Depth-Bounded Computation

H. R. Strong

IBM Watson Research Center, Yorktown Heights, New York 10598

Received February 19, 1969

ABSTRACT

A programming language designed for studies of parallelism and based on Wagner's uniformly reflexive structures is introduced. The measure of depth of computation in the language is studied. The partial recursive functions are shown to be computable in uniformly bounded depth. A comparison of the measure with other proposed measures of computational complexity leads to the suggestion of a list of properties to be checked in classifying such measures.

1. INTRODUCTION

The task of the field of computational complexity at this time is to classify computations in a way that maximizes the information available for further study and application to the problems of program and machine construction. The desired classification obviously lies somewhere between treating each separate computation as a unique individual unlike any other and lumping together all those computations which take, for example, the same amount of time on some fixed machine. A number of parameters have been introduced for measuring computational complexity. It is hoped that studies of the relationships among these measures will establish an ideal combination for the purposes of this classification.

Blum [1] has specified a number of desirable properties for any measure of computational complexity. In particular, he requires that the measure be finite if and only if the computation converges and that it be effectively computable. However, one of the important properties of the measure of depth of computation introduced in this paper is that it can be defined and finite even for a computation which does not converge. Its restriction to computations which do converge is effectively computable, but it is not known whether the measure itself is. It is a special purpose measure designed to operate on computations expressed in a programming language which is adapted to an environment of parallelism. The programming language is based on an axiomatic theory of the partial recursive functions introduced by E. G. Wagner in his paper, "Uniformly Reflexive Structures: Towards an Abstract Theory of Computability," [2], and later studied by the author of this paper [3].
The measure is not a measure of parallelism as such but of a quantity which bounds (from above) the number of discrete computations which can take place in parallel during the course of a computation in the language (which will be called the URS-language). The depth of computation may be thought of as a kind of storage requirement which more closely resembles the “state-symbol product” for Turing machines and various other “length of program” measures than it does the “step counting” measures of time and space [1]. The URS-language is not proposed as a practical programming language, but rather as a vehicle for studies of parallelism. However, some modification of it might be implemented as a lower-level programming language into which such functional programming languages as LISP could be compiled. The language involves operators similar to those of combinatory logic.\footnote{\begin{footnotesize}Except that they operate only on normal-form expressions, the operators $A$, $B$, and $I$ strongly resemble operators usually called $S$, $K$, and $I$, respectively, in combinatory logic. The operator $E$ is not available in combinatory logic. See Ref. 4.\end{footnotesize}}

Computation in the URS-language involves a sequence of discrete changes made to an expression which embodies both “program” and “data.” During most computations no distinction can be made between the two. The parallelism referred to here arises from the fact that the discrete changes can often be applied independently to different parts of the expression in such a way that the order of their application has no effect on the result of a terminating computation. The parallelism in question bears the same relationship to that discussed by Karp and Miller in “Parallel Program Schemata” [5] as the relationship between a functional programming language and a “flow chart” programming language. The notion of “bounded schema” in “Parallel Program Schemata” corresponds here to the notion of an index (“program”) which achieves a uniform depth bound for the computations of a function. However, many aspects of parallelism in functional programming languages are not exploited by the URS model. This model is deterministic so that the questions of “determinacy” investigated by Karp and Miller do not apply here. Moreover, while “Parallel Program Schemata” concentrates on decision problems for schemata, this paper emphasizes the properties of the depth measure as a measure of computations of partial recursive functions.

