1975

Optimization Among Provably Equivalent Programs

Paul Young

Report Number:
75-156

OPTIMIZATION AMONG PROVABLY EQUIVALENT PROGRAMS

Paul Young
Computer Sciences and Mathematics Department
Purdue University
Lafayette, Indiana 47907

CSD-TR 156

Abstract. We consider the extent to which it is possible, given a program p for computing a function, f, to find an optimal program p' which also computes f and is either provably equivalent to p or else provably an optimal program. Our methods and problems come chiefly from abstract recursion-theoretic complexity theory, but some of our results may be viewed as directly challenging the intuitive interpretation of earlier results in the area.

Key words and phrases: abstract complexity, theory, proving equivalence or correctness of programs.

OPTIMIZATION AMONG PROVABLY EQUIVALENT PROGRAMS

Paul Young[1]
Computer Sciences and Mathematics Departments
Purdue University
Lafayette, Indiana 47907

1. Motivation

In traditional recursion-theoretic studies of complexity theory, two programs are regarded as equivalent if they compute the same function. The efficiency of the running time of a program is studied by comparing its (ultimate, a.e.) behavior with the running times of all other programs for the same function.

It is not very difficult to find programs p and p' which in fact both compute the same function, which can both be proven to halt on all inputs, yet which can not be proven to compute the same function. Suppose then that one is trying to optimize some class of programs, starts with a program p which is known to compute some desired function, f, and then is given program p' as a better program for f. The user must regard the situation as unsatisfactory: off hand, his only method for verifying that the use of program p' is legitimate is: after having run p' on a given input x, he must then run program p on input x to verify that p' is correct on that input.

In practice then, it seems clear that if one wants to consider program p' as a possible optimization of program p, one first needs a proof that p and p' really perform the same task. (In this paper, we assume that this means that we can prove that programs p and p' really do compute the same function, i.e., p and p' are provably equivalent.)

We limit our attention to programs for total functions, and for the purpose of this study we do not question the usual recursion-theoretic assumptions that (i) all programs are legitimate objects of study, and (ii) infinite functions and the ultimate behavior of run times on large argument yield useful

---

[1]Supported by NSF Grant GJ 27127A1. Research performed while the author was a visiting professor in the Electrical Engineering and Computer Sciences Department, University of California, Berkeley, 1972-73.

insights into computational complexity. Our work may be viewed as questioning the traditional view that any two programs which compute the same function should be regarded as equivalent. If this view is taken seriously, it implies that many results of recursion theoretic complexity theory should be reexamined. In the long run, work such as this may have implications for studies of how one proves equivalence of programs (Floyd, Manna, etc.), but we make no such explicit claims here.

In intuitive terms, our two main results may be stated as follows:

i) Every computable function has very good programs which are (nearly) optimal (among the provably equivalent programs).

ii) Every computable function has good classes of provably equivalent programs in which it is possible to effectively find very large speed-ups and to prove both the correctness and the good running times of the sped-up programs.

Thus from any practical standpoint, the traditional view that questions of optimality and speed-up are properties of the functions to be computed is erroneous: from our point of view, questions of optimality and speed-up are completely independent of the function to be computed, but depend instead on the description of the program used to compute the function. With a little reflection, we believe the reader will find these results not at all surprising.

We also observe that the standard method [M-F] of constructing functions with well controlled speed-up cannot possibly produce very good programs for which it is possible to have many provably equivalent programs which are sped-up. Thus if we are concerned about provably equivalent programs, this standard method does not produce any "good" infinite chains of speed-ups. (Of course, by (i) above, not all good programs can have infinite chains of provably equivalent speed-ups, even when the function itself has speed-up.)

II. Notation

In stating our results, we use standard intuitive and formal terminology for abstract complexity theory. Our proof theory for proving equivalence of programs is any formal mathematical system (so the set of theorems is recursively enumerable), which is adequate for carrying out elementary arithmetical arguments and which is sound for arithmetic (no false arithmetic statements

are provable). For examples, first order Peano-arithmetic meets these requirements, and it is commonly believed that any of the standard axiomatic systems for all of set theory meet these requirements. Although the reader is free to choose any proof system and set of axioms he pleases within these constraints, this system is to remain fixed throughout this paper.

