Functional Characterizations of Uniform Log-depth and Polylog-depth Circuit Families

Stephen Bloch*
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093

Abstract

We characterize the classes of functions computable by uniform log-depth (NC¹) and polylog-depth circuit families as closures of a set of base functions. (The former is equivalent to ALOCTIME, the latter to polylogarithmic space.) The closures involve the “safe” composition of Bellantoni and Cook as well as a safe “divide and conquer” recursion; a simple change to the definition of the latter distinguishes between log and polylog depth. The proofs proceed, in one direction, by showing that safe composition and divide-and-conquer recursion preserve growth rate and circuit depth bounds, and in the other, by simulating alternating Turing machines with divide-and-conquer recursion.

1 Introduction

Much of the history of the study of polynomial-time computability has been informed by Cobham’s elegant characterization in [9] of the polynomial-time functions as the closure of certain simple functions under composition and a form of recursion. Specifically, Cobham’s “bounded recursion on notation” iterates a number of times equal not to the integer value of some parameter \( x \), but rather to \( \lfloor \log_2 x \rfloor \), the length of \( x \) when written in binary notation, and also requires that the function computed by such recursion grow no faster than some pre-existing function, i.e. no faster than some function defined by composition of base functions alone. This bounding requirement effectively oblige anyone using bounded recursion on notation to accompany every use thereof with a proof that the function so defined has only polynomial growth rate.

Bellantoni and Cook, in [3], present an alternative characterization of polynomial-time-computable functions which omits the bounding requirement and instead distinguishes syntactically between “normal” function parameters and “safe” ones, on which a restricted set of operations are permitted. Most importantly, recursion is permitted only on “normal” parameters, while each step of that recursion must treat the results of previous steps as safe. Thus the function iterated at each step can be computed in time and space bounds depending only on \( x \), not on the recursive definition. As a result, the size bounds on functions hold automatically rather than being a separate and inconvenient stipulation in the recursion scheme.

Other researchers have used similar techniques to Cobham’s to describe space-bounded and parallel complexity classes such as NC¹, which comprises problems solvable by log-depth, constant-fanin Boolean circuits and which is equivalent to O(log) time on an alternating Turing machine; NC, the problems solvable by polylog-depth, polynomial-size circuits; and \( \text{AC}^k \), the problems solvable by \( \log^k \)-depth, polynomial-size, polynomial-fanin circuits. Lind, in [11], gave a characterization of logspace, and Allen [1] a characterization of NC, both (like Cobham) explicitly bounding the growth rates of functions defined in their systems. Compton and LaFlamme [10] present a functional characterization of uniform NC¹, relying on a polynomial size bound implicit in their finite models approach. Clote, in [7], introduced a scheme of concatenation recursion on notation that corresponds to computing each bit of a function in parallel, and characterized uniform NC¹ by combining concatenation recursion on notation with a somewhat unnatural tree-evaluation function. A more elegant characterization, in [8], replaces the tree-evaluation function by Cobham-style recursion on notation with a constant (rather than polynomial) bound.

By combining Bellantoni & Cook’s “safe parame-
ters" technique with Allen's "divide and conquer" recursion, we have avoided both explicit size bounds and unnatural base functions, and characterized both alternating log time (ALOGTIME, or uniform NC$^1$) and alternating polylog time (or uniform polylog-depth circuits, or polylog space). In addition, the close similarity between these two characterizations may shed light on the difference between the corresponding complexity classes.

Section 2 defines the terminology, recursion schemes, and functions used in this paper. Section 3 states and proves the characterization of NC$^1$; most of this section is concerned with simulating the computation of a machine within our function algebra. Section 4 states and proves the similar characterization of functions computable by uniform polylog-$d$-depth circuits. Finally, Section 5 compares our methods with previous ones and discusses directions for further research.

2 Definitions

We shall think of our data not as natural numbers but primarily as finite strings over the alphabet $\Sigma = \{0,1\}$. Such strings will however often be interpreted as binary numbers, treating the rightmost symbol as least significant; by convention, the rightmost symbol of a string will be called bit 0, the second-rightmost bit 1, and so on. We use the symbol $\lambda$ to denote the empty string.