\section*{2. Preliminary Definitions}

The basic operators of the URS-language are $A$ and $E$ ($A \neq E$) satisfying the following application rules on normal form expressions of the language (to be defined below):

\begin{enumerate}
  \item $Axyz = xa(yz)$,
  \item $Ewxyz = x$, if $w = x$; $y$, otherwise.
\end{enumerate}
Note that the omitted parentheses are understood to be supplied from the left; thus, the expression $Axyz$ abbreviates $((Ax)y)z$. An additional operator which is necessary and sufficient for computing the partial recursive functions on the natural numbers is $S$:

(3) $Sx =$ the normal-form expression for its successor if $x$ is the normal-form expression for a natural number; $A$, otherwise.\

Two further operators which can be defined in terms of $A$ and $E$ but are added as a convenience are $B$ and $I$:

(4) $Bxy = x$,

(5) $Ix = x$.

In order to understand the way the operators work, it may be a helpful exercise for the reader to verify that $I = A(EEAE)(EEEE)$, and $B = A(EE)I$, satisfy (4) and (5).

Normal-form expressions are defined inductively as follows:

(a) $A$, $E$, $S$, $B$, and $I$ are normal-form expressions;

(b) there is a unique normal-form expression $\bar{n}$ for each natural number $n$ ($A$, $E$, $S$, $B$, and $I$ are not normal-form expressions for natural numbers);

(c) if $x$, $y$, and $z$ are normal-form expressions, then so are $Ax$, $Axy$, $Bx$, $Ex$, $Exy$, and $Exyz$.

A computation in the URS-language is a sequence of trees, where tree is defined inductively as follows:

(a) if $x$ is a normal-form expression, $[x]$ is a tree;

(b) if $t$ and $u$ are trees, $[tu]$ is a tree.

The two-dimensional and linearized forms of trees are identified for convenience; thus, a tree is a binary tree in the graph theory sense with normal-form expressions at the leaves. A sequence of trees is a computation if each succeeding tree is obtained from its predecessor by one of the following replacement rules ($w$, $x$, $y$, and $z$ stand for arbitrary normal-form expressions):

(1) if $P$ is the operator $A$, $E$, or $B$, $[[P][x]] \rightarrow [Px]$;

(2) if $P$ is the operator $A$ or $E$, $[[Px][y]] \rightarrow [Pxy]$;

(3) [[Ax$y$][z]] $\rightarrow [[[x][z]][[y][z]]]$;

(4) $[[E$xy$][z]] \rightarrow [Exyz]$;

(5) $[[E$wxy$][z]] \rightarrow [x]$, if $w = z$; $[y]$, otherwise;

(6) $[[S][n]] \rightarrow [n + 1]$;

(7) $[[S][x]] \rightarrow [A]$, if $x$ is not the normal-form expression for a natural number;

\footnote{See Ref. 3, where the necessity and sufficiency are demonstrated.}
The most important replacements are illustrated in two-dimensional form in Fig. 1.

A computation terminates, if it is a finite sequence of trees the last of which has a single leaf. A sequence of trees is a reduced computation if each succeeding tree is obtained from its predecessor by application of all simultaneously applicable replacement rules. An example of a terminating computation and its corresponding terminating reduced computation is given in Fig. 2. A computation or reduced computation is complete if it cannot be extended.

**Theorem 1.** For each tree $t$, there is a unique complete reduced computation beginning with $t$. Moreover, if a terminating computation begins with $t$, ends with $u$, and contains exactly $n$ trees, then each computation beginning with $t$ can be extended to a terminating computation ending with $u$ and containing exactly $n$ trees.
The depth of a tree is the length of its longest branch; thus, the depth of $[[A][B]]$ is 1, and the depth of $[A]$ is 0. Theorem 1 is proved by induction on the depth of $t$. The proof is left to the reader.

The depth of a sequence of trees is the maximum of the depths of its elements (possibly infinite). A proof of Theorem 1 can easily be extended to a proof of the following theorem.

**Theorem 2.** The depth of a complete computation is uniquely determined by its beginning tree and is equal to the depth of the corresponding complete reduced computation.

Figure 3 gives an example of a complete nonterminating reduced computation of depth 2. Every complete computation of depth 1 terminates.

A complete nonterminating reduced computation of depth 2.