If S is a sentence which is provable in this fixed theory, we write "$\vdash$ S". We state our results, not in terms of an arbitrary standard indexing of the partial recursive functions, but instead in terms of any provably standard indexing. That is an indexing for which we have a partial recursive function U and a total recursive function S and computable pairing function <, >, for which

$\vdash$'S is total'

$\vdash$'< > is a total one-one onto function from NxN→N'

$\vdash'(\forall i)(\forall x)(\forall y)[U(i,<x,y>)=U(S(<i,x>),y)]'$

$\vdash$'if ψ is a partial function computable by a Turing machine, then there exists an integer e_o such that $\psi = \lambda x U(e_o,x)$'

We state without further comment the following fact, which should be clear to the reader after a little reflection. Every indexing of the partial recursive functions mentioned in the literature as a reasonable model for computing all partial recursive functions, including, e.g., Turing machines and ALGOL, is provably standard. We follow standard recursion theoretic procedure and abbreviate $\lambda x U(i,x)$ as ϕ_i. We frequently drop universal quantifiers, e.g., writing line 3 above as

$\vdash' U(i,<x,y>)=U(S(<i,x>),y)'$

Furthermore, examination of the standard proof of the recursion theorem proves that there is a total recursive function n with the following properties:

$\vdash$'n is total'

$\vdash'\phi_i$ total implies $\phi_{\phi_i(n(i))}=\phi_{n(i)}'$

We use such facts without further proof.

Finally, we consider, not arbitrary Blum measures for computational complexity, but instead only provable measures; specifically, we require

$\vdash'(\forall i)[\text{domain } \phi_i=\text{domain } \Phi_i]'$

and $\vdash$'the relation $\Phi_i(x)\leq y$ is decidable.'

Again, no examples of nonprovable measures have ever been seriously proposed in the literature.

III Results

The proof of Theorem 2 depends on the following somewhat paradoxical result. The reader should bear in mind that a result of this form is possible only because (as is well-known) our proof methods cannot be strong enough to prove their own correctness.

Theorem 1. There is an effective procedure which, given any program p finds a program $\sigma(p)$ which does compute the same function as p and whose run time in the limit is (almost) as good as that of any program provably equivalent to p.

Note in particular that if every program provably equivalent to p should have a large speed-up ppovably equivalent to it, then we would effectively have found a program computing the same function as does p, but doing so more rapidly than any program provably equivalent to p. (Hence we could not prove (in our formal proof theory) that p and $\sigma(p)$ compute the same function.) Theorem 1 is proven by a straightforward enumeration and simulation of all programs provably equivalent to p, which we now outline.

In multi-tape Turing machine time, we might, for example, define $\phi_{\sigma(p)}$ as follows: to compute $\phi_{\sigma(p)}(x)$, do log (x) steps in the enumeration of theorems to obtain a list of programs p, p_0, p_1, p_2, ..., p_m ($m<\log x$) provably equivalent to program p. Then begin dovetailing the computations of $\phi_p(x)$, $\phi_{p_0}(x), \ldots, \phi_{p_m}(x)$, taking as output for $\phi_{\sigma(p)}(x)$ the first output obtained from these dovetailed computations. Thus, if $\phi_p=\phi_{p(i)}$, then for all sufficiently large x

$$\phi_{\sigma(p)}(x) \leq \log(x)(\phi_{p(i)}(x))^2.$$

Furthermore, since for all i, $\vdash'\phi_p=\phi_{p_i}'$, if we believe that our proof theory is sound, we believe that $\phi_p=\phi_{\sigma(p)}$. In fact, for each fixed x, $\phi_p(x)=\phi_{\sigma(p)}(x)$, even if $\phi_p(x)$ is undefined. However, since in general we cannot hope that $\vdash'(\forall i)[\phi_p=\phi_{p_i}]'$, there is no a priori method to obtain $\vdash'\phi_p=\phi_{\sigma(p)}'$, and in general we shall see that we cannot prove the latter statement.

A careful statement of Theorem 1 would read as follows: For any provably standard indexing and any provable measure, there are provably total recursive functions σ and r such that for all p, $\phi_p=\phi_{\sigma(p)}$ and for any p' such that $\vdash'\phi_p=\phi_{p'}'$, $\phi_{\sigma(p)}(x) \leq r(x, \phi_{p'}(x))$ a.e.

Our next theorem guarantees that all functions have near optimal programs.

Theorem 2. (Optimization) Given any program p, there is an equivalent program p' for which (i) the running time of p' is not much worse than the running time of p (and may be much better) and (ii) among all the programs provably equivalent to p', p' has a (nearly) optimal running time (a.e.). p' can be effectively found from p.

Theorem 2 has an obvious Corollary, which we state in highly intuitive terms:

Corollary 1. For every function f which has speed-up among all its programs, there are programs p for f which have very good running times and which have, among all the programs provably equivalent to p, (nearly) optimal running times (a.e.).

Corollary 1 suggests both that the speed-up phenomenon of Blum is highly complicated and that its intuitive implications are not easily summarized.

The proof of theorem 2, which we now sketch, is similar to the proof of Theorem 1. Given the program p, we let p_o be either p or $\sigma(p)$ where σ is as in Theorem 1. (We may (non-effectively) choose p_o to keep ϕ_{p_o} the smaller of ϕ_p or $\phi_{\sigma(p)}$.) We define a total recursive function f as follows: to compute $\phi_{f(i)}(x)$, do log x steps enumerating theorems, letting $p_1, p_2, p_3, \ldots, p_m$ ($m<\log x$) be the programs for which we obtain

$\vdash' \phi_i = \phi_{p_j}$' in these log x steps.

Set

$$\phi_{f(i)}(x) = \phi_{p_{j_o}}(x) + j_o \quad \text{where } j_o \text{ is chosen so that } \phi_{p_{j_o}}(x) = \min_{0\le j\le m} [\phi_{p_j}(x)].$$

The function f is provably total, so we may find a fixed point i_o for which $\phi_{f(i_o)} = \phi_{i_o}$. Since for all $j>0$, $\vdash' \phi_{i_o} = \phi_{p_j}$', if we believe the soundness of our proof theory, we must have that for all x, the chosen j_o is $p_o=0$. Thus $\phi_{i_o} = \phi_{f(i_o)} = \phi_{p_o}$. Furthermore, to achieve this, we must have for any $j\ge 1$ for which $\vdash' \phi_{f(j_o)} = \phi_{i_o} = \phi_{p_j}$' that $\phi_{p_j}(x) \ge \phi_{p_o}(x)$ for all sufficiently large x. But by the construction of f, $\phi_{f(i_o)}$ is never much greater than ϕ_{p_o}, showing that $f(i_o)$ is near optimal among the programs provably equivalent to it.

Corollary 1 clearly applies not just to functions with speed-up almost everywhere (a.e.) but also to functions which have speed-up infinitely often, (i.o.). In this version, it contrasts very sharply with Blum's result [B-2] that there are functions which not only have large i.o. speed-up, but for which programs for this process can be effectively found. Theorem 2 guarantees that although we can, given a program p for one of Blum's functions f effectively find $\sigma(p)$, a program for f which is an i.o. speed-up of f, the task of proving for such a p that p and $\sigma(p)$ actually compute the same function is hopeless! We state this as

Corollary 2. Let f be a function with sufficiently large i.o. speed-up, and let σ be any algorithm which, given a program p for f, finds a program $\sigma(p)$ which also computes f and which is an i.o. speed-up of p. Then there are (lots of) programs p for f which have good running times but for which we cannot prove that p and $\sigma(p)$ compute the same function.

Corollary 2 is consistent with Blum's results on effective i.o. speed-ups because Blum constructs a function f with i.o. speed-up by giving a specific program p_o for f and constructing his algorithm σ so that if p computes f, so does $\sigma(p)$. (Since in general we cannot, for an arbitrary program p for f, prove that p and p_o compute the same function, we cannot in general prove that p and $\sigma(p)$ compute the same function!) As pointed out to us by Albert Meyer, examination of Blum's proof must prove that not every class of provably equivalent programs contains a program with a (near) optimal a.e. running time. However, as we next show, a much stronger result holds: every function has classes of provably equivalent programs with arbitrarily large speed-up in the classes. The proof of this result is similar to any of the standard proofs of speed-up [B-1], [M-F],[Y], but surprisingly simpler than these usual proofs. (Its proof is by a cancellation argument in which nothing gets cancelled!!)

Theorem 3. (Speed-up). Let $r(x,y)$ be any total function. Let p be any program for any total function. From p we can effectively find a program p' such that

(i) p_o and p' compute the same function,

(ii) for any p'' for which we can prove that p'' and p' compute the same function we can effectively find a p''' for which we can prove that

(ii,a) 'p''' and p'' compute the same function', and

(ii,b) 'p''' is an r-speed-up of p'' on the run times of p''' for which r is defined.'

