alpha = \mathcal{R}^n = E_{\omega^n}.$$

Here the C_α , $C_0 = \mathcal{R}^1$, are the classes of the Kleene subrecursive hierarchy, $O(<_\alpha)$ are the unnested α -recursive functions, \mathcal{E}^α are the classes of Robbin's Extended Grzegorzczk Hierarchy, \mathcal{R}^n are Péter's n -fold recursive functions and E_α are the classes of the ERCH.

For $\alpha < \omega$ in part (a), the result is essentially Cleave's theorem 3, the theorem we promised in Chapter 4. We prove 7.1 by establishing (a) and then citing Robbin[44] for (b).

Most of the real work in proving this theorem is done in the Actual Time Theorem 6.35 and in 7.2.

7.2 Normal Form Theorem.

There exist predicates $T_m^S()$ elementary in $S = \{h_1(), \dots, h_n()\}$ and there exists an elementary function $U()$ such

that if $g() : N^n \rightarrow N$ and $g()$ is $M_1(\Sigma_S)$ computable, then $\exists e$ such that

$$(a) \quad g(X) = U(\mu y T_{n+2}^S(e, X, y)) \quad \forall X \in N^n,$$

(b) if $g()$ is $M_1(\Sigma_0)$ computable by program $\pi_g()$ and $h()$ is such that $\sigma\pi_g(X) < h(X) \quad \forall X \in N^n$, then $\exists b() \in \mathcal{E}(h())$ such that $g(X) = U(\mu y \leq b(X) T(e, X, y))$ so that $g() \in \mathcal{E}(h())$.

(c) if $g() \in E_\alpha$ then $g() \in \mathcal{E}(f_\alpha())$ for any normal element $f_\alpha()$ for $\mathcal{D}(E_\alpha)$.

This theorem is proved in Appendix C. What we must do here is show 7.3 and 7.4.

7.3 Lemma. If $W_0(x) = 2^x$, $W_{\alpha+1}(x) = W_\alpha^{(x)}(1)$, and $W_\alpha(x) = W_{\alpha_x}(x)$ for α a limit ordinal with $\alpha_x \rightarrow \alpha$, then $W_{\alpha+1}() \in E_{(1+\alpha)+1}$ and $W_\alpha() \in \mathcal{D}(E_\alpha) \quad \alpha < \epsilon_0$.

7.4 Lemma. For all α , $-1 \leq \alpha < \omega^\omega$, there is a $b() \in \mathcal{E}$ and $\exists m$ such that $f_{1+\alpha+1}(x) < W_{\alpha+1}^{(m)}(b(x)) \quad \forall x$.

With these two lemmata we can easily prove 7.1 (a) as follows. First to show $\mathcal{E}^{\alpha+1} \stackrel{\text{def}}{=} \mathcal{E}(W_{\alpha+1}()) \subseteq E_{(1+\alpha)+1}$ we observe that by 6.33 E_β is closed under composition, explicit transformations and limited recursion for all β . Thus since $W_{\alpha+1}() \in E_{(1+\alpha)+1}$ by Lemma 7.3, it follows from the definition of $\mathcal{E}(W_\alpha())$, 3. that $\mathcal{E}^{\alpha+1} \subseteq E_{(1+\alpha)+1}$.

Conversely, $E_{(1+\alpha)+1} \subseteq \mathcal{D}(W_{\alpha+1}())$ for $-1 \leq \alpha < \epsilon_0$ because by the Normal Form Theorem, $g() \in E_{(1+\alpha)+1}$ implies $g() \in \mathcal{E}(h())$ where $h()$ is any function such that $\exists \pi_g \quad \sigma\pi_g(X) <$

$h(X) \forall X$ and by the Actual Time Theorem 6.35 and Lemma 7.4 we have $\sigma\pi_g(X) < f_{1+\alpha+1}^{(d)}(\max X) = h(X) \in \mathcal{E}(W_{\alpha+1}(\))$. Thus the burden is to prove the lemma.

Proof of 7.3. First define $W_{-1}(x) = 2 \cdot x$, then $W_{-1}^{(x)}(1) = 2^x = W_0(x)$. We notice that $W_{-1}(\) \in E_0$ since multiplication is a basic operation. Now proceeding as in Appendix A we can compute $W_\alpha(\)$ according to a slight modification of the procedure outlined there. We need only show that for this procedure, say $\pi_{W_\alpha}(\)$, we have

$$\sigma\pi_{W_{\alpha+1}}(x) < f_{(1+\alpha)+1}^{(t(x))}(x) \quad t(\) \in E_\alpha$$

and

$$W_\beta(x) < f_{\beta}^{(t(x))}(x) \quad t(\), b(\) \in E_\beta.$$

But since the procedure for the $W_\alpha(\)$ takes less steps than the procedure for $f_{(1+\alpha)+1}(\)$, the result follows by the methods of Appendix B.

Proof of 7.4. First we notice that every $f_1(\)$ is elementary. We can see this quickly by noticing $\sigma\pi_{f_1}(\) < E_0 \subset \mathcal{E}$, so by 7.2 part (c) $f_1(\) \in \mathcal{E}$.

Next we observe that if $\lambda xg(\) \in \mathcal{E}$, then $\exists m$ such that $g(x) < W_0^{(m)}(x)$. This is another tedious result, but fortunately it has essentially been demonstrated by Ritchie[42] where he shows

$g(\) \in \mathcal{E}$ implies $\exists n \ g(x) < W_0^{(n)}(k \cdot x + k) \ \forall x$
and since $k \cdot x + k < W_0^{(k)}(x)$, we have $g(x) < W_0^{(n+k)}(x)$.
So $m = n+k$. Thus $\exists m \ f_1(x) < W_0^{(m)}(x) \ \forall x$, and since

$W_0^{(n-1)}(\cdot) \in \mathcal{E}$ we have the result for $\alpha = -1$. Now since $W^{(m)}(x) < W^{(m+x)}(1) \forall x$ we can continue by induction to show

$$f_2(x) = f_1^{(x)}(x) < W_0^{(m+x)}(x) < W_0^{(m+2x)}(1) = W_1(2x+m).$$

Further it follows that

$$W_1^{(x)}(2x+m) < W_1^{(4x+m)}(1)$$

since $W_1(x) > 2x+m \forall x > m$. So that

$$\begin{aligned} W_1^{(x)}(2x+m) &= W_1(2W(2W(\dots)+m)+m) \\ &< W_1(W_1(W_1(W_1(\dots)))) \\ &< W_1^{(4x+m)}(1) \end{aligned}$$

since $W_1(2x+m) < W_1^{(2x+m)}(1)$.

Thus

$$f_3(x) = f_2^{(x)}(x) < W_1^{(x)}(2x+m) < W_1^{(4x+m)}(1) = W_2(4x+m).$$

We can continue in this manner knowing that $W_n(x) > \omega n \cdot x+m$ $\forall x > \max\{n, m\}$ to conclude

$$f_{n+1}(x) < W_n(2n \cdot x+m).$$

Thus

$$f_\omega(x) = f_x(x) < f_{x+1}(x) < W_x(2x^2+m).$$

Since

$$W_\omega(x) > 2x^2+m \forall x > m-1$$

we even have

$$f_\omega(x) < W_\omega^{(m)}(x).$$

Using induction and extending the above techniques it is thus clear that for each $\alpha < \omega^\omega$ we will have

$$f_{(1+\alpha)+1}(x) < W_{\alpha+1}(x)$$

as desired. For details see the method of Appendix B.

7.5 In this section we will examine a relativized version of the ERCH. The goal is to show the hierarchy in a more abstract light and to point out how the standard fundamental sequences for $\alpha < \epsilon_0$ can be explained in terms of the hierarchy. We do not eliminate reference to normal elements altogether, but we show that the standard sequences can be defined in terms of the hierarchy process.

The extended Ritchie-Cleave hierarchy process can be relativized to any set of basic instructions Σ . However, if Σ is not normal, then the methods of proof used in Chapters 4-6 are no longer applicable, and it is not known how such relativized hierarchies behave in general. Let $E_\alpha()$ denote the relativized hierarchy up to α and call it the class of functions E_α computable in Σ . We are interested in $E_\alpha(f_\beta())$ for $f_\beta()$ a normal element. In this special case, the relativized hierarchy can be treated like the ERCH itself.

We first notice that although $E_\omega n = \mathcal{R}^n$ $n = 0, 1, \dots$ ($\mathcal{R}^0 = \mathcal{E}$), the notion of " $E_\omega n$ computable in" is not the same as the notion " \mathcal{R}^n recursive in". For example, $\mathcal{E}(f_1()) = \mathcal{E}$ whereas $E_1(f_1()) = E_2$. Moreover, the notion of RASP relative computability allows Σ to be infinite, raising the possibility of using program modification over a highly non-constructive set Σ . No analogue to this con-

dition occurs in ordinary recursion theory. It is probably an interesting task to attempt defining " E_α recursive in" for $\alpha > \omega^\omega$.

Next we abstract from the ERCH process the following notion.

7.6 An E_δ -extending sequence from E_γ is a sequence of unary recursive functions, $g_1()$, such that

- (a) $g_1() \notin E_\gamma$
- (b) $g_{n+1}() \notin E_\delta^n = E_\delta(g_n())$
- (c) $E_\delta^1 \supset E_\gamma$
- (d) $E_\delta^{n+1} \supset E_\delta^n$

If the $g_i()$ are normal elements generated from the same $f_0()$, then the sequence is called a normal E_δ -extending sequence from E_γ .

We say that the E_δ -extending sequence from E_γ , $g_1()$, reaches the class E_β iff E_β is the least hierarchy class containing $\bigcup_{i=1} E_\delta^i$.

7.7 Given a pair of hierarchy classes $\langle A, B \rangle$ define the ordering $\langle A, B \rangle < \langle A', B' \rangle$ iff $B' \supset B$ or if $B' = B$ then $A' \supset A$. For E_α define the minimal pair $\langle E_\gamma, E_\delta \rangle$ for E_α to be the $<$ -least pair such that every normal E_δ -extending sequence from E_γ reaches E_α . Call E_δ the basic class for E_α .

7.8 If E_α has a minimal pair $\langle E_\gamma, E_\delta \rangle$, then define $E_{\alpha_m} =$ least hierarchy class containing $\bigcap E_\delta(g_n())$ for all normal E_δ extending sequences $g_n()$ from E_γ .

If E_{α_n} exists it is called the s.s.h.c for E_α ; clearly it is unique. We will prove that the α_n of E_{α_n} constitute the standard fundamental sequence for α . Thus the s.s.h.c. can be used to select the s.s.n.e. This method will allow generation of the hierarchy up to E_{ω^ω} . The class E_{ω^ω} does not possess a basic class, but there exists for it a sequence of basic classes $E_{\omega^n} = B_n$. This sequence can be used to define the s.s.n.e. for the extension of E_{ω^ω} , namely using the correspondence $\omega^\alpha \rightarrow \alpha$, the standard fundamental sequence for E_ω defined by s.s.h.c. determines a fundamental sequence for ω^ω .

We show that in general every E_α for $\alpha < \epsilon_0$ possesses either a basic class or a sequence of basic classes. In each case the basic classes determine a s.s.h.c. which can be used to define the s.s.n.e.

First we prove the following basic relationship between the ERCH and the notion of E_α computability.

7.9 Theorem. For all $\alpha > 1$ $E_\alpha(f_\beta()) = E_{\beta+\alpha}$.

Proof by induction on α . We claim $E_1(f_\beta()) = E_{\beta+1} = \mathcal{E}(f_{\beta+1}())$. Let $\Sigma_\beta = \Sigma_0 \cup \{f_\beta\}$, then Σ_β is normal with $f_\beta()$ as normal element. Taking $f_\beta()$ as a new instruction we can define $f_{\beta+1}()$ just as it is defined using $f_\beta()$ as a primitive subprogram. Since Σ_β is normal, the methods of Chapter 6 imply that $E_1(f_\beta())$ is closed under composition, explicit transformation and limited recursion. Thus we get

$\xi(f_{\beta+1}(\)) \subseteq E_1(f_{\beta}(\))$. On the other hand, whenever $f_{\beta}(\)$ is used in a program as a primitive instruction, it can be replaced by a subprogram for $f_{\beta}(\)$ which by the Hierarchy Theorem can be generated as primitive at level β of ERCH. So $E_1(f_{\beta}(\)) \subseteq E_{\beta+1}$.

By induction using the above and the equations

$$E_{\alpha+1}(f_{\beta}(\)) = \bigcup_{i=0}^{\infty} E_{\alpha, i}(f_{\beta}(\)) = \bigcup_{i=0}^{\infty} E_{\beta+\alpha, i} = E_{\beta+\alpha+1}$$

$$E_{\alpha}(f_{\beta}) = \bigcup_{\gamma < \alpha} E_{\gamma}(f_{\beta}(\)) = \bigcup_{\gamma < \alpha} E_{\beta+\gamma} = E_{\beta+\alpha}$$

the theorem follows.

q.e.d.

The main theorems are

7.10 Theorem. The basic classes are $E_{\omega^{\beta}}$ $\beta < \epsilon_0$.

7.11 Theorem. For every α a limit ordinal $< \epsilon_0$, E_{α} either

(a) has a basic class or

(b) is limit of a sequence of basic classes.

In each case the basic classes determine the s.s.h.c. E_{α_n} and the s.s.n.e. $f_{\alpha_n}(\)$ for E_{α} .

To state theorem 7.11 in more detail, let $a = \omega^{\beta_1}.a_1 + \dots + \omega^{\beta_s}.a_s$ be the base ω decomposition of α with $a_s \neq 0$. For brevity write $\alpha = \gamma + \omega^{\beta}.a$ (so $\beta = \beta_s$, $a = a_s$, $\gamma = \alpha - \omega^{\beta}.a$). The term $\omega^{\beta}.a$ determines the case (a) or (b). If $a > 1$ then (a) holds with $E_{\omega^{\beta}}$ as basic class and $\langle E_{\gamma}, E_{\omega^{\beta}.a} \rangle$ as minimal

pair. If $\alpha=1$ and β is a successor ordinal, then case (a) holds with basic class $E_{\omega^{\beta-1}}$, minimal pair $\langle E_{\gamma}, E_{\omega^{\beta-1}} \rangle$. If $\alpha=1$ and β is a limit ordinal, then case (b) holds with $E_{\omega^{\beta_n}}$ as the sequence of basic classes. These theorems are proved by a simple application of Theorem 7.9 and the definitions.

Chapter 8 Algorithmic Complexity

The class $\mathcal{L}(E_\alpha) = \{\pi_f \mid f(\) \in E_\alpha\}$ is quite badly behaved. It is not r.e. Programs may have arbitrarily long running time. They may display exceedingly complex and inefficient program structure and some programs may be for all practical purposes unrecognizable as elements of E_α . Thus $\mathcal{L}(E_\alpha)$ is a bad notation system for E_α .

L_α is a better system but it too has drawbacks. In particular the program structure for $\pi_f \in L_\alpha$ may involve essential use of subprograms π_g belonging to higher languages, L_β , $\beta > \alpha$. Such was the case with example 5.3 of page 64 (Chapter 5). Furthermore for certain programs of L_α it may be virtually impossible to prove that they halt for each input (i.e. are total).

For $L_{\omega n}$ we know there are subsystems, $L'_{\omega n}$ which are well behaved, namely those arising as direct translations of the Péter formalism. The L_ω has some particularly simple subsystems. It is interesting that among these systems are those possessing measures of algorithmic complexity which lead naturally to a new way of stating the condition on program modification over E_ω . Using such languages it becomes possible to state the extension condition for E_ω in terms of the time it takes to "process certain program descriptions".

What we can prove is that for certain languages for E_ω , L_{s_ω} , there are measures of algorithmic complexity, $d(\)$, $\ell(\)$, such that

$$\pi g() \in Ls_{\omega}$$

implies

$$\sigma \pi g(X) < f_d(\pi g)(\max X).$$

Moreover $\ell()$ and $d()$ can be used to define the concept of program modification over Ls_{ω} . The basic fact is

8.1 The programs $\pi_{g_{i_x}}$ $i_x = 1, 2, \dots, n_x$ for $x \in N^n$ can be generated as primitive in π over Ls_{ω} iff

$$\pi_{g_{i_x}} \in Ls_{\omega} \text{ and}$$

$$\exists t(), b() \in E_{\omega} \text{ such that } \forall X \in N^n$$

$$i) \ell(\pi_{g_{i_x}}) < t(X)$$

$$ii) d(\pi_{g_{i_x}}) < b(X)$$

and $\exists \pi' \in L_{\omega}$ which generates $\pi_{g_{i_x}}$ at X .

Thus by using the special language Ls_{ω} and the notion of algorithmic complexity we can avoid explicit reference to the $f_n()$ in defining conditions on program modification.

The special language Ls_{ω} we originally used was obtained from Kleene's calculating system for primitive recursion. This system was chosen because S. McCleary in a 1967 paper had already analysed the computation bounds in this system in terms of a measure of algorithmic complexity, (number of R's in a computation expression.)

In the Kleene system if

$$f(X, 0) = g(X)$$

$$\text{and } f(X, y+1) = h(X, y, f(X, y))$$

then a standard way to compute $f(X, y_0)$ is to set up a loop which computes $g(X)$, $h(X, 0, g(X))$, $h(X, 1, h(X, 0, g(X)))$, $h(X, 2, h(X, 1, (h(X, 0, g(X))))$, ..., until y reaches y_0 . In

general, we pick a uniform method of translating recursion schemata into programs and we use only programs in this manner. Thus by restricting ourselves to programs which can do only this type of looping along with compositions, we get a system adequate for \mathcal{R}^1 and easy to analyse with respect to computation bounds. This original procedure was somewhat ad hoc. Since then Meyer & Ritchie have published a paper which allows a much neater treatment of this idea. Using the new Meyer-Ritchie results not only affords a more elegant treatment, but it affords a simpler treatment because they use bounding functions, $b_n()$ very similar to our $f_n()$ whereas tedious effort is involved in showing that McCleary's bounds are compatible with our $f_n()$. Thus instead of basing our observations on McCleary's work with Kleene's system as originally planned, we base it on the system of loop programs of Meyer & Ritchie.

Loop programs are defined as certain finite sequences of the basic instructions (1) $X \leftarrow Y$ (2) $X \leftarrow X+1$ (3) $X \leftarrow 0$ (4) LOOP X and (5) END. The instructions (4) and (5) always come in pairs of the form LOOP X, \underline{P} , END where \underline{P} is some loop program. The meaning is that the program segment P will be repeated x times where x is the content of register X. Thus for example LOOP X, $X \leftarrow X+1$, END will double the contents of X. The execution of LOOP X places the contents of X into a special register which is used to control the loop. The end statement can be regarded

as a test and decrement of this special register (performed just prior to execution of \underline{P}). The set of loop programs is denoted Loop.

We may translate these loop programs into RASP programs in a uniform manner regarding them as higher level languages which are compiled as RASP programs before being run. The translation is clear. The instructions (1) - (3) can be directly translated (by fixing a 0 register for each program). Given the loop instructions LOOP X, \underline{P} , END, they are translated into the RASP statements

$$\begin{array}{l} \text{XL} \leftarrow \text{X} \\ n \quad \text{C}(\text{XL}, 0,) \\ \text{XL} \leftarrow \text{XL} - 1 \\ \quad \underline{\text{P}} \\ \quad \text{C}(0, 0, n) \\ m \end{array}$$

where SL, n and m are chosen not to conflict with other registers.

Loop programs will compute functions in the same manner as general RASP programs, i.e. input locations and value locations are specified in core. We need only discuss loop programs which compute functions. Following Meyer & Ritchie we define Loop_n = the set of all loop programs in which LOOP instructions are nested to depth of at most n. Recalling $\text{Loop}_n^{*a} = \{f() \mid \pi_f \in \text{Loop}_n\}$ it can be proved

8.2 Theorem $\text{Loop}_n^{*a} = \mathcal{E}^{n+1} = E_{n-1} \quad n \geq 2$

(see Meyer & Ritchie). This allows us to assert that if $f() \in E_n$, then $\exists \pi_f \in \text{Loop}_{n+1} \cap L_n$.

Now define these measures of algorithmic complexity on loop programs.

8.3 $d(\pi)$ = maximum depth of nesting of loops, $\pi \in \text{Loop}$

$\ell(\pi)$ = length of the program, $\pi \in \text{Loop}$.

The running time, $\sigma\pi$, of a loop program π will simply be the running time of the corresponding RASP program. For $W_n(\)$ as defined in Chapter 7, Ritchie & Meyer prove

8.4 Theorem. If $\pi_f \in \text{Loop}$ and $d(\pi)=n$, then $\exists p \leq 6 \cdot \ell(\pi)$
 $\sigma\pi_f(X) < W_n^{(p)}(\max X)$.

We have shown in 7.4 that $\exists q \forall n \forall x W_n(x) < f_{n+1}^{(q)}(x)$. Thus we can say $\pi_f \in \text{Loop}$ implies

$$\sigma\pi_f(X) < f_{d(\pi)+1}^{(6 \cdot \ell(\pi)+q)}(\max X) \quad \forall X.$$

In terms of algorithmic complexity the condition on program modification over LS_ω becomes

8.5 The programs $\pi_{g_{i_X}} \in LS_\omega$ $i_X = 1, 2, \dots, n_X$ $x \in N^n$

can be generated as primitive in π over LS_ω iff $\exists t(\)$,

$b(\) \in E_\omega$ such that $\forall X \in N^n$

$$(i) \quad \ell(\pi_{g_{i_X}}) < t(X)$$

$$(ii) \quad d(\pi_{g_{i_X}}) < b(X)$$

and $\exists \pi \in LS_\omega$ which generates $\pi_{g_{i_X}}$ at input X .