We define, not within the system but purely for notational convenience, the following functions:

1. $|z|$, the number of bits in a string $z$, or $\log(z+1)$ when thinking of $z$ as a number
2. $\text{Bit}(i,z)$, the bit in position $i$ of $z$, treating $i$ as a number

We'll also frequently use $\lfloor \log(z) \rfloor$, which has the useful property that whenever $z > 1$, $\lfloor \log(\lfloor z/2 \rfloor) \rfloor = \lfloor \log(z) \rfloor - 1$.

We now define a set of base functions. Following [3], we shall distinguish between "normal" and "safe" parameters of a function by writing the function with all normal parameters first, separated by a semicolon from the safe parameters, e.g. $f(x; y; z)$.

**Definition 1** Let BASE denote the following set of functions:

- $\lambda, 0, 1$ constants
- $\text{MSP}(z; y) =$ the leftmost $|z| - |y|$ bits of $z$ ("most significant part")
- $\text{LSP}(z; y) =$ the rightmost $|y|$ bits of $z$, with $|y| - |z|$ leading zeroes ("least significant part")
- $\text{Cond}(b; u, v) = \begin{cases} \text{LSP}(u; \max(u, v)) & \text{if } b \text{ is odd} \\ \text{LSP}(v; \max(u, v)) & \text{if } b \text{ is even} \end{cases}$ (treating $b$ as a number)
- $\text{Conc}(x, y) =$ the concatenation of $x$ and $y$
- $\text{Bh}(x) =$ the rightmost $\lfloor |x|/2 \rfloor$ bits of $x$ ("back half")
- $\text{Fh}(x) =$ the leftmost $\lfloor |x|/2 \rfloor$ bits of $x$ ("front half")
- $\text{Not}(x) =$ the one's complement of $x$
- $\text{Or}(x, y) =$ the bitwise OR of $x$ and $y$, with the length of the longer

**Example:** $\text{Ins}_0(0110) = 00101000$. Of course, $\text{Ins}_1(z)$ is redundant: it could be defined as $\text{Not}(\text{Ins}_0(\text{Not}(z)))$.

Note that each BASE function is defined in such a way that the length of the output is a simple function of the lengths of the inputs, to avoid worries about circuit uniformity. (This is the reason for the LSP's in the definition of Cond: the answer $u$ or $v$ is padded to a uniform length that will hold either.)

As we shall be using these BASE functions heavily, we shall often omit the semicolon from those that take only safe parameters for the sake of readability.

**Definition 2** (Bellantoni & Cook) We say function $f(x; y)$ is defined by safe composition from functions $g(x_1 \ldots x_r; y_1 \ldots y_s)$, $u_1(x) \ldots u_r(x)$, and $v_1(x; y) \ldots v_s(x; y)$ if

$$f(x; y) = g(u_1(x) \ldots u_r(x); v_1(x; y) \ldots v_s(x; y))$$

Note that, according to this definition, a safe parameter to $f$ can never be used as a normal parameter to any of the $u$'s or $v$'s, nor can it affect the normal parameters of $g$. However, a normal parameter to $f$ can be passed into a safe position by using a projection
function for one of the $v$'s. This “trap-door” effect will also be preserved by our recursion schemes.

**Definition 3 (Bellantoni & Cook)** A function $f(z, x; y)$ is defined by safe recursion on notation from functions $g(z, x; y)$, $h_0(z, x; u, y)$, and $h_1(z, x; u, y)$ if

\[
\begin{align*}
 f(0, x; y) &= g(x; y), \\
 f(s_i(z), x; y) &= h_i(z, x; f(z, x; y), y), \\
\end{align*}
\]

for $s_i(z) \neq 0$

where $s_i(z) = 2z + i$, for $i = 0, 1$.

When a function accepts a parameter as “safe” it effectively promises not to use that parameter to control the depth of a recursion. (So what is “safe” from the caller’s point of view might better be called “sensitive” from that of the called function.) All else being equal, then, a function which can make that promