COMPUTATIONAL COMPLEXITY

ON

REGISTER MACHINES

by

Richard Simpson Bird

Thesis submitted for the degree of
Doctor of Philosophy of the
University of London

October 1973

Institute of Computer Science
Gordon Square, London WC1

ABSTRACT

This thesis investigates the computational complexity of programs based on their running time. A program consists of an arbitrary flowchart defined over some instruction set of assignments and test instructions, and computes a unique function. Each instruction set specifies an order code for a particular register machine, whose registers A_1, A_2, ... can contain arbitrary integers. Each instruction set includes the basic set I_0:

assignments	$A_j:=A_k+1$	$A_j:=A_k$
	$A_j:=A_k-1$	$A_j:=0$
tests	$A_j=0$	$A_j\geq 0$.

It is shown that programs defined over just I_0 can be speeded up by an arbitrary linear factor. A formal demonstration of this result uses a new technique for verifying the correctness of equivalence preserving transformations. It is shown that this speed up property fails to hold if I_0 is augmented with the assignments

$$A_i:=A_j+A_k \qquad A_i:=A_j-A_k.$$

It is shown that the problem of determining whether or not a given function can be computed within a certain time bound can be reduced to the problem of showing whether or not a different function can be computed in real time. Finally, criteria are established for showing that, under certain conditions, a function is not real time computable.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor and colleague, Michael Bell, for his help, advice and encouragement throughout the preparation of this thesis. My thanks are also due to Professor R.W. Hockney of the Computer Science Department, at the University of Reading, for allowing me time from academic duties to finish this work. Finally, my gratitude to Miss Margaret Moir for her patience in typing the manuscript, and my gratitude to my wife for everything.

CONTENTS

Page

CHAPTER 0: INTRODUCTION AND SUMMARY 6

CHAPTER 1: PROGRAMS, MACHINES, AND COMPILERS 14

1.1 Programs 14

1.2 Machines 15

1.3 Translations and compilers 18

1.4 Composition of compilers 27

1.5 Modified programs 29

CHAPTER 2: BASIC DEFINITIONS AND RESULTS 30

2.1 Register machines and their programs 30

2.2 Functions computable within zero time 40

2.3 Asymptotic notation 43

2.4 Three basic theorems 44

2.5 On-line programs 47

2.6 The composition and addition of programs 50

2.7 The inversion of programs 55

CHAPTER 3: THE SPEED UP THEOREM 60

3.1 Finite sequences of assignments 62

3.2 Halving the number of assignments 74

3.3 Conversion to an I_0-program 77

3.4 Concluding the proof 79

3.5 A worked example 82

3.6 Further results and discussion 88

CHAPTER 4: THE HONEST FUNCTIONS 93

4.1 Honesty and linear speed up 94

4.2 The real-time characterisation 97

4.3 Honesty and running time 106

4.4 Closure under composition 116

4.5 Subtraction of honest functions 120

CHAPTER 5: NECESSARY CONDITIONS FOR ON-LINE COMPUTATION 125

5.1 The equivalence relation $\equiv_n$ 126

5.2 The equivalence relation $\sim_n$ 135

5.3 The basic conditions 138

5.4 Applications 143

5.5 Instruction sets not possessing speed up 147

5.6 Further results 151

5.7 Structural characterisation of real time functions 156

APPENDIX A: MATHEMATICAL BACKGROUND FOR CHAPTER 1 161

APPENDIX B: LIST OF SPECIAL SYMBOLS 165

REFERENCES 167

CHAPTER ZERO

INTRODUCTION AND SUMMARY

The present work is a contribution to model-theoretic computational complexity. A model for the computational process is defined (in our case, register machine programs) and studied with respect to some natural measure of complexity (in our case, the running time). Such studies have intrinsic mathematical interest, but are also important in two practical respects: as an aid to the investigation of particular problems, and as a basis for the exploration of optimisation procedures. We briefly discuss these areas.

In the investigation of the complexity of particular problems, two distinct types of analysis are required; Knuth [25] calls them <u>local</u> (or type A) analysis, and <u>global</u> (or type B) analysis. In the local analysis, an algorithm for the solution of a problem is given, and its performance is analysed under various input assumptions, e.g. worst case analysis. The criteria by which the performance is judged may be storage requirements or, more usually, some measure of the running time based on certain natural operational units such as the number of additions and multiplications in arithmetic algorithms, or the number of comparisons in sorting algorithms. Such an analysis serves to put an upper bound to the complexity of the problem. Among many examples of this type of analysis, one may mention: Strassen's algorithm for matrix multiplication (Strassen [37]), the Euclidean algorithm (Knuth [27], Collins [11]), algorithms for the recognition

of context free languages (e.g. Younger [39]), and more recently, algorithms for the selection problem (Blum et al [8]).

On the other hand, global analysis attempts to put a lower bound on the complexity of the given problem. Such an analysis depends on the means admissible for a solution, and hence on some firmly established computational model to which the complexity of the problem can be related. Since this type of analysis is often sensitive to small changes in the model's technology, it is important that the model be both natural and flexible. For example, the complexity of polynomial evaluation or matrix multiplication is naturally given in terms of a program model in which assignments involving multiplication and addition can appear, but a different instruction set would be proper in the study of how long it takes to multiply two numbers (e.g. Cook and Anderaa [13]). A computational model based on programs over a variety of instruction sets appears to be both natural and flexible. By studying it, one hopes to develop general techniques for estimating lower complexity bounds. (Chapter 5 contains an example of such a technique). In contrast, it is felt that for many problems, computational models based on Turing machines fail to meet the criteria mentioned above. The language of Turing machines is not a particularly natural model of real programming languages, and the basic mode of operation is not easily augmented to correspond to real operational units. Furthermore, some of the Turing machine complexity measures, such as the number of tape reversals (Hartmanis [20]) appear difficult to relate to actual programming situations. Of course, most models have features in common, and in the axiomatic development of computational complexity (e.g. Blum [7]

McCreight [30]), it is precisely these features that are brought to light.

The second use of model-based complexity studies is in the investigation of efficiency increasing program transformations, which McCarthy [29] lists as one of the goals of a mathematical theory of computation. Here, the object is to describe translations which transform programs into other equivalent programs, but at the same time increase some measure of the program's performance, possibly at the expense of another (e.g. Allen [2], Marill [28], Aho and Ullman [1], Hopcroft and Ullman [23]). It is an important problem to discover the nature of these trade-off relationships. (Chapter 3 contains an example of such a transformation).

The present work contains an investigation of the computational complexity of programs based on their running time. Each program consists of an arbitrary flowchart defined over some specified set of assignments and test instructions and computes a unique (partial number-theoretic) function. The instruction set is regarded as an order code for a particular machine. Thus the basic framework is the very natural one suggested by Scott [35], in which programs and machines are treated as distinct but closely related concepts, and the notion of computable function, rather than decidable set or enumerable sequence is given prime importance. Each machine possesses the same basic hardware consisting of a denumerable number of registers each of which can contain an arbitrary integer, and is therefore called a register machine.

The first comprehensive study of register machines was given by Shepherdson and Sturgis [36], and concentrated mainly on a particular

machine (the URM) which had an order code containing instructions for incrementing and decrementing a register, copying one register into another, initialising a register to zero, and testing for zero. (A precise description of this instruction set is given in Chapter 2). They showed that such programs could compute just the class of partial recursive functions. Minsky [33] showed that with suitable coding of the input and output, only two registers were in fact needed. Cleave [9] studied time-limited complexity classes of register machine programs, based on a somewhat less natural definition of running time than the one we use, relating them to the Grzegorzyk [18] hierarchy of classes of recursive functions. Elgot and Robinson [14] introduced the idea of a stored program register machine, and the complexity of this model has recently been investigated by Hartmanis [19]. Fischer, Meyer and Rosenberg [16], [17], studied an automata-theoretic version of register machines, concentrating on language recognition and sequence generation. Some of the results in the present work, parallel and extend the work in these last two papers. Cobham [10] also studied sequence generation based on a register machine model.

Meyer and Ritchie [31] studied register machine programs based on a more restrictive control mechanism than that given by an arbitrary flowchart (effectively, the Fortran DO-loop), and this work was extended by Constable and Borodin [12], who explored the relative efficiency of flowchart languages and DO-loop languages.

The computational model which has been studied the most is the Turing machine model. This study, which was initiated by Yamada [40] and Hartmanis and Stearns [21] now has a large literature (see the

bibliography of Ireland and Fischer [24]).

We now summarize the main results presented in the thesis.

Chapter 1 is of a rather different nature than the other chapters, and consists of a modified version of Bird [4] on the subject of compilers.

A number of theorems in the present work are proved in the same manner; a translation is described which converts an arbitrary program or programs into another program with certain desirable properties. Usually, although not always, the translated program is equivalent to the original one, i.e. it computes the same function. In such a case, we call the translation a <u>compiler</u>. One major difficulty arises about the description of compilers: since we are proving theorems and not just writing software, it must somehow be proved that the compilers are correct, i.e. do produce equivalent object programs. In some simple cases, this is obvious from an informal description of the translation, and a formal demonstration of correctness would be unnecessarily pedantic. In other cases, the compilers may be quite complicated, and it is necessary to make use of some formalism for describing translations, and appeal to some general verification technique. This problem arises especially in Chapter 3, where compilers are defined which make heavy use of 'label logic' to control the flow of computation.

Accordingly, Chapter 1 is devoted to establishing just such a formalism and verification technique. The main result, the compiler theorem, gives five conditions the conjunction of which is sufficient

to verify that a given translation is indeed a compiler. The mathematical background necessary to understand the details of the proof is given separately in Appendix A. The compiler theorem can be invoked to formally prove the correctness of all compilers described in the present work.

Chapter 2 introduces, in more detail, the models to be studied. Some straightforward consequences of the definitions are proved, and three useful program transformations (addition, composition, and inversion) are described.

In Chapter 3, we prove a speed up theorem for programs defined over the original instruction set described by Shepherdson and Sturgis (i.e. URM programs). This theorem says that it is always possible to translate a given program into another equivalent one which runs twice as fast. By applying the translation a number of times, it is therefore possible to speed up the running time of any program by an arbitrary linear factor. Speed up theorems of this sort are by no means new, but the author feels that their importance has been underestimated in the past. This is partly because the linear speed up theorem was first proved for Turing machines, and these machines only possess speed up of a very trivial sort, essentially reducing to: given a program P_1 for a Turing machine T_1 (which uses an alphabet of k symbols), an equivalent program P_2 for another Turing machine T_2 (which uses $2k$ symbols) can be found. In other words, the underlying hardware can be changed. More interesting is the question of whether an equivalent program P_2 for the same machine T_1 can be found; but for Turing machines this is not possible.

The proof of speed up for the Shepherdson and Sturgis instruction set is not trivial. Fischer, Meyer and Rosenberg [16] prove, in a different setting, a speed up theorem for a subset of these instructions. It is an interesting problem to explore the nature of the boundary between instruction sets that possess speed up, and those that do not. As a consequence of a general theorem in Chapter 5, it is proved that if the SS instruction set is augmented with instructions for adding and subtracting the contents of two registers, then the resulting set does not possess speed up. It is important to mention that the new set possesses exactly the same input and output instructions as the original; were this not the case, the proof that speed up is impossible would be trivial. Using the same technique, it is possible to show that if the SS set is given the ability to address registers indirectly through index registers, then the resulting instruction set again does not possess speed up. This last result can also be proved by a straightforward diagonalisation argument, but such arguments are not used in this work.

The main result of Chapter 4 shows that to a large extent the study of time-limited computations of register machine programs can be reduced to the study of real time computations, i.e. where the running time is bounded by a linear function of the input. More precisely, we show that, under certain restrictions on t, the following two statements are equivalent for an arbitrary function f:

(i) f is computable within time t,

(ii) $\lambda x.f(\max y\ [x \geq t(y)])$ is real time computable.

The restriction on t is that it should be superhonest, and Chapter 4

also contains a discussion of this class of functions.

In Chapter 5, we give a general technique for establishing lower bounds on the time for the computation of given functions. The results, which extend those of [17], can be used to show that there is a slowly increasing function which cannot be computed, no matter what instruction set is specified, in under εx steps for each input x, where ε is some fixed positive real number. This enables us to prove that certain instruction sets cannot possess the speed up property.

CHAPTER ONE

PROGRAMS, MACHINES, AND COMPILERS

This chapter presents, under a fairly general definition of program and machine (similar in spirit to Scott [35]), a technique for defining compilers and proving them correct. Here, a compiler simply means a translation between programs which preserves equivalence in the sense that corresponding source and target programs compute the same function. The compiler theorem (Theorem 1.1), which is used extensively in subsequent chapters, gives certain conditions, the verification of which is sufficient to demonstrate that a specified translation does indeed preserve equivalence. Similar but somewhat less general conditions are given in Milner [32] and Knuth [26, Ex.1.1.9]. The proof of the theorem depends heavily on a certain induction principle for establishing equality between recursively defined partial functions and the necessary mathematical background and notation is given, for convenience, separately in Appendix A.

§1. Programs

The basic components of programs are labelled instructions. The space I of labelled instructions is defined with respect to three sets of identifiers:

L a set of label identifiers,

F a set of function identifiers,

T a set of test identifiers.

Each instruction $i \in I$ has one of the forms

$$\ell: f \rightarrow \ell'$$

or $$\ell: t \rightarrow \ell', \ell'',$$

where $f \in F$, $t \in T$, and $\ell, \ell', \ell'' \in L$. The label which appears to the left of the colon sign in an instruction i is called the <u>label of</u> i, and is denoted by $\lambda(i)$. The precise meaning of an instruction depends on specifying a machine, as we shall see, but the above forms are intended to suggest the Algol - like equivalents:

ℓ: <u>do</u> f <u>then</u> <u>goto</u> ℓ',

and ℓ: <u>if</u> t <u>then</u> <u>goto</u> ℓ' <u>else</u> <u>goto</u> ℓ''.

A <u>program</u> P is any finite subset of I for which the following condition holds:

for all $i, j \in P$, if $\lambda(i) = \lambda(j)$ then $i = j$.

Thus a program is a finite set of instructions each of which possesses a different label.

The collection of programs over I is denoted by $P(I)$, and the set $\{\lambda(i): i \in P\}$ by $\lambda(P)$. We shall say that ℓ is a <u>terminal label</u> of P if ℓ is referenced by P (i.e. is contained in the right hand part of some instruction of P), but ℓ is not in $\lambda(P)$.

§2. Machines

A machine $M = M(L, F, T)$ is defined when the following objects are given:

1. an input set X,
2. an output set Y,
3. a memory set V,
4. an input function $i_M: X \to V$,
5. an output function $O_M: V \to Y$,
6. for each $f \in F$, a function $f_M: V \to V$,
7. for each $t \in T$, a predicate $t_M: V \to \{\text{true, false}\}$,
8. a particular element <u>start</u> of L, called the initial or start label.

The <u>transition</u> <u>function</u> μ of M is a function

$$\mu: I \to [(L\times V) \to (L\times V)]$$

with the definition

$$\begin{aligned}
\mu(i)(\ell, v) &= (\ell', f_M(v)) && \text{if } i = \ell: f \to \ell', \\
&= (\ell', v) && \text{if } i = \ell: t \to \ell', \ell'' \\
& \quad \text{and } t_M(v) = \text{true}, \\
&= (\ell'', v) && \text{if } i = \ell: t \to \ell', \ell'' \\
& \quad \text{and } t_M(v) = \text{false}, \\
&= \text{undefined, otherwise.}
\end{aligned}$$

Note this definition implies that the execution of a test does not change the current value of the memory set.

It is convenient in the sequel to modify the input and output functions, by defining

1. <u>in</u>: $X \to (L\times V)$ by <u>in</u> $(x) = (\underline{\text{start}}, i_M(x))$,

and 2. <u>out</u>: $(L\times V) \to Y$ by <u>out</u> $(\ell, v) = O_M(v)$.

Suppose that P is a given program, and M is a given machine. In order to describe the function fP computed by P on M, we first define two useful functions:

(a) θ: $P(I) \to [(L \times V) \to \{true, false\}]$

by $\theta(P)(\ell, v) = true$ if $\ell \in \lambda(P)$,

$= false$ otherwise,

and (b) ϕ: $P(I) \to [(L \times V) \to (L \times V)]$, recursively by the equation

$$\phi(P) = (\theta(P) \to \phi(P) \cdot \bigsqcup_{i \in P} \mu(i), 1).$$

(For the meaning of $\bigsqcup$ see Appendix A). Here, 1 denotes the identity function on $L \times V$. The definition of μ guarantees that $\{\mu(i): i \in P\}$ is a set of disjoint functions provided P is a well formed program, and so the upper bound is defined.

The function $fP: X \to Y$ is now given by the equation

$$fP = \underline{out} \cdot \phi(P) \cdot \underline{in}.$$

Thus $fP(x)$ expresses the result of executing P on M with initial value $i_M(x)$ of the memory set, beginning with the instruction labelled <u>start</u>, and proceeding instruction by instruction, as determined by the transition function, until an element (ℓ, v) of $L \times V$ is reached, if it ever is, where no instruction of P has the label ℓ, in which case $fP(x) = O_M(v)$.

Having established these preliminaries, we now proceed to the main definitions and theorem.

§3. Translations and compilers

A translation is a procedure for transforming programs into other programs. The translation may be defined by a step by step process in which individual instructions of a source program are translated into subprograms of the target program, or possibly by a more general process in which subprograms of the source program are translated as a whole. In general, corresponding source and target programs need not be intended for the same machine, but for the purposes of this thesis we can suppose that they are.

In order to ensure that the set of instructions produced by a translation forms a program, the following definitions are adopted.

1. A translation is a mapping $\Sigma:\Pi \to P(I)$, where Π is a partition of I consisting of programs (i.e. a collection of disjoint subsets of I, each of which is a program, and whose union is I), for which the following condition holds:

   for all $S_1, S_2 \in \Pi$, if $\lambda(S_1)$ and $\lambda(S_2)$ are disjoint sets, then so are $\lambda(\Sigma S_1)$ and $\lambda(\Sigma S_2)$.

2. A translation $\Sigma:\Pi \to P(I)$ is applicable to a program P if P is the union of some of the sets in Π. In such a case, the collection $\{S: S \subseteq P, S \in \Pi\}$ partitions P and is denoted by Π_P. It follows from the first definition that if Σ is applicable to P, then the set

$$\Sigma(P) = \bigcup_{S\in\Pi_P} \Sigma(S)$$

   is a well formed program.

3. A translation $\Sigma:\Pi \to P(I)$ is a <u>step by step</u> translation if Π is the identity partition $\{(i): i\varepsilon I\}$. Clearly, step by step translations are applicable to any program over I.

4. A translation $\Sigma:\Pi \to P(I)$ is called a <u>compiler</u> (for a machine M) if the equation

$$f(P) = f(\Sigma P)$$

holds for each program P to which Σ is applicable.

The compiler theorem can now be stated.

<u>Theorem</u> 1.1 A sufficient condition for a translation $\Sigma:\Pi \to P(I)$ to be a compiler (for M) is that there exists a (possibly partial) function $\sigma: L\times V \to L\times V$ for which the following conditions hold:

(1) $\sigma \cdot \underline{in} = \underline{in}$,

(2) $\phi(\Sigma S)$ maps D into D for all $S\varepsilon\Pi$, where D = domain (σ),

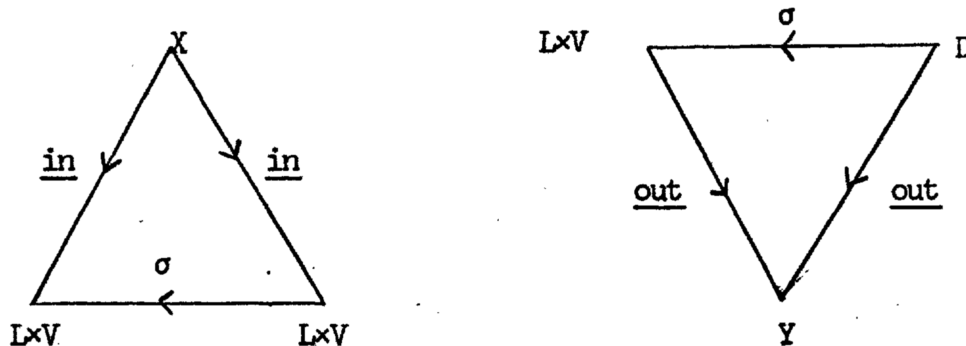
(3) $\underline{out} \cdot \sigma = \underline{out}$ on D,

(4) $\theta(S) \cdot \sigma = \theta(\Sigma S)$ on D, for all $S\varepsilon\Pi$,

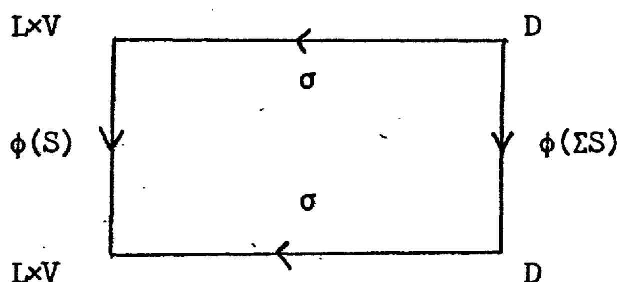
(5) $\phi(S) \cdot \sigma = \sigma \cdot \phi(\Sigma S)$ on D, for all $S\varepsilon\Pi$.

(An equation $f = g$ holds on D if for all $x\varepsilon D$, $f(x) = g(x)$).

<u>Remark</u>. The conditions are best explained by pictures. Conditions (1) and (3) say that the following diagrams commute:



and conditions (2) and (5) entail the commutativity of



The proof of the theorem depends on three main lemmas. We suppose that an arbitrary program P is given, to which Σ is applicable, and that Q denotes the translated program $\Sigma P = \bigcup_{S \in \Pi_P} \Sigma S$. As an aid to readability, we shall sometimes write functional application without parentheses, i.e. ϕP for $\phi(P)$ etc.

<u>Lemma</u> 1.2 Let ψP and ψQ be recursively defined by the equations

$$\psi P = (\theta P \to \psi P \cdot \bigsqcup_{S \in \Pi_P} \mu(S), 1),$$

$$\psi Q = (\theta Q \to \psi Q \cdot \bigsqcup_{S \in \Pi_P} \mu(\Sigma S), 1),$$

where for any program R

$$\mu(R) = (\theta R \to \phi R, \Omega).$$

Then

$$\psi P \cdot \sigma = \sigma \cdot \psi Q \quad \text{on} \quad D.$$

(For the meaning of Ω see Appendix A).

<u>Lemma</u> 1.3 With the above definitions, ψQ maps D into D.

The last lemma is independent of the concept of a translation.

<u>Lemma</u> 1.4 Let R be any program, and Π_R any partition of R. If ψR is recursively defined by the equation

$$\psi R = (\theta R \to \psi R \cdot \bigsqcup_{S \in \Pi_R} \mu(S), 1),$$

then $\psi R = \phi R$.

Assuming the truth of these lemmas, Theorem 1.1 follows easily. We have

$fP = \underline{out} \cdot \phi P \cdot \underline{in}$	by definition,
$= \underline{out} \cdot \psi P \cdot \underline{in}$	by Lemma 1.4,
$= \underline{out} \cdot \psi P \cdot \sigma \cdot \underline{in}$	by condition (1) on Σ,
$= \underline{out} \cdot \sigma \cdot \psi Q \cdot \underline{in}$	by Lemma 1.2,
$= \underline{out} \cdot \psi Q \cdot \underline{in}$	by condition (3) and Lemma 1.3.

Now, $\psi Q = (\theta Q \to \psi Q \cdot \bigsqcup_{S \in \Pi_P} \mu(\Sigma S), 1)$

$$= (\theta Q \to \psi Q \cdot \bigsqcup_{S \in \Pi_Q} \mu(S), 1),$$

where Π_Q is the partition $\{\Sigma S : S \in \Pi_P\}$ of Q. Hence Lemma 1.4 can be applied again, and we have

$$fP = \underline{out} \cdot \phi Q \cdot \underline{in} = fQ.$$

Proof of Lemma 1.2

First, it follows from condition (4) on Σ that

$$\begin{aligned} \theta P \cdot \sigma &= (\exists S \in \Pi_P) \theta S \cdot \sigma \\ &= (\exists S \in \Pi_P)\ \theta(\Sigma S) \text{ on } D \\ &= \theta Q \text{ on } D. \end{aligned}$$

Second, it follows from condition (5) on Σ that

$$\begin{aligned} \mu(S) \cdot \sigma &= (\theta S \cdot \sigma \to \phi S \cdot \sigma, \Omega) \\ &= (\theta(\Sigma S) \to \sigma \cdot \phi(\Sigma S), \Omega) \text{ on } D \\ &= \sigma \cdot \mu(\Sigma S) \text{ on } D. \end{aligned}$$

Having established, these facts, we can prove by induction that

$$\psi_k P \cdot \sigma = \sigma \cdot \psi_k Q \text{ on } D \text{ for all } k \geq 0.$$

The assertion is clearly true for $k = 0$, since both $\psi_0 P$ and $\psi_0 Q$ are Ω, and the induction step is

$$\psi_{k+1} P \cdot \sigma = (\theta P \cdot \sigma \to \psi_k P \cdot \bigsqcup \mu(S) \cdot \sigma, \sigma),$$

where the upper bound is taken over all $S \in \Pi_P$,

$$= (\theta Q \to \psi_k P \cdot \sigma \cdot \bigsqcup \mu(\Sigma S), \sigma)$$

$$= (\theta Q \to \sigma \cdot \psi_k Q \cdot \bigsqcup \mu(\Sigma S), \sigma),$$

using the induction hypothesis. Since this last expression is just $\sigma \cdot \psi_{k+1} Q$, the induction step is established.

Proof of Lemma 1.3

Using condition (2) on Σ, the assertion

$$\psi_k Q \text{ maps } D \text{ into } D \text{ for all } k \geq 0,$$

can be proved by a similar induction argument. Since, for each $d \in D$ there exists a k such that

$$\psi Q(d) \neq \Omega \text{ implies } \psi Q(d) = \psi_k Q(d),$$

the lemma is proved.

The proof of Lemma 1.4 is more complicated and depends on four further lemmas. For shorthand, we denote

$$\bigsqcup_{i \in R} \mu(i) \text{ by } A \text{ and } \bigsqcup_{S \in \Pi_R} \mu(S) \text{ by } B.$$

Lemma 1.5 If $i \in R$, then $\phi R \cdot \mu(i) \sqsubseteq \phi R$.

Proof. $\phi R \cdot \mu(i) = (\theta R \to \phi R \cdot \mu(i), \Omega)$ since $i \varepsilon R$,

$$= (\theta R \to \phi R \cdot A, \Omega)$$

$$\sqsubseteq \phi R \quad \text{by definition of } \phi R.$$

Lemma 1.6 If $S \subseteq R$, then $\phi R \cdot \mu S \sqsubseteq \phi R$.

Proof. We prove by induction that

$$\phi R \cdot \phi_k S \sqsubseteq \phi R \quad \text{for all} \quad k \geq 0,$$

from which it follows that $\phi R \cdot \phi S \sqsubseteq \phi R$.

Since $\mu S \sqsubseteq \phi S$, the conclusion follows.

The assertion is clearly true for $k = 0$, and the induction step is

$$\phi R \cdot \phi_{k+1} S = (\theta S \to \phi R \cdot \phi_k S \cdot \bigsqcup_{i \varepsilon S} \mu(i), \phi R)$$

$$\sqsubseteq (\theta S \to \phi R \cdot \bigsqcup_{i \varepsilon S} \mu(i), \phi R) \quad \text{by induction,}$$

$$= (\theta S \to \bigsqcup_{i \varepsilon S} \phi R \cdot \mu(i), \phi R)$$

$$\sqsubseteq (\theta S \to \phi R, \phi R) \quad \text{by lemma 1.5 and the fact}$$

that $S \subseteq R$. Since

$$(\theta S \to \phi R, \phi R) \sqsubseteq \phi R,$$

the induction is complete.

<u>Lemma</u> 1.7 If $S \in \Pi_R$, then $\psi R \cdot \phi S = \psi R$.

<u>Proof</u>. $\psi R \cdot \phi S = \psi R \cdot (\theta S \rightarrow \mu S, 1)$ by definition of μS,

$$= (\theta S \rightarrow \psi R \cdot \mu S, \psi R)$$

$$= (\theta S \rightarrow \psi R \cdot B, \psi R),$$

since if Π_R is a partition of R, then the set $\{\mu(S): S \in \Pi_R\}$ is a set of disjoint functions. The last expression is equal to

$(\theta S \rightarrow (\theta R \rightarrow \psi R \cdot B, 1), \psi R)$ since θS implies θR,

$$= (\theta S \rightarrow \psi R, \psi R)$$

$$= \psi R.$$

<u>Lemma</u> 1.8 If $i \in R$, then $\psi R \cdot \mu(i) \subseteq \psi R$.

<u>Proof</u>. Let S be the unique member of Π_R which contains i. Then

$\psi R \cdot \mu(i) = \psi R \cdot \phi S \cdot \mu(i)$ by Lemma 1.7,

$\subseteq \psi R \cdot \phi S$ by Lemma 1.5 since $i \in S$,

$= \psi R$ by Lemma 1.7 again.

<u>Proof of Lemma 1.4</u>

(a) $\psi R \subseteq \phi R$. We prove by induction that

$$\psi_k R \subseteq \phi R \quad \text{for all} \quad k \geq 0.$$

The assertion is clearly true for $k = 0$, and the induction step is

$$\psi_{k+1} R = (\theta R \to \psi_k R \cdot B, 1)$$

$$\sqsubseteq (\theta R \to \phi R \cdot B, 1) \text{ by induction hypothesis,}$$

$$= (\theta R \to \bigsqcup_{S \in \Pi_R} \phi R \cdot \mu S, 1)$$

$$\sqsubseteq (\theta R \to \phi R, 1) \text{ by Lemma 1.6,}$$

$$= \phi R.$$

(b) $\phi R \sqsubseteq \psi R$. Here, the induction step is

$$\phi_{k+1} R = (\theta R \to \phi_k R \cdot A, 1)$$

$$\sqsubseteq (\theta R \to \psi R \cdot A, 1) \text{ by induction hypothesis,}$$

$$= (\theta R \to \bigsqcup_{i \in R} \psi R \cdot \mu(i), 1)$$

$$\sqsubseteq (\theta R \to \psi R, 1) \text{ by Lemma 1.8,}$$

$$= \psi R.$$

The proof of Theorem 1.1 is finally complete. It can be shown by considering suitable counter examples that the theorem fails to hold if any of the five conditions is omitted.

In the next section, we consider the composition of translations.

§4. Composition of compilers

We say that a translation Σ is a compiler under σ, if Σ together with σ satisfies the five conditions of Theorem 1.1. The following result is of interest, but is not used in the sequel.

Theorem 1.9 Let $\Sigma: \Pi_1 \to P(I)$ and $\Delta: \Pi_2 \to P(I)$ be two translations with the property that Δ is applicable to $\Sigma(S)$ for each $S \varepsilon \Pi_1$. Suppose that Σ is a compiler under σ, and Δ is a compiler under δ. Then the composition translation $\Delta \cdot \Sigma: \Pi_1 \to P(I)$, defined by

$$\Delta \cdot \Sigma(S) = \bigcup_{T \varepsilon \Pi_S} \Delta(T)$$

for each $S \varepsilon \Pi_1$, where

$$\Pi_S = \{T: T \varepsilon \Pi_2,\ T \subseteq \Sigma(S)\},$$

is a compiler under $\sigma \cdot \delta$.

Proof.

It is easy to verify that

(1) $\sigma \cdot \delta\ \underline{in} = \underline{in}$.

For the other conditions, let $D = \text{domain } (\sigma \cdot \delta)$. It follows that $D \subseteq \text{domain } (\delta) = D(\delta)$ and that $\delta(d) \varepsilon \text{ domain } (\sigma) = D(\sigma)$ for each $d \varepsilon D$. The proof that

(3) $\underline{out} \cdot \sigma \cdot \delta = \underline{out}$ on D,

is now straightforward. Next, if $S \varepsilon \Pi_1$ we have

$$\theta(S)\cdot\sigma\cdot\delta = \theta(\Sigma S)\cdot\delta \quad \text{on} \quad D,$$

$$= (\exists T \in \Pi_S)\ \theta(T)\cdot\delta \quad \text{on} \quad D,$$

$$= (\exists T \in \Pi_S)\ \theta(\Delta T) \quad \text{on} \quad D,$$

$$= \theta(\Delta\cdot\Sigma(S)) \quad \text{on} \quad D.$$

Thus

$$(4)\quad \theta(S)\cdot\sigma\cdot\delta = \theta(\Delta\cdot\Sigma(S)) \quad \text{on} \quad D$$

is established.

Next, using Lemmas 1.2 and 1.4, we have for each $S \in \Pi_1$ that

$$\phi(\Sigma S)\cdot\delta = \delta\cdot\phi(\Delta\cdot\Sigma(S))$$

on $D(\delta)$, and hence on D, since $D \subseteq D(\delta)$. Thus

$$\phi(S)\cdot\sigma\cdot\delta = \sigma\cdot\phi(\Sigma S)\cdot\delta = \sigma\cdot\delta\cdot\phi(\Delta\cdot\Sigma(S)) \text{ on } D,$$

which is just condition (5).

Finally, we must verify

$$(2)\quad \phi(\Delta\cdot\Sigma(S)) \text{ maps } D \text{ into } D$$

for all $S\epsilon\Pi_1$. Suppose, by way of contradiction, that for some $S\epsilon\Pi_1$ and $d\epsilon D$, we have

$$\phi(\Delta\cdot\Sigma(S))(d) \in D(\delta) - D.$$

It follows that

$$\delta\cdot\phi(\Delta\cdot\Sigma(S))(d) \notin D(\sigma),$$

and so

$$\phi(\Sigma S) \cdot \delta(d) = \delta \cdot \phi(\Delta \cdot \Sigma(S))(d) \notin D(\sigma).$$

But this contradicts the condition that $\phi(\Sigma S)$ maps $D(\sigma)$ into $D(\sigma)$, since we know $\delta(d) \in D(\sigma)$. Thus the final condition is verified, and the theorem is proved.

§5. Modified programs

It is often useful, when describing translations, to consider target programs that contain unconditional jump instructions. Unconditional jumps, which will be written in the form

$$\ell : \rightarrow \ell'$$

can always be eliminated by systematic label conversion. Thus, if $\ell : \rightarrow \ell'$ occurs as an instruction in a program P, we can delete it and replace all references to ℓ in P by references to ℓ'. Some care must be exercised in certain situations; for example, if both $\ell : \rightarrow \ell'$ and $\ell' : \rightarrow \ell$ occur in program P, all references to both ℓ and ℓ' must be converted to a reference to an infinite loop.

In fact, it is often convenient to assume that every program for a given machine contains a single initial jump

$$\underline{\text{start}} : \rightarrow \ell \ , \qquad \text{for some} \quad \ell \in L,$$

where start is the initial label given for the machine, and that no program ever contains an instruction which can reference start. This assumption, which enables the description of translations to be given more simply, entails no loss of generality.

CHAPTER TWO

BASIC DEFINITIONS AND RESULTS

In this chapter, register machines and their programs are described formally, and the definition of the running time of a program made precise (Section 1). As a consequence of the definition, certain programs may have a zero running time, and Section 2 characterises the class of functions such programs compute. Section 3 develops some useful notation on the relative growth of functions, and this is used in Section 4 to state some simple relationships between the function computed by a program and its running time. Section 5 considers the important notion of on-line programs, and Sections 6 and 7 describe some useful ways of combining and modifying programs.

§1. Register machines and their programs

The basic hardware of the machines which we shall study consists of a denumerable sequence of <u>registers</u>

$$A_1, A_2, \ldots A_n, \ldots,$$

each of which is capable of containing an arbitrary positive or negative integer, including zero. We refer to n as the <u>address</u> of register A_n. In addition, there are two special registers X and Y, called the <u>input register</u> and <u>output register</u> respectively;

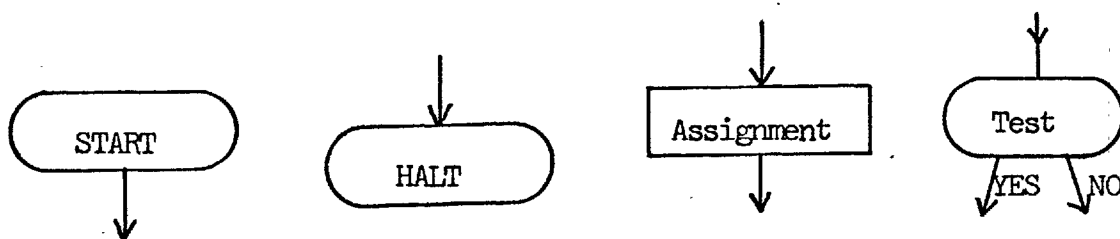
these can only contain non-negative integers. Thus our machines, when supplied with programs, compute partial 1-place number theoretic functions. This is not critical for the work that follows; only slight changes would be needed to deal with input and output registers that contain arbitrary strings of symbols over some finite alphabet, or both positive and negative integers.

In addition to these registers, the machines supply an <u>input function</u> and an <u>output function</u>, which respectively initialise a machine for a computation, and extract a final result.

(1) The input function loads a given non-negative integer in the register X, and sets all other registers to zero.

(2) The output function extracts the final contents of the output register Y as the result of the computation.

Programs for these machines consist of arbitrary flowcharts made up out of the following objects:



Flowcharts, which are defined in the obvious way, can be represented, where convenient, as sets of labelled instructions (as in Chapter 1). Both representations of programs have their advantages; with flowcharts there is no need to make the decision

as to what labels to use, but sets of labelled instructions are most useful when describing transformations on programs.

The instruction set (or order code) of a machine determines what assignments and tests the program can use. For the most part, we shall not be concerned with exactly what instructions are allowed, except to make the following remarks.

(1) The only instructions involving the input and output registers are:

assignments	X:=X-1	Y:=Y+1
tests	X=0.	

These have the obvious interpretation - we assume that if the assignment X:=X-1 is executed when X is empty, then X is unchanged - and ensure that each machine treats the input register as a read-only register, and the output register as a write-only register.

(2) Each instruction set contains the basic instruction set I_0, where I_0 consists of:

assignments	$A_i:=A_j+1$	$A_i:=A_j-1$
	$A_i:=A_j$	$A_i:=0$
tests	$A_i=0$	$A_i \geq 0$,

where i and j are arbitrary positive integers. These instructions, which effectively define the original Shepherdson-Sturgis set, have their obvious interpretation, and their effect is not formally defined.

(3) Each instruction set defines programs which are relocatable. This term is defined precisely later, but means roughly that we can always modify programs so that they do not refer to certain areas of store, and so do not interfere with other programs which may be running at the same time.

When an instruction set is specified, the machine is completely defined. Where necessary, we refer to a program defined over an instruction set I, as an I - program.

When a program P is executed (on its register machine) a function f_P is defined, and we say P computes f_P. A formal definition of how the values of this function are obtained is not given; sufficient details can be found in Chapter 1. We shall be concerned throughout only with programs that compute total functions. For each I, the class of I-programs which compute total functions is, of course, undecidable. Most of the results are proved constructively, i.e. usually by showing that a given program or programs can be transformed in some fashion in order to satisfy certain properties, and consequently can be given a valid interpretation when the programs do not compute total functions.

We say that program P is equivalent to program Q if $f_P=f_Q$.

Along with f_P, each program P defines another (total) function t_P, called the running time of P, which is defined as follows:

$t_P(x)$ = the total number of work register instructions (both assignments and tests) executed by P, when run on input x.

This measure of the complexity of a program is the only one with which we shall be concerned (Bird [5] [6] deals with space-restricted register machine computations). It is to be noted that input and output instructions are not counted towards the running time of a program. This assumption has certain convenient consequences, and is counter balanced by the fact that such instructions can only read the input and store the output.

Occasionally we shall use the terms instantaneous description, and computation sequence. An instantaneous description of a point during a computation of a program P is a vector,

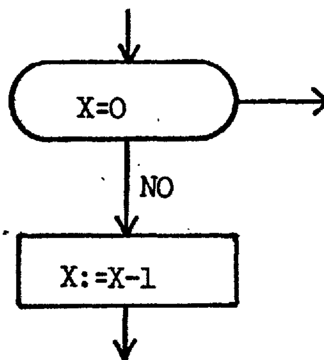
$$(\ell, x, y, a_1, a_2, \ldots),$$

which represents the fact that P is just about to execute the instruction with label ℓ, (in some consistent labelling of the instructions of P) and the current contents of the registers X, Y, A_1, ... etc., are x, y, a_1, a_2, ... etc. A computation sequence is a seqeunce of instantaneous descriptions. A register A_j is referred to by a program P if there is a computation sequence of P with some input, in which a label of an instruction involving register A_j occurs.

A set of instructions I is said to be relocatable if, given any two programs P and Q defined over I, we can find programs P' and Q', equivalent to P and Q respectively, such that no

work register referred to by P' is referred to by Q' and vice-versa.

Finally, two simple translations. Sometimes it is useful to suppose that the input instructions of a program only ever occur in the form



that is, every input test is immediately followed by an X:=X-1, and each such instruction is always preceeded by a test. A program with this property is said to be in standard input form. We can always convert a program into an equivalent one in standard input form. This is fairly obvious, but we give details of the formal translation as it indicates the general manner of describing translations which we shall employ.

The step by step compiler Δ which achieves the desired result, produces instructions with labels (apart from start) of the form

$$(\alpha, m),$$

where m is arbitrary, and $\alpha = 0$ or 1. The translation Δ satisfies the five conditions of the compiler theorem under the mapping δ, where

$$\delta(\underline{\text{start}}, x, y, a_1, a_2, \ldots) = (\underline{\text{start}}, x, y, a_1, a_2, \ldots)$$

and $$\delta((\alpha, m), x, y, a_1, a_2, \ldots) = (m, x+\alpha, y, a_1, a_2, \ldots)$$

Δ is given by:

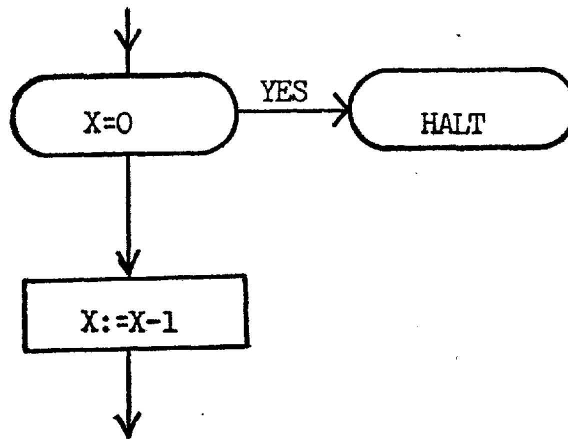
1. $\Delta\ [\underline{\text{start}}: \to m] = [\underline{\text{start}}: \to (0, m)]$
2. $\Delta\ [m: X=0 \to m', m''] = \{(0,m): X=0 \to (0, m'), \ell_m,$
   $\ell_m: X:=X-1 \to (1, m''),$
   $(1, m): \to (1, m'')\}$

3. $\Delta\ [m: X: = X-1 \to m'] = \{(0, m): X = 0 \to (0, m'), \ell_m,$
   $\ell_m: X: = X-1 \to (0, m'),$
   $(1, m): \to (0, m')\}$

4. $\Sigma[i] = \bigcup_\alpha i_\alpha$, for all other instructions i, where if
   $i: = m: f \to m'$, then $i_\alpha = (\alpha, m): f \to (\alpha, m')$ or
   if $i: = m: t \to m', m''$, then $i_\alpha = (\alpha, m): t \to (\alpha, m'), (\alpha, m'')$.

If $\Delta(P) = Q$, then P is equivalent to Q since Δ is a compiler. Q contains unconditional jumps which can be eliminated by label conversion (Section 1.5), and is clearly in standard input form. The verification that Δ satisfies the five conditions of the compiler theorem is left to the reader. Note that in (3), we assume that the effect of executing X:=X-1, when X is empty, is to leave X unchanged.

When all the input and halt instructions of a program occur in the form



We say that the program is on-line. On-line programs possess a number of different characteristics to ordinary programs and are very important in the sequel. Section 2.5 goes into more detail about them.

The last translation of this section is concerned with the instruction set I_0. We prove that for any I_0-program P we can find another equivalent I_0-program Q which stores only non-negative integers in its work registers (and so makes no use of the test $A_i \geq 0$). Q is a step by step simulation of P in which only the absolute values of the contents of the registers is stored and the labels of Q contain information as to the proper sign of these contents. More formally, suppose k is the maximum address referred to by P. The step by step translation Δ_k, for which $\Delta_k(P) = Q$, produces instructions with labels (apart from start) of the form

$$(\alpha, m)$$

where m is arbitrary, and $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ with $\alpha_j = 0$ or 1. The mapping δ_k, which is associated with Δ_k, is

$$\delta_k\ (\underline{start}, x, y, a_1, \ldots, a_k) = (\underline{start}, x, y, a_1, \ldots, a_k)$$

$$\delta_k\ ((\alpha, m), x, y, a_1, \ldots, a_k) = (m, x, y, b_1, \ldots, b_k),$$

where $b_j = a_j$ if $\alpha_j = 0$, and $b_j = -a_j$ if $\alpha_j = 1$.

Δ_k is given by:

(1) $\Delta_k\ [m: A_i: = A_j+1 \rightarrow m']$

$$= \bigcup_{\alpha_j=0} \{(\alpha, m): A_i: = A_j+1 \rightarrow (\alpha', m')\}$$

$$\bigcup_{\alpha_j=1} \{(\alpha, m): A_j=0 \rightarrow \ell_m, \ell_m'$$

$$\ell_m: A_i: = A_j+1 \rightarrow (\alpha', m')$$

$$\ell_m': A_i: = A_j-1 \rightarrow (\alpha'', m')\}$$

where $\alpha_r' = \alpha_r'' = \alpha_r$ for $r \neq i$, $\alpha_i' = 0$ and $\alpha_i'' = 1$.

(2) $\Delta_k\ [m: A_i: = A_j-1 \rightarrow m']$

$$= \bigcup_{\alpha_j=1} \{(\alpha, m): A_i: = A_j+1 \rightarrow (\alpha', m')\} \cup$$

$$\bigcup_{\alpha_j=0} \{(\alpha, m): A_j = 0 \rightarrow n_m, n_m'$$

$$n_m: A_i: = A_j+1 \rightarrow (\alpha', m'),$$

$$n_m': A_i: = A_j-1 \rightarrow (\alpha'', m')\}$$

where $\alpha_r' = \alpha_r'' = \alpha_r$ for $r \neq i$, $\alpha_i' = 1$ and $\alpha_i'' = 0$.

(3) $\Delta_k [m: A_i \geq 0 \to m', m''] = \bigcup_{\alpha_i=0} \{(\alpha, m): \to (\alpha, m')\} \cup$

$\bigcup_{\alpha_i=1} \{(\alpha, m): A_i = 0 \to (\alpha, m'), (\alpha, m'')\}$

(4) $\Delta_k[i] = \bigcup_{\alpha} i_\alpha$, for all other instructions i.

In the definition, $\bigcup_{\alpha_j=0}$ means the union over all α for which $\alpha_j = 0$; similarly for $\bigcup_{\alpha_j=1}$.

The verification that Δ_k satisfies the conditions of the compiler theorem under δ_k is again left to the reader.

§2. Functions computable within zero time

Programs, which consist only of input and output instructions, have zero running time and compute functions of a simple but important type: the ultimately linear functions.

A function f is said to be ultimately linear, if there exist integers c, d, and x_o with $c \geq 1$, such that

$$f(x+c) = f(x)+d \quad \text{for all} \quad x \geq x_o .$$

If $d = 0$, then f is also said to be ultimately periodic.

Theorem 2.1 A function f can be computed by a program with zero running time if and only if f is ultimately linear.

Proof. (a) necessity. Suppose f is computed by P such that $t_P(x) = 0$ for all x, and suppose P contains k distinct instructions of the form X:=X-1. If for no input does P obey more than k such instructions, then for each input $x > k$, the computation sequence of P with input x will be identical, and so

$$f(x+1) = f(x) \quad \text{for} \quad x > k,$$

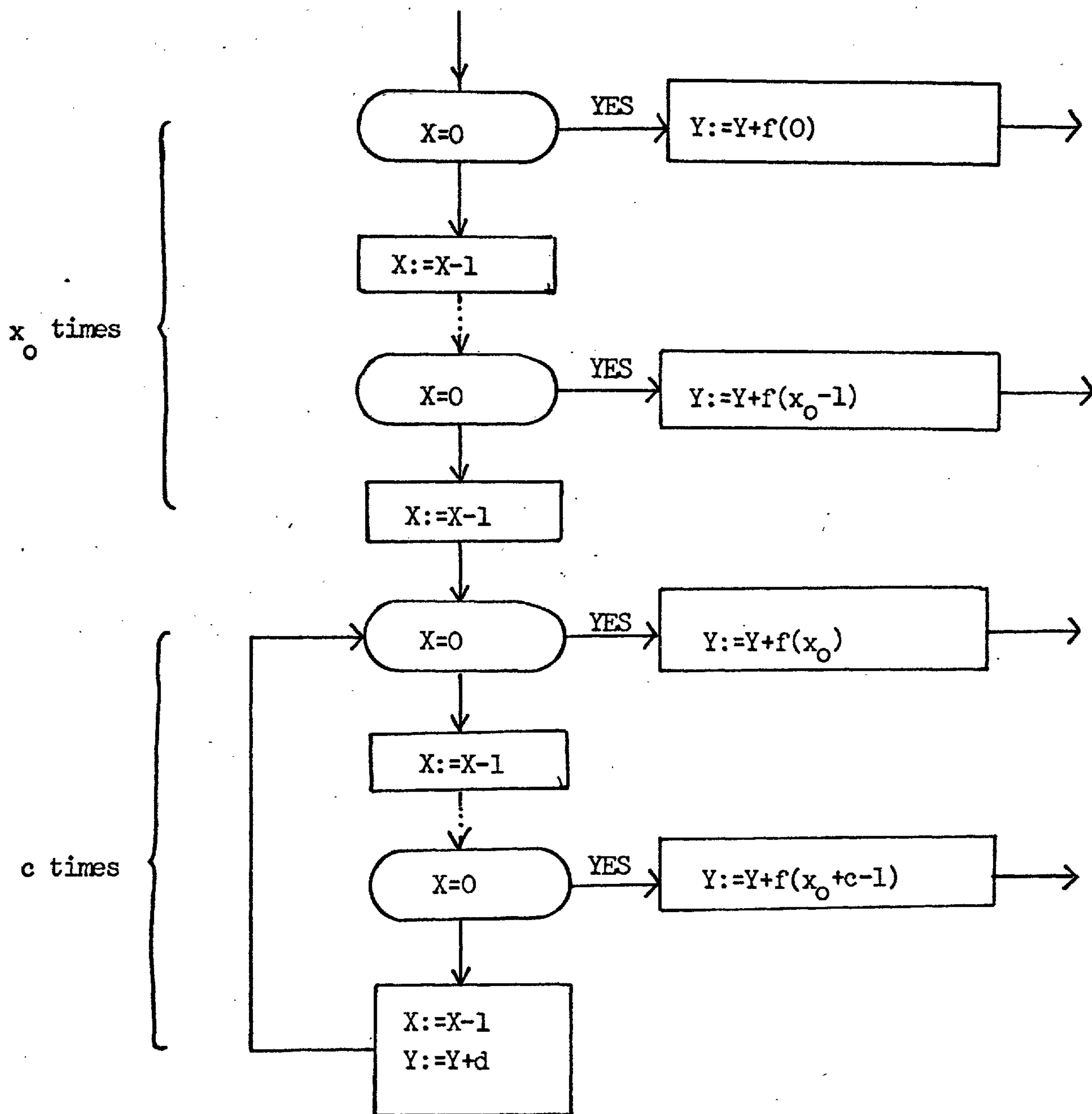
whence f is ultimately periodic. If, on the other hand, P does obey more than k such instructions for some input x_o, then some instruction will be executed at least twice. Let I be the seqeunce of instructions executed between the two occurrences of the first

such repeated instruction. Since the running time of P is zero, no work register instruction appears in I, and so the computation of P with input x_o will continue to cycle on I until the input is reduced to zero. Moreover, exactly the same situation will arise for the computation of P with an input $x \geq x_o$. If c is the number of instructions of the form X:=X-1 and d is the number of output instructions contained in I, then it follows that

$$f(x+c+1) = f(x)+d \quad \text{for} \quad x \geq x_o,$$

whence f is ultimately linear.

(b) sufficiency. Suppose that $f(x+c) = f(x) + d$ for $x \geq x_o$. The following program computes f and has zero running time:


X=0
YES
Y:=Y+f(0)
X:=X-1
X=0
YES
Y:=Y+f(x_o-1)
X:=X-1
x_o times
X=0
YES
Y:=Y+f(x_o)
X:=X-1
X=0
YES
Y:=Y+f(x_o+c-1)
X:=X-1
Y:=Y+d
c times

Hence the theorem is proved.

We note, for later use, the following fact about the ultimately linear functions.

<u>Lemma</u> 2.2 If f is ultimately periodic, then Σf is ultimately linear.

<u>Proof</u>. Suppose $f(x+c) = f(x)$ for $x \geq x_o$,

whence $$f(x) = f(x_o + [x-x_o, c]) \text{ for } x \geq x_o.$$

It follows that $$\sum_{y=x+1}^{x+c} f(y) = \sum_{y=x_o}^{x_o+c-1} f(y) = d \text{ say,}$$

for $x \geq x_o$. Restated, this says

$$(\Sigma f)(x+c) = (\Sigma f)(x) + d \quad \text{for} \quad x \geq x_o,$$

whence Σf is ultimately linear.

§3. <u>Asymptotic notation</u>

Before proceeding further, it is useful to describe some notation for comparing the rate of growth of functions.

(1) We shall write $f \preccurlyeq g$ to mean that there exists a positive constant c and an integer x_o such that

$$f(x) \leq cg(x) \text{ for all } x > x_o.$$

(2) We shall write $f \lhd g$ to mean that $g \trianglelefteq f$ is false.

(3) We shall write $f \sim g$ to mean both $f \trianglelefteq g$ and $g \trianglelefteq f$.

The notation $f \trianglelefteq g$ is equivalent to the more usual mathematical notation $f = Og$; each has its advantages, but the former is more natural in that it emphasizes the transitive nature of $\trianglelefteq$. It is straightforward to show that

(a) $f \lhd g$ if and only if $\liminf_{x \to \infty} \frac{f(x)}{g(x)} = 0$.

Further facts noted without proof are:

(b) $\trianglelefteq$ is reflexive and transitive,

(c) If $f \trianglelefteq h$ and $g \trianglelefteq h$, then $f+g \trianglelefteq h$.

The notation will frequently be abused to the extent that we shall write $f(x) \trianglelefteq x$ etc., to mean more precisely that $f \trianglelefteq i$, where i is the identity function.

§4. Three basic theorems

We now prove three basic results which will be referred to subsequently as the almost everywhere theorem, the minimal growth rate theorem, and the size theorem.

Theorem 2.3 Suppose P is a program such that

$$t_P(x) \leq f(x) \quad \text{for all} \quad x > x_0,$$

where f is some specified function. Then we can find a program Q, equivalent to P and on-line if P is, such that

$$t_Q(x) \leq f(x) \quad \text{for all} \quad x.$$

Proof. Let $c = \max\limits_{x \leq x_0} t_P(x)$. Program P can be converted into an equivalent program Q for which $t_Q(x) = 0$ if $t_P(x) \leq c$, and $t_Q(x) = t_P(x)$ if $t_P(x) > c$. It follows from this that $t_Q(x) \leq f(x)$ for all x. We give only an informal description of Q. Q simulates P in a step by step fashion, except that, in any computation, Q delays the execution of the first c work register instructions executed by P. These instructions are remembered in the label structure of Q, and are only executed when the $(c+1)$st work register instruction of P is about to be executed. Q can clearly be made on-line if P is, and the running time estimate of Q follows at once.

Theorem 2.4 If P is a program such that $t_P(x) \lhd x$, then f_P is ultimately linear.

Proof.

Suppose that P contains k instructions of the form X:=X-1. By hypothesis, there is an integer x_0, as large as we please and hence greater than k, such that $t_P(x_0) < [x_0/k]$. Consider the computation sequence of P with input x_0. If this sequence contains less than x_0 instructions of the form X:=X-1, then f_P is ultimately periodic by the same reasoning as Theorem 2.1. On the other hand, if this sequence contains x_0 such instructions, then it must contain two occurrences of the same input instruction, between which no work register instruction occurs. To see this, suppose $n > 0$ is such that $nk \leq x_0 < (n+1)k$. If the above situation does not arise, then $t_P(x_0) \geq n = [x_0/k]$ contrary to hypothesis. The rest of the proof is now identical with Theorem 2.1.

Theorem 2.4 is the best possible in the sense that one can construct a program P for which $t_P(x) \trianglelefteq x$, such that f_P is not an ultimately linear function; for example, $f_P = a$, where

$$a(x) = (x \text{ even} \rightarrow x, 0).$$

In fact, Theorem 2.4 can be used to show that no program P can have an unbounded running time t_P such that $t_P(x) \triangleleft x$. Suppose such a P exists. By removing the output instructions of P, and inserting new ones after each work register instruction, we obtain a program Q such that $f_Q = t_P$ and $t_Q = t_P$. Since $t_Q(x) \triangleleft x$, we must have that $f_Q = t$ is ultimately linear; but since $t_P(x) \triangleleft x$, t_P must be ultimately periodic and hence bounded.

<u>Theorem</u> 2.5 Suppose P is a program such that $x \trianglelefteq t_P(x)$.

Then $f_P(x) \trianglelefteq t_P(x)$.

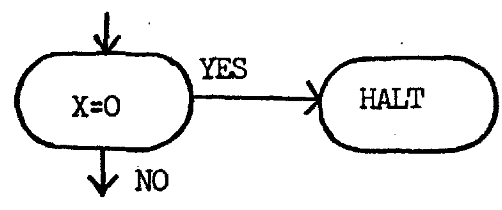
<u>Proof</u>. Suppose P has k output instructions. Consider the computation of P with an arbitrary input x. During this computation, no output instruction can be obeyed twice with exactly the same contents of the input and work registers, since otherwise P would go into an infinite loop and fail to terminate. Thus between successive additions of k to Y, either the contents of X or some work register must have changed, and so

$$f_P(x) \leqslant k(1+x+t_P(x)) \trianglelefteq t_P(x),$$

since $x \trianglelefteq t_P(x)$.