The if part follows from the preceding remarks. For the only if part, suppose $\sigma\pi_{f_{i_X}}(\)$ are generated as primitive; then

$$\sigma \pi_{f_{1_X}}(X) < f_{b(X)}^{(t(X))}(\max Y)$$

so $f(\) \in \mathcal{C}(f_{b(X)}^{(t(X))}(\)) = E_{b(X)}$ so by theorem 8.2 above,

$\pi_{f_{1_X}} \in \text{Loop}_{b(X)}$. Thus $d(\pi_{f_{1_X}}) \leq b(X)$. Also by taking $t'(X)$

$= \max\{\ell(\pi_{f_{1_X}}), i_X = 1, 2, \dots, n_X\}$ we have $\ell(\pi_{f_{1_X}}) \leq t'(X)$ and

$t'(\) \in E_\omega$ because a program $\pi \in \text{Ls}_\omega$ must generate the $\pi_{f_{1_X}}$.

Thus n_X can be computed in E_ω . The significance of this observation is simply that the RCH can be extended without reference to normal elements.

These observations can be pursued in two interesting directions. First they lead to consideration of hierarchies of languages starting with Loop which will allow the entire ERCH to be generated on the basis of algorithmic complexity conditions rather than normal elements. The resulting language is type theoretic, being based on nesting of diagonalization and looping.

The other main direction involves the attempt to find a general description of the class of languages or measures of algorithmic complexity which behave as do the loop programs or the direct translations of primitive recursion. We have seen some properties that are crucial for these "natural languages" in example 5.3 of Chapter 5. We would like an abstract characterization of such languages for the classes E_α .

We shall not pursue either of these paths into algorithmic complexity but shall follow a third path in this direction. We are interested in directing our results toward a level of investigation more relevant to a practical theory of complexity. This means we shall limit ourselves to functions within E_{ω} .

PART II

Chapter 9 Finite Automata and RASP Machines

9.1 The ERCH emphasizes only one dimension of function complexity. We picture the ordinals as measuring the height of a function along this dimension. Another dimension, breadth, is measured by the complexity of the elementary operations which define $E_{\alpha+1}$ from $f_{\alpha+1}(\)$ or equivalently is measured by the structural (algorithmic) complexity of the programs used at each level.

The breadth of the ERCH has been fixed at \mathcal{E} (in the sense of theorem 7.1), so in order to study this dimension we shall look more closely at \mathcal{E} 's role in ERCH. One of the first questions is whether a simpler class \mathcal{B} of operations would generate a hierarchy B_α compatible with E_α , i.e., for each $E_\alpha \exists \beta \geq \alpha$ such that $B_\beta = E_\alpha$. The goal would be to find an extremely simple \mathcal{B} which sufficed. The simplest basis would probably be the finite automaton operations. But it is easy to see that they are not adequate, being in a sense much too narrow.

To pursue this problem we have decided to examine computations within \mathcal{E} with respect to the simplest operations, finite automaton operations. We shall attempt to build up to an adequate class \mathcal{B} starting with this simple basis.

From a practical point of view we recognize that most functions of interest in normal computing probably occur within the class $E_1(\mathcal{E})$, certainly within $E_\omega(\mathcal{R}^1)$. Thus

our subsequent investigations can be construed as an effort to better understand the "practically computable functions."

9.2 The first step in this new direction is to define the class of finite automaton computable functions in terms of the RASP computing system. At this level of complexity research it is usually necessary to specify an encoding of N in terms of a finite alphabet, $A = \{a_0, \dots, a_p\}$. Given alphabet A , let A^* denote the set of all finite sequences of elements of A . We use a binary encoding of N based on the alphabet $\{0, 1, B\}$ where B is a special symbol read as "blank."¹

9.3 We distinguish between a binary encoding (bc) and a unique binary encoding (ubc). By a ubc of $n \in N$ we mean the binary numeral for n preceded by a finite sequence of blanks, e.g. a ubc of 4 is BB100. By a bc of n we mean a member of $\{0, 1, B\}^*$ such that the removal of all B 's and those 0's which are not preceded (left to right) by a 1 results in the binary numeral for n . Thus BB1B0BB0B is a bc of 4 and so is 0B00BB100. If $X \in \{0, 1, B\}^*$, let $|X|$ denote the length of X , e.g. $|10B1| = 4$.

9.4 We next define a special type of RASP. Let $A = \{a_1, \dots, a_p\}$ and

$$F = \langle N, N \cup A, B, K, F_1(), F_2() \rangle$$

where $K = (N \cup A)^N$ and $F_1(), F_2()$ are generated by

This alphabet is standard in Turing machine Theory, see Myhill[38], Shepherdson & Sturgis[50], Ritchie[42] and Hermes[22], but not in automata theory where $\{0, 1\}$ is used, see Moore[35], Rabin & Scott[35].

specific instructions to be given below. The registers, addressed by N , hold either integers or the symbols of A . When the registers hold only elements of A they are regarded as cells. The cells will hold the data while the other registers will hold instructions or will serve special purposes. The following instructions are used to define the particular RASPs of interest here. Let $d() : N \xrightarrow{1-1} N$, $\bar{d}() : N \rightarrow N$ such that $\bar{d}(d(x)) = x$ and $d(x) > x \ \forall x$ and let $\alpha \in A$. Also let $ctl() : N \rightarrow N$ such that $ctl(x) > x$ and if $ctl(x) \neq d(x)$, then for all n $ctl^{(n)}(x) \neq d^{(n)}(x)$.

(1) $W(s, \alpha)$

Write the symbol α in the register whose address is stored in $s \in N$, then put $d(k(s))$ in s and if the current control address is e , go to $ctl(e)$ for the next instruction.

(2) $\bar{W}(s, \alpha)$

Same as above except $\bar{d}(k(s))$ is stored in s .

(3) $C(s; a_0, b_0; a_1, b_1; \dots; a_p, b_p)$

If the content of the cell $k(s)$ is a_i then go to location b_i for the next instruction.

(4) $s \leftarrow "b"$

Store the integer b in register s .

The address s is called the Storage Address Register (STAR). It holds the address of the cell to be operated on by the instructions. The instructions are said to use STAR, s . The functions $d()$ and $\bar{d}()$ control changes in STAR and

are called locator functions. These functions must be chosen so that there is room in the RASP memory for the program and the STAR. For example,

$$d(x) = x + 2$$

$$\bar{d}(x) = x \div 2$$

or $d(x) = x \cdot p$ p prime

$$d(x) = [x/p]$$

are possible. The control function $ctl()$ must be made compatible with the locators, e.g. take $ctl(x) = d(x)$ in the above.

9.5 A program π on a RASP F with locator functions $d()$ and $\bar{d}()$ is said to process $X \in A^*$ at s iff

(a) $s, d(s), d(d(s)), \dots, d^{(m)}(s)$ hold X when π begins, where $m = |X|$ and

(b) $d^{(n)}(s)$ is disjoint from the domain of π for all $n \in N$.

9.6 A Turing program on F is a finite sequence I_0, I_1, \dots, I_n of instructions such that

(a) I_0 is $s \leftarrow "b"$

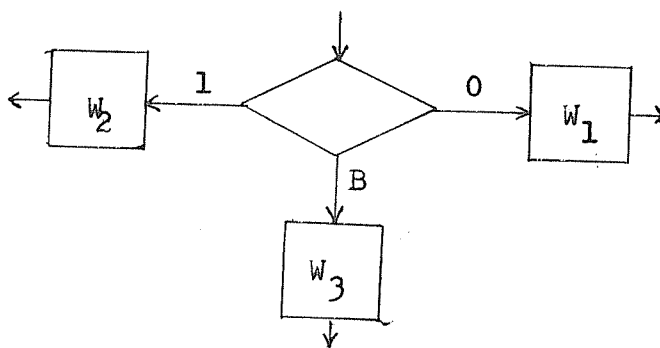
(b) I_i $i > 0$ are either conditional or write instructions each of which uses the same STAR, s .

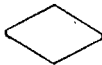

9.7 A finite automaton (f.a.) program on F is a Turing program such that

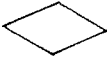
(b') I_i $i > 0$ are either conditional or write instructions each of which uses the same STAR, s , and the same locator function $d()$, i.e. no f.a. program

contains both W and \bar{W} instructions.


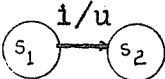
9.8 A RASP F with an f.a. program we shall call a finite automaton. A state diagram (graph) can be derived from a flow chart for an f.a. program. First observe that flow charts can be made uniform if successive conditionals are compressed (since they are redundant), and successive write operations are considered to have implied conditionals between them. In this form the flow chart for the case $A = \{0, 1, B\}$ can be represented with blocks of the type




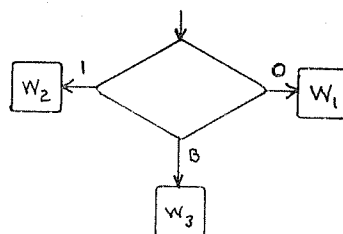
and an initial block of the form $a \leftarrow b$ where  represents a conditional and  a write statement. To

form a flow chart, the blocks are connected in obvious fashion, outward arrows go to the in terminal of some .

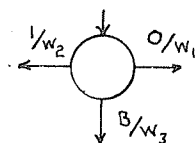
For terminating conditions, outward arrows are absent.

9.9 A state diagram can be derived from the flow chart by letting conditionals, represented now by 's, correspond to states and writing  to mean that

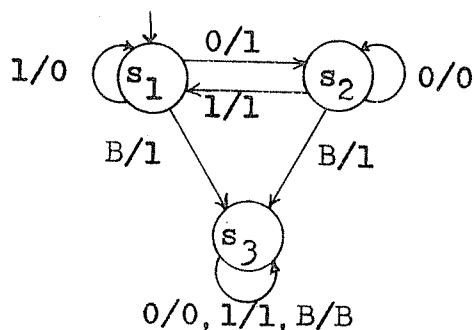
when the symbol being tested by conditional s_1 is 1, then u is written and control is transferred to conditional s_2 . The initial block  is simply represented by an arrow into a state symbol. Thus



becomes



The states are called the states of π and the set of them is written S_π . The following is a complete state diagram.



9.10 A transition function $\delta_\pi(\alpha, y)$ is defined for a program as follows. $\delta_\pi(\alpha, s_i) = s_j$ iff there is a connection \rightarrow between s_i and s_j in the state diagram ($\alpha \in A$). An output function $\lambda_\pi(, x)$ is defined so that $\lambda_\pi(\alpha, x) = \beta$ iff

there is a connection \longrightarrow from state x in the state diagram. A machine table of π is defined to be the set of quadruples $\langle \lambda_{\pi}(\alpha, s), \delta_{\pi}(\alpha, s), \alpha, s \rangle$ for all $\alpha \in A$ and all $s \in S_{\pi}$. We determine the initial state in the transition table by assigning it the least value among the addresses (N) assigned to conditionals.

9.11 The usual definition of a finite automaton over A with initial state, s_0 , is a quintuple

$$Q = \langle S, s_0, A, \delta, \lambda \rangle$$

where S is a set of states, $s_0 \in S$ is the initial state, A a finite alphabet, δ a map from $A \times S$ into S called the transition function and λ a map from $A \times S$ into A called the output function. The machine table for Q is defined as above taking δ for δ_{π} and λ for λ_{π} .

9.12 Given the finite automaton Q over A there is a RASP

$$QR = \langle N, N \cup A, a_0, K, F_1, F_2 \rangle$$

with an f.a. program π_Q such that the machine table of π_Q is isomorphic to that Q . Thus the study of RASPs of type F with f.a. programs includes the study of the usual finite automata. Among the concepts we need from automata theory is the notion of a state machine. This is a finite automaton with no output function $\lambda()$.

9.13 There are two quasi-physical models of finite automata commonly in use. One we call the tape model, the other the channel model. For the tape model imagine a tape infinite on the left end composed of squares over which a

control head moves from right to left one square on each step. The head starts at the right most square and is allowed to write a symbol of A in the square under scan on each step. The head stops moving upon completion of its work. This is the model Ritchie [42] employs.

For the channel model imagine a black box having input terminals x_1, \dots, x_n and output terminals y_1, \dots, y_n and having a single exterior on-off light. The box is fed symbols on its input terminals and may or may not produce output symbols on terminals y_i . For the case $A = \{0, 1, B\}$ we say as long as the box is receiving 0 or 1 symbols it remains on, but after receiving only B symbols it may go into an off state (light goes off).

The stop states in each model correspond to those states s for which $\lambda(\alpha, s) = \beta$ and $\delta(\alpha, s) = s$ $\alpha \in A$.

9.14 Considering automata over $\{0, 1, B\}$, the channel model provides the imagery for the definition of forced and autonomous response. When a finite automaton is responding to input pulses (0 and 1) its output is said to be its forced response. When the pulses cease (only B 's are input) any remaining machine response is called autonomous. In the tape model, if Q is processing a string $X \in A^*$ which is the ubc of $n \in \mathbb{N}$, then the autonomous response begins when the control head reaches the first B of the terminal string of blanks.

9.15 We wish to use our finite automata to compute

functions $f() : N^n \rightarrow N$. We need conventions for the input of multiple argument functions. Given an n -tuple $\langle x_1, \dots, x_n \rangle \in N^n$ and letting \bar{x}_1 denote the binary numeral of x_1 we consider the following input formats.

(a) series format: $\bar{x}_n B \bar{x}_{n-1} B \dots B \bar{x}_1$, for example $\langle 101, 100 \rangle$ is 101B100 in series format.

(b) mesh parallel format: $m_p m_{p-1} \dots m_1$ where the m_i are defined by the rule; the first n digits, m_n, \dots, m_1 are the rightmost digit of \bar{x}_1 in the order $\bar{x}_n, \dots, \bar{x}_1$, the next n digits are the second digit of \bar{x}_1 in order (if for some i \bar{x}_i is exhausted then 0 is taken as the second digit), etc. For example $\langle 101, 100 \rangle$ is 110010 in mesh parallel format or more generally $\langle d_3 d_2 d_1, e_4 e_3 e_2 e_1 \rangle$ is $e_4 d_3 e_3 d_2 e_2 d_1 e_1$ and $\langle e_4 e_3 e_2 e_1, d_3 d_2 d_1 \rangle$ is $e_4 0 e_3 d_3 e_2 d_2 e_1 d_1$.

(c) parallel format: for this format we need the alphabet $\{0, 1, B\}^n$ where the elements are written in column form, e.g. $\begin{pmatrix} 0 \\ 1 \\ \dots \\ B \end{pmatrix}$, then

$$\begin{pmatrix} d_{1,p} \\ d_{2,p} \\ \dots \\ d_{n,p} \end{pmatrix} \begin{pmatrix} d_{1,p-1} \\ d_{2,p-1} \\ \dots \\ d_{n,p-1} \end{pmatrix} \dots \begin{pmatrix} d_{1,1} \\ d_{2,1} \\ \dots \\ d_{n,1} \end{pmatrix}$$

is the parallel format where $p = \max |\bar{x}_1|$ and

$d_{1,p} d_{1,p-1} \dots d_{1,1}$ are the digits $0\ 0 \dots 0 \overline{x_1}$ where there are $p - |x_1|$ 0's. For example $\langle 101, 100 \rangle$ is

100
101

and $\langle d_3 d_2 d_1, e_4 e_3 e_2 e_1 \rangle$ is

$e_4 e_3 e_2 e_1$
$0 d_3 d_2 d_1$

We shall primarily use the mesh parallel format. This allows us to consider only finite automata over $\{0,1,B\}$ and gives canonical mappings $N^n \rightarrow N$ so that attention can be restricted to single argument functions.

9.15 We define the class FA, finite automaton computable functions over $\{0,1,B\}$ to be the class of all functions $f() : N^n \rightarrow N$ such that $\exists q$ and \exists an f.a. program π_f on F which started with $\langle x_1, \dots, x_n \rangle$ in mesh parallel format, beginning at a , will produce the bc of $f(x_1, \dots, x_n)$ in locations a through $d^{(m)}(a)$ for all $m > q$.

Other definitions of the f.a. computable functions over $\{0,1,B\}$ differ slightly from ours. For example, Ritchie uses parallel input format and requires the output in ubc. Most automata theorists use a channel model with parallel format over the alphabet $\{0,1\}$ and output is a member of $\{0,1\}^*$.

Our class FA is slightly broader than either of these usual classes. Nevertheless, the important basic theorems on finite automata continue to hold for FA. Among these results we shall need the following.

9.17 Theorem. If $f() \in \text{FA}$, then $\exists K \in \mathbb{N}$ such that $|f(x)| \leq |x| + K \quad \forall x \in \mathbb{N}$.

For a proof see Ritchie[42] p. 164.

9.18 Theorem. Let F be a finite automaton on $\{0,1,B\}$ with $|F| = n = \text{number of states of } F$, then if F has a 1 output, there is an input sequence x , $|x| \leq n$ which causes that output.

For a proof see Rabin & Scott[35] p. 75.

In closing this chapter we should point out that the RASP computing system is able to subsume the numerous machine models in current use such as multi-tape Turing machines, push down stack automata, Post machines, multi-tape finite automata and so on. The technique of defining these machines is similar to that used above in defining Turing programs and finite automata programs. The unifying treatment given the class of extant machines in Scott[49] could just as well be given using the RASP system.

Chapter 10 Rewind Automata

10.1 Finite automata are the stuff of genuine computers. For the computer designer they are good mathematical models. But for the computer scientist they are not adequate because they fail to provide a natural model of computing; for instance, even the function $x.y$ is not f.a. computable. What the computer scientist seeks is a generalization of finite automaton computability. This generalization should permit all recursive functions of practical importance to be computable.

Previously we approached the same problem from the viewpoint of Turing machines when we asked for a subclass of the Turing computable functions which is more realistic. That was an approach "from the outside". Now we are looking for a similar kind of class, i.e. the practically computable functions, but we approach the problem "from the inside" by seeking a generalization of finite automaton computability. From this perspective we stress the relationship to finite automata whereas from the previous perspective we demanded only a vague type of constructivity.

There already exist generalizations of finite automata, Turing machines, push down stack machines, and others. But there is little transfer of concepts from these machines to finite automata and conversely. The notion of an iterative array is another generalization of finite automata which is

more related to automata theory in technique and concepts, but these devices do not lend themselves to a theory of practical computability.

In looking for a way to extend finite automata to more general computing devices we are led to consider various compositions of automata. Hartmanis & Sterns[19] define two types of composition, series and parallel. But these concepts do not lead outside the class FA. We introduce below a type of composition which does go beyond FA, viz. input controlled composition.

10.2 Let $C = \langle S, s_0, A_m, \delta_0(\) \rangle$ be a state machine over $A_m = \{a_0, \dots, a_m\}$ with $S_0 = \{s_{0\ 1}, s_{0\ 2}, \dots, s_{0\ n_0}\}$. Let $F_i = \langle S_i, s_{i\ 1}, \{0, 1, B\}, \delta_i(\), \lambda_i(\) \rangle$ $i = 1, \dots, q \leq n_0$ be a sequence of finite automata over $\{0, 1, B\}$ having states $S_i = \{s_{i\ 1}, s_{i\ 2}, \dots, s_{i\ n_i}\}$.

Consider an assignment θ which is a pair of maps, a total map $f_1(\) : S_0 \xrightarrow{\text{onto}} \bigcup_{i=1}^q \{s_{i\ 1}\}$ and a partial map $f_2(\) : (\bigcup S_i - \bigcup_{i=1}^q \{s_{i\ 1}\}) \rightarrow A_m$. That is, $f_1(\)$ assigns to each state of C the initial state of some F_i and $f_2(\)$ assigns non-initial states of the F_i to the alphabet of C . For certain states $f_2(\)$ may be undefined. The F_i , C and θ will be used to define a composite machine denoted $C_\theta(F_1, \dots, F_q)$ and called an automatic composite of C, F_1 .

10.3 To precisely define the $f(\)$ - C composite machine of F_i , let $P = \langle S_0, A_m, \bigcup S_i, f_1(s_{0\ 1}), \{0, 1, B\}, \delta_p(\),$

$\lambda_p(\cdot), \theta >$ where S_0 is the set of control states, A_m is the control alphabet, $\cup S_i$ is the set of component states, $f_1(s_{01})$ is the initial state, $\{0,1,B\}$ is the working alphabet, $\delta_p(\cdot)$ is the transition function, $\lambda_p(\cdot)$ is the output function and θ is the assignment. P is called the C automatic composite of F_i . The F_i are called the component machines and C is called the control machine. The transition and output functions of the composite machine P satisfy the following where $\delta_i(\cdot), \lambda_i(\cdot)$ are the corresponding functions of the component machines and state machine, $i = 0,1,\dots,q$: If $f_2(s_{ij})$ is undefined then

$$\delta_p(\alpha, s_{ij}) = \delta_i(\alpha, s_{ij}) \text{ and}$$

$$\lambda_p(\alpha, s_{ij}) = \lambda_i(\alpha, s_{ij})$$

and if $f_2(s_{ij})$ is defined

$$\delta_p(\alpha, s_{ij}) = f_1(\delta_0(f_2(s_{ij}), f_1^{-1}(s_{i1}))) \text{ and}$$

$$\lambda_p(\alpha, s_{ij}) = \lambda_i(\alpha, s_{ij}).$$

10.4 The product machine operates as follows; if F_{i_1} is the machine assigned to the start state, s_{01} , of C then F_{i_1} begins processing the input x , if during processing F_{i_1} goes into a state s_{i_1t} which is assigned to a letter a_j of A_m , then processing by F_{i_1} terminates and the tape is sent to the machine assigned to $\delta_0(a_j, s_{01})$ (say $\delta_0(a_j, s_{01}) = s_{02}$ and $f_1(s_{02}) = s_{i_21}$) then F_{i_2} starts processing the tape (starting at the left edge) until it goes into a state

s_{i_2} which is assigned to a letter a_k of A_m , then processing by F_{i_2} terminates and $\delta_0(a_k, s_{i_2})$ determines the next component which will process the tape. This process continues until the control machine goes into its stop state and the machine assigned to this stop state halts. The process is undefined at x if C or the component machines used never goes into a stop state when given x or the machine F_1 , assigned to some control state reached given x , never goes into a state for which $f_2(\)$ is defined.

10.5 The notion of a computation of P can be made precise as follows. Define a configuration to be a triple $\langle t, s, q \rangle$ where t is an element of $\{0, 1, B\}^*$, $s \in \cup_{i=1}^n S_i$, $q \in N$, $0 \leq q \leq |t|$. Call t the tape in P , s the current state of P and q the number of the scanned square of t . Next define a yield relation, \rightarrow , between configurations.