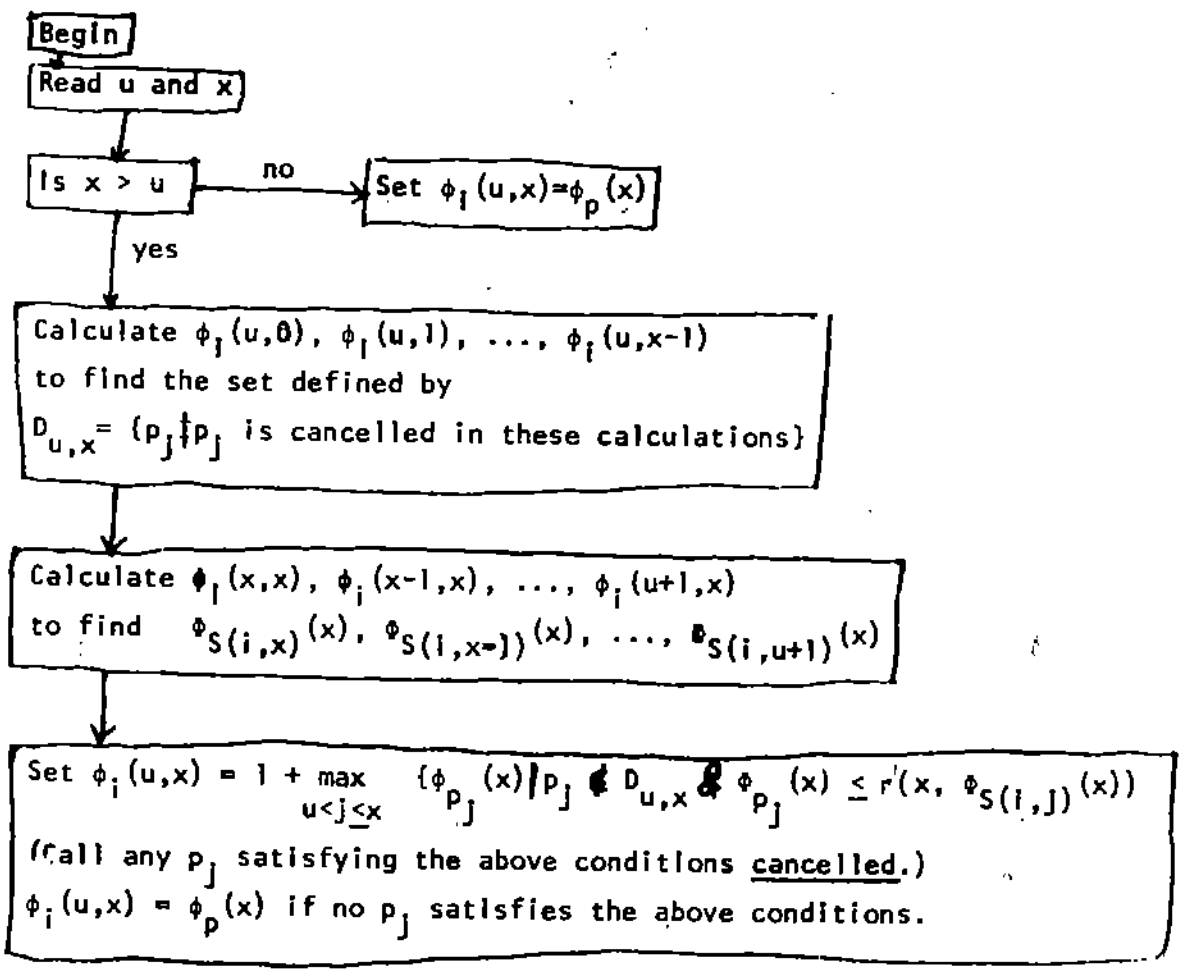
Furthermore, if r is honest, the run times of the programs provably equivalent to p' are "about" as fast as any program provably equivalent to p.

Proof. We implicitly employ the recursion theorem to obtain a function ϕ_i. For any i (and in particular for the program i obtained by the recursion theorem) we will be able to list p_1, p_2, p_3, ..., an infinite list, possibly with repetions, of all programs provably equivalent to S(i,0). Here S is the S_1^1 function for which

$$\vdash'(\forall i)(\forall n)(\forall x)[\phi_{S(i,n)}(x) \simeq \phi_i(n,x)]'.$$

p is as in the statement of the theorem.

