1975

Optimization Among Provably Equivalent Programs

Paul Young

Report Number:
75-156

Young, Paul, "Optimization Among Provably Equivalent Programs" (1975). Department of Computer Science Technical Reports. Paper 103.
https://docs.lib.purdue.edu/cstech/103

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
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.
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 cannot 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
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, Hanna, etc.), but we make no such
explicit claims here.

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

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

2) 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 \([H-F]\) 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 equi-
valent 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 recur-
sively enumerable), which is adequate for carrying out elementary arithmetical
arguments and which is sound for arithmetic (no false arithmetical 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

\[ S \text{ is total} \]
\[ \langle > \text{ is a total one-one onto function from } \mathbb{N} \times \mathbb{N} \]
\[ \forall (\forall l)(\forall x)(\forall y)[(U(l,\langle x, y \rangle) = U(S(<l,x>),y))] \]
\[ \text{If } \psi \text{ is a partial function computable by a Turing machine, then there exists an integer } e_0 \text{ such that } \psi = \lambda x U(e_0, x) \]

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 reasonabbles 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 $\lambda U(l,x)$ as $\phi_i$. We frequently drop universal quantifiers, e.g., writing line 3 above as

\[ U(l,\langle x, y \rangle) = U(S(<l,x>),y) \]

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

\[ n \text{ is total} \]
\[ \phi_i \text{ total implies } \phi_i(n(i)) = \phi_i(n(i)) \]

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

\[ \forall (\forall l)[\text{domain } \phi_i = \text{domain } \phi_i] \]

and $\text{the relation } \phi_i(x) \leq y \text{ is decidable}$.

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 provably 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 $\phi_p(x)$, $\phi_{p_0}(x), \phi_{p_1}(x), \phi_{p_2}(x), ...$, 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 $x$

$$\phi_\sigma(p)(x) \leq \log(x) \phi_p(i)(x) \leq 2.$$

Furthermore, since for all $i$, $\phi_p = \phi_p$, 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 $K(V_i)[\phi_p = \phi_\sigma(p)]$, there is no a priori method to obtain $K[V_i][\phi_p = \phi_\sigma(p)]$, 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 $\sigma$ and $r$ such that for all $p$, $\phi_p = \phi_\sigma(p)$ and for any $p'$ such that $\phi_p = \phi_{p'}$, $\phi_\sigma(p)(x) \leq r(x, \phi_{p'}(x))$ 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 \( p_0 \) be either \( p \) or \( \sigma(p) \) where \( \sigma \) is as in Theorem 1. (We may (non-effectively) choose \( p_0 \) to keep \( p_0 \) the smaller of \( p \) or \( \sigma(p) \).) We define a total recursive function \( f \) as follows: to compute \( f_i(x) \), do \( \log x \) steps enumerating theorems, letting \( p_1, p_2, p_3, \ldots, p_m \) be the programs for which we obtain

\[ f_i(x) = \min_{0 \leq j \leq m} \{ f_j(x) \} \]

Set

\[ \phi_f(i)(x) = \phi_{p_j}(x) + J_o \] where \( J_o \) is chosen so that \( \phi_{p_j}(x) = \min_{0 \leq j \leq m} \{ \phi_{p_j}(x) \} \).

The function \( f \) is provably total, so we may find a fixedpoint \( p \) for which

\[ \phi_f(i_0) = \phi_{i_0} \].

Since for all \( j > 0 \), \( \phi_f(i_0) \neq \phi_{j_0} \), if we believe the soundness of our proof theory, we must have that for all \( x \), the chosen \( J_o \) is \( p_o \). Thus \( \phi_{i_0} = \phi_f(i_0) = \phi_{p_o} \). Furthermore, to achieve this, we must have for any \( j \geq 1 \) for which \( \phi_f(i_0) = \phi_{j_0} \), that \( \phi_{p_j}(x) \neq \phi_{p_o}(x) \) for all sufficiently large \( x \).

But by the construction of \( f \), \( \phi_f(i_0) \) is never much greater than \( \phi_{p_o} \), showing that \( (i_0) \) 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 σ(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 σ(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 σ(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 σ(p) compute the same function.