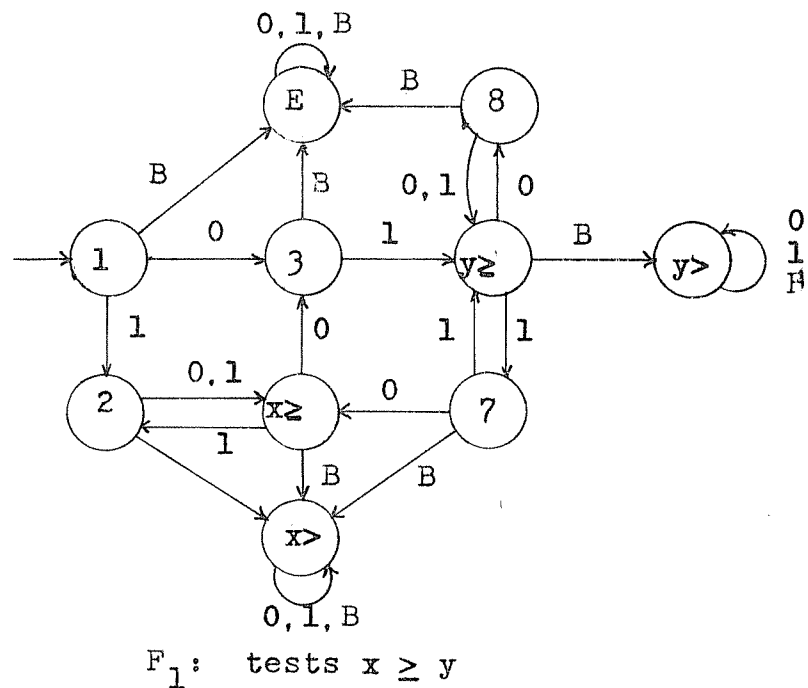
$\langle a_m a_{m-1} \dots a_1, s, q \rangle \rightarrow \langle b_m b_{m'-1} \dots b_1, s', q' \rangle$
if and only if

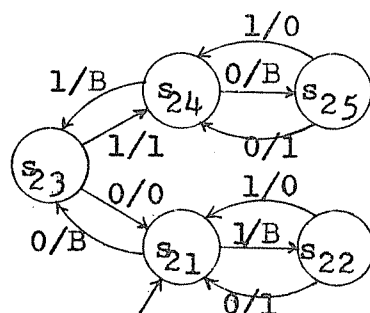
- (a) $b_i = a_i \quad \forall i \leq m \text{ and } i \neq q \text{ and } \delta_p(a_q, s) = s',$
 $\lambda_p(a_q, s) = b_q$ and
- (b) if $f_2(s)$ is defined, then $q' = 1, m' = m$ and
- (c) if $f_2(s)$ is undefined, then $q' = q+1$ and if $q = m$,
then $m' = m+1$ and $b_{m'} = B$, otherwise $m' = m$.

Using the above tape model the product automaton can be thought of as an automaton on a one way tape which not only moves along the tape square by square in one direction (as a finite automaton) but which can also move in the opposite

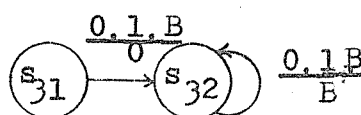
direction from any tape square to the end of the tape in a single step. A suggestive description is that during processing the tape moves along under the head and can be rewound at any moment. This leads to calling the machine a rewind automaton.

10.6 Below is an example of a rewind automaton which computes the difference $x - y$. This function is clearly not f.a. computable since there is no way to determine whether $x \geq y$ before the subtraction must be started. The machine below performs the subtraction in two passes; the first determines whether $x \geq y$ while the second either subtracts ($x \geq y$) or prints out 0 ($x < y$). The component machines are F_1 , f_2 , and F_3 , the control machine is C.

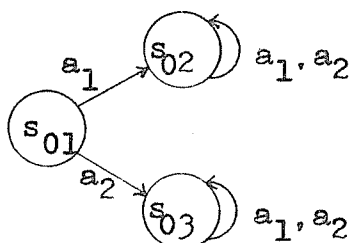




F_2 : subtraction



F_3 : 0 output



C: control

The assignment θ is

$f_1()$	$f_2()$
$s_0 1 \rightarrow s_1 1$	$x \rightarrow a_1$
$s_0 2 \rightarrow s_2 1$	$y \rightarrow a_2$
$s_0 3 \rightarrow s_3 1$	

10.7 We can make the definition of a rewind automata more general by defining a machine $\langle S, R, A, s_0, \delta(), \lambda() \rangle$ where S and R are sets of states, $S \cap R = \emptyset$, A an alphabet, s_0 an initial state, $\delta()$ a transition function and $\lambda()$ an

output function such that $\delta() : A \times S \cup R \rightarrow S$ and $\lambda() : A \times S \cup R \rightarrow A$, and $\delta(\alpha, r) = \delta(\beta, r) \quad \forall \alpha, \beta$ for $r \in R$.

A configuration is defined as before and the yield relation is defined as follows.

$$\langle a_m a_{m-1} \dots a_1, s, q \rangle \rightarrow \langle b_{m'}, b_{m'-1} \dots b_1, s', q' \rangle$$

if and only if

$$(i) \quad b_i = a_i \quad \forall i \leq m \text{ and } i \neq q \text{ and } \delta(a_q, s) = s' \\ \text{and } \lambda(a_q, s) = b_q$$

and

$$(ii) \quad \text{if } s \in S, \text{ then } q' = q + 1 \text{ and if } q = m \text{ then } m' = m + 1 \text{ and } b_{m'} = B \text{ otherwise } m' = m$$

and

$$(iii) \quad \text{if } s \in R, \text{ then } q' = 1, m' = m.$$

10.8 Using this more general definition of a rewind automaton the C_θ composite machine can be interpreted as a canonical form for rewind automata. Given a C_θ composite machine of F_1 it is clear how to define a rewind automaton, namely let $R = \{s_{1j} \mid j \neq 1 \text{ and } f_2(s_{1j}) \text{ is defined}\}$ and take $\delta_p(), \lambda_p()$ as $\delta(), \lambda()$.

10.9 Conversely, given a rewind automaton, F , there is a unique control automaton C and canonical form for F . Namely, let the states following rewind states be denoted $s_{21}, s_{31}, \dots, s_{q1}$ and let s_{01} be the initial state of F . Consider the s_{i1} as initial states and define a finite automaton F_1 for each as follows.

- (i) $s_{i-1} \in S_i$ and if for $s \in S_i$ and $s \notin R$ $\delta(\alpha, s) = s'$, then $s' \in S_i$.
- (ii) if $s \in S_i$ and $s \notin R$, then $\delta_i(\alpha, s) = \delta(\alpha, s)$, but if $s \in R$, then $\delta_i(\alpha, s) = s$.
- (iii) if $s \in S_i$ and $s \notin R$, then $\lambda_i(\alpha, s) = \lambda(\alpha, s)$, but if $s \in R$, then $\lambda_i(\alpha, s) = \alpha$.

To define the control machine pick an alphabet A such that $|A| = |R|$ and for each $r \in R$ assign $a_r \in A$. Then $C = \langle \{s_{i-1}\}, A, \delta_0(\) \rangle$ where $\delta_0(a_r, s_{i-1}) = s_{j-1}$ iff $(\alpha, r) = s_{j-1}$.

10.10 We can require further uniformity in the canonical form by stipulating that if $f_2(s_{i-j})$ is defined, then

$$\delta_i(\alpha, s_{i-j}) = s_{i-j}$$

and by stipulating that for stop states $\lambda_i(\alpha, s_{i-j}) = \alpha$. We can also require that no component automaton go into a rewind state until after its forced response. Without loss of generality we shall consider only C_0 composite machines of the restricted type.

10.11 In terms of the RASP computing system a rewind automaton can be presented by defining a rewind program π to be a finite sequence of instructions $\leftarrow "b", C(a;), W(a,)$ beginning with $a \leftarrow "b"$. That is π is like an f.a. program except unrestricted use of the $a \leftarrow "b"$ instruction, which initializes STAR, is allowed.

10.12 Another interpretation of rewind automata as a RASP computing system results from taking FA along with the

class FA_n of finite automaton n -valued relations and the conditionals on these relations as basic instructions on a RASP which uses one register, a , for computing and the others to hold the program. (If $R()$ is an n -valued f.a. relation with values $0, 1, \dots, n-1$, then $C(a; 0, b_0; 1, b_1; \dots; n-1, b_{n-1})$ means "if $R(k(a))$ has value m ($0 \leq m \leq n-1$) then go to statement b_m " and is the conditional on $R()$.)

10.13 The rewind automaton is a very natural result of an attempt to pursue the question we raised in the last chapter, how to refine the Ritchie hierarchy of elementary functions. Ritchie's original definition of his hierarchy contains a "discontinuity". He begins with the class F_0 of f.a. functions and then jumps to the class F_1 by allowing his machine to operate as a full Turing machine, i.e. move in both directions. It seems "natural" from the viewpoint of finite automata to define F_{n+1} = set of all rewind automaton computable functions whose computations require an amount of tape bounded by a function in F_n .

It is easy to show that the resulting rewind automaton based hierarchy still reaches precisely \mathcal{E} at the limit ω . Thus the work which follows can be construed as an investigation of refinements of a version of the Ritchie hierarchy.

10.14 To begin these investigations we define the following parameters on computation

- (1) $\tau\pi(x)$ = maximum number of tape squares used in the computation of π at input x .
- (2) $\sigma\pi(x)$ = number of steps used in the computation of π at x .
- (3) $\rho\pi(x)$ = number of rewinds in the computation of π at x .

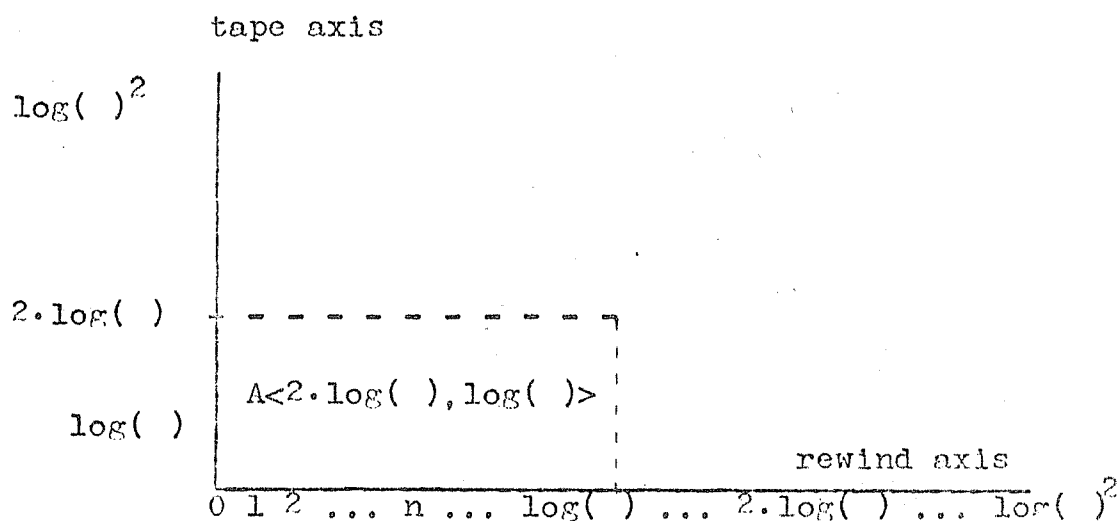
10.15 We are primarily concerned with the parameters τ and ρ and we define the classes

$$L\langle t(\), r(\) \rangle = \{ \pi_f \mid \tau\pi_f(x) < t(x) \ \& \ \rho\pi_f(x) < r(x) \}$$

$$A\langle t(\), r(\) \rangle = \{ f(\) \mid \exists \pi_f \in L\langle t(\), r(\) \rangle \}.$$

Notice that because of our input format these classes include multiple argument functions when $t(\)$ and $r(\)$ are only single argument functions, i.e. the same string of symbols can be treated as either a single numeral or as a uniquely decodable merger of n separate numerals.

10.16 Employing a rough analogy with geometry we can speak of co-ordinatizing a subset S of \mathcal{R} using tape and rewind parameters. By selecting a principal axis in each of the partially ordered sets of tape and rewind bounds we have something analogous to co-ordinate axes for which we can then seek some appropriate scale. It turns out that a scale based on $\log_2(\)$ is very natural and convenient (as we shall see later). The diagram below illustrates these ideas.



A natural question suggested by the diagram is

Q1: How are the functions distributed in this plane, e.g. is there a function which is in $A<\log(x), \log(x)^2>$ but not in $A<\log(x), \log(x)>$ or more generally in $A<t(x), r(x)>$ but not in $A<t'(x), r'(x)>$?

Questions of the type Q1 are in general difficult to answer for close values of $t(x)$, $t'(x)$ or $r(x)$, $r'(x)$. Another difficult type of question is

Q2: Given a function $f(x)$ what are bounds on $t(x)$ and $r(x)$ such that $f(x) \notin A<t(x), r(x)>$, e.g. is multiplication in $A<\log(xy) + 1, \log(x)^2>$?

Another question suggested by this context is

Q3: Is there a trade-off relationship between tape and rewind parameters, i.e. if $f(x) \in A<t(x), r(x)>$ do there exist $t'(x) < t(x)$ and $r'(x) > r(x)$ such that $f(x) \in A<t'(x), r'(x)>$?

Another basic question is

Q4: How do various known classes such as \mathcal{E}^1 , E_n relate to $A \langle t(), b() \rangle$? An especially interesting case is the location of a basis for the recursively enumerable sets. (A set of recursive functions, B , is called a basis for the r.e. sets iff every r.e. set is enumerated by some $f() \in B$.)

Questions Q1 - Q3 are all closely related. They are similar to questions considered in the papers of Hartmanis & Stearns[17], Rabin[41], and Cook[8]. Such questions are crucial to the problem of deciding when a given function can be computed under specified conditions. Such problems must be solved in order to yield the most basic evidence for a theory of computational complexity. For example, knowing under what conditions multiplication is harder to perform than addition is central to any complexity theory. Unfortunately such results are hard to come by. At this stage of our knowledge we are apparently forced to examine specific computing systems and in particular the "low level" computations of these systems in great detail in order to produce the required evidence. Such work requires a kind of combinatorial analysis which is tedious and difficult.

10.17 In attempting to develop new techniques for such problems, one is led in many directions. We are especially interested in those directions which lead toward automata theory. Since the problem of relating intrinsic function

properties, such as rate of growth or speed of oscillation, to properties of computations, such as number of steps or volume of memory, is apparently central to answering the questions raised above, the following technique recommends itself.

Technique 1. Given a property $P()$ of f.a. functions, discover how $P()$ behaves under increasing rewinds.

We will illustrate this technique below, but our main effort in the next chapter will be to give answers to special cases of Q1 and give an answer to Q4 as it pertains to a basis for r.e. sets. We use the later result to shed light on the main question of the previous chapter, is there a smaller basis for the ERCH than \mathcal{E} ?

Chapter 11 Low Level Complexity Classes

Rewind automata are well suited for the investigation of complexity at low levels of the hierarchy. In this chapter we prove some fundamental facts about $A\langle t(\), r(\) \rangle$. When we are only concerned about rewind behavior, we let $A(r(\)) = \bigcup_{t(\) \in \mathcal{R}} A\langle t(\), r(\) \rangle$. The basic theorems are given below.

11.1 Theorem. $A(n) = A(1) \quad n > 0$.

11.2 Theorem. If $\forall c \exists n$ such that $b(x) < \log \log(x)/c$ for all $x \geq n$, then $A(b(\)) = A(1)$.

11.3 $A\langle \log(\), \log(\) \rangle$ contains a basis for the r.e. sets.

The theorems are all proved by generalizing the techniques of finite automata theory. Theorems 1 and 2 offer a start on Q1 of Chapter 10 by giving a complete description of the rewind axis below $\log \log(\)$. Theorem 5 answers Q4. Numerous other results locating known classes $A\langle t(\), b(\) \rangle$ can be obtained in a routine manner, but they are not needed.

11.4 For $f(\) : \mathbb{N} \rightarrow \mathbb{N}$ a value $f(x)$ will be called singular iff there are infinitely many $n \in \mathbb{N}$ such that $f(n) = f(x)$. n is called a singular argument of $f(\)$.

11.5 Define the lower derivative, $f^0(\)$, of $f(\)$ as follows. Let y_1, y_2, \dots be the non-singular values of $f(\)$ in their natural ordering. Define x_1 as the largest x such that $f(x) = y_1$. Then put $f^0(x_1) = \min f(x_j) \mid x_j \geq x_1$. If

$x_i < x < x_{i+1}$, then $f^0(x) = f(x_{i+1})$. The lower derivative is well defined iff $f^0(\cdot)$ is total, i.e. infinitely many y_i 's exist.

An equivalent definition of $f^0(\cdot)$ is that it is formed by taking the slowest growing monotone subsequence of $f(\cdot)$'s non-singular values and filling it in with the largest step function. Notice, $f^0(x) \leq f(x) \forall x$ and $f^0(x)$ is monotone increasing. Also notice that $f^0(\cdot)$ is not recursive (uniformly) in $f(\cdot)$ nor necessarily recursive if $f(\cdot)$ is. We call $f^0(\cdot)$ the minimum rate of growth function for $f(\cdot)$.

Another way to see $f^0(\cdot)$ is to consider the usual partial ordering of functions, $f(\cdot) \leq_{a.e.} g(\cdot)$ iff $f(x) \leq g(x)$ for all but finitely many points. Put $f(\cdot) \leq_{n.s.} g(\cdot)$ iff $f(x) \leq g(x)$ a.e. on non-singular points.

We can then characterize $f^0(\cdot)$ by the following.

11.6 Proposition. If $m(\cdot)$ is monotone increasing and $m(\cdot) \leq_{n.s.} f(\cdot)$, then $m(\cdot) \leq f^0(\cdot) \leq_{n.s.} f(\cdot)$, i.e. $f^0(\cdot)$ is the greatest monotone function below $f(\cdot)$.

Let $[y]$ denote the greatest integer less than y . We now wish to prove

11.7 Theorem. Suppose $f(\cdot) \in FA$ and $f^0(\cdot)$ is defined, then $\exists L$ such that $f^0(x) > [x^{1/L}]/2 \forall x$.

The theorem follows from the lemma. Recall that for $x \in \{0,1,B\}^*$, $|x|$ denotes the length of x .

11.8 Lemma. Let $f(\cdot) \in FA$, then there is an L such that for all x if $f(x)$ is non-singular, then $|\overline{f(x)}| > |x|/L$

where $\overline{f(x)}$ is the binary numeral of $f(x)$.

Assuming for the moment that the lemma holds, let us derive the theorem. We ask for the minimum possible value that $f(\)$ could have while satisfying $|\overline{f(x)}| > |x|/L$. Such a value occurs when the digits of $\overline{f(x)}$ are 0's except for the last. So mapping each x to its minimum possible value means $x \rightarrow 2^{\lceil |x|/L \rceil}$. Put $L(x) = 2^{\lceil |x|/L \rceil}$.

Now compare $L(x)$ with $\lceil x^{1/L} \rceil$, notice $\lceil x^{1/L} \rceil$ is monotone increasing. For x 's of the form 2^k , we have $|\overline{2^k}| = k+1$, and $\lceil (2^k)^{1/L} \rceil = \lceil 2^{k/L} \rceil$. Compare $2^{\lceil k/L \rceil}$ with $\lceil 2^{k/L} \rceil$. Suppose $k/L = a + b/L$, so that $\lceil k/L \rceil = a$ and $2^{k/L} = 2^a \cdot 2^{b/L}$. Since $b/L < 1$ we get $2^{k/L} < 2^{a+1} = 2^{\lceil k/L \rceil + 1}$. Since $\lceil x \rceil \leq x$ we have $\lceil 2^{k/L} \rceil \leq 2^{\lceil k/L \rceil + 1}$. So $\lceil 2^{k/L} \rceil / 2 \leq 2^{\lceil k/L \rceil}$. Next,

$$* \quad L(2^k) = 2^{\lceil k+1/L \rceil} \geq \lceil 2^{k+1/L} \rceil / 2$$

by taking $k+1$ for k above.

We now claim $L(x) \geq \lceil x^{1/L} \rceil / 2 \quad \forall x$ because for $2^k < x < 2^{k+1}$ we know $L(2^k) \leq L(x) \leq L(2^{k+1})$ and $\lceil 2^{k/L} \rceil \leq \lceil x^{1/L} \rceil \leq \lceil 2^{(k+1)/L} \rceil$. So

$$\lceil x^{1/L} \rceil / 2 \leq \lceil 2^{(k+1)/L} \rceil / 2 \leq L(2^k) \leq L(x) \quad \forall x$$

by *.

Hence we now have $f(x) \geq_{n.s.} L(x) \geq \lceil x^{1/L} \rceil / 2 \quad \forall x$. But since $L(\)$ is a monotone function lying below $f(\)$, we know by Proposition 11.6 that $L(\) < f^0(\) \leq_{n.s.} f(\)$.

q.e.d.

11.9 We now prove 11.8, $|\overline{f(x)}| \geq |x|/L$ for $f(x)$ non-

singular, $f() \in \text{FA}$.

Suppose finite automaton F with N states computes $f()$. Given any input x_1 , partition it into disjoint sections of length $L > N$ (starting from the right). If on every segment there is a 1 output or if on every segment which is followed by a segment with a 1 output, there is a 0 or a 1 output, then the result clearly holds. We consider now the remaining case.