![Figure 3](image)

A tree $t$ is an index of a partial recursive function $f$ of $n$ variables if, for each $a_1, ..., a_n$, there is a complete computation (called a computation of $f(a_1, ..., a_n)$ by $t$) beginning with $\cdot \cdot \cdot [[f[a_1]][a_2]] \cdot \cdot \cdot [a_n]]$, which terminates with $[f(a_1, ..., a_n)]$, when $f(a_1, ..., a_n)$ is defined and fails to terminate otherwise. Here $f$ is assumed to be a function of natural numbers; other kinds of functions are handled similarly. It is easy to show that, if $f$ has an index, it has one of the form $[x]$, where $x$ is a normal-form expression. In Fig. 2, the tree $[[A(BS)][S]]$ is an index for the function $f(a) = a + 2$; so is $[A(BS)] S$. The tree $t$ converges to the tree $u$, if there is a computation beginning with $t$ and terminating with $u$. The tree $t$ produces the tree $u$ if there is a computation beginning with $t$ and containing $u$.

The following theorem is the major result of this paper.

**Theorem 3.** Each partial recursive function $f$ has an index $t$, such that the depths of all computations of $f$ by $t$ are uniformly bounded from above by a finite number $d(f)$.

**Corollary 1.** For each positive integer $n$, there is a positive integer $D_n$, such that every partial recursive function $f$ of $n$ variables has an index $t$, with all computations of $f$ by $t$ confined to depth $D_n$.

The proof of Theorem 3 depends on the following lemmas.

**Lemma 1.** Let $x$ be any normal-form expression except $A$, $B$, or $I$. Let $t(x)$ be a tree of depth $d$, in which $x$ may occur only as a leaf (not imbedded in some other normal-form expressions).
expression). There is a tree $\alpha_{x,t}(x)$ and a normal-form expression $\lambda x t(x)$ satisfying the following properties:

1. the depth of $\alpha_{x,t}(x)$ is at most $2d + 1$;
2. the leaves of $\alpha_{x,t}(x)$ are $A$, $B$, $I$, or a leaf of $t(x)$ other than $x$, so that $x$ does not occur in $\alpha_{x,t}(x)$; i.e., $x$ plays the role of a dummy variable;
3. $\alpha_{x,t}(x)$ converges to $[\lambda x t(x)]$ within depth $2d + 1$;
4. if $y$ is any normal-form expression, $[[\lambda x t(x)][y]]$ produces $t(y)$ within depth $d + 1$, where $t(y)$ is the result of replacing each occurrence of $x$ in $t(x)$ by $y$.

**Proof of Lemma 1.** The method of proof is induction on $d$.

**Case 1.** $d = 0$. Let $\alpha_{x,t}(x) = [I]$, if $t(x) = [x]$; $[[B][w]]$, if $t(x) = [w]$ and $w \not= x$. Then $\lambda x t(x) = I$ or $Bw$, and $\alpha_{x,t}(x)$ converges to $\lambda x t(x)$ within depth 0 or 1, respectively. If $y$ is any normal-form expression, $[[I][y]]$ converges to $[y]$ and $[[Bw][y]]$ converges to $[w]$ within depth 1.

**Case 2.** Assume Lemma 1 is true for depths less than $d$, $d > 0$. Since $d > 0$, $t(x) = [u(x) \nu(x)]$, where $u(x)$ and $\nu(x)$ are trees with depth less than $d$ and with $x$ occurring only as a leaf. Let $\alpha_{x,t}(x) = [[[A] \alpha_{x,u}(x)] \alpha_{x,\nu}(x)]$. Then $\lambda x t(x) = A\lambda x u(x) \lambda x \nu(x)$, and the depth of computation from $\alpha_{x,t}(x)$ to $[\lambda x t(x)]$ is the maximum of the depth from $\alpha_{x,u}(x)$ to $[\lambda x u(x)] + 2$ and the depth from $\alpha_{x,\nu}(x)$ to $[\lambda x \nu(x)] + 1$, which is less than or equal to $2(d - 1) + 1 + 2 = 2d + 1$. The depth of $\alpha_{x,t}(x)$ is thus less than or equal to $2d + 1$. If $y$ is any normal-form expression, $[[\lambda x t(x)][y]]$ produces $[[[\lambda x u(x)][y]][\lambda x \nu(x)][y]]$ (within depth 2), which produces $[u(y) \nu(y)] = t(y)$, within depth $(d - 1) + 1 + 1 = d + 1$.

