ADDENDUM

I thank the examiners for drawing two obscurities in the thesis to my attention.

In the proof of Theorem 1.4 (see page 11, line -4), for $x > 0$, the Turing machine $W(a,r,x)$ attempts to calculate

$$\min \{y \mid \varphi_a(x) \leq y \text{ , } W(a,r,x-1) \leq y \text{ , and } \forall i \leq x \ \Phi_i(x) \leq y \text{ or } \Phi_i(x) > \varphi_r(x,y)\}$$

in the obvious way. If this calculation should diverge, there is no problem in the proof, for then it is trivially true that $W(a,r,x) \geq \varphi_a(x)$ and $\forall i \leq x \ \Phi_i(x) \leq W(a,r,x)$.

In the proof of Theorem 5.7 (see page 65, line -7), the algorithm for the Turing machine $W(i,t,n,x)$ is stated too briefly. The algorithm intended is :

For $x \leq n$, calculate and output $\varphi_i(x)$.

If $x > n$, then begin cycling between calculating $\Phi_i(x)$, $\varphi_t(x)$ and $\Phi_t(x)$. As soon as the calculation for $\varphi_t(x)$ converges, test (a) $\Phi_i(x) > \varphi_t(x)$. As soon as the calculation for $\Phi_i(x)$ or for $\Phi_t(x)$ converges, test (b) $\Phi_i(x) > \Phi_t(x)$. If a negative result is obtained for test (a) or for test (b), then at once calculate and output $\varphi_i(x)$. If positive results are obtained for both tests (a) and (b), then output 0.

With this expanded version of the algorithm, the steps in the proof of Theorem 5.7 should be quite clear. In particular, note that if $\varphi_t(x)$ diverges, then $W(i,t,n,x) = \varphi_i(x)$.

PROVABLE CONDITIONS IN COMPUTATIONAL COMPLEXITY THEORY

by

Daryel Sachse-Åkerlind

A thesis submitted for the degree of
Doctor of Philosophy at the
Australian National University
April 1983.

THE AUSTRALIAN NATIONAL UNIVERSITY LIBRARY

Except where explicitly stated otherwise, all the results in this thesis are my own.

D. Sachse-Åkerlind

Acknowledgements

I thank my supervisor, Professor Richard Brent,
for his support throughout my post-graduate studies, and
I thank Anna Zalucki for her superb job of typing my thesis.

ABSTRACT

Computational complexity measures and indexings of algorithms are considered within a formal axiomatic system S. S is meant to mimic the formal system within which the study of computational complexity is (implicitly) carried out - so, for example, S can be a conventional axiomatization of set theory.

The main thrust of the thesis is that for many natural questions about the complexity of algorithms, what can be formally proved falls unpleasantly short of what is actually true.

We consider abstract Blum measures over indexings of the partial recursive functions. Our results fall into three categories.

First we consider complexity questions involving some arbitrary given partial recursive function f. Associated with f will be an algorithm used to define f. Before any other algorithm can be admitted as a means of calculating f, it must be proved equivalent to our defining algorithm for f. The requirement of being provably equivalent defines an equivalence relation on the set of all algorithms. We call the equivalence classes provable equivalence classes. We show that for natural complexity questions about f, what can be proved about f depends on the provable equivalence class to which the defining algorithm for f belongs.

Having had our attention focussed on provable equivalence classes, we next investigate the relationship between provable equivalence and the complexity of algorithms. This relationship is complex and not readily summarized, but it is closely involved with

provable containment between the domains over which algorithms are defined. A general conclusion we can draw is that as the difference between the complexities of algorithms increases, what can be proved about the relationship between the algorithms decreases.

Finally, we consider provable analogues of complexity classes. Two possible definitions for provable complexity classes are proposed, based on different bounding conditions - (1) the usual almost-everywhere bounding used to define complexity classes, and (2) almost-everywhere bounding with the additional requirement that an explicit starting-point for the bounding be given. Various results are developed relating the two types of provable complexity classes to each other and to ordinary complexity classes. In particular, we show that for infinitely many recursive functions f the provable complexity class of f defined using bounding conditions (1) is equal to the ordinary complexity class of f and is strictly larger than the provable complexity class of f defined using bounding conditions (2).

TABLE OF CONTENTS

INTRODUCTION I

1. PRELIMINARIES 1

1.1 THE FORMAL SYSTEM S 1

1.2 BLUM MEASURES 2

1.3 VARIOUS DEFINITIONS 7

1.4 SOME USEFUL RESULTS 10

2. SURVEY OF THE FIELD 15

3. ANOMALOUS ALGORITHMS AND PROVABLE COMPLEXITY PROPERTIES 21

3.1 INTRODUCTION 21

3.2 RESULTS 23

4. PROVABLE EQUIVALENCE AND COMPLEXITY OF ALGORITHMS 37

4.1 INTRODUCTION 37

4.2 RESULTS 38

5. COMPLEXITY CLASSES AND PROVABLE COMPLEXITY CLASSES 52

5.1 INTRODUCTION 52

5.2 RESULTS 56

CONCLUSION 78

BIBLIOGRAPHY 81

INTRODUCTION

The study of computational complexity is carried out within a formal axiomatic system. All work done should, in principle, be able to be encoded into a formal system, such as a conventional axiomatization of set theory.

The observation that a computer scientist studying the behaviour of algorithms is dealing with formally provable properties opens the door to a wealth of new insights into the nature of problems in computational complexity theory. As Hartmanis says in [9], "the results about complexity of computations change quite radically if we consider only properties of computations which can be proven formally".

There are several motivations behind work on provable conditions in computational complexity. First, there is the growing interest in computer science in proving properties of programs. Then, our continued failure to solve certain outstanding problems in complexity theory - such as the famous P = NP ? problem - has raised the suspicion that the answers to such problems may be independent of the axioms of set theory. (See [11]) Third, a precedent for the study of provable conditions in complexity theory has been set by earlier studies of provable conditions in the theory of recursive functions. Finally, the intimate connection between formal logical systems and computation, the ability to code a formal system into a machine to generate theorems, makes the study of the behaviour of algorithms an obvious subject for an enquiry into the limitations of what can be formally proved.

We investigate provable conditions in computational complexity theory by introducing a formal system S of sufficient power to allow encoding of all the standard concepts and reasoning used in the study of computational complexity. S is to mimic the formal system within which a computer scientist studying algorithms works. So, for example, S could be a conventional axiomatization of set theory.

We can now study how results in computational complexity change when we demand that certain conditions be formally provable in S; we can investigate discrepancies between what is true about algorithms and what is formally provable about them in S. The conclusions we draw will apply directly to the work done by the computer scientist studying algorithms.

The results in this thesis are not restricted to any particular complexity measures. We deal with partial recursive functions on the natural numbers $\mathbb{N}$ and with abstract Blum measures of complexity.

In Chapter 1 we set up the preliminaries. We describe the formal system S, establish definitions and terminology, and present a number of preliminary theorems which will be useful in proving our later results.

Chapter 2 is a survey of the work that has already been done on provable conditions in computational complexity theory.

In Chapters 3,4 and 5 we present our results. Each chapter has an introduction which motivates the work of that chapter and previews the results obtained.

Chapter 3 concerns the problem of establishing complexity properties for some (arbitrary) given partial recursive function f.

We consider three natural questions involving the complexity properties of f, and we show in each case that what can be proved about f depends on the algorithm initially used to define f. In fact, for each complexity property considered, we can explicitly construct possible defining algorithms for f using which it cannot be proved that f has the property - even when f does have the property.

The work in Chapter 3 makes great use of the notion of provable equivalence of algorithms. Two algorithms are said to be provably equivalent if they can be proved to be equivalent. Algorithms that are provably equivalent are related in their complexity. In Chapter 4 we develop a number of results relating provable equivalence to the computational complexity of algorithms.

In Chapter 5 we consider provable analogues of complexity classes. Two possible definitions for provable complexity classes are proposed, based on different bounding conditions - (1) the usual almost-everywhere bounding used to define complexity classes, and (2) almost-everywhere bounding with the additional requirement that an explicit starting-point for the bounding be given. We develop various results relating the two types of provable complexity classes to each other and to ordinary complexity classes. In particular, we show that for infinitely many recursive functions f the provable complexity class of f defined using bounding conditions (1) is equal to the ordinary complexity class of f and is strictly larger than the provable complexity class of f defined using bounding conditions (2).

Except in Chapter 2, where we adopt a special convention to aid reference, results are numbered consecutively in each chapter. Thus, Theorem 3.4 denotes the fourth result in Chapter 3.

Our terminology for partial recursive functions follows that of [15]. Our notation and terminology for complexity measures follow that of [3] and [10].

1 PRELIMINARIES

In this chapter we establish the basic definitions and terminology, and present some preliminary theorems.

1.1 THE FORMAL SYSTEM S

The idea behind the definition of S is to make S sufficiently powerful that all of the standard concepts and reasoning used in the study of computational complexity can be reproduced in S . It turns out that there is no need to become involved in the details of the formal description of S , and so we shall not dwell on them.

Let S be a formal axiomatic system containing a conventional axiomatization of Elementary Number Theory (ENT) and enough of the power of axiomatic set theory to enable formalization of straightforward mathematical argument[1]. We further assume that S is sound for ENT , that is, there is no formula which is a theorem of S and which is false under the standard interpretation of ENT . S is to be fixed but arbitrary within these constraints.

In the usual way, through number-theoretic predicates, we can encode into S any of the standard enumerations of the Turing machines, and can carry out all of the standard reasoning about them[2]. Then, since S is sound for ENT , any theorem of S which is (under our intended interpretation) a statement about Turing machines will also be true.

---

[1] An example of such a system is first-order Peano Arithmetic. By ENT we mean the theory of number-theoretic predicates expressible in first-order arithmetic.

[2] This encoding process is explained in most texts on logic and recursiveness. See, for example, [4] or [12].

S will have its own formal language. Rather than concern ourselves with the details of such a language, we shall take advantage of the intended interpretation of it, and enclose in quotation marks those informal statements which are to be understood as having been written out in the formal language of S . Statements about partial recursive functions will appear in S as statements about Turing machines. Thus, when we write "$\forall x\ f(x) = x$" , it must be understood that f is being referred to in S via a specific Turing machine representation.

We use the usual notation to denote theorems of S . Thus, $\vdash$ "$\forall x\ f(x) = x$" indicates that the statement in quotation marks, when translated into the formal language of S , is a theorem of S . Conversely, $\nvdash$ "$\forall x\ f(x) = x$" indicates that the statement is not a theorem of S .

Finally, note that, since S is a formal axiomatic system, the theorems of S are recursively enumerable. It follows that the theorems of S can be generated primitive recursively. (See [15].)

1.2 BLUM MEASURES

An abstract Blum measure consists of two parts : an acceptable Gödel-numbering φ (which indexes the partial recursive functions) and a Blum measure Φ for φ . (See [3].) For the usual reasoning about φ and Φ to be reproducible in S , the defining properties of φ and Φ must be theorems of S , in which case we call φ a provably acceptable Gödel-numbering and Φ a provable Blum measure for φ . In this section we present a proper definition of a provably acceptable Gödel-numbering and a provable Blum measure.

Throughout the following, let $\{M_i \mid i \in \mathbb{N}\}$ be some standard enumeration of the Turing machines with some appropriate input-output conventions. For convenience, we use M_i to denote both the i'th machine and the function of one variable that it defines.

PROVABLY ACCEPTABLE GÖDEL-NUMBERING

Let φ be a partial recursive function of two variables. For each i, we write the function $\lambda x\, \varphi(i,x)$ as φ_i. Then φ defines an effective enumeration of a set $\{\varphi_i \mid i \in \mathbb{N}\}$ of partial recursive functions of one variable. We extend this to functions of more than one variable by associating with φ a pairing function, that is, a recursive bijection $<,> : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. By convention, we avoid explicit reference to $<,>$ and write $\varphi_i(<x,y>)$ as $\varphi_i(x,y)$. We write $\varphi_i(<<x,y>,z>)$ as $\varphi_i(x,y,z)$, and so on.

We can now present the definition of an acceptable Gödel-numbering as follows:

DEFINITION φ is an acceptable Gödel-numbering if

(i) $\forall i\ \exists j\ M_i = \varphi_j$,

(ii) for some recursive function s,

$$\forall i\ \forall x\ \forall y\ \ \varphi_i(x,y) = \varphi_{s(i,x)}(y),$$

(iii) for some index v,

$$\forall i\ \forall x\ \ \varphi_v(i,x) = \varphi_i(x).$$

In the definition, (i) states that φ enumerates all the partial recursive functions, (ii) states that Kleene's Iteration Theorem [12] holds for φ, and (iii) states that there is a universal-machine index for φ. For a full discussion of the definition, see [14].

To be able to reproduce in S all the usual reasoning about the φ_i's , the basic properties of φ must be theorems of S . For purposes of encoding into S , φ will be represented by a particular Turing machine for calculating it. Similarly for $<,>$.

We now define a provable analogue of an acceptable Gödel-numbering.

DEFINITION φ is a provably acceptable Gödel-numbering if

(i) $\vdash$ "$<,>$ is a total, one-one, onto function",

(ii) $\forall i \; \exists j \vdash$ "$M_i = \varphi_j$" ,

(iii) for some recursive function s

$\vdash$ "s is a total function and

$\forall i \; \forall x \; \forall y \;\; \varphi_i(x,y) = \varphi_{s(i,x)}(y)$" ,

(iv) for some index v

$\vdash$ "$\forall i \; \forall x \;\; \varphi_v(i,x) = \varphi_i(x)$" .

The conditions of the definition correspond to what we would establish in determining that φ is an acceptable Gödel-numbering.

(i) We can prove that $<,>$ is a pairing function.

(ii) Given any Turing machine M_i , there is an index j for which we can prove that φ_j calculates the same partial function as M_i . (Actually, we would find a uniform procedure to produce φ_j from M_i , but that is not needed. See Theorem 1.1.)

(iii) For some function s , we can show that the Iteration Theorem holds with s .

(iv) There is an index v which we can show to be a universal-machine index.

PROVABLE BLUM MEASURE

Let φ be a provably acceptable Gödel-numbering. Let Φ be a collection $\{\Phi_i \mid i \in \mathbb{N}\}$ of partial recursive functions.

DEFINITION Φ is a Blum measure for φ if

(i) for some index e, φ_e is a recursive function and

$$\forall i\ \forall x\ \forall n\ \ \varphi_e(i,x,n) = 1 \Leftrightarrow \Phi_i(x) = n\ ,$$

(ii) $\forall i$ domain φ_i = domain Φ_i .

In the definition, (i) says that the relation $\Phi_i(x) = n$ is recursive in i,x,n, and (ii) says that φ_i and Φ_i are defined on precisely the same set of inputs. For a discussion of the definition, see [3].

To be able to reproduce in S all the usual reasoning about the measure Φ, the basic properties of Φ must be theorems of S.

We now define a provable analogue of a Blum measure.

DEFINITION Φ is a provable Blum measure for φ if for some index e

(i) $\forall i\ \forall x\ \forall n\ \ \varphi_e(i,x,n) = 1 \Leftrightarrow \Phi_i(x) = n$,

(ii) $\vdash$ "φ_e is a total function" ,

(iii) $\vdash$ "$\forall i\ \forall x\ (\exists y\ \varphi_i(x) = y) \Leftrightarrow (\exists n\ \varphi_e(i,x,n) = 1)$" .

Let Φ be a provable Blum measure for φ. From the index e and the Turing machine representing φ in S, we can easily construct a Turing machine U such that

$$\forall i\ \forall x\ U(i,x) = \min \{n \mid \varphi_e(i,x,n) = 1\}$$

and the relationship between U and φ_e can be established in S. By part (i) of the definition, $\forall i\ \forall x\ U(i,x) = \Phi_i(x)$. Thus, U effectively indexes the Φ_i's. We shall consider the Φ_i's as being encoded into S via U.