§5. On-line programs

As mentioned in Section 2.1, on-line programs are programs in which input tests and halt instructions only occur in the form



Programs with this property must operate in a particular fashion: if P is an on-line program, then for all x and all $y \leq x$, the computation sequence of P with input x is identical for the first $t_P(y)$ steps with the computation sequence of P with input y. This fact is easily proved by induction. It follows that every computation sequence of P is an initial subsequence of just one computation sequence - the computation sequence of P with infinite input. It is sometimes useful to consider properties of on-line programs in terms of this hypothetical sequence.

It also follows that on-line programs compute monotone functions, where a function f is said to be monotone if for all x and y

$$x \leq y \quad \text{implies} \quad f(x) \leq f(y).$$

Below, we shall show that every monotone computable function is on-line computable, i.e. is computable by some on-line program. The concept of on-line programs is crucial to certain constructions, most important of which is the inversion theorem (Section 2.7). Because of the restrictive nature of on-line computations, it is possible to obtain sharper bounds on the running time of on-line programs, than would otherwise be possible (Chapter 5). Arbib [3, 1.3] contains an interesting discussion of the on-line phenomenon in Automata

theory; see also Hennie [22].

<u>Theorem</u> 2.6 For each program P which computes a monotone function we can find an on-line program Q, equivalent to P, such that

$$t_Q \trianglelefteq \Sigma t_P .$$

<u>Proof</u>. First suppose that $t_P(x) \triangleleft x$. In this case, f_P is ultimately linear by the minimal growth rate theorem, and there exists an on-line program Q computing f_P for which $t_Q = 0$.

Suppose, on the other hand, that $x \trianglelefteq t_P(x)$. Without going into details, it is possible to construct from P a program R such that

$$\begin{aligned} f_R(x) &= f_P(x) - f_P(x-1) \quad \text{if } x > 0 \\ &= f_P(0) \qquad\qquad\qquad \text{if } x = 0 \end{aligned}$$

and

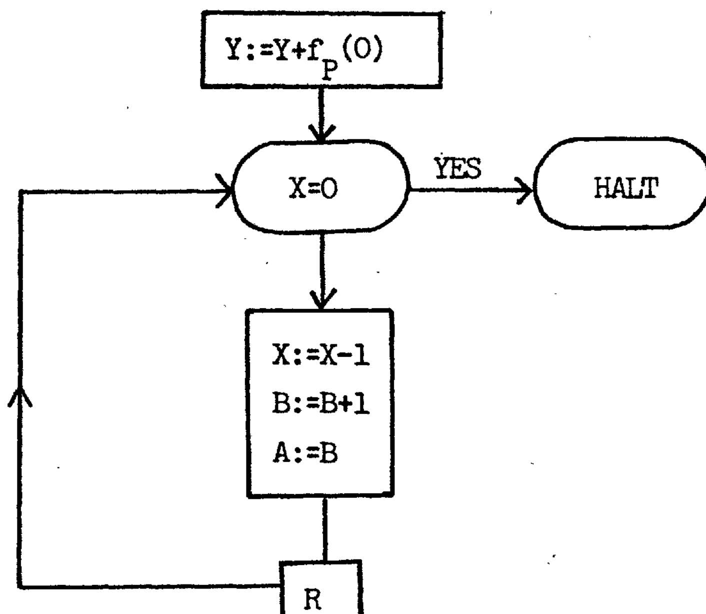
$$t_R(x) \trianglelefteq t_P(x) + t_P(x-1) + f(x) + f(x-1) + x.$$

For each input x, R works by first using P to compute $f_P(x-1)$ into a work register not used by P, and then using P again to compute $f_P(x)$, sending only the difference to the output register. (Since programs are relocatable, it is always possible to select work registers not used by P.) Since the running time of R has to include the time required to store $f_P(x-1)$ and $f_P(x)$ in order to compute the difference, and also the time required to preserve the contents of the input register for the second computation, it satisfies the above inequality. Using the size theorem and the fact

that $x \leqslant t_P(x)$, we have

$$t_R(x) \leqslant t_P(x) + t_P(x-1).$$

Suppose that A and B are two registers not appearing in R. R is modified so that all references to the input register X are converted to references to register A, and Q is defined to be the program:



Q is on-line and, by construction, equivalent to P.

Moreover

$$t_Q(x) \leqslant \sum_{y=0}^{x} (t_R(y) + y) \leqslant \sum_{y=0}^{x} (t_P(y) + t_P(y-1)) \leqslant \sum_{y=0}^{x} t_P(y).$$

Corollary 2.7 Suppose P computes a monotone function and

$$\liminf_{x\to\infty} \frac{t_P(x+1)}{t_P(x)} > 1.$$

Then there is an on-line program Q equivalent to P, such that

$$t_Q \preceq t_P .$$

Proof. The result follows from Theorem 2.6 by showing that $\Sigma t_P \preceq t_P$. By hypothesis, there is a real number $\delta > 1$ and an integer x_o such that

$$t_P(x+1) \geq \delta\, t_P(x) \quad \text{for all} \quad x \geq x_o.$$

Thus

$$\sum_{y=o}^{x} t_P(y) \leq \sum_{y=o}^{x_o-1} t_P(y) + t_P(x) \sum_{r=o}^{x-x_o} \delta^{-r} \preceq t_P(x),$$

Since $\sum_{r=o}^{\infty} \delta^{-r}$ converges.

In Chapter 5, it is shown that this corollary fails to hold if the condition $\liminf_{x\to\infty} \frac{t_P(x+1)}{t_P(x)} > 1$ is dropped.

§6. The composition and addition of programs

Among the various ways of combining programs, there are two which will be used frequently. These relate to the following functional operations:

(a) <u>composition</u>. The composition $f \cdot g$ of two functions f and g is that function which is defined by

$$(f \cdot g)(x) = f(g(x)) \quad \text{for all } x.$$

(b) <u>addition</u>. The addition $f+g$ of two functions f and g is that function which is defined by

$$(f+g)(x) = f(x) + g(x) \quad \text{for all } x.$$

Below, we give constructions which carry out these operations within tight time bounds.

<u>Theorem</u> 2.8 Given any two programs P and Q, we can construct a program $P \cdot Q$ such that

$$f_{P \cdot Q} = f_P . f_Q$$

and

$$t_{P \cdot Q} = t_Q + t_P . f_Q .$$

Moreover, if P and Q are on-line, then so is $P \cdot Q$.

<u>Proof</u>.

The standard way of computing the composition $f \cdot g$ is to store the partial result $g(x)$ in a new work register, and use this as input to the program computing f. This method cannot be used here, since the time taken to store and retrieve $g(x)$ adds to the total running time of the program. However, another method is available which uses P and Q as co-routines. P and Q must first be modified so that they make use of distinct sets of work registers. Since we are assuming that programs are relocatable, this causes no trouble. Informally, the program $P \cdot Q$ works as follows: control

starts off in P which remembers the label of the first instruction of Q in its label structure. P is executed until it attempts to carry out an input test, when control is handed to Q which begins computing at the remembered entry point. Q continues until it attempts to obey an output instruction when control is handed back to P which continues from where it left off, under the assumption that the input was not empty. When P asks for more input, Q is re-entered until another output is given. If P halts instead, control is returned to Q until it too halts. If Q halts first, then control is returned to P which continues on alone under the assumption that the input was empty.

We now describe P·Q formally. Let L denote the set of labels of P which are either terminal labels or labels of input tests. It is convenient to suppose that each terminal label ℓ corresponds to an explicit halt instruction, written as ℓ: halt.

We have $$P\cdot Q = \bigcup_{\ell\varepsilon L} \Sigma_\ell(Q) \cup \bigcup_{m\varepsilon\lambda(Q)} \Delta_m(P),$$

where $\lambda(Q)$ is the label set of Q, and Σ_ℓ and Δ_m are two step by step translations defined as follows:

1. If ℓ labels an input test ℓ: X=0 $\rightarrow \ell', \ell''$ of P, then

   (a) $\Sigma_\ell\ [m\text{: Y: = Y+1} \rightarrow m'] = \{(\ell, m)\text{: } \rightarrow (m', \ell'')\}$

   (b) $\Sigma_\ell\ [m\text{: halt}] = \{(\ell, m)\text{: } \rightarrow (m, \ell')\}$

   (c) $\Sigma_\ell\ [I] = I_\ell$ otherwise, where

   if $I = m\text{:}f \rightarrow m'$, then $I_\ell = (\ell, m)\text{: } f \rightarrow (\ell, m')$

   or if $I = m\text{:}t \rightarrow m', m''$, then $I_\ell = (\ell, m)\text{: } t \rightarrow (\ell, m'), (\ell, m'')$.

2. If ℓ labels ℓ: halt of P, then

$$\Sigma_\ell[I] = I_\ell \quad \text{for all } I.$$

3. Δ_m is defined by:

(a) Δ_m [start: → ℓ] = {start: → (m', ℓ)},

where start: → m' is contained in Q.

(b) $\Delta_m\,[\ell{:}X{=}0 \rightarrow \ell', \ell''] = \{(m, \ell){:} \rightarrow (\ell, m)\}$

(c) $\Delta_m\,[\ell{:}X{:=}X{-}1 \rightarrow \ell'] = \{(m, \ell){:} \rightarrow (m, \ell')\}$

(d) $\Delta_m\,[\ell{:}\text{halt}] = \{(m, \ell) \rightarrow (\ell, m)\}$

(e) $\Delta_m\,[I] = I_m$ otherwise.

The program P·Q contains unconditional jumps which can be eliminated by label conversion. In order to see that P·Q will be on-line if P and Q are, note that the only input tests of P·Q come from Q via 1(c) or 2. If Q contains m: X=0 → m', m'', where m is terminal, then P·Q contains (ℓ, m): X=0 → (ℓ, m'), (ℓ, m'') for each $\ell \in L$. But by 1(b), P·Q also contains (ℓ, m'): → (m', ℓ''), where ℓ: X=0 → ℓ', ℓ'' is contained in P. If P is on-line, then ℓ' is terminal, and so (m', ℓ') is a terminal label of P·Q. Hence P·Q is on-line. The timing estimate follows at once.

We shall refer to the above theorem as the <u>composition theorem</u>, and to the next as the <u>addition theorem</u>.

Theorem 2.9 Given any two programs P and Q, we can construct a program P+Q such that

$$f_{P+Q} = f_P + f_Q$$

and
$$t_{P+Q} = t_P + t_Q.$$

Moreover, P+Q is on-line if both P and Q are.

Proof.

The construction is very similar to the previous one, and only an informal description is given. In P+Q, the programs P and Q act as co-routines which share the input. P and Q are again modified to refer to distinct sets of work registers, and Q is put in standard input form. In the combined program, Q alone is given the task of testing and decrementing the input register. Control starts off in P which remembers the first label of Q in its label structure. Each time P attempts to execute an input test, control is handed to Q which begins computing from where it left off. Each time Q executes an input test, control is returned to P. This process continues until one of P and Q wants to halt, when the other program is entered until it too halts. The final contents of the output register will be the sum of the contributions from P and Q, and the running time of the combined program, will be the sum of the running times of P and Q.

§7. The inversion of programs

Suppose f is a monotone and unbounded function. We define the *inverse* f^* of f to be the function whose values are given by

$$f^*(x) = \min y \left[x < f(y)\right].$$

f^* is also monotone and unbounded. In this section, we show how to construct from an on-line program P computing f, an on-line program P^* which computes f^*. The following facts about inverses are important and are used in Chapter 4.

Lemma 2.10 Suppose f and g are monotone and unbounded. Then

(i) $f^{**} = f$ and (ii) $(f \cdot g)^* = g^* \cdot p \cdot f$,

where $p(x) = x-1$.

Proof. (i) By definition,

$$f^{**}(x) = \min y \left[x < \min z \left[y < f(z)\right]\right]$$

whence

$$\min z \left[f^{**}(x) < f(z)\right] > x.$$

From this it follows that $f^{**}(x) \geq f(x)$. Since f is monotone and unbounded, we have

$$\min z \left[f(x) < f(z)\right] > x,$$

from which it follows that $f^{**}(x) \quad f(x)$. Hence $f^{**}(x) = f(x)$.

(ii) We have

$$(f\cdot g)^*(x) = \min y \;[x < f\cdot g(y)]$$

$$= \min y \;[g(y) \geq \min z \;[x < f(z)]],$$

since f and g are monotone. Now

$$\min y \;[g(y) \geq x] = \min y \;[g(y) > x-1],$$

whence $(f\cdot g)^*(x) = g^*(f^*(x)-1)$.

Suppose next, that P is an arbitrary on-line program computing an unbounded monotone function. In such a case, the function θ_P, where

$\theta_P(x)$ = total number of work register instructions executed by P on any sufficiently large input (equivalently, on infinite input), prior to the execution of the (x+1)st output instruction,

is well defined, total and monotone.

We shall refer to the following theorem as the inversion theorem.

Theorem 2.11 Suppose P is an on-line program computing an unbounded function. Then we can construct an on-line program P* such that

$$f_{P*} = f_P^*$$

and

$$t_{P*} = \theta_P \leq t_P\cdot f_P^* \;.$$

Proof. Without loss of generality, we can assume that P is in standard input form. The program P* is exactly the same as P except that the input instructions are replaced by output instructions and vice-versa. More precisely, the translation Δ which produces P* is given by:

1. $\Delta\,[m\colon X{=}0 \rightarrow 0,\ m'] \;=\; [m\colon \rightarrow m']$

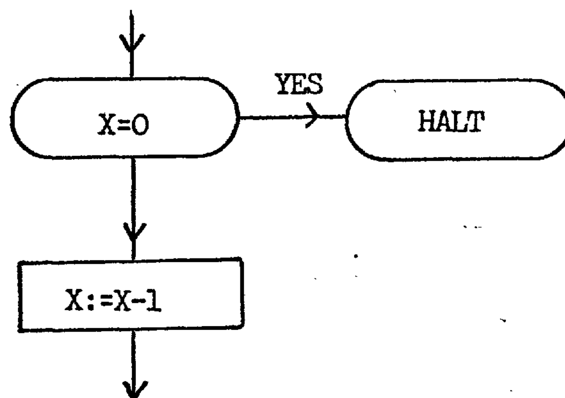
2. $\Delta\,[m\colon X{:=}X{-}1 \rightarrow m'] \;=\; [m\colon Y{:=}Y{+}1 \rightarrow m']$

3. $\Delta\,[m\colon Y{:=}Y{+}1 \rightarrow m'] \;=\; [m\colon X{=}0 \rightarrow 0,\ \ell$
   $\ell\colon X{:=}X{-}1 \rightarrow \ell']$

4. $\Delta[i] = i$, for all other instructions i.

(0 is regarded as the terminal label).

In order to prove that P* does indeed compute f_P^*, it is convenient to define two further functions. In the definitions, the term 'input instruction' refers to the pair



The functions i and o are defined by

$i(x)$ = total number of instructions, including input and output instructions, executed by P on any sufficiently large input, prior to the execution of the $(x+1)$st input instruction

$o(x)$ = similar to $i(x)$, but prior to the $(x+1)$st output instruction.

If P computes an unbounded function f, then i and o are both total and monotone (in fact, $i(x) = t_P(x) + x + f(x)$) and we have

$$o\,(f(n) - 1) < i(n) < o\,(f(n)), \qquad \ldots (1)$$

for all n. If i^* and o^* denote similar functions for P^*, then

$$i^* = o \quad \text{and} \quad o^* = i, \qquad \ldots (2)$$

since both P and P^* are in standard input form.

Consider the computation of P^* on an arbitrary input x. Since f is unbounded, there exists an n such that

$$f(n) \leqslant x < f(n+1), \qquad \ldots (3)$$

and we have

$$\begin{aligned} o^*(n) &= i(n) && \text{from (2)},\\ &< o(f(n)) && \text{from (1)},\\ &= i^*(f(n)) && \text{from (2)},\\ &\leqslant i^*(x), \end{aligned}$$

from (3), since i* is monotonic.

Also, $i^*(x) = o(x)$

$$\leq o(f(n+1)-1) \quad \text{since } o \text{ is monotonic,}$$

$$< i(n+1) \quad \text{from (1),}$$

$$= o^*(n+1).$$

Therefore,

$$o^*(n) < i^*(x) < o^*(n+1).$$

It follows from this that exactly (n+1) output instructions are executed, before the computation of P^* with input x terminates. Thus

$$f_{P*}(x) = n+1 = f^*(x).$$

Finally, we estimate t_{P*}. We have

$$t_{P*} = \theta_P$$

by definition of P^*, and moreover

$$\theta_P(f(x)-1) \leq t_P(x) \quad \text{for all } x, \qquad \ldots (4)$$

since P is on-line. Since

$$f(f^*(x)) > x \geq x + 1,$$

we have

$$\theta_P(x) \leq \theta_P\ (f(f^*(x))-1) \leq t_P(f^*(x)),$$

from equation (4) and the fact that θ_P is monotone.

CHAPTER THREE

THE SPEED UP THEOREM

Throughout this chapter, we shall be concerned only with I_0 - programs; that is, flowcharts defined over the instruction set:

assignments	$A_i:=A_j+1$	$A_i:=A_j-1$
	$A_i:=A_j$	$A_i:=0$
tests	$A_i=0$	$A_i \geq 0$.

(To avoid messy subscripts, register A_i will sometimes be denoted by Ai, etc.)

The central object of the chapter is to prove the following theorem, which will be referred to subsequently as the <u>speed-up-property</u> (for I_0).

<u>Theorem</u> 3.1 Given any program P, we can always find an equivalent program Q, which is on-line if P is, such that

$$t_Q \leq t_P/2.$$

Because the proof is fairly long (and will not be complete until the end of Section 4), we first give an informal summary of the main steps.

The running time t_P of a program P is the sum of two functions a_P and b_P, where

$a_P(x)$ = the number of assignments executed by P when run on input x,

$b_P(x)$ = similarly, the number of tests.

It is shown, in Section 4, that in order to prove the speed up property, it is sufficient to reduce a_P by a factor of two; i.e. we can forget about tests and concentrate on cutting the assignments by half. The reason this can be done, and the pivot upon which speed up turns, is the fact that, by taking a sufficiently long sequence of assignments, another equivalent sequence (in the strong sense of having the same effect on the work registers) can be found of no more than half the length of the first. On the other hand, the second sequence uses the more general assignments

$$A_i := A_j + d \qquad A_i := e$$

for integers d and e, where $|e| < |d|$. Section 1 is devoted to a proof of this fact. This result is used in the construction of two step by step compilers (although they can be combined into one) to achieve the desired reduction in a_P. In the first, (defined in Section 2), P is converted into an equivalent program R for which $a_R \leq a_P/2$. Program R uses a more general instruction set than I_0. In Section 3, R is converted into a proper I_0- program Q for which $a_Q \leq a_R$. Combining these two translations gives the final result.

Section 5 shows speed up at work on an example, and Section 6 contains some further results and a discussion.

§1. Finite sequences of assignments

Throughout this section, the letters S and S′ will denote finite sequences of assignments of the form

$$A_i := A_j + d \quad \text{and} \quad A_i := e,$$

where $1 \leq i, j \leq k$, and d and e are arbitrary integers with $|e| < |d|$. We say that S is a D-sequence if it consists only of assignments with $|d| \leq D$. Thus a 1 - sequence is a sequence of I_0 - assignments. The length of a sequence S is denoted by $||S||$, and the same notation is used to denote the number of elements in a finite set. With each sequence S we can associate two mappings:

$$\sigma_S: \{1, 2, \ldots, k\} \to \{0, 1, \ldots k\}$$

and

$$\rho_S: \{1, 2, \ldots, k\} \to \{\ldots, -1, 0, 1, \ldots\}$$

which serve to characterise S. These mappings are defined by the criterion that for each i, in the range $1 \leq i \leq k$, the effect of executing S changes the contents of register A_i, as if the single instruction

$$A_i := A\sigma_S(i) + \rho_S(i) \quad \text{if} \quad \sigma_S(i) > 0,$$

or

$$A_i := \rho_S(i) \quad \text{if} \quad \sigma_S(i) = 0,$$

were executed instead.

Note that if S is a D-sequence, then for $1 \leq i \leq k$,

$$|\rho_S(i)| \leq D \times ||S|| \quad \text{if} \quad \sigma_S(i) \neq 0;$$

and

$$|\rho_S(i)| \leq D \times (||S|| - 1) \text{ if } \sigma_S(i) = 0.$$

If S and S' are two sequences, then we say that S is equivalent to S if $\sigma_S = \sigma_{S'}$ and $\rho_S = \rho_{S'}$. Equivalent sequences thus have exactly the same effect when executed.

Our object in this section is to prove the following theorem.

Theorem 3.2 Given any D-sequence S, it is possible to construct a D' - sequence S', equivalent to S, such that

$$||S'|| \leq \min (||S||, k + \left[\frac{k+1}{2}\right] - 1).$$

In particular, if $||S|| = 3k$ (or 4, if $k=2$), then $||S'|| \leq ||S||/2$ and $D' = 3kD$ (or $4D$, if $k = 2$).

In order to prove this theorem, it is necessary to investigate certain properties associated with σ_S. Let N_k denote the set $\{0, 1, ..., k\}$ and for convenience, extend σ_S to a function $\sigma_S : N_k \to N_k$ by defining $\sigma_S(0) = 0$.

Suppose σ is any mapping $\sigma : N_k \to N_k$ with $\sigma 0 = 0$ (we often omit parentheses for brevity). The mapping $\sigma^n : N_k \to N_k$ is defined iteratively for each $n \geq 0$, by the equations

$$\sigma^0 i = i \quad \text{and} \quad \sigma^{n+1} i = \sigma(\sigma^n i)$$

for each $i \in N_k$. A cycle of σ is a subset C of N_k consisting of the integers

$$i, \sigma i, \sigma^2 i, \ldots, \sigma^{t-1} i,$$

for some $i \in N_k$ and positive integer t, such that the conditions

$$\text{(i)}\quad \sigma^t i = i,$$

and $\quad$ (ii) $\sigma^a i \neq \sigma^b i \quad$ for $\quad 0 \leq a < b < t,$

are satisfied.

Since each cycle of σ is uniquely determined by any one of its elements, we immediately have

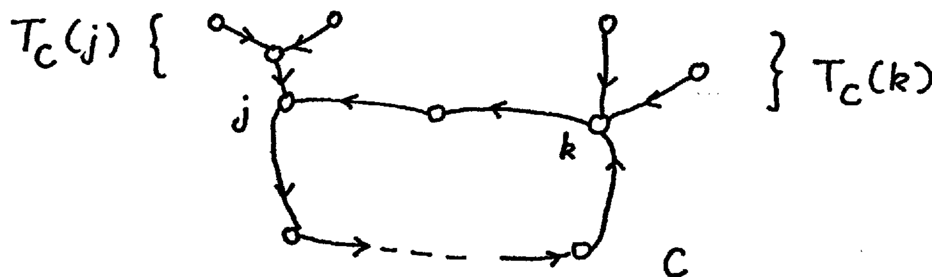
<u>Lemma</u> 3.3 If C and D are two cycles of σ which have an element in common, then $C = D$.

The <u>tree set</u> T_C of a cycle C is defined to be

$$T_C = \bigcup_{j \in C} T_C(j),$$

where $\quad T_C(j) = \{i:\ \sigma^n i = j \text{ for some } n > 0,$ and $\sigma^m i \notin C$ for all $m,\ 0 \leq m < n\}.$

A cycle and its tree set can be pictured as follows:



The relevant facts about tree sets are:

Lemma 3.4 (i). If $i,j \in C$ with $i \neq j$, then the sets $T_C(i)$ and $T_C(j)$ are disjoint.

(ii). If C and C' are two disjoint cycles, then the sets T_C and $T_{C'}$ are disjoint.

(iii). If C and C' are two cycles, possibly the same, then the sets T_C and C' are disjoint.

Proof. (i). Suppose $T_C(i)$ and $T_C(j)$ have the element m in common. There therefore exists two positive integers r and s with $\sigma^r m = i$ and $\sigma^s m = j$. If $s < r$, then by definition of $T_C(i)$, it is impossible that $j = \sigma^s m$ is in C. Similarly, if $r < s$, then it is impossible that $i \in C$. So if i,j are in C, we must have $r = s$ and so $i = j$.

(ii). Suppose T_C and $T_{C'}$ have the element m in common. There therefore exists positive r and s such that $\sigma^r m \in C$ and $\sigma^s m \in C'$. If $r \leqslant s$, then since C is a cycle, C contains

$$\sigma^{r+(s-r)} m = \sigma^s m.$$

So C and C' are not disjoint. A similar argument holds for $r \geqslant s$.

(iii). Suppose T_C and C' have the element m in common. It follows that $\sigma^n m$ is in both C and C' for some n. By Lemma 3.3 we must have $C = C'$, but by definition of T_C, the sets T_C and C are disjoint.

<u>Lemma</u> 3.5 For any σ, the collection of sets $\{C, T_C: C \text{ is a cycle of } \sigma\}$ partitions N_k, and so

$$\sum_C (||C|| + ||T_C||) = k+1$$

<u>Proof</u>. It is immediate from Lemmas 3.3 and 3.4 that the collection is pairwise disjoint. For each $i \in N_k$, consider the sequence of integers

$$i, \sigma i, \ldots, \sigma^{k+1} i.$$

Since there are (k+2) integers in this sequence, all of which are in N_k, they cannot all be different. Suppose $\sigma^a i = \sigma^b i$, where $a < b$, and suppose further that a is the least integer for which this is true. If $a = 0$, then i belongs to the cycle $\{i, \ldots, \sigma^{b-1} i\}$, while if $a \neq 0$, then i belongs to the tree set of the cycle generated by $\sigma^a i$. Thus the collection covers N_k.

There is one important difference between an arbitrary $\sigma: N_k \to N_k$ and one given by a sequence of instructions:

<u>Lemma</u> 3.6 Suppose S is a finite sequence of instructions with associated mapping σ_S. Either $\sigma_S(i) = i$ for all $i \in N_k$ (in which case S is said to be <u>simple</u>), or else there is at least one cycle of σ_S whose tree set is not empty.

<u>Proof</u>. If S is not simple, then it must contain some instruction of the form

$$A_i := A_j \pm d \quad \text{where} \quad i \neq j$$

or

$$A_i := e.$$

The first such instruction destroys the original contents of A_i, and so $i \notin$ range (σ_S). Therefore i cannot belong to any cycle of σ_S and so it must belong to some tree set.

We now show how to construct from a given sequence S, a sequence S' equivalent to S, which satisfies the hypothesis of Theorem 3.2. Suppose S is given and σ and ρ are the associated functions. If S is simple (i.e. $\sigma i = i$ for each i), then we can at once define S to be the sequence consisting of all the instructions of the form

$$A_i := A_i + \rho(i) \quad \text{where} \quad \rho(i) \neq 0 \quad \text{and} \quad 1 \leq i \leq k,$$

written in any order. Clearly $||S'|| \leq \min(||S||, k)$. Otherwise we determine the cycles and tree sets of σ. By Lemma 3.6, at least one tree set is not empty, so suppose, without loss of generality that $1 \in T_{C_0}$ and $\sigma 1 \in C_0$. For each cycle C and tree set T_C we define instruction sequences $S(C)$ and $S(T_C)$ as determined below. Supposing the cycles of σ are $C_0, C_1, \ldots, C_n$, the sequence S' is defined to be

$$S \;\; = S(T_{C_1}); \; S(C_1) \; \ldots\ldots. \; S(T_{C_n}); \; S(C_n): \; S(T_{C_0}); \; S(C_0)$$

The sequences $S(C)$ and $S(T_C)$ are determined as follows:

1. Sequence $S(T_C)$. This sequence is a concatenation of subsequences $K(m)$ for $m \in C$, written in any order. Each subsequence $K(m)$ serves to assign the correct final value to register A_j for each $j \in T_C(m)$ (except for j=1, which is a special case). To

define $K(m)$, we must order the elements $\{i_1, i_2, \dots i_n\}$ of $T_C(m)$ so that

$$\sigma i_r = i_s \text{ implies } r < s.$$

This is the familar end ordering of nodes in a rooted tree (with root m) and can be carried out by a standard procedure (e.g. Knuth [26]). For $1 \leq j \leq n$, define $I(j)$ to be the instruction

$$Ai_j := A\sigma i_j + \rho(i_j) \qquad \text{if} \qquad \sigma i_j > 0$$

or

$$Ai_j := \rho(i_j) \qquad \text{if} \qquad \sigma i_j = 0,$$

and $K(m)$ to be the sequence $I(1); I(2); \dots; I(n)$. Executing $K(m)$ assigns the correct final values, since the end ordering condition means that the sequence $I(1); I(2); \dots; I(r-1)$ for $1 \leq r \leq n$, does not alter the contents of register $A\sigma i_r$. The special case of register A_1 arises when $||C_0|| \geq 2$. In implementing cycles of length greater than 1 the register A_1 is going to be used as work space, so there is no point in assigning A_1 its correct final value, when there is still C_0 to come. In such a case, we define the instruction $I(j)$ (where $i_j = 1$) to be

$$A_1 := A_{\sigma 1}.$$

This means that to give A_1 its correct final value, $\rho(1)$ must be added at a later stage.