This concludes the proof of Lemma 1. An illustration is given in Fig. 4.

The following corollary to Lemma 1 is proved by induction. The proof is left to the reader.

**Corollary 2.** Let $t(x_1, ..., x_n)$ be a tree with depth $d$ with $n (>1)$ normal-form expressions $x_1, ..., x_n$ (not equal to $A$, $B$, or $I$), which may occur in $t(x_1, ..., x_n)$ only as leaves. There is a tree $\alpha_{x_1, ..., x_n,t}(x_1, ..., x_n)$ and a normal-form expression $\lambda x_1 ... x_n t(x_1, ..., x_n)$ satisfying the following properties:

1. the depth of $\alpha_{x_1, ..., x_n,t}(x_1, ..., x_n)$ is at most $2^n(d + 1) - 1$;
2. the leaves of $\alpha_{x_1, ..., x_n,t}(x_1, ..., x_n)$ are $A$, $B$, $I$, or a leaf of $t(x_1, ..., x_n)$ other than $x_1, ..., x_n$;
3. $\alpha_{x_1, ..., x_n,t}(x_1, ..., x_n)$ converges to $[[\lambda x_1 ... x_n t(x_1, ..., x_n)]$, within depth $2^n(d + 1) - 1$.
(4) if \(y_1, \ldots, y_m\) are any normal-form expressions \(m \leq n\), then
\[\ldots([\lambda x_1 \ldots \lambda x_n t(x_1, \ldots, x_n)][y_1] \ldots [y_m]]\]
produces \(\alpha_{x_n, \ldots, x_m+1, \ldots, x_n} t(y_1, \ldots, y_m, x_{m+1}, \ldots, x_n)\) within depth \(2^{n-1}(d + 1) + m - 1\).

Note that \(\alpha_{x_n} t(x)\) and \(\alpha_{x} t(x)\) are independent of \(x\). Thus, the explicit recognition of symbols such as \(x\) or \(y\) as variables for normal-form expressions allowed to appear in trees and normal-form expressions will not be confusing: below, the result of replacing normal-form expressions by variables in a tree or normal-form expression will be called a tree or a normal-form expression, respectively. This is just a way of avoiding expressions like "let \(x_1\), \(x_2\), and \(x_3\) be distinct normal-form expressions not equal to \(A\), \(B\), or \(I\)."

The next lemma establishes the existence of an index for a fixed-point finding function.\(^3\)

\(^3\) See the uniform recursion, or fixed-point theorem, of Ref. 3. In the terminology of this paper a fixed point for a function is an index which reproduces itself during computation as an argument of one of its arguments (treated as an index) making general recursion possible.
Lemma 2. There is a normal-form expression $p$, such that, for all normal-form expressions $x$ and $y$, the tree $[[p][x]]$ converges to a tree called $[\phi x]$ within depth 67, and $[[\phi x][y]]$ produces $[[x][\phi x]][y]$ within depth 18.

Proof of Lemma 2. Let

$$t(x_1, x_2, x_3) = [[[x_1][x_2][x_3]][x_3]].$$

Let $u = \alpha_{x_3 \cdot x_2 \cdot x_1} t(x_1, x_2, x_3)$. Let $p = \alpha_x [[u[x]][u[x]]]$. Then

$$\phi x = \lambda x_3 f(x, \lambda x_2 x_3 t(x, x_2, x_3), x_3);$$

and $p$ and $\phi x$ have the desired properties, as is illustrated in Fig. 5. An arrow between two trees indicates that the left-hand tree produces the right-hand one within a depth bounded by the number written over the arrow. The numbers are obtained by application of Lemma 1 and its corollary.