Given that the Φ_i's are being represented in S by a Turing machine U, we can rewrite the definition of a provable Blum measure as :

(i),(ii) for some index e

$\vdash$ "φ_e is a total function and

$\forall i\ \forall x\ \forall n\ \varphi_e(i,x,n) = 1 \Leftrightarrow \Phi_i(x) = n$" ,

(iii) $\vdash$ "$\forall i$ domain φ_i = domain Φ_i" .

It can now be seen that the conditions of the definition of a provable Blum measure correspond to what we would establish in determining that Φ is a Blum measure.

(i),(ii) We have an algorithm $\varphi_e(i,x,n)$ for testing whether $\Phi_i(x) = n$, and we can prove that the algorithm is total.

(iii) We can show that for any index i, φ_i and Φ_i are defined on precisely the same inputs.

EXAMPLE

We may define a partial recursive function M of two variables so that $\forall i \, \forall x \, M(i,x) = M_i(x)$. Let $\langle , \rangle$ be some simple pairing function. M with $\langle , \rangle$ will form an acceptable Gödel-numbering. Let M and $\langle , \rangle$ be encoded into S via some straightforward Turing machine representations. Then, since all of the usual reasoning about Turing machines can be reproduced in S , the basic properties of M and $\langle , \rangle$ will be theorems of S , and thus they will form a provably acceptable Gödel-numbering.

Consider the TIME and TAPE measures on the Turing machines. Both will be Blum measures for M . All of the usual reasoning about TIME and TAPE requirements can be reproduced in S . Thus, the basic properties of TIME and TAPE will be theorems of S , and both TIME and TAPE will be provable Blum measures for M .

Indeed, for any of the usual acceptable Gödel-numberings and Blum measures, the reasoning used to establish their defining properties will be reproducible in S , and so they will be provably acceptable Gödel-numberings and provable Blum measures.

Throughout the rest of this thesis, let φ be a fixed but arbitrary provably acceptable Gödel-numbering, and let Φ be a fixed but arbitrary provable Blum measure for φ .

1.3 VARIOUS DEFINITIONS

DEFINITIONS

(i) φ_i is provably equivalent to φ_j , written

$\varphi_i \approx \varphi_j$, if $\vdash$ "$\varphi_i = \varphi_j$" .

If φ_i is not provably equivalent to φ_j , we write $\varphi_i \not\approx \varphi_j$.

(ii) The provable equivalence class of φ_i is the set of algorithms $\{\varphi_j \mid \varphi_j \approx \varphi_i\}$.

(iii) M_i is provably total if $\vdash$ "M_i is total".
Similarly, φ_i is provably total if $\vdash$ "φ_i is total".

(iv) f is a p-function if there is a provably total M_i that calculates f .

If $f(x)$ is defined, we say that $f(x)$ converges and we write $f(x)\downarrow$. If $f(x)$ is not defined, we say that $f(x)$ diverges and we write $f(x)\uparrow$.

We follow the usual convention for interpreting inequalities between partial recursive functions - for example, we write $f(x) \leq g(x)$ if either $f(x)\downarrow$ and $g(x)\downarrow$ and $f(x) \leq g(x)$, or $g(x)\uparrow$.

When defining algorithms, we adopt the convention that the maximum of the empty set is zero.

Almost everywhere x (a.e. x) signifies $\exists y\ \forall x > y$. Infinitely often x (i.o. x) signifies $\forall y\ \exists x > y$. We shall often write a.e. or i.o. when the associated variable is clear from the context.

We define the complexity class of f (under the measure Φ) to be the class

$$C[f] = \{g \mid \exists i\ \varphi_i = g \text{ and } \Phi_i(x) \leq f(x) \text{ a.e. } x\} .$$

It is usual to require in the definition of $C[f]$ that f and g be recursive. However, it is convenient for the statement of our

results in Chapter 5 to allow f and g to be partial recursive. We will discuss the matter further in Chapter 5. Note that all of our results involving $C[f]$ will continue to hold if the class is restricted to recursive functions.

DEFINITIONS

(i) We say that φ_i is r-optimal a.e. (i.o.), or that φ_i is optimal a.e. (i.o.) modulo r, if

$$\forall j \; \varphi_j = \varphi_i \Rightarrow \Phi_i(x) \leq r(x,\Phi_j(x)) \text{ a.e. } x \text{ (i.o. } x) .$$

(ii) We say that φ_i has an r speed-up a.e. if

$$\exists j \; \varphi_j = \varphi_i \text{ and } r(x,\Phi_j(x)) < \Phi_i(x) \text{ a.e. } x .$$

We noted earlier that TAPE and TIME are provable Blum measures for the Turing machine enumeration M. We shall denote by $TAPE_i$ the TAPE measure function associated with the Turing machine M_i.[3] Similarly for $TIME_i$.

It will be a convenient shorthand, when the Turing machine representing a function f is clear from the context, to let TAPE $f(x)$ denote the number of tape squares used by that Turing machine in calculating $f(x)$.

---

[3] The details of the TAPE measure can be formulated in various ways. These details are generally not important to our discussions, and we leave it to the reader to fill them in.

1.4 SOME USEFUL RESULTS

It is a consequence of our definitions that the proofs of the standard results for acceptable Gödel-numberings and Blum measures can be reproduced in S for provably acceptable Gödel-numberings and provable Blum measures. The following theorems make use of this fact.

The Isomorphism Theorem for acceptable Gödel-numberings [14] can be reproduced in S . We shall use this result in the following form :

THEOREM 1.1 For some recursive function γ

$$\vdash \text{"}\gamma \text{ is a total, one-one, onto function and } \forall i\ \varphi_{\gamma(i)} = M_i\text{"} .$$

Actually, Theorem 1.1 can serve as an alternative definition of a provably acceptable Gödel-numbering. It is easy to show that if Theorem 1.1 holds for a partial recursive function φ , then φ (together with some simple pairing function) will form a provably acceptable Gödel-numbering. The proof starts with the fact that the Turing machine enumeration M is a provably acceptable Gödel-numbering, and then uses the relationship between M and φ to establish the requisite properties for φ .

Any of the various forms of the Recursion Theorem [15] can be reproduced in S . We shall use this result in the following form :

THEOREM 1.2 For every partial recursive function f , there is a recursive function m such that

$\vdash$ "m is a total function and

$\forall i\ \forall r\ \varphi_{m(i,r)} = \varphi_{f(i,r,m(i,r))}$" .

Kleene's Iteration Theorem for Turing machines [12] can be reproduced in S . So we have :

THEOREM 1.3 Let $W(i,j,x)$ be a Turing machine. Then for some recursive function w

$\vdash$"w is a total function and

$\forall i\ \forall j\ \forall x\ \ W(i,j,x) = M_{w(i,j)}(x)$" .

The Gap Theorem [10] can be reproduced in S . Our version of this result incorporates details that are not made explicit in the usual proofs, and so we sketch out a proof of our own.*

THEOREM 1.4 For some recursive function α

$\vdash$"α is a total function and $\forall a\ \forall r$

(i) $\varphi_{\alpha(a,r)}$ is monotonically increasing and

$\forall x\ \ \varphi_{\alpha(a,r)}(x) \geq \varphi_a(x)$,

(ii) $\forall i\ \forall x \geq i\ \ \Phi_i(x) \leq \varphi_{\alpha(a,r)}(x)$ or

$\Phi_i(x) > \varphi_r(x,\varphi_{\alpha(a,r)}(x))$,

(iii) if φ_a and φ_r are total functions, then

$\varphi_{\alpha(a,r)}$ is a total function" .

* PROOF Define a Turing machine $W(a,r,x)$ by

1. $W(a,r,0) = \varphi_a(0)$;

2. For $x > 0$ $W(a,r,x) = \min \{y \mid \varphi_a(x) \leq y$, $W(a,r,x-1) \leq y$, and $\forall i \leq x\ \ \Phi_i(x) \leq y$ or $\Phi_i(x) > \varphi_r(x,y)\}$.

* See the Addendum.

It is now a straightforward exercise to establish that

(*) $\forall a\ \forall r$

(i) $W(a,r,x)$ is monotonically increasing in x and

$\forall x\ \ W(a,r,x) \geq \varphi_a(x)$,

(ii) $\forall i\ \forall x \geq i\ \ \Phi_i(x) \leq W(a,r,x)$ or

$\Phi_i(x) > \varphi_r(x,W(a,r,x))$,

(iii) if φ_a and φ_r are total functions, then

$W(a,r,x)\downarrow$ for every x .

The arguments used to establish (*) will translate easily into proofs in S . Thus, the statement of (*) will be a theorem of S .

By Theorem 1.3, for some recursive function w

$\vdash$"w is a total function and

$\forall a\ \forall r\ \forall x\ \ W(a,r,x) = M_{w(a,r)}(x)$" .

Let $\alpha = \gamma \circ w$, where γ is as in Theorem 1.1. Then

$\vdash$"α is a total function and

$\forall a\ \forall r\ \forall x\ \ W(a,r,x) = \varphi_{\alpha(a,r)}(x)$" .

This, combined with the statement of (*) as a theorem of S , establishes Theorem 1.4. □

The next theorem shows a relationship between the TAPE measure and Φ .

THEOREM 1.5 Let γ be as in Theorem 1.1. Then for some recursive function h

$\vdash$"h is a total function,

$h(x,y)$ is monotonically increasing in y, and

$\forall i\ \forall x \geq i \quad TAPE_i(x) \leq h(x,\Phi_{\gamma(i)}(x))$ and

$\Phi_{\gamma(i)}(x) \leq h(x,TAPE_i(x))$" .

PROOF Define h by

$$h(x,y) = \max \{TAPE_i(x) , \Phi_{\gamma(i)}(x) \mid 0 \leq i \leq x , \text{ and } TAPE_i(x) \leq y \text{ or } \Phi_{\gamma(i)}(x) \leq y\} .$$

Then, for a straightforward Turing machine representation of h, it will be easy to prove in S that h has the properties claimed for it. The proofs in S will be simple translations of the standard arguments that we would use. □

It is perhaps worth noting that $\{\Phi_{\gamma(i)} \mid i \in \mathbb{N}\}$ is a provable Blum measure for M, and that the proof above is adapted from the proof that any two Blum measures are recursively related. (See [10].)

Our next theorem reproduces in S another standard result.

THEOREM 1.6 For some recursive function u

$\vdash$"u is a total function and

$\forall i \quad \varphi_{u(i)} = \Phi_i$" .

PROOF Recall from Section 1.2 the Turing machine U,

$$U(i,x) = \min \{n \mid \varphi_e(i,x,n) = 1\} = \Phi_i(x) .$$

By Theorem 1.3, for some recursive function w

$\vdash$ "w is a total function and

$\forall i \; \forall x \quad U(i,x) = M_{w(i)}(x)$" .

Let $u = \gamma \circ w$, where γ is as in Theorem 1.1. Then

$\vdash$ "u is a total function and

$\forall i \; \forall x \quad U(i,x) = \varphi_{u(i)}(x)$" .

Since the Φ_i's are encoded into S via U , we have

$\vdash$ "$\forall i \quad \varphi_{u(i)} = \Phi_i$" . □

The following result was also observed in [5].

PROPOSITION 1.7 If g is a primitive recursive function, then g is a p-function.

PROOF From a primitive recursive schema for g , we can construct a Turing machine M_i for g . A little consideration of the possible steps in a primitive recursive schema and of their translation into a Turing machine description makes it clear that M_i is provably total. □

We shall often make implicit use of this result by noting that a primitive recursive process must be provably total.

We observed in Section 1.1 that any theorem of S which is (under our intended interpretation) a statement about Turing machines will be true. Since algorithms φ_i , measure functions Φ_i and all other partial recursive functions are being represented in S by Turing machines, it follows that any theorem of S which is a statement about these objects will also be true. We shall often use this fact in our proofs.

2 SURVEY OF THE FIELD

In this chapter we survey the papers that have already been published on provable conditions in computational complexity theory. Although the setting-up in these papers of the formal system, the indexing of the algorithms and the complexity measure may differ to some degree from ours, the basic approach remains the same, and we shall translate the major results of these papers into our own notation. As an aid to reference, we shall number the translated theorems as they appear in the original papers.

The study of provable conditions in the theory of computation began in the 1950's. Fischer in his own paper on provable recursive functions [5] reviews the work done on provable conditions in recursive function theory. In the 1970's, provable conditions were introduced into the study of computational complexity.

However, despite the promising results achieved and the contention of both Hartmanis [9] and Young [20] that the area deserves a thorough investigation, relatively little work has been done on provable conditions in computational complexity. We have found only five papers in this area, and all were published between 1976 and 1979. We survey these papers below.

Gordon [7] considered complexity classes of p-functions. He showed

THEOREM 1 If f is a p-function and g is recursive with $g \in C[f]$, then g is a p-function.

THEOREM 2 There is a recursive function t such that for any recursive function g, g is a p-function iff $g \in C[t]$.

A simple observation from Theorem 2 is that if a function is too complex, then we cannot prove that it is total. Our own Theorem 4.11 is a generalization of Theorem 2.

Young [20] produced some surprising results about optimization and speed-up among provably equivalent algorithms.

THEOREM 2 For some p-functions σ and r, $\forall i$

(i) $\varphi_{\sigma(i)} = \varphi_i$,

(ii) $\Phi_{\sigma(i)}(x) \leq r(x,\Phi_i(x))$ a.e. x,

(iii) for any $\varphi_j \approx \varphi_{\sigma(i)}$

$\Phi_{\sigma(i)}(x) \leq r(x,\Phi_j(x))$ a.e. x.

Basically, (iii) says that $\varphi_{\sigma(i)}$ is a.e. r-optimal within its provable equivalence class. Thus, for some p-function r, given any algorithm φ_i for a partial recursive function f, we can effectively construct another algorithm $\varphi_{\sigma(i)}$ for f which is a.e. r-optimal within its provable equivalence class.

THEOREM 3 Let φ_r be provably total. For any provably total φ_i,[1] we can effectively construct a φ_j such that

---

[1] In Young's statement of the theorem, φ_r and φ_i are allowed to be recursive. However, we can see the proof working only when φ_r and φ_i are provably total.

(i) $\varphi_j = \varphi_i$,

(ii) for any $\varphi_h \approx \varphi_j$, we can effectively construct a φ_k such that

$$\vdash \text{"}\varphi_k = \varphi_h \text{ and } \varphi_r(x,\Phi_k(x)) < \Phi_h(x) \text{ a.e. } x\text{"} .$$

Thus, if f is the function calculated by φ_i , then φ_j also calculates f and within the provable equivalence class of φ_j every algorithm has an effectively constructible φ_r speed-up a.e. Furthermore, the speed-up relationship can be proved.

The next theorem shows that an algorithm may be optimal (modulo some recursive function) without our being able to prove so.

THEOREM 6 There exist recursive r and φ_i such that

(i) φ_i is r-optimal a.e. ,

(ii) for any $\varphi_j = \varphi_i$

$$\nvdash \text{"}\varphi_j \text{ is r-optimal a.e."} .$$

Young gave another result, Theorem 5, about functions having algorithms that cannot be proved to be optimal. However, the proof is flawed. Our own Theorem 3.3 is a strengthening of Theorem 5.

The other three papers on provable conditions in computational complexity deal with the TIME and TAPE measures on the Turing machines.

This work is done in the context of Turing machines as recognizers of formal languages. Inputs are finite strings from some alphabet. The

halting states of a Turing machine are either ACCEPT or REJECT . The language recognized by M_i is the set $L(M_i)$ of all inputs on which M_i finally enters the state ACCEPT . For any integer x , $TIME_i(x)$ is the maximum number of machine steps used by M_i on any input string of length x . Similarly for TAPE .

Naturally, the famous $P = NP$? problem[2] has received attention.

Baker [1] showed that the addition of various provable conditions to the definitions of P and NP does not simplify the $P = NP$ question.

Hartmanis and Hopcroft [11] considered a relativized version of the $P = NP$? problem. Let P^A denote the set of all languages recognized in polynomial-time by deterministic Turing machines operating with the set A as an oracle, and let NP^A denote the set of all languages recognized in polynomial-time by non-deterministic Turing machines operating with the set A as an oracle.

