"NATURAL" COMPLEXITY MEASURES AND TIME VERSUS MEMORY: SOME DEFINITIONAL PROPOSALS

Donald A. Alton
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242 U.S.A.

1. INTRODUCTION

Blum's notion of a complexity measure [7] models some of the principal properties of various dynamic computational resources. Unfortunately Blum's axioms are weak enough that some complexity measures have pathological properties which do not correspond to intuitive notions of how dynamic resources such as time or memory should behave for any programming language on any machine. Whereas Blum's axioms regard different programs as unrelated atomic entities, I approach efficiency considerations by viewing complicated programs (or instructions) as being built up structurally from simpler programs (instructions) via synthesizing functions. I consider limits (via the size of resource estimation functions) on the "overhead costs" associated with the coordination of the resource requirements of the simpler component programs (instructions) during execution of the synthesized programs (instructions). E.g., a synthesizing function might take a program, change an input variable to a control variable, and surround it by a for loop indexed by that variable. It is reasonable to expect to be able to bound the resource requirements of the loop by a "very subrecursive" function of information concerning the resource requirements of the various passes through the loop and the number of passes. Such an approach, which is related to previous literature cited in Section 4, allows one to avoid considering pathological submeasures and thus provides a step toward Hartmanis' goal of characterizing "natural" complexity measures [10].

This paper suggests axioms concerning the form of the relationship between a resource estimation function and a synthesizing function, independent of what the synthesizing function does (e.g., independent of whether the synthesizing function takes an instruction and uses it as the body of a for loop or uses it as the then clause of an if statement). Since the ease of different ways of synthesizing programs from simpler ones may vary radically from one reasonable programming language to another, such an approach to "natural" complexity measures seems appropriate. Such efforts lead me to model an underlying notion of computer memory and how it behaves, and I propose ways to axiomatically distinguish between "consumable" computational resources such as execution time and "reusable" resources such as computer memory (including the

possibility of dynamic storage allocation). The need for axiomatic distinctions between time and memory has been emphasized by Hartmanis [10].

The subrecursive speed-up theorem which will be announced in [1] and is proven in [2] depends upon specific axioms of the general form discussed here, for specific synthesizing functions corresponding to for loops, S-m-n, etc. Section 4 discusses it briefly.

2. FORM OF AXIOMS

Let $N = \{0,1,\cdots\}$. All relevant values will belong to N. Let $\{f_i\}$ index some subset of the collection of partial recursive functions, with "program" $i \in N$ requiring $d(i) \geq 1$ inputs. Let $\{\Phi_i\}$ be an associated complexity measure.

The synthesis method for for loops produces equivalent programs when applied to equivalent programs. Other synthesis methods might depend upon resource requirements (e.g., the output of the synthesized program equals the execution time of the component program) or upon parameters (e.g., given a component program and a finite table, the synthesized program gives the output prescribed by the table if the input is relevant to the table and behaves like the component program otherwise). A synthesizing function $s(\vec{p};\vec{k})$ has numbers ℓ_1, ℓ_2, and ℓ_3 associated with it and maps a sequence of $\ell_1 + \ell_3$ members of N, the first ℓ_1 representing input programs to the synthesizing function and the last ℓ_3 encoding parameters, to members of N. The first ℓ_2 input programs are distinguished as Φ-independent, the rest being Φ-dependent. Say $\vec{p} = (p_1,\cdots,p_{\ell_1})$ and $\vec{p}' = (p_1',\cdots,p_{\ell_1}')$ are similar if $d(p_i) = d(p_i')$ for all i. If $\vec{p}$ and $\vec{p}'$ are similar, $d(s(\vec{p};\vec{k}))$ must equal $d(s(\vec{p}';\vec{k}))$ for all parameters $\vec{k}$. A synthesizing function induces a Φ-operator if equivalent programs are synthesized whenever the Φ-independent programs are replaced by equivalent programs and the Φ-dependent programs are replaced by programs which are both equivalent and have identical computational resource requirements. If all the programs are Φ-independent, a Φ-operator is an operator. Although previous efforts to deal with the abstract form of axioms relating program structure to complexity have been limited to operators, numerous other Φ-operators are of interest. An obvious example is a synthesizing function r such that $f_{r(j)} = \Phi_j$. For acceptable universal languages, the existence of such a recursive r follows from [7 , Th. 2] by unbounded minimalization (unbounded search, i.e., while loops), and the Combining Lemma [11, p. 451] yields the existence of a recursive function R such that $\Phi_{r(i)}(x) \leq R(i,x,\Phi_i(x))$ for each i such that $d(i) = 1$. Many "natural" languages, both universal and subrecursive, allow straightforward introduction of "tracing