Fig. 5. Illustration of Lemma 2: $p$ is the index of a fixed point finding function.

The particular numbers are not important, so the details of their calculation are omitted. It is important that $p$ is an index of a total function of one variable such that the depth of its computation is uniformly bounded. If $x$, as an index of a function of two variables, has uniformly bounded depth of computation, then so does $\phi x$, as an index of a function of one variable.

A function $f$ is computable in uniformly bounded depth if it has an index with uniformly bounded depth of computation (in all the arguments of $f$). In the proof of Theorem 3, “function” means “function from natural numbers to natural numbers.”

Proof of Theorem 3. The partial recursive functions are characterized (by Davis [6]) as the smallest class of functions containing the successor function, $S(a) = a + 1$, the constant zero function, $N(a) = 0$, the projection functions, $U_i^n(a_1, \ldots, a_n) = a_i$ ($1 \leq i \leq n$), and closed under the operations of composition, primitive recursion (of
total functions), and minimalization (of total functions). It will be shown that the class of functions computable in uniformly bounded depth satisfies these properties (it should be clear that every function with an index is partial recursive).

An index for the successor function is \([S]\); its depth is uniformly bounded by 1. An index for the constant zero function is \([B0]\); its depth is also uniformly bounded by 1. An index for \(U_i^n\) is \([x_1 \cdots x_n[x_i]]\); its depth is uniformly bounded by \(2^{n-1} + n - 1\).

If \(f\) is computable in uniformly bounded depth, let \(d(f)\) be the minimum such bound. Thus, \(d(S) = 1\), \(d(N) = 1\), and \(d(U_i^n)\) is less than or equal to \(2^{n-1} + n - 1\).

**Composition**

Suppose that the function \(f\) of \(n\) variables is obtained from a function \(g\) of \(m\) variables and functions \(h_1, \ldots, h_m\) of \(n\) variables by composition,

\[
f(a_1, \ldots, a_n) = g(h_1(a_1, \ldots, a_n), \ldots, h_m(a_1, \ldots, a_n))\]

and depth bounds \(d(g), d(h_1), \ldots, d(h_m)\) are achieved respectively by indices \(u, v_1, \ldots, v_m\). The tree \(t(x_1, \ldots, x_n)\) displayed in Fig. 6 has depth

\[
d_t \leq \max(d(g) + m, d(h_1) + m, d(h_2) + m - 1, \ldots, d(h_m) + 1),
\]

and for all \(a_1, \ldots, a_n\), computations beginning with \(t(a_1, \ldots, a_n)\) have depth bounded by \(d_t\). If \(f(a_1, \ldots, a_n)\) is defined, \(t(a_1, \ldots, a_n)\) converges to \([f(a_1, \ldots, a_n)]\). Thus, \([\lambda x_1 \cdots x_n t(x_1, \ldots, x_n)]\) is an index for \(f\) and \(d(f) \leq 2^{n-1}(d_t + 1) + n - 1\).

![Diagram](image-url)