What we show is that if a segment not followed by a 1 produces all 0's or B's as output or if a segment which is followed by a 1 produces all B's as output, then $f(x_1)$ is singular. Consider the later case. Let the digits of the input segment which has produced all B's be denoted by $x_L x_{L-1} \dots x_2 x_1$. Record beneath each digit the state s_i of F when it receives the input digit x_i , e.g.,

$$\begin{array}{ccccccc} x_L & x_{L-1} & \dots & x_2 & x_1 \\ s_{i_L} & s_{i_{L-1}} & \dots & s_{i_2} & s_{i_1} \end{array}$$

Since $L > N$, two of these states must be identical, say

$s_{i_p} = s_{i_{p+q}}$. Let w_1, w_2, \dots, w_q be the digits between x_{i_p} and $x_{i_{p+q}}$. Let w denote the entire sequence $w_q w_{q-1} \dots w_1$.

Let $y = x_L \dots x_{i_{p+q}}$, $z = x_{i_p} \dots x_1$ and define

$$x_1 = y w z$$

$$x_2 = y w w z$$

$$x_3 = y w w w z$$

....

Observe that $f(x_1) = f(x_i)$ $i = 1, 2, 3, \dots$ since the output on w is all B's and since F is in state $s_{1_{p+q+1}}$ when it starts y for each x_1 . Thus $f(x_1)$ is singular which is a contradiction.

In the case when the segment is not followed by a 1, the 0's count as blanks in determining the value of $f(x_1)$ so the same argument applies.

q.e.d.

11.10 A class S of functions is said to have a minimum rate of growth, $m(\)$, iff $f(\) \in S$ implies $f^0(\) \geq_{a.e.} m(\)$ whenever $f^0(\)$ exists. What we have shown is that FA has a minimum rate of growth. That is, since $\log(x) \leq_{a.e.} x^{1/L}$, $\log(\)$ is a lower bound on the growth rate in FA.

11.11 We now ask how $f^0(\)$ behaves as the number of rewinds increases. We will prove that the only classes $A\langle t(\), b(\) \rangle$ which possess a minimum rate of growth are $A\langle t(\), 0 \rangle = \text{FA}$ and $A\langle t(\), 1 \rangle$ ("1 rewinds"). In each case $t(\) \leq \log(\) + c$. This result shows that minimum rate of growth unlike maximum rate of growth can not be used to characterize $A\langle t(\), b(\) \rangle$.

11.12 If we attempt to directly generalize the method of Theorem 11.7 to the case of multiple rewinds, we encounter the following difficulty. The input sequences $x_1, y w z, y w w z, \dots$ which were used to produce the singular value in the FA case may not produce a singular value in the rewind case. The response of the automaton can

depend on future digits as well as past digits since it may survey the input before processing it. There is however a way to reduce this problem to the case of finite automata. The method illustrates one of the nice features of rewind automata, their close connection with finite automata.

11.13 We first introduce the notion of a path automaton. Suppose that the rewind automaton F in canonical form is composed of the component automata F_1, \dots, F_p . If F is defined at the input x (computation terminates), then x is processed by a finite sequence of finite automata $F_{i_1}, F_{i_2}, \dots, F_{i_n}$ (n depending on x) where $F_{i_{j+1}}$ processes the output of F_{i_j} and F_{i_1} processes x itself. This sequence of automata can be put together to define a single finite automaton, P , called a path automaton which processes x so that the output of $P(x)$ is identical to the output $F(x)$. We define P precisely next.

$$P = \langle S_{i_1} x \dots x S_{i_n}, \langle s_{i_1}, \dots, s_{i_n} \rangle, \{0, 1, B\}, \delta_p(), \lambda_p() \rangle$$

Let R = rewind states of F , then

$$\lambda_p(\alpha, \langle s_1, \dots, s_n \rangle) = \gamma$$

and

$$\delta_p(\alpha, \langle s_1, \dots, s_n \rangle) = \langle s_1', \dots, s_n' \rangle$$

iff

$$(\delta_{i_1}(\alpha, s_1) = s_1' \ \& \ \lambda_{i_1}(\alpha, s_1) = \beta_1) \ \&$$

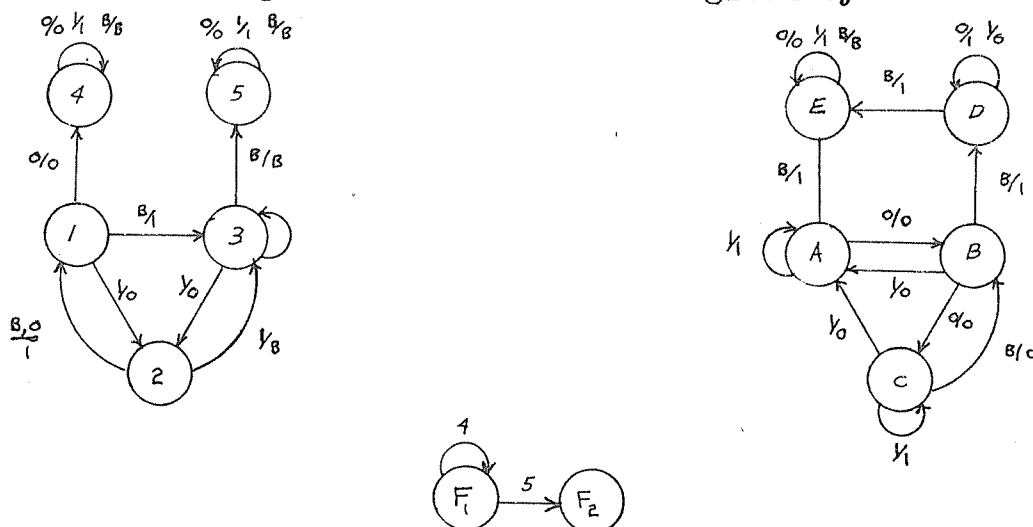
$$(\delta_{i_2}(\beta_1, s_2) = s_2' \ \& \ \lambda_{i_2}(\beta_1, s_2) = \beta_2) \ \&$$

...

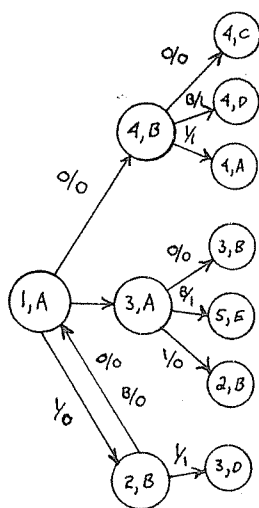
$$(\delta_{1_n}(\beta_{n-1}, s_n) = s_n' \text{ \& \; } \lambda_{1_n}(\beta_{n-1}, s_n) = \gamma)$$

Recall in the canonical form $\delta_1(\alpha, s) = s$ and $\lambda_1(\alpha, s) = \alpha$ if $s \in R$.

11.14 To illustrate the notion of a path automaton consider the simple rewind automaton F given by



Consider the path automaton $F_1 F_2$. It begins as follows.



11.15 We now consider a generalization of Theorem 11.7 to the case of multiple rewinds. Suppose $f() \in A^*(t(), 1)$ and is computed by F , say $|F| = N$. If P_1, P_2, \dots, P_q are the path automata of F , then $|P_i| \leq N^2$ $i = 1, \dots, q$. Arguing as in Theorem 11.7 we suppose that for some input x with $|x| > N^2$ the output sequence contains N^2 or more consecutive B 's. Then the technique of Theorem 11.7 can be applied to the path automaton P which processes x . We need only guarantee that the state cycle produced in P can actually be realized by F .

This will happen if the rewind behavior is not altered by repeating the tape segments, w , which produce the state cycle. But during the state cycle in P no component can go into a rewind state, r_{1j} . This is because once it goes into such a state it remains in that state until the pass is completed. In the canonical form the pass is not repeated until after the forced response. Thus the component state would remain r_{1j} , and it would thus be impossible for a state cycle in P to occur. Thus a rewind state r_{1j} can occur as a component state in a state cycle of P only if it occurs in the first state of the cycle which means that the rewind behavior has been determined before the state cycle.

It now follows that the segments w can be repeated without changing the rewind behavior. Thus the inputs $x_1, y w z$,

$y w w z, y w w w z, \dots$ are all processed by P since rewind state behavior controls selection of the component automata.

Hence we conclude

$$|\overline{f(x)}| \geq |x|/N^2$$

if x is non-singular. We state this result formally.

11.16 Theorem. If $f(\cdot) \in A\langle t(\cdot), 1 \rangle$, then $\exists L$ such that $f^0(\cdot) > [x^{1/L}]$.

The above argument clearly generalizes to $b(\cdot)$ rewinds so that we can conclude more.

11.17 Theorem. If $f(\cdot) \in A\langle t(\cdot), b(\cdot) \rangle$, then $\exists N$ such that $|\overline{f(x)}| \geq |x|/N^{b(x)}$.

But now unless $b(x) < \log\log(x)$ $x > n$ the statement is meaningless since $|x|/N^{\log\log(x)} < 1$ and clearly for all $g(\cdot)$ $|\overline{g(x)}| > 1 \forall x$. We can in fact show that the above result is applicable to only a narrow class of functions. Namely if $\forall c \exists m$ such that for $x > m$

$$b(x) < \log\log(x)/c,$$

then $A(b(x)) = A(n)$. Thus the range of application is somewhere between $\log\log(\cdot)$ and $\log\log(\cdot)/c$. We will prove this fact in 11.21. First we prove a result that allows a further reduction.

11.1 Theorem. $A(n) = A(1)$, $n > 0$.

11.18 The theorem asserts that a bounded number of rewinds provides no more power than a single rewind. Moreover we can arrange the computation so that the first pass is merely a recognition pass, i.e. no writing is done on

that pass. This result offers an interesting contrast with Hartmanis[20]. He shows that for Turing machine complexity classes based on tape reversals, $n+1$ tape reversals gives more functions than n reversals.

The idea of the proof is that if $f()$ is computed by F which requires at most n rewinds regardless of input, then it is possible to list all path automata of F and build a new machine F' which uses its first pass to decide which path automaton is applied and its second pass to actually apply that automaton. The only fact to be proved carefully is that F' can select the path automaton in one pass.

The intuitive argument is that given F it is known which component automaton is applied first, say F_{11} . As F' passes over the tape the first time it simultaneously computes with each path automaton and keeps track of the states of all path machines that it is running, say it keeps a table of current states in the path automata P_1, \dots, P_q . When the first pass component F_{11} goes into a rewind state, then F' knows what the second pass component is, say F_{12} and can then eliminate from consideration certain of the P_i , those not having F_{12} as their second component. It keeps track of this elimination by marking its table of states with 0's. F' continues along the tape until F_{12} rewinds unless it already has in which case this fact is recognizable from the state

of P_1 (the second component will be in a rewind state). In either case after a finite number of steps (possibly 0) F' will know the third pass component, F_{1_3} , and can eliminate more table entries. This process continues until one of the path automata remaining under consideration, say P_{i_0} , stops. Then F' rewinds and processes the tape using P_{i_0} . The fact that all this can be done by a finite automaton follows because the table for keeping track of the P_i is finite which in turn follows since the number of rewinds is bounded.

11.19 To be more precise about the above arguments let P_1, \dots, P_q be the path automata and F_1, \dots, F_p be the component automata of F . Construct a sequence, L , whose elements are states of the P_i , integers $k \leq n$ and integers $c \leq p$. The integer k keeps track of passes over the tape while c keeps track of which component of F is used on that pass. The states S_i $i=1, \dots, p$ are assumed to be disjoint. The states of P_i are denoted $\langle e_{i,1}, e_{i,2}, \dots, e_{i,t} \rangle$ for $t \leq n$ where each $e_{i,j} \in S_{k_j}$. Where k_j depends on the order of the components F_i in the path.

11.20 The sequence L is $\langle s_{1,j}, s_{2,j}, \dots, s_{q,j}, k, c \rangle$ where $s_{i,j}$ is a state of P_i . The initial state L_1 will be $\langle s_{1,1}, \dots, s_{q,1}, 1, i_1 \rangle$ where $s_{i,1}$ are the initial states of P_i and i_1 is the first pass component of F . We can define transition and output functions $\delta_F, (\)$, $\lambda_F, (\)$. The "usual" transition will be $\delta_F(\alpha, L) = L' = \langle s'_{1,j}, \dots, s'_{q,j}, k', c' \rangle$

where $s'_{i,j} = \delta_{P_1}(\alpha, s_{i,j})$, $k' = k$ and $c' = c$. And the output on the first pass of F' is $\lambda_{F'}(\alpha, L) = \alpha$. But, whenever F rewinds this usual transition is interrupted and certain states $s_{i,j}$ are set to be zero (and remain zero).

The first rewind is detected when for some i , $s_{i,j} = \langle e_{i,1}, e_{i,2}, \dots, e_{i,t_i} \rangle$ and $e_{i,1}$ goes into a rewind state, i.e. $e_{i,1} \in R$. When this happens the next component, F_{i_2} , can be found from the control component for F . Then $c' = i_2$ and $k' = k+1$. Also all path automata which do not have F_{i_2} as second component are eliminated, i.e. if $\delta_{P_1}(\alpha, s_{i,j}) = \bar{s}_{i,j} = \langle e_{i,1}, \dots, e_{i,t_i} \rangle$ and $e_{i_2} \notin F_{i_2}$, then $s'_{i,j} = 0$. Once the i -th component of L is set to 0, it remains 0, i.e. if $s_{i,j} = 0$ then $s'_{i,j} = 0$ for all L .

This transition for L to L' continues as described above until a stop state is reached in the remaining path automata, those not represented by 0 in L . Then F' rewinds and applies that automaton to the tape.

q.e.d.

Pursuing the line of questioning implicit in the last theorem we wonder for what value of $r(\)$ will new functions appear in $A(r(\))$. We can use the methods of studying minimum growth to obtain the following interesting result.

11.2 Theorem. If $\forall c \forall n \exists m$ such that $n < b(x) < \log \log(x)/c \forall x > m$, then $A(b(\)) = A(1)$.

11.21 Suppose $f(\) \in A(b(\))$ and is computed by F . Let $|F| = 2^b$ (b a real number). Pick $d > 2b$ and m such that $b(x) < \log\log(x)/d \ \forall x > m$. In order for the hypothesis to hold for d , there must exist an $x > m$ such that F rewinds less than $b(x)$ times for all $y < x$ and $b(x)$ times at x . Let x have digits $x_n x_{n-1} \dots x_1$. If F rewinds at most $b(x)$ times, then the largest number of states in the path automaton P followed by x is less than $|F|^{\log\log(x)/d} = |F|^{\log(n)/d} < 2^{b \cdot \log(n)/d} \leq n^{1/2}$. So $|P| < n^{1/2}$.

Thus while processing x , P must go into a state loop of length $L \leq |P| < n$ during the forced response. Since the loop occurs in P (rather than only in component automata) there is an input y with $|y| < |x|$ which is processed by P and which avoids at least one passage through this loop (using inverse of process in Theorem 11.7).

Moreover we know that removing the loop preserves the rewind behavior because no component machine can go into a rewind state during one of these loops. It must do so either before or after because a rewind is represented by the appearance of a rewind state as a component in a state of P . Such a component can not change until the first pass automaton rewinds which it does not do until after the forced response. Thus F exhibits the same rewind behavior on x and y . But this means that F must rewind $b(x)$ times at y although $|y| < |x|$. This contradiction means that F can not rewind $b(x)$ times for $b(x)$ satisfying the hypothesis.

But if $f() \in A(b())$ this means $f()$ must be computed within n rewinds. By Theorem 11.1 we have the result.

q.e.d.

11.22 We conclude with 11.3 showing that once $\log()$ rewinds are used there is enough computing power to produce a basis for the r.e. sets. We also point out that a certain especially interesting class of rewind automata can be used to produce a basis.

For this result we need an arithmetization of a classical theory of one-way tape Turing machines. Ritchie's [42] version is adequate and fortunately comes already arithmetized. We alter his presentation inessentially by using mesh parallel input rather than series input. It is a simple exercise to construct a Turing machine which will unscramble the mesh input to produce a series input. We also modify his arithmetization by using configurations $\langle t, s \rangle$ where t is a number representing the tape of the Turing machine along with the cell being scanned and where s is the current state of the Turing machine.

11.23 What we shall do is show that a rewind automaton can be built to recognize the "T-predicate",

$T(e, x, y)$ iff e is the number of a Turing machine with input x and y is the number of a computation of e at x .

The rewind automaton will recognize $T()$ in $A\langle \log(), \log() \rangle$. Then we show that the function $U()$ of the normal form theorem can be computed in $A\langle \log(), 0 \rangle$. These facts

will allow us to enumerate any r.e. set in $A\langle \log(), \log() \rangle$.

11.24 As a by-product of this effort we obtain the result that rewind automata compute all partial recursive functions (the converse is trivial since a Turing machine can directly simulate a rewind automaton). The result follows by simply building a rewind automaton which given input x successively produces $e x y$ on its tape for $y = 0, 1, 2, \dots$ and then checks $T(e, x, y)$. If the predicate holds, $U(e x y)$ is output, if not $e x (y+1)$ is produced, $T(e, x, y+1)$ checked and the process repeated.

11.25 Preliminary to our basis result are some coding conventions, similar to those of Ritchie[42]. A Turing machine over $\{0, 1, B\}$ is a set of quintuples

$\langle \text{output, next state, direction, input, present state} \rangle$, where the directions are L (left), R (right), S (no move). Inputs and outputs are members of $\{0, 1, B\}^*$.

The states s_1, s_2, \dots are represented by sequences of 0's assigned as follows.

$$\begin{array}{l} s_1 \text{ to } 0 \ 0 \ 0 \ 0 \ 0 \\ s_2 \text{ to } 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ s_3 \text{ to } 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \vdots \\ s_n \text{ to } \underbrace{0 \ 0 \ 0 \ 0}_4 \underbrace{0 \ \dots \ 0}_n \\ \vdots \end{array}$$

This leaves $0, \ 0 \ 0, \ 0 \ 0 \ 0, \ 0 \ 0 \ 0 \ 0$ available as special

markers in appropriate contexts. For the alphabet we make the assignment

1 to 1
0 to 1 1
B to 1 1 1.

Directions are

L to 1
R to 1 1
S to 1 1 1.

Quintuples are represented as strings

output-state direction 0 input-state.

For example, $\langle B, s_2, R, 1, s_1 \rangle$ is represented by 1 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0. The representation has a unique decomposition from right to left.

If q_1, q_2, \dots, q_n are quintuples, then the set of quintuples is represented by $\bar{q}_n 0 \bar{q}_{n-1} 0 \dots 0 \bar{q}_1$ where \bar{q}_i represent the quintuples. Again the sequence has a unique right to left decomposition as a sequence of quintuples. We also observe that the resulting representation is a binary numeral.

We next describe the binary representation of tapes as part of a representation of configurations (Ritchie calls them instantaneous descriptions). We use 0 as a tape spacer and also as a special marker to mark the square which is exactly one square to the right of the square being scanned (unless that square is the rightmost). The tape

segment

1	B	0	1	1	B	1	0
---	---	---	---	---	---	---	---

 is represented by

1 0 1 1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 1 0 1 1. A configuration consists of $\langle \text{tape}, \text{state} \rangle$ with 0 again being used as a separator. Thus if t is the above tape then $\langle t, s_2 \rangle$ is represented by $\bar{t} 0 \bar{s}_2$, i.e.

1 0 1 1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 1 0 1 1 0 0 0 0 0 0 0.

Since exactly three and exactly four consecutive 0's do not occur in any representation given above, we can use 0 0 0 and 0 0 0 0 as markers between configurations. Thus if K_1, K_2, \dots, K_n is a sequence of configurations, then $\bar{K}_n 0 0 0 0 \bar{K}_{n-1} 0 0 0 \dots 0 0 0 \bar{K}_1$ represents the sequence where \bar{K}_i represent the K_i . The sequence 0 0 0 0 is used to separate off the last item in the sequence of configurations.

It is clear what it means for a sequence of configurations to represent a computation, for details on this see Ritchie [42].

11.26 Computing $T(e, x, y)$ involves two basic procedures.

1. Checking state transitions: if $K_1 = \langle t_1, s_1 \rangle$, then given the computation K_n, K_{n-1}, \dots, K_1 it must be verified that $\delta_e(\alpha, s_1) = s_{1+1}$ for α the scanned value on t_1 .

2. Checking tape transitions: it must be verified that if $\lambda_e(\alpha, s_1) = \beta$ and $\mu(\alpha, s_1) = \gamma$ (where γ is R, L or S), then t_{1+1} 's head marker is in the right place (e.g. one left if $\gamma = L$) and $\bar{\gamma}$ immediately follows the marker.

The basic task for 1. is to locate the proper quadruple in the encoding of the sequence of quadruples represented by e . This is a linear "table look up" procedure. Once

the quadruple is located it is possible to check the next state against the next configuration state symbol by symbol.

The basic task for procedure 2, is simply to compare the tape representations symbol by symbol using the relative locations of the special markers to determine whether the head moved right, left, neither or invalidly (e.g., if the special marker appears first on the tape t_{i+1} and in the next square on t_i then the head moved one right). Also procedure 2, must locate the scanned symbol of t_i , say p_i , and communicate it to procedure 1. (a task performed on one pass over t_i).

The work is done by careful use of blanks as markers using internal memory to distinguish the various uses of the blanks. Recall that e , x , y are stored in mesh parallel format so that the tape appears as

			y_4	x_4	e_4	y_3	x_3	e_3	y_2	x_2	e_2	y_1	x_1	e_1
--	--	--	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

But it is best thought of as a parallel input in the form

e
 x
 y .

In more detail, if $y = \bar{K}_n \ 0 \ 0 \ 0 \ 0 \ \bar{K}_{n-1} \ 0 \ 0 \ 0 \ \dots \ 0 \ 0 \ 0 \ \bar{K}_1$,
 $e = \bar{q}_m \ 0 \ \bar{q}_{m-1} \ 0 \ \dots \ 0 \ \bar{q}_1$ and $x = x_r \ x_{r-1} \ \dots \ x_1$ where
 $q_i = \langle \beta_i, b_i, \gamma_i, \alpha_i, a_i \rangle$ and $\bar{K}_i = \langle t_i, s_i \rangle$, then the tape is
 best thought of as

$$\begin{array}{cccccccccccccccccccc}
 \dots & 0 & \beta_3 & b_3 & \gamma_3 & 0 & \alpha_3 & a_3 & 0 & \beta_2 & b_2 & \gamma_2 & 0 & \alpha_2 & a_2 & 0 & \beta_1 & b_1 & \gamma_1 & 0 & \alpha_1 & a_1 \\
 & \\
 & \dots & 0 & 0 & x_r & \dots & x_3 & x_2 & x_1 \\
 & \\
 & \dots & \underline{t_2} & 0 & s_1 & 0 & \underline{t_1} & 0 & s_1
 \end{array}$$

11.27 Procedure 1. in detail involves these steps.

(a) On the first pass put a B in the first digit of s_1 and in the first digit of each a_i . The a_i are determined by their appearance in the squares $3 \cdot n + 1$ in a specified order, i.e. that given by $\bar{q}_m 0 \dots 0 \bar{q}_1$. So it is possible to tell when the 1st digit of an a_i begins. Rewind after encountering a B following an appearance of 0 0 0 0 in the configuration, y , positions, e.g. $3 \cdot n$ for $n = 1, 2, \dots$.