facilities." (E.g., initialize a new output variable to zero, precede each statement of the original program by an associated statement which updates the variable to reflect the resource requirements needed to execute that statement, and modify branches so that the new targets are the new statements associated with the old targets.) Thus, for "natural" languages, r and R can be expected to be "very subrecursive." A Φ-operator can be used to model the use of operating system clocks to monitor the amount of computational resource which a given program may use, for instance via a synthesizing function $s(i,j)$ having two component programs and having the property that if $d(i) = d(j) = 1$ then $f_{s(i,j)}(x)$ never halts if $f_i(x)$ never halts, has output 0 (indicating program j tried to exceed the resources allowed by the "clock") if $f_i(x)$ halts and $\Phi_j(x) \neq f_i(x)$, and has output $f_j(x)$ otherwise. In this example, the "clock" program i is Φ-independent but program j is Φ-dependent. Another example of a Φ-operator which is not an operator stems from the parallel computation property [13]: $f_{s(p_1,p_2)}(\vec{x})$ is 0 if $d(p_1) \neq d(p_2)$, $f_{p_1}(\vec{x})$ if $d(p_1) = d(p_2)$ and $\Phi_{p_1}(\vec{x}) \leq \Phi_{p_2}(\vec{x})$, and $f_{p_2}(\vec{x})$ otherwise.

A subcomputation description (sd) for $s(\vec{p};\vec{k})$ encodes a sequence $\langle i,\vec{y},r,e,sp,ap,cr,na\rangle$. Intended meanings are: Component program p_i, for $1 \leq i \leq \ell_1$, has been computed on the $d(p_i)$ inputs $\vec{y}$ as far as is possible using only $r \leq \Phi_{p_i}(\vec{y})$ units of computational resource Φ. The sd is incomplete if $r < \Phi_{p_i}(\vec{y})$. Each of e, sp, ap, and cr must be 0 or 1. If $e = 1$, as must be the case if $i \leq \ell_2$, the sd is entire, indicating that only $f_{p_i}(\vec{y})$, not $\Phi_{p_i}(\vec{y})$, is relevant and that if the subcomputation is incomplete then it is being completed. If $sp = 1$, the information used to perform the subcomputation is being preserved. (If the sd is incomplete and Φ is time, e.g., this will allow the subcomputation to be continued without wasting execution time re-simulating the first r steps.) If $ap = 1$, the answer is being preserved. If the sd is complete, this means that $f_{p_i}(\vec{y})$ is stored in memory if the sd is entire and that $f_{p_i}(\vec{y})$ and $\Phi_{p_i}(\vec{y})$ are both stored in memory if the sd is not entire. If the sd is incomplete, $ap = 1$ means that the fact that $r < \Phi_{p_i}(\vec{y})$ is stored in memory. If $sp = 1$, then $ap = 1$. If $cr = 1$, the sd is "currently relevant" to the amount of Φ used. The number of activations is given by na. The intended meanings of cr and na will be clarified later.

Think of a synthesizing function s as providing a "skeleton" for the bookkeeping efforts and coordination of subcomputations of component programs via a finite number of instructions (or states) ni which is independent of the programs $\vec{p}$ and parameters $\vec{k}$ which are being