2. Sequence $S(C)$. If C is the special cycle $\{0\}$, then no instructions are defined, i.e. $S(C) = \text{null}$. If C is a cycle $\{i\}$ of length 1, then $S(C)$ is

$$A_i := A_i + \rho(i).$$

If $C = \{i, \sigma i, \ldots, \sigma^{t-1} i\}$, where $t > 1$, then $S(C)$ is the sequence

$$A_1 := A_i$$

$$A_i := A\sigma i + \rho(i)$$

$$A\sigma i := A\sigma^2 i + \rho(\sigma i)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$A\sigma^{t-1} i := A_1 + \rho(\sigma^{t-1} i),$$

provided $C \neq C_0$. For C_0 we have (supposing $||C_0|| \geq 2$),

$$C_0 = \{\sigma 1, \sigma^2 1, \ldots \sigma^t 1\}$$

for some $t > 1$, and we define $S(C_0)$ to be

$$A\sigma 1 := A\sigma^2 1 + \rho(\sigma 1)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$A\sigma^t 1 := A1 + \rho(\sigma^t 1)$$

$$A1 := A1 + \rho(1).$$

(The instruction $A_1 := A_{\sigma 1}$ is omitted because it occurs in $S(T_{C_0})$, and

the instruction $A_1 := A_1 + \rho(1)$ is inserted at the end to assign the correct final value to A_1. Actually, a single instruction could be saved: $S(T_{C_0})$ could assign A_1 its correct final value, the last line above omitted, and the penultimate one modified to read

$$A\sigma^t 1 := A1 + \rho(\sigma^t 1) - \rho(1),$$

but this may increase the bound on the constant in the assignment statements).

This completes the rather lengthy description of S'. By the foregoing remarks, S' is equivalent to S, and if S is a D-sequence, then S' is a D' - sequence where $D' \leq D \times ||S||$. In order to estimate the total length of S , we note:

(1) If $||C|| = 1$, then $||S(C)|| = 1$ if $C \neq \{0\}$,
$= 0$ if $C = \{0\}$.

(2) If $||C|| > 1$, then $||S(C)|| = ||C|| + 1$.

(3) $||S(T_C)|| = ||T_C||$.

Hence $||S'|| = \sum_C (||C|| + ||T_C|| + \sum_{||C|| > 1} 1 - 1$.

From Lemma 3.5, it follows that

$$||S'|| = k + \sum_{||C|| > 1} 1 \quad .$$

At this point, we can complete the proof of Theorem 3.2.

Proof of Theorem 3.2

The length of S' is maximised by taking an S which has as many cycles of length greater than 1 as possible. If $k = 2m+1$, we can take at most m distinct cycles, each of length 2, so that

$$k + \sum_{||C|| > 1} 1 \leq k + m = k + \left[\frac{k+1}{2}\right] - 1.$$

If $k = 2m$, we can take at most $(m-1)$ distinct cycles of length greater than 1, since by Lemma 3.6, there is at least one element which does not belong to a cycle. Hence

$$k + \sum_{||C|| > 1} 1 \leq k + (m-1) = k + \left[\frac{k+1}{2}\right] - 1,$$

and the proof is complete.

To clarify the concepts involved, we work an example. Let S be the sequence

$A_1 := A_2$	$A_7 := A_6+1$
$A_1 := A_2+1$	$A_7 := A_7+1$
$A_2 := A_4$	$A_6 := A_5+1$
$A_3 := A_3+1$	$A_5 := A_8$
$A_2 := A_3+1$	$A_8 := A_9+1$
$A_3 := A_1-1$	$A_9 := A_5$
$A_4 := A_5+1$	$A_5 := A_7-1$
$A_5 := A_7$	$A_5 := A_5-1$

We have $||S|| = 16$ and $k = 9$. The functions σ and ρ associated with S are given by the following table:

	0	1	2	3	4	5	6	7	8	9
σ	0	2	3	2	5	6	7	6	9	8
ρ	-	1	2	0	1	0	1	2	1	0

The cycles and tree sets of σ are:

$C_0 = \{2, 3\}$ $\quad T_0 = T_0(2) \cup T_0(3)$

$T_0(2) = \{1\}$

$T_0(3) = \text{null}$

$C_1 = \{0\}$ $\quad T_1 = \text{null}$

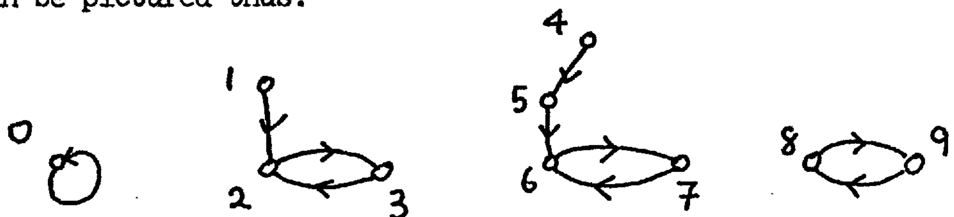
$C_2 = \{6,7\}$ $\quad T_2 = T_2(6) \cup T_2(7)$

$T_2(6) = \{4, 5\}$

$T_2(7) = \text{null}$

$C_3 = \{8,9\}$ $\quad T_3 = \text{null}.$

They can be pictured thus:



The reduced sequence S' is:

$A_4 := A_5 + 1$ $A_5 := A_6$	$k(6)$, the endordering of $T_2(6)$ being $\{4, 5\}$
$A_1 := A_6$ $A_6 := A_7 + 1$ $A_7 := A_1 + 2$	$S(C_2)$, using A_1 as work space.
$A_1 := A_8$ $A_8 := A_9 + 1$ $A_9 := A_1$	$S(C_3)$, ditto
$A_1 := A_2$	$K(2)$. Since $\|\|C_0\|\| \geq 2$, this is treated specially.
$A_2 := A_3 + 2$ $A_3 := A_1$ $A_1 := A_1 + 1$	$S(C_0)$, the special cycle

§2. Halving the number of assignments

In this section, we use the result of Section 1 to halve the number of assignments in a program P. It is convenient to first transform P into an equivalent program which stores only non-negative integers in its work registers. Although this preliminary transformation is not strictly necessary, it will simplify the details of a subsequent transformation. The second translation of Section 1 of Chapter 2 gives the details. It is to be noted that the translated program P does not use the test $A_j \geq 0$, and, more importantly, $a_{P'} = a_P$ (although $b_{P'}$ will be greater than b_P). From now on, each program will only store non-negative integers in its work registers.

Lemma 3.7 Given any program P, we can find an equivalent program R, which is on-line if P is, such that

$$a_R \leq a_P/2.$$

On the other hand, R makes use of more general instructions than P. In fact, if k is the maximum address used by P, then R is defined over the instruction set

$$A_i := A_j + d \qquad A_i := A_j - d$$

$$A_i := e \qquad A_i = e?$$

where $1 \leq i, j \leq k$, $0 \leq d \leq 3k$ and $0 \leq e < 3k$

Proof.

Informally, R works by simulating P in a step by step manner, except that it delays the execution of assignments. Each assignment executed by P is saved in the label structure of R until a sequence S of sufficient length (i.e. 3k) has been built up to enable an equivalent sequence S', defined over the extended instruction set and with $||S'|| \leq ||S||/2$, to be constructed. Only at this point does R execute S . Provided $||S|| = 3k$, the existence of such an S is guaranteed by Theorem 3.2. Moreover, if S is a 1-sequence (i.e. a sequence of I_0 assignments), then S' is a 3k-sequence. Since S never makes a register negative, we have $e \geqslant 0$.

Formally, the compiler Δ_1, for which $\Delta_1(P) = R$, produces labels (apart from start) of the form

$$(S, m),$$

where $m \in L$ and S is a sequence of assignments of length at most $K = 3k-1$.

To clarify what Δ_1 does, we first define the mapping δ_1 under which Δ_1 satisfies the conditions of the compiler theorem.

$$\delta_1 (\underline{\text{start}}, x, y, a_1, \ldots, a_k) = (\underline{\text{start}}, x, y, a_1, \ldots, a_k)$$

$$\delta_1 ((S, m), x, y, a_1, \ldots, a_k) =$$

$$(m, x, y, a_{\sigma(1)} + \rho(1), \ldots, a_{\sigma(k)} + \rho(k)),$$

where σ and ρ are the functions associated with the sequence S

and are defined in Section 1. Note that the definition of Δ_1 depends on k, and consequently is only applicable to programs which use registers $A_1, \ldots, A_k$.

The definition of Δ_1 is:

1. Δ_1 [start: → m] = {start: → (null, m)}.

2. (For work register assignments F):

$$\Delta_1 [m: F \to m'] = \bigcup_{||S|| < K} \{(S, m): \to (S;F,m')\} \cup$$

$$\bigcup_{||S|| = K} \{(S, m): S' \to (\text{null}, m')\},$$

where S' is the sequence equivalent to S; F, with

$$||S'|| \leq ||S;F||/2.$$

3. $$\Delta_1 [m: A_i = 0 \to m', m''] = \bigcup_{\rho_S(i) \leq 0} \{(S,m): A\sigma_S(i) = -\rho_S(i) \to (S, m'), (S, m'')\} \cup$$

$$\bigcup_{\rho_S(i) > 0} \{(S, m): \to (S, m'')\},$$

where the union is taken over all S with $||S|| \leq K$, such that the subsidiary condition holds.

4. $\Delta_1 [I] = \bigcup_{||S|| \leqslant K} I_S$, for all other (i.e. input and output) instructions I. (Here the current sequence S is just carried along in the labels).

The verification that Δ_1 under δ satisfies the conditions of the compiler theorem will not be given. The first four conditions are immediate, and the fifth follows from Theorem 3.2. As $||S|| \leqslant K = 3k-1$, we have $|\rho_S(i)| \leqslant K < 3k$ for all i, so that Δ_1 produces tests $A_i = e$ with $0 \leqslant e < 3k$. Hence the conclusions can be verified.

§3. Conversion to an I_0-program

Lemma 3.8 Given any program R satisfying the conditions of Lemma 3.7, we can find an equivalent I_0-program Q, which is on-line if R is, such that $a_Q \leqslant a_R$.

Proof.

Informally, Q works by simulating R in a step by step manner, except that Q stores only $\frac{1}{K}$ (where K = 3k) of the contents of the registers. The remainders of division are stored in the label structure of Q. Formally, the step by step compiler Δ_2, for which $\Delta_2(R) = Q$, produces labels (apart from start) of the form

$$(\alpha, m)$$

where m is arbitrary and $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ where $0 \leqslant \alpha_j < K$. The mapping δ_2 under which Δ_2 is a compiler is:

$$\delta_2(\underline{start}, x, y, a_1, \ldots, a_k) = (\underline{start}, x, y, a_1, \ldots, a_k)$$

$$\delta_2((\alpha,m), x, y, a_1, \ldots, a_k) = (m, x, y, Ka_1+\alpha_1, \ldots, Ka_k+\alpha_k).$$

The formal definition of Δ_2 is:

1. $\Delta_2 [\underline{start}: \to m] = \{start: \to (0, m)\}$

2. $\Delta_2 [m: A_i: = A_j+d \to m'] = \bigcup_\alpha \{(\alpha,m): A_i:=A_j+d(\alpha_j) \to (\alpha',m')\}$,
   where $d(\alpha_j) = \left[\frac{\alpha_j+d}{K}\right]$, $\alpha_r' = \alpha_r$ for $r \neq i$, and

   $\alpha_i' = [\alpha_j+d, K]$. Since $-K \leq d \leq K$ and $0 \leq \alpha_j < K$, we have $-1 \leq d(\alpha_j) \leq 1$, so the translated instruction is in I_0.

3. $\Delta_2 [m: A_i: = e \to m] = \bigcup_\alpha \{(\alpha,m): A_i: = 0 \to (\alpha',m')\}$

   where $\alpha_r' = \alpha_r$ for $r \neq i$ and $\alpha_i' = e$. Since $0 \leq e < K$, this instruction also translates correctly.

4. $\Delta_2 [m: A_i = e \to m', m''] = \bigcup_{\alpha_i=e} \{(\alpha,m): A_i = 0 \to (\alpha, m'), (\alpha,m'')\}$

   $$\cup \bigcup_{\alpha_i \neq e} \{(\alpha,m): \to (\alpha,m'')\}.$$

5. $\Delta_2[I] = \bigcup_\alpha I_\alpha$, for input and output instructions I.

The verification that Δ_2 under δ_2 satisfies the conditions of the compiler theorem is left to the conscientious reader. The first four conditions are immediate, and the fifth is straightforward but somewhat tedious. The conditions of the lemma can be verified from the definition of Δ_2. At this point, we have succeeded in converting an arbitrary program P into an equivalent program Q for which $a_Q \leq a_P/2$. Moreover, Q only stores non-negative integers in the work registers, and consequently makes no use of the test $A_i \geq 0$. In the next section, we use these facts to complete the proof of the speed up theorem.

§4. Concluding the proof.

In this section we prove:

Lemma 3.9 Given any program P which uses only the test $A_i = 0$, we can find an equivalent program Q, which is on-line if P is, such that

$$t_Q \leq 2a_P.$$

From this result, the speed up theorem follows easily. Suppose P is an arbitrary program, and the translations of Sections 2 and 3 are applied twice to P, giving a program Q for which $a_Q \leq a_P/4$. Lemma 3.9 then guarantees that a program R can be found so that

$$t_R \leq 2a_Q \leq a_P/2 \leq t_P/2,$$

and so speed up is assured.

<u>Proof of Lemma</u> 3.9

Informally, P is modified to ensure that, in every execution of P, there is at most one test $A_i = 0$ executed between any two executions of assignments to A_i. Because the input convention initialises each A_i to zero, we can further arrange that no test $A_i = 0$ occurs before the first assignment to A_i. If Q denotes the resulting program, then clearly $b_Q \leq a_Q = a_P$, from which the conclusion follows.

More formally, we describe the step by step compiler Δ_0, for which $\Delta_0(P) = Q$. This compiler is the composition of compilers $\Delta_0^{(i)}$, where each $\Delta_0^{(i)}$ reduces just the tests involving A_i. $\Delta_0^{(i)}$ produces instructions with labels (apart from <u>start</u>) of the form

$$(\alpha, m),$$

where $m \in L$ and α is either 0, 1 or 2. The mapping δ which guarantees that $\Delta_0^{(i)}$ is a compiler, according to the compiler theorem, is

$$\delta_0\ (\underline{start}, x, y, a_1, a_2, \ldots) = (\underline{start}, x, y, a_1, a_2 \ldots)$$

and $$\delta_0\ ((\alpha, m), x, y, a_1, a_2, \ldots) = (m, x, y, a_1, a_2, \ldots)$$

provided that, either $\alpha = 2$ or ($\alpha = 1$ and $a_i \neq 0$) or ($\alpha = 0$ and $a_i = 0$),

and δ_0 is undefined otherwise.

The definition of $\Delta_0^{(i)}$ is:

1. $\Delta_0^{(i)} [\underline{start} \to m] = \{\underline{start} \to (0,m)\}$,

2. $\Delta_0^{(i)} [m:F \to m'] = \bigcup_{\alpha} \{(\alpha,m): F \to (\alpha',m')\}$,

   where $\alpha' = 2$ if F is an assignment to A_i, and
   $\alpha' = \alpha$ otherwise,

3. $\Delta_0^{(i)} [m:A_i = 0 \to m', m''] = \{(2,m): A_i=0 \to (0,m'), (1, m'')$

$$(1, m): \to (1,m'')$$

$$(0, m): \to (0,m')\ \}$$

4. $\Delta_0^{(i)} [m: t \to m', m''] = \bigcup_{\alpha} \{(\alpha,m): t \to (\alpha, m'), (\alpha,m'')\}$

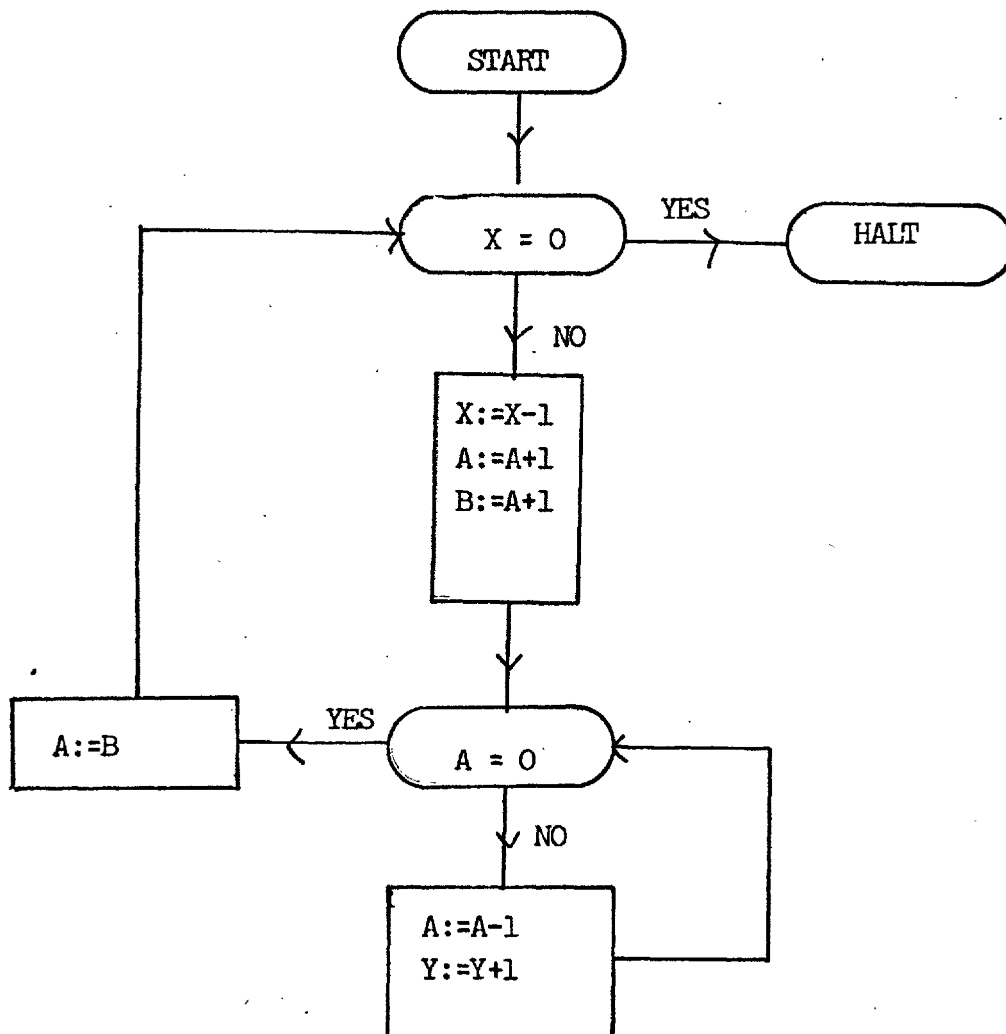
   for all other tests.

It is left to the reader to verify that $\Delta_0^{(i)}$ is indeed a compiler with the desired properties. It is clear from the last line of the definition, that $\Delta_0^{(i)}$ preserves on-line programs.

This completes the proof of the speed up theorem.

§5. A worked example

We speed up the program P by a factor of two, where P =



We can write P as the following set of labelled instructions:

1: $X=0\to 0,2$	5: $A=0\to 8,6$
2: $X:=X-1\to 3$	6: $A:=A-1 \to 7$
3: $A:=A+1 \to 4$	7: $Y:=Y+1 \to 5$
4: $B:=A+1 \to 5$	8: $A:=B \to 1$.

It is easy to verify that $f_P(x) = x^2$,

and $t_P(x) = 2x^2 + 4x$, for all x.

Moreover, P stores only non-negative integers in A and B. The first translation to be applied is Δ_1 (Section 2). Since $k = 2$, the special case of Theorem 3.2 shows that we need only build up sequences of length 4. To denote sequences, we use the code

n	null sequence
a	A:=A+1
b	B:=A+1
c	A:=A-1
d	A:=B

The first two instructions of $\Delta_1(P)$ are

$$(n,1): X = 0 \to (n,0), (n,2)$$
$$(n,2): X = 0 \to (n,3),$$

since no assignments have yet to be remembered.

The next two are

$$(n,3): \rightarrow (a,4)$$
$$(a,4): \rightarrow (ab,5),$$

and the next is

$$(ab,5): \rightarrow (ab,6),$$

since it can be determined from the remembered sequence ab, that A cannot be zero at this point. Continuing, we construct

$$(ab,6): \rightarrow (abc,7)$$
$$(abc,7): \quad Y:=Y+1 \rightarrow (abc,5)$$
$$(abc,5): A=0 \rightarrow (abc,8), (abc,6)$$

The effect of the sequence abc is to leave A unchanged so the test A=0 must be performed. Since the sequence abcc is equivalent to B:= A+2; A:=A-1, the next instructions are

$$(abc,6): B:=A+2 \rightarrow \ell_1$$
$$\ell_1: \quad A:=A-1 \rightarrow (n,7)$$
$$(n,7): \quad Y:=Y+1 \rightarrow (n,5).$$

where ℓ_1 is some new label. Continuing in this fashion, the rest of $\Delta_1(P)$ is found to be

(abc,8): B:=A+2 → ℓ_2

ℓ_2: A:=B → (n,1)

(n,5): A=0 → (n,8),(n,6)

(n,6): → (C,7)

(c,7): Y:=Y+1 → (c,5)

(c,5): A=1 → (c,8), (c,6)

(c,6): A:=A-2 → (n,7)

(c,8): → (cd,1)

(cd,1): X=0 → (cd,0),(cd,2)

(cd,2): X:=X-1 → (cd,3)

(cd,3): → (cda,4)

(cda,4): A:=B+1 → ℓ_3

ℓ_3: B:=B+2 → (n,5)

(n,8): → (d,1)

(d,1): X=0 → (d,0), (d,2)

(d,2): X:=X-1 → (d,3)

(d,3): → (da,4)

(da,4): → (dab,5)

(dab,5): → (dab,6)

(dab,6): A:=B → ℓ_4

ℓ_4: B:=A+1 → (n,7)

We have cheated a bit, in instruction (c,6), replacing the sequence cc by the single instruction A:=A-2, to save an instruction or two. The unconditional jumps can now be eliminated, leaving the following 21 instructions for R = Δ_1(P):

1: X=0 → 0,2

2: X:=X-1 → 3

3: Y:=Y+1 → 4

4: A=0 → 12,5

5: B:=A+2 → 6

6: A:=A-1 → 7

7: Y:=Y+1 → 8

8: A=0 → 14,9

9: Y:=Y+1 → 10

10: A=1 → 18,11

11: A:=A-2 → 7

12: B:=A+2 → 13

13: A:=B → 1

14: X=0 → 0,15

15: X:=X-1 → 16

16: A:=B → 17

17: B:=A+1 → 7

18: X=0 → 0,19

19: X:=X-1 → 20

20: A:=B+1 → 21

21: B:=B+2 → 8

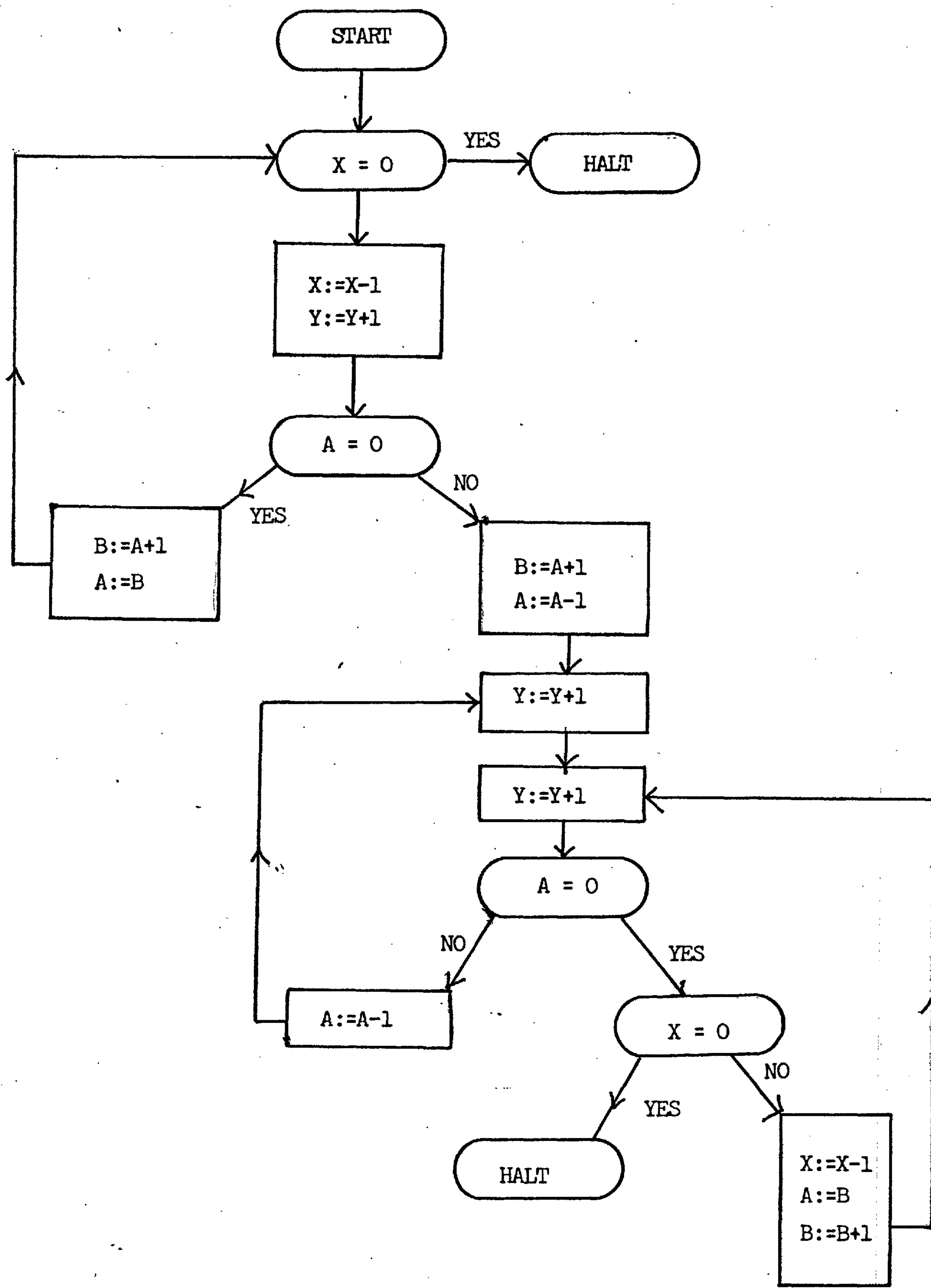
The next translation to be applied is Δ_2 (section 3). Here, we can take K=2, and obtain, for $Q = \Delta_2(R)$;

```
(00,1): X=0 → (00,0), (00,2)
(00,2): X:=X-1 → (00,3)
(00,3): Y:=Y+1 → (00,4)
(00,4): A=0 → (00,12),(00,5)
(00,5): B:=A+1 → (00,6)
(00,6): A:=A-1 → (10,7)
(10,7): Y:=Y+1 → (10,8)
(10,8): → (10,9)
(10,9): Y:=Y+1 → (10,10)
(10,10): A=0 → (10,18), (10,11)
(10,11): A:=A-1 → (10,7)
(00,12): B:=A+1 → (00,13)
(00,13): A:=B → (00,1)
(10,18): X=0 → (10,0), (10,19)
(10,19): X:=X-1 → (10,20)
(10,20): A:=B → (10,21)
(10,21): B:=B+1 → (10,8)
```

At this point, we know $f_Q = f_P$ and $a_Q \leq a_P/2$. Actually, Q has a running time given by

$$t_Q(x) = \begin{cases} x^2 + 2x - 1 & \text{for } x \geq 2 \\ 0 & \text{for } x = 0, \\ 3 & \text{for } x = 1, \end{cases}$$

so that $t_Q \leq t_P/2$ already, and so in this case Lemma 3.9 does not have to be invoked. The flowchart version of Q is


START
X = 0
YES
HALT
X:=X-1
Y:=Y+1
A = 0
NO
YES
B:=A+1
A:=B
B:=A+1
A:=A-1
Y:=Y+1
Y:=Y+1
A = 0
NO
YES
A:=A-1
X = 0
NO
YES
HALT
X:=X-1
A:=B
B:=B+1

§6. Further results and discussion

It immediately follows, by using Theorem 3.1 repeatedly, that given an arbitrary integer c and program P, we can find a program Q, equivalent to P and on-line if P is, such that $t_Q \leq t_P/c$. Hence I_0 - programs can be speeded up by an arbitrary linear factor.

It is worth emphasising that the speed-up property for an instruction set I asserts that given any I - program P, an I - program Q can be found such that

(i) Q is on-line if P is,

(ii) Q is equivalent to P,

(iii) $t_Q \leq t_P/2$.

Condition (i) is important in that, only by assuming it necessary, can we prove that if I_0 is augmented with addition and subtraction, then the resulting instruction set does not possess the speed up property (Chapter 5). Similarly, if I_0 is augmented with instructions to address work registers indirectly. We do not know whether condition (i) can be dropped. That is, if condition (i) is omitted and the weakened version called the weak speed up property, then we do not know whether, for an arbitrary instruction set I, weak speed up for I implies speed up for I. A similar ignorance exists for exact speed up. An instruction set I has the exact speed up property if, given an integer c > 0 and an I-program P, an I - program Q,

equivalent to P, can be found such that

(i) Q is on-line if P is,

and (ii) $t_Q = \left[\frac{t_P}{c}\right]$.