(b) On subsequent passes replace the B of s_1 with a 0 and move the B one left if next symbol is 0. Also replace B of a_i with 0 and if next symbol is part of the state representation, then write B. If not, then no B appears which will indicate that $a_i \neq s_1$.

(c) On the pass when the end of s_1 is recognized (by spotting a 1 in next position), a check is made of all quintuples of the still candidates for the one beginning $\dots p_1 s_1$. Those beginning with s_1 (only three) can be spotted on this pass and marked with a B following the B placed by step (b). After placing this marker B the machine continues over the tape to see if $\alpha_1 = p_1$ (the scanned symbol of t_1). Another B marker is placed after α_1 to indicate the correct quintuple, $\langle \beta_1, b_1, \gamma, p_1, s_1 \rangle$. The direction symbol, γ , and the output, β_1 , are remembered internally to be

used in conjunction with process 2.

(d) All B's except that following the double B are replaced with their 0's and the first digit of the next configuration state, s_{i+1} , is made blank. The B in the correct quintuple is advanced one and a check is made for the end of the state. If the end is detected, then $s_{i+1} = b_1$ is checked. If it is false, then the computation terminates with a no result. If it is true, then step S is executed. If the end is not detected, then the leading digit of s_{i+1} remains B and a second B is used as in (a) for a comparison marker with b_1 and the digit by digit comparison continues.

(e) When $s_{i+1} = b_1$ is determined, the marks of e are all removed and the machine returns to s_{i+1} , whose location is determined by B, and starts process (a) with s_{i+1} for s_1 .

11.28 Simultaneous with the above state transition process a tape comparison process is run. For the initial configuration a check is made to see that x is recorded on the tape. On subsequent tapes a check is made that the right symbol is written and the right move made. The detailed steps are below.

(a) The initial configuration tape, t_1 , must be checked to see that it contains exactly x. This is done by making a symbol by symbol comparison ($\log(t_1)$ rewinds) until t_1 is exhausted. Blanks are used for markers as in process 1. The end of t_1 is indicated by 0 0 0. Then a check is made to see if remaining x digits are all 0 (requiring one pass).

where $c_i x$ is the i -th argument of a three mesh parallel decomposition of x .

The function $g()$ can be computed by first defining a machine which stores a and e internally and computes $T(c_3 x, c_2 x, c_1 x) \ \& \ c_3 x = e$. The computation of the predicate requires less than $\log(x)$ rewinds (in fact $\log(c_1 x) + 1$) according to the above procedure. Next the machine either computes $U(x)$ or a each of which require only one pass over the tape. Thus $g() \in A\langle \log(), \log() \rangle$. But $g()$ enumerates precisely S because if $z \in S$ then $\exists x_1$ such that $f(x_1) = z$ and $\exists e \exists x$ such that $c_3 x = e$, $c_2 x = x_1$ & $T(c_3 x, c_2 x, c_1 x)$. Thus $g(x) = U(x) = f(x_1) + z \in S$.

q.e.d.

Appendix A Computing Procedure for $f_\alpha(\)$ $\alpha < \epsilon_0$

The computing procedure given here is designed to be intelligible and general rather than efficient. It is designed to be applicable to s.s.n.e.'s defined for $\alpha < \beta$ where β is a constructive ordinal satisfying some additional conditions (not made precise here) which intuitively correspond to the existence of very effective notation systems for β . (The first strongly critical number, (Schütte[48]), is probably an example of such a β .)

The procedure can be seen clearly if it is first presented in terms of a RASP having ordinals as addresses. After presenting the procedure on such a machine, we will convert it to a procedure on our basic machine $M_1(\Sigma_0)$ by picking a notation system for a segment of ω_1 which includes ϵ_0 .

A.1 The special RASP, OM, has a memory which is made up of blocks. The blocks come in pairs and are addressed by an ordinal and an ordered pair of integers. Thus

$$A_{OM} = \langle \alpha, n, m \rangle \quad \alpha < \epsilon_0, \ n \in \mathbb{N}, \ m \in \mathbb{N}.$$

The contents of OM come from \mathbb{N} , i.e. $B_{OM} = \mathbb{N}$.

The idea behind this memory organization is that the registers $\langle \alpha, _ , _ \rangle$ will hold the program $\pi_{f_\alpha}(\)$, the first column for the instructions, the second for data and work space. A program $\pi_{f_\alpha}(\)$ operates on input x as follows.

A.2 (1) If α is a successor ordinal, say $\beta + n$, then π_{f_α}

successively

$$f_{\beta}^{(x)}(x), f_{\beta+1}^{(x)}(x), \dots, f_{\beta+n-1}^{(x)}(x).$$

(2) If α is a limit ordinal, then $\pi_{f_{\alpha}}$ uses α and the input x to compute α_x . If α_x is a successor ordinal, then $\pi_{f_{\alpha}}$ proceeds as in (b). Otherwise

(a) $\pi_{f_{\alpha}}$ loads a copy of $\pi_{f_{\alpha_x}}$ into $\langle \alpha_x, _ \rangle$ (the in-

struction in $\langle \alpha_x, _ \rangle$ the data in $\langle \alpha_x, 1, _ \rangle$) and gives

$\pi_{f_{\alpha_x}}$ instructions to return its value (and control)

back to $\pi_{f_{\alpha}}$. Thus $\pi_{f_{\alpha}}$ having assured itself it will

regain control, turns over to $\pi_{f_{\alpha_x}}$ the input x and control of the machine.

(b) For α_x a successor, $\pi_{f_{\alpha}}$ loads in locations

$\langle \alpha_x, _ \rangle, \dots, \langle \alpha_x - x, _ \rangle$ subprograms which perform

the iterations of step (1). Then $\pi_{f_{\alpha}}$ loads $\pi_{f_{\beta}}$ in β ,

where β is the limit ordinal $\alpha_x - x$. It also instructs

$\pi_{f_{\alpha_x}}$ to return its value (and control) to $\pi_{f_{\alpha}}$ and then

sends the input x and control to $\pi_{f_{\alpha_x}}$.

This method of computing $f_{\alpha}(\)$ follows the definition of $f_{\alpha}(\)$ quite closely as we will see below. The procedure is wasteful of space because it reproduces entire programs (in step (2)) which are nearly exact copies of itself. Thus during the course of the computation of $f_{\alpha}(\)$, the core is highly populated with subprograms most of which are almost

exactly the same. Such a situation indicates that a procedure must be available to compress these copies and save considerably on space. Indeed there is, but the cost of this space saving is that the computation is harder to visualize and harder to analyse. (Furthermore, since a space saving compression seems possible without drastically altering the time bounds, the program presented here can be thought of as an instructional version of a good one which also works.)

An example will help clarify the computing procedure. First recall the definition of the $f_\alpha(\)$.

A.3 Rule I (Iteration). If $\alpha = \beta + 1$, then $f_\alpha(x) = f_\beta^{(x)}(x)$.

Rule II (Diagonalization). If α is a limit ordinal and α_n is its standard fundamental sequence, then $f_\alpha(x) = f_{\alpha_x}(x)$.

Notice an appropriate choice for $f_0(\)$ is $\lambda x(x+1)^2$.

A.4 Now compare the computation of $f_{\omega^{2.2}}(4)$ by hand (using an informal equation calculus) and by the program $\pi_{f_{\omega^{2.2}}}$ on OM. Since $\omega^{2+\omega \cdot n} \rightarrow \omega^{2.2}$ as $n \rightarrow \omega$, Rule I yields the equation

$$f_{\omega^{2.2}}(4) = f_{\omega^{2+\omega \cdot 4}}(4).$$

The program $\pi_{f_{\omega^{2.2}}}$ with input 4 will apply step (2). Thus it will compute $\omega^{2+\omega \cdot 4}$, load the program $\pi_{f_{\omega^{2+\omega \cdot 4}}}$ into

$\omega^{2+\omega \cdot 4}$, give that program the value 4 and turn control over to it. Since $\omega^{2+\omega \cdot 3+n} \Rightarrow \omega^{2+\omega \cdot 4}$, Rule II yields the equation

$$f_{\omega^{2+\omega \cdot 4}}^{(4)} = f_{\omega^{2+\omega \cdot 3+4}}^{(4)}$$

and Rule II yields

$$f_{\omega^{2+\omega \cdot 3+4}}^{(4)} = f_{\omega^{2+\omega \cdot 3+3}}^{(4)} (4).$$

At this point in the computation by hand it is expedient to begin working on $f_{\omega^{2+\omega \cdot 3+3}}^{(4)} (4)$ without fooling with reductions of the type

$$f_{\omega^{2+\omega \cdot 3+3}}^{(4)} (4) = f_{\omega^{2+\omega \cdot 3+3}}^{(3)} (f_{\omega^{2+\omega \cdot 3+3}}^{(4)} (4)).$$

Thus the next equation, by Rule I, is

$$f_{\omega^{2+\omega \cdot 3+3}}^{(4)} (4) = f_{\omega^{2+\omega \cdot 3+2}}^{(4)} (4).$$

Then

$$f_{\omega^{2+\omega \cdot 3+2}}^{(4)} (4) = f_{\omega^{2+\omega \cdot 3+1}}^{(4)} (4)$$

and

$$f_{\omega^{2+\omega \cdot 3+2}}^{(4)} (4) = f_{\omega^{2+\omega \cdot 2}}^{(4)} (4).$$

Then application of Rule II yields

$$f_{\omega^{2+\omega \cdot 3}}^{(4)} (4) = f_{\omega^{2+\omega \cdot 2+4}}^{(4)} (4).$$

Corresponding to these steps, the program $\pi_{f_{\omega^{2+\omega \cdot 4}}}$ applies step (2b) and loads iteration subprograms at locations $\omega^{2+\omega \cdot 3+4}, \dots, \omega^{2+\omega \cdot 3+1}$. The iteration subprograms are linked

up so that they send data and control back and forth properly (see page 175). At location $\omega^{2+\omega.3}$ it loads $\pi_f^{2+\omega.3}$.

Then control is turned over to subprogram $\pi_f^{2+\omega.3+4}$ which

passes control through $\pi_f^{2+\omega.3+1}$ to $\pi_f^{2+\omega.3}$. These program steps are seen to be quite similar to the hand steps.

The reason the $\pi_f^{2+\omega.4}$ behaves differently than the other

limit program, $\pi_f^{2.2}$, is that loading all iteration sub-

programs at once will allow us to use simple techniques in estimating running time for the iteration subprograms.

The computation of $f_{\omega^{2.2}}^{(4)}$ now continues

$$f_{\omega^{2+\omega.2+4}}^{(4)} = f_{\omega^{2+\omega.2+3}}^{(4)}$$

$$\vdots$$

$$f_{\omega^{2+\omega.2}}^{(4)} = f_{\omega^{2+\omega.4}}^{(4)}$$

$$f_{\omega^{2+\omega+4}}^{(4)} = f_{\omega^{2+\omega+3}}^{(4)}$$

$$\vdots$$

$$f_{\omega^{2+\omega}}^{(4)} = f_{\omega^{2+\omega}}^{(4)}$$

$$f_{\omega^{2+\omega}}^{(4)} = f_{\omega^{2+\omega}}^{(4)}$$

$$\vdots$$

$$f_{\omega^2}^{(4)} = f_{\omega.4}^{(4)}$$

⋮

$$f_0(4) = \underline{\hspace{2cm}}$$

A value can be computed at $f_0(4)$ and the "long journey" back up through the equations begins. The journey is "long" because for every step up, we must repeat the entire procedure below again. In the machine computation the trip back up to $\pi_{f_0^{2.2}}$ is controlled by the programs which have been located in memory on the "downward trip". Thus, for example, π_{f_0} having computed $f_0(4) = y_4$, the value y_4 is taken by the iteration subprogram in location 1 and sent back to π_{f_0} . This is done four times producing $f_0^{(4)}(4)$ which is sent by the iteration subprogram in location 2 back to the iteration subprogram still in location 1.

A.5 Let us now describe the computing procedure on OM in more detail. First look at an iteration subprogram. This program will compute $g^{(x)}(x)$ if a program for $g(\)$ is linked to it properly.

< $\alpha, 1, 1$ >	1	HOLD \leftarrow HOLD + 1
< $\alpha, 2, 1$ >	2	IF INPUT = 0, GO TO 8
< $\alpha, 3, 1$ >	3	HOLD \leftarrow INPUT
< $\alpha, 4, 1$ >	4	LINKgIN \leftarrow HOLD
< $\alpha, 5, 1$ >	5	GO TO LINKgCTL
< $\alpha, 6, 1$ >	6	INPUT \leftarrow INPUT - 1
< $\alpha, 7, 1$ >	7	GO TO 2
< $\alpha, 8, 1$ >	8	OUTPUT \leftarrow HOLD
< $\alpha, 9, 1$ >	9	GO TO EXIT

This program will be located at some registers $\langle \alpha, n, 1 \rangle$ $n = 1, \dots, 9$. The word LINKgIN will refer to some register

$\langle \beta, 1, n \rangle$ while LINKgCTL will refer to some register $\langle \beta, m, 1 \rangle$. We also allow that EXIT may refer to some $\langle \gamma, n, 1 \rangle$ in which case OUTPUT will refer to $\langle \gamma, 1, m \rangle$.

With the above conventions we can use the format $B(\beta, \alpha, \gamma)$ to indicate the above program where α is the location of the program, β is the location of the program for $g(\)$, and γ is the location of the upward exit which may either be another program or a stop condition. We also say b is the location of the downward link and c is the location of the upward link. Using the notion of an iteration subprogram we can describe the computation of π_{f_α} on OM in detail. We consider the following.

A.6 (1) If $\alpha = \beta + n$, then π_{f_α} consists of a program π_{f_β} and iteration subprograms linked as follows;

$B(\text{STOP}, \alpha, \gamma_1) \overset{\curvearrowright}{B(\alpha, \gamma_1, \gamma_2)} \overset{\curvearrowright}{B(\gamma_1, \gamma_2, \gamma_3)} \dots \overset{\curvearrowright}{B(\gamma_{n-3}, \gamma_{n-2}, \beta)} \overset{\curvearrowright}{\pi_{f_\beta}}$

(2) If α is a limit ordinal, then among the program constants of π_{f_α} is an identification constant, referred to in the program by ID, which is α . If the input to π_{f_α} is $x \in N$, then π_{f_α} operates as follows;

- (a) compute the ordinal α_x (described as computing the downward link),
- (b) test whether α_x is a successor ordinal, if it is go to step f, otherwise,
- (c) load a copy of this program in α_x and store α_x

- as the identification constant of that program,
 store α as the upward exit,
- (d) transfer the input, x , to the input of $\pi_{f_{\alpha_x}}$.
 - (e) transfer control to $\pi_{f_{\alpha_x}}$,
 - (f) load the iteration subprograms $B(\alpha, \alpha_x, \alpha_x - 1), \dots, B(\beta + 2, \beta + 1, \beta)$ where β is the largest limit ordinal $< \alpha_x$ ($\beta = \alpha_x - x$),
 - (g) load a copy of this program in β and store β as the identification constant, store $\beta + 1$ as the upward exit,
 - (h) go to step (d).

We now take up the task of implementing this computing procedure on a mere RASP of the type M. To accomplish this we must represent the triples $\langle \alpha, n, m \rangle$ for $\alpha < \epsilon_0$ by integers. This entails representing the ordinals $\alpha < \epsilon_0$. Moreover we need an effective and manageable system of ordinal notation. To keep the system manageable we are led away from maximal systems such as Kleene's $\mathcal{O}(S_3)$ and led to systems such as Takeuti's [57] or Schütte's [48]. We choose the system given in Schütte since it meets our requirements. It is readily available in both English and German and is a frequently studied system. We denote this system by \mathcal{S} . To describe it we need the following notation.

A.7 Let p_0 denote the prime number 2 and p_n denote the n -th odd prime number. If $a \neq 0$, put $(a)_{p_1} =$ exponent of p_1

in the prime factorization of a . Now define the relation \leq_s ($a \not\leq_s b$ denotes the negation of $a \leq_s b$) inductively as follows; $a \leq_s b$ iff at least one of the conditions below is true.

- (1) $a=0$ and $b=0$
- (2) $b \neq 0$ and $a \leq_s (b)_{p_1}$ for at least one i
- (3) $a \neq 0$, $b \neq 0$ and $(a)_{p_1} \leq_s (b)_{p_1}$ for all i
- (4) $a \neq 0$, $b \neq 0$ and there are numbers $m \leq n$ such that
 - (a) $(a)_{p_i} = 0$ for all $i < m \neq 0$
 - (b) $(a)_{p_m} \leq_s b$
 - (c) $b \not\leq_s (a)_{p_j}$ for all j $m < j < n$
 - (d) $(b)_{p_n} \not\leq_s (a)_{p_n}$
 - (e) $(a)_{p_k} \leq_s (b)_{p_k}$ for all $k > n$.

It is easy to show that \geq_s is a reflexive total ordering and therefore that $a \equiv b$ iff $a \leq_s b$ and $b \leq_s a$ is an equivalence relation on N . An irreflexive ordering, $<_s$, is defined as; $a <_s b$ iff $b \not\leq_s a$.

In the familiar manner the integers can now be associated with ordinals, and we say that a represents a finite ordinal or a transfinite ordinal (with respect to $<_s$) according as $a <_s 3$ or $3 \leq_s a$.

A.8 By a path in \mathcal{J} we mean a set $P \subset N$ such that

- (a) P is well ordered by $<_s$.
- (b) If $a < P$ and $b <_s a$, then $\exists b' < P$ such that $b' \equiv b$.

Thus a path provides a unique set of notations for some initial segment of ordinals. We shall be interested in a path for ϵ_0 .

We can define addition on the ordinals using the following as auxiliary functions

$$\begin{aligned} \text{A.9} \quad \text{tw}(a) &= \begin{cases} a_0 & \text{if } 2^{a_0} \cdot 3^{a_1} \\ 0 & \text{otherwise} \end{cases} \\ \text{th}(a) &= \begin{cases} a_1 & \text{if } a = 2^{a_0} \cdot 3^{a_1} \\ a & \text{otherwise} \end{cases} \end{aligned}$$

A.10 Now define

$$a \oplus b = \begin{cases} b & \text{if } a = 0 \\ 2^{(\text{tw}(a) \oplus b)} \cdot 3^{(\text{th}(a))} & \text{otherwise.} \end{cases}$$

Since $\text{tw}(a) < a$ the definition is recursive and $\lambda a b (a \oplus b)$ is computable.

It is easy to show, see Schütte[48] p. 284, that if a represents α then 3^a represents ω^α . This allows us to use Cantor's normal form, i.e. for any $a \neq 0$ there exist c_1, \dots, c_m ($m \geq 1$) such that

$$\text{A.11} \quad a \equiv 3^{c_1} \oplus \dots \oplus 3^{c_m} \quad c_m \leq_s c_{m-1} \leq_s \dots \leq_s c_1$$

where the c_i are unique mod \equiv . The c_i are also computable for $a \neq 0$ so that we can give a computable definition of multiplication of ordinals as follows.

$$\text{A.12} \quad (1) \quad a \times b = 0 \quad \text{if } a=0 \text{ or } b=0.$$

$$(2) \quad a \times 1 = a \quad \text{if } a \neq 0.$$

$$(3) \quad \text{If } c_m \leq_s \dots \leq_s c_1 \quad (m \geq 1) \text{ and } e \neq 0, \text{ then}$$

$$3^{c_1} \oplus \dots \oplus 3^{c_m} \times 3^e \quad 3^{c_1} \oplus e.$$

(4) If $a \neq 0$ and $e_n \leq_s \dots \leq_s e_1$ ($n > 1$), then

$$a \times (3^{e_1} \oplus \dots \oplus 3^{e_n}) = (a \times 3)^{e_1} \oplus \dots \oplus (a \times 3)^{e_n}.$$

(5) If $a \equiv c$ and $b \equiv d$, then $a \times b \equiv c \times d$.

Using $+$, \times and the condition A.11 it is easy to pick out a path in \mathcal{L} for ϵ_0 . Namely using 3 for ω we just perform the operations used to define the ordinals $< \epsilon_0$. To facilitate a quick familiarity with \mathcal{L} and with the path for ϵ_0 , we list examples from ϵ_0 . We first present some functions which are useful in manipulating and representing these ordinals.