synthesized, with start instruction 0 and halt instruction ni-1. It should be possible to "trace" which subcomputations are performed as a computation proceeds, with each new subcomputation corresponding to a new stage s. The values of a tracing function t should encode sequences $t(\vec{p};\vec{k};\vec{x};s) = \langle i,S_1,\cdots,S_n\rangle$, $n \geq 0$, where $i < ni$ indicates what instruction is about to be executed and the sd's S_ℓ encode the current status of (and some of the history of) all subcomputations which have been initiated. When $\vec{p}$, $\vec{k}$, and $\vec{x}$ are clear from context, $t(\vec{p};\vec{k};\vec{x};s)$ will be abbreviated as t_s. If $t_{s+1} = \langle \tilde{i},\tilde{S}_1,\cdots,\tilde{S}_m\rangle$, then $m \geq n$. If $1 \leq \ell \leq n$, the successor $\tilde{S}_\ell$ of S_ℓ must concern the same i and $\vec{y}$ as S_ℓ and must have a resource bound $\tilde{r} \geq r$. If S_ℓ has $e = 1$, $sp = 0$, or $ap = 0$, $\tilde{S}_\ell$ must have the same property. If $\tilde{r} > r$, $\tilde{S}_\ell$ is an active continuation. If this is the case, S_ℓ must have $sp = 1$. (If a subcomputation encoded in S_ℓ is not preserved, any efforts to "continue" it will have to be associated with an index other than ℓ, and we will be charged for all the resources required to begin the subcomputation anew; e.g., this will help model execution time sensibly.) If $\tilde{S}_\ell$ is an active continuation, $\tilde{S}_\ell$ must have $cr = 1$. (This will allow us to consider the computational resources used in the sd. E.g., if Φ is memory, it will allow us to charge for the memory required to obtain an answer (even if it is then deallocated), not just for the memory required to store that answer.) If S_ℓ is entire and incomplete then $\tilde{S}_\ell$ must be an active continuation. $\tilde{S}_\ell$ is active if $\ell \leq n$ and it is an active continuation, or if $\ell > n$. If $\ell > n$, $\tilde{S}_\ell$ must have $cr = 1$. If $i = ni-1$, $t_{s+1} = t_s$. Otherwise precisely one of $\tilde{S}_1,\cdots,\tilde{S}_m$ must be active. Thus $m \in \{n,n+1\}$. If $m = n+1$, $\tilde{S}_m$ must have $na = 1$. If $\ell \leq n$, $\tilde{S}_\ell$ and S_ℓ must have the same values of na unless $\tilde{S}_\ell$ is active and S_ℓ is not active, in which case the na value of $\tilde{S}_\ell$ must be one greater than the na value of S_ℓ. Thus na is, roughly speaking, the number of times attention has switched to pursuit of that subcomputation.

A resource correlation function c indicates that all of the subcomputations performed while computation of program $s(\vec{p};\vec{k})$ on input $\vec{x}$ is limited to r units of Φ are encoded in values of t for stages $s \leq c(\vec{p};\vec{k};\vec{x};r)$, abbreviated c_r. It must be nondecreasing in its last variable r. If $r_1 \geq r_2 \geq \Phi_{s(\vec{p};\vec{k})}(\vec{x})$, then $c_{r_1} = c_{r_2}$.

If the execution of a complicated program merely involves bookkeeping activities to coordinate various executions of component programs, these bookkeeping activities shouldn't increase the complexity too much. Thus, it is reasonable to require that a resource estimation function R exist with the properties that the "rate of growth" of R is reasonably

small and for all r,

(*) $R(\vec{p};\vec{k};\vec{x};\hat{t}_r) \geq r$, where $\hat{t}_r$ encodes t_s for all $s \leq c_r$.

In particular, for $r = \Phi_{s(\vec{p};\vec{k})}(\vec{x})$, R takes information concerning the complexities of all the relevant subcomputations and creates an upper bound on the complexity of the entire computation. The function R should be such that its value only depends upon those sd's in $\hat{t}_r$ which have $cr = 1$ and should have the monotonicity property that replacing the r value of an sd in $\hat{t}_r$ by a larger value does not decrease the value of R.

Several contrasts between time and memory will be given later. For the moment, note that for time we may require that $cr = 1$ for all sd's and that for memory an sd should have $cr = 1$ if and only if (iff) it has $ap = 1$.

Inequality (*) implies that the bookkeeping activities involved in coordinating the subcomputations performed by a synthesized program cannot themselves create non-halting computations; the synthesized program can fail to halt only because one of the subcomputations fails to halt or because the synthesized program requires infinitely many subcomputations. When we consider the special case of operators, this amounts to a restriction on the operators I consider. If I were attempting to axiomatize the complexity of operators, this would be an absurd restriction. Rather, I'm attempting to use a reasonably small class of operators (and Φ-operators) to characterize "natural" programming languages and complexity measures. As such, I feel this restriction on the source of non-halting behavior is helpful; a designer of a universal programming language should make sure that the primitives of the language do not hide the potential sources of non-halting computations from the programmer.

If the various notions discussed above do conform to their intended meanings, we may gauge how "natural" a programming language and complexity measure are in terms of "how subrecursive" the functions s, t, c, and R can be for various means of synthesizing programs with various kinds of structure, e.g., what the "rate of growth" is for a function R which relates the resource requirements of an entire for loop to the resource requirements of the various passes through the loop and the number of passes. However, we have not yet assumed enough to guarantee that s, t, c, and R correspond to our intuitive notions. E.g., a complexity measure which purports to behave like memory might involve very small amounts of computational resource simply because most sd's are arbitrarily given $cr = ap = 0$. Also, t might be "padded" to mention lots of spurious, irrelevant subcomputations. If this is the case, $\hat{t}_r$ may