We consider ϕ_i to be a function of two variables where computation rules are given by the following flowchart:



Now in the event that r is provably total, we readily see that for any fixed u,

$\vdash$ 'λx $\phi_i(0,x) \simeq \lambda x\ \phi_i(u,x)$ unless some p_j with $i \leq j \leq u$ is cancelled while calculating λx $\phi_i(0,x)$.'

But for each j, $i \leq j \leq u$,

$\vdash$ '$\phi_{S(i,0)} \simeq \phi_{p_j}$ & λx $\phi_i(0,x) \simeq \phi_{S(i,0)}$.'

Examining the construction therefore immediately yields for each j, $i \leq j \leq u$,

$\vdash$ 'p_j is not cancelled while calculating λx $\phi_i(0,x)$.'

Hence for each fixed u,

$\vdash$ '$\phi_{S(i,0)} \simeq \phi_{S(i,u)}$.'

Furthermore, for each fixed j,

$\vdash$ '$\phi_{p_j} \simeq \phi_{S(i,0)} \simeq \phi_{S(i,j)}$'

while $\vdash$ 'p_j is not cancelled while calculating λx $\phi_i(0,x)$,'

immediately yields by examining the construction,

(2) $\vdash$ '$(\forall x)[x \geq j \Longrightarrow r(x, \phi_{S(i,j)}(x)) < \phi_{p_j}(x)]$.'

The proof of (i) and (ii) is now complete once we observe that, since for each j, $\vdash$ 'p_j is not cancelled,' if we believe in the soundness of our proof theory, it follows that in fact no p_j is cancelled, so from the construction we see that for every u, $\phi_{S(i,u)} \simeq \phi_p$. (An easy double induction, similar to one given below, is used to show that if p and r are total, so is ϕ_i. The bases of the induction are that $(\lambda u)\phi_i(u,0) \simeq \phi_p$ and $(\forall x_o)[u \geq x_o \Longrightarrow \phi_i(u,x_o) \simeq \phi_p(x_o)]$.)

The final remark follows, because from Theorem 1, we may assume that p is already about as fast as any program provably equivalent to p, while from the construction, $\phi_{S(i,u)}(x) \approx r^{x-u} \circ \phi_p$ and in the event that p is nearly an optimal program, we can hardly expect to obtain equivalent programs with r-speed-up and better run times than this.

To complete the proof for the case when r is not provably total, we now assume (as we may without loss of generality) that r is provably honest (or near honest) and provable monotone. Under these dondtions we may interpret

$$\phi_{p_j}(x) \leq r\ (x, \phi_{S(i,j)}(x))$$

as meaning either that all of the above calculations terminate and that the inequality holds or that $\phi_{p_j}(x)$ is defined but one of $\phi_{S(i,j)}(x)$ or

$r(x, \Phi_{S(i,j)}(x))$ is undefined. Thus for any x, successful computation of $\phi_{p_j}(x)$ leads to cancelling p_j unless both $\Phi_{S(i,j)}(x)$ and $r(x, \Phi_{S(i,j)}(x))$ are defined and

$$\Phi_{p_j}(x) > r(x, \Phi_{S(i,j)}(x)).$$

Thus, the proof goes about as before, except that for fixed u, we see that for all j, $1 \leq j \leq u$

$$\vdash' \lambda x\ \phi_i(0,x) = \phi_{S(i,0)} = \phi_{p_j}'$$

$$\vdash' (\forall x)[\phi_i(0,x) \text{ defined implies } \phi_{p_j}(x) \text{ defined})'$$

Also $\vdash' \phi_{S(i,0)} = \phi_{p_j}$ so p_j is not cancelled'

$$\vdash' (\forall x)[\phi_i(0,x) \text{ defined implies } \Phi_{S(i,j)}(x) \text{ defined } \underline{\text{and}}\ r(x, \Phi_{S(i,j)}(x)) \text{ defined, } \underline{\text{and}}\ \Phi_{p_j}(x) > r(x, \Phi_{S(i,j)}(x)) \text{ for all } x \geq j.]'$$

From this, for each fixed u

$$\vdash' (\lambda x)\ \phi_i(0,x) = \lambda x\ \phi_i(u,x),'$$

completing the proof.