Does the speed up property for I imply exact speed up? We can show that I_0 possesses exact speed up. This can probably be proved by modifying the translations, but we give an alternative proof, based on the fact (which we do not formally verify), that the given translations actually prove the following stronger result:

Corollary 3.10 Given an I_0 - program P, we can find an I_0 - program Q, which is equivalent to P and on-line if P is, such that

$$\Delta t_Q \leq (\Delta t_P)/2,$$

(where $\Delta f(x) = f(x) - f(x-1)$ if $x > 0$, and $\Delta f(0) = f(0)$).

This fact is used in case (ii) of the following theorem. The proof technique, which also appears again in Chapter 4, is similar to that used in Fischer [15] on an analogous result for Turing machines.

Theorem 3.11 Given any program P and integer c, we can find a program Q, equivalent to P and on-line if P is, such that

$$t_Q = \left[\frac{t_P}{c}\right]$$

Proof. Let $t = \left[\frac{t_P}{c}\right]$. There are two cases to be considered depending on whether P is on-line or not.

(i) P is not on-line. By suppressing the output instructions of P, inserting appropriate new ones after the work register instructions and using speed-up, we can find a program R such that

$$f_R = t \quad \text{and} \quad t_R \leq t_p/6c.$$

Also by speed up a program S can be found such that

$$f_S = f_P \quad \text{and} \quad t_S \leq t_p/6c.$$

Since $\frac{t_P}{3c} \leq \frac{1}{3}\left[\frac{t_P}{c}\right]$, it follows that

$$t_R + t_S \leq \left[\frac{t}{3}\right], \tag{1}$$

with equality only when the right hand side is zero. The final program Q is defined from modified versions of S and R, described as follows:

(a) program R′ is formed from R by replacing the output instructions of R by instructions of the form A:=A+1, where A is a register not appearing in S′ or R′, so that, through these instructions, R′ computes $\left[\frac{t}{3}\right]$ in A. In addition, R remembers in its label structure whether an instruction A:=A+1 is ever executed, and arranges to waste [t, 3] steps by obeying some dummy instruction. Next, each original work register instruction of R has inserted after it,

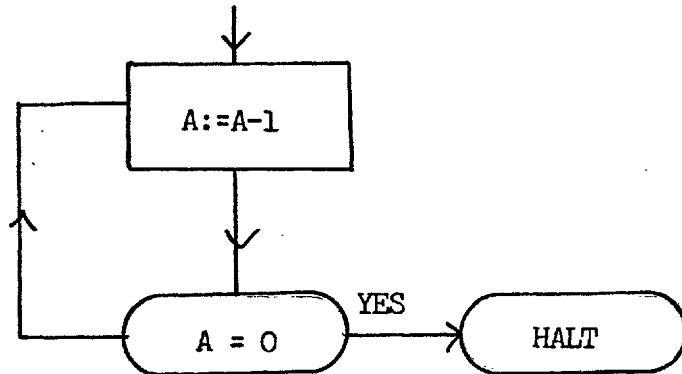
an instruction A:=A-1. Thus R' computes $\left[\frac{t}{3}\right] - t_R$ in register A, and

$$f_{R'} = 0 \text{ and } t_{R'} = 2t_R + \left[\frac{t}{3}\right] + [t,3]$$

(b) Program S' is formed from S by inserting an instruction A:=A-1 after each work register instruction of S. Thus

$$f_{S'} = f_S \text{ and } t_{S'} = 2t_S.$$

The final program Q is formed by following the program $R' + S'$(given by the addition theorem) by the code



which is only executed when R indicates that at least one A:=A+1 instruction has been obeyed. Since the effect of $R' + S'$ is to leave $\left[\frac{t}{3}\right] - t_R - t_S$ in register A, the running time of Q is given by

$$t_Q = 2t_R + 2t_S + \left[\frac{t}{3}\right] + [t,3] + 2\left(\left[\frac{t}{3}\right] - t_R - t_S\right),$$

by inequality (1). Thus $t_Q = t$ and as $f_Q = f_{S'} + f_{R'} = f_P$, the theorem is proved in this case. Note that the construction never yields an on-line program Q, so that a slightly different construction has to be given in the case that P is on-line.

(ii) P is on-line. Using the stronger conclusion of Corollary 3.10 we can find, in a similar fashion to case (i), on-line programs R and S such that

$$f_R = t \quad \text{and} \quad f_S = f_P$$

and
$$\Delta t_R + \Delta t_S \leq \left[\frac{\Delta t}{3}\right] ,$$

with equality holding only when the right hand side is zero. The programs R' and S' are formed as in case (i), except that R arranges to waste $[\Delta t,3]$, steps before the execution of each input test storing only $\left[\frac{\Delta t}{3}\right]$ Q is also the same as in case (i), except that the code which reduces A to zero is inserted before each input test of $R' + S'$. Program Q is therefore on-line, and since A is reduced to zero before each input test, we have

$$\Delta t_Q = 2\Delta t_R + 2\Delta t_P + \left[\frac{\Delta t}{3}\right] + [\Delta t,3] + 2\left(\left[\frac{\Delta t}{3}\right] - \Delta t_R - \Delta t_P\right).$$

Thus $t_Q = t$ and as Q is equivalent to P and on-line, the theorem is proved.

In the next chapter, we show that linear speed up is the best that can be obtained.

CHAPTER FOUR

THE HONEST FUNCTIONS

Throughout this chapter, except where otherwise stated, the term program denotes a program defined over some fixed, but arbitrary instruction set I; thus all concepts are defined relative to I. The object of the chapter is to investigate the class of honest functions. These functions have the following definition:

(1) A program P is said to be honest if $t_P \trianglelefteq f_P$.

(2) A function f is honest if there is an honest program which computes f.

(3) A function f is superhonest if there is an on-line honest program which computes f.

The main reason behind the introduction of the honest functions, is that it is exactly the concept we need to show that, no matter what instruction set I is specified, linear speed-up is the best possible (Section 1). However, having introduced this notion, a more fundamental fact emerges (Section 2), which we now outline.

For any function t, let $T[t]$ denote that class of functions which can be computed by programs with a running time bounded by ct, for some constant c. More briefly,

$$T[t] = \{f_P : t_P \trianglelefteq t\}.$$

Further, let R, the class of real-time computable functions, be the set

$$R = \{f_p : t_p(x) \trianglelefteq x\}.$$

The main theorem of Section 2 then says: there is a simple 2-place functional F, such that if t is a strictly increasing superhonest function, then

$$f \in T[t] \text{ if and only if } F(f,t) \in R .$$

The import of this result is that to a large extent (more precisely, to the extent that the superhonest functions form a sufficiently embracing class of functions), the study of time limited computation on register machines can be reduced to the study of real-time computation. In order to show that a given function f is or is not computable within time t, it is sufficient to show that F(f,t) is or is not real-time computable. This remark motivates Chapter 5, wherein methods, more subtle than crude size arguments, are developed for showing that functions are not real-time computable.

Further sections clarify the relationship between the honest and superhonest functions and the running times of programs (Section 3), and explore some of the closure properties of these classes (Sections 4 and 5).

§1. Honesty and linear speed-up

In this section, we show that no instruction set I can have a better than linear speed-up property.

Suppose I does possess such a speed-up. This means, in particular, that given any program P, one can find an equivalent program Q for which

$$t_Q \lhd t_P.$$

Let P be an honest program which computes a non-ultimately linear function (the existence of such a program is guaranteed by the fact that I always contains I), and let Q be the speeded up version. Since $f_Q = f_P$, we have $x \unlhd t_Q(x)$ by the minimal growth rate theorem. But now,

$$\begin{aligned} f_Q &\unlhd t_Q && \text{by the size theorem,} \\ &\lhd t_P && \text{by hypothesis,} \\ &\unlhd f_P && \text{since } P \text{ is honest,} \\ &= f_Q && \text{since } Q \text{ is equivalent to } P. \end{aligned}$$

Thus $f_Q \lhd f_Q$, which is impossible; hence Q cannot exist.

The same idea can be stated differently.

<u>Theorem</u> 4.1 Suppose f is an unbounded honest function, and g is arbitrary. Then

$$T[f] \subseteq T[g] \quad \text{if and only if} \quad f \unlhd g.$$

The proof makes use of the following lemma, which is also used in subsequent sections.

Lemma 4.2 If f is honest and $f(x) \lhd x$, then f is ultimately periodic.

Proof. By definition, there is a program P computing f for which $t_P \trianglelefteq f$. If $f(x) \lhd x$, then $t_P(x) \lhd x$, and so f is ultimately linear, by the minimal growth rate theorem. But since $f(x) \lhd x$, f must in fact be ultimately periodic.

Proof of Theorem 4.1

The fact that $f \trianglelefteq g$ implies $T[f] \subseteq T[g]$ is obvious by the transitive property of $\trianglelefteq$. For necessity, suppose $g \lhd f$. We show that there is some function $h \in T[f] - T[g]$.

Case 1. $x \trianglelefteq g(x)$. Clearly, $f \in T[f]$ since f is honest. If $f \in T[g]$, then $f \trianglelefteq g$ by the size theorem and the fact that $x \trianglelefteq g(x)$. This contradicts the assumption that $g \lhd f$.

Case 2. $g(x) \lhd x$. In this case, $T[g]$ is just the class of ultimately linear functions. The function a, where

$$a(x) = (x \text{ even} \rightarrow x, 0)$$

is not therefore in $T[g]$. The function a can be computed by a I - program P (since I contains I_0), for which

$$t_P(x) \trianglelefteq x.$$

Now, since f is unbounded, it is not ultimately periodic, and so $x \trianglelefteq f(x)$ by Lemma 4.1. Hence $a \in T[f]$.

Corollary 4.3 If f and g are unbounded honest functions, then

$$T[f] = T[g] \text{ if and only if } f \sim g.$$

Proof. Immediate from Theorem 4.1.

§2. The real-time characterisation

In order to prove the main result of this section, we need to consider certain properties of the output function θ_P of an on-line program P. It will be recalled from Section 2.7 that

$\theta_P(x)$ = total number of work register instructions executed by P, on any sufficiently large input, prior to the execution of the (x+1)st output instruction,

= undefined, if no such output occurs.

If P is on-line and f_P is unbounded, then θ_P is a well defined total function. (If P is not on-line, then θ_P is not well defined, since the definition depends on the particular input chosen).

Theorem 4.4 If f is unbounded and superhonest, then there is an on-line program P computing f, for which $\theta_P(x) \trianglelefteq x$.

Proof. There are two cases to be considered. First, if f is ultimately linear, then there is an on-line program P computing f for which $\theta_P(x) = 0$ for all x, and in this case the theorem follows at once. Accordingly, we suppose for the rest of the proof,

that f is not ultimately linear.

Since f is superhonest, we can assume, by the almost everywhere theorem and the composition theorem, that there is an on-line program Q computing cf, for which

$$t_Q(x) \leqslant cf(x) \quad \text{for all} \quad x,$$

where c is some integer greater than zero.

Below, we construct a program R, for which

$$f_R(x) = cf(x) \quad \text{and} \quad \theta_R(x) \trianglelefteq x.$$

R can be converted into the final program P by permitting only one out of every c outputs to be given. We have

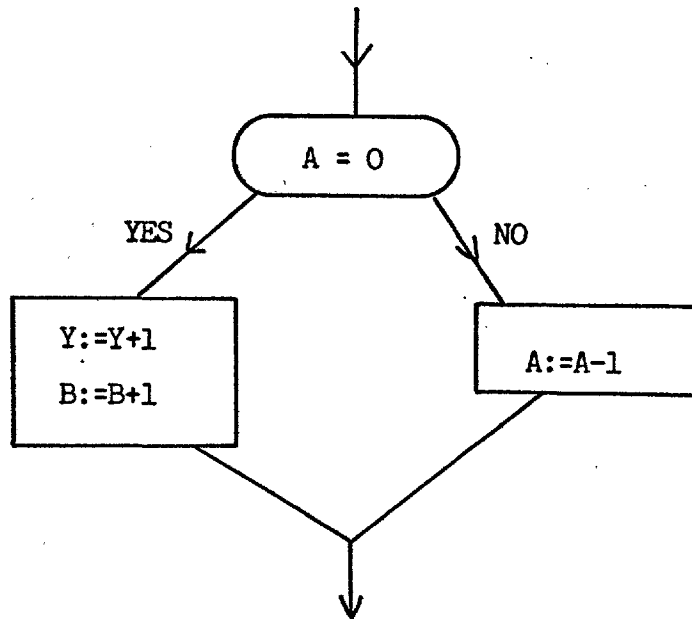
$$\theta_P(x) \quad = \quad \theta_R(cx) \trianglelefteq x,$$

and $$f_P(x) \quad = \quad f(x),$$

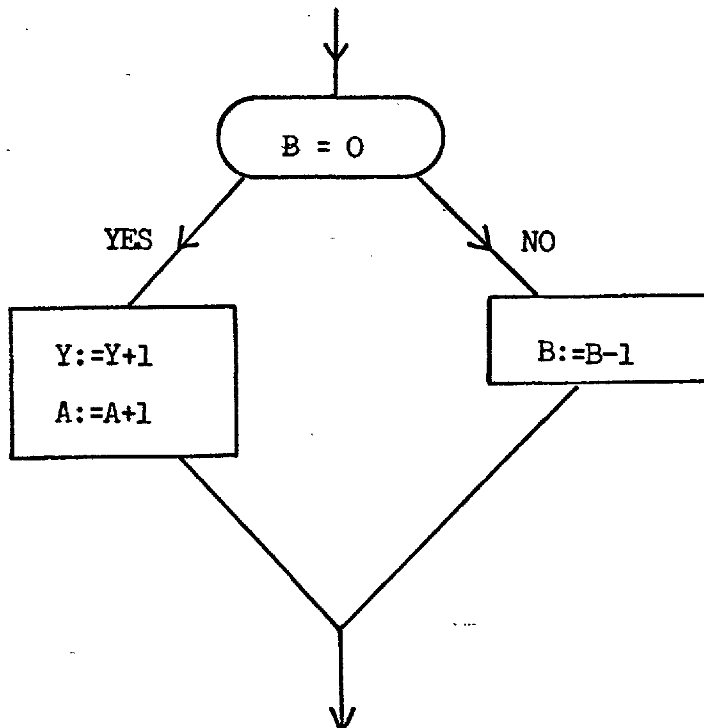
giving the desired result.

The program R is a modified version of Q, in which (supposing A and B are two registers not appearing in Q):

(1) after each work register instruction of Q is inserted the code:



(ii) each (original) output instruction of Q is replaced by the sequence



First, it is immediate that R is an on-line program if Q is. To show that R has the desired properties, we compare the computation sequences of R and Q on some sufficiently large input, and examine the contents of the registers A, B and Y at those points in the computation sequence of R which correspond to points in the computation sequence of Q immediately prior to an output or work register instruction. We refer to the nth such point as point n, allowing point 0 to designate the very start of the computations.

Suppose that at point n, t_n work register instructions of R have been executed, and a_n, b_n, y_n denote the contents of A, B, and Y. At the corresponding point of Q, suppose t work register instructions have been executed, and y is the contents of Y.

It is immediate that

$$t_n = 3t + 2y. \tag{1}$$

Moreover, we claim:

(i) if $y \geq t$, then $a_n = y-t$, $b_n = 0$, $y_n = y$

(ii) if $y < t$, then $a_n = 0$, $b_n = t-y$, $y_n = t$.

The proof is by induction. (i) and (ii) trivially hold for n=0, since all registers are zero at the start. Suppose (i) and (ii) hold at point n. Between point n and point (n+1) the computation sequence of Q contains exactly one output or work register instruction. These two possibilities are examined separately.

Let y' and t' be y and t for point $(n+1)$.

(a) an output instruction. Here, $y' = y+1$ and $t' = t$.

(i) if $y \geq t$, then $y' > t'$. By inductive assumption, $b_n = 0$, and so by definition of R:

$$a_{n+1} = y - t + 1 = y' - t'$$

$$b_{n+1} = 0$$

$$y_{n+1} = y + 1 = y'.$$

(ii) if $y < t$, then $y' \leq t'$. By assumption $b_n \neq 0$ and so:

$$a_{n+1} = 0$$

$$b_{n+1} = t - y - 1 = t' - y'$$

$$y_{n+1} = t = t'.$$

(b) a work-register instruction. Here $y' = y$ and $t' = t+1$.

(i) if $y > t$, then $y' \geq t'$. Since $a_n \neq 0$:

$$a_{n+1} = y - t - 1 = y' - t'$$

$$b_{n+1} = 0$$

$$y_{n+1} = y = y'$$

(ii) if $y \leq t$, then $y' < t'$. Since $a_n = 0$:

$$a_{n+1} = 0$$

$$b_{n+1} = b - y + 1 = t' - y'$$

$$y_{n+1} = t + 1 = t'.$$

The induction step is complete. It immediately follows that

$$f_R(x) = \max (t_Q(x), f_Q(x)) = cf(x)$$

for all x.

By an argument similar to that used in the size theorem, we also have

$$y \leq k(1+t), \tag{2}$$

since f_Q is not ultimately linear. Here, k is the number of distinct output instructions in Q. From (1) and (2) we have

$$t_n \leq (3 + 2k)t + 2k,$$

whence

$$y_n \geq \frac{t_n-2k}{3+2k} ,$$

since $y_n = \max(y,t)$. By definition of θ_R, we have

$$\theta_R(y_n-1) \leq t_n,$$

whence
$$\theta_R \left(\frac{t_n-2k}{3+2k} - 1\right) \leq t_n , \tag{3}$$

since θ_R is monotonic.

Finally, let x be arbitrary. By choosing a sufficiently large input, we can find an n, such that

$$(2k+3)x \leq t_n - (4k+3) < (2k+3)x + 3,$$

since $t_n < t_{n+1} \leqslant t_n+3$. Hence, using (3),

$$\theta_R(x) \leqslant t_n \trianglelefteq x,$$

and the theorem is proved.

We can now prove

<u>Theorem</u> 4.5 An unbounded function f is superhonest if and only if f^* is on-line real time computable.

<u>Proof</u>.

Suppose f is superhonest, so that by Theorem 4.4, there is an on-line program P computing f, for which $\theta_P(x) \trianglelefteq x$ for all x. By the inversion theorem of Section 2.7, P^* computes f^* with a running time given by

$$t_{P*}(x) = \theta_P(x) \trianglelefteq x.$$

Thus f^* is on-line real time computable. Conversely, suppose P is an on-line program computing f^* and

$$t_P(x) \trianglelefteq x.$$

Using the inversion theorem again, P^* computes $f^{**} = f$, and

$$t_{P*}(x) \leqslant t_P \cdot f_{P*}(x) \trianglelefteq f_P^*(x) = f(x),$$

whence f is superhonest.

The main characterisation can now be stated.

Theorem 4.6 Let t be a strictly increasing superhonest function, and p be the function $p(x) = x-1$. Then

(i) $f \in T[t]$ if and only if $f \cdot p \cdot t^* \in R$,

(ii) $f \in T_{ON}[t]$ if and only if $f \cdot p \cdot t^* \in R_{ON}$.

(where $T_{ON}[t] = \{f_P : t_P \trianglelefteq t, P \text{ on-line}\}$ etc.)

Proof. (a) necessity. Suppose P is a program for which

$$f_P = f \quad \text{and} \quad t_P \trianglelefteq t. \tag{1}$$

Since p is ultimately linear, we can find an on-line Q for which

$$f_Q = p \quad \text{and} \quad t_Q = 0. \tag{2}$$

Since t is superhonest and unbounded, we can find an on-line program R, by Theorem 4.5, such that

$$f_R = t^* \quad \text{and} \quad t_R(x) \trianglelefteq x. \tag{3}$$

By the composition theorem of Section 2.6, the program $P \cdot Q \cdot R$ which is on-line if P is, computes $f \cdot p \cdot t^*$ with a running time

$$t_{P \cdot Q \cdot R}(x) = t_R(x) + t_Q \cdot f_R(x) + t_P \cdot f_Q \cdot f_R(x)$$

$$\trianglelefteq x + t \cdot p \cdot t^*(x),$$

using (1), (2) and (3). By definition of t^*, we have

$t(t^*(x)-1) \leq x$ for all $x \geq t(0)$, and so

$$t_{P \cdot Q \cdot R}(x) \trianglelefteq x.$$

(b) Sufficiency

Let P be a program such that

$$f_P = f \cdot p \cdot t^* \quad \text{and} \quad t_P(x) \trianglelefteq x. \tag{4}$$

Since t is superhonest, we can find an on-line Q such that

$$f_Q = t \quad \text{and} \quad t_Q \trianglelefteq t. \tag{5}$$

By composition, the program $P \cdot Q$ which is on-line if P is, computes $f \cdot p \cdot t^* \cdot t$ with a running time

$$t_{P \cdot Q} = t_Q + t_P \cdot f_Q \trianglelefteq t,$$

using (4) and (5). However, if t is strictly increasing,

$$t^*(t(x)) = x+1,$$

i.e. $p \cdot t^* \cdot t(x) = x$. Thus $P \cdot Q$ computes f, and the theorem is proved.

It is worth noting that the statement of Theorem 4.6 cannot be simplified to read:

$$f \in T[t] \text{ if and only if } f \cdot t^* \in R,$$

as this assertion is false. Consider $f(x) = t(x) = x!$ It can be shown (Example 4.17) that the factorial function is I_0-superhonest so that $f \in T[t]$. However, $f \cdot f^*(x!) = (x+1)!$, which shows that $f \cdot f^*(x) \not\leq x$, so by the size theorem, $f \cdot f^*$ cannot be real time computable.

Corollary 4.7 Suppose t is strictly increasing and superhonest. Then

$f \in T_{ON}[t]$ if and only if $t.f^*$ is superhonest.

Proof. Immediate, from Theorems 4.5 and 4.6, since by Lemma 2.10

$$(t \cdot f^*)^* = f^{**} \cdot p \cdot t^* = f \cdot p \cdot t^*.$$

§3. Honesty and running time

In this section, we examine the relationship between the honest functions and the running times of programs. It turns out that for I_0 - programs these classes are identical. The following theorem is the register machine analogue to the main result of Fischer [15].

<u>Theorem</u> 4.8 Let f be an arbitrary honest function and P a program for which $t_P \trianglelefteq f$. Then there is a program Q equivalent to P for which $t_Q = cf$ for some constant c.

<u>Proof</u>. The proof is very similar to that of Theorem 3.11. Suppose R is an honest program which computes f, i.e. $t_R \trianglelefteq f$. In fact, using the almost everywhere theorem, we can assume that

$$t_R(x) + t_P(x) \leqslant kf(x) \quad \text{for all } x,$$

for some positive integer k, with equality holding only when $f(x) = 0$. We modify these programs as follows:

(i) program R' is formed from R by replacing every output instruction of R, by a sequence of k instructions of the form A:=A+1, where A is some register not appearing in P and R.

Moreover, R' remembers in its label structure whether such a sequence is ever executed. In addition, an instruction A:= A-1 is inserted after each original work register instruction of R. Thus R' computes $kf - t_R$ in register A, and has

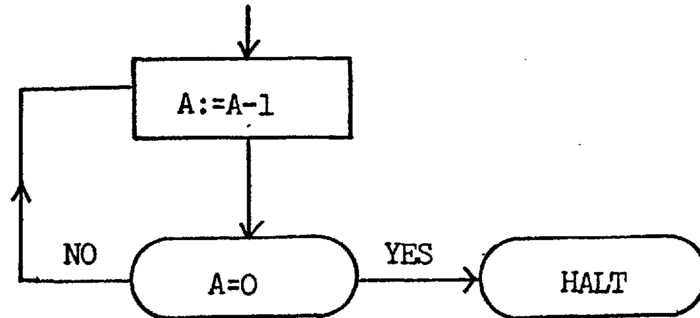
$$f_{R'} = 0 \quad \text{and} \quad t_{R'} = 2t_R + kf.$$

(ii) Program P' is formed from P by inserting an instruction A:=A-1 after each work register instruction.

Thus

$$f_{P'} = f_P \quad \text{and} \quad t_{P'} = 2t_P.$$

The final program Q is formed by following the program $P' + R'$ by the code



which is only executed when R' indicates that at least one sequence of A:= A+1 instructions has been executed. Since the effect of $P' + R'$ is to leave $kf - t_R - t_P$ in register A, the running time of Q is given by

$$t_Q = 2t_R + kf + 2t_P + 2(kf-t_R-t_P),$$

since, by supposition, $kf \geq t_R + t_P$. Thus $t_Q = 3kf$ and $f_Q = f_P$, proving the theorem.

<u>Corollary</u> 4.9 If f is I_0-honest, then f is the running time of some I_0-program.

<u>Proof.</u> For I_0 -programs, we can speed up the Q of Theorem 4.8 by exactly $\frac{1}{c}$.

<u>Lemma</u> 4.10 The running time of every (on-line) program is honest (superhonest).

<u>Proof</u>. Let P be an arbitrary program. Delete all output instructions from P and insert new ones after each work register instruction. The resulting program Q has

$$f_Q = t_P \text{ and } t_Q = t_P,$$

showing that t_P is honest. If P is on-line, Q will be also, and so t_P is superhonest.

<u>Corollary</u> 4.11 A function is I_0 -honest if and only if it is the running time of some I_0 program.

The proof of Theorem 4.8 does not carry over in the case of on-line programs and superhonest functions.

<u>Lemma</u> 4.12 The function h, where $h(x) = [\sqrt{x}]^2$, is superhonest, but for no integer c is ch the running time of any on-line program.

<u>Proof</u>.

We have $h(x) = S(S^*(x)-1)$, where $S(x) = x^2$. Since S is superhonest (Example 4.15), h is on-line real time computable. Hence h is superhonest, as $x \trianglelefteq h(x)$.

Suppose that P is an on-line program with $t_P = ch$, for

some integer c. Suppose P has k distinct input tests, and consider the computation of P with an input of the form x^2+2x, where $2x > k$. Since P is on-line and

$$t_P(x^2+2x) - t_P(x) = ch(x^2+2x) - ch(x^2) = 0,$$

the last 2x input tests of this computation were executed on the same contents of the work registers, and so some input test was obeyed twice with the same work register configuration. This means that for any input $y \geqslant x^2+2x$, $t_P(y) = t_P(x^2)$ and so h is bounded, which is clearly false.

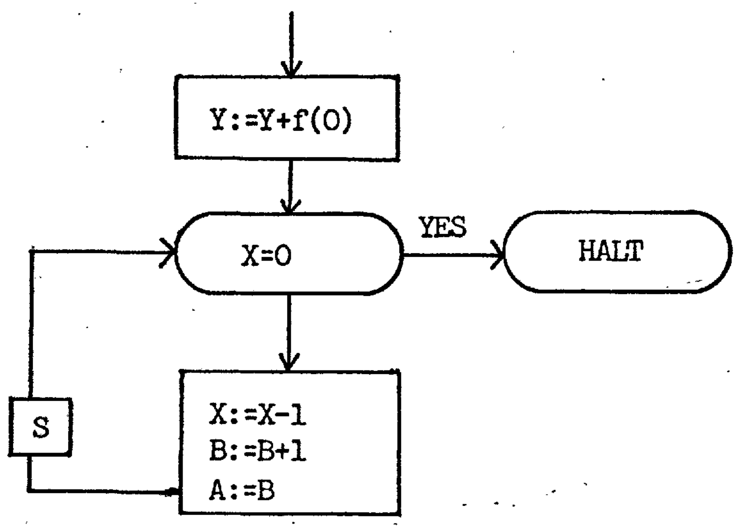
For on-line programs, the best result is:

<u>Theorem</u> 4.13 Suppose Δf is honest and P is an on-line program for which $\Delta t_P \trianglelefteq \Delta f$. Then there is an on-line program Q, equivalent to P, such that $t_Q = cf$ for some constant c. (For I_0 -programs we can take c=1).

<u>Proof</u>. Once we prove that if Δf is honest, then there is an on-line program R computing f such that $\Delta t_R \trianglelefteq \Delta f$, the rest of the proof follows along the same lines as Theorem 3.11. There are two cases to be considered

(1) $\Delta f(x) \triangleleft x$. In this case Δf must be ultimately periodic by Lemma 4.2, and so f is (monotone) ultimately linear. Here, we can take $\Delta t_R = 0$.

(2) $x \trianglelefteq \Delta f(x)$. Suppose S is an honest program which computes Δf. Modify S to read its input from a new register A, and let R be the program



where B is another register not appearing in Q. R is an on-line program which computes f, and

$$\Delta t_R(x) = 2 + kx + t_S(x) \quad \text{for some } k$$

$$\trianglelefteq \Delta f(x),$$

since S is honest and $x \trianglelefteq \Delta f(x)$.

<u>Corollary</u> 4.14 If Δf is honest, then f is superhonest.

<u>Example</u> 4.15 Let $S(x) = x^2$. Since $\Delta S(x) = (x=0 \rightarrow 0, 2x-1)$, ΔS is ultimately linear and so honest. Thus S is superhonest.

Restated, the last corollary says that if f is honest, then Σf is superhonest. We end this section by showing that if f is honest, then Πf is superhonest, where

$$(\Pi f)(x) = f(0) \times f(1) \times \ldots \times f(x).$$

<u>Theorem</u> 4.16 If f is honest and $f > 0$, then Πf is superhonest.

<u>Proof</u>. There are two cases to be considered.

(i) $f(x) \triangleleft x$. In this case f is ultimately periodic.

Suppose $f(x+c) = f(x)$ for $x \geq x_0$.