be large and it may be possible for R, the function which purports to relate the complexity of a computation to the complexities of its subcomputations, to be "extremely subrecursive" and to have a very small "rate of growth" due to this pathological tracing function. If we assume we are dealing with a specific programming language with specific notions of time and memory and with a specific subroutine structure which is responsible for placing information in a tracing function, we don't need to worry about such issues. However, a more abstract setting is desirable. (E.g., consider the problem of modeling recursive procedures. It is plausible to expect that a synthesized program will always be syntactically distinct from its component programs, although it may be equivalent to one of its component programs. If this is the case, the restriction that sd's can only mention component programs does not let us mention recursive calls directly in the trace. A more abstract approach can reflect a recursive call to the synthesized program indirectly in terms of the subcomputations involving component programs which are executed during that recursive call.)

Three issues need to be dealt with: (1) How can we force the value of ap to reflect a reasonable notion of memory utilization? (2) How can we guarantee that t encodes sufficiently many subcomputations to represent the basic structure of the synthesizing function? (3) How can we guarantee that t doesn't encode too many subcomputations?

To deal with (1), we may formalize the notion that the "history" of a computation can only be retained in memory and implicitly in terms of what instruction is about to be executed. Such information should be adequate to determine what subcomputation will be performed, and such information plus the result of that subcomputation should in turn be adequate to determine which instruction will be executed after that subcomputation. Such an approach also allows us to force the nature of the subcomputations recorded in a tracing function to suffice to determine the answer, thus dealing with (2). The formalization of these ideas is contained in the Appendix. The basic idea is that the value produced by a synthesized program on a given input depends upon only finitely much information about programs in $\vec{p}$, possibly including information about run-times of non-entire subcomputations. This generalizes the effective continuity of partial recursive operators by allowing the run-times of non-entire subcomputations to influence the result of the computation (as in the parallel computation property, e.g.) and via the introduction of notions related to memory; effective continuity is given in terms of all subcomputations ever performed, corresponding to algorithms with adequate storage to retain the results of all subcomputations, whereas the

above notion includes conditions under which two computations can lead to the same answer even if they disagree on some subcomputations, if limitations on memory eventually lead to situations where those differences are no longer remembered. The above conditions imply that s induces a Φ-operator. Material related to (3) is also discussed in the Appendix.

Inequality (*) concerns an upper bound on how great the complexity of the synthesized program can be in terms of the complexities of the relevant subcomputations of component programs. Reasonable conditions which imply lower bounds can also be formulated. For fixed $\vec{k}$ and $\vec{x}$ and arbitrary $\vec{p}$, if an sd in t_s has $cr = 1$ and has a large r value or large values among $\vec{y}$, $\Phi_{s(\vec{p};\vec{k})}(\vec{x})$ can be expected to be large. (If a component program is executed for r units of resource in a subcomputation, then the synthesized program should require at least r units of resource. If a subcomputation involves input $\vec{y}$ and some member of $\vec{y}$ is much larger than all members of $\vec{x}$ and $\vec{k}$, we may expect that a great deal of resource is associated with producing $\vec{y}$, e.g., that it took a great deal of time to compute $\vec{y}$ and that it takes a great deal of memory to store $\vec{y}$.) If t_s contains lots of sd's having $cr = 1$, $\Phi_{s(\vec{p};\vec{k})}(\vec{x})$ can be expected to be large, even if each such sd has small values of r and $\vec{y}$. (If Φ is "consumable," e.g. time, this is clear. If Φ is "reusable," e.g. memory, all the resource requirements are required simultaneously.) Thus, it is reasonable to assume that for each $\vec{r}$, $\vec{k}$, and $\vec{x}$ there are only finitely many collections of sd's which for arbitrary $\vec{p}$ can appear among the sd's of t_s having $cr = 1$ and still allow $\Phi_{s(\vec{p};\vec{k})}(\vec{x}) < r$, and it is reasonable to assume that we can enumerate that finite collection of collections of sd's and know when we are done enumerating it (e.g., via canonical indices [15]). Such an approach to lower bounds is in the spirit of Symes [16] and Constable [8]. The condition can frequently be used to detect computations which cycle forever on r units of resource. (See Ausiello [6] and Havel [12].) E.g., suppose Φ is such that every sd in t which has $ap = 1$ also has $cr = 1$, e.g., Φ = memory. It is possible to simulate the computation until either the lower bounds imply more than r units of resource are required or two distinct stages have identical configurations (so that the computation cycles, by effective continuity) or the computation halts.