Although Theorem 3, guarantees that we can, in a very strong sense obtain infinite chains of provably equivalent speed-ups, the standard methods of constructing functions with speed-up place some limits on how good the run-time of such infinite chains can be:

Theorem 4. There are arbitrarily large total recursive functions r for which there is a total recursive function f which does have h-speed-up, but which also has a fixed program p such that for any program p_0 with $\phi_p = \phi_{p_0} = f$ and $\Phi_{p_0} \leq \Phi_p$ a.e. there is no infinite sequence of programs $p_1, p_2, p_3, p_4, \ldots$ such that each program computes f and for all $i \geq 0$ (a) $\vdash' \phi_{p_0} = \phi_{p_i}'$ and (b) $\Phi_{p_i}(x) > r(x, \Phi_{p_{i+1}}(x))$ a.e. (In fact, from the size of program p_0, we can effectively bound the number of programs in any such r-decreasing sequence of provably equivalent programs with run times below Φ_{p_0}.)

The proof of this is relatively straightforward. The standard method ([M-F]) for constructing functions, f, with speed-ups start with an honest function r and makes the good run times of the function f cofinal in the infinite

complexity sequence of functions $S_0 > S_1 > S_2 \ldots$ where $p_i(x) = S^{x-i}(x)$. If we take any program p for f for which $\phi_p \leq p_o$, whenever p_o computes f and $\phi_{p_o} \leq \phi_{p'}$ we observe that for σ as in Theorem 1, $\sigma(p_o)$ computes f and is roughly at least as fast as any program provable equivalent to p_o. Since the sequence $S_0, S_1, \ldots$ is cofinal in the good run times of f, $\phi_{\sigma(p_o)} \geq S_i$ for some i. This proves the theorem except for the parenthetical remark. To actually calculate the bound, one must do the proof from scratch, based on the speed-up construction in [H-Y] where techniques for calculating bounds in complexity sequences are introduced. (If one wishes a similar calculated bound for speed-ups simply by recursive functions one adopts the extension of these techniques used in [M-F].) We forgo the details, since they are lengthy.

We now understand several theoretical reasons why we may be unable to find optimal programs for a function. By the standard speed-up Theorem [B-1], optimal programs may not exist. By Theorem 3, even when optimal programs exist for a function, we may be unable to find them if we start with a program which is not provably equivalent to an optimal program. Theorem 5 gives yet another reason.

Theorem 5. For every program p_i we can effectively find a program p' such that $\phi_p = \phi_{p'}$, the run times of p and p' are about the same, and p' can not be proven to be near optimal. (In particular if p is optimal or near optimal, p' is a near optimal program which cannot be proven to be nearly optimal.)

Proof. Employing the recursion theorem, we define p' as follows: on input x, p' spends about log x steps attempting to prove its own optimality. If it doesn't succeed, it simulates p on x, taking about $\phi_p(x) + \log x$ steps (in standard time measure). If it does succeed, it simulates p'' on x where p'' is a program for which $\phi_p = \phi_{p''}$ but the computation of $\phi_{p''}$ by p'' wastes lots of resource and hence is not optimal. The result follows directly. (With care we may obviously have $\vdash \phi_p = \phi_{p'}$!)

Finally, we remark that some functions which do have (near) optimal computational methods have no programs which compute the function which can be proven near optimal:

Theorem 6. There is a total recursive function r and a total recursive function f with program p such that

(i) $\phi_i = p$ implies $r(x,\phi_i(x)) \geq \phi_p(x)$ a.e.

and (ii) if $[\phi_p = \phi'_i \Longrightarrow r(x,\phi_i(x)) \geq \phi_p(x)$ a.e.'

then $\phi_p \neq f$.

Proof. This result is a direct consequence of a result of Albert Meyer's listed as Theorem 1 of [G-B]. Specifically, Meyer's result states that for every sufficiently large total recursive function, h, every r.e. class of a.e. h-complex partial recursive functions (for example, the provably near-optimal functions which are at least h-complex) omits arbitrarily complex 0-1 valued recursive functions, f. Meyer's proof actually proceeds as follows: Given a total recursive function t and an enumeration $\sigma(0)$, $\sigma(1)$, $\sigma(2)$,... of programs for a.e. h-complex functions, Meyer constructs f so that f is at least t difficult a.e. but f fails to agree with any of $\phi_{\sigma(0)}$, $\phi_{\sigma(1)}$, $\phi_{\sigma(2)}$, ... But examination of Meyer's construction of f makes clear that f can be chosen so that it is not much more difficult than t. Thus f is near optimal.