THEOREM We can effectively construct a Turing machine M_i such that $L(M_i)$ is the empty set and "$P^{L(M_i)} = NP^{L(M_i)}$" is independent of S .

---

[2] P is the set of all languages recognized in polynomial-time by deterministic Turing machines. NP is the set of all languages recognized in polynomial-time by non-deterministic Turing machines. For an extensive discussion of P , NP and the $P = NP$? problem, see [6].

Since $L(M_i)$ is empty, $P^{L(M_i)} = NP^{L(M_i)}$ iff $P = NP$. However, it is not known in S that $L(M_i)$ is empty, and so the theorem above does not actually say that "P = NP" is independent of S. Nevertheless, this theorem does raise the suspicion that the P = NP question might not be resolvable within the axioms of set theory.

Hartmanis and Hopcroft established in [11] two other independence results for the TIME measure.

THEOREM We can exhibit a recursive function t such that the equality of the TIME complexity classes

$$\{L(M_i) \mid \forall n \ \ TIME_i(n) \leq t(n)\} \quad \text{and}$$

$$\{L(M_i) \mid \forall n \ \ TIME_i(n) \leq t^2(n)\}$$

is independent of S.

THEOREM We can exhibit a Turing machine M_i such that $\forall n \ \ TIME_i(n) = n^2$ but $\nvdash$ "$\forall n \ \ TIME_i(n) < 2^n$".

We generalize this last result in our Theorem 3.1.

Hartmanis in [8] defined a provable analogue of a complexity class for the TIME and TAPE measures, and presented a number of results relating the provable complexity classes to ordinary complexity classes. We discuss this paper further in Chapter 5 where we generalize many of its results to abstract provable Blum measures, and so we shall give here only two of the major theorems from [8].

THEOREM There exist recursive functions t such that

$$\{L(M_i) \mid \vdash \text{"}\forall n \ \ TIME_i(n) \leq t(n)\text{"}\} \subsetneq \{L(M_i) \mid \forall n \ \ TIME_i(n) \leq t(n)\} .$$

Thus, there are functions t for which the provable TIME complexity class of t is strictly smaller than the ordinary TIME complexity class of t. We generalize this result in our Theorem 5.3.

THEOREM If M_j is recursive and $\forall n \ \ TAPE_j(n) \geq n$, then

$$\{L(M_i) \mid \vdash \text{"}\forall n \ \ TAPE_i(n) \leq TAPE_j(n)\text{"}\} = \{L(M_i) \mid \forall n \ \ TAPE_i(n) \leq TAPE_j(n)\} .$$

Thus, for a certain natural class of resource-bounding functions, the provable TAPE complexity classes coincide with the corresponding ordinary TAPE complexity classes. We generalize this result in our Theorem 5.6.

Finally, let us note that many of the results from [8], [11] and [20] are collected in [9].

3 ANOMALOUS ALGORITHMS AND PROVABLE COMPLEXITY PROPERTIES

3.1 INTRODUCTION

In any practical situation, when considering some particular partial recursive function f, we must have (at least implicitly) an algorithm to define f. Before we can admit any other algorithm as a means of calculating f, that algorithm must be proved equivalent to our defining algorithm for f.

The requirement of being provably equivalent defines an equivalence relation on the set of all algorithms. We call the equivalence classes of this relation provable equivalence classes. (See the definitions in Section 1.3.) Theorem 4.5 shows that for any partial recursive function f, the set of all algorithms that calculate f divides into infinitely many provable equivalence classes.

Let f be an arbitrary partial recursive function. Depending on which provable equivalence class our defining algorithm for f lies in, we will form quite different answers to typical questions involving f. In this chapter, we consider three basic questions about f:

1. To what complexity classes does f belong?
2. Does f have an algorithm which is optimal modulo some given recursive function?
3. When f itself is taken as the resource-bounding function for a complexity class, by how much must we increase f before admitting new functions into the complexity class?

In each instance we demonstrate the existence of anomalous defining algorithms for f, that is, defining algorithms for f under which there is a significant discrepancy between what is true about f and what can be proved about f. Our results are presented properly in Section 3.2. They can, however, be paraphrased as follows :

THEOREM 3.1 For any recursive function r, there is a defining algorithm for f under which f cannot be proved to be calculable by an algorithm of complexity bounded by r - even though, in many cases, the defining algorithm itself will have complexity bounded by r.

THEOREM 3.3 For any recursive function r, there is a defining algorithm for f under which f cannot be proved to have an algorithm optimal modulo r - although f may well have such an algorithm and, in many such cases, the defining algorithm for f will itself be optimal modulo r.

THEOREM 3.5 For any recursive function r, there is a defining algorithm for f under which it cannot be proved that the complexity class $C[r(x,f(x))]$ is strictly larger than $C[f]$ - although, in many cases, this will be true.

Further, in each theorem, such anomalous defining algorithms can be effectively constructed.

These theorems show that when investigating the computational complexity properties of functions we must relate our results to the algorithms used to define the functions rather than to the functions themselves, and that our results may be severaly limited by the peculiarities of these defining algorithms within the formal system we employ.

3.2 RESULTS

THEOREM 3.1 For some p-functions σ , g

(i) $\vdash$ "$\forall i\ \forall r\ \forall x$ if $\varphi_i(x)\downarrow$ and $\varphi_r(x)\downarrow$, then $\varphi_{\sigma(i,r)}(x)\downarrow$" ;

and $\forall i\ \forall r$ if φ_r is recursive, then

(ii) $\not\vdash$ "$\exists k\ \ \varphi_k = \varphi_{\sigma(i,r)}$ and $\Phi_k \leq \varphi_r$ i.o." ,

(iii) $\varphi_{\sigma(i,r)} = \varphi_i$,

(iv) $\Phi_{\sigma(i,r)}(x) \leq g(x,\Phi_i(x))$ a.e. x .

DISCUSSION

We shall talk as if S were the formal system within which the study of computational complexity is carried out.

In [11], it was shown that a Turing machine could be explicitly produced which ran in n^2 time but which could not be formally proved to run faster than 2^n time. Our result is a generalization of this.

Let φ_r be recursive, and let φ_i calculate a function f . By (iii), $\varphi_{\sigma(i,r)}$ also calculates f . Note that σ is a p-function. Thus, $\varphi_{\sigma(i,r)}$ can be effectively constructed from φ_i and φ_r , and by a process which is provably total.

By (ii), if the defining algorithm for f is taken to be $\varphi_{\sigma(i,r)}$ or any algorithm provably equivalent to $\varphi_{\sigma(i,r)}$, we will not be able to prove that f is calculable by an algorithm of complexity bounded infinitely often by φ_r . However, if $\Phi_i \leq \varphi_r$ a.e., then f can in fact be calculated by an algorithm of complexity bounded almost everywhere by φ_r . Further, by (iv), if $g(x,\Phi_i(x)) \leq \varphi_r(x)$ a.e. x , then $\Phi_{\sigma(i,r)} \leq \varphi_r$ a.e. , so that the discrepancy between what we can prove and what is true is even greater. Indeed, we can prove at most

that $\Phi_{\sigma(i,r)}$ is bounded by φ_r on finitely many inputs.

The question arises as to whether we are actually in danger of encountering such anomalies in practical situations. The difficulty here is that there is no way we can know whether our defining algorithm for a function is anomalous.

It might be hoped that we could avoid such anomalies by restricting ourselves to what is arguably the more practically meaningful class of provably total algorithms. However, (i) shows that if φ_i and φ_r are provably total, then so is $\varphi_{\sigma(i,r)}$.

Since the function g in (iv) may be large, $\Phi_{\sigma(i,r)}$ may itself always be large, and perhaps we can be safe from anomalies when dealing with algorithms of small complexity. We shall discuss this further, after the proof, for the more concrete case of the Turing machines with the TIME and TAPE measures.

PROOF OF THEOREM 3.1 :

ALGORITHM IN (i,r,j,x)

1. Mark off $\log x$ tape. Generate the theorems of S and write them down until the $\log x$ tape is full. Check whether the formula "$\exists k\ \varphi_k = \varphi_j$ and $\Phi_k \leq \varphi_r$ i.o." has been written down.

2. If the formula has not been written down, then calculate and output $\varphi_i(x)$.

3. If the formula has been written down, then calculate and output

$$1 + \max \{\varphi_n(x) \mid 0 \leq n \leq x \text{ and } \Phi_n(x) \leq \varphi_r(x)\} .$$

DISCUSSION OF THE ALGORITHM

1. Log will be to some appropriate base. We don't actually calculate $\log x$ but rather, say, the greatest integer $q \leq \log x$. As we noted in Section 1.1, the theorems of S can be generated primitive recursively. So, all of this step can be done primitive recursively.

2. $\varphi_i(x)$ is calculated by inputting (i,x) to the Turing machine representing φ in S.

3. The test $\Phi_n(x) \leq \varphi_r(x)$ is carried out by testing progressively $\varphi_e(n,x,0) = 1$, $\varphi_e(n,x,1) = 1$, ... up to $\varphi_e(n,x,\varphi_r(x)) = 1$, where e is the index associated with Φ in the definition of a provable Blum measure. (See Section 1.2.)

The algorithm translates into a Turing machine $T(i,r,j,x)$. By Theorem 1.3, for some p-function t

$$\vdash \text{"}\forall i\ \forall r\ \forall j\ \forall x \quad T(i,r,j,x) = M_{t(i,r,j)}(x)\text{"} .$$

Let $v = \gamma \circ t$, where γ is as in Theorem 1.1. Then

$$\vdash \text{"}\forall i\ \forall r\ \forall j \quad \varphi_{v(i,r,j)} = M_{t(i,r,j)}\text{"} .$$

By Theorem 1.2, for some p-function m

$$\vdash \text{"}\forall i\ \forall r \quad \varphi_{m(i,r)} = \varphi_{v(i,r,m(i,r))}\text{"} .$$

Let $\sigma = m$. Then σ is a p-function and

$$(*) \quad \vdash \text{"}\forall i\ \forall r\ \forall x \quad \varphi_{\sigma(i,r)}(x) = T(i,r,\sigma(i,r),x)\text{"} .$$

We now establish each section of the theorem in turn.

(i) Consider the algorithm operating on some (i,r,j,x) . Suppose that $\varphi_i(x)\downarrow$ and $\varphi_r(x)\downarrow$. Step 1 is primitive recursive and therefore converges. Step 2 converges because $\varphi_i(x)\downarrow$. Since $\varphi_r(x)\downarrow$ and φ_e is total, step 3 must also converge. So the algorithm will converge.

This sort of reasoning could be carried out in S for T . Thus

$$\vdash \text{"}\forall i\ \forall r\ \forall j\ \forall x \text{ if } \varphi_i(x)\downarrow \text{ and } \varphi_r(x)\downarrow \text{ , then } T(i,r,j,x)\downarrow \text{ "} .$$

So, from (*) ,

$$\vdash \text{"}\forall i\ \forall r\ \forall x \text{ if } \varphi_i(x)\downarrow \text{ and } \varphi_r(x)\downarrow \text{ , then } \varphi_{\sigma(i,r)}(x)\downarrow \text{ "} .$$

For the remaining sections of the proof, let i be arbitrary but assume that φ_r is recursive.

(ii) Suppose $\vdash$ "$\exists k\ \varphi_k = \varphi_{\sigma(i,r)}$ and $\Phi_k \leq \varphi_r$ i.o." .
We shall establish a contradiction.

By the theorem, for some n , $\varphi_n = \varphi_{\sigma(i,r)}$ and $\Phi_n \leq \varphi_r$ i.o. So, for some $x \geq n$, we must have that $\Phi_n(x) \leq \varphi_r(x)$ and the theorem will appear on $\log x$ tape in step 1 of the algorithm.

Now, consider the algorithm on $(i,r,\sigma(i,r),x)$. The theorem is found in step 1, so we go to step 3. Step 3 converges as φ_r is total. Since $n \leq x$ and $\Phi_n(x) \leq \varphi_r(x)$, the output from step 3 will be greater than $\varphi_n(x)$.

Thus, from (*), $\varphi_{\sigma(i,r)}(x)\downarrow$ and $\varphi_{\sigma(i,r)}(x) > \varphi_n(x)$. But $\varphi_n = \varphi_{\sigma(i,r)}$. Contradiction.

(iii) It follows from (ii) that for every x the algorithm on $(i,r,\sigma(i,r),x)$ goes to step 2, and therefore $T(i,r,\sigma(i,r),x) = \varphi_i(x)$.

Hence $\varphi_{\sigma(i,r)} = \varphi_i$.

(iv) Consider T operating on some $(i,r,\sigma(i,r),x)$. The calculation for step 1 requires $\log x$ tape. Since the algorithm goes to step 2, it follows that

$$\text{TAPE } T(i,r,\sigma(i,r),x) \leq \log x + \text{TAPE } \varphi_i(x) .$$

Define g by

$$g(x,y) = \max \{\Phi_{\sigma(a,b)}(x) \mid 0 \leq a \leq x \ , \ 0 \leq b \leq x \ ,$$

$$\Phi_a(x) = y \text{ and } \text{TAPE } T(a,b,\sigma(a,b),x) \leq \log x + \text{TAPE } \varphi_a(x)\} .$$

Note that if $\Phi_a(x) = y$ and $\text{TAPE } T(a,b,\sigma(a,b),x) \leq \log x + \text{TAPE } \varphi_a(x)$, then $T(a,b,\sigma(a,b),x)\downarrow$ and so $\Phi_{\sigma(a,b)}(x)\downarrow$.

Clearly then, g is total.

The proof that g is total could be carried out in S for some straightforward Turing machine representation of g .

Therefore, g is a p-function.

Finally, observe that for $x \geq \max \{i,r\}$, $\Phi_{\sigma(i,r)}(x) \leq g(x,\Phi_i(x))$. □

Let us consider the result for the Turing machine enumeration M with the TAPE and TIME measures.

It is easy to arrange the function t in the proof so that $M_{t(i,r,j)}$ calculates step 1 using $\log x$ tape and step 2 with the instruction set for M_i built into it. Then, whenever the theorem is not found in step 1, $\text{TAPE}_{t(i,r,j)}(x) \leq \log x + \text{TAPE}_i(x)$.

By Theorem 1.2 for M , for some p-function m

$$\vdash \text{"}\forall i\ \forall r \quad M_{m(i,r)} = M_{t(i,r,m(i,r))}\text{"} .$$

Now, define $\sigma(i,r)$ to be $t(i,r,m(i,r))$ rather than $m(i,r)$. Results (i), (ii) and (iii) will still hold. However, we can tighten (iv) to

(iv)' $\text{TAPE}_{\sigma(i,r)}(x) \leq \log x + \text{TAPE}_i(x) \quad \forall x$.

Further, if we alter the algorithm in step 1 to lay off and work in $\log \log x$ tape, then to calculate step 1 will require no more than x Turing machine steps[1] for sufficiently large inputs x . Following the same development as above, we will again have (i), (ii) and (iii), but will be able to tighten (iv) to

(iv)" $\text{TIME}_{\sigma(i,r)}(x) \leq x + \text{TIME}_i(x) \quad$ a.e. x .

These results show that anomalous algorithms may be very close in complexity to even the 'fastest' algorithms for calculating functions. Two examples of the consequences of this are that anomalous algorithms

---

[1] The TIME cost may, of course, be greater - x^2 , for example - if we are using inappropriate input-output conventions.

will appear in the class of LOG-SPACE algorithms, and it is possible that our failure to prove P = NP is the result of using an anomalous defining algorithm for an NP-complete problem.

So, anomalies cannot be avoided by a restriction to provably total algorithms or to 'fast' algorithms. It is only a hope that intuitively natural defining algorithms will not be anomalous.

Next, we have some preliminaries for Theorem 3.3.