Even if the programming language is subrecursive, so that $\{f_i\}$ enumerates a small subset $\mathcal{I}_{\mathcal{L}}$ of the collection of recursive functions, it is frequently reasonable to assume that $\mathcal{I}_{\mathcal{L}}$ itself contains the functions s, T, c, and R associated with a synthesizing function, where

$T(\vec{p};\vec{k};\vec{x};s;r) = t_s$ whenever $s \leq c_r$. Such an assumption is in the spirit of the axioms required for the subrecursive speed-up theorem [1 , 2].

3. TIME VERSUS MEMORY

If Φ is a "reusable" resource, the crucial issue concerns which uses of the resource need to be performed simultaneously, so it is reasonable to assume the existence of a function R' such that for all r

$$(*) \qquad R(\vec{p};\vec{k};\vec{x};t_r) = \max\{R'(\vec{p};\vec{k};\vec{x};t_s): s \leq c_r\}$$

and such that R' satisfies the monotonicity property and values of R' only depend upon those sd's in t which have $cr = 1$. If Φ is "consumable" and an sd S_ℓ in t has $cr = 1$, its successor S_ℓ should also have $cr = 1$. If Φ is time, it is reasonable to assume that there is a function R' which satisfies the monotonicity property and is such that $R(\vec{p};\vec{k};\vec{x};t_r) = R'(\vec{p};\vec{k};\vec{x};t(\vec{p};\vec{k};\vec{x};c_r))$, since the trace t at a given stage s encodes adequate information concerning the history of the subcomputations; na records how many times attention has switched back to the subcomputation, and the values of sp and ap tell whether time has been devoted to deallocation of storage. By memory I mean the memory used to store the data relevant to execution of a program (e.g., the data stored in activation records associated with dynamic storage allocation), not the memory used to store a static representation of the program text. Time and memory should have the following contrasting properties:

(a) For time, every sd has $cr = 1$. For memory, an sd has $cr = 1$ iff it has $ap = 1$. This distinction emphasizes that time must be consumable whereas memory may be reusable. E.g., suppose index ℓ of a tracing function indicates a computation has performed an incomplete subcomputation, the subcomputation and answer are not preserved, and then at a later stage the subcomputation is attempted again. (This might happen when dealing with the parallel computation property and attempting to minimize the memory used, e.g.). Since the old subcomputation was not preserved, the new active sd must be associated with some index $\ell' \neq \ell$. For time, the sd's having indices ℓ and ℓ' will both still have $cr = 1$, hence will both be relevant to the function R', indicating we are still being "charged for" the time spent on the old subcomputation. For memory, the sd having index ℓ at the new stage will have $cr = 0$. Since R' only depends on sd's having $cr = 1$, the old memory once associated with index ℓ may be being reused currently.

(b) Although some synthesizing functions may fail to exploit the potential reusability of memory, preserving all subcomputations and

answers so that these particular uses of memory somewhat resemble time, it is the information preserved in memory, not the execution time, which affects how the computation will proceed and whether the computation will cycle, as the discussion of (1)-(3) indicated.

(c) The time spent on a subcomputation always affects the time of the computation but the memory required to just store the answer to a subcomputation doesn't always depend on the amount of memory required to compute the answer: For time, replacing an r value in any sd by a strictly larger r value gives a strictly larger value for R'. For memory: The analogous property holds for sd's having $sp = 1$. (In such cases the memory used in performing the subcomputation is still allocated.) If an sd has $sp = 0$ and $ap = 1$ and is incomplete or not entire, increasing the value of r gives at least as large a value for R' and increasing the value of r sufficiently much gives a strictly larger value for R'. (In such cases preserving the "answer" includes storing the value of r.) If an sd has $sp = 0$ and $ap = 1$ and is complete and entire, increasing the value of r leaves the value of R' unchanged. (In such cases only the output of the computation, not the amount of memory required to obtain that output, is stored.)

(d) Deallocating storage decreases the memory used but requires extra time: Replacing an sp or ap value of 1 by a value of 0 gives an R' value which is at least as large for time but at most as large for memory. Similar comments apply if an e value of 0 is replaced by 1 in a complete sd having $sp = 0$ and $ap = 1$, corresponding to deallocation of the memory used to store the r value of a non-entire computation.

(e) The number of times control switches back and forth to continuing a subcomputation which is being preserved affects execution time but doesn't affect memory usage: Increasing the value of na in an sd gives a larger value of R' for time and does not change the value of R' for memory. (E.g., if the parallel computation property is approached by preserving subcomputations in memory and switching back and forth between the two non-entire subcomputations as two coroutines, execution time may depend upon the number of coroutine calls but memory utilization will not.)