A.13 Let $p(x, y) = 2^x \cdot 3^y$, $p^{(z)}(x, y) = z$ iterations of $p(x, y)$ in the first argument, e.g. $p^{(0)}(x, y) = x$, $p^{(2)}(x, y) = p(p(x, y), y)$. Let $\exp_a(y) = a^y$ for $a, y \in \mathbb{N}$. Then $\exp_a^{(x)}(y)$ is the x -fold iteration of $\exp_a(y)$, e.g. $\exp_a^{(0)}(y) = y$.

$$\exp_a^{(3)}(y) = a^{a^{a^y}}.$$

A.14 ordinal integer representation

$0, 1, 2$	$0, 1, 2$
3	2^2
n	$\exp_2^{(n-1)}(1)$
$\omega+1$	$2 \cdot 3$
$\omega+n$	$p(\exp_2^{(n)}(1), 1)$
$\omega \cdot 2$	$2^3 \cdot 3$
$\omega \cdot n$	$p^{(n-1)}(3, 1)$

$$\begin{array}{ll}
\omega^2 & 3^2 \\
\omega^n \cdot a & p^{(a-1)}(\exp_3(\exp^{(n-1)}(1)), \exp^{(n-1)}(1)) \\
\omega^n \cdot a + m & p^{(a)}(\exp_2^{(m)}(1), \exp_2^{(n-1)}(1)) \\
\omega^\omega & 3^3
\end{array}$$

One final example,

$$\omega^\omega \cdot a_1 + \omega^{n_1} \cdot a_2 + m$$

is represented by

$$\exp_3(p^{(a_1-1)}(p^{(a_2)}(\exp_2^{(m)}(1), \exp_2^{(n_1-1)}(1)), 3)).$$

It is convenient to represent the ordinals $\alpha < \epsilon_0$ in terms of the functions $\exp_2^{(\cdot)}(\cdot)$, $\exp_3^{(\cdot)}(\cdot)$ and $p^{(\cdot)}(\cdot)$. To be somewhat precise about this we define a term A as follows.

A.15 (1) If $n \in \mathbb{N}$, then $\exp_2^{(n)}(1)$, $\exp_3^{(n)}(1)$ and 0 are terms.

(2) If $n \in \mathbb{N}$ and if A and B are terms $\neq 0$, then $\exp_3^{(n)}(A)$ and $p^{(n)}(A, B)$ are terms.

We can now prove A.16.

A.16 If $\alpha < \epsilon_0$, then α is represented by a term and the unique decomposition of α to the base ω is represented by a unique term, t_α , referred to as the term representing α .

A.17 Conversely, every term represents an ordinal $\alpha < \epsilon_0$.

We prove only A.16 since we never have occasion to use A.17. First we notice that every $n \in \mathbb{N}$ is represented by a

term; 0 represents 0, $\exp_2^{(0)}(1)$ represents 1, $\exp_2^{(1)}(1)$ represents 2. In general $\exp_2^{(n-1)}(1)$ represents n . These are defined to be the unique terms representing n . The unique term for ω is 3. Inductively, if t_α is the unique term for α , then $3^{t_\alpha} = \exp_3(\bar{\alpha})$ is made the unique term for ω^α . That it represents ω^α was proved in Schütte[48]. Now we show that if $n > 1$, then $p^{(n-1)}(\exp_3(t_\alpha), t_\alpha)$ represents $\omega^\alpha \cdot n$. It will be designated as the unique term for that ordinal. Proceed by induction on n . If $n=2$, then

$$p^{(1)}(\exp_3(\alpha), \alpha) = 2 \cdot 3^{t_\alpha} \cdot 3^{t_\alpha}$$

but

$$3^{t_\alpha} \oplus 3^{t_\alpha} = 2 \cdot 3^{t_\alpha} \cdot 3^{t_\alpha}.$$

Now suppose the result for n . We claim that $p^{(n)}(\exp_3(t_\alpha), t_\alpha)$ represents $\omega^{t_\alpha} \cdot n+1$. By the induction hypothesis

$$p^{(n-1)}(\exp_3(t_\alpha), t_\alpha) = 3^{t_\alpha} \oplus \dots \oplus 3^{t_\alpha}$$

(associativity is shown in Schütte[48]). But

$$p(p^{(n-1)}(\exp_3(t_\alpha), t_\alpha), t_\alpha) = p^{(n-1)}(\exp_3(t_\alpha), t_\alpha) \oplus 3^{t_\alpha}$$

by definition of \oplus , and is also equal to

$$(3^{t_\alpha} \oplus \dots \oplus 3^{t_\alpha}) \oplus 3^{t_\alpha}$$

by the induction hypothesis.

Finally we claim that if t_α uniquely represents α and 3^{t_β} uniquely represents ω^β , and $\alpha < \omega^\beta$, then $p^{(n)}(t_\alpha, t_\beta)$ represents $\omega^{\beta \cdot n + \alpha}$. Proceed by induction on n . For $n=1$, $\omega^{\beta + \alpha}$

is represented by $3^{t_\beta} \oplus t_\alpha = 2^{t_\alpha} \cdot 3^{t_\beta} = p(t_\alpha, t_\beta)$. Notice that this is

$$p^{(0)}(\exp_3(t_\beta), t_\beta) \oplus t_\alpha = p(t_\alpha, t_\beta).$$

Suppose for induction that

$$\omega^{\beta \cdot n+1+\alpha}$$

is represented by

$$p^{(n)}(\exp_3(t_\beta), t_\beta) \oplus t_\alpha = p^{(n+1)}(t_\alpha, t_\beta).$$

Then by the above result on a unique representation for ω^β and by definition of \oplus ,

$$p^{(n+2)}(\exp_3(t_\beta), t_\beta) \text{ represents } \omega^{\beta \cdot n+1}$$

and

$$p^{(n+2)}(\exp_3(t_\beta), t_\beta) \oplus t_\alpha \text{ represents } \omega^{\beta \cdot n+1+\alpha}.$$

By definition of \oplus

$$p^{(n+2)}(\exp_3(t_\beta), t_\beta) \oplus t_\alpha = p(p^{(n+1)}(\exp_3(t_\beta), t_\beta) \oplus t_\alpha, t_\beta).$$

So by the induction hypothesis

$$\begin{aligned} p^{(n+1)}(\exp_3(t_\beta), t_\beta) \oplus t_\alpha &= p(p^{(n+1)}(t_\alpha, t_\beta) \\ &= p^{(n+2)}(t_\alpha, t_\beta). \end{aligned}$$

Combining the above cases we can conclude that if $\alpha < \epsilon_0$ then there is a unique term t_α corresponding to α . This is because α can be represented uniquely to the base ω as $\omega^{\beta_1 \cdot a_1} + \dots + \omega^{\beta_n \cdot a_n}$ where $a_i \in \mathbb{N}$, $\beta_i < \epsilon_0$ and $\beta_i < \beta_j$ if $i < j$. And using induction on the level of β_i (ω is of level 1, ω^ω of level 2 and ω raised to the ω n times is of level n , an ordinal whose unique base ω decomposition contains a summand of level n is of level n) and applying the above

definition of unique terms on α from right to left produces a unique term for α .

q.e.d.

A.18 Given a term t_α we define the depth of t_α , n , as the number of \exp_3 , \exp_2 and p 's occurring in t_α .

A.19 For this representation of ordinals $\alpha < \epsilon_0$ write $\bar{\alpha}$ to denote the unique integer representing α . Using $\bar{\alpha}$ we can map the addresses of OM into N in a simple manner. Namely the triple $\langle \alpha, n, m \rangle$ is represented by $\bar{\alpha} \cdot 5^n \cdot 7^m$. Since for $\alpha < \epsilon_0$, $\bar{\alpha}$ does not contain a 5 or a 7, this representation is unique.

Our task now is to implement the computation process on $M_1(\Sigma_0)$ using the above representation of addresses. The only steps which require work are those involved in computing the fundamental sequences and the loading addresses.

A.20 We handle this task using the functions $p(\)$, $\exp_a(\)$, $(\)_2$ and $(\)_3$. We use $(\)_{p_i}^{(n)}$ to denote n iterations of $(\)_{p_i}$. Thus $(x)_2(0) = x$, $(x)_2(1) = \text{exponent of 2 in prime factorization of } x$, $(x)_2(2) = ((x)_2)_2$. Notice that in this notation $(\exp_2^{(n)}(x))_{2(n)} = x$

The functions $(\)_2$ and $(\)_3$ are sufficient to break down any $\bar{\alpha}$ for $\alpha < \epsilon_0$ into its component parts. To build a routine which computes $\bar{\alpha}_x$ it is only necessary to apply the functions $(\)_2$, $(\)_3$ in the proper order to break down α according to

the definition A.15. Then build a routine to compute the term determining the fundamental sequence. Finally apply $p(\)$ and $\exp_a(\)$ to build back up to $\bar{\alpha}_x$. The only bothersome part is keeping the "logic" of the breakdown straight. A detailed program is given covering this part of the computing procedure. Before presenting that program we offer an example of how it works.

A.21 Suppose the identification constant is $\alpha = \omega^3 \cdot 2 + \omega^2 + \omega$. Then

$$\bar{\alpha} = 2^{2^2 \cdot 3} \cdot 3^2 \cdot 3^{2^2} \cdot 3^{2^2}$$

which is equal to

$$p^{(2)}(p(\exp_3(1), \exp_2(1)), \exp_2^{(2)}(1)) = t_\alpha.$$

The program π_{f_α} recognizes α to be a sum. It locates the first summand (from left to right) which is $\omega^3 \cdot 2$ and it stores the sequence $2^2, 2, p$ on a list (denoted LST). The sequence will guide the program in building up α_x . The program again recognizes a sum and records $2, 1, p$ on LST. Finally the program recognizes the critical term, ω . It then forms $\exp_2^{(x)}(1)$, stores this in L and begins the process of building α_x using the list

$$\text{LST} = 2^2, 2, p, 2, 1, p$$

and

$$L = \exp_2^{(x)}(1)$$

as a guide. The p indicates that the next term to be built is p (next sequence value, first from next sequence value).

e.g. $p(1,2)$ or $p(2,2^2)$ in the above.

The details of the program for computing the downward link are given below. We do not however give the details of the copying routines, steps (c), (f) and (g) of A.6, because they are quite straight forward. (To copy the main program we merely write a routine which goes through the instructions of the program itself and loads the same operation code with appropriately changed addresses into the new program location. To load the $B()$ subprograms the program can use a schema for $B()$ which it has stored as data.)

A.22 Program for determining downward link.

Decide whether limit or sum

START	Y ← ID	
	$(Y)_2 = 0 \Rightarrow L0$	Is Y of form ω^α ? Yes, go to limit, L0.
	$(Y)_3 = 0 \Rightarrow (Pf)$	
	A ← Y	If not a limit, then go to sums and products case.
	GO TO SP1	

Logic for limit case

L0	a = 0	Initialize counter
	A ← $(Y)_3$	Reduce Y by ω -power, i.e. take " $\log_\omega()$ "
L1	a ← a + 1	Increment counter
	$(A)_2 = 0 \Rightarrow L20$	See if reduction is still a limit
	$(A)_3 = 0 \Rightarrow (Lf)$	If not an ω -limit, check for integer limit exponent, i.e. ω^n
	Exit	
	LST ← a	Record number of iterations of $\log()$ on LST
	LST ← "e"	. means to advance the list
		Put e on LST to signal type of iteration for RETURN
	GO TO SP1	

 $\log_\omega()$

L20	$(A)_3 = 0 \Rightarrow L30$	Check for form ω^1
	A ← $(A)_3$	Execute $\log_\omega()$
	GO TO L1	

 ω^ω form, this sets up ω^{x_0}

L30	a ← a - 1	
	a = 0 ⇒ L35	If a=0 do not record a or e
	LST ← a	Record information on iterations of "log"
	LST ← e	
	L ← 3^{x_0}	Load "fund seq" in L for RETURN

L35 GO TO RETURN
 $L \leftarrow x_0$
 GO TO RETURN

Set up $\omega^n \cdot x_0$

Lf $a \leftarrow a - 1$
 $a = 0 \Rightarrow$ Lf5 If $a=0$ do not load a or e
 $LST \leftarrow a$
 $\dot{LST} \leftarrow e$

 Lf5 $\dot{LST} \leftarrow (A)_2$
 $\dot{LST} \leftarrow x - 1$
 $\dot{LST} \leftarrow P$
 $\dot{} \leftarrow (A)_2$
 $L \leftarrow 3$
 GO TO RETURN

Logic for sums and products case

SP1 $a = 1$
 $W \leftarrow (A)_3$
 $B \leftarrow (A)_2$

 SP4 $(B)_3 = W \Rightarrow$ prod
 $(B)_3 = 0 \Rightarrow$ (Pf)

Load information on terms of sum

(sp) $LST \leftarrow W$
 $\dot{LST} \leftarrow a$
 $\dot{LST} \leftarrow "P"$
 $\dot{Y} \leftarrow B$
 GO TO START

Iteration loop for products

(prod) $(B)_2 = 0 \Rightarrow$ (PP)
 $a \leftarrow a = 1$
 $B \leftarrow (B)_2$
 GO TO SP4

Pure product information loading

```
(PP)  LST $\leftarrow$ W
      LST $\leftarrow$ a
      LST $\leftarrow$ "P"
      Y $\leftarrow$ B
      GO TO START
```

Special exit for ordinals of type $\omega^{\alpha+n}$

```
(pf)  LST $\leftarrow$ W
      LST $\leftarrow$ a
      LST $\leftarrow$ P
      LST $\leftarrow$ (B)2
      SPL $\leftarrow$ "1"
      GO TO RETURN
```

Logic for return

```
RETURN  LST = P  $\Rightarrow$  (P)
        LST = e  $\Rightarrow$  (L)
        LST = 0  $\Rightarrow$  (end)
```

Reassembling sums and products

```
P      . . . decrements the list
      a $\leftarrow$ LST
      L $\leftarrow$ p(a)(L, LST)
      GO TO RETURN
```

Reassembling powers

```
L      SPL = 1  $\Rightarrow$  (SPL)
      a $\leftarrow$ LST
      L $\leftarrow$ exp3(a)(L)
      GO TO RETURN
```

Encorporate term determining the fundamental sequence.

```
SPL    . . .
```

```
SPL $\leftarrow$ 0  
L $\leftarrow$ p(x0)(3L, L)  
a $\leftarrow$ LST-1  
L $\leftarrow$ exp(a)(L)  
GO TO RETURN
```

(end)

Program exit

Appendix B Estimating $\sigma f_\alpha(\)$.

B.1 $\forall \alpha \exists T(\) \exists B(\) \forall x \forall n$ if α is a limit ordinal and $\alpha < \epsilon_0$, then

$$(a) \sigma f_{\alpha_n}(x) < f_{\alpha_n}^{(T(n))}(x) \ \& \ T(\) \in E_\alpha.$$

$$(b) \sigma f_\alpha(x) < f_{\alpha_{B(x)}}^{(\bar{T}(x))}(x) \ \& \ \bar{T}(\) \in E_\alpha \ \& \ B(\) \in E_\alpha.$$

We actually prove a stronger statement, namely that $B(\)$ can be chosen as

$$B(\) = \begin{cases} x & \text{if } \alpha > \omega \\ x+2 & \text{if } \alpha = \omega \end{cases}$$

and that $\bar{T}(\)$ and $T(\)$ can be chosen in E_2 if $\alpha > \omega$. We recall from 6.31 step 3 part (b) that (b) above implies $\exists C$

$$(c) \sigma f_\alpha(x) < f_\alpha^{(C)}(x).$$

The proof of B.1 is based on the idea that the steps in the computation of $f_\alpha(\)$ according to the method of Appendix A can be classified into two categories. First there are the steps associated with carrying out the actual iterations and diagonalizations. Second there are the steps associated with generating subprograms, i.e. steps which involve calculating the downward links, modifying the loading instructions, and loading the subprograms. The second category of steps is said to constitute the overhead computing. According to the computation procedure overhead computing occurs only at limit ordinals. In terms of that procedure, the function, $L_\alpha(x)$ = number of overhead computing steps of

level α (for α a limit ordinal), is precisely defined if we agree to regard all steps up to and including transfer of control from π_{f_α} to $\pi_{f_{\alpha_x}}$ as overhead steps.

In terms of counting σ_α , the overhead steps are those used to generate $\pi_{f_{\alpha_n}}$. By establishing B.1 we are showing that $\pi_{f_{\alpha_n}}$ can be generated as primitive at level α in the hierarchy. For simplicity in the work below we will use the notation $\sigma f_\alpha()$ in place of $\sigma \pi_{f_\alpha}()$.

We notice immediately that

$$B.2 \quad \sigma f_\alpha(x) = L_\alpha(x) + \sigma f_{\alpha_x}(x).$$

A useful notion for the proof of B.1 is that of the total overhead computing time, σ_0 , for $f_\alpha()$ which is defined inductively as follows

$$B.3 \quad \sigma_0 f_0(x) = 0 \quad \forall x$$

$$\sigma_0 f_{\alpha+1}(x) = \sigma_0 f_\alpha(x) \quad \forall x$$

If $\alpha_x \rightarrow \alpha$, then $\sigma_0 f(x) = L_\alpha(x) + \sigma_0 f_{\alpha_x}(x)$.

Now define

$$B.4 \quad \sigma_1 f_\alpha(x) = \sigma f_\alpha(x) - \sigma_0 f_\alpha(x).$$

We then have

$$B.5 \quad \sigma f_\alpha(x) = \sigma_0 f_\alpha(x) + \sigma_1 f_\alpha(x)$$

and at limit ordinals by B.5 and B.2

$$B.6 \quad \sigma f_\alpha(x) = L_\alpha(x) + \sigma_0 f_{\alpha_x}(x) + \sigma_1 f_{\alpha_x}(x).$$

Equation B.6 is the basis of the proof procedures for B.1.

Functions $s_\alpha(\cdot)$ and $t_\alpha(\cdot)$ will be found such that

B.7 For $\forall x \forall n \forall \alpha$ if α is a limit ordinal $< \epsilon_0$, then

$$(a) \quad \sigma_1 f_{\alpha_n}(x) < f_{\alpha_n}^{(t_\alpha(n))}(x) \quad t_\alpha(\cdot) \in E_2$$

$$(b) \quad \sigma_0 f_{\alpha_n}(x) < f_{\alpha_n}^{(s_\alpha(n))}(x) \quad s_\alpha(\cdot) \in E_2$$

and a constant L_α will be found such that

$$(c) \quad L_\alpha(x) = f_{\alpha_{B(x)}}^{(L_\alpha)}(x).$$

Then by B.5, B.6 and B.7 defining $T_\alpha(x) = s_\alpha(x) + t_\alpha(x) + a$ and $\bar{T}_\alpha(x) = T_\alpha(x) + (a + L_\alpha)$ we have

B.8 For $\forall x \forall n \forall \alpha$, if α is a limit ordinal $< \epsilon_0$,

$$(a) \quad \sigma f_{\alpha_n}(x) < f_{\alpha_n}^{(s_\alpha(n))}(x) + f_{\alpha_n}^{(t_\alpha(n))}(x) < f_{\alpha_n}^{(T_\alpha(n))}(x)$$

for $T_\alpha(\cdot) \in E_2$ and

$$(b) \quad \sigma f_\alpha(x) < f_{\alpha_{B(x)}}^{(\bar{T}_\alpha(x))}(x) \quad \bar{T}_\alpha(\cdot), B(\cdot) \in E_2.$$

Thus showing B.7 will complete the proof of B.1.

In what follows we show B.7 (a) then B.7 (b) and in the course of showing (b) we get (c). The techniques in the proofs involve only simple bounding procedures. We could find much better estimates for σ_0 and σ_1 , but that would involve even more tedious and unattractive work. We do not offer completely formal inductive proofs at every point in the analysis, but it is clear how the formal proof would go.

First we define some important constants, a , p , c_1 , c_0 , whose denotation remains fixed throughout this appendix.

Since $x+y \in \Sigma_0$, the definition of $f_0(\)$ implies $\exists a$ such that

$$B.9 \quad x+y < f_0^{(a)}(\max\{x,y\}) \quad \forall x,y;$$

also since $x.y.z \in \Sigma_0^*$ $\exists p$ such that $p > a$ and

$$B.10 \quad x.yz < f_0^{(p+1)}(\max\{x,y,z\}).$$

Let c_1 be the number of lines in a $B(\)$ iteration subprogram ($c_1 = 8$ in Appendix A) and given $f_0(\)$ pick c_0 such that

$$B.11 \quad c_0 > \max\{c_1, a, \sigma f_0(\)\}.$$

Notice then

$$f_0(x) << f_0^{(\bar{c}_0)}(x) \quad \forall x \text{ for some } \bar{c}_0 \leq c_0$$

since $f_0^{(y)}(0) > y$.

According to the computation procedure described in Appendix A

$$B.12 \quad \sigma f_1(x) = \sigma f_0(x) + c_1 \cdot \sigma f_0(f_0(x)) + \dots + c_1 \cdot \sigma f_0(f_0^{(x-1)}(x)) \\ < c_1 \cdot (x-1) \cdot f_0^{(\bar{c}_0+x-1)}(x).$$

Recalling the definition of p and noticing $f_0^{(\bar{c}_0+x-1)}(x) > c_1 \cdot (x-1)$ it follows from B.12 that

$$\sigma f_1(x) < f_0^{(\bar{c}_0+p+x)}(x) << f_1(\bar{c}_0+p+x) < f_1^{(\bar{c}_0+p)}(x).$$

These are simple estimates which could be much improved if it were necessary (e.g. since $f_1^{(y)}(0) > f_1(y) \gg f_0(y)$, \bar{c}_0 could be reduced eventually to 1 for some $f_n(\)$ $n << \bar{c}_0$).

Continuing

$$B.13 \quad \sigma_1 f_2(x) = \sigma_1 f_1(x) + \dots + c_1 \cdot \sigma_1 f_1(f_1^{(x-1)}(x))$$

$$< c_1 \cdot (x-1) \cdot f_1^{(\bar{c}_0+p+x-1)}(x),$$

and as before

$$\sigma_1 f_2(x) << f_1^{(\bar{c}_0+2p+x)}(x) < f_1^{(\bar{c}_0+2p)}(x).$$

So that in general

$$B.14 \quad \sigma_1 f_n(x) << f_n^{(\bar{c}_0+np)}(x).$$

Thus at the limit stage,

$$B.15 \quad \sigma_1 f_\omega(x) = \sigma_1 f_x(x) << f_x^{(\bar{c}_0+xp)}(x).$$

Putting $d_\omega(x) = \bar{c}_0 + xp$ we can state this as

$$B.16 \quad \sigma_1 f_\omega(x) << f_x^{(d(x))}(x).$$

We now set out to find a d_ω such that $\sigma_1 f_\omega(x) < f_\omega^{(d_\omega)}(x)$.