The Speed-up Theorem states :

For any recursive function r, there is a recursive function f such that

$$\forall i \ \ \varphi_i = f \Rightarrow \varphi_i \text{ has an } r \text{ speed-up a.e.}$$

It is a simple exercise to observe that in the theorem we can also have

$$\forall i \ \ \varphi_i = f \text{ a.e.} \Rightarrow \varphi_i \text{ has an } r \text{ speed-up a.e.}$$

Examination of a standard proof of the Speed-up Theorem, such as that in [19], shows that f can be effectively constructed from r, that this construction process is provably total, and that if r is provably total, then so is f. In fact, we have

THEOREM 3.2 For some p-function λ

(i) for any recursive φ_r, $\varphi_{\lambda(r)}$ is recursive and

$$\forall i \ \ \varphi_i = \varphi_{\lambda(r)} \text{ a.e.} \Rightarrow \varphi_i \text{ has a } \varphi_r \text{ speed-up a.e. ;}$$

(ii) $\vdash$ "$\forall r$ if φ_r is total, then $\varphi_{\lambda(r)}$ is total" .

We use this result in the following theorem.

THEOREM 3.3 For some p-functions σ , g

(i) $\vdash$ "$\forall i\ \forall r\ \forall x$ if $\varphi_i(x)\downarrow$ and φ_r is total, then $\varphi_{\sigma(i,r)}(x)\downarrow$ " ;
and $\forall i\ \forall r$ if φ_r is recursive, then

(ii) $\nvdash$ "$\exists k\ \ \varphi_k = \varphi_{\sigma(i,r)}$ and φ_k is φ_r-optimal i.o." ,

(iii) $\varphi_{\sigma(i,r)} = \varphi_i$,

(iv) $\Phi_{\sigma(i,r)}(x) \leq g(x,\Phi_i(x))$ a.e. x .

DISCUSSION

Let φ_r be recursive, and let φ_i calculate a function f . By (iii), $\varphi_{\sigma(i,r)}$ also calculates f .

By (ii), if the defining algorithm for f is taken to be $\varphi_{\sigma(i,r)}$ or any algorithm provably equivalent to $\varphi_{\sigma(i,r)}$, we will not be able to prove that f is calculable by an algorithm that is φ_r-optimal i.o. However, φ_i may in fact be φ_r-optimal a.e. Further, by (iv) if φ_i is optimal a.e. modulo a sufficiently small function, then $\varphi_{\sigma(i,r)}$ will itself be φ_r-optimal a.e.

As we observed for Theorem 3.1, the anomalous algorithms are not readily avoided. From (i), if φ_i and φ_r are provably total, then so is $\varphi_{\sigma(i,r)}$. Further, by the same sort of construction as was presented in the discussion after the proof of Theorem 3.1, for the Turing machine enumeration M we can tighten (iv) to

(iv)' $\text{TAPE}_{\sigma(i,r)}(x) \leq \log x + \text{TAPE}_i(x)$ $\forall x$, and

(iv)" $\text{TIME}_{\sigma(i,r)}(x) \leq x + \text{TIME}_i(x)$ a.e. x .

PROOF OF THEOREM 3.3 :

ALGORITHM IN (i,r,j,x)

1. Mark off $\log x$ tape. Generate the theorems of S and write them down until the $\log x$ tape is full. Check whether the formula "$\exists k$ $\varphi_k = \varphi_j$ and φ_k is φ_r-optimal i.o" has been written down.

2. If the formula has not been written down, then calculate and output $\varphi_i(x)$.

3. If the formula has been written down, then calculate and output $\varphi_{\lambda(r)}(x)$, where λ is as in Theorem 3.2.

The algorithm translates into a Turing machine $T(i,r,j,x)$. Following the same procedure as in the proof of Theorem 3.1, for some p-function σ

(*) $\quad \vdash$ "$\forall i\ \forall r\ \forall x\ \ \varphi_{\sigma(i,r)}(x) = T(i,r,\sigma(i,r,),x)$" .

We now establish each section of the theorem in turn.

(i) Consider the algorithm operating on some (i,r,j,x) . Suppose that $\varphi_i(x)\downarrow$ and φ_r is total. Step 1 is primitive recursive and therefore converges. Step 2 converges as $\varphi_i(x)\downarrow$. By Theorem 3.2, since φ_r is total, $\varphi_{\lambda(r)}$ is total. Also, the calculation of the index $\lambda(r)$ converges since λ is total. Therefore, step 3 converges. So the algorithm will converge.

This sort of reasoning could be carried out in S for T . Thus

$\vdash$ "$\forall i\ \forall r\ \forall j\ \forall x$ if $\varphi_i(x)\downarrow$ and φ_r is total, then $T(i,r,j,x)\downarrow$" .

So, from (*)

$\vdash$ "$\forall i\ \forall r\ \forall x$ if $\varphi_i(x)\downarrow$ and φ_r is total, then $\varphi_{\sigma(i,r)}(x)\downarrow$" .

For the remaining sections of the proof, let i be arbitrary but assume that φ_r is recursive.

(ii) Suppose $\vdash$ "$\exists k\ \varphi_k = \varphi_{\sigma(i,r)}$ and φ_k is φ_r-optimal i.o." . We shall establish a contradiction.

Almost everywhere x , the theorem will appear on $\log x$ tape in step 1 of the algorithm.

Therefore, $T(i,r,\sigma(i,r),x) = \varphi_{\lambda(r)}(x)$ a.e. x .

So, from (*), $\varphi_{\sigma(i,r)} = \varphi_{\lambda(r)}$ a.e.

By the theorem, for some n , $\varphi_n = \varphi_{\sigma(i,r)}$ and φ_n is φ_r-optimal i.o.

Thus $\varphi_n = \varphi_{\lambda(r)}$ a.e. So, by Theorem 3.2, for some b , $\varphi_b = \varphi_n$ and $\varphi_r(x,\Phi_b(x)) < \Phi_n$ a.e. x .

But, since φ_n is φ_r-optimal i.o., $\Phi_n(x) \leq \varphi_r(x,\Phi_b(x))$ i.o. x .

Thus $\Phi_n(x) < \Phi_n(x)$ i.o. x . So $\varphi_n(x)\uparrow$ i.o. x .

However, $\varphi_n = \varphi_{\lambda(r)}$ a.e. and, since φ_r is total, $\varphi_{\lambda(r)}$ is total. Contradiction.

(iii) It follows from (ii) that $\forall x\ T(i,r,\sigma(i,r),x) = \varphi_i(x)$.

Hence $\varphi_{\sigma(i,r)} = \varphi_i$.

(iv) It again follows from (ii) that

$\forall x$ TAPE $T(i,r,\sigma(i,r),x) \leq \log x +$ TAPE $\varphi_i(x)$.

Let g be defined in the same way as in the proof of (iv) in Theorem 3.1.

Then g is a p-function and $\Phi_{\sigma(i,r)}(x) \leq g(x,\Phi_i(x))$ a.e. x . □

As a simple corollary to Theorem 1.4, we have the following version of the Gap Theorem :

THEOREM 3.4 For some p-function β

(i) $\forall r\ \forall i\ \forall x \geq i\ \ \Phi_i(x) \leq \varphi_{\beta(r)}(x)$ or

$\Phi_i(x) > \varphi_r(x,\varphi_{\beta(r)}(x))$,

(ii) $\vdash$ "$\forall r$ if φ_r is total, then $\varphi_{\beta(r)}$ is total" .

We use this result in the following theorem.

THEOREM 3.5 For some p-functions σ , g

(i) $\vdash$ "$\forall i\ \forall r\ \forall x$ if $\varphi_i(x)\downarrow$ and φ_r is total, then

$\varphi_{\sigma(i,r)}(x)\downarrow$" ;

and $\forall i\ \forall r$ if φ_r is recursive, then

(ii) $\not\vdash$ "$\exists k\ \ \varphi_{\sigma(i,r)}(x) < \Phi_k(x) \leq \varphi_r(x,\varphi_{\sigma(i,r)}(x))$ i.o. x" ,

(iii) $\varphi_{\sigma(i,r)} = \varphi_i$,

(iv) $\Phi_{\sigma(i,r)}(x) \leq g(x,\Phi_i(x))$ a.e. x .

DISCUSSION

Let φ_i calculate a function f. Consider f as the resource-bounding function for a complexity class. A typical question is whether a certain increase of the bound f will admit new functions into the complexity class. This question can be posed as follows :

Let φ_r be recursive. Does $C[\varphi_r(x,f(x))]$ properly contain $C[f]$?

Theorem 3.5 states that if the defining algorithm for f is taken to be $\varphi_{\sigma(i,r)}$ or any algorithm provably equivalent to $\varphi_{\sigma(i,r)}$, then even if $C[\varphi_r(x,f(x))]$ does properly contain $C[f]$, we will not be able to prove it. In fact, we will not even be able to prove that there is an algorithm whose complexity lies between $f(x)$ and $\varphi_r(x,f(x))$ infinitely often.

As we observed for Theorem 3.1, the anomalous algorithms are not readily avoided. If φ_i and φ_r are provably total, then so is $\varphi_{\sigma(i,r)}$. For the Turing machine enumeration M we can tighten (iv) to

(iv)' $\text{TAPE}_{\sigma(i,r)}(x) \leq \log x + \text{TAPE}_i(x) \quad \forall x$, and

(iv)" $\text{TIME}_{\sigma(i,r)}(x) \leq x + \text{TIME}_i(x)$ a.e. x.

PROOF OF THEOREM 3.5 :

ALGORITHM IN (i,r,j,x)

1. Mark off $\log x$ tape. Generate the theorems of S and write them down until the $\log x$ tape is full. Check whether the formula

   "$\exists k \quad \varphi_j(x) < \varphi_k(x) \leq \varphi_r(x,\varphi_j(x))$ i.o. x"

   has been written down.

2. If the formula has not been written down, then calculate and output $\varphi_i(x)$.

3. If the formula has been written down, then calculate and output $\varphi_{\beta(r)}(x)$, where β is as in Theorem 3.4.

The algorithm translates into a Turing machine $T(i,r,j,x)$. Following the same procedure as in the proof of Theorem 3.1, for some p-function σ

(*) $\vdash$ "$\forall i\ \forall r\ \forall x\ \ \varphi_{\sigma(i,r)}(x) = T(i,r,\sigma(i,r),x)$" .

We now establish each section of the theorem in turn.

(i) Using Theorem 3.4 in place of Theorem 3.2, the argument here follows the same pattern as in the proof of Theorem 3.3 (i).

For the remaining sections of the proof, let i be arbitrary but assume that φ_r is recursive.

(ii) Suppose $\vdash$ "$\exists k\ \ \varphi_{\sigma(i,r)}(x) < \Phi_k(x) \leq \varphi_r(x,\varphi_{\sigma(i,r)}(x))$ i.o. x" . We shall establish a contradiction.

Almost everywhere x , the theorem will appear on $\log x$ tape in step 1 of the algorithm. Therefore $T(i,r,\sigma(i,r),x) = \varphi_{\beta(r)}(x)$ a.e. x . So, from (*), $\varphi_{\sigma(i,r)} = \varphi_{\beta(r)}$ a.e.

By the theorem, for some n ,

$\varphi_{\sigma(i,r)}(x) < \Phi_n(x) \leq \varphi_r(x,\varphi_{\sigma(i,r)}(x))$ i.o. x .

So, for some $x \geq n$, $\varphi_{\beta(r)}(x) < \Phi_n(x) \leq \varphi_r(x,\varphi_{\beta(r)}(x))$.

But by Theorem 3.4, $\Phi_n(x) \leq \varphi_{\beta(r)}(x)$ or $\Phi_n(x) > \varphi_r(x,\varphi_{\beta(r)}(x))$.

This is a contradiction since φ_r is recursive and, therefore, $\varphi_{\beta(r)}$ is recursive.

(iii) It follows from (ii) that $\forall x \;\; T(i,r,\sigma(i,r),x) = \varphi_i(x)$.

Hence $\varphi_{\sigma(i,r)} = \varphi_i$.

(iv) Let g be defined in the same way as in the proof of (iv) in Theorem 3.1.

Then g is a p-function and $\Phi_{\sigma(i,r)}(x) \leq g(x,\Phi_i(x))$ a.e. x . □

4 PROVABLE EQUIVALENCE AND COMPLEXITY OF ALGORITHMS

4.1 INTRODUCTION

Usually in complexity theory, all algorithms that calculate the same function are classed together; but to the computer scientist working within a formal system the algorithms fall into provable equivalence classes : algorithms can be recognized as equivalent if and only if they are provably equivalent. The results in Chapter 3 show that what we can prove about the complexity properties of a function depends on the provable equivalence class to which our defining algorithm for the function belongs. It is natural then to enquire as to the relationship between provable equivalence and the computational complexity of algorithms. For example, we might pose such questions as : Do the algorithms in a single provable equivalence class all have complexities which are, in some sense, close together? Do the different provable equivalence classes for a function form separate bunches of increasingly complex algorithms, or do they interleave?

In this chapter we investigate these and related questions. Many of our results are rather complicated to state. However, we paraphrase some of them below.

THEOREM 4.1 Every provable equivalence class contains infinitely many algorithms.

COROLLARY 4.4 Let the partial recursive function f be defined on an infinite domain. Then from any algorithm φ_i for f we can effectively construct another algorithm $\varphi_{\lambda(i)}$ for f such that every algorithm

provably equivalent to $\varphi_{\lambda(i)}$ has greater complexity a.e. than every algorithm provably equivalent to φ_i .

THEOREM 4.5 Every partial recursive function has infinitely many provable equivalence classes.

THEOREM 4.11 From any recursive φ_i we can effectively construct a recursive function t such that for any recursive function g the following three conditions are equivalent :

(i) $g \in C[t]$,

(ii) g is less complex than some algorithm provably equivalent to φ_i ,

(iii) domain g provably contains domain φ_i .

4.2 RESULTS

THEOREM 4.1 $\forall i$ the provable equivalence class of φ_i is infinite.

PROOF Consider an arbitrary φ_i . Let γ be as in Theorem 4.2. There are infinitely many minor modifications we can make to the instruction set for $M_{\gamma^{-1}(i)}$ each of which obviously will not affect the machine's output. Each one of these modifications produces a different Turing machine M_j such that $\vdash$ "$M_{\gamma^{-1}(i)} = M_j$" . Now, γ is one-one and $\vdash$ "$\varphi_i = M_{\gamma^{-1}(i)} = M_j = \varphi_{\gamma(j)}$" . Thus, the provable equivalence class of φ_i is infinite. □

The following result will be the key to many others.

THEOREM 4.2 For some p-function σ

(i) $\vdash$ "$\forall i\ \forall r\ \forall x$ if $\varphi_i(x)\downarrow$ and $\varphi_r(x)\downarrow$, then $\varphi_{\sigma(i,r)}(x)\downarrow$" ;

and $\forall i\ \forall r$ if domain $\varphi_i \subseteq$ domain φ_r, then

(ii) $\varphi_{\sigma(i,r)} = \varphi_i$,

(iii) for any $\varphi_k \approx \varphi_{\sigma(i,r)}$, $\Phi_k > \varphi_r$ a.e.

Basically, this theorem says that each function has provable equivalence classes of arbitrarily large complexity.

Let f be a partial recursive function, let $\varphi_i = f$, and let φ_r be defined wherever f is. Then $\varphi_{\sigma(i,r)} = f$ and every member of the provable equivalence class of $\varphi_{\sigma(i,r)}$ has greater complexity a.e. than φ_r.

This illustrates again that there are 'bad' defining algorithms for functions. For example, the function f may be calculable in linear time, but our defining algorithm for f can be so 'bad' that every algorithm provably equivalent to it runs slower than super-exponential time.

PROOF OF THEOREM 4.2 :

ALGORITHM IN (i,r,j,x)

1. Generating the theorems of S primitive recursively, let $p_0, p_1, \ldots, p_x$ be the first $x + 1$ indices such that $\vdash$ "$\varphi_{p_n} = \varphi_j$" .

2. For each $n = 0,\ldots,x$ test $\Phi_{p_n}(x) \leq \varphi_r(x)$.

(a) If some n satisfies the test, then calculate and output

$$1 + \max \{\varphi_{p_n}(x) \mid 0 \leq n \leq x \text{ and } \Phi_{p_n}(x) \leq \varphi_r(x)\} .$$

(b) If no n satisfies the test, then calculate and output $\varphi_i(x)$.