It follows that for $x \geq x_0$,

$$f(x) = f(x_0 + [x - x_0, c]),$$

whence

$$\prod_{y=x+1}^{x+c} f(y) = \prod_{y=x_0+1}^{x_0+c} f(y) = d \quad \text{say.}$$

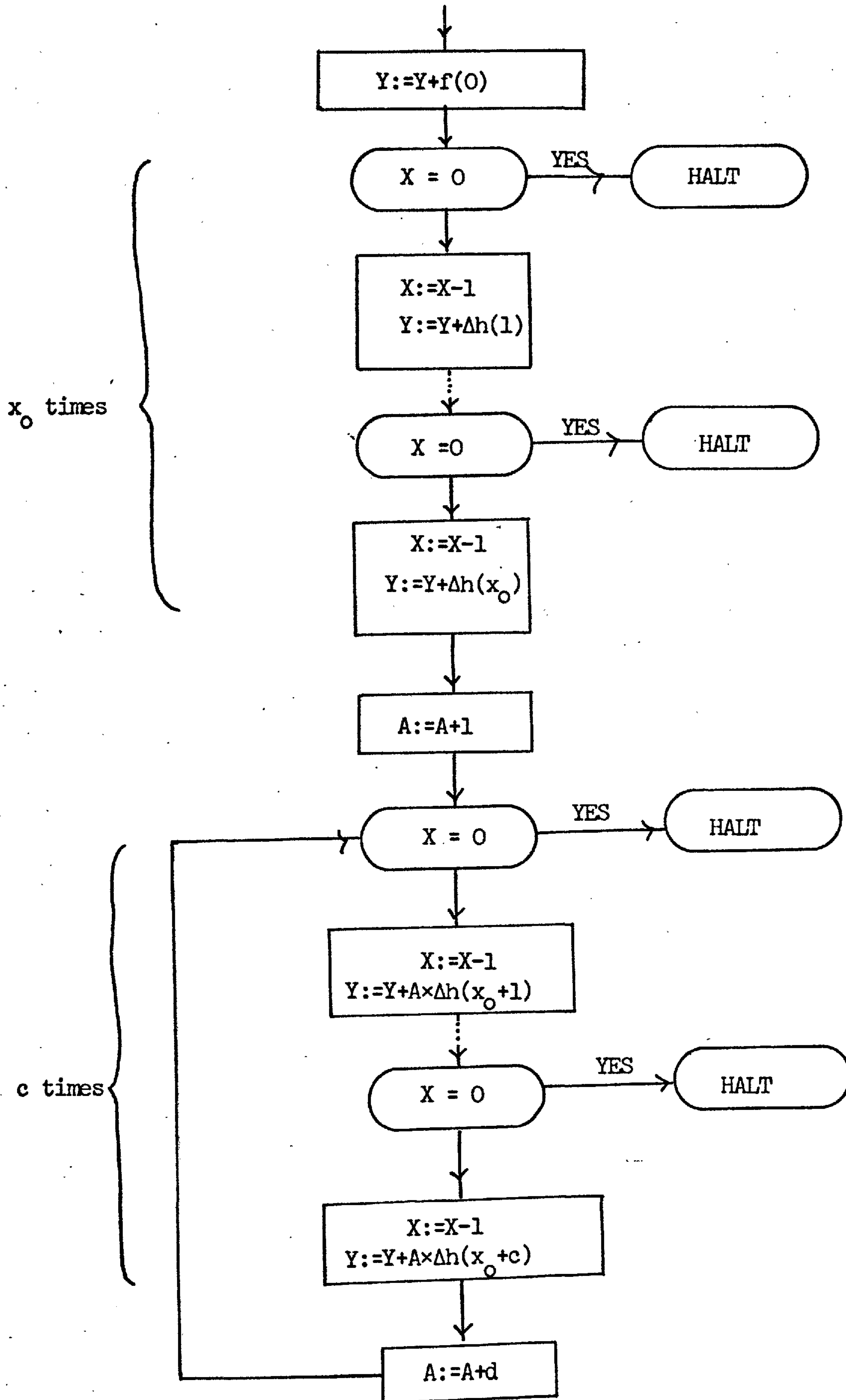
Thus, if $\Pi f = h$, then

$$h(x+c) = dh(x) \quad \text{for} \quad x \geq x_0 ,$$

and so

$$\Delta h(x+c+1) = d\Delta h(x+1) \quad \text{for} \quad x \geq x_0.$$

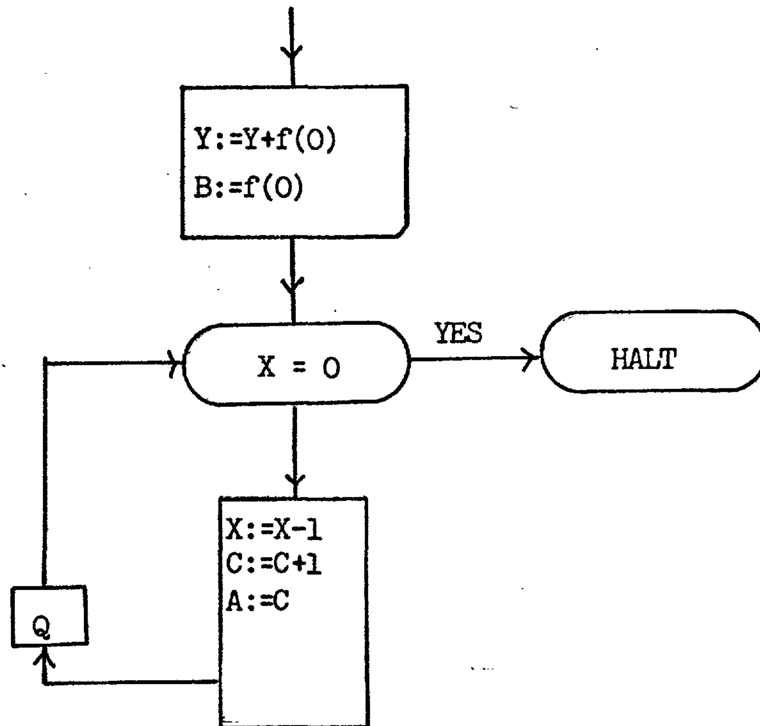
It follows from this that the following on-line program computes h:



Y:=Y+f(0)
X = 0
YES
HALT
X:=X-1
Y:=Y+Δh(1)
x_o times
X =0
YES
HALT
X:=X-1
Y:=Y+Δh(x_o)
A:=A+1
X = 0
YES
HALT
X:=X-1
Y:=Y+A×Δh(x_o+1)
c times
X = 0
YES
HALT
X:=X-1
Y:=Y+A×Δh(x_o+c)
A:=A+d

When the instructions involving multiplication of A by a constant are replaced by I_0 - subprograms, it is possible to verify that the resulting program P has a running time $t_P \trianglelefteq h$, whence h is superhonest. The details are omitted.

(ii) $x \trianglelefteq f(x)$. In this case, suppose P is a program which computes f honestly. Q is a modified version of P in which the input is read from a new register A. In addition, for each input x, Q computes f(x) into some work register, and outputs the quantity b(f(x)-1), where b is the initial contents of another register B. Finally, before halting, Q stores the value bf(x) in B. Now let R be the following program:



where C is yet another register not appearing in Q. It is easy to verify by induction that the contents of B immediately prior to the execution of the $(x+1)$st input test of R is just $h(x) = (\Pi f)$ and so

$$f_R(x) = f(0) + f(0) \times (f(1) - 1) + \dots$$

$$+ f(0) \times f(1) \times \dots \times f(x-1) \times (f(x) - 1)$$

$$= h(x).$$

Moreover, the running time of R satisfies

$$t_R(x) \leqslant \sum_{y=o}^{x} (y + h(y)) \leqslant (\Sigma h)(x),$$

since $y \leqslant f(y) \leqslant h(y)$. It remains to show that $\Sigma h \leqslant h$.

We have

$$\frac{(\Sigma h)(x)}{h(x)} = 1 + \frac{1}{f(x)} + \frac{1}{f(x) \times f(x-1)} + \dots + \frac{1}{f(x) \times \dots \times f(1)}$$

$$\leq 1 + \frac{x}{f(x)} .$$

Since $x \leqslant f(x)$, the conclusion follows.

<u>Example</u> 4.17 The factorial function is superhonest.

Clearly $x! = (\Pi f)(x)$, where f is the ultimately linear function

$$f(x) = (x=0 \rightarrow 1, x).$$

§4. Closure under composition

The honest functions are not closed under unrestricted composition as we can easily show.

Let f be the function

$$f(x) = 2x+1 \text{ if } x \text{ is square,}$$
$$= 2x \quad \text{otherwise.}$$

We have

$$f(x) = 2x + \Delta sq^*(x),$$

where $sq(x) = x^2$. Since sq is superhonest, sq* and hence Δsq* are real time computable. It follows from the addition theorem that f is real time computable and hence honest, since $x \leqslant f(x)$. Let g be the ultimately linear, and so honest, function

$$g(x) = (x \text{ even} \rightarrow 1, 0).$$

The function g·f, whose values are

$$(g \cdot f)(x) = (x \text{ square} \rightarrow 0, 1)$$

is not honest by Lemma 4.2

Theorem 4.17 (i). If f and g are honest and f is unbounded then f·g is honest.

(ii). If f and g are superhonest, then so is f·g.

(iii). If f is honest and g is ultimately linear, then f·g is honest.

Proof. First, suppose g is bounded in each case, whence both g and $f \cdot g$ are ultimately periodic. This shows that $f \cdot g$ is honest in cases (i) and (iii) and superhonest in case (ii). Supposing g is not bounded so $x \trianglelefteq g(x)$, and P and Q are honest (on-line) programs which compute f and g, so that $P \cdot Q$ is an (on-line) program computing $f \cdot g$:

(i) We have $t_{P \cdot Q} = t_Q + t_P \cdot f_Q \trianglelefteq g + f \cdot g \trianglelefteq f \cdot g$,

if f is unbounded, since we have $x \trianglelefteq f(x)$ by Lemma 4.2.

(ii) Similar to case (i) if f is unbounded. If f and g are monotone, and f is bounded, then $f \cdot g$ is constant almost everywhere and so superhonest.

(iii) If g is ultimately linear, then $t_Q = 0$, and so

$$t_{P \cdot Q} \trianglelefteq f \cdot g.$$

The above theorem can be generalised to prove certain other closure properties of the honest functions. Just for the remainder of this section we introduce functions of more than one argument, and say that a n-place function f is honest if there is a program P, with n input registers $X_1, X_2, \ldots, X_n$, which computes f with

$$t_P(x_1, x_2, \ldots, x_n) \trianglelefteq f(x_1, x_2, \ldots x_n).$$

It is easy to see that each of the functions

$$\lambda xy \cdot x+y, \quad \lambda xy \cdot x \times y, \quad \lambda xy \cdot x^y$$

are honest in this extended sense.

<u>Theorem</u> 4.18 Suppose $g_1, g_2, \dots g_n$ are 1-place honest functions, and f is a n-place honest function. Then the function h defined by

$$h(x) = f(g_1(x), g_2(x), \dots, g_n(x))$$

is honest, if either

(a) each function g_j is bounded,

or (b) $\Sigma x_i \trianglelefteq f(x_1, x_2, \dots, x_n)$.

<u>Proof</u>. For each i, $1 \leq i \leq n$, let G_i be an honest program which computes g_i but stores the result in a new register A_i. Let F be a program which computes f, but reads the input from registers $A_1, A_2, \dots, A_n$. By the addition theorem, the program $G_1 + G_2 + \dots + G_n = G$ say, computes $g_i(x)$, for each input x, in register A_i for $1 \leq i \leq n$, and has a running time

$$t_G(x) \trianglelefteq \sum_{i=1}^{n} g_i(x).$$

Let H be the program G followed by program F. H computes h, and

$$t_H(x) \trianglelefteq \sum g_i(x) + f(g_1(x), \dots, g_n(x)).$$

In the case that each g_j is bounded, the right hand side of this inequality is bounded and so h is honest. If at least one g_j is unbounded, then $x \trianglelefteq g_j(x)$, and so $\sum x_i \trianglelefteq f(x_1, x_2, \dots, x_n)$ implies

$$t_H(x) \quad f(g_1(x), \ldots, g_n(x)).$$

Hence h is again honest.

<u>Corollary</u> 4.19 If f and g are honest, then so are

(i) $f+g$,

(ii) $f\times g$, provided $f>0$ and $g>0$,

(iii) f^g, provided $f>2$ and $g>0$.

<u>Proof.</u> (i) is immediate.

(ii) If f and g are honest, with $f,g>0$, then the functions $f-1$ and $g-1$ are honest by Theorem 4.17 (iii). Since

$$x+y < (x+1)(y+1) \text{ for all } x,y,$$

and $\lambda xy.(x+1)(y+1)$ is an honest function, we have

$$f\times g = (f-1+1)\times(g-1+1)$$

is honest.

(iii) If f and g are honest with $f\geq 2$ and $g>0$, then so are the functions $f-2$ and $g-1$. Since

$$x+y \leq (x+2)^{y+1} \text{ for all } x,y,$$

and $\lambda xy.(x+2)^{y+1}$ is honest, the conclusion follows.

Note that the conditions in (ii) and (iii) are necessary. Both f and g are honest, where

$$f(x) = x$$

and $$g(x) = (x \text{ even} \rightarrow 1,0),$$

but neither $f \times g$ or f^g are honest.

§5. Subtraction of honest functions

In this section, we show, under certain conditions, that the function f-g is honest if both f and g are. This result enables us to state more clearly the relationship between the honest and superhonest functions. The following subsidiary result is needed.

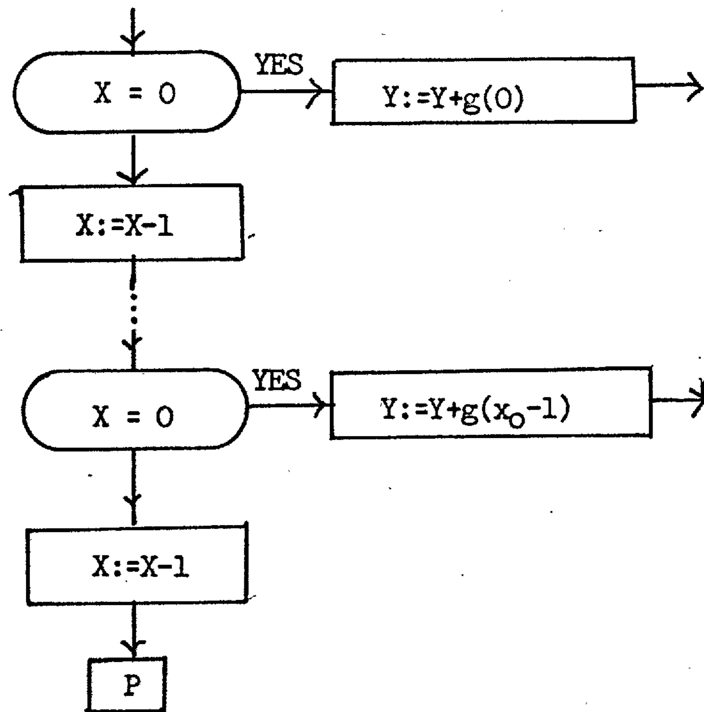
Lemma 4.20 Suppose that f is honest and g=f almost everywhere. Then g is honest.

Proof.

Suppose x_0 is such that $x \geqslant x_0$ implies $g(x) = f(x)$. If we define h by

$$h(x) = f(x+x_0),$$

then h is honest, by Theorem 4.17. Suppose P is an honest program which computes h, and let Q be the program:



It is clear that $f_Q(x) = (x<x_o \rightarrow g(x),\ h(x-x_o))$

$$= g(x),$$

and $\quad t_Q(x) = (x < x_o \rightarrow 0,\ t_p(x-x_o))$

$$g(x),$$

and so g is honest.

<u>Theorem</u> 4.21. If f and g are honest with $f \succcurlyeq g$ and $\liminf_{x\to\infty} \frac{f(x)}{g(x)} > 1$, then the function $f-g$ is honest.

<u>Proof</u>.

The theorem is proved by constructing a program R for which $t_R(x) = k(f(x) - g(x)) + 1$ almost everywhere, where k is some positive integer. Since the running time of any program is honest, Lemma 4.20 shows that the function $k(f-g)+1$ is honest, and the final

result follows by using Theorem 4.17.

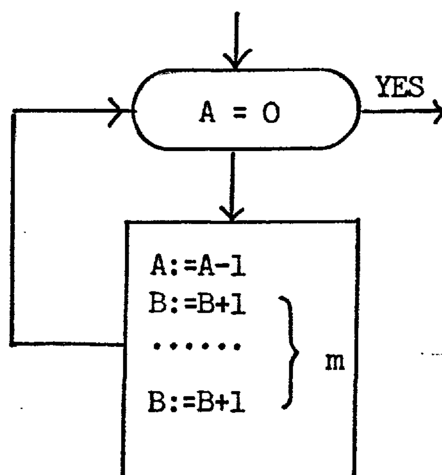
By Theorem 4.8, we can suppose that there are programs P and Q such that

$$f_P = f, \quad t_P = cf \quad \text{and} \quad f_Q = g, \; t_Q = cg,$$

for some positive integer c. By hypothesis, we can find an integer n so that $nf(x) \geq (n+1)g(x)$ almost everywhere. We modify programs P and Q, by replacing every output instruction of P by a sequence of n assignments A:=A+1, and every output instruction of Q by a sequence of (n+1) assignments A:=A-1, where A is a register not appearing in P and Q. The resulting programs are added, giving a program R' with a running time

$$t_{R'} = (c+n)f + (c+n+1)g,$$

and which leaves in A the quantity $nf-(n+1)g$. The final program R consists of R' followed by the code



where m is a positive integer that will be determined in a moment, and B is an arbitrary register. As long as $nf \geq (n+1)g$, i.e. almost everywhere, the running time of R is given by

$$t_R = (c+n)f + (c+n+1)g + 1 + (m+2)(nf - (n+1)g).$$

We now choose m to satisfy

$$(m+2)n + c + n = (m+2)(n+1) - (c+n+1),$$

which gives $m = 2c + 2n-1$. With this value of m,

$$t_R = k(f-g) + 1,$$

where $k = 2n(n+c+1)+c$, and the theorem is proved.

The following example shows that the condition

$$\liminf_{x\to\infty} \frac{f(x)}{g(x)} > 1$$

is, in general, necessary.

<u>Example</u> 4.22 Let f and g be the honest functions

$$f(x) = (x \text{ even} \to x,\ 2x)$$

and $$g(x) = x$$

we have $f \geq g$ and $\liminf_{x\to\infty} \frac{f(x)}{g(x)} = 1$, but $f-g$ is not honest.

Corollary 4.23 Suppose f is a monotone honest function for which

$$\liminf_{x \to \infty} \frac{f(x+1)}{f(x)} > 1.$$

Then f is superhonest.

Proof. Immediate from Corollary 4.14, since the condition implies Δf is honest.

In Chapter 5, we show the existence of a monotone honest function, which is not superhonest.

CHAPTER FIVE

NECESSARY CONDITIONS FOR ON-LINE COMPUTATION

The size theorem of Chapter 2 gives a simple condition that a function f must satisfy in order to be computable within a time bound t; namely, $f \trianglelefteq t$. However, this condition does not give any information on the computational complexity of 01-valued or slowly increasing functions. In this chapter we develop a sharper condition, but one that only pertains to on-line computation (Sections 1-3). Using this condition it is possible to show the existence of functions f, with f monotone and $f(x) \trianglelefteq x$, which are not on-line real time I_0 - computable. In the case of a general instruction set I, the condition can only be used to construct functions f such that for any on-line I-program P computing f, we must have $t_P(x) \geqslant \varepsilon x$, where $\varepsilon > 0$ is a constant independent of P. (Section 4). Supposing f is such a function, this means that if f is on-line real time I-computable, then I cannot possess the speed up property of Chapter 3. In Section 5, we use this fact to show that certain instruction sets do not possess speed up. Section 6 shows that there is a montonic honest but not superhonest function - answering a question raised in Chapter 4, and Section 7 gives a structural characterisation of the real time I-computable functions, in terms of a certain type of programming language.

The basic necessary condition turns on two equivalence relations, one associated with functions, and the other with on-line programs. Sections 1 and 2 define these relations and investigates their properties.

§1. The equivalence relation $\equiv_n$

Let f be an arbitrary (total) function. For each integer $n \geq 0$, we define the equivalence relation $\equiv_n \pmod f$ on N by the condition:

$$\text{for all} \quad x, y \in N, \quad x \equiv_n y \pmod f \quad \text{if}$$
$$\text{for all} \quad z \leq n, \quad f(x+z) - f(x) = f(y+z) - f(y).$$

It is easily verified that $\equiv_n$ is an equivalence relation. We are interested in the number of equivalence classes induced by $\equiv_n$ on N. Taking $S(x) = x^2$, it is clear for any $n > 0$ that $x \equiv_n y \pmod S$ if and only if $x = y$, so that each integer stands in an equivalence class by itself. On the other hand, if $E(x) = x$, then $x \equiv_n y \pmod E$ is satisfied for any integers x, y, and n, so there is only one equivalence class.

For $m \geq n$, we let $J_f(m,n)$ denote the number of equivalence classes induced by $\equiv_n \pmod f$ on the initial subset

$$\{0, 1, \ldots, m-n\}$$

of N. Clearly, $J_f(m,n)$ is always finite, and the following two properties of J_f are immediate:

(i) $1 \leq J_f(m,n) \leq m-n+1$.

(ii) J_f is monotone in each argument.

Moreover, the bounds given in (i) can be obtained for $n > 0$, since $J_E(m,n) = 1$ and $J_S(m,n) = m-n+1$, where E and S are the functions defined above.

It may be mentioned that the reason we do not define J_f over the, possibly more natural, initial subset $\{0, 1, \ldots, m\}$ of N, is simply because the above definition enables certain properties to be stated more elegantly.

The definition of $J_f(m,n)$ means just that we can find a sequence

$$x_1, x_2, \ldots, x_J$$

of integers x_j, where $0 \leq x_J \leq m-n$ for $1 \leq j \leq J = J_f(m,n)$, such that

$$j \neq k \text{ implies } x_j \not\equiv_n x_k (\text{mod } f).$$

This sequence is not necessarily unique, and we want to be able to choose one with, roughly speaking, minimum density. Define $S_f^{(i)}(m,n)$ for $1 \leq i \leq J$, by the condition

$$S_f^{(i)}(m,n) = \text{size } \{x_k: |x_i - x_k| < n\}.$$

In words, $S_f^{(i)}(m,n)$ is the number of elements of the above sequence which lie within distance n of the element x_i. If $T = \{x_1, x_2, \ldots, x_J\}$ is chosen so that

$$\max_{1 \leq i \leq J} \quad S_f^{(i)}(m,n)$$

is minimised, we refer to T as a <u>spanning set with minimum density</u>, and define the <u>density</u> S_f to be

$$S_f(m,n) = \min_{T} \max_{1 \leq i \leq J} S_f^{(i)}(m,n).$$

It is easily verfied that S depends only on f,m and n, and

$$1 \leq S_f(m,n) \leq 2n-1$$

for all m and n, with $n > 0$.

We consider one example to see how these definitions work out in practice. This example turns out to be important in the following sections.

<u>Example</u> 5.1 Suppose $D = \Sigma\delta$, where δ is the function whose sequence of values

$$\delta(0), \delta(1), \ldots \quad \text{etc.},$$

is identical with the sequence of dyadic integers, written one after the other. The first few terms of this sequence are

0 1 0 0 0 1 1 0 1 1 0 0 0 0 0 1 ...

D is clearly monotone and satisfies $D(x) \leq x$. The first few values of D are

0 1 1 1 1 2 3 3 4 5 ...

Since every binary pattern of length $n \geq 1$ has appeared after at most $\sum_{j=0}^{n} j2^j$ terms of δ, i.e. by $\delta(\sum^{n} j2^j - 1)$, it follows that

$$J_D\left(\sum^{n} j2^j + n - 1, n\right) = 2^n \quad \text{for all} \quad n \geq 1.$$

Since $\sum^{n} j2^j = (n-1)2^{n+1} + 2$, we therefore have

$$J_D\ ((n-1)(2^{n+1} + 1) + 2, n) = 2^n.$$

Moreover,

$$S_D\ ((n-1)(2^{n+1} + 1) + 2, n) = 1,$$

since we can choose a spanning set T with minimum density, by taking

$$T = \{(\sum_{j=1}^{n-1} j2^j) + kn : 0 \leq k \leq 2^n - 1\}.$$

The following five lemmas summarise the important properties of J_f and S_f. The first is an analogue of the Nerode theorem for regular sets (Rabin and Scott [34]).

Lemma 5.2 If J_f is bounded, then f is ultimately linear.

Proof. If J_f is bounded, then for some constant k, the equivalence relation $\equiv$, where

$$x \equiv y \quad \text{if for all } z \geq 0 \quad f(x+z) - f(x) = f(y+z) - f(y),$$

induces just k equivalence classes on N. This means that among the numbers

$$0, 1, \ldots, k$$

there exists i,j with $i < j$ such that $i \equiv j$. Suppose $j = i+c$, where $c > 0$. It follows that

$$f(i+z) - f(i) = f(i+c+z) - f(i+c)$$

for all $z \geq 0$. Taking $y = z + i$, we have

$$f(y+c) - f(y) = f(i+c) - f(i) = d \quad \text{say},$$

for all $y \geq i$. Hence f is ultimately linear.

Lemma 5.3 For all m and n (with $m \geq n$), we have $nJ_f(m,n) \leq m\, S_f(m,n)$.

Proof. Let m and n be fixed. Choose a spanning set $T = \{x_1,\ldots,x_J\}$ with minimum density $S = S_f(m,n)$. Suppose without loss of generality that $x_i < x_j$ for $i < j$. The definition of S implies that $x_{jS+1} \geq x_1 + jn \geq jn$, for $j=0,1,\ldots$ etc. Since $x_i \leq m-n$, it follows that

$$J \leq \left(\frac{m-n}{n}\right) S + 1 \leq \frac{m}{n}\, S,$$

since $S \geq 1$.

Lemma 5.4 $x \equiv_n y \pmod f$ if and only if for all z, $1 \leq z \leq n$, we have $\Delta f(x+z) = \Delta f(y+z)$.

Proof. Suppose $x \equiv_n y \pmod f$. Since

$$f(x+z) - f(x) = \sum_{r=1}^{z} \Delta f(x+r),$$

it follows that

$$\sum_{r=1}^{z} \Delta f(x+r) = \sum_{r=1}^{z} \Delta f(y+r),$$

for all $z \leq n$. Taking $z = 1,2, \ldots$ in succession, we have

$$\Delta f(x+z) = \Delta f(y+z) \quad \text{for} \quad 1 \leq z \leq n.$$

Conversely, this condition ensures that $x \equiv_n y \pmod f$.

<u>Lemma</u> 5.5 Suppose $k = \text{size}\,\{\Delta f(x) : 1 \leq x \leq m-n+1\}$.
Then

$$J_{\Delta f}(m+1, n-1) \leq J_f(m,n) \leq kJ_{\Delta f}(m+1, n-1).$$

<u>Proof</u>. By lemma 5.4 $x \equiv_n y \pmod f$ if and only if

$$\Delta f(x+z) = \Delta f(y+z) \quad \text{for} \quad 1 \leq z \leq n.$$

In turn, this condition is equivalent to:

(i) $\Delta f(x+1+z) - \Delta f(x+1) = \Delta f(y+1+z) - \Delta f(y+1)$ for $0 \leq z < n$,

and (ii) $\Delta f(x+1) = \Delta f(y+1)$.

Therefore, a necessary and sufficient condition that $x \equiv_n y \pmod f$ is

$$(x+1) \equiv_{(n-1)} (y+1) \pmod{\Delta f} \quad \text{and} \quad \Delta f(x+1) = \Delta f(y+1),$$

from which the estimates follow.

<u>Lemma</u> 5.6 Suppose f is monotone. Then there exists a constant c such that for all m and n with

$$\frac{\log_e J_f(m,n)}{n}$$

sufficiently large, we have

$$\frac{J_f(m,n)}{S_f(m,n)} \quad \log_e J_f(m,n) \leq cf(m).$$

Proof.

Let m and n be given, and let $J = J_f(m,n)$ and $S = S_f(m,n)$ for short. It follows from Lemma 5.4 that we can find a spanning set $T = \{x_1, x_2, \ldots x_J\}$ with minimum density S such that the set of sequences

$$\Delta f(x_1+1), \ldots, \Delta f(x_1+n)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\Delta f(x_J+1), \ldots, \Delta f(x_J+n)$$

are pairwise distinct. Let

$$\sigma_j = \sum_{r=1}^{n} \Delta f\ (x_j+r).$$

Since each term in any sequence can occur in upto as many as S-1 of the other sequences, we have

$$f(m) \geq \frac{1}{S} \sum_{j=1}^{J} \sigma_j.$$

The proof is now completed by putting an appropriate lower bound to

$$\sum_{j=1}^{J} \sigma_j.$$

Let $\gamma_n(j)$ denote the number of distinct integer sequences

$$a_1, a_2, \dots a_n \quad \text{where} \quad a_j \geq 0,$$

with $\sum a_i = j$. Since the set of such sequences can be described, recursively, as the union from k = 0 to j of sequences of the form

$$a_1, a_2, \dots, a_{n-1}, k \quad \text{where} \quad \sum a_i = j-k,$$

it follows that

$$\gamma_n(j) = \sum_{r=o}^{j} \gamma_{n-1}(r), \tag{1}$$

for all $n > 1$. Since $\gamma_1(j) = 1$, the recurrence relation can be solved to give

$$\gamma_n(j) = \binom{j+n-1}{n-1} . \tag{2}$$

The sum $\sum \sigma_j$ will be minimised by having distinct sequences

$$\Delta f(x_i+1), \dots \quad \Delta f(x_i+n),$$

with as small sums as possible. Since f is monotone, no term is negative and no sum σ_i is negative. Thus

$$\sum_{j=1}^{J} \sigma_j \geq \sum_{j=o}^{B} j\gamma_n(j) + (B+1)(J - \sum_{j=o}^{B} \gamma_n(j)), \tag{3}$$

where B is defined by the condition

$$\sum_{j=0}^{B} \gamma_n(j) \leq J < \sum_{j=0}^{B+1} \gamma_n(j),$$

that is

$$\gamma_{n+1}(B) \leq J < \gamma_{n+1}(B+1), \qquad (4)$$

Using (1).