**Fig. 6.** Composition of a function with index \(u\) with functions with indices \(v_1, \ldots, v_m\).
Suppose that the function $f$ of $n + 1$ variables is obtained from a total function $g$ of $n$ variables and a total function $h$ of $n + 2$ variables by primitive recursion,

\[
\begin{align*}
  f(0, a_1, \ldots, a_n) &= g(a_1, \ldots, a_n), \\
  f(a + 1, a_1, \ldots, a_n) &= h(a, f(a, a_1, \ldots, a_n), a_1, \ldots, a_n);
\end{align*}
\]

and depth bounds $d(g)$ and $d(h)$ are achieved, respectively, by indices $u$ and $v$. In the tree $t(w, x_1, \ldots, x_n, y, z, x)$ displayed in Fig. 7, $w$ will play the role of a fixed point for the desired function, $z$ will play the role of a counter, and $y$ will play the role of an accumulator. The tree has depth $d_t \leq d(h) + 5$. The tree $t'(x, x_1, \ldots, x_n)$ displayed in Fig. 7 has depth $d_t' \leq d(g) + 3$. For each $a, a_1, \ldots, a_n$, $t'(a, a_1, \ldots, a_n)$ converges to \([f(a, a_1, \ldots, a_n)]\), as illustrated in Fig. 8. Thus \([\lambda x_1 \cdots x_n t'(x, x_1, \ldots, x_n)]\) is an index for $f$ and $d(f) \leq \max(2n^{a_0}(d(h) + 6) + n + 5, 2^n(d(g) + 4) + n)$, as also illustrated in Fig. 8.
In Figs. 7 and 8, $\omega$ stands for $\lambda w x_1 \cdots x_n y z t(w, x_1, \ldots, x_n, y, z, x)$, and $\omega'$ stands for $\lambda x x_1 \cdots x_n t'(x, x_1, \ldots, x_n)$.

**MINIMALIZATION**

Suppose the function $f$ of $n$ variables is obtained from a total function $g$ of $n + 1$ variables by minimalization,

$$f(a_1, \ldots, a_n) = \text{the least } a \text{ such that } g(a, a_1, \ldots, a_n) = 0,$$

if such an $a$ exists;

and depth bound $d(g)$ is achieved by index $u$. The trees $t(w, x_1, \ldots, x_n, z)$ and $t'(x_1, \ldots, x_n)$ displayed in Fig. 9 play roles similar to those of Fig. 7. They have depths $d_t \leq \max(5, n + 3, d(g) + 2)$, and $d_t' = n + 1$. If $f(a_1, \ldots, a_n)$ is defined, $t'(a_1, \ldots, a_n)$ converges to $[f(a_1, \ldots, a_n)]$, as illustrated in Fig. 10; and, in any case, $[\lambda x_1 \cdots x_n t'(x_1, \ldots, x_n)]$ is an index for $f$ and $d(f) \leq \max(2^{n+1}(d(g) + 3) + n + 1, n + 21)$.

In Figs. 9 and 10, $\omega$ stands for $\lambda w x_1 \cdots x_n t(w, x_1, \ldots, x_n, z)$, and $\omega'$ stands for $\lambda x x_1 \cdots x_n t'(x_1, \ldots, x_n)$.

Figure 10 completes the proof of Theorem 3. Corollary 1 is easily proved by considering a universal function of $n + 1$ variables.
4. Gödel Numberings and Complexity Measures

A Gödelization, or Gödel numbering, of the partial recursive functions is a map from the natural numbers to the partial recursive functions (of one variable) satisfying certain “effectiveness” properties. Rogers [7] has shown that any two Gödelizations are recursively isomorphic. Thus, any Gödelization can be “effectively reached” from any other. There is a Gödelization $\pi$ such that a derivation of $\pi(n)$, listing each composition,

\[ t'(x_1, \ldots, x_n) = \]

\[ t(\omega, x_1, \ldots, x_n, z) = \]

Fig. 9. Details of setting up minimalization of a total function with index $u$ by means of the fixed point $\phi_0$.