The algorithm translates into a Turing machine $T(i,r,j,x)$. By Theorem 1.3, for some p-function t

$$\vdash \text{"}\forall i\ \forall r\ \forall j\ \forall x \quad T(i,r,j,x) = M_{t(i,r,j)}(x)\text{"} .$$

Let $v = \gamma \circ t$, where γ is as in Theorem 1.1. Then

$$\vdash \text{"}\forall i\ \forall r\ \forall j \quad \varphi_{v(i,r,j)} = M_{t(i,r,j)}\text{"} .$$

By Theorem 1.2, for some p-function m

$$\vdash \text{"}\forall i\ \forall r \quad \varphi_{m(i,r)} = \varphi_{v(i,r,m(i,r))}\text{"} .$$

Let $\sigma = m$. Then σ is a p-function and

(*) $$\vdash \text{"}\forall i\ \forall r\ \forall x \quad \varphi_{\sigma(i,r)}(x) = T(i,r,m(i,r),x)\text{"} .$$

We now establish each section of the theorem in turn.

(i) Consider the algorithm operating on some (i,r,j,x) . Suppose that $\varphi_i(x)\downarrow$ and $\varphi_r(x)\downarrow$. Step 1 is primitive recursive and therefore converges. Since $\varphi_r(x)\downarrow$, the tests in step 2 will converge and therefore so will the calculations for (a) . Since $\varphi_i(x)\downarrow$, (b) will converge. Thus, the algorithm will converge on (i,r,j,x) .

This sort of reasoning could be carried out in S for T . Thus

$$\vdash \text{"}\forall i\ \forall r\ \forall j\ \forall x \quad \text{if } \varphi_i(x)\downarrow \text{ and } \varphi_r(x)\downarrow \text{ , then } T(i,r,j,x)\downarrow \text{ "} .$$

So, from (*),

$$\vdash \text{"}\forall i\ \forall r\ \forall x \quad \text{if } \varphi_i(x)\downarrow \text{ and } \varphi_r(x)\downarrow \text{ , then } \varphi_{\sigma(i,r)}(x)\downarrow \text{ "} .$$

For the remaining sections of the proof, let i be arbitrary but assume that domain $\varphi_i \subseteq$ domain φ_r .

(ii) Consider an arbitrary x .

Suppose that $\varphi_r(x)\uparrow$. Then by our assumption, $\varphi_i(x)\uparrow$.

Also $T(i,r,\sigma(i,r),x)\uparrow$, since the calculation of $\varphi_r(x)$ in step 2 diverges.

So, by (*), $\varphi_{\sigma(i,r)}(x)\uparrow$. Thus, $\varphi_i(x) = \varphi_{\sigma(i,r)}(x)$.

Now suppose that $\varphi_r(x)\downarrow$. We claim that $T(i,r,\sigma(i,r),x)$ is defined through step(b) of the algorithm.

For if the test in step 2 is satisfied, say $\varphi_{p_n}(x) \leq \varphi_r(x)$, then $T(i,r,\sigma(i,r),x)\downarrow$ and $T(i,r,\sigma(i,r),x) > \varphi_{p_n}(x)$.

But, by (*), $\varphi_{\sigma(i,r)}(x) = T(i,r,\sigma(i,r),x)$ and, from step 1, $\varphi_{p_n} = \varphi_{\sigma(i,r)}$. Contradiction.

So, $T(i,r,\sigma(i,r),x)$ is defined through (b).

Thus, $\varphi_{\sigma(i,r)}(x) = \varphi_i(x)$.

So $\varphi_{\sigma(i,r)} = \varphi_i$.

(iii) Suppose that $\varphi_k \approx \varphi_{\sigma(i,r)}$.

Considering step 1 of the algorithm for $T(i,r,\sigma(i,r),x)$, we see that for some n , $p_n = k$.

If $\varphi_r(x)\uparrow$, then as we saw in (ii), $\varphi_{\sigma(i,r)}(x)\uparrow$, and so $\varphi_{p_n}(x)\uparrow$ and $\Phi_{p_n}(x)\uparrow$.

If $x \geq n$ and $\varphi_r(x)\downarrow$, then as we saw in (ii), the test in step 2 is not satisfied, and so $\Phi_{p_n}(x) > \varphi_r(x)$.

Thus, $\Phi_k > \varphi_r$ a.e. □

The following result tells us, among other things, that the algorithms in a single provable equivalence class all have complexities which are, in a sense, bunched together.

COROLLARY 4.3 For some p-functions κ and λ, $\forall i$

(i) domain $\varphi_{\kappa(i)}$ = domain φ_i ,

(ii) for any $\varphi_j \approx \varphi_i$, $\Phi_j < \varphi_{\kappa(i)}$ a.e. ,

(iii) $\varphi_{\lambda(i)} = \varphi_i$ and for any $\varphi_b \approx \varphi_{\lambda(i)}$, $\varphi_{\kappa(i)} < \Phi_b$ a.e.

PROOF Define a Turing machine $W(i,x)$ by the following algorithm :

1. Generating the theorems of S , let $p_0, p_1, \ldots, p_x$ be the first $x + 1$ indices such that $\vdash$ "$\varphi_{p_n} = \varphi_i$" .
2. Calculate and output $\Phi_{p_0}(x) + \Phi_{p_1}(x) + \ldots + \Phi_{p_x}(x)$.

By Theorem 1.3, for some p-function w

$\forall i \; \forall x \quad M_{w(i)}(x) = W(i,x)$.

Let $\kappa = \gamma \circ w$, where γ is as in Theorem 1.1.

Then κ is a p-function and $\forall i \; \forall x \; \varphi_{\kappa(i)}(x) = W(i,x)$.

It is now easy to see that $\forall i$

(i) domain $\varphi_{\kappa(i)}$ = domain φ_i ,

(ii) for any $\varphi_j \approx \varphi_i$, $\Phi_j < \varphi_{\kappa(i)}$ a.e.

Define λ by $\lambda(i) = \sigma(i,\kappa(i))$, where σ is as in Theorem 4.2.

Then λ is a p-function and $\forall i$

(iii) $\varphi_{\lambda(i)} = \varphi_i$ and for any $\varphi_b \approx \varphi_{\lambda(i)}$, $\varphi_{\kappa(i)} < \Phi_b$ a.e. □

Let $f = \varphi_i$ be defined on an infinite domain. Then φ_i and $\varphi_{\lambda(i)}$ represent distinct provable equivalence classes for f, with every member of the provable equivalence class of $\varphi_{\lambda(i)}$ having greater complexity a.e. than every member of the provable equivalence class of φ_i. Let us say that the provable equivalence class of $\varphi_{\lambda(i)}$ has greater complexity a.e. than the provable equivalence class of φ_i. Similarly, $\varphi_{\lambda\circ\lambda(i)} = f$ and the provable equivalence class of $\varphi_{\lambda\circ\lambda(i)}$ has greater complexity a.e. than the provable equivalence class of $\varphi_{\lambda(i)}$. By successive applications of λ, we can form representatives for an infinite chain of provable equivalence classes for f, with each class having greater complexity a.e. than the preceding classes.

We can state this observation as

COROLLARY 4.4 For some p-function λ, $\forall i$

(i) $\varphi_{\lambda(i)} = \varphi_i$,

(ii) if φ_i is defined on an infinite domain, then the provable equivalence class of $\varphi_{\lambda(i)}$ has greater complexity a.e. than the provable equivalence class of φ_i.

A natural question is : How many provable equivalence classes does a function have? Our next result gives the expected answer.

THEOREM 4.5 Every partial recursive function has infinitely many provable equivalence classes.

PROOF Let f be a partial recursive function.

If f is defined on an infinite domain, then Corollary 4.4 shows that f has infinitely many provable equivalence classes.

Now suppose that f is defined on only a finite domain. Further, suppose that f has only finitely many provable equivalence classes - say n distinct provable equivalence classes represented by $\varphi_{p_1}, \varphi_{p_2}, \ldots, \varphi_{p_n}$. We shall establish a contradiction by constructing a representative φ_i for a new provable equivalence class for f.

Let $x_0 = \max \{x \mid f(x)\downarrow\}$. Define a Turing machine $W(j,x)$ by the following algorithm :

1. For $x \leq x_0$, calculate and output $\varphi_{p_1}(x)$.
2. For $x > x_0$, generate the theorems of S seeking

   "$\varphi_j = \varphi_{p_1}$" or "$\varphi_j = \varphi_{p_2}$" or ... "$\varphi_j = \varphi_{p_n}$".

   If one of these theorems is found, output 1.

For some recursive function w,

$\forall j \; \forall x \quad M_{w(j)}(x) = W(j,x)$.

Let $t = \gamma \circ w$, where γ is as in Theorem 1.1.

Then $\forall j \; \forall x \quad \varphi_{t(j)}(x) = W(j,x)$.

By the Recursion Theorem, for some i, $\varphi_i = \varphi_{t(i)}$.

Therefore, $\forall x \quad \varphi_i(x) = W(i,x)$.

We first show that φ_i is not provably equivalent to any of $\varphi_{p_1}, \ldots, \varphi_{p_n}$.

Suppose $\vdash$ "$\varphi_i = \varphi_{p_k}$", where $1 \leq k \leq n$.

Then $\forall x > x_0 \quad \varphi_{p_k}(x) = \varphi_i(x) = W(i,x) = 1$.

But $\forall x > x_0 \quad \varphi_{p_k}(x) = f(x)\uparrow$. Contradiction.

It now follows that $\forall x > x_0 \quad \varphi_i(x) = W(i,x)\uparrow$.

Hence $\forall x > x_0 \quad \varphi_i(x) = f(x)$.

Further, $\forall x \leq x_0 \quad \varphi_i(x) = \varphi_{p_1}(x) = f(x)$. Therefore, $\varphi_i = f$.

Thus, φ_i represents a new provable equivalence class for f . Therefore, f has infinitely many provable equivalence classes. □

Our next result is the first of several which show that the relationship between provable equivalence and the complexity of algorithms is somehow involved with provable relations between the domains of algorithms.

THEOREM 4.6 $\forall i \; \forall r$ if $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" , then for some $\varphi_j \approx \varphi_i$, $\varphi_r < \Phi_j$ a.e.

Results 4.3 and 4.6 give some idea of how widely spread (in terms of computational complexity) the algorithms provably equivalent to φ_i can be. Corollary 4.3 shows that there is a $\varphi_{\kappa(i)}$ which is defined whenever φ_i is, and which bounds the complexities of all the algorithms provably equivalent to φ_i . Theorem 4.6 shows that if we can prove that φ_r is defined whenever φ_i is, then φ_r does not bound the complexities of all the algorithms provably equivalent to φ_i .

PROOF OF THEOREM 4.6 :

Suppose $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" . Let h be as in Theorem 1.5. Define a Turing machine M_k by the following algorithm :

1. Waste $1 + h(x, \varphi_r(x))$ tape squares.

2. Calculate and output $\varphi_i(x)$.

Let $j = \gamma(k)$, where γ is as in Theorem 1.1.

We first show that $\varphi_j \approx \varphi_i$.

If $\varphi_i(x)\uparrow$, then $M_k(x)\uparrow$ and so $\varphi_j(x)\uparrow$.

Now suppose that $\varphi_i(x)\downarrow$. Then $\varphi_r(x)\downarrow$ and so, since h is total, the calculation in step 1 converges. Therefore, $M_k(x) = \varphi_i(x)$.

So, $\varphi_j(x) = \varphi_i(x)$.

Thus, $\forall x \ \varphi_j(x) = \varphi_i(x)$.

This reasoning could be reproduced in S . So, $\varphi_j \approx \varphi_i$.

Now, $\forall x \ \text{TAPE}_k(x) > h(x,\varphi_r(x))$.

By Theorem 1.5, $\forall x \geq k \quad h(x,\Phi_j(x)) \geq \text{TAPE}_k(x)$.

Therefore, $h(x,\Phi_j(x)) > h(x,\varphi_r(x))$ a.e. x .

So, since h is monotonically increasing in its second argument, $\Phi_j > \varphi_r$ a.e. □

A modification of this result will also prove useful.

COROLLARY 4.7 $\forall i \ \forall r$ if $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" , then for some $\varphi_j \approx \varphi_i$, $\Phi_r < \Phi_j$ a.e.

PROOF Suppose $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" .

Now $\vdash$ "domain φ_r = domain Φ_r" , from the definition of a provable Blum measure.

So $\vdash$ "domain $\varphi_i \subseteq$ domain Φ_r" .

By Theorem 4.6, for some $\varphi_j \approx \varphi_i$, $\Phi_r < \Phi_j$ a.e. □

Let f be a partial recursive function defined on an infinite domain. In our remarks following Corollary 4.3, we showed that starting with any provable equivalence class for f there is an infinite chain of provable equivalence classes for f , with each class having greater complexity a.e. than the preceding classes. The next corollary shows

that it is also the case that every provable equivalence class for f interleaves with others.

COROLLARY 4.8 If φ_i is defined on an infinite domain, then for some $\varphi_j \approx \varphi_i$ and some $\varphi_a = \varphi_i$, $\Phi_i < \Phi_a < \Phi_j$ a.e. but $\varphi_a \not\approx \varphi_i$.

PROOF By Theorem 1.6, for some p-function u

$$\vdash \text{"}\forall i \ \ \varphi_{u(i)} = \Phi_i\text{"} .$$

Suppose that φ_i is defined on an infinite domain.

Let $a = \sigma(i,u(i))$, where σ is as in Theorem 4.2.

Then $\varphi_a = \varphi_i$ and for any $\varphi_k \approx \varphi_a$, $\Phi_k > \Phi_i$ a.e.

So, $\varphi_a \not\approx \varphi_i$ and $\Phi_i < \Phi_a$ a.e.

Now $\vdash$ "domain φ_i = domain Φ_i " .

Therefore $\vdash$ "domain $\varphi_i \subseteq$ domain $\varphi_{u(i)}$" .

By Theorem 4.2 $\vdash$ "$\forall x$ if $\varphi_i(x)\downarrow$ and $\varphi_{u(i)}(x)\downarrow$, then $\varphi_a(x)\downarrow$" .

So $\vdash$ "domain $\varphi_i \subseteq$ domain φ_a" .

By Corollary 4.7, for some $\varphi_j \approx \varphi_i$, $\Phi_a < \Phi_j$ a.e. □

We now observe that Corollary 4.7 has a partial converse.

THEOREM 4.9 $\forall i$, for any recursive φ_r,

if for some $\varphi_j \approx \varphi_i$, $\Phi_r \leq \Phi_j$ a.e.,

then for some $\varphi_t = \varphi_r$ $\vdash$ "domain $\varphi_i \subseteq$ domain φ_t" .

PROOF Suppose that φ_r is recursive and that for some $\varphi_j \approx \varphi_i$, $\Phi_r(x) \leq \Phi_j(x)$ $\forall x > m$. Define φ_t by the following algorithm :

1. For $x \leq m$, output $\varphi_r(x)$ from a table of values $\varphi_r(0), \varphi_r(1), \ldots, \varphi_r(m)$.

2. For $x > m$, if $\Phi_r(x) \leq \Phi_j(x)$, then calculate and output $\varphi_r(x)$, else output 0 .

Clearly $\varphi_t = \varphi_r$. We now show that $\vdash$ "domain $\varphi_i \subseteq$ domain φ_t" .

For $x \leq m$, $\varphi_t(x)\downarrow$ since values are looked up in a table. For $x > m$, if $\varphi_i(x)\downarrow$, then $\Phi_j(x)\downarrow$ and so $\varphi_t(x)\downarrow$. Thus, domain $\varphi_i \subseteq$ domain φ_t .

This reasoning could be reproduced in S . Therefore $\vdash$ "domain $\varphi_i \subseteq$ domain φ_t" . □

For our next result we call on the Union Theorem, a proof of which may be found in [10].