In order to evaluate the right hand side of (3), suppose, conventionally, that $\gamma_{n+1}(x) = 0$ for $x < 0$. Using (1), we have

$$\sum_{j=0}^{B} j\gamma_n(j) = \sum_{j=0}^{B} j\,(\gamma_{n+1}(j) - \gamma_{n+1}(j-1))$$

$$= \sum_{j=0}^{B} j\gamma_{n+1}(j) - \sum_{j=0}^{B} (j-1)\,\gamma_{n+1}(j-1)$$

$$- \sum_{j=0}^{B} \gamma_{n+1}(j-1),$$

$$= B\,\gamma_{n+1}(B) - \gamma_{n+2}(B-1).$$

Now from (2), we have $\gamma_{n+2}(B-1) = \frac{B}{n+1}\gamma_{n+1}(B)$. Thus

$$\sum_{j=0}^{B} j\gamma_n(j) = \frac{Bn}{n+1}\,\gamma_{n+1}(B). \qquad (5)$$

Substituting (5) into (3) and using (1), we obtain

$$\sum \sigma_j \geq (B+1)J - \frac{B+n+1}{n+1}\,\gamma_{n+1}(B).$$

Hence using (4)

$$\sum \sigma_j \geq \frac{Bn}{n+1} J \geq \frac{(B+1)}{4} J,$$

provided $B \geq 1$. To estimate B, we have from (4) and (2), that

$$J < \binom{B+1+n}{n} .$$

It can be shown, using Stirlings approximation, that

$$\binom{B+1+n}{n} \leq e^{B+1+n} ,$$

whence

$$B+1 \geq \log_e J - n \geq \tfrac{1}{2} \log_e J,$$

provided $\dfrac{\log_e J}{n}$ is sufficiently large. Thus the lemma is proved.

§2. The equivalence relation $\sim_n$

The second equivalence relation on N is defined with respect to an arbitrary on-line program P. Let P be given and, for each $x \geq 0$ and $n \geq 0$, let $C_P(x,n)$ denote that subsequence of the computation sequence of P with infinite input, which begins with the instantaneous description at the point where the (x+1)st input test is obeyed and ends with the instantaneous description immediately prior to the point where the (n+1)st next work register instruction is executed. We now define

$$x \sim_n y \pmod{P} \quad \text{if} \quad C_P(x,n) = C_P(y,n).$$

The number of equivalence classes induced by $\sim_n$ will be denoted by $K_P(n)$. Since $x \sim_{n+1} y$ implies $x \sim_n y$, it is clear that K_P is monotone. Moreover, as the following lemma shows, K_P is always well defined.

<u>Lemma</u> 5.7 Suppose P is an on-line program (defined over an arbitrary instruction set) and has c distinct input tests. Then

$$K_P(n) \leq c2^n$$

for all n.

<u>Proof</u>. It is sufficient to put an upper bound to the number of distinct sequences of the form $C_P(x,n)$ for some x. If two such sequences begin with the same input test, they can only differ after the execution of some work register test. Since each test can yield at most two possible continuations, the number of such sequences is no greater than 2^n. If there are c possible input tests to begin with, then the number of distinct sequences is no greater than $c2^n$.

This bound can be much improved if P is an I_0- program.

<u>Lemma</u> 5.8 If P is an on-line I_0- program which refers to just k work registers, then

$$K_P(n) \leq n^k.$$

Proof. We can write the instantaneous description $I_P(x)$ at the point where the (x+1)st input test is obeyed, in the form

$$(\ell, x_1, y_1, a_1, a_2, \ldots, a_k)$$

where ℓ labels the (x+1)st input test to be obeyed, and $a_1, a_2, \ldots, a_k$ denote the contents of the work registers at this point. Suppose, similarly, that $I_P(y)$ is of the form

$$(m, x_2, y_2, b_1, b_2, \ldots, b_k).$$

Now if P is an I_0- program, then $C_P(x,n)$ and $C_P(y,n)$ can only differ if either:

(i) $\ell \neq m$,

or (ii) there is a j, where $1 \leqslant j \leqslant k$ such that

$$-n \leqslant a_j, b_j \leqslant n \text{ and } a_j \neq b_j.$$

Since the number of distinct instantaneous descriptions satisfying (i) or (ii) is at most $c(2n+2)^k$, where c is the number of input tests appearing in P, we have

$$K_P(n) \leqslant c(2n+2)^k,$$

from which the conclusion follows.

§3. The basic conditions

The fundamental relationship which connects the two equivalence relations can be expressed as follows.

Lemma 5.9 Let P be an on-line program. Suppose $n \geq 0$ is given, and T is a set of integers such that

$$x \neq y \quad \text{implies} \quad x \not\equiv_n y \pmod{f_P}$$

for all $x, y \in T$. Then

$$\text{size}(T) \leq K_P(t),$$

where $t = \max_{x \in T} \{t_P(x+n) - t_P(x)\}$.

Proof.

The proof follows immediately from the fact that

$$x \sim_t y \pmod{P} \quad \text{implies} \quad x \equiv_n y \pmod{f_P}$$

for all $x, y \in T$.

Corollary 5.10 If K_P is bounded, then f_P is ultimately linear.

Proof. It follows, from Lemma 5.9, that the size of T is bounded if K_P is. But this means that J_f is bounded, where $f = f_P$,

and so, by Lemma 5.2, f_P is ultimately linear.

From now on, we write J_P for J_{f_P} and S_P for S_{f_P} to avoid messy subscripts. If f_P is not ultimately linear, the function K_P^{-1} defined by

$$K_P^{-1}(x) = \min y\ [x \leq K_P(y)]$$

is total. This follows from Corollary 5.10. Moreover, we have $K_P^{-1}(K_P(x)) \leq x$.

<u>Lemma</u> 5.11 Suppose P is an on-line program and f_P is not ultimately linear. Then for all m and n,

$$t_P(m) \geq \sum_{j=o}^{R} K_P^{-1}\ (jS_P(m,n)),$$

$$\text{where} \qquad R = \left\lceil \frac{J_P(m,n)}{S_P(m,n)} \right\rceil .$$

<u>Proof</u>. Let m and n be fixed and let $J = J_P(m,n)$ and $S = S_P(m,n)$. By definition of $\equiv_n$, there is a spanning set T_1 with minimum density S, and containing exactly J elements, such that

$$x \neq y \ \text{ implies } \ x \not\equiv_n y \ (\text{mod } f_P)$$

for all $x, y \in T_1$. Choose $x_1 \in T_1$ so that

$$\max_{x \in T_1} \{t_P(x+n) - t_P(x)\} = t_P(x_1+n) - t_P(x_1),$$

and let $t_P(x_1+n) - t_P(x_1)$ be denoted by t_1 for short.
Applying Lemma 5.9, we have

$$J \leq K_P(t_1).$$

Remove x_1 from T_1 and also all y for which $|x_1 - y| < n$.
By definition of S, the remaining set T_2 contains at least J-S members. In a similar fashion, we can now choose $x_2 \in T_2$ so that

$$J-S \leq K_P(t_2) \quad \text{where} \quad t_2 = t_P(x_2+n) - t_P(x_2).$$

Proceeding in this way, we can choose a sequence $x_1, x_2, \ldots, x_r$ where $0 \leq x_j \leq m-n$ and $r = [J/S] + 1$, such that for $1 \leq j \leq r$ we have

$$J - (j-1)S \leq K_P(t_j),$$

$$\text{where} \qquad t_j = t_P(x_j+n) - t_P(x_j).$$

Moreover, by construction, $|x_i - x_j| \geq n$ for $i \neq j$, whence

$$t_P(m) \geq \sum_{j=1}^{r} t_j.$$

Now

$$t_j \geq K_P^{-1} (K_P(t_j)) \geq K_P^{-1} (J-(j-1)S)$$

since K_P^{-1} is total and monotone if f_P is not ultimately linear.

Therefore

$$t_P(m) \geq \sum_{j=0}^{R} K_P^{-1}(A + jS),$$

where $R = [J/S]$ and $A = [J,S]$. Since K_P^{-1} is monotone, the conclusion follows.

The next task is to simplify the lower bound given in the statement of Lemma 5.11 by using the estimates on K_P given in the last section. These lead directly to the following two theorems.

Theorem 5.12 Suppose P is an arbitrary on-line program, where f_P is not ultimately linear. Then for all m and n such that $\frac{J_P(m,n)}{n}$ is sufficiently large, we have

$$t_P(m) \geq \frac{J_P(m,n)}{4S_P(m,n)} \log_2 J_P(m,n)$$

Proof.

Lemma 5.7 states that $K_P(x) \leq c2^x$ for some c, whence $K_P^{-1}(x) \geq \log_2 (\frac{x}{c})$, since K_P^{-1} is total. Using this estimate in Lemma 5.11, it follows that

$$t_P(m) \geq \sum_{j=1}^{R} \log_2 (\frac{jS}{c}) = \log_2 (R! \ (\frac{S}{c})^R).$$

Now by Stirlings approximation,

$$R! \geq (\frac{R}{e})^R \text{ provided } R \text{ is sufficiently large,}$$

and also

$$R = \left[\frac{J}{S}\right] \geq \frac{J}{2S} ,$$

and $$\frac{RS}{ec} \geq \frac{J-S}{ec} \geq J^{\frac{1}{2}} ,$$

provided $\frac{J}{S}$ is sufficiently large. Since $S < 2n$, this means that provided $\frac{J}{n}$ is sufficiently large,

$$t_P(m) \geq \frac{J}{4S} \log_2 J,$$

and the theorem is proved.

<u>Theorem</u> 5.13 Suppose P is an on-line I_0- program which uses k work registers, and f_P is not ultimately linear. Then for some $\varepsilon > 0$,

$$t_P(m) \geq \varepsilon \frac{\{J_P(m,n)\}^{1+\frac{1}{k}}}{S_P(m,n)}$$

for all m and n.

<u>Proof</u>. Lemma 5.8 states that $K_P(x) \leq cx^k$ for some constant c, whence $K_P^{-1}(x) \geq \varepsilon x^{1/k}$ for some $\varepsilon > 0$. Using this estimate in Lemma 5.11, we have

$$t_P(m) \geq \varepsilon \sum_{j=0}^{R} (jS)^{\frac{1}{k}} .$$

Since

$$\sum_{j=0}^{R} j^{1/k} \geq \delta R^{\left(1+\frac{1}{k}\right)} \quad \text{for some} \quad \delta > 0,$$

we have

$$t_P(m) \geq \varepsilon\delta\, S^{\frac{1}{k}} \left[\frac{J}{S}\right]^{\left(1+\frac{1}{k}\right)}$$

and the conclusion follows.

The following corollaries are immediate from Theorems 5.12 and 5.13 by using the estimate $S_P(m,n) < 2n$.

<u>Corollary</u> 5.14 Under the hypotheses of Theorem 5.12,

$$t_P(m) \geq \frac{1}{8n} J_P(m,n) \log_2 J_P(m,n).$$

<u>Corollary</u> 5.15 Under the hypotheses of Theorem 5.13

$$t_P(m) \geq \frac{\varepsilon}{n} \{J_P(m,n)\}^{1+\frac{1}{k}}.$$

§4. <u>Applications</u>

Theorem 5.13 can be used to show the existence of slowly increasing functions which are not on-line real-time I_0- computable. We consider, as an example, the function D defined in Example 5.1.

<u>Theorem</u> 5.16 If D is computable by an on-line I_0- program P, then for some $\delta > 1$

$$\left(\frac{x}{\log_2 x}\right)^{\delta} \leq t_P(x).$$

In particular, it follows that D is not real time on-line I_0- computable.

Proof. From the discussion in Example 5.1 we know that

$$J_D(h(n),n) = 2^n$$

and

$$S_D(h(n),n) = 1$$

for all n, where $h(n) = (n-1)(2^{n+1}+1) + 2$. Thus, if P is an on-line I_0- program which computes D, then there is an $\varepsilon > 0$ and a $\delta > 1$ such that

$$t_P(h(n)) \geq \varepsilon 2^{n\delta}.$$

Let x be arbitrary and suppose n is such that $h(n) \leq x < h(n+1)$. This gives

$$n \geq \log_2 x - \log_2\log_2 x - 3,$$

whence

$$t_P(x) \geq t_P(h(n)) \geq \varepsilon 2^{(\log_2 x - \log_2\log_2 x - 3)\delta} \geq \left(\frac{x}{\log_2 x}\right)^{\delta},$$

proving the theorem.

Later on we shall show that there is an on-line I_0- program P which computes D for which

$$t_P(x) \leq \left(\frac{x}{\log_2 x}\right)^2.$$

(Theorem 5.19).

Unfortunately, Theorem 5.12 cannot be used to show the existence of a monotone function f with $f(x) \leq x$, which is not on-line I -

computable for any instruction set I. This can be seen by showing that the lower bound given by Theorem 5.12 satisfies

$$\frac{J_P(m,n)}{S_P(m,n)} \log_2 J_P(m,n) \leq m. \tag{1}$$

There are two cases to be considered:

(i) If $\log_2 J_P(m,n) \leq cn$ for some constant c, then (1) follows by using Lemma 5.3.

(ii) If, on the other hand,

$$\frac{\log_2 J_P(m,n)}{n}$$

is sufficiently large, we have

$$\frac{J_P}{S_P} \log_2 J_P \leq f(m)$$

by Lemma 5.6. Since $f(m) \leq m$, inequality (1) holds in this case also.

However, Theorem 5.12 can be used to give some sort of lower bound.

<u>Theorem</u> 5.17 If D is computable by an on-line program P, then

$$t_P(x) \geq \frac{x}{64}$$

for sufficiently large x.

Proof. Suppose P is an on-line program which computes D.

From Theorem 5.12, we have

$$t_P(h(n)) \geq n2^{n-2},$$

where $h(n) = (n-1)(2^{n+1}+1)+2$ as in Theorem 5.16. From this estimate, we can deduce that

$$t_P(x) \geq (\log_2 x - \log_2\log_2 x - 3)2^{(\log_2 x - \log_2\log_2 x - 5)},$$

whence

$$t_P(x) \geq \frac{x}{64}$$

for sufficiently large x.

This result has the following interesting consequence:

Theorem 5.18 If I is any instruction set such that D is on-line real time I-computable, then I does not possess the speed up property.

Proof. If I does possess the speed up property, we could use it to produce an on-line program P which contradicted Theorem 5.17.

We now give two examples of this theorem.

§5. Instruction sets not possessing speed up

Let I_1 be the instruction set which includes I_0 and has, in addition, the instructions

assignments	$A_j:=A_j+A_k$	$A_j:=A_j-A_k$
tests	$A_j \geq A_k$.	

Let I_2 be the instruction set which includes I_0 and has, in addition,

assignments	$A_j:=A_j+1$	$A_j:=A_j-1$	$A_j:=0$	
	$j:=j+1$	$j:=j-1$	$j:=0$	
tests	$A_j=0$	$A_j\geq 0$	$j=0$	$j\geq 0$

That is I_2 consists of I_0 together with the possibility of referring to the work registers A_1, A_2, ... indirectly through an index register j.

Theorem 5.19 (i) D is on-line real time I_1 - computable.

(ii) D is on-line real time I_2 - computable.

(iii) D is on-line I_0- computable within time

$$\lambda x. \left(\frac{x}{\log_2 x}\right)^2 .$$

Proof. The proof in each case is by direct construction of a program P to compute D.

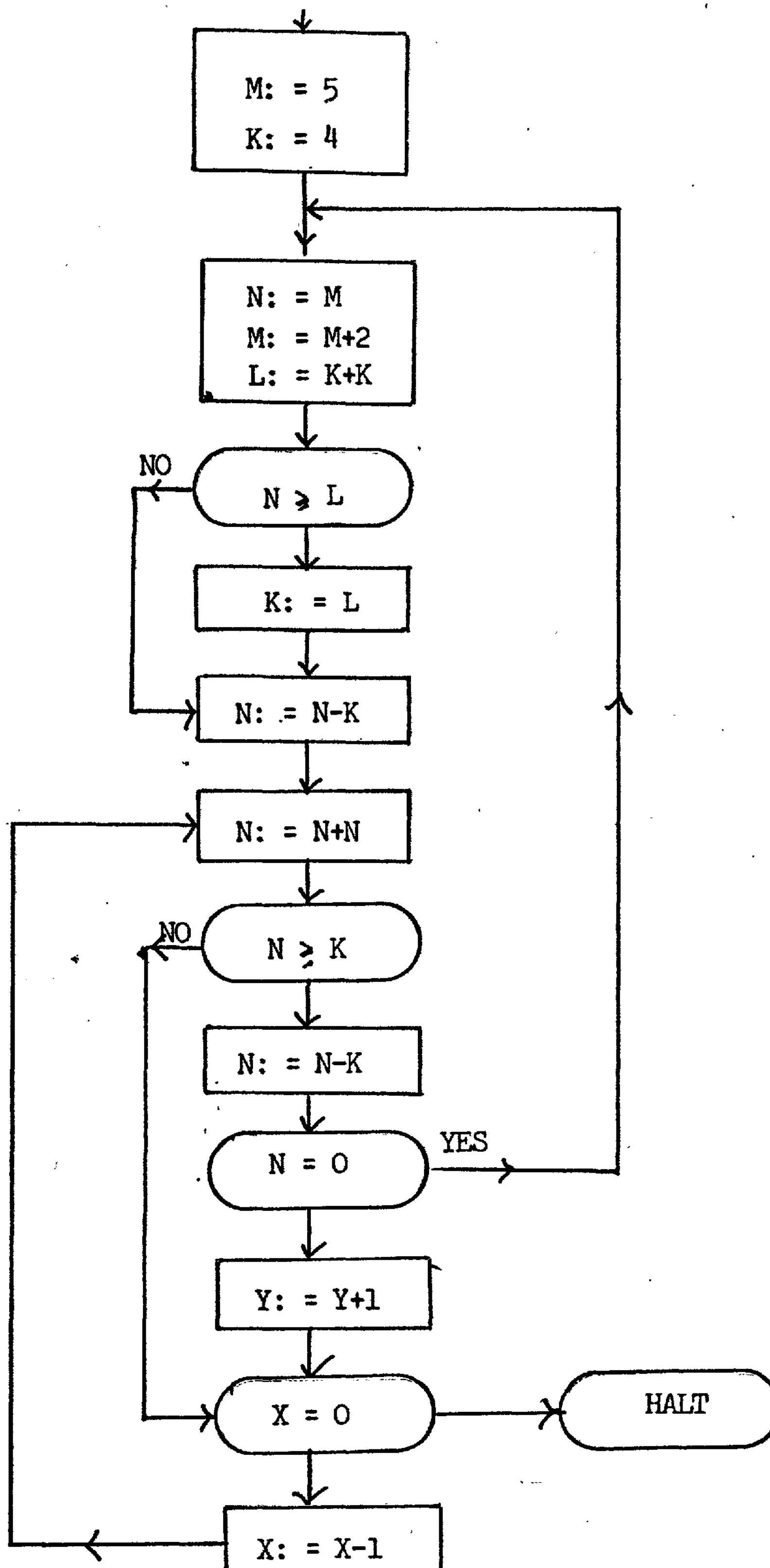
(i) In the first case, P works by determining, between successive input tests, the next symbol in the sequence

$$0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ \ldots$$

and incrementing the output register if the next symbol happens to be 1. P does this by extracting the binary representation of the odd numbers 5, 7, 9, ... one after the other. The binary representations of these numbers are

101, 111, 1001, 1011, ... etc.,

and the sequence formed from the interior digits of each number (i.e. every digit except the first and last of each number) is just the sequence we want. The interior digits of the odd number n are extracted as follows. Initially, register N contains n and register K contains 2^k, where $2^k \leq n < 2^{k+1}$. The first digit of n is removed by subtracting K from N and leaving $n-2^k$ in N. To extract the next digits, the contents of N are successively doubled and compared against K. If $N < K$, then the next digit is 0 and no output is given. If $N > K$, then the next digit is 1 and Y is incremented. If $N = K$, then the processing of N is complete, and the next odd number is set up for processing. With these remarks, we now give the complete program P.


M: = 5
K: = 4
N: = M
M: = M+2
L: = K+K
NO
N ≥ L
K: = L
N: = N-K
N: = N+N
NO
N ≥ K
N: = N-K
N = 0
YES
Y: = Y+1
X = 0
HALT
X: = X-1

Since $f_P = D$ and $t_P(x) \leq 18x$, part (i) is established.

(iii). Program P can be modified into an I_0- program that computes D. Instead of doubling N and comparing against K, the modified version Q halves K and compares against N. Thus during the processing of the number n, no number greater than 2n is stored in any register. The various instructions of Q can be expanded as macros defined over I_0, and it is possible to verify that the resulting program R processes n (i.e. extracts the interior digits of n) within cn steps for some constant c. Since this processing produces a further $[\log_2 n] - 1$ values of D, we have

$$t_R\left(\sum_{j=2}^{y} ([\log_2(2j+1)] - 1)\right) \leq \sum_{j=2}^{y} j \qquad \text{for all} \quad y,$$

which simplifies to

$$t_R\left(\sum_{j=1}^{y} [\log_2 j]\right) \leq y^2.$$

Let x be arbitrary, and suppose y satisfies

$$\sum_{j=1}^{y} [\log_2 j] \leq x < \sum_{j=1}^{y+1} [\log_2 j],$$

whence it follows that $y \leq (\frac{x}{\log_2 x})$, and so

$$t_R(x) \leq t_R\left(\sum_{j=1}^{y+1} [\log_2 j]\right) \leq (y+1)^2 \leq \left(\frac{x}{\log_2 x}\right)^2$$

This establishes part (iii).

(ii) The I_2- program P which computes D, evaluates for $n = 1,2, \ldots$, the dyadic expansion of n in the registers $A_1, A_2, \ldots A_{[\log_2 n]}$. Generating the expansion for n+1, given the expansion for n, can be achieved in $c[\log_2 n]$ steps for some constant c. As a result of expanding n, a further $[\log_2 n]$ values of D are computed, so that P operates in real-time. The details are left to the reader.

Corollary 5.20 Neither I_1 nor I_2 possess the speed up property.

It is worth mentioning that Theorem 5.19 also shows that both I_1 and I_2 are strictly more powerful instruction sets than I_0, in the sense that on-line real-time I_1 and I_2-programs compute a strictly larger class of functions than on-line real-time I_0- programs.

§6. Further results

The techniques of Section 3 can be used to settle some of the questions raised in previous chapters. In particular, we can now establish the existence of a monotone I_0- honest function which is not I_0- superhonest.

Theorem 5.21 Let δ be the 0-1 valued function described in Example 5.1, and let E be defined by

$$E(x) = x + \delta(x).$$

(i) E is a monotone I_0- honest function, but is not I_0- superhonest.

(ii) E is real time I_0- computable, but not on-line real time I_0- computable.

The proof makes use of some of the closure properties enjoyed by the real time I_0- computable functions. The following lemma can be verified by direct construction of the appropriate programs:

<u>Lemma</u> 5.22 If f and g are real time I_0- computable, then so are each of the following functions:

(i) $f \cdot g$,

(ii) $f+g$,

(iii) $f-g$ provided $f \geq g$,

(iv) $[f/g]$,

(v) $[f, g]$.

<u>Proof of Theorem</u> 5.21

E is clearly total and monotone and

$$x \leq E(x) \leq 2x.$$

It follows that, if E is real time computable, then it is honest, and if it is superhonest, then it is on-line real time computable. Hence it suffices to prove the second assertion only. In order to show that E is not on-line real time I_0- computable, it will be enough to prove

$$J_D(m,n) \leq J_E(m,n) \quad \text{for all } m,n,$$

where $D = \Sigma\delta$ is the function used in Section 4. By definition, $x \equiv_n y \pmod{E}$ if and only if

$$E(x+z) - E(x) = E(y+z) - E(y)$$

for $0 \leq z \leq n$, i.e.

$$\delta(x+z) - \delta(x) = \delta(y+z) - \delta(y)$$

for $0 \leq z \leq n$. In other words,

$$J_E(m,n) = J_{\Delta D}(m,n).$$

Lemma 5.5 now shows

$$J_D(m,n) \leq 2J_E(m+1, n-1) \leq 2(J_E(m,n)+1),$$

and the demonstration follows.

In order to show E is real time I_0- computable, it is sufficient to show that δ is. This is accomplished by deriving a closed formula for δ and using Lemma 5.22.

For each $x \geq 0$, we characterise the xth position in the sequence S of dyadic integers, by three quantities $a(x)$, $b(x)$ and $p(x)$. The first, $a(x)$, is a positive integer and denotes the number of the area into which x falls, where the jth area is that subsequence of S which consists of all the dyadic integers of length j. Next, $b(x)$ is an integer in the range $0 \leq b(x) < 2^{a(x)}$ and denotes the number of the block in area $a(x)$ into which x falls. A block is just the subsequence of S consisting of a single dyadic integer.

Finally, p(x) is an integer in the range $0 \leq p(x) < a(x)$ and denotes the position in block b(x) of area a(x) at which x occurs.

Given these three quantities, $\delta(x)$ can be determined as follows: if $p(x) = a(x) - 1$, then $\delta(x)$ will be 0 or 1 depending on whether b(x) is even or not; similarly, if $p(x) = a(x) - 2$, then $\delta(x)$ will be 0 or 1 depending on whether $[b(x), 4] < 2$. In general, the condition is

$$\delta(x) = 0 \text{ if and only if}$$

$$[b(x), 2^{a(x)-p(x)}] < 2^{a(x)-p(x)-1} .$$

Thus, in order to compute $\delta(x)$ for a given x, it is sufficient to determine a(x), b(x) and p(x) and see whether the above condition holds. The functions a, b, and p are determined as follows. Since the jth area is of length j^{2^j}, a given x falls in area k, where

$$A(k-1) \leq x < A(k) \quad \text{and} \quad A(k) = \sum_{j=1}^{k} j^{2^j}$$

In other words,

$$a(x) = A^*(x), \quad \text{where} \quad A(x) = (x-1)2^{x+1} + 2.$$

Furthermore, it is easily seen that

$$b(x) = \left[\frac{x-A(a(x)-1)}{a(x)}\right]$$

and

$$p(x) = [x-A(a(x)-1), a(x)] .$$

At this point, it is only necessary to remark:

(1) A is a strictly monotone I_0- superhonest function, whence by Theorems 4.5 and 4.6 both a and A(a-1) are on-line real time I_0- computable.

(2) From (1) and Lemma 5.22 we have that b, p and also a-p are real time I_0- computable.

(3) Since $\lambda x.2^x$ is superhonest, the function 2^{a-p} is I_0- computable by some program P

for which $t_P(x) \leq x + 2^{a(x)-p(x)}$.

However,

$$2^{a(x)-p(x)} \leq 2^{a(x)} \leq A(a(x)-1) \leq x,$$

which shows that 2^{a-p} is real time I_0- computable.

(4) It follows that the truth value of the condition

$$[b(x), 2^{a(x)-p(x)}] < 2^{a(x)-p(x)-1}$$

can be computed within real time.

Thus δ is real time I_0- computable and the theorem is proved.

§7. Structural characterisation of real time functions

Because the real time computable functions are so important, it is interesting to obtain some sort of structural characterisation of the class. In this final section a programming language is described, relative to a given instruction set I, whose programs compute exactly the real-time I- computable functions. This language is a modified version of the language LOOP first described by Meyer and Ritchie [31].

Suppose I is a given instruction set. The language LOOP(I), which we shall write as LOOP whenever I is implicity understood, is defined recursively as follows:

(1) Each assignment in I, but not X:=X-1, standing by itself is a LOOP program.

(2) If P_1 and P_2 are LOOP programs, and t is some test in I, but not X=0, then the conditional

$$(t \rightarrow P_1, P_2)$$

is a LOOP program. This is equivalent to the Algol:

$$\underline{\text{if}}\ t\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2.$$

(3) If P_1 and P_2 are LOOP programs, then so is

$$P_1;\ P_2$$

The meaning of this program is: first do P_1 then do P_2.

(4) If P is a LOOP program, then so is

<u>loop</u> X P <u>end</u>,

where X denotes the input register. This program is equivalent to

P; P; ...; P (x times)

where x is the initial contents of X, i.e. the input.

(5) By convention, the program E of no instructions is a LOOP program.

For example, the following LOOP (I_0) program computes $\lambda x.[\sqrt{x}]$:

```
A:=1;
loop X
    A:=A-1;
    B:=B+1;
    (A=0 → A:=B; Y:=Y+1; A:=A+2)
end
```

For each $n \geq 1$, we define $LOOP_n(I)$ to be the class of LOOP(I) programs which have a depth of nesting of <u>loop</u> ... <u>end</u> statements no greater than n.