Note that (*) will be satisfied for time as well as for memory. A typical R' would add the sum of the r values of the sd's having $sp = 1$ to the sum of the base 2 logarithms of the numbers which need to be stored due to sd's having $ap = 1$ for memory or would add the sum of the na values to the sum of the r values for time. Note these two examples are very similar, involving additions in both cases. Thus, the

specific nature of R' (e.g., addition versus multiplication) is not as relevant to distinctions between time and space as the other considerations given above.

4. RELATIONSHIPS WITH THE LITERATURE

For the special case of operators and acceptable enumerations of the entire collection of partial recursive functions, recursive synthesizing functions correspond to effective operators, which are the same as the partial recursive operators, each of which is effectively continuous. The basic idea of viewing complicated programs as synthesized from simpler ones and considering limits on the "overhead costs" of coordinating the resource requirements of the component programs during eeecution of synthesized programs is present implicitly or explicitly in various works of various authors including Ausiello, Baker, Běcvář, Chytil, Constable, Lynch, and Young and is especially prominent in the concluding chapter of Symes [16] and is consistent with informal prose at the end of Hartmanis [10]. E.g., several authors propose an axiom dealing with composition such that $f_{s(p_1,p_2)}(x) = f_{p_1}(f_{p_2}(x))$ and $\Phi_{s(p_1,p_2)}(x) \leq \Phi_{p_2}(x) + \Phi_{p_1}(f_{p_2}(x))$ whenever $d(p_1) = d(p_2) = 1$. The present paper contrasts with the previous work in the following ways: Φ-operators such as that related to the parallel computation property are treated as well as operators. An underlying notion of memory is modeled. Conditions are given for gauging whether a proposed tracing function contains too few or too many subcomputations. An effort is made to distinguish between "consumable" resources such as time and "reusable" resources such as memory.

In addition, previous literature does not adapt major results from universal programming languages to subrecursive languages, as is done in the subrecursive speed-up theorem [1 , 2]. I have formulated specific axioms for several specific synthesizing functions which imply the existence of speed-upable functions [7], functions which do not possess optimal programs. Previously known proofs of speed-up only apply to universal programming languages (languages which compute all partial recursive functions), in part because they require that a universal function for the language must itself belong to the language. My result also applies to a broad class of "natural" subrecursive languages (languages in which every program always halts). The technique employed appears to provide a basis for an abstract theory of subrecursive complexity roughly along the lines sought by Constable and Borodin [9]. The subrecursive speed-up theorem is much more general [3] than the special cases implicit in [9, p. 561], which are "transferred" to specific subrecursive

languages from specific universal languages.

5. ACKNOWLEDGMENTS

I am very grateful to Prof. Juris Hartmanis of Cornell University. I had proposed means of supplementing Blum's axioms to deal with specific synthesizing functions which proved relevant to Lowther's thesis [14] (which stems from [4] and is generalized in [5]), and discussions with Prof. Hartmanis had a catalytic effect which very much helped in treating other similar axioms and in approaching other questions in such a setting. Prof. Don Epley of The University of Iowa has generously served as a sounding board in many helpful conversations. This research was supported by National Science Foundation grant MCS76-15648 and by a Faculty Developmental Assignment awarded by The University of Iowa.

6. REFERENCES

[1] D. Alton, "Natural" complexity measures and a subrecursive speed-up theorem, to appear, Proceedings of IFIP Congress '77, Toronto, Canada, August, 1977.

[2] D. Alton, "Natural" complexity measures, subrecursive languages, and speed-up, Computer Science report 76-05, University of Iowa, Iowa City, Iowa 52242 U.S.A., December, 1976.

[3] D. Alton, Are there lots of "natural" subrecursive programming languages and complexity measures?, in preparation.

[4] D. Alton, Nonexistence of program optimizers in several abstract settings, J. Comput. System Sci. 12(1976), 368-393.

[5] D. Alton and J. Lowther, Nonexistence of program optimizers: subrecursive languages and "natural" complexity measures, in preparation.

[6] G. Ausiello, Abstract computational complexity and cycling computations, J. Comput. System Sci. 5(1971), 118-128.

[7] M. Blum, A machine-independent theory of the complexity of recursive functions, J. Assoc. Comput. Mach. 14(1967), 322-336.

[8] R. Constable, Type two computational complexity, Proceedings of Fifth Annual ACM Symposium on Theory of Computing (1973), 108-121, available from Association for Computing Machinery, New York.

[9] R. Constable and A. Borodin, Subrecursive programming languages, part 1: efficiency and program structure, J. Assoc. Comput. Mach. 19(1972), 526-568.

[10] J. Hartmanis, On the problem of finding natural computational complexity measures, Proceedings of the Symposium and Summer School on Mathematical Foundations of Computer Science (Mathematical Institute of the Slovak Academy of Sciences, 1973), 95-103.

[11] J. Hartmanis and J. Hopcroft, An overview of the theory of computational complexity, J. Assoc. Comput. Mach. 18(1971), 444-475.

[12] I. Havel, Weak complexity measures, SIGACT News (January 1971), 21-30, available from Special Interest Group on Automata and Computability Theory, Association for Computing Machinery, New York.

[13] L. Landweber and E. Robertson, Recursive properties of abstract complexity classes, J. Assoc. Comput. Mach. 19(1972), 296-308.

[14] J. Lowther, The non-existence of optimizers and subrecursive languages, Computer Science report 75-07, University of Iowa, Iowa City, Iowa 52242 U.S.A., December, 1975.

[15] H. Rogers, Jr., Theory of recursive functions and effective computability, McGraw-Hill, New York, 1967.

[16] D. Symes, The extension of machine-independent computational complexity theory to oracle machine computation and to the computation of finite functions, Department of Applied Analysis and Computer Science report CSRR 2057, University of Waterloo, Waterloo, Canada, October, 1971.

7. APPENDIX

Abbreviate $t'_s = t(\vec{p}';\vec{k};\vec{x};s)$. To formalize the above notions, say $t = t_s$ and $t' = t'_s$, have identical configurations if $\vec{p}$ and $\vec{p}'$ are similar, t and t' encode the same instruction, and the sd's of t and t' which have $ap = 1$ can be put into one-to-one correspondence so that corresponding sd's have the same $i,\vec{y}$, and e values, corresponding sd's are either both complete or both incomplete, corresponding sd's which are not entire have the same r values (so that $\Phi_{p_i}(\vec{y}) \le r$ iff $\Phi_{p_i'}(\vec{y}) \le r$), and corresponding sd's which are complete are such that $f_{p_i}(\vec{y}) = f_{p_i'}(\vec{y})$.

For all $\vec{p}$, $\vec{k}$, and $\vec{x}$, t_0 must encode the start instruction and no sd's, and $f_{s(\vec{p};\vec{k})}(\vec{x})$ must be defined iff t_s encodes the halt instruction for some s, in which case every entire sd in t_s must be complete. Suppose $t = t_s$ and $t' = t'_{s'}$ have identical configurations. If they encode the halt instruction, $f_{s(\vec{p};\vec{k})}(\vec{x})$ must equal $f_{s(\vec{p}';\vec{k})}(\vec{x})$. Suppose they do not encode the halt instruction. Then it is reasonable to require that the active sd's of $\tilde{t} = t_{s+1}$ and $\tilde{t}' = t'_{s'+1}$ must have the same i, $\vec{y}$, and e values, and if they are not entire they must also have the same r value. If they are entire and $f_{p_i}(\vec{y})$ and $f_{p_i'}(\vec{y})$ are defined and equal, then require that $t_{\hat{s}}$ and $t'_{\hat{s}'}$ must have identical configurations, where $\hat{s} \ge s+1$ and $\hat{s}' \ge s'+1$ are the first stages at which the active sd's of $t_{\hat{s}}$ and $t'_{\hat{s}'}$ are complete. (Those active sd's concern i and $\vec{y}$.) Suppose the active sd's of $\tilde{t}$ and $\tilde{t}'$ are not entire. Suppose also that either both or neither of $\Phi_{p_i}(y) \le r$ and $\Phi_{p_i'}(y) \le r$ hold, and that if both hold then $f_{p_i}(\vec{y}) = f_{p_i'}(\vec{y})$. Then $\tilde{t}$ and $\tilde{t}'$ must have identical configurations. This completes conditions for (1) and (2).

Let $\vec{p}$ and $\vec{p}'$ be similar. Let S be an sd which concerns i, $\vec{y}$, and r. If S is entire, say $\vec{p}$ and $\vec{p}'$ are compatible with respect to (wrt) S if $f_{p_i'}(\vec{y}) = f_{p_i}(\vec{y})$. If S is not entire, say they are compatible wrt S if the conditions $\Phi_{p_i}(\vec{y}) \leq r$ and $\Phi_{p_i'}(\vec{y}) \leq r$ either both hold or both fail and if in addition $f_{p_i}(\vec{y}) = f_{p_i'}(\vec{y})$ in case both conditions hold. If t_s and t_s', have identical configurations and $\vec{p}$ and $\vec{p}'$ are compatible wrt the active sd's of $t_{\hat{s}}$ for every $\hat{s}$ such that $s < \hat{s} \leq c(\vec{p};\vec{k};\vec{x};\Phi_{s(\vec{p};\vec{k})}(\vec{x}))$, the above conditions imply $f_{s(\vec{p}';\vec{k})}(\vec{x}) = f_{s(\vec{p};\vec{k})}(\vec{x})$. This is the notion of effective continuity referred to in the body of this paper.