UNION THEOREM Let $\{f_n \mid n \in \mathbb{N}\}$ be a recursively enumerable sequence of recursive functions such that $\forall n \, \forall x \; f_n(x) < f_{n+1}(x)$. Then for some recursive function t, $\forall r$

$$\Phi_r \leq t \text{ a.e. iff } \exists n \; \Phi_r \leq f_n \text{ a.e.}$$

Corollary 4.3 shows that we can bound the complexities of the algorithms provably equivalent to φ_i . When φ_i is recursive, that result can be strengthened.

THEOREM 4.10 For any recursive φ_i , there is a recursive function t_i (effectively computable from i) such that $\forall r$

$$\Phi_r \leq t_i \text{ a.e. iff for some } \varphi_j \approx \varphi_i \text{ , } \Phi_r \leq \Phi_j \text{ a.e.}$$

Let φ_i be recursive. Then not only can we bound the complexities of the algorithms provably equivalent to φ_i, but we can bound them so tightly that there is no 'gap' between them and our resource-bound.

PROOF Suppose that φ_i is recursive.

Generating the theorems of S, let $p_0, p_1, \ldots, p_n$ be the first $n+1$ indices such that $\vdash$ "$\varphi_{p_m} = \varphi_i$". Define the sequence $\{f_n \mid n \in \mathbb{N}\}$ by

$$f_0 = \Phi_{p_0} \quad \text{and} \quad \forall n \quad f_{n+1} = f_n + \Phi_{p_{n+1}} + 1 .$$

This sequence satisfies the requirements of the Union Theorem, and so for some recursive function t_i, $\forall r$

$$\Phi_r \leq t_i \text{ a.e.} \quad \text{iff} \quad \exists n \ \Phi_r \leq f_n \text{ a.e.}$$

In the proof of the Union Theorem, t is constructed from the sequence $\{f_n \mid n \in \mathbb{N}\}$. Examining this construction, we see that in our case t_i is effectively computable from i. (In fact, $t_i = \varphi_{\kappa(i)}$ for some p-function κ.)

Now, $\forall n \ \Phi_{p_n} \leq f_n$.

Therefore, if for some $\varphi_j \approx \varphi_i$, $\Phi_r \leq \Phi_j$ a.e., then $\exists n \ \Phi_r \leq f_n$ a.e. and so $\Phi_r \leq t_i$ a.e.

Further, for each $n \vdash$ "$\varphi_i = \varphi_{p_0} = \ldots = \varphi_{p_n}$".

So $\vdash$ "domain $\varphi_i \subseteq$ domain f_n".

By Theorem 4.6, for some $\varphi_j \approx \varphi_i$, $f_n \leq \Phi_j$ a.e.

Therefore, if $\Phi_r \leq t_i$ a.e., then $\exists n \ \Phi_r \leq f_n$ a.e. and so for some $\varphi_j \approx \varphi_i$, $\Phi_r \leq \Phi_j$ a.e.

Thus, $\forall r$, $\Phi_r \leq t_i$ a.e. iff for some $\varphi_j \approx \varphi_i$, $\Phi_r \leq \Phi_j$ a.e. □

Let φ_i be recursive. From results 4.7 and 4.10, we see that $\forall r$

if $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" , then $\Phi_r \leq t_i$ a.e.

Thus, if an algorithm's complexity is not bounded by t_i , then we cannot prove that the algorithm's domain contains domain φ_i . This illustrates that as the difference between the complexities of algorithms increases, what we can prove about the relationships between the algorithms decreases.

Results 4.7, 4.9 and 4.10 show a relationship between provable containment of domains, provable equivalence and computational complexity. We draw these results together in

THEOREM 4.11 For any recursive φ_i , there is a recursive function t_i (effectively computable from i) such that for any recursive function g the following three conditions are equivalent :

(i) $g \in C[t_i]$,

(ii) for some $\varphi_j \approx \varphi_i$, $g \in C[\Phi_j]$,

(iii) for some $\varphi_r = g$ $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" .

PROOF Let φ_i be recursive, and let t_i be as in Theorem 4.10.
The equivalence of (i) and (ii) is given by Theorem 4.10.
That (ii) implies (iii) follows from Theorem 4.9.
That (iii) implies (ii) follows from Corollary 4.7. □

The equivalence of (i) and (iii) is especially interesting because it equates a purely formal property (a provable relationship between

domains) with a complexity property (computability within a given resource-bound).

The equivalence of (i) and (iii) also generalizes the following result from [7] :

COROLLARY 4.12 There is a recursive function t such that for any recursive function g

$g \in C[t]$ iff g is a p-function.

PROOF Let φ_i be provably total, and let $t = t_i$ be as in Theorem 4.10. Then for any recursive function g

$g \in C[t]$

iff for some $\varphi_r = g$ $\vdash$ "domain $\varphi_i \subseteq$ domain φ_r"

iff for some $\varphi_r = g$, φ_r is provably total

iff g is a p-function. □

Actually, in the above, t bounds the complexities of the provably total algorithms as tightly as possible in the sense that there is no 'gap' between them and the resource-bound t :

If φ_r is provably total, then

$\vdash$ "domain $\varphi_i \subseteq$ domain φ_r" and so $\Phi_r \leq t$ a.e.

If $\Phi_r \leq t$ a.e., then for some $\varphi_j \approx \varphi_i$, $\Phi_r \leq \Phi_j$ a.e. and φ_j is provably total.

5 COMPLEXITY CLASSES AND PROVABLE COMPLEXITY CLASSES

5.1 INTRODUCTION

Many results in abstract computational complexity theory revolve around the notion of a complexity class, and much of the work with particular complexity measures - such as the TIME and TAPE measures on the Turing machines - consists of determining to what complexity classes a given function belongs. In this chapter we consider provable analogues of complexity classes.

We define the complexity class of f (under the measure Φ) to be the class

$$C[f] = \{g \mid \exists i \ \varphi_i = g \text{ and } \Phi_i \leq f \text{ a.e.}\} .$$

It is usual to require in the definition of $C[f]$ that the functions f and g be recursive. For the work in this chapter, however, it is convenient to admit a greater generality by allowing f and g to be only partially recursive. Nevertheless, if the requirement that f and g be recursive is added to the definition of $C[f]$ and to the (forthcoming) definitions of $B[f]$ and $A[f]$, then all of our results will continue to hold.

Suppose that we have a defining algorithm φ_d for a partial recursive function g. To show that g is in $C[f]$, we must prove

"$\exists i \ \varphi_i = \varphi_d$ and $\Phi_i \leq f$ a.e."

However, proving the mere existence of an algorithm for g that runs within the resource-bound f is not particularly satisfactory. In practice, we strive to exhibit such an algorithm. That is, we aim to prove for some i that

$$"\varphi_i = \varphi_d \text{ and } \Phi_i \leq f \text{ a.e."}$$

Theorem 3.1 tells us that for any recursive function f, there will be infinitely many possible defining algorithms φ_d for g such that we cannot even prove

$$"\exists i \ \varphi_i = \varphi_d \text{ and } \Phi_i \leq f \text{ i.o."}$$

So certainly, if our defining algorithm for g is a 'bad' one, then we will not be able to exhibit an algorithm for g that runs within the resource-bound f. But perhaps there is a 'good' defining algorithm for g using which we can exhibit such an algorithm.

The question can be posed as follows : Does there exist an algorithm $\varphi_d = g$ such that for some i we can prove

$$"\varphi_i = \varphi_d \text{ and } \Phi_i \leq f \text{ a.e."} \ ?$$

Clearly, this is equivalent to : Does there exist an algorithm $\varphi_i = g$ such that we can prove

$$"\Phi_i \leq f \text{ a.e."} \ ?$$

The above question suggests a provable analogue of the complexity class of f. We define the class $B[f]$ as follows :

$$B[f] = \{g \mid \exists i \ \varphi_i = g \text{ and } \vdash "\Phi_i \leq f \text{ a.e."}\} .$$

The class $C[f]$ consists of all the partial recursive functions for which there is an algorithm that runs within the resource-bound f. The class $B[f]$ consists of all the partial recursive functions for which there is an algorithm that can be proved to run within the resource-bound f.

A variation on the class $B[f]$ is suggested by consideration of the almost-everywhere bounding conditions. Writing out in full the provable condition for $B[f]$, we have :

$$\vdash \text{"}\exists n \quad \forall x > n \quad \Phi_i(x) \leq f(x)\text{"} .$$

This condition guarantees the existence of a starting point n for the bounding, but gives no value for n. In a practical situation, it would be natural to ask that some explicit value for n be given. The required condition can be written as :

$$\exists n \quad \vdash \text{"}\forall x > n \quad \Phi_i(x) \leq f(x)\text{"} .$$

Let us denote by $A[f]$ the class defined with this condition. That is

$$A[f] = \{g \mid \exists i \quad \varphi_i = g \quad \text{and} \quad \exists n \quad \vdash \text{"}\forall x > n \quad \Phi_i(x) \leq f(x)\text{"}\} .$$

Even more stringent than the requirement that an explicit starting-point for the bounding be given is the requirement that the complexity of the algorithm be bounded by f everywhere, that is, on every input. This sort of bounding is considered in the study of the TIME and TAPE measures on Turing machines. Hartmanis in [8] considered provable complexity classes defined with everywhere bounding for the TIME and TAPE measures. His work, like most work on these measures, was in the context of Turing machines as recognizers of formal languages.

THEOREM 5.11 For infinitely many recursive functions f

$$A[f] \subsetneq B[f] = C[f] .$$

5.2 RESULTS

To begin the results, we note the obvious.

OBSERVATION 5.1 It follows immediately from the definitions that for any partial recursive function f

$$A[f] \subseteq B[f] \subseteq C[f] .$$

We shall be concerned with conditions that force the above inclusions to be either proper inclusions or equality. The next half-dozen results deal with the relationship between the A-classes and the C-classes. First, we have a means of producing from φ_t a function not in $A[\varphi_t]$.

THEOREM 5.2 For some p-functions κ and g , $\forall t$

(i) $\forall x \geq t \quad \Phi_{\kappa(t)}(x) \leq g(x,\varphi_t(x))$,

(ii) if φ_t is defined on an infinite domain, then

$\varphi_{\kappa(t)} \notin A[\varphi_t]$.

PROOF In the following algorithm for a Turing machine, let the p-function γ be as in Theorem 1.1.

ALGORITHM in (t,x)

1. Mark off $\log x$ tape. Within that length of tape, generate and write down the theorems of S . Whenever a theorem of the

form "$\forall y \geq n \ \Phi_i(y) \leq \varphi_t(y)$" appears, check whether $n \leq \log x$ and $i \leq \log x$, and if they both are, then write down the description of $M_{\gamma^{-1}(i)}$ before going on to the next theorem. Stop when the $\log x$ tape is full.

2. For each $M_{\gamma^{-1}(i)}$ description written down, simulate $M_{\gamma^{-1}(i)}$ operating on the input x. Record the maximum of the values $M_{\gamma^{-1}(i)}(x)$.

3. Add 1 to this maximum and output that value. (If no $M_{\gamma^{-1}(i)}$ descriptions were written down, then output 1.)

Let us consider each of the steps for the algorithm operating on some arbitrary (t,x).

1. Log will be to some appropriate base. We don't actually deal with $\log x$ but rather, say, the greatest integer $q \leq \log x$. The marking-off of $\log x$ tape requires no more than $\log x$ tape.

As we noted in Section 1.1 the theorems of S can be generated primitive recursively. The calculations of the indices $\gamma^{-1}(i)$ will converge since γ^{-1} is a p-function. All of the other operations in this step can be done primitive recursively.

Thus, the calculations in step 1 will converge and will use $\log x$ tape.

2. The description of $M_{\gamma^{-1}(i)}$ will be in some standard form such as quintuples. Suppose the description fills D_i tape squares. Then the simulation of $M_{\gamma^{-1}(i)}(x)$ will require no more than $D_i \cdot \text{TAPE}_{\gamma^{-1}(i)}(x)$ tape.

From step 1, we have that $\vdash$ "$\forall y \geq n \ \Phi_i(y) \leq \varphi_t(y)$" and also $n \leq \log x$, $i \leq \log x$ and $D_i < \log x$.

Since $n < x$, $\Phi_i(x) \leq \varphi_t(x)$. Let us note here that if $\varphi_t(x)\downarrow$, then $\Phi_i(x)\downarrow$, and so $M_{\gamma^{-1}(i)}(x)\downarrow$.

Let h be as in Theorem 1.5.

Then, since $i < x$, $\text{TAPE}_{\gamma^{-1}(i)}(x) \leq h(x,\Phi_i(x))$.

Therefore, since h is monotonically increasing in its second argument,

$\text{TAPE}_{\gamma^{-1}(i)}(x) \leq h(x,\varphi_t(x))$.

So, $D_i \cdot \text{TAPE}_{\gamma^{-1}(i)}(x) < \log x \cdot h(x,\varphi_t(x))$.

By running the successive simulations over the same tape, we can arrange that the tape used in step 2 is just the tape used in the lengthiest simulation.

Thus, step 2 requires less than $\log x \cdot h(x,\varphi_t(x))$ tape.

Also, if $\varphi_t(x)\downarrow$, then the calculations in step 2 will converge.

3. For the addition of 1, we allow one extra tape square.

Finally (assuming without loss of generality that $h \geq 1$) if we overlay the calculations in step 1 and the subsequent calculations, then the entire algorithm requires no more than $\log x \cdot h(x,\varphi_t(x))$ tape. Also, if $\varphi_t(x)\downarrow$, then the algorithm will converge on (t,x).

The algorithm translates into a Turing machine $W(t,x)$. By Theorem 1.3, for some p-function w

$$\vdash \text{"}\forall t \ \forall x \ \ W(t,x) = M_{w(t)}(x)\text{"} .$$

Further, from our discussion above, we can ensure that

$$\forall t \ \forall x \ \ \text{TAPE}_{w(t)}(x) \leq \log x \cdot h(x,\varphi_t(x)) .$$

By Theorem 1.5,

$$\forall t\ \forall x \geq t\ \ \Phi_{\gamma(w\ (t))}(x) \leq h(x,\mathrm{TAPE}_{w(t)}(x)) .$$

So, $\forall t\ \forall x \geq t\ \ \Phi_{\gamma(w\ (t))}(x) \leq h(x,\log x \cdot h(x,\varphi_t(x)))$.

Let $\kappa = \gamma \circ w$, and define g by $g(x,y) = h(x,\log x \cdot h(x,y))$. Then κ and g are p-functions, and

$$\forall t\ \forall x \geq t\ \ \Phi_{\kappa(t)}(x) \leq g(x,\varphi_t(x)) .$$

Thus, we have established part (i) of the theorem.

We next prove part (ii).

Suppose that φ_t is defined on an infinite domain and $\varphi_{\kappa(t)} \in A[\varphi_t]$. We shall establish a contradiction.

Since $\varphi_{\kappa(t)} \in A[\varphi_t]$, for some i and some n ,

$$\varphi_{\kappa(t)} = \varphi_i \quad \text{and} \quad \vdash \text{"}\forall y \geq n\ \ \Phi_i(y) \leq \varphi_t(y)\text{"} .$$

Therefore, for some x , $\varphi_t(x)\downarrow$ and the theorem "$\forall y \geq n\ \ \Phi_i(y) \leq \varphi_t(y)$" and also the description of $M_{\gamma^{-1}(i)}$ will be written down in step 1 of the algorithm in (t,x) .

Since $\varphi_t(x)\downarrow$, the algorithm will converge on (t,x). By the construction of the algorithm, $W(t,x) > M_{\gamma^{-1}(i)}(x)$.

By Theorem 1.1, $\vdash \text{"}\forall t\ \ \varphi_{\gamma\circ w(t)} = M_{w(t)}\text{"}$. Therefore

$$\varphi_{\kappa(t)}(x) = M_{w(t)}(x) = W(t,x) > M_{\gamma^{-1}(i)}(x) = \varphi_i(x) .$$

That is, $\varphi_{\kappa(t)}(x) > \varphi_i(x)$.

But $\varphi_{\kappa(t)} = \varphi_i$ and $\varphi_{\kappa(t)}(x)\downarrow$. Contradiction.