<u>Theorem</u> 5.23 For an arbitrary function f and instruction set I, the following two statements are equivalent:

(i) f is $LOOP_n(I)$ - computable.

(ii) f is computable by some I - program P for which $t_P(x) \leqslant x^n$.

In particular, the $LOOP_1(I)$ programs compute just the real time I - computable functions.

<u>Proof</u>. (i) ⇒ (ii). It is straightforward to translate LOOP(I) programs into I - programs using additional registers to control the number of times a loop is executed. Since the program

<u>loop</u> X <u>loop</u> X ... P <u>end</u> ... <u>end</u> (n times),

where P contains no <u>loop</u> instructions, is equivalent to

P; P; ...; P (x^n times),

and the execution time of P is bounded, the simulation by a I program can be carried out within $\leqslant \lambda x.x^n$ steps. Further details are omitted.

(ii) ⇒ (i). Suppose P is an I-program and $t_P(x) \leqslant x^n$.

Let P have k instructions with labels 1, 2, ..., k and let label 0 denote the termination condition. P is translated into a $LOOP_n(I)$ - program by making use of new registers A, F_0, F_1, ..., F_n

and translating the instructions of P as follows:

(1) All references to X are replaced by references to A, where A does not appear in P.

(2) An instruction ℓ: $a \to m$ of P is translated into the program L_ℓ, where

$$L_\ell = (F_\ell=0 \to E,\ a;\ F_\ell:=0;\ F_m:=1)$$

(3) An instruction ℓ: $t \to m_1, m_2$ of P is translated into L_ℓ, where

$$L_\ell = (F_\ell=0 \to E,\ (t \to F_\ell:=0:F_{m_1}:=1,\ F_\ell:=0;\ F_{m_2}:=1).$$

Let L(P) denote the program $L_1; L_2; \ldots; L_k$ and $L^c(P)$ denote the program L(P); ...; L(P) (c times), where c is such that $t_P(x) \leq cx^n$ for all x. We now claim that the program

A:=0; <u>loop</u> X A:=A+1 <u>end</u>; <u>loop</u> X ... <u>loop</u> X L^c(P) <u>end</u> ... <u>end</u>

is equivalent to P. The first part merely ensures that register A contains the input x. The second part guarantees that the program L(P) is executed cx^n times. The flag registers $F_0, F_1, \ldots, F_k$ appearing in L(P) control the flow of computation to follow that of P.

It is possible that the above characterisation can be used to show that certain functions f, for which $f(x) \leq x$, are not real time I- computable, at least for $I=I_0$. It is worth noting, in this respect, that Tsichritzis [38] has characterised the functions computable by $LOOP_1(I_0)$ - programs which do not use the conditional statement, showing them to be just the simple functions. A function f is <u>simple</u> if there exists numbers x_0 and c and a function d such that

$$f(x+c) = f(x) + d([x, c]) \quad \text{for} \quad x \geq x_0.$$

The simple functions therefore represent a slight generalisation of the ultimately linear functions. The function $\lambda x.[\sqrt{x}]$ is not simple, but is computable by a $LOOP_1$ (I_0) program, thus showing that the presence of conditional statements in LOOP yields a definite increase in capability.

APPENDIX A

MATHEMATICAL BACKGROUND FOR CHAPTER ONE

In Chapter 1, the basic objects under consideration are partial functions. A partial function $f:D \rightarrow D'$ can be extended to a total function by introducing a new element Ω, standing for the undefined, into both D and D', and defining

$$f(x) = \Omega \text{ if } x = \Omega \text{ or } f(x) \text{ is undefined.}$$

A partial ordering $\sqsubseteq$, on functions with the same domain, can then be set up according to the rule

$$f \sqsubseteq g \quad \text{if for all } x, \ f(x) \neq \Omega \text{ implies } f(x)=g(x).$$

It follows from this that

$$f = g \text{ if and only if } f \sqsubseteq g \text{ and } g \sqsubseteq f.$$

If $f:D \rightarrow D$ and D' is a subset of D, we say that f <u>maps</u> D' <u>into</u> D' to mean that for all $x \in D'$, if $f(x) \neq \Omega$, then $f(x) \in D'$.

New partial functions are either defined explicitly, or by making use of one or more of the following operations:

(1) <u>Composition</u>. The composition $f \cdot g$ of two functions f and g is that function which is defined by the equation

$$f \cdot g(x) = f(g(x)) \quad \text{for all } x.$$

Implicit use is made of the fact that composition is associative, and the following deductive rule is used:

if $f \sqsubseteq f'$ and $g \sqsubseteq g'$ then $f \cdot g \sqsubseteq f' \cdot g'$.

(2) Conditional expressions. If P is a predicate, and f and g are functions, then $(P \rightarrow f,g)$ denotes the function defined by

$$
\begin{aligned}
(P \rightarrow f,g)(x) &= f(x) \quad \text{if} \quad P(x) = \text{true}, \\
&= g(x) \quad \text{if} \quad P(x) = \text{false}, \\
&= \Omega \ \text{otherwise}.
\end{aligned}
$$

The following facts about conditional expressions are used:

(a) if $f \sqsubseteq f'$ and $g \sqsubseteq g'$, then $(P \rightarrow f,g) \sqsubseteq (P \rightarrow f',g')$.

(b) $(P \rightarrow h \cdot f,\ h \cdot g) = h \cdot (P \rightarrow f,\ g)$.

(c) $(P \cdot h \rightarrow f \cdot h,\ g \cdot h) = (P \rightarrow f,g) \cdot h$.

(3) Upper bounds. If S is a set of functions with disjoint domains, i.e. if for all $f, g \in S$ and all x, either $f(x) = \Omega$ or $g(x) = \Omega$ or both, then a unique function $\bigsqcup_{f \in S} f$ is defined by the condition

$$
\begin{aligned}
(\bigsqcup_{f \in S} f)(x) &= f(x) \quad \text{if} \quad f(x) \neq \Omega \quad \text{for some} \quad f \in S, \\
&= \Omega \quad \text{otherwise}.
\end{aligned}
$$

If S is a set of functions with disjoint domains, so is the set $\{h \cdot f : f \in S\}$ for any function h, and we have

$$h \cdot (\bigsqcup_{f \in S} f) = \bigsqcup_{f \in S} h \cdot f .$$

It is of course possible to take upper bounds of a more general type of set, but we shall not need to do so.

(4) Recursion. If $E(f)$ is some functional expression involving the function letter f, then a unique function f is specified by the conditions:

(i) $f = E(f)$,

(ii) for all functions g, if $g = E(g)$, then $f \sqsubseteq g$.

In such a case, we say that f is recursively defined by the equation $f = E(f)$.

In order to prove statements about recursively defined functions, use is made of the following induction principle, which is justified in Morris [42] or Manna [41]:

to prove that a certain statement $S(f)$ holds, it is sufficient to prove by induction that

$$S(f_k) \text{ holds for all integers } k \geq 0,$$

where the sequence $f_0, f_1, \ldots$ of functions is defined by

$$f_0(x) = \Omega \quad \text{for all } x,$$
$$f_{k+1} = E(f_k).$$

This principle is not applicable to all possible statements involving f, as can be seen from consideration of the statement

$(\exists x)(f(x) = \Omega)$. However, all statements which are assertions about inclusion, e.g. f   g, can be proved in this way, and these are the only ones needed.

APPENDIX B

LIST OF SPECIAL SYMBOLS

The following list of special symbols used frequently in the text, supplements that given in Appendix A.

Symbol	Meaning
f_P	function computed by program P (section 2.1)
t_P	running time of program P (section 2.1)
N	the set of natural numbers $\{0,1,\ldots\}$
$[x]$	the integral part of x
$[x, y]$	the (non-negative) remainder when x is divided by y.
$\|x\|$	the absolute value of x.
$\lambda x.f(x)$	Church's Lambda notation for denoting functions.
$f \trianglelefteq g$, $f \triangleleft g$, $f \sim g$	defined in Section 2.3
$f \cdot g$, $f+g$, f^*	defined in Section 2.6-2.7.

Symbol	Meaning
$\|\|S\|\|$, Size(S)	two notations for number of elements in a finite set
$\exists x \ldots.$	there exists an x such that ...
Σf	denotes function h, where $h(x) = f(0) + f(1) + \ldots + f(x)$.
Δf	denotes function h, where $h(0) = f(0)$, $h(x+1) = f(x+1)-f(x)$
$\subseteq$ $\cup$ $\bigcup$	set theoretic inclusion and union notation.

REFERENCES

[1] AHO, A.V. and ULLMAN, J.D., Transformations on straight-line programs, 2nd ACM Symposium on Theory of Computing 136-148, (1970).

[2] ALLEN, F.E., Program Optimisation, Ann. Rev. in Automatic Programming, 5, pp.239-307 (1969).

[3] ARBIB, M.A., Theories of Abstract Automata, Prentice-Hall (1969).

[4] BIRD, R.S., On transformations of programs, J. Computer and System Sciences to appear (1973).

[5] BIRD, R.S., Computational complexity on register machines (Part 1 - space requirements), Tech. Report, 1CS1 252, Institute of Computer Science (1970).

[6] BIRD, R.S., Space restricted computations on fixed register machines Tech. Report, ICS1 253, Institute of Computer Science (1970).

[7] BLUM, M., A machine-independent theory of the complexity of recursive functions, JACM 14, 322-336 (1967).

[8] BLUM, M., FLOYD, R.W., PRATT, V., RIVEST, R.L., TARJAN, R.E., Time bounds for selection, Tech. Report CS-349 Comp. Sci. Dept. Stanford (1973).

[9] CLEAVE, J.P., A hierarchy of primitive recursive functions, Zeitschr. f. math. Logitz und Grundlagen d. Math 9, pp.331-345 (1963).

[10] COBHAM, A., Functional equations for register machines, Proc. Hawaii Int. Conf. on System Sciences 10-13 (1968).

[11] COLLINS, G.E., The computing time of the Euclidean algorithm, Tech. Report. CS-331 Comp. Sci. Dept. Stanford (1973).

[12] CONSTABLE, R.L. and BORODIN, A.B., On the efficiency of programs in subrecursive formalisms, Tech. Report 70-54 Department of Comp. Sci. Cornell (1970).

[13] COOK, S.A. and AANDERAA, O., On the minimum computation time of functions, Trans. Amer. Math. Soc. 142, 291-314 (1969).

[14] ELGOT, C.C. and ROBINSON A., Random-access stored-program machines, J. ACM 11, pp.365-399 (1964).

[15] FISCHER, P.C., Turing machines with a schedule to keep, Inf. and Control.11, pp.138-146 (1967).

[16] FISCHER, P.C., MEYER, A.R. and ROSENBERG, A.L., Counter machines and counter languages, Math. Systems Theory 2, pp.265-283, (1968).

[17] FISCHER, P.C. MEYER, A.R. and ROSENBERG, A.L., Time-restricted sequence generation, J. Computer and System Sciences 4 50-73 (1970).

[18] GRZEGROCZYK, A., Some classes of recursive functions, Rozprawy Matematczyne, 4 pp.1-45 (1953).

[19] HARTMANIS, J., Computational complexity of random access stored program machines, Tech. Report. 70-70, Dept of Comp. Sci. Cornell (1970).

[20] HARTMANIS, J., Tape reversal bounded Turing machine computations, J. Computer and System Sciences 2, pp.117-135 (1968).

[21] HARTMANIS, J. and STEARNS, R.E., On the computational complexity of algorithms, Trans. Amer. Math. Soc.117, pp.285-306 (1965).

[22] HENNIE, F.C., On-line Turing machine computations, IEEE Trans EC-15, 35-44 (1966).

[23] HOPCROFT, J.E. and ULLMAN, J.D., Relation between time and tape complexities, J. ACM 15, 414-427 (1968).

[24] IRELAND, M.I. and FISCHER, P.C., A Bibliography on Computational Complexity, Research Report CSRR 2028, Dept. of Applied Analysis and Comp. Sci. Waterloo (1970).

[25] KNUTH, D.E., Mathematical analysis of algorithms, Technical Report CS-206 Comp. Sci. Dept. Stanford (1971).

[26] KNUTH, D.E. The Art of Computer Programming, Vol.1 Addison-Wesley (1968).

[27] KNUTH, D.E. The Art of Computer Programming, Vol.11 Addison-Wesley (1969).

[28] MARILL, T., Computational chains and the simplification of computer programs, IRE Trans. EC 11 173-180 (1962).

[29] McCARTHY, J., Towards a mathematical science of computation., Proc. IFIP Congress 1962, pp.21-28, North Holland (1963).

[30] McCREIGHT, E.M., Classes of computable functions defined by bounds on computation, Thesis, Dept. of Comp. Sci. Carnegie-Mellon (1969).

[31] MEYER, A.R. and RITCHIE, D.M., The complexity of loop programs, Proc. 22nd Nat. Conf. ACM 465-469 (1967).

[32] MILNER, R., An algebraic definition of simulation between programs, Tech. Report. CS-205 Comp. Sci. Dept. Stanford (1971).

[33] MINSKY, M.L., Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines, Ann. Math. 74, 437-454 (1961).

[34] RABIN, M.O. and SCOTT, D., Finite automata and their decision problems, IBM Journal of Research and Development 3, 114-125 (1959).

[35] SCOTT, D., Some definitional suggestions for Automata theory, J. Computer and System Sciences 1, 187-212 (1967).

[36] SHEPHERDSON, J.C. and STURGIS, H.E., Computability of recursive functions, J. ACM 10, 217-255 (1963).

[37] STRASSEN, V., Gaussian elimination is not optimal, Numerische Mathematik, 13, 354-356 (1969).

[38] TSICHRITZIS, D., The equivalence problem of simple programs, J. ACM 17, 4, pp.729-738 (1970).

[39] YOUNGER, D.H., Recognition and passing of context free languages in time n^3, Information and Control, 10 189-208 (1967).

[40] YAMADA, H., Real-time computation and recursive functions not real-time computable, IRE Trans. EC-11 pp.753-760 (1962).

[41] MANNA, Z., and VUILLEMIN, J., Fixpoint approach to the theory of computation, CACM 15 p.528-536 (1972).

[42] MORRIS, J.H. Jr., Another recursion induction principle, CACM pp.351-354 (1971).

LIST OF SUPPLEMENTARY EVIDENCE

1. Integers with given initial digits, Am. Math. Monthly 79,1971, 367-370.

2. A note on definition by cases, Zeitschr. f. math. Logik und Grundlagen d. Math. Bd. 19 207-208 (1973).

3. On transformations of programs, Journal of Comp. and System Science (to appear, 1974)

4. non-recursive functionals, Zeitschr. f. math. Logik und Grundlagen d. Math. (to appear, 1974)

5. Speeding up programs, Computer Journal (to appear, 1974) (copy not yet available)

Zeitschr. f. math. Logik und Grundlagen d. Math.
Bd. 19, S. 207–208 (1973)

A NOTE ON DEFINITION BY CASES

by RICHARD BIRD in Reading, Berkshire (Great Britain)

The usual proofs that the function h defined by the conditions

$$h(x) = \begin{cases} f(x) & \text{if } P(x), \\ g(x) & \text{otherwise,} \end{cases}$$

can also be defined using the operations of primitive recursion and substitution in terms of f, g and the characteristic function of P, are incorrect if it is not assumed that f and g are total functions. According to the natural interpretation of the above conditions, the value of $h(x)$ is defined to be $f(x)$ if $P(x)$ is true (whether or not the value of $g(x)$ is defined), and $g(x)$ if $P(x)$ is false (whether or not the value of $f(x)$ is defined). The most common translation into a primitive recursive definition of this function is to define h_1 by

$$h_1(x) = \overline{sg}(p(x)) \times f(x) + p(x) \times g(x). \tag{1}$$

Here, $\overline{sg}$ is the primitive recursive function that satisfies

$$\overline{sg}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0, \end{cases}$$

and p is the characteristic function of P, which takes the values 0 or 1 depending on whether $P(x)$ is true or not. However h_1, which is primitive recursive in f, g, and p, only coincides with h if f and g are assumed total. This is because in the semantics of the operation of substitution which specifies k by

$$k(x) = r(s_1(x), s_2(x), \ldots, s_n(x)),$$

$k(x)$ is only defined if all of $s_1(x), s_2(x), \ldots, s_n(x)$ are defined. It follows by (1), that if either $f(x)$ or $g(x)$ is undefined for some x, so is $h_1(x)$, and this is not necessarily true of the original function h.

As another attempt at giving a correct primitive recursive definition of h, we can define A by

$$A(x, 0) = f(x), \quad A(x, y+1) = g(x), \tag{2}$$

which represents a degenerate case of the operation of primitive recursion, and set

$$h_2(x) = A(x, p(x)).$$

However this attempt also fails because of the way the operation of primitive recursion is defined. The equations (2) are not to be regarded as an instance of definition by cases (otherwise we would be admitting definition by cases as a primitive operation), but rather as the special case of

$$A(x, 0) = f(x), \quad A(x, y+1) = k(x, y, A(x, y)) \tag{3}$$

in which $k(x, y, z) = g(x)$.

The values of $A(x, y)$ are defined *inductively* from the equations (3). It follows that if $f(x)$ is undefined so are the values of $A(x, y)$ for every y. Therefore, according to the definition of A by (2), if $f(x)$ is undefined so is $h_2(x)$, and once again this is not necessarily true of the original function h.

However, by using a not too obvious trick, it is possible to give a proper primitive recursive definition of h. We define two functions A and B by primitive recursion as follows.

$$B(x, 0) = x, \quad B(x, y + 1) = f(B(x, y))$$

and

$$A(x, y, 0) = B(x, y), \quad A(x, y, z + 1) = g(A(x, y, z)).$$

The function h_3 which is defined by

$$h_3(x) = A(x, \overline{sg}(p(x)), p(x))$$

can now be proved to coincide everywhere with h. There are three cases to be considered:

(a) $p(x)$ is undefined. In this case, $h_3(x)$ is undefined, and so $h_3(x) = h(x)$.

(b) $p(x) = 0$. According to the inductive definition of A and B,

$$h_3(x) = A(x, 1, 0) = B(x, 1) = f(B(x, 0)) = f(x)$$

so that $h_3(x) = h(x)$.

(c) $p(x) = 1$. Again, according to the inductive definition of A and B,

$$h_3(x) = A(x, 0, 1) = g(A(x, 0, 0)) = g(B(x, 0)) = g(x)$$

so that $h_3(x) = h(x)$ and the proof is complete.

It may be mentioned that the author arrived at the difficulty mentioned in this note, while attempting the apparently straightforward task of arithmetising a class of algorithms in terms of the functions corresponding to the atomic steps of the calculation. These unspecified functions were not assumed to be total. The usual arithmetisation procedure does assume that the atomic steps of the algorithm correspond to total functions (as in the case of Turing machines), and so the primitive recursive definition of the next-step function, which implicitly uses definition by cases, causes no trouble. That such an artificial trick has to be employed for the more general situation, argues the case for adopting a recursive function formalism based directly on conditional expressions such as that given by McCarthy in [1].

Reference

[1] McCarthy, J., A basis for a mathematical theory of computation. In: Computer Programming and Formal Systems (Editors: P. Braffort and D. Hirschberg,) North-Holland Publ. Comp., Amsterdam 1963.

(Eingegangen am 14. Februar 1972)

Reprinted from the AMERICAN MATHEMATICAL MONTHLY
Vol. 79, No. 4, April 1972
pp. 367–370

INTEGERS WITH GIVEN INITIAL DIGITS

R. S. BIRD, Institute of Computer Science, London, England

Consider the following situation. Two mathematicians called X and Y are talking, and X announces that he has just computed a large prime, which he begins to recite to Y digit by digit. There are two possible responses open to Y. He can either wait until X has finished and then check the assertion, or he can interrupt X at some point in the recitation with the information that no prime can begin with those digits. The problems we are interested in are these:

(1) Assuming that Y knows his primes, can we prove that there is no sequence of digits that allows Y to interrupt X?

(2) Can the same be said about other sets of integers such as the squares, the factorial numbers, or the powers of 2?

To make things precise, suppose that S is an (infinite) set of positive integers. We shall say that S is **extendable in base** b if for each integer $x \geqq 1$, there are integers y and n, with $y < b^n$, such that $x b^n + y$ is in S.

If S is extendable in base b and consists of the integers $s_0, s_1, \cdots$, then

(1) for each integer $x \geqq 1$, there are integers m and n such that $b^n x \leqq s_m \leqq b^n(x+1)$.

Conversely, (1) implies that S is extendable in base b, as we can take y to be $s_m - b^n x$.

If we use the prime number theorem, the proof of the extendability of P, the set of primes, in every base is fairly easy.

THEOREM 1. *Let $\pi_S(n)$ be the number of members of S less than n. A sufficient*

condition for S to be extendable in every base is that if $\pi_S(n)/\pi_S(\lambda n) \to \theta(\lambda)$ for all real λ satisfying $1 \leqq \lambda \leqq 2$, then $\theta(\lambda) = 1$ only if $\lambda = 1$.

Proof. Assume that S is not extendable in some base b. Then by (1), there exists an x such that $\pi_S(b^n(x+1)) = \pi_S(b^n x)$ for all n. Let $\lambda_0 = (x+1)/x$ (whence $1 < \lambda \leqq 2$), and $m_n = b^n x$, so that

$$\pi_S(m_n)/\pi_S(\lambda_0 m_n) = 1 \text{ for all } m_n.$$

It follows that if $\pi_S(n)/\pi_S(\lambda n) \to \theta(\lambda)$, then $\theta(\lambda_0) = 1$ for some $\lambda_0 \neq 1$, which contradicts the hypothesis of the theorem.

Now the prime number theorem asserts that $\pi_P(n) \sim n/\log n$, whence $\pi_P(n)/\pi_P(\lambda n) \to 1/\lambda$, which is 1 only if $\lambda = 1$. Therefore P is extendable in every base. A similar argument shows that for each k, the set of kth powers is extendable in every base. However, in other interesting cases the ratio $\pi_S(n)/\pi_S(\lambda n)$ fails to converge, or converges to 1, for all λ, and a sharper condition is needed. The following theorem effectively characterises the extendable sets of numbers and reduces the question to a problem of Diophantine Approximation.

THEOREM 2. *A necessary and sufficient condition for the set $S = \{s_0, s_1, \cdots\}$ of positive integers to be extendable in base b is that the set of fractional parts of the real numbers $\log_b s_0$, $\log_b s_1, \cdots$ be dense in the unit interval.*

Proof. In the following, all logarithms are taken to the base b.

(a) *Necessity.* Suppose S is extendable in base b, so that condition (1) holds. Take logarithms and write $u_m = \log s_m$, $\alpha = \log x$, and $\delta(\alpha) = \log(1 + 1/x)$. Then, by assumption, for each α of the form $\log x$, there exist integers m and n such that

$$0 \leqq u_m - n - \alpha < \delta(\alpha). \tag{2}$$

If we write (z) for the fractional part and $[z]$ for the integral part of the real number z, then (2) can be expanded to

$$(\alpha) - (u_m) \leqq [u_m] - [\alpha] - n < \delta(\alpha) + (\alpha) - (u_m) \leqq \delta(\alpha) + (\alpha). \tag{3}$$

Since $(\alpha) - (u_m) > -1$, and $\delta(\alpha) + (\alpha) = \log(x+1) - [\log x]$, which has a maximum value of 1 (obtained when x is of the form $b^k - 1$, for some k), the above inequalities imply that $[u_m] - [\alpha] = n$, whence (2) can be simplified to

$$0 \leqq (u_m) - (\alpha) < \delta(\alpha). \tag{4}$$

Let ε be any positive real number, and x any integer. Define $\alpha_k = \log b^k x$. Clearly $(\alpha_k) = (\log x)$, for each k. Let n be any integer such that

$$\delta(\alpha_n) = \log(1 + 1/b^n x) < \varepsilon.$$

For such an n, there exists, by (4), an m such that $0 \leqq (u_m) - (\alpha_n) < \delta(\alpha_n)$; i.e., an m satisfying

$$0 \leqq (u_m) - (\log x) < \varepsilon. \tag{5}$$

Given ε, we can also find an integer x_0 such that $0 \neq (\log x_0) < \varepsilon$. The sequence of points

$$(\log x_0),\ (2\log x_0),\ (3\log x_0), \cdots$$

therefore marks a chain across the interval $(0,1)$, where the distance between consecutive points is less than ε. Hence, given any θ in $(0,1)$, one can find a number $x = x_0^k$, for some k, such that $0 \leqq \theta - (\log x) < \varepsilon$. It follows, using (5), that a number m exists such that

$$|\theta - (u_m)| < \varepsilon. \tag{6}$$

Since θ and ε were arbitrary, (6) is just the condition for the set of fractional parts of $\log s_0, \log s_1, \cdots$ to be dense in the unit interval.

(b) *Sufficiency.* Suppose that (6) holds for arbitrary θ and ε. Let x be any positive integer, and take $\theta = (\log x)$. Then there must be an infinite number of integers m such that

$$0 \leqq (u_m) - (\log x) < \varepsilon,$$

for otherwise, we can construct an interval to the right of $(\log x)$ that contains no point of the form (u_m), contrary to assumption. If we take an $\varepsilon < \delta(\alpha)$ and an m such that $u_m \geqq \alpha$, where $\alpha = \log x$, then $n = [u_m] - [\alpha]$ is a non-negative integer that satisfies condition (1). Hence S is extendable in base b.

The following lemma is based on a proof by J. W. S. Cassels [1].

LEMMA. *Let U be a sequence $u_0, u_1, \cdots$ of real numbers of increasing size. A sufficient condition for the fractional parts of U to be dense in the unit interval is given by either*

(i) $\Delta u_n \to \theta$, *where θ is either irrational or zero, or*
(ii) $\Delta u_n \to \infty$, *and* $\Delta^2 u_n \to 0$. *(By definition, $\Delta u_n = u_{n+1} - u_n$.)*

Proof. To begin with, assume that $\Delta u_n \to 0$, and let ϕ be an arbitrary real number. Since $u_n \to \infty$, it is easy to verify that, given any $\varepsilon > 0$ and any integer m, there exist integers p and n_0 such that

$$|u_n - \phi - p| < \varepsilon \text{ for all } n \text{ satisfying } n_0 \leqq n \leqq n_0 + m. \tag{7}$$

In particular, it follows that the fractional parts of U are dense in the unit interval. Actually (7) asserts slightly more, and this is used below.

In the case $\Delta u_n \to \infty$ and $\Delta^2 u_n \to 0$, it follows from (7) (with u_n replaced by Δu_n) that, given $\varepsilon > 0$ and m, there exist integers p and n_0 such that

$$|\Delta u_n - \phi - p| < \varepsilon/m \text{ for all } n \text{ satisfying } n_0 \leqq n \leqq n_0 + m.$$

The above statement (with $p = 0$ and $\phi = \theta$) also holds true in the third case $\Delta u_n \to \theta$, so that in either case, given ε and m, there exist p and n_0, and some irrational θ, such that

$$(8) \qquad \left| u_{n_0+k} - u_{n_0} - k\theta - kp \right| \leqq \sum_{r=0}^{k-1} \left| \Delta u_{n_0+r} - \theta - p \right| < \varepsilon$$

provided that $0 \leqq k \leqq m$.

Next, one version of Kronecker's theorem asserts that if θ is irrational, then given $\varepsilon > 0$, there is an n_1, such that for any real α there exist integers q and k_0, with $0 \leqq k_0 < n_1$, such that $\left| k_0\theta - \alpha - q \right| < \varepsilon$.

In substance, this says that the set of points $\{(\theta), (2\theta), \cdots\}$ is dense in the unit interval. For a proof see Cassels [2], or Hardy [3].

Now, if in (8) we take $m = n_1$, let β be arbitrary, and set $\alpha = \beta - u_{n_0}$, then Kronecker's theorem asserts the existence of integers q and k_0, with $k_0 \leqq n_1$, such that

$$\left| k_0\theta - \beta + u_{n_0} - q \right| < \varepsilon.$$

Setting $s = k_0 p + q$, it follows that

$$\left| u_{n_0+k_0} - \beta - s \right| \leqq \left| u_{n_0+k_0} - u_{n_0} - k_0 p - k_0\theta \right| + \left| k_0\theta - \beta + u_{n_0} - q \right| < 2\varepsilon.$$

Since β and ε were arbitrary and s is an integer, the lemma is proved.

THEOREM 3. *A sufficient condition for the set $S = \{s_0, s_1, \cdots\}$ of positive integers to be extendable in base b is that*

either (i) $s_{n+1}/s_n \to \theta$, *where* $\theta = 1$ *or* θ *is not a rational power of* b,

or (ii) $s_{n+1}/s_n \to \infty$ *and* $s_n s_{n+2}/s_{n+1}^2 \to 1$.

The proof is a straightforward consequence of the lemma and Theorem 2. The second condition is independent of b, and so asserts the extendability of S in every base.

Now we can show, for example, that the set of powers of a given integer p is extendable in base b, provided that p is not a power of b, as the first condition is satisfied. Also, the set of factorial numbers is extendable in every base as $(n+1)!/n! = n+1 \to \infty$ and $n!(n+2)!/(n+1)!^2 = (n+2)/(n+1) \to 1$.

References

1. J. W. S. Cassels, Personal Communication.

2. J. W. S. Cassels, An Introduction to Diophantine Approximation, Cambridge Tracts in Math., No. 45, C. U. P., England, 1965.

3. G. H. Hardy and E. M. Wright, An Introduction to the Theory of Numbers, 4th Edition, Clarendon Press, Oxford, England, 1960.