Several approaches to (3) come to mind. If t_s does not encode all the relevant subcomputations (i.e., does not encode the halt instruction), then one approach would be to require that $\vec{p}'$, $\vec{p}''$, s', and s'' exist such that $t = t_s$ and $t' = t_{s'} = t(\vec{p}';\vec{k};\vec{x};s')$ and $t'' = t_{s''} = t(\vec{p}'';\vec{k};\vec{x};s'')$ all have identical configurations but such that $\vec{p}'$ and $\vec{p}''$ give different results to the (similar, by effective continuity) active computations performed at stages s'+1 and s''+1 and that this results in either a different instruction at the end of those stages or in initiation of different subcomputations at stages s'+2 and s''+2. However, this is not a reasonable general approach to (3). E.g., consider a synthesizing function s such that $f_{s(p_1,p_2)}(\vec{x})$ equals 0 if $d(p_1) \neq d(p_2)$ or $f_{p_1}(\vec{x})$ and $f_{p_2}(\vec{x})$ both halt with different outputs and such that $f_{s(p_1,p_2)}(\vec{x})$ never halts otherwise. A reasonable tracing function for s might be such that after computing $f_{p_1}(\vec{x})$, the next subcomputation would always be $f_{p_2}(\vec{x})$, regardless of the value of $f_{p_1}(\vec{x})$. An alternative approach to (3) would require that t, t', and t'' all have identical configurations but that $f_{s(\vec{p}';\vec{k})}(\vec{x}) \neq f_{s(\vec{p}'';\vec{k})}(\vec{x})$. However, this alternative only insists that some of the subcomputations after stage s are necessary, not that all of them are necessary. Another possibility is to insist that the active subcomputation at stage s+1 should be such that subcomputations which agree up to that point but which disagree on that subcomputation are guaranteed to eventually produce different outputs, even though further subcomputations may be required to determine what those outputs are. This approach is too restrictive, as illustrated by the synthesizing function s discussed earlier in this paragraph.

Instead of the above approaches to (3), the following approach appears to be reasonable: Suppose that t_s does not encode the halt instruction and that the active sd of t_{s+1} concerns i and $\vec{y}$. It is reasonable to require that there exist $\vec{p}'$, $\vec{p}''$, s', and s'' such that t_s, $t_{s'}$, and $t_{s''}$ all have identical configurations, such that

for every $\hat{s}$ such that $s'+2 \leq \hat{s} \leq c(\vec{p}';\vec{k};\vec{x};\Phi_{s(\vec{p}';\vec{k})}(\vec{x}))$ it is the case that $\vec{p}''$ and $\vec{p}'$ are compatible wrt the active sd of $t'_{\hat{s}}$ unless it concerns i and $\vec{y}$, and such that $f_{s(\vec{p}';\vec{k})}(\vec{x}) \neq f_{s(\vec{p}'';\vec{k})}(\vec{x})$. (If in addition $f_{p_i'}(\vec{y})$ equalled $f_{p_i''}(\vec{y})$ and $\Phi_{p_i'}(\vec{y})$ equalled $\Phi_{p_i''}(\vec{y})$, effective continuity would imply $f_{s(p';\vec{k})}(\vec{x}) = f_{s(p'';\vec{k})}(\vec{x})$. Thus, the subcomputation performed by program $s(\vec{p}';\vec{k})$ at stage $s'+1$, corresponding to the active sd of t_{s+1}, really did affect the outcome.) If the active sd of t_{s+1} is not entire, we may also require $f_{p_i'}(\vec{y}) = f_{p_i''}(\vec{y})$ (so that the only relevant discrepancy between $\vec{p}'$ and $\vec{p}''$ concerns the fact that $\Phi_{p_i'}(\vec{y})$ and $\Phi_{p_i''}(\vec{y})$ are not equal).

If such a condition is satisfied, clearly the tracing function only encodes essential information. Of course some moderately reasonable synthesizing functions do sometimes perform unnecessary subcomputations. Thus, sometimes in concrete situations where we are certain that the tracing function reasonably models a specific notion of time or memory for a specific programming language, we may be willing to live with tracing functions which fail to have this property. However, in more abstract situations use of the above condition appears to be a reasonable way of guaranteeing that a tracing function doesn't encode too many subcomputations.