Thus, if φ_t is defined on an infinite domain, then $\varphi_{\kappa(t)} \notin A[\varphi_t]$. □

Using Theorem 5.2, we can generalize a number of results from [8]. First, we generalize Corollary 2 [8].

COROLLARY 5.3 For some p-function g , for any partial recursive function f defined on an infinite domain,

$$A[f] \subsetneq C[g(x,f(x))] .$$

PROOF Let κ and g be as in Theorem 5.2.
Assume without loss of generality that $\forall x \forall y \; g(x,y) \geq y$.
Then $A[f] \subseteq C[g(x,f(x))]$.
Let $\varphi_t = f$. Then $\varphi_{\kappa(t)} \in C[g(x,f(x))]$,
but if f is defined on an infinite domain, then $\varphi_{\kappa(t)} \notin A[f]$. □

Next, we generalize Theorem 3 [8].

THEOREM 5.4 For infinitely many recursive functions f

$$A[f] \subsetneq C[f] .$$

Thus, A-classes may be strictly smaller than C-classes. If we take $A[f]$ to represent what we can prove to be computable within the resource-bound f , then for infinitely many resource-bounds f , what we can prove to be computable within f is strictly less than what is computable within f .

Actually, we can prove a stronger version of Theorem 5.4 :

THEOREM 5.4' For some p-function τ, $\forall a$

(i) $\varphi_{\tau(a)}$ is monotonically increasing and
$\forall x \ \varphi_{\tau(a)}(x) \geq \varphi_a(x)$;

(ii) if φ_a is provably total, then $\varphi_{\tau(a)}$ is provably total;

(iii) if φ_a is recursive, then $\varphi_{\tau(a)}$ is recursive and
$A[\varphi_{\tau(a)}] \subsetneq C[\varphi_{\tau(a)}]$.

The existence of the p-function τ shows that there is actually an algorithm to produce functions $f = \varphi_{\tau(a)}$ such that $A[f] \subsetneq C[f]$. Part (i) tells us that f can be made arbitrarily large. From part (ii), f will be a p-function when φ_a is provably total.

PROOF OF THEOREM 5.4' :

Let g be as in Corollary 5.3. Since g is a p-function, g can be calculated by a provably total φ_r .

Let α be as in Theorem 1.4.
Define τ by $\tau(a) = \alpha(a,r)$. Then τ is a p-function.
By Theorem 1.4, $\forall a$

(i) $\varphi_{\tau(a)}$ is monotonically increasing and
$\forall x \ \varphi_{\tau(a)}(x) \geq \varphi_a(x)$;

(ii) if φ_a is provably total, then $\varphi_{\tau(a)}$ is provably total.

Now suppose that φ_a is recursive.
By Theorem 1.4, $\varphi_{\tau(a)}$ is recursive and

$$\forall i \ \forall x \geq i \ \ \Phi_i(x) \leq \varphi_{\tau(a)}(x) \ \text{ or}$$

$$\Phi_i(x) > g(x, \varphi_{\tau(a)}(x)) \ .$$

Therefore, $C[g(x,\varphi_{\tau(a)}(x))] \subseteq C[\varphi_{\tau(a)}]$. By Corollary 5.3, $A[\varphi_{\tau(a)}] \subsetneq C[g(x,\varphi_{\tau(a)}(x))]$. Thus, $A[\varphi_{\tau(a)}] \subsetneq C[\varphi_{\tau(a)}]$. □

We should note further that if κ is the p-function from Theorem 5.2, then whenever φ_a is recursive

$$\varphi_{\kappa\circ\tau(a)} \notin A[\varphi_{\tau(a)}] \quad \text{but} \quad \varphi_{\kappa\circ\tau(a)} \in C[\varphi_{\tau(a)}] .$$

Thus, as well as having an algorithm to produce functions f such that $A[f] \subsetneq C[f]$, we also have an algorithm to produce functions that lie in the difference $C[f] - A[f]$.

Our next result is an extension of Theorem 5.4. It generalizes Corollary 4 [8].

COROLLARY 5.5 For any recursive function G , for infinitely many recursive functions f ,

$$A[G(x,f(x))] \subsetneq C[f] .$$

This shows that even if we increase the resource-bound f by an arbitrary recursive function G , there will be infinitely many f such that what we can prove to be computable within $G(x,f(x))$ is strictly less than what is computable within f .

PROOF The proof follows the same pattern as for Theorem 5.4'. Use Theorem 1.4 to produce recursive functions f such that

$$C[g(x,G(x,f(x)))] \subseteq C[f] .$$

Then, by Corollary 5.3, $A[G(x,f(x))] \subsetneq C[g(x,G(x,f(x)))]$. □

We have not managed to establish, for general provable Blum measures, the existence of recursive functions f such that $A[f] = C[f]$. However, if Φ has certain special properties, we can show that $\forall t \;\; A[\Phi_t] = C[\Phi_t]$.

First, we need some definitions.

DEFINITIONS

(i) A Blum measure Φ is said to be finitely invariant if

$$\forall i \; \forall j \quad \text{if} \quad \varphi_i = \varphi_j \quad \text{a.e., then}$$
$$\exists k \quad \varphi_k = \varphi_i \quad \text{and} \quad \Phi_k \leq \Phi_j \quad \text{a.e.}$$

(ii) A Blum measure Φ is said to have the Parallel Computation Property (PCP) if $\forall i \; \forall j \; \exists \; k$

$$\forall x \quad \varphi_k(x) = \begin{cases} \varphi_i(x) & \text{if} \quad \Phi_i(x) \leq \Phi_j(x) \\ \varphi_j(x) & \text{otherwise} \end{cases}$$

$$\text{and} \quad \Phi_k(x) = \min \{\Phi_i(x), \Phi_j(x)\} .$$

The idea behind finite invariance is that we can modify an algorithm's behaviour on finitely many inputs without increasing the algorithm's almost-everywhere complexity.

The idea behind the PCP is that we can run two algorithms in parallel without any extra cost in terms of complexity.

We now introduce provable analogues of definitions (i) and (ii).

DEFINITIONS

(iii) We say that Φ is provably finitely invariant if

$\forall i\ \forall j$ if $\varphi_i = \varphi_j$ a.e., then

$\exists k\ \varphi_k = \varphi_i$ and $\exists n \vdash$ "$\forall x > n\ \Phi_k(x) \leq \Phi_j(x)$" .

(iv) We say that Φ has the provable PCP if $\forall i\ \forall j\ \exists k$

$$\forall x\ \varphi_k(x) = \begin{cases} \varphi_i(x) & \text{if } \Phi_i(x) \leq \Phi_j(x) \\ \varphi_j(x) & \text{otherwise} \end{cases}$$

and $\exists n \vdash$ "$\forall x > n\ \Phi_k(x) \leq \Phi_j(x)$" .

Note that, in general, if we can show that a Blum measure is finitely invariant and has the PCP , then our arguments will be reproducible in S and will show that the measure is provably finitely invariant and has the provable PCP .

THEOREM 5.6 If Φ is provably finitely invariant and has the provable PCP , then

$$\forall t\ A[\Phi_t] = B[\Phi_t] = C[\Phi_t] .$$

PROOF Suppose that $f \in C[\Phi_t]$.

Then for some i , $\varphi_i = f$ and $\Phi_i \leq \Phi_t$ a.e.

Since Φ has the provable PCP , for some k ,

$$\forall x\ \varphi_k(x) = \begin{cases} \varphi_i(x) & \text{if } \Phi_i(x) \leq \Phi_t(x) \\ \varphi_t(x) & \text{otherwise} \end{cases}$$

and $\exists n \vdash$ "$\forall x > n\ \Phi_k(x) \leq \Phi_t(x)$" .

Notice that $\varphi_k = \varphi_i$ a.e.

Since Φ is provably finitely invariant, for some j ,

$\varphi_j = \varphi_i$ and $\exists m \vdash$ "$\forall x > m \ \Phi_j(x) \leq \Phi_k(x)$" .

Thus, $\varphi_j = f$ and $\exists v \vdash$ "$\forall x > v \ \Phi_j(x) \leq \Phi_t(x)$" .

Therefore, $f \in A[\Phi_t]$. □

The above result generalizes Theorem 7(1) [8] and Corollary 8 [8]. It is a simple exercise to check that the TAPE measure is provably finitely invariant and has the provable PCP .

Theorem 5.6 indicates what strong conditions are required to guarantee the existence of recursive functions f such that $A[f] = C[f]$. Many natural measures are finitely invariant. However, most natural measures do not have the PCP - for example, the TIME measure does not have the PCP [2].

For general provable Blum measures, we have the following result, which generalizes Corollaries 9 and 10 [8].

THEOREM 5.7 For some p-function b , $\forall t$

$$C[\varphi_t] \subseteq A[b(x,\Phi_t(x))] \text{ and}$$
$$C[\Phi_t] \subseteq A[b(x,\Phi_t(x))] .$$

* PROOF Define a Turing machine $W(i,t,n,x)$ by

IF $x > n$ and $\Phi_i(x) > \varphi_t(x)$ and $\Phi_i(x) > \Phi_t(x)$
THEN output 0 ELSE calculate and output $\varphi_i(x)$.

By Theorem 1.3, for some p-function w

$$\vdash \text{"}\forall i \ \forall t \ \forall n \ \forall x \ \ W(i,t,n,x) = M_{w(i,t,n)}(x)\text{"} .$$

Let $\sigma = \gamma \circ w$, where γ is as in Theorem 1.1.
Then σ is a p-function and $\vdash$ "$\forall i \ \forall t \ \forall n \ \forall x \ \ W(i,t,n,x) = \varphi_{\sigma(i,t,n)}(x)$" .

* See the Addendum.

Define b by

$$b(x,y) = \max \{\Phi_{\sigma(i,t,n)}(x) \mid 0 \leq i,t,n < x \text{ and } \Phi_t(x) = y\} .$$

It is a simple exercise to show that b is a total function and

$$\forall i\ \forall t\ \forall n\ ,\ \text{for}\ x > \max\{i,t,n\}\ ,\ \Phi_{\sigma(i,t,n)}(x) \leq b(x,\Phi_t(x)) .$$

Our arguments will be reproducible in S for a straightforward Turing machine representation of b.
Thus, b is a p-function and

$$\vdash \text{"}\forall i\ \forall t\ \forall n\ ,\ \text{for}\ x > \max\{i,t,n\}\ ,\ \Phi_{\sigma(i,t,n)}(x) \leq b(x,\Phi_t(x))\text{"} .$$

Suppose that $f \in C[\varphi_t]$ or $f \in C[\Phi_t]$.
Then for some i and some n, $\varphi_i = f$ and

$$\forall x \geq n\ \ \Phi_i(x) \leq \varphi_t(x) \ \text{ or } \ \Phi_i(x) \leq \Phi_t(x) .$$

Therefore $\varphi_{\sigma(i,t,n)} = \varphi_i = f$.
Let $m = \max\{i,t,n\}$. Then

$$\vdash \text{"}\forall x \geq m\ \ \Phi_{\sigma(i,t,n)}(x) \leq b(x,\Phi_t(x))\text{"} .$$

Thus, $f \in A[b(x,\Phi_t(x))]$. □

By making the proof more complicated - for example, by incorporating arguments from Lemma 5.12 - we could have forced the inclusions between the classes in Theorem 5.7 to be proper inclusions.

We now present some result for the B-classes.

For the following theorem we adopt the convention that the TAPE cost of a Turing machine computation does not include the number of tape squares initially required to write down the input value.

THEOREM 5.8 For the TAPE measure on the Turing machine enumeration M, for any partial recursive function f,
if $\vdash$ "$f(x) \geq \log x$ a.e." , then $B[f] = C[f]$.

In Lemma 11 [8] Hartmanis establishes a similar result for the TIME measure. He attributes the result to A. Meyer.[1]

Note that f may be greater than $\log$ without our being able to prove it. Indeed, we can produce arbitrarily large recursive functions f such that $\nvdash$ "$f(x) \geq \log x$ a.e." . Nevertheless, for the TAPE resource-bounds f usually considered, if $f(x) \geq \log x$ a.e., then we can prove it.

PROOF OF THEOREM 5.8

Suppose $\vdash$ "$f(x) \geq \log x$ a.e." , and suppose that $g \in C[f]$. We shall show that $g \in B[f]$.

For some i and some n , $M_i = g$ and

$\forall x > n \quad TAPE_i(x) \leq f(x)$.

Define a Turing machine M_j by the following algorithm :

---

[1] In Lemma 11 [8] there is no provable condition corresponding to our condition that $\vdash$ "$f(x) \geq \log x$ a.e." . It seems to us that such a provable condition is necessary for the proof to work.

1. Lay-off $\log x$ tape. Within that length of tape, seek a $y > n$ such that $TAPE_i(y) > f(y)$. (It is easy to arrange the search procedure so that if there were a $y > n$ such that $TAPE_i(y) > f(y)$ and $f(y)\downarrow$, then for sufficiently large x, the search would find such a y.) Stop when the $\log x$ tape is full.

2. If such a y is found, then output 0.

3. If no such y is found, then, re-using the $\log x$ tape, calculate $M_i(x)$ using the instruction set for M_i.

Since $\forall x > n \quad TAPE_i(x) \leq f(x)$, the algorithm always goes through step 3.

Therefore, $M_j = M_i = g$.

We now present an argument, which can be reproduced in S, to show that $TAPE_j(x) \leq f(x)$ a.e.

Suppose that $\forall y > n \quad TAPE_i(y) \leq f(y)$.

Then for every x, the algorithm goes through step 3.

Therefore, $\forall x \quad TAPE_j(x) \leq \max \{\log x, TAPE_i(x)\}$.

So, $\forall x > n \quad TAPE_j(x) \leq \max \{\log x, f(x)\}$.

Since $f(x) \geq \log x$ a.e., $TAPE_j(x) \leq f(x)$ a.e.

Now suppose it is not the case that $\forall y > n \quad TAPE_i(y) \leq f(y)$.

Then $\exists y > n \quad TAPE_i(y) > f(y)$ and $f(y)\downarrow$.

Because of the search procedure in step 1, for all sufficiently large x, the algorithm goes through step 2.

Therefore, $TAPE_j(x) = \log x$ a.e.

Since $f(x) \geq \log x$ a.e., $TAPE_j(x) \leq f(x)$ a.e.

Thus, $TAPE_j(x) \leq f(x)$ a.e.

The above argument can be reproduced in S.

Thus, $\vdash$ "$TAPE_j(x) \leq f(x)$ a.e."

So, $g \in B[f]$.

For general provable Blum measures we have the following result :

THEOREM 5.9 For some p-function d, for any partial recursive function f,

$$C[f] \subseteq B[d(x,f(x))] .$$

PROOF Let h be as in Theorem 1.5.

Define d by

$$d(x,y) = h(x,\max\{\log x, h(x,y)\}) .$$

Then d is a p-function.

Suppose that $g \in C[f]$.

Then, for some i, $\varphi_i = g$ and $\Phi_i \leq f$ a.e.

Let γ be as in Theorem 1.1. Then, by Theorem 1.5,

$$TAPE_{\gamma^{-1}(i)}(x) \leq h(x,\Phi_i(x)) \leq h(x,f(x)) \text{ a.e.}$$

It follows from Theorem 5.8 that, for some j,

$$M_j = M_{\gamma^{-1}(i)} \text{ and } \vdash \text{"}TAPE_j(x) \leq \max\{\log x, h(x,f(x))\} \text{ a.e."}$$

Applying Theorem 1.5 and the definition of d, we have

$$\vdash \text{"}\Phi_{\gamma(j)}(x) \leq d(x,f(x)) \text{ a.e."}$$

Now, $\varphi_{\gamma(j)} = M_j = M_{\gamma^{-1}(i)} = \varphi_i = g$.

Thus, $g \in B[d(x,f(x))]$. □

From Theorem 5.9 we can deduce

THEOREM 5.10 For infinitely many recursive functions f ,

$B[f] = C[f]$.

Thus, if we do not require that some explicit starting-point for the bounding be given, then for infinitely many resource-bounds f , what we can prove to be computable within f is equal to what is computable within f .