Clearly $d_\omega(\) \in E_1$ so that $\exists c'$ such that

$$B.17 \quad c_0 + x \cdot p < f_1^{(c')}(x) \quad \forall x.$$

In fact we claim c can be chosen $\max\{\bar{c}_0, p\} \leq c' \leq c_0 + p$ because if $c = \max\{c_0, p\}$, then

$$f_1^{(c)}(x) > f_1^{(x+c)}(x+c) > f_0^{(x+c)}(x+c) > c \cdot x$$

and

$$c + c \cdot x < f_0^{(a)}(f_0^{(x+c)}(x+c)) = f_0^{(a+c+x)}(x+c).$$

So since $\bar{c}_0 + p > c + a$ the claim is justified. We thus have from B.16 that

$$B.18 \quad \sigma_1 f_\omega(x) << f_x^{(f_1^{(c')}(x))}(x),$$

and we notice that for $x \geq c' + 1$

$$B.19 \quad f_x^{(f_1^{(c')}(x))}(x) << f_{2+x}^{(f_1^{(c')}(x))} < f_{2+x}^{(c'+1)}(x) << f_{3+x}(x).$$

Thus $\sigma_1 f_\omega(x) \ll f_{x+3}(x)$ if $x > c'+1$. So since $c' > 2$

$$\begin{aligned} \text{B.20 } \sigma_1 f_\omega(x) &\ll f_\omega(x+c'+1) \\ &< f_\omega^{(c'+1)}(x) \\ &< f_\omega^{(\bar{c}_0+p)}(x) \quad \forall x. \end{aligned}$$

So we have found that $d_\omega = \bar{c}_0 + p$ is an acceptable choice.

The same analysis now shows that

$$\text{B.21 } \sigma_1 f_{\omega \cdot n+1}(x) < f_{\omega \cdot n+1}^{(\bar{c}_0 + (n+1) \cdot p)}(x)$$

by taking $\bar{c}_0 + np$ as \bar{c}_0 in the above analysis. (In general,

$$\begin{aligned} \sigma_1 f_{\omega \cdot n+m}(x) &< f_{\omega \cdot n+m}^{(\bar{c}_0 + n \cdot p + m \cdot p)}(x). \text{) Thus} \\ \sigma_1 f_\omega(x) &< f_{\omega \cdot n}^{(\bar{c}_0 + xp)}(x). \end{aligned}$$

So that as before

$$\sigma_1 f_\omega(x) < f_{\omega \cdot x}^{(f_1^{(c')}(x))}(x) < f_{(\omega \cdot x)+3}(x)$$

if $x > c'+1$. Since by 6.24

$$f_{\alpha_x+n}(x) < f_\alpha(x) \quad x > n$$

It follows that

$$\text{B.22 } \sigma_1 f_\omega(x) \ll f_\omega(x+c'+1) < f_\omega^{(\bar{c}_0+p)}(x) \quad \forall x.$$

In general

$$\text{B.23 } \sigma_1 f_{\omega^{n_1} \cdot a_1 + \dots + \omega^{n_s} \cdot a_s}(x) < f_{\omega^{n_1} \cdot a_1 + \dots + \omega^{n_s} \cdot a_s}^{(c_0 + a_1 \cdot p + \dots + a_s \cdot p)}(x)$$

the principle being if $\sigma_1 f_{\alpha+\omega^m}(x) < f_{\alpha+\omega^m}^{(d+p)}(x)$, ($d > c_0$), then

by taking $\bar{c}_0 = d+p$ and starting the process with $f_{\alpha+\omega^m}(\)$

we obtain

$$\sigma_1 f_{\alpha + \omega^m}(x) < f_{\alpha + \omega^m \cdot n}^{(d+n \cdot p)}(x).$$

B.24 We can now analyse $f_\alpha(\)$ for $\alpha > \omega^\omega$ using the principle that

$$\sigma_1 f_{\omega^\beta}(x) < f_{\omega^\beta}^{(c_0+p)}(x) \quad \forall x \quad \forall \beta < \varepsilon_0.$$

We prove this by a simple induction. Suppose the result for β . The $\omega^\beta \cdot n \rightarrow \omega^{\beta+1}$ as $n \rightarrow \omega$ so that by the above principle taking c_0+p and c_0 and repeating the process up to ω^β we have

$$\sigma_1 f_{\omega^\beta \cdot x}(x) < f_{\omega^\beta \cdot x}^{(\bar{c}_0+xp)}(x).$$

And as before

$$B.25 \quad \sigma_1 f_{\omega^{\beta+1}}(x) < f_{\omega^{\beta+1}}^{(\bar{c}_0+p)}(x) \quad \forall x.$$

If β is a limit ordinal, then assume the result for all $\alpha < \beta$, thus for all β_x such that $\beta_x \rightarrow \beta$. Then

$$B.26 \quad \sigma_1 f_{\omega^\beta}(x) = \sigma_1 f_{\omega^{\beta_x}}(x) < f_{\omega^{\beta_x}}^{(\bar{c}_0+p)}(x) < f_{\omega^\beta}^{(\bar{c}_0+p)}(x) \quad \forall x.$$

This concludes the analysis of σ_1 because given $f_\alpha(\)$, the function $t_\alpha(\)$ either has the form $t_\alpha = c_0 + d \cdot p + x \cdot p$ or $t_\alpha(x) = c_0 + p$. In either case $t_\alpha(\) \in E_2$ as was to be shown. Thus B.7 part (a) follows.

B.27 We now turn to a look at the overhead computing and $\sigma_0 f_\alpha(\)$. The first overhead steps arise in computing $f_\omega(\)$. The overhead, $L_\omega(\)$, satisfies $L_\omega(x) < d_1 \cdot \exp_2^{(x)}(1) + x \cdot b + d_2$ where b is the number of steps required to load

an iteration subprogram (we will look at a general method of estimating $L_\alpha(\cdot)$ below). Thus $L_\omega(\cdot) < E_2$ so that $\exists m$ such that

$$L_\omega(x) < f_2^{(m)}(x)$$

and $\exists s_\omega$ such that

$$\sigma_0 f_\omega(x) = L_\omega(x) < f_{x+2}^{(m)}(x) < f_\omega^{(s_\omega)}(x) \quad \forall x.$$

Now applying techniques used above to the iteration process we can infer

$$\begin{aligned} \text{B.28} \quad \sigma_0 f_{\omega+1}(x) &= \sigma_0 f_\omega(x) + \sigma_0 f_\omega(f_\omega(x)) + \dots + \sigma_0 f_\omega(f_\omega^{(x-1)}(x)) \\ &< (x-1) \cdot f_\omega^{(s_\omega+x-1)}(x) \\ &< f_\omega^{(s_\omega+p+x)}(x) \end{aligned}$$

so that

$$\sigma_0 f_{\omega+1}(x) < f_{\omega+1}^{(s_\omega+p)}(x).$$

Continuing as before we generalize to

$$\text{B.29} \quad \sigma_0 f_{\omega+(n+1)}(x) < f_{\omega+n}^{(s_\omega+n \cdot p+x)}(x) < f_{\omega+(n+1)}^{(s_\omega+n \cdot p)}(x).$$

At stage $\omega \cdot 2$ we have an added complication. We can conclude as before that

$$\text{B.30} \quad \sigma_0 f_{\omega+x}(x) < f_{\omega+x}^{(s_\omega+x \cdot p)}(x) < f_{\omega \cdot 2}^{(s_\omega+p)}(x).$$

But now

$$\sigma_0 f_{\omega \cdot 2}(x) = L_{\omega \cdot 2}(x) + \sigma_0 f_{\omega+x}^{(s_\omega+x \cdot p)}(x).$$

We can easily show that $\exists s_{\omega \cdot 2}$ such that

$$L_{\omega \cdot 2}(x) < f_{\omega \cdot 2}^{(s_{\omega \cdot 2})}(x).$$

So that combining the above two lines yields

$$\text{B.31} \quad \sigma_0 f_{\omega \cdot 2}(x) < f_{\omega \cdot 2}^{(s_{\omega \cdot 2})}(x) + f_{\omega \cdot 2}^{(s_\omega+p)}(x) <$$

$$< f_{\omega \cdot 2}^{(a+\max\{s_{\omega \cdot 2}, s_{\omega}+p\})}(x).$$

Now assuming $s_{\omega} < s_{\omega \cdot 2}$ we have

$$\text{B.32 } \sigma_0 f_{\omega \cdot 2}(x) < f_{\omega \cdot 2}^{(a+p+s_{\omega \cdot 2})}(x).$$

Starting the analysis again with $a+p+s_{\omega \cdot 2}$ for s_{ω} and continuing inductively with the assumption that

$$\text{B.33 } s_{\omega \cdot n} < s_{\omega \cdot n+1}$$

we have

$$\text{B.34 } \sigma_0 f_{\omega \cdot n}(x) < f_{\omega \cdot n}^{(n(a+p)+s_{\omega \cdot n})}(x) \quad \forall x.$$

Now we want to show that $\lambda x x(a+p)+s_{\omega \cdot n} \in E_2$ (this will be done in more generality below). Then finding s_{ω^2} such that

$$L_{\omega^2}(x) < f_{\omega^2}^{(s_{\omega^2})}(x)$$

and finding s_{ω^2} such that

$$f_{\omega \cdot x}^{(x(a+p)+s_{\omega \cdot x})}(x) < f_{\omega^2}^{(a+p+s_{\omega^2})}(x) \quad \forall x$$

we have

$$\begin{aligned} \text{B.35 } \sigma_0 f_{\omega^2}(x) &< f_{\omega^2}^{(s_{\omega^2})}(x) + f_2^{(a+p+s_{\omega^2})}(x) < \\ &< f_{\omega^2}^{(a+\max\{s_{\omega^2}, (a+p)+s_{\omega^2}\})}(x). \end{aligned}$$

Now assuming

$$\text{B.36 } s_{\omega^2} > s_{\omega^2}$$

we have

$$\text{B.37 } \sigma_0 f_{\omega^2}(x) < f_{\omega^2}^{(2a+p)+s_{\omega^2}}(x).$$

Given the assumptions

$$B.38 \quad (1) \quad s_{\alpha_n} \geq s^{\alpha_n}$$

$$(2) \quad \lambda x s_{\alpha_x} \in E_{\alpha}$$

$$(3) \quad s_{\alpha+\omega \cdot n} < s_{\alpha+\omega \cdot n+1}$$

we find that the form $(2a+p)+s_{\alpha_x}$ is the basic form for the σ_0 analysis in the same sense that (\bar{c}_0+p) was the basic form for the σ_1 analysis (with $(2a+p)$ corresponding to p and s_{α_x} corresponding to \bar{c}_0). To see this, take $a+(a+p)+s_{\alpha_x}^2$ for s_{ω} in the preceding analysis. Use the above assumptions, and proceed with an inductive argument as in B.19 - B.26 to conclude

$$\sigma_0 f_{\omega^{n_1 \cdot a_1 + \dots + \omega^{n_r \cdot a_r}}}(x)$$

is less than

$$\left(\sum_{i=1}^r a_i \cdot (2a+p) + s_{\omega^{n_1 \cdot a_1 + \dots + \omega^{n_r \cdot a_r}}} \right) f_{\omega^{n_1 \cdot a_1 + \dots + \omega^{n_r \cdot a_r}}}(x).$$

We can carry this over to $n_1 = \beta_1$, $\beta < \epsilon_0$ just as in B.24.

B.39 In order to justify the assumptions (1), (2), (3) which are sufficient to complete our analysis of $\sigma_0 f_{\alpha}(\)$, we must examine the form of the $L_{\alpha}(\)$ $\omega \leq \alpha < \epsilon_0$. The overhead computing procedure involves the following basic steps:

1. breaking down α , this involves the functions $(\)_2(\)$ and $(\)_3(\)$.

2. computing $\exp_2^{(x)}(1)$,
3. building $\bar{\alpha}$ back up with $\exp_2^{(x)}(1)$ as a new term,
this involves $\exp_2^{()}()$, $\exp_3^{()}()$ and $p^{()}()$,
4. copying main program which is c_n lines long,
5. copying iteration subprograms each b lines long.

We can estimate the number of steps involved in terms of $\bar{\alpha}$ and x if we have estimates for $(x)_{2(n)}$, $(x)_{3(n)}$, $\exp_2^{(x)}(1)$, $\exp_3^{(x)}(1)$ and $p^{(z)}(x, y)$. Very crude estimates are given by the following (recalling that proper subtraction is a basic operation).

$$B.40 \quad (a) \quad (x)_{2(n)} < 2 \cdot (c_1 \cdot x) + 3 \cdot (n-x)$$

$$(b) \quad (x)_{3(n)} < 2 \cdot (c_1 \cdot x) + 3 \cdot (n-x)$$

To see (a) and (b) we consider the program which computes $(x)_y$.

```

      b ← 0
      x = 0 ⇒ error
      x = 1 ⇒ (4)
(1)  a ← 0
(2)  a ← a+1
      x ÷ y · a = 0 ⇒ (3)
      TO (2)
(3)  x = y · a ⇒ (5)
(4)  OUT ← b
(5)  x ← a
      b ← b+1
      TO (1)

```

Since $+$, \div , and x are primitive and since $x/d + x/d^2 + \dots + x/d^n < x$ for $d = 2$ or 3 we have (a) and (b). By similar analysis it is clear that the following hold.

$$(c) \quad \sigma \exp_2^{(x)}(y) < c_2 \cdot \exp_2^{(x-1)}(y) < \exp_2^{(x)}(y)$$

$$(d) \quad \sigma \exp_3^{(x)}(y) < c_2 \cdot \exp_3^{(x-1)}(y) < \exp_3^{(x)}(y)$$

$$(e) \quad \sigma p^{(z)}(x, y) < c_3 \cdot (p^{(z-1)}(x, y) + y) < p^{(z)}(x, y)$$

With these estimates at hand it is easy to provide a crude but workable bound on $L_\alpha(\cdot)$. We associate a bound with each of the steps 1. - 5. of B.39: 1. $2 \cdot c_1 \cdot \bar{\alpha}$, 2. $\exp_2^{(x)}(1)$, 3. $c_4 \cdot \bar{\alpha}$, 4. c_m , 5. $b \cdot x$. Taking $c = \max\{c_1, \dots, c_4\}$ and summing we find

$$B.41 \quad L_\alpha(x) < 3 \cdot c \cdot \bar{\alpha} + \exp_2^{(x)}(1) + b \cdot x + c_m.$$

Recalling from A.16 the construction of the α , we observe that $\bar{\alpha}_x$ is at worst in E_2 for all $\alpha < \epsilon_0$. Thus B.41 yields B.7 part (c).

Now to verify assumptions (1) to (3) of B.38 we must define s_α and s^α . We can easily see that an adequate (but crude) choice for s_α is just $\bar{\alpha} + m$ where $m = 3 + c + b + c + e$ and e satisfies $\exp_2^{(x)}(1) < f_2^{(e)}(x) \quad \forall x$ since

$$B.42 \quad L_\alpha(x) < f_2^{(\bar{\alpha} + m)}(x).$$

For reasons which will be clear below we include another term, $3 \cdot c \cdot (M + a)$, in s_{α_x} . Thus define

$$B.43 \quad s_{\alpha_x} = 3 \cdot c \cdot (M + a) \cdot \bar{\alpha}_x + m \in E_2.$$

Notice, assumption (2) of B.38 is justified.

Now to define s^α , notice that if p satisfies

$$B.44 \quad \alpha_x < f_2^{(p_\alpha)}(x),$$

then

$$\begin{aligned}
 \text{B.45} \quad f_{\alpha_x}^{(d+s_{\alpha_x})}(x) &< f_{\alpha_x}^{(d+f_2^{(p_\alpha+m)}(x))}(x) \\
 &< f_{\alpha}^{(d+p_\alpha+m+2)}(x)
 \end{aligned}$$

where the last inequality follows by the methods of B.21 - B.24. So

$$\text{B.46} \quad s_{\alpha}^{\alpha} = p_{\alpha} + m + 2.$$

We now find a uniform method for choosing p and then show that $s_x \geq s_y$ and in the process that $s_x < s_y$ if $x < y$. The idea here is that we want to choose p_{α} to be small. The only fact we must really show to prove $s_{\alpha} > s^{\alpha}$ is that there is a uniform method of choosing s_{α} large and s_{α} small simultaneously. The fact is intuitively clear because $\lambda x s_{\alpha_x} \in E_2$ and we can use any $f_{\alpha}(\cdot)$ to majorize the s_{α_x} . The details which follow are tedious but quite elementary. (It should be noted that even if $s_{\alpha} > s^{\alpha}$ did not hold, we could still prove the main result on $\sigma_0 f_{\alpha}(\cdot)$, but the arguments might be even more tedious.)

The numbers $\bar{\alpha}$ and the functions $\bar{\alpha}_x$ arise from compositions of $p(\cdot)$, $\exp_2^{(1)}(1)$ and $\exp_3(\cdot)$, e.g.

$$\omega^{\omega^2} \cdot 3 + \omega^3 \cdot 2 + \omega^2 \cdot 3 + \omega$$

is equal to

$$p^{(2)}(p^{(1)}(p^{(2)}(\exp_3(1), \exp_2(1)), \exp_2^{(2)}(1)), \exp_3(2))$$

We know $\exists \bar{p}, e_2, e_3$ such that

$$B.47 \quad (a) \quad p^{(x)}(q, r) < f_2^{(p)}(x+q+r)$$

$$(b) \quad \exp_2^{(x)}(s) < f_2^{(e_2)}(x+s) \text{ and}$$

$$\exp_3^{(x)}(s) < f_2^{(e_3)}(x+s)$$

B.48 Let $M = \max\{\bar{p}, e_1, e_2\}$ and consider a composition of the following type

$$p^{(x)}(p^{(a_1)}(q, t), r).$$

Then using the inequalities above

$$B.49 \quad p^{(x)}(p^{(a_1)}(q, t), r) < f_2^{(x+f_2^{(M)}(a_1+s+t)+r)} \\ < f_2^{(2 \cdot M+a)}(x+a_1+q+t+r).$$

B.50 Likewise for compositions of the types $\exp_3^{(x)}(A)$ and $p^{(x)}(A, B)$ where A and B are $\exp_2^{(\quad)}(\quad)$, $\exp_3^{(\quad)}(\quad)$ or $p^{(\quad)}(\quad)$. Continuing inductively on the levels of composition we can conclude that

$$B.51 \quad \bar{\alpha}_x < f_2^{(n \cdot M + n \cdot a)}(x + \sum_{i=1}^n a_i),$$

where n is the level of nesting in $\bar{\alpha}$ (defined in A.18 as the total number of $p^{(\quad)}$, $\exp_2^{(\quad)}$ and $\exp_3^{(\quad)}$'s in representation of $\bar{\alpha}$) and a_i are iteration parameters of $p^{(\quad)}$ and $\exp^{(\quad)}$.

Letting $A_n = \sum_{i=1}^n a_i$ we have

$$B.52 \quad 3 \cdot c \cdot (M+a) \cdot \bar{\alpha}_x < f_2^{((n+2) \cdot (M+a) + A_n + 3 \cdot c)}(x) \quad \forall x.$$

Thus put $p_{\alpha} = n \cdot (M+a) + A_n$.

We now turn to showing that $s_{\alpha} > s^{\alpha}$. We have

$$3 \cdot c \cdot (M+a) \cdot \bar{\alpha} + m = s_{\alpha}$$

and

$$s^{\alpha} = p + m + 2.$$

We check that

$$3 \cdot c \cdot (M+a) \cdot \bar{\alpha} \geq p_{\alpha} + 2 = (n+2) \cdot (M+a) + A_n + 3 \cdot c + 2.$$

Proceeding by induction, for $n=1$ the least transfinite $\bar{\alpha}$

is $3 = \exp_3(1)$ so we have

$$3 \cdot c \cdot (M+a) \cdot 3^x \geq 3 \cdot (M+a) + x + 3 \cdot c + 2 \quad \text{for } x = 1, 2, 3, \dots$$

Since $(M+a) > 3$ and $c > 1$, taking $x = 1$

$$3 \cdot 3 \cdot c \cdot (M+a) > 3 \cdot c \cdot (M+a) + 3 \cdot c + 3 > 3 \cdot (M+a) + 3 \cdot c + 3$$

so clearly the result holds for all $x \geq 1$.

Now suppose the result for n . Then α has the form

$\exp_3^{(x)}(A)$ or $p^{(x)}(A, B)$ where A and B are level n expressions.

In the case of $\exp_3^{(x)}(A)$, we have by the induction hypothesis

$$3 \cdot c \cdot (M+a) \cdot A \geq (n+2) \cdot (M+a) + A_n + 3 \cdot c + 2$$

where clearly $A \geq A_n$ and $A \geq 3$. Thus as above for $x = 1, 2, \dots$

$$3 \cdot c \cdot (M+a) \cdot \exp_3^{(x)}(A) \geq (n+3) \cdot (M+a) + (A_n + x) + 3 \cdot c + 2.$$

In the case $p^{(x)}(A, B)$ we know

$$3 \cdot c \cdot (M+a) \cdot A \geq (n_1+2) \cdot (M+a) + A_n + 3 \cdot c + 2$$

$$3 \cdot c \cdot (M+a) \cdot B \geq (n_2+2) \cdot (M+a) + B_n + 3 \cdot c + 2$$

where $n_1 + n_2 \leq n$,

To show

$$3 \cdot c \cdot (M+a) \cdot p^{(x)}(A, B) > (n+2) \cdot (M+a) + (A_n + B_n + x) + 3 \cdot c + 2$$

just notice

$$3 \cdot c \cdot (M+a) \cdot p^{(x)}(A, B) \geq 3 \cdot c \cdot (M+a) \cdot (A+B) + 2 \cdot (M+a) + x$$

$$\geq (n_1+n_2+4) \cdot (M+a) + (A_n+B_n+x) + 6 \cdot c + 4$$

$$\geq (n+2) \cdot (M+a) + (A_n+B_n+x) + 3 \cdot c + 2.$$

We have now verified assumptions (1) and (2) of B.38.