Corollary 2 is consistent with Blum's result on effective i.o. speed-ups because Blum constructs a function f with i.o. speed-up by giving a specific program p₀ for f and constructing his algorithm σ so that if p computes f, so does σ(p). (Since in general we cannot, for an arbitrary program p for f, prove that p and p₀ compute the same function, we cannot in general prove that p and σ(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], [V], 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) p₀ 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 \( \phi_1 \). For any \( i \) (and in particular for the program \( I \) obtained by the recursion theorem) we will be able to list \( p_1, p_2, p_3, \ldots \), an infinite list, possibly with repetitions, of all programs provably equivalent to \( S(1,0) \). Here \( S \) is the \( S_1 \) function for which

\[
\forall i \forall n \exists x [\phi_S(i,n)(x) = \phi_I(n,x)].
\]

\( p \) is as in the statement of the theorem.

We consider \( \phi_1 \) to be a function of two variables where computation rules are given by the following flowchart:

```
Begin
Read u and x

Is x > u no Set \( \phi_1(u,x) = \phi_p(x) \)

yes

Calculate \( \phi_1(u,0), \phi_1(u,1), \ldots, \phi_1(u,x-1) \)
to find the set defined by
\( D_{u,x} = \{ p_j | p_j \text{ is cancelled in these calculations} \} \)

Calculate \( \phi_1(x,x), \phi_1(x-1,x), \ldots, \phi_1(u+1,x) \)
to find \( \phi_S(i,x)(x), \phi_S(i,x+1)(x), \ldots, \phi_S(i,u+1)(x) \)

Set \( \phi_i(u,x) = 1 + \max \{ \phi_p(x) | p_j \not\in D_{u,x} \text{ and } \phi_p(x) \leq r(x, \phi_S(i,j)(x)) \text{ for all } p_j \text{ satisfying the above conditions cancelled.} \} \)
\( \phi_i(u,x) = \phi_p(x) \text{ if no } p_j \text{ satisfies the above conditions.} \)
```
Now in the event that $r$ is provably total, we readily see that for any fixed $u$,

$$\vdash \lambda x \phi_1(0,x) = \lambda x \phi_1(u,x) \text{ unless some } p_j \text{ with } 1 \leq j \leq u \text{ is cancelled while calculating } \lambda x \phi_1(0,x).$$

But for each $j$, $1 \leq j \leq u$,

$$\vdash \phi_j(1,0) \equiv p_j \equiv \lambda x \phi_1(0,x) = \phi_j(1,0).$$

Examining the construction therefore immediately yields for each $j$, $1 \leq j \leq u$,

$$\vdash p_j \text{ is not cancelled while calculating } \lambda x \phi_1(0,x).$$

Hence for each fixed $u$,

$$\vdash \phi_j(1,0) \equiv \phi_j(1,u).$$

Furthermore, for each fixed $j$,

$$\vdash \phi_j = \phi_j(1,0) \equiv \phi_j(1,j)$$

while $\vdash p_j$ is not cancelled while calculating $\lambda x \phi_1(0,x)$,

immediately yields by examining the construction,

(2) $\vdash (\forall x) [x > 1 \rightarrow r(x, \phi_j(1,j)(x)) < \phi_j(x)].$

The proof of (1) and (2) 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 $u$, $\phi_j(1,u) \equiv p_j$. (An easy double induction, similar to one given below, is used to show that if $p$ and $r$ are total, so is $\phi_1$. The bases of the induction are that $(\lambda u)\phi_1(u,0) = \phi_p$ and $(\forall x) [u > 1 \rightarrow \phi_1(u,x) = \phi_p(x)].$

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, $\phi_j(1,u)(x) \approx r^x - u \circ \phi$ 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 conditions we may interpret

$$\phi_p(x) \leq r(x, \phi_j(1,j)(x))$$

as meaning either that all of the above calculations terminate and that the inequality holds or that $\phi_p(x)$ is defined but one of $\phi_j(1,j)(x)$ or $\phi_j(1,u)$
r(x, φ_{S(1,j)}(x)) is undefined. Thus for any x, successful computation of
φ_p_j(x) leads to cancelling p_j unless both φ_{S(1,j)}(x) and r(x, φ_{S(1,j)}(x))
are defined and
φ_p_j(x) > r(x, φ_{S(1,j)}(x)).

Thus, the proof goes about as before, except that for fixed u, we see
that for all j, 1≤j≤u
L'λx φ_l(0,x) = φ_{S(1,0)} = φ_p_j
L'((λx)[φ_l(0,x) defined implies φ_p_j(x) defined])
Also L'φ_{S(1,0)} = φ_p_j so p_j is not cancelled
L'((∀x)[φ_l(0,x) defined implies φ_{S(1,j)}(x) defined and r(x, φ_{S(1,j)}(x))
defined, and φ_p_j(x) > r(x, φ_{S(l,j)}(x)) for all x ≥ j].]

From this, for each fixed u
L'(λx) φ_l(0,x) = λx φ_l(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 con­
structing 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 r-speed-up, but which
also has a fixed program p such that for any program p_0 with φ_p = φ_{p_0} = f and
φ_{p_0} ≤ φ_p a.e. there is no infinite sequence of programs p_1, p_2, p_3, p_4, ...
such that each program computes f and for all l≥0 (a) L'φ_{p_0} = φ_{p_l} and (b) φ_{p_l}(x) >
r(x, φ_{p_l}(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 equi­
valent 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 $S_o > S_1 > S_2 \ldots$ where $p_1(x) = S^{x-1}(x)$. If we
take any program $p$ for $f$ for which $\phi_{p_0} \leq \phi_{p}$, whenever $p_0$ computes $f$ and $p_0 \leq p$, we observe that for $\sigma$ as in Theorem 1, $\sigma(p_0)$ computes $f$ and is roughly at
least as fast as any program provably equivalent to $p_0$. Since the sequence $S_o, S_1, \ldots$ is co-Final in the good run times of $f$, $\phi_{\sigma(p_0)} > S_1$ 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 [H-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-I], 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 $p$, we can effectively find a program $p'$ such that $\phi_{p} \equiv \phi_{p'}$, the run times of $p$ and $p'$ are about the same, and $p'$ cannot 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 $\phi_{p} \equiv \phi_{p''}$, 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 $\phi_{p} \equiv \phi_{p''}$.)
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

1. \( \phi_1 \equiv p \) implies \( \forall(x, \phi_1(x)) \geq \phi_p(x) \) a.e.

and

2. if \( [\phi_1, \phi_p] \Rightarrow r(x, \phi_1(x)) \geq \phi_p(x) \) a.e.,

then \( \phi_p \not\equiv f \).

**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 \( \sigma(0), \sigma(1), \sigma(2), \ldots \) 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 \( \phi_0(0), \phi_1(1), \phi_2(2), \ldots \). 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 \( P_0, P_1, P_2, P_3, \ldots \) is a list of programs which are provably "near" optimal, where "near" here means the fixed recursive functions \( h \) such that the preceding \( t \) and \( f \) of Meyer's proof satisfies \( \Phi_f \leq h \cdot t \) a.e., where \( \Phi_f \) is the difficulty of a good way of computing \( f \). In reasonable measures, we can expand \( P_0, P_1, P_2, \ldots \) to a list \( P_{ij} \) \((0 \leq i < \omega; 0 \leq j, < \omega)\) such that \( P_{ij} = P_{ij} \) for all \( i \) and \( j \), \( \phi_{P_{ij}} = \phi_{P_i} \) 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 \( \sigma(0), \sigma(1), \sigma(2), \ldots \) such that every program \( \sigma(i) \) in the list satisfies \( \sigma(i) > h \) everywhere and \( \sigma(i) = \phi_{P_j} \) for some \( j \).

Furthermore, for every \( j \), if \( \phi_{P_i} > h \), a.e. then \( \phi_{P_j} = \phi_{\sigma(i)} \) for some \( i \), i.e., \( \phi_{\sigma(0)} \phi_{\sigma(1)} \phi_{\sigma(2)} \ldots \) is a list of the provably near optimal functions which are \( h \)-complex a.e., making Meyer's proof applicable.
Bibliography


ERRATA SHEET

Optimization among provably equivalent programs

<table>
<thead>
<tr>
<th>Page</th>
<th>line</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b 1</td>
<td>arguments</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>That is,</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>drop comma</td>
</tr>
<tr>
<td>4</td>
<td>b 7-8</td>
<td>from $\phi_p(x) = \phi_g(p)(x)$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to $\phi'_p(x) = \phi_g(p)(x)'$,</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>optimal</td>
</tr>
<tr>
<td>6</td>
<td>b 3</td>
<td>$p_0$ to $p$</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>where to whose</td>
</tr>
<tr>
<td>8</td>
<td>b 6</td>
<td>when to where</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>which do have</td>
</tr>
<tr>
<td>11</td>
<td>1.6</td>
<td>a.e.</td>
</tr>
</tbody>
</table>