Actually, we can prove a stronger version of Theorem 5.10 :

THEOREM 5.10' For some p-function τ , $\forall a$

(i) $\varphi_{\tau(a)}$ is monotonically increasing and
$\forall x \ \varphi_{\tau(a)}(x) \geq \varphi_a(x)$;

(ii) $\vdash$ "if φ_a is total, then $\varphi_{\tau(a)}$ is total" ;

(iii) $B[\varphi_{\tau(a)}] = C[\varphi_{\tau(a)}]$.

The existence of the p-function τ shows that there is actually an algorithm to produce functions $f = \varphi_{\tau(a)}$ such that $B[f] = C[f]$. Part (i) tells us that f can be made arbitrarily large. From part (ii), f will be recursive when φ_a is recursive, and f will be a p-function when φ_a is provably total.

PROOF OF THEOREM 5.10'

Let d be as in Theorem 5.9. Since d is a p-function, d can be calculated by a provably total φ_r.

Let α be as in Theorem 1.4.

Define τ by $\tau(a) = \alpha(a,r)$. Then τ is a p-function.

By Theorem 1.4, for any a

$$\vdash \text{"}\forall i\ \forall x \geq i\ \ \Phi_i(x) \leq \varphi_{\tau(a)}(x) \ \text{ or } \ \Phi_i(x) > d(x,\varphi_{\tau(a)}(x))\text{"} .$$

Therefore, $B[d(x,\varphi_{\tau(a)}(x))] \subseteq B[\varphi_{\tau(a)}]$.

By Theorem 5.9, $C[\varphi_{\tau(a)}] \subseteq B[d(x,\varphi_{\tau(a)}(x))]$.

So, $B[\varphi_{\tau(a)}] = C[\varphi_{\tau(a)}]$.

It also follows from Theorem 1.4 that for any a

(i) $\varphi_{\tau(a)}$ is monotonically increasing and
$\forall x\ \ \varphi_{\tau(a)}(x) \geq \varphi_a(x)$;

(ii) $\vdash$ "if φ_a is total, then $\varphi_{\tau(a)}$ is total". □

The proofs of Theorems 5.4' and 5.10' can be combined to yield

THEOREM 5.11 For infinitely many recursive functions f,

$$A[f] \subsetneq B[f] = C[f] .$$

Thus for infinitely many resource-bounds f, what we can prove to be computable within f will differ depending on whether we require that an explicit starting-point for the bounding be given or simply ask that almost-everywhere bounding be demonstrated.

Although our results so far for the A-classes and the B-classes have shown differences, Theorem 5.11 is the first result to actually demonstrate that the two types of classes are different.

PROOF OF THEOREM 5.11

Let g and d be as in results 5.3 and 5.9 respectively.

Define m by $m(x,y) = \max \{g(x,y),d(x,y)\}$.

Then m is a p-function and can be calculated by a provably total φ_r .

Let α be as in Theorem 1.4.

Define τ by $\tau(a) = \alpha(a,r)$. Then τ is a p-function.

We can now follow the proofs of Theorems 5.4' and 5.10' to show that for any a

(i) $\varphi_{\tau(a)}$ is monotonically increasing and
$\forall x \ \varphi_{\tau(a)}(x) \geq \varphi_a(x)$;

(ii) $\vdash$ "if φ_a is total, then $\varphi_{\tau(a)}$ is total" ;

(iii) $B[\varphi_{\tau(a)}] = C[\varphi_{\tau(a)}]$;

(iv) if φ_a is recursive, then $\varphi_{\tau(a)}$ is recursive and
$A[\varphi_{\tau(a)}] \subsetneq C[\varphi_{\tau(a)}]$. □

We showed in Theorem 5.6 that for some measures,

$\forall i \ A[\Phi_i] = B[\Phi_i] = C[\Phi_i]$.

On the basis of Theorem 5.6 and of Theorem 7 [8], it may be wondered whether the complexity classes of the form $C[\Phi_i]$ are the only ones for which there can be equality with the corresponding provable complexity classes. In the case of the B-classes, we can show that for many measures this is not so.

First, some preliminaries.

DEFINITION A Blum measure Φ is said to be proper if $\forall i \ \Phi_i \in C[\Phi_i]$.

Many natural measures are proper - for example, the TIME and TAPE measures are proper.

LEMMA 5.12 For some p-function k

(i) $k(x,y)$ is monotonically increasing in y ;

(ii) $\forall t$ if φ_t is defined on an infinite domain, then

$C[\varphi_t] \subsetneq C[k(x,\Phi_t(x))]$.

PROOF We can define a p-function κ such that

$$\forall t \ \forall x \ \varphi_{\kappa(t)}(x) = 1 + \max \{\varphi_i(x) \mid 0 \leq i \leq x \text{ and } \Phi_i(x) \leq \varphi_t(x)\} .$$

Note that if $\varphi_t(x)\downarrow$, then $\varphi_{\kappa(t)}(x)\downarrow$.

Suppose that φ_t is defined on an infinite domain and that $\Phi_i \leq \varphi_t$ a.e.

Then for some x , $\varphi_t(x)\downarrow$, $i \leq x$ and $\Phi_i(x) \leq \varphi_t(x)$.

Therefore, $\varphi_{\kappa(t)}(x)\downarrow$ and $\varphi_{\kappa(t)}(x) > \varphi_i(x)$.

So, $\varphi_{\kappa(t)} \neq \varphi_i$.

Thus, if φ_t is defined on an infinite domain, then $\varphi_{\kappa(t)} \notin C[\varphi_t]$.

Define k by

$$k(x,y) = \max \{\varphi_r(x), \Phi_{\kappa(r)}(x) \mid 0 \leq r \leq x \text{ and } \Phi_r(x) \leq y\} .$$

Then k is a p-function and $k(x,y)$ is monotonically increasing in y .

Further, $\forall t\ \forall x \geq t \quad \varphi_t(x) \leq k(x,\Phi_t(x))$ and

$$\Phi_{\kappa(t)}(x) \leq k(x,\Phi_t(x)) .$$

So, if φ_t is defined on an infinite domain, then

$$C[\varphi_t] \subsetneq C[k(x,\Phi_t(x))] . \qquad \square$$

We can now show that for proper provable Blum measures there are classes $C[f]$ that are not equal to some $C[\Phi_i]$ and for which $B[f] = C[f]$.

THEOREM 5.13 Let Φ be proper.
Then for infinitely many recursive functions f ,

$$A[f] \subsetneq B[f] = C[f] \text{ and}$$

$$\forall i \quad C[\Phi_i] \neq C[f] .$$

PROOF Let g , d and k be as in results 5.3, 5.9 and 5.12 respectively.
Define m by $m(x,y) = \max \{g(x,y),d(x,y),k(x,y)\}$.
Then m is a p-function and can be calculated by a provably total φ_r .

Let α be as in Theorem 1.4.
Define τ by $\tau(a) = \alpha(a,r)$. Then τ is a p-function.

We can now show that for any a

(i) $\varphi_{\tau(a)}$ is monotonically increasing and
$\forall x \quad \varphi_{\tau(a)}(x) \geq \varphi_a(x)$;

(ii) $\vdash$ "if φ_a is total, then $\varphi_{\tau(a)}$ is total" ;

(iii) $B[\varphi_{\tau(a)}] = C[\varphi_{\tau(a)}]$;

(iv) if φ_a is recursive, then $\varphi_{\tau(a)}$ is recursive and $A[\varphi_{\tau(a)}] \subsetneq C[\varphi_{\tau(a)}]$;

(v) if Φ is proper and φ_a is recursive, then $\forall i \; C[\Phi_i] \neq C[\varphi_{\tau(a)}]$.

(i) to (iv) follow as in the proof of Theorem 5.11.

We now prove (v).

Suppose that Φ is proper, that φ_a is recursive and that $C[\Phi_i] = C[\varphi_{\tau(a)}]$. We shall establish a contradiction.

Since Φ is proper, $\Phi_i \in C[\Phi_i] = C[\varphi_{\tau(a)}]$.

Therefore, for some j , $\varphi_j = \Phi_i$ and $\Phi_j \leq \varphi_{\tau(a)}$ a.e.

Since φ_a is recursive, $\varphi_{\tau(a)}$ is recursive.

Therefore, $\varphi_j(x)\downarrow$ a.e. x .

By Lemma 5.12, $C[\varphi_j] \subsetneq C[k(x,\Phi_j(x)]$.

Since k is monotonically increasing in its second argument,

$k(x,\Phi_j(x)) \leq k(x,\varphi_{\tau(a)}(x))$ a.e.

Therefore, $C[\varphi_j] \subsetneq C[k(x,\varphi_{\tau(a)}(x))]$.

Now, by Theorem 1.4, $C[m(x,\varphi_{\tau(a)}(x))] \subseteq C[\varphi_{\tau(a)}]$.

Therefore, $C[k(x,\varphi_{\tau(a)}(x))] \subseteq C[\varphi_{\tau(a)}]$.

So, $C[\varphi_j] \subsetneq C[\varphi_{\tau(a)}]$.

That is, $C[\Phi_i] \subsetneq C[\varphi_{\tau(a)}]$.

But this contradicts our supposition that $C[\Phi_i] = C[\varphi_{\tau(a)}]$. □

It is perhaps worth noting that the argument to show (v) above can be adapted to give a proof that

THEOREM For any proper Blum measure Φ , the functions Φ_i do not form a class-determining set.

This proof differs from the usual proofs of the theorem in that it uses the Gap Theorem rather than the more difficult Union Theorem. (See [13].)

As for showing differences between B-classes and C-classes, the best we have is

THEOREM 5.14 There exist provable Blum measures Φ and recursive functions f such that

$$B[f] \subsetneq C[f] .$$

To prove Theorem 5.14, we need a few preliminaries.

DEFINITION A set F of partial recursive functions is said to be recursively presentable if for some recursively enumerable $Y \subseteq \mathbb{N}$, $F = \{\varphi_i \mid i \in Y\}$.

LEMMA 5.15 For any partial recursive function f,

$B[f]$ is recursively presentable.

PROOF An appropriate set Y is enumerated by the following algorithm :

Generate the theorems of S . Whenever a theorem of the form "$\Phi_i \leq f$ a.e." is generated, output i .

The following result appears in [10].

THEOREM There exist Blum measures Φ and recursive functions f such that $C[f]$ is not recursively presentable.

In the proof of this Theorem, a Blum measure Φ is constructed for which $C[0]$ is not recursively presentable.

It is easy to check that Φ is a provable Blum measure.

By Lemma 5.15, $B[0] \subsetneq C[0]$.

So, Theorem 5.14 is established.

CONCLUSION

The study of provable conditions in computational complexity provides new insights into the nature of problems in the complexity of algorithms, for it considers the limitations on what we (working as we do within a formal axiomatic system) can come to know about an algorithm's properties. The main thrust of this thesis is that for many natural questions in computational complexity what we can come to know, that is, what we can formally prove, falls unpleasantly short of what is actually true.

The results in Chapter 3 show that what we can establish about the complexity properties of a partial recursive function f depends on the algorithm we initially use to define f.

For every partial recursive function f, there exist anomalous defining algorithms - anomalous because of the discrepancy between what is true about the algorithms and what can be proved about them. We can never know what limitations our particular defining algorithm for f imposes on us, for the only algorithms that we can recognize as calculating f are those provably equivalent to our defining algorithm for f. Anomalous algorithms exist among the provably total algorithms and also among the 'very fast' algorithms. It is only a hope that intuitively natural defining algorithms are not anomalous.

In Chapter 4 we investigated the relationship between provable equivalence and the computational complexity of algorithms. This relationship is complex and not readily summarized, but it is closely

involved with provable relationships between the domains over which algorithms are defined. A general conclusion we can draw from the results in this chapter is that as the difference between the complexities of algorithms increases, what we can prove about the relationships between the algorithms decreases.

Chapter 5 concerned provable analogues of complexity classes. The differences between the B-classes and the A-classes show that what can be proved to be computable within a given resource bound may differ depending on whether we ask merely that almost-everywhere bounding be demonstrated or require further that an explicit starting-point for the bounding be given.

In a practical situation, it seems natural to require that an explicit starting-point for the bounding be given. In that case, the differences between the C-classes and the A-classes show that for infinitely many recursive functions there will be a discrepancy between what is true and what can be proved about the complexity of the function, no matter what algorithm is used to define the function. Furthermore, such discrepancies will occur even among those functions that can be proved to be total.

Our work here has left some obvious open questions - for example, are there functions f such that $A[f] = C[f]$? Perhaps answering such questions requires only the invention of more ingenious algorithms, or perhaps it awaits the introduction of provable conditions into deeper areas of complexity theory, such as the rich theorems relating recursive enumerability of classes to complexity properties.

Beyond these things, however, lies the haunting question of what it is about an algorithm that limits our ability (working within our formal system) to analyse its complexity properties. Certainly, very complex algorithms are difficult for us to analyse, but the answer is not that simple since, as we showed in Chapter 3, anomalous algorithms are to be found even among the LOG-SPACE and LINEAR-TIME algorithms. Clearly, a much deeper understanding of the relationship between formal provability and computational complexity is required.

BIBLIOGRAPHY

1. Baker, T.P.
On "provable" analogs of P and NP .
Math. Systems Theory 12 (1979), 213-218.

2. Biskup, J.
The TIME measure of one-tape Turing machines does not have the parallel computation property.
SIAM J. Comp. 7, 1 (Feb. 1978), 115-118.

3. Blum, M.
A machine-independent theory of the complexity of recursive functions.
J. ACM 14, 2 (April, 1967), 322-336.

4. Boolos, G. and Jeffrey, R.
Computability and Logic.
Cambridge University Press, 1974.

5. Fischer, P.C.
Theory of provable recursive functions.
Trans. AMS 117 (1965), 494-520.

6. Garey, M.R. and Johnson, D.S.
Computers and Intractability.
W.H. Freeman and Company, 1979.

7. Gordon, D.
Complexity classes of provable recursive functions.
J. Comp. Sys. Sci. 18, 3 (1979), 294-303.

8. Hartmanis, J.
Relations between diagonalization, proof systems and complexity gaps.
Theor. Comp. Sci. 8 (1979), 239-253.

9. Hartmanis, J.
Feasible computations and provable complexity properties.
SIAM 1978 monograph.

10. Hartmanis, J. and Hopcroft, J.E.
An overview of the theory of computational complexity.
J. ACM 18, 3 (July 1971), 444-475.

11. Hartmanis, J. and Hopcroft, J.E.
Independence results in computer science.
ACM SIGACT News 8 (Oct.-Dec. 1976), 13-24.

12. Kleene, S.C.
Introduction to Metamathematics.
North-Holland Publishing Company, 1967.

13. McCreight, E.M. and Meyer, A.R.
Classes of computable functions defined by bounds on computation.
Conference Record 1st ACM Symp. on Theory of Computing, 1969, 79-88.

14. Rogers, H. Jr.
Gödel numberings of partial recursive functions.
J. Symbolic Logic 23, 3 (Sep. 1958), 331-341.

15. Rogers, H. Jr.
Theory of Recursive Functions and Effective Computability.
McGraw-Hill Book Company, 1967.

16. Sachse-Åkerlind, D.
Anomalous algorithms and provable complexity properties.
Tech. Report TR-CS-82-16, Dept. of Computer Science, A.N.U., (Dec. 1982) 23pp.

17. Sachse-Åkerlind, D.
Computational complexity and provable equivalence of algorithms.
Tech. Report TR-CS-83-03, Dept. of Computer Science, A.N.U., (Feb. 1983) 22pp.

18. Sachse-Åkerlind, D.
Relations between complexity classes and provable complexity classes.
Tech. Report TR-CS-83-04, Dept. of Computer Science, A.N.U., (Feb. 1983) 43pp.

19. Young, P.
Easy constructions in complexity theory : Gap and Speed-up theorems.
Proc. AMS 37, 2 (Feb. 1973), 555-563.

20. Young, P.
Optimization among provably equivalent programs.
J. ACM 24, 4 (Oct. 1977), 693-700.