To verify (3), that $s_{\alpha+\omega \cdot n} < s_{\alpha+\omega \cdot n+L}$ we need only verify that

$$\overline{\alpha+\omega \cdot n} < \overline{\alpha+\omega \cdot n+1}.$$

This is easy since $\overline{\omega \cdot n} = p^{(n-1)}(3,1) < p^{(n)}(3,1) = \overline{\omega \cdot n+1}$ and to form $\overline{\alpha+\omega \cdot n}$ we can use $\overline{\alpha+\omega \cdot n}$ which we see by induction satisfies

$$a < b \implies c \oplus a < c \oplus b$$

This concludes the case for $\sigma_1 f_\alpha(\)$, namely B.7 part (b), and we recall that B.41 gives part (c). This concludes proof of B.7 and thus by B.8, B.1 is proved.

q.e.d.

Appendix C Arithmetizing the RASP

In this appendix we prove the following theorem referred to in Chapter 7.

C.1 Normal Form Theorem for RASP, $M_1(\Sigma)$, Computable functions. There exist predicates $T_m^S(\)$ elementary in $S = h_1(\), \dots, h_n(\)$ and there exists an elementary function $U(\)$ such that if $g(\) : N^n \rightarrow N$ and $g(\)$ is $M_1(\Sigma_S)$ computable, then $\exists e$ such that

- (a) $g(X) = U(\mu y T_{n+2}^S(e, X, y)) \quad X \in N^n.$
- (b) if there exists a program π_g for $g(\)$ with respect to Σ such that $\sigma \pi_g(X) < h(X) \quad \forall X \in N^n$, then $\exists b(\) \in \mathcal{E}(h(\))$ such that $g(X) = U(\mu y \leq b(X) T(e, X, y))$ so that $g(\) \in \mathcal{E}(h(\))$,
- (c) applying the above for $\Sigma_S = \Sigma_0$, if $g(\) \in E_\alpha$, then $g(\) \in \mathcal{E}(f_\alpha(\))$,

We prove this theorem in the standard manner, by arithmetizing the theory of RASP, $M_1(\Sigma)$, computability. The

T-predicate here will have the meaning $T(e, X, y)$ iff e is the number of an initial condition and y is the number of a sequence of instantaneous descriptions for a computation $\sigma_0, \sigma_1, \dots, \sigma_n, \dots$ which satisfies the initial condition e at X on a RASP $M_1(\Sigma_S)$.

C.2 The initial condition for a RASP is the triple $\langle D, \pi, V \rangle$ where D is the array of input locations, π is a program and V is the array of output locations. For

$g(\cdot):N^n \rightarrow N$, D has the form (d_1, \dots, d_n) and V has the form v_1 .

We first arithmetize the notion of a program. Suppose $\Sigma_S = \{+, -, \times, T, C, h_1(\cdot), \dots, h_q(\cdot)\}$, and suppose that the standard designation (c.f. 2.3 and 2.14) assigns operation codes (op codes) to the instructions by the correspondence

$$\begin{array}{ccccccc} + & - & \times & T & C & h_1(\cdot) & \dots, h_q(\cdot) \\ 2 & 3 & 5 & 7 & 11 & 13 & p_{4+q} \end{array}$$

To an instruction $h_i(x_1, \dots, x_s, y)$ is assigned the number

$$2^{\bar{h}_i} 3^{x_1} \dots p_{s+1}^y \text{ where } \bar{h}_i \text{ is the op code.}$$

Given a program $\pi = \langle p(\cdot), a_0, e_1, \dots, e_m \rangle$ it will be represented by $\bar{\pi} =$

$$2^{\langle\langle a_0, p(a_0) \rangle\rangle} 3^{\langle\langle a_1, p(a_1) \rangle\rangle} \dots p_{n-1}^{\langle\langle a_{n-1}, p(a_{n-1}) \rangle\rangle} 2^{e_1} \dots p_n^{e_m}$$

where $\langle\langle x, y \rangle\rangle = 2^x \cdot 3^y$.

To represent the initial condition $\langle D, \pi, V \rangle$ we first represent $D = (d_1, d_2, \dots, d_n)$ by $\bar{D} = 2^{d_1} 3^{d_2} \dots p_{n-1}^{d_n}$, and then the initial condition is represented by $2^{\bar{D}} 3^{\bar{\pi}} 5^V$.

C.3 We now consider the representation of a computation, which is a sequence of states, $\sigma_0, \sigma_1, \dots, \sigma_n, \dots$. We actually represent a sequence of pairs called instantaneous descriptions $\langle a_i, \langle \bar{m}_i, m_i \rangle \rangle$ where a_i is the control location of σ_i and $\langle \bar{m}_i, m_i \rangle$ is a representation of a finite section of memory. The only memory we need represent is the part that is active during the computation. So we start with the

registers holding the program and then add to this segment any registers referred to in the computation. Since $M_1()$ is finitely determined, the memory represented in $\sigma_0, \sigma_1, \dots, \sigma_n$ is only finite. Let

$$m_1 = p_0^{b_1} \cdot p_1^{b_2} \cdot \dots \cdot p_{\bar{m}_1-1}^{b_{\bar{m}_1}}$$

for some integer \bar{m}_1 . The number \bar{m}_1 bounds the memory actually needed at state σ_1 and m_1 is a numerical representation of the segment of core containing the addresses $0, 1, \dots, \bar{m}_1-1$ and for which register $r < \bar{m}_1$ contains b_{r+1} . Thus $\langle \bar{m}_1, m_1 \rangle$ represents the active memory at σ_1 .

C.4 We next consider a notion of cause and effect between id's (instantaneous descriptions), i.e., we define $C_i \rightarrow C_{i+1}$ for C_i, C_{i+1} id's and we say that C_i causes C_{i+1} or C_i yields C_{i+1} . Suppose $C_i = \langle a_i, \langle \bar{m}_1, m_1 \rangle \rangle$, then $C_i \rightarrow C_{i+1} = \langle a_{i+1}, \langle \bar{m}_{i+1}, m_{i+1} \rangle \rangle$ iff $a_i < \bar{m}_1$ and

(a) if $(m_1)_{a_i} = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3}$, then

$$a_{i+1} = a_i + 1 \text{ and}$$

$$\bar{m}_{i+1} = \max \{ m_1, x_1, x_2, x_3 \} \text{ and}$$

$$(m_{i+1})_j = (m_1)_j \text{ if } j < \bar{m}_1 \text{ and } j \neq x_3$$

$$(m_{i+1})_{x_3} = (m_1)_{x_1} + (m_1)_{x_2}$$

$$(m_{i+1})_j = 0 \text{ if } \bar{m}_1 < j < \bar{m}_{i+1} \text{ and } j \neq x_3.$$

(b) if $(m_1)_{a_i} = 2^3 \cdot 3^{x_1} \cdot 5^{x_2} \cdot 7^{x_3}$, then $a_{i+1} = a_i + 1$ and

$$m_{i+1} = \max \{m_i, x_1, x_2, x_3\} \text{ and}$$

$(m_{i+1})_j$ is just as above replacing $+$ by \div .

(c) if $(m_i)_{a_i} = 2^5 \cdot 3^{x_1} \cdot 5^{x_2} \cdot 7^{x_3}$, then repeat the above replacing \div by \times .

(d) if $(m_i)_{a_i} = 2^7 \cdot 3^{x_1} \cdot 5^{x_2}$, then

$$a_{i+1} = a_i + 1 \text{ and}$$

$$m_{i+1} = \max \{m_i, x_1, x_2\} \text{ and}$$

$$(m_{i+1})_j = (m_i)_j \text{ if } j < \bar{m}_i \text{ and } j \neq x_2$$

$$(m_{i+1})_{x_2} = (m_i)_{x_1}$$

$$(m_{i+1})_j = 0 \text{ if } \bar{m}_i < j < \bar{m}_{i+1} \text{ and } j \neq x_3.$$

(e) if $(m_i)_{a_i} = 2^{11} \cdot 3^{x_1} \cdot 5^{x_2} \cdot 7^{x_3}$, then

$$\text{if } (m_i)_{x_1} = (m_i)_{x_2}, \text{ then } a_{i+1} = x_3$$

$$\text{otherwise } a_{i+1} = a_i + 1 \text{ and}$$

$$\bar{m}_{i+1} = \bar{m}_i, m_{i+1} = m_i.$$

(f) if $(m_i)_{a_i} = 2^p \cdot 3^{x_1} \cdot 5^{x_2} \cdot \dots \cdot p_s^{x_s} \cdot p_{s+1}^y$ for $p \leq q$, then

$$a_{i+1} = a_i + 1 \text{ and}$$

$$\bar{m}_{i+1} = \max \{ \bar{m}_i, x_1, \dots, x_s, y \} \text{ and}$$

$$(m_{i+1})_j = (m_i)_j \text{ if } j < \bar{m}_i, j \neq y$$

$$(m_{i+1})_y = h((m_i)_{x_1}, \dots, (m_i)_{x_s})$$

$$(m_{i+1})_j = 0 \text{ if } \bar{m}_i < j < \bar{m}_{i+1}$$

(g) otherwise $a_{i+1} = a_i$ and $m_{i+1} = m_i$. This is equivalent to the machine halting.

C.5 Let $I_{\pi} = \{i \mid ((\bar{\pi})_1)_j = 0 \ \forall j \ 0 < j < \bar{\pi}\}$

We now say that a computation $\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n, \dots$ satisfies the initial condition $e = \langle D, \pi, V \rangle$ at x iff

- (i) $(m_0)_{d_1} = x_1$
- (ii) $(m_0)((\bar{\pi})_1)_0 = ((\bar{\pi})_1)_1 \ \forall i \in I_{\pi}$
- (iii) $(m_0)_j = 0$ if $j \notin (I_{\pi} \cup \{d_1, \dots, d_n\})$.

C.6 We let $\sum_{i=0}^n p_i \langle \langle a_i, \langle \langle m_i, \bar{m}_i \rangle \rangle \rangle \rangle$ represent the sequence

of instantaneous descriptions $\langle a_i, \langle \bar{m}_i, m_i \rangle \rangle \ i = 1, 2, \dots, n$.

We now show that the predicate $T^S(e, X, y) \equiv e$ is the number of an initial condition and y is the number of a sequence of instantaneous descriptions for a computation $\sigma_0, \sigma_1, \dots, \sigma_n, \dots$ which satisfies the initial condition e at X for a RASP

$M_1(\Sigma_S)$ is elementary in S .

To see that $T^S(\)$ is elementary in S we first recall that if a function $f(\)$ is elementary in S , then the predicate $f(X) = y \ X \in \mathbb{N}^n$ is elementary in S . Furthermore if the predicates $P_1(\)$, $P_2(\)$ are elementary, then so are $P_1(\) \& P_2(\)$, $P_1(\) \vee P_2(\)$ and $\neg P_1(\)$ and so are $Q_1(X, Y) = \exists x \leq y P(X, x)$ and $Q_2(X, x) = \forall x \leq y P(X, y)$. Using this information we look at the definition of "sequence of instantaneous descriptions for a computation" and "sequence of instantaneous descriptions which satisfies the initial condition e at X ".

The first definition merely requires that we show $x \rightarrow y$

is an elementary predicate because if it is, then to verify that $z = p_0^{C_0} \cdot p_1^{C_1} \cdot \dots \cdot p_n^{C_n}$ is the number of a sequence of id's for a computation we need only check that $\forall i \leq z \ P(z, i)$ holds where $P(z, i) \equiv \{(z)_i \neq 0 \ \& \ (z)_{i+1} \neq 0 \implies (z)_i \rightarrow (z)_{i+1}\}$.

To see that $P(\)$ is elementary in S we merely notice that $(\)_i$ is elementary as are $-$, x and $[/]$ so that all clauses in the definition of \rightarrow are elementary in S , for example consider clause (f).

In that case, for $C_i = 2^{a_i} \cdot 3^{m_i}$ the condition for \rightarrow is $C_i \rightarrow C_{i+1}$ iff

$$(((C_i)_1)_1)_0 = p \ \&$$

$$(((C_i)_1)_1)_{n_p} = 0 \ \&$$

$$(C_{i+1}) = (C_i) + 1 \ \&$$

$$((C_{i+1})_1)_0 = \max_{j \leq n_p} \{((C_i)_1)_0, (((C_i)_1)_1)_j\} \ \&$$

$$\forall j < ((C_i)_1)_0$$

$$\{(((C_{i+1})_1)_1)_j = (((C_i)_1)_1)_j \ \& \ j \neq (((C_i)_1)_1)_{n_p} \ \&$$

$$(((C_{i+1})_1)_1)((C_i)_1)_{n_p}\} =$$

$$h_p(((C_i)_1)_1, \dots, (((C_i)_1)_1)_{n_p}) \ \&$$

$$\forall j < ((C_{i+1})_1)_0 \ \{(((C_{i+1})_1)_1)_j = 0 \ \& \ ((C_i)_1)_0 < j\}.$$

All the conjuncts are predicates elementary in $h_p(\)$, so clause (f) is elementary in $h_p(\)$.

The definition of a sequence of id's satisfying an initial condition is clearly elementary by the same reasoning.

Thus the predicate $T^S()$ is elementary in S , being a conjunction of predicates elementary in S .

C.7 We now turn to considering parts (b) and (c) of the Normal Form Theorem, namely if $\exists \pi_g$ and $h()$ such that $\pi_g(X) < h(\max X)$ $X \in N^n$, then there is a function $b() \in \mathcal{E}h()$ such that

$$\exists e \ g(X) = U(\mu y \leq b(X) \ T(e, X, y)).$$

Let e be the number of the initial condition $\langle D, \pi, V \rangle$. Since $f_0()$ is normal for Σ_0 , the largest new number that can be introduced into a computation satisfying the initial condition e is $f_0(d)$ where $d = \max\{\max X, \text{program constants}\}$. Surely $d < e$, thus we can use $f_0(e)$. After s computing steps, the largest possible value of the number z representing the sequence of id's for s steps satisfies $z < 2 \cdot f_0^{(s)}(e) \cdot 3^{d(s,e)} = b(s,e)$ where $d(s,e) = P_u^u$ where $u = f_0^{(s)}(e)$. We first observe that $b(x,e)$ is elementary. This follows since $f_0^{(y)}(x)$ can be defined by limited recursion from $W_1()$ as we observed in Chapter 7 and since the operations of multiplication, exponentiation, and $p_n = n$ -th prime are elementary (see Kleene p. 230 & p. 285). Thus $b(h(\max X), e)$ is elementary in $h()$. Take $b() = b(h(), e)$.

For part (c) notice that according to the Actual Time Theorem, if $g() \in E_\alpha$, then $\exists d$ such that $\pi_g(X) < f_\alpha^{(d)}(\max X)$ and observe that for fixed d $f_\alpha^{(d)}(\max X)$ is elementary in $f_\alpha()$.

q.e.d.

BIBLIOGRAPHY

- [1] Axt, Paul. On a subrecursive hierarchy and primitive recursive degrees, Trans. A.M.S., 92, 1959, pp.85-105.
- [2] Axt, Paul. Enumeration and the Grzegorzcyk hierarchy, Zeitschr. f. math. Logik und Grund. d. Math., 9, 1963, pp.53-65.
- [3] Blum, Manuel. Machine-independent theory of the complexity of recursive functions, J.A.C.M., 14, 1967, pp.322-336.
- [4] Cannonito, Frank B. Hierarchies of computable groups and the word problem, J.S.L., 31, 1966, pp.376-392.
- [5] Chaitin, Gregory J. On the length of programs for computing finite binary sequences. J.A.C.M., 14, 1966, pp.547-569.
- [6] Cleave, John P. A hierarchy of primitive recursive functions, Zeitschr. f. math. Logik und Grund. d. Math., 9, 1963, pp.331-345.
- [7] Cobham, Alan. The intrinsic computational difficulty of functions, Logic, Methodology and Philosophy of Science, Amsterdam, 1965, pp.24-30.
- [8] Cook, Stephen A. On the minimum computation time of functions, Ph.D. Diss., Harvard, 1966.
- [9] Davis, Martin. Computability and Undecidability, New York, 1958.
- [10] Engler, Erwin, Algorithmic properties of structures, Math. Syst. Theory, 1, 1967, pp.183-195.

- [11] Fabian, Robert J. Hierarchies of general recursive functions and ordinal recursion, Ph.D. Diss., Case Inst. of Tech., 1965.
- [12] Feferman, Solomon. Classification of recursive functions by means of hierarchies, Trans. A.M.S., 1962, pp.101-122.
- [13] Feferman, S., and Spector, C. Incompleteness along paths in progressions of theories, J.S.L., 1962, pp.383-390.
- [14] Golomb, Solomon W. Shift Register Sequences, San Fransisco, 1967.
- [15] Grzegorzczk, A. Some classes of recursive functions, Rozprawy Matematyczne, 1953, pp.1-45.
- [16] Guard, James R. The independence of transfinite induction up to ω^ω in recursive arithmetic, Ph.D. Diss., Princeton, 1961.
- [17] Hartmanis, J, and Stearns, R. On the computational complexity of algorithms, Trans. A.M.S., 117, 1965, pp.285-306.
- [18] Hartmanis, J, Stearns, R, and Lewis, P. Classification of functions by time and memory requirements, Proc. IFIP Int. Cong.,1, 1965, pp.31-36.
- [19] Hartmanis, J, and Stearns, R. Algebraic Structure Theory of Sequential Machines, Englewood Cliffs, 1966.
- [20] Hartmanis, Juris. Tape reversal bounded Turing machine computations, Cornell Tech. Report No. 68-7, 1968.

- [21] van Heijenoort, J. From Frege to Godel, Cambridge, 1967.
- [22] Hermes, H. Enumerability, Decidability, Computability, New York, 1965.
- [23] Heyting, A. Infinitistic methods from a finitist point of view, Infinitistic Methods, New York, 1961, pp.185-92
- [24] Kleene, Stephen C. Introduction to Metamathematics, New York, 1952.
- [25] Kleene, S.C. Extension of an effectively generated class of functions by enumeration, *Colloq. Math.*, 6, 1958, pp. 67-78.
- [26] Kleene, S.C. On the forms of the predicates in the theory of constructive ordinals (second paper), *Amer. J.Math.*, 77, 1955, pp. 405-428.
- [27] Kreider, D.L, and Ritchie, R.W. A basis theorem for a class of two-way automata, *Zeitchr. f. math. Logik und Grund. d. Math.*, 12, 1966, pp. 243-255.
- [28] Kreider, D.L, and Ritchie, R.W. Predictably computable functionals and definition by recursion, *Zeitchr. f. math Logik und Grund. d. Math.*, 5, 1963, pp. 65-80.
- [29] Kreisel, Georg. Mathematical logic, Lectures on Modern Mathematics, vol. III, New York, 1965. pp.95-195.
- [30] Krohn, K, Mateosian, R, and Rhodes, J. Complexity of ideals in finite semigroups and finite state machines, *Math. Syst. Theory*, 1, 1967, pp. 59-66.

- [31] Lachlan, S.H. Multiple recursion, Zeitschr. f. math Logik und Grund. d. Math., 8, 1962, pp. 81-107.
- [32] McCarthy, John. A basis for a mathematical theory of computation, Computer Programming and Formal Systems, Amsterdam, 1963, pp. 33-70.
- [33] McCleary, Stephen. Primitive recursive computations, submitted for publication to Notre Dame J. Logic.
- [34] Meyer, A.R, and Ritchie, D.M. Computational complexity and program structure, IBM Research, RC-1817, 1967.
- [35] Moore Edward F.(editor) Sequential Machines, Reading, 1964.
- [36] Moschovakis, Yiannis N. A remark on subrecursive hierarchies, personal communication.
- [37] Myhill, John. A stumbling block in constructive mathematics (abstract), J.S.L., 18, 1953, p. 190.
- [38] Myhill, John. Linear bounded automata, WADD Tech. note, 60-165, Wright-Patterson AFB, 1960.
- [39] Péter, Roza. Recursive Functions, 3d ed., New York, 1967.
- [40] Platek, Richard A. Foundations of recursion theory, Ph.D. Diss., Stanford, 1966.
- [41] Rabin, Michael O. Real time computation, Israel J. Math., 1, 1963, pp. 203-211.
- [42] Ritchie, Robert W. Classes of predictably computable functions, Trans. A.M.S., 106, 1963, pp. 139-173.

- [43] Ritchie, Robert W. Classes of recursive functions based on Ackermann's function, *Pacific J. Math.*, 15, 1965, pp. 1027-1044.
- [44] Robbin, Joel W. Subrecursive hierarchies, Ph.D. Diss., Princeton, 1965.
- [45] Robinson, A, and Elgot C.C. Random-access stored program machines, an approach to programming languages, *J.A.C.M.*, 11, 1964, pp. 365-399.
- [46] Rogers, Hartley Jr. Theory of Recursive Functions and Effective Computability, New York, 1967.
- [47] Routledge, N.A. Ordinal recursion, *Proc. Cambridge Phil. Soc.*, 49, 1953, pp. 175-182.
- [48] Schütte, Kurt. Predicative well-orderings, Formal Systems and Recursive Functions, Amsterdam, 1965, pp. 280-303.
- [49] Scott, Dana. Some definitional suggestions for automata theory, *J. Comptr.& Syst. Sci.*, 1, 1967, pp. 187-212.
- [50] Shepherdson, J.C, and Sturgis, H.E. Computability of recursive functions, *J.A.C.M.*, 10, 1963, pp.217-255.
- [51] Shoenfield, Joseph R. The class of recursive functions, *Proc. A.M.S.*, 11, 1958, pp. 61-62.
- [52] Shoenfield, Joseph R. Mathematical Logic. Reading, 1967.
- [53] Sierpinski, Waclaw. Cardinal and Ordinal Numbers. Warsaw, 1965.

- [54] Smullyan, Raymond M. Theory of Formal Systems.
Princeton, 1961.
- [55] Spector, Clifford. Recursive well-orderings, J.S.L.,
20, 1955, pp. 151-163.
- [56] Tait, W.W. Nested recursion, Math. Ann., 143,
1961, pp. 236-250.
- [57] Takeuti, Gaisi. Ordinal diagrams I, J. Math. Soc.
Japan, 9, 1957, pp. 386-394.
- [58] Takeuti, Gaisi. Ordinal diagrams II, J. Math. Soc.
Japan, 12, 1960, pp. 385-391.