Suppose, for example that P_0, P_1, P_2, P_3, ... is a list of programs which are provably "near" optimal, where "near" here means the fixed recursive functions h such that the preceeding t and f of Meyer's proof satisfies $\phi_f \leq h \cdot t$ a.e., where ϕ_f is the difficulty of a good way of computing f. In reasonable measures, we can expand P_0, P_1, P_2, ... to a list P_{ij} $(0 \leq i < \infty\ ;\ 0 \leq , < \infty)$ such that $P_i = P_{ij}$ for all i and j, $\phi_{P_i} = \phi_{P_{ij}}$ a.e. and P_{ij} is obtained from P_i by computing $\phi_{P_{ij}}(x) = \phi_{P_i}(x)$ unless $x \in D_i$ and are taking at least h(x)steps to look up the value of $\phi_{P_{ij}}(x)$ in a table if $x \in P_j$, where P_j is the j^{th} canonically enumerable finite set. Then if g is a function which is provably near optimal and h-complex a. e., there must be some program in the list P_{ij} which computes g and takes at least h steps everywhere. Thus, from the list P_{ij} it is possible to extract a new list of programs $\sigma(0)$, $\sigma(1)$, $\sigma(2)$, ... such that every program $\sigma(i)$ in the list satisfies $\phi_{\sigma(i)} \geq h$ everywhere and $\phi_{\sigma(i)} = \phi_{P_j}$ for some j. Furthermore, for every j, if $\phi_{P_j} \geq h$, a. e. then $\phi_{P_j} = \phi_{\sigma(i)}$ for some i. I, e., $\phi_{\sigma(0)}, \phi_{\sigma(1)}, \phi_{\sigma(2)}$, ... is a list of the provably near optimal functions which are h-complex a.e., making Meyer's proof applicable.

Bibliography

[B-1] Blum, M., A machine independent theory of the complexity of recursive functions, JACM, 14 (1967), 322-336.

[B-2] Blum, M., On effective procedures for speeding-up algorithms, JACM, 18 (1971), 290-305.

[C] Chaitin, Gregory, Information-Theoretic limits of formal systems, JACM, 21 (1974), 403-423.

[Co] Collins, William, Elements of provably recursive analysis, Notre Dame Journal of Formal Logic, to appear.

[Co-Y] Collins, W., and Young, P., Discontinuities of provably correct computable real valued functions, to appear.

[F] Fischer, P. C., Theory of provable recursive functions, Trans. A.M.S. (1965), 117, 494-520.

[Fl] Floyd, R. W., Assigning meanings to programs, Proc. Symp. App. Math., 19 (1967), Math. Aspects Comp. Sc., Amer. Math. Soc., 19-32.

[G-B] Gill, J., and Blum, M., On almost everywhere complex recursive functions, JACM, 21 (1974), 425-435.

[H-Y] Helm, J., and Young, P., On size vs efficiency for programs admitting speed-ups, J. Symbolic Logic, 36 (1971), 21-27.

[M] Manna, Z., The correctness of programs, J. Comp. Syst. Sci. 3, (1969), 114-127.

[M-F] Meyer, A., and Fischer, P., Computational speedup by effective operators, J. Symbolic Logic, 37 (1972), 55-68.

[R-Y] Ritchie, R., and Young, P., Strong representability of partial functions in arithmetic theories, Inf. Sci. 1 (1969), 189-204.

[Y-1] Young, Paul, Easy constructions in complexity theory: gap and speed-up theorems, Proc. A. M. S., 36 (1973), 555-563.

[Y-2] Young, Paul, Optimization among provably equivalent programs: preliminary abstract, Proceedings of 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, 197-199.

ERRATA SHEET

Optimization among provably equivalent programs

Page	line	
1	b 1	arguments
3	9	That is,
3	10	drop comma
4	b 7-8	from , $\phi_p(x) = \phi_{\sigma(p)}(x)$,
		to, $\vdash' \phi_p(x) = \phi_{\sigma(p)}(x)$',
5	5	optimal
6	b 3	p_o to p
7	13	where to whose
8	b 6	when to where
11	1	which do have
11	16	e. e.