Fig. 10. Minimalization.

\[ \omega' \]

\[ 2^{n-1}(d'_I + 1) + n - 1 \]

\[ t'(\delta_1, \ldots, \delta_n) = \]

\[ t(\phi_0, \delta_1, \ldots, \delta_n, \delta) \]

\[ \max(n+2, d(g)+2) \]

\[ g(0, q_1, \ldots, q_n) = 0 \]

\[ g(0, q_1, \ldots, q_n) \neq 0 \]

\[ t(\phi_0, \delta_1, \ldots, \delta_n, \delta) \rightarrow \text{etc} \]

\[ 4 \text{ See Refs. 1 or 7 for a careful treatment. The reader is assumed to be familiar with the properties of Gödelizations.} \]
primitive recursion, and minimalization, can be effectively obtained from \( n \) (\( n \) can be a fairly simple code for the derivation). Thus, the method of proof of Theorem 3 provides the means for computing the recursive functions \( f \) and \( g \) of the following corollary (the details are left to the reader).

**Corollary 3.** Let \( \pi \) be a Gödelization of the partial recursive functions. There is a total recursive function \( f \) such that \( f(n) \geq d(\pi(n)) \). Moreover, there is total recursive function \( g \) such that \( g(n) \geq D_n \).

Blum's axioms for step-counting functions (complexity measures) could be translated into the context of this paper as requirements that the measure be an effectively computable (partial) function from trees (especially index-argument pairs) to the natural numbers, that it be defined on a tree if and only if the tree converges, and that the set of pairs \( \langle t, n \rangle \) such that \( n \) is the measure of the tree \( t \) be a computable set. Theorem 3 shows that depth of computation cannot satisfy these axioms. Corollary 3 is the kind of result which cannot be obtained for measures which do satisfy the axioms: The minimum depth of computation (as a function of the index) is bounded by a total recursive function. It is not known whether the set of pairs \( \langle t, n \rangle \) such that \( n \) is the maximum depth of computations beginning with \( t \) is computable. It is known that the function \( h(n) = d(\pi(n)) \) of Corollary 3 is not a recursive function.

**Theorem 4.** Let \( \pi \) be a Gödelization of the partial recursive functions. Then \( h(n) = d(\pi(n)) \) is not a recursive function.

**Proof of Theorem 4.** The proof is based on the Rice–Myhill–Shapiro Theorem as presented in Ref. 3. The result used is that, if a set of numbers is the inverse image of a set of partial recursive functions under a Gödelization \( \pi \), then the set is either the empty set, the natural numbers, or a nonrecursive set. Now the function \( h(n) = d(\pi(n)) \) is total; so, if it were recursive, then the set of \( n \) such that \( h(n) = 1 \) would be recursive. This set is not empty, it is not the natural numbers, and it is the inverse image under \( \pi \) of the set of partial recursive functions uniformly computable within depth 1. Thus \( h \) cannot be recursive.

**5. Conclusion**

Of the various measures proposed in the Turing machine approach to partial recursive functions, the state-symbol product seems to be most similar to the depth measure. It satisfies a property analogous to Corollary 3. However, it is always finite, never depends on the arguments of the computation, and is a total recursive function of the index. The set of pairs \( \langle z, n \rangle \) such that \( n \) is the state-symbol product for Turing machine \( z \) is obviously computable.
The comparison of the depth measure with other proposed measures of computational complexity leads to the following list of properties suggested as useful in classifying complexity measures:

(1) the measure is finite for each computation;
(2) the measure is independent of arguments;
(3) for each partial recursive function, there is an index with uniformly minimum measure for all arguments;
(4) each partial recursive function has an index with uniformly bounded measure of computation;
(5) the maximum of the minimum measures of computation of partial recursive functions of $n$ variables exists and, as a function of $n$, is bounded by a total recursive function;
(6) the measure is an effectively computable (partial) function;
(7) the set of index-argument-measure triples is a computable set;
(8) the measure is defined, if and only if, the computation converges.

The depth measure satisfies (4) and (5) with (3), (6), and (7) open. The state-symbol product satisfies (1), (2), (3), (4), (5), (6), and (7). Blum's step counting measures satisfy (6), (7), and (8), and cannot satisfy (1), (2), (3), (4), or (5).

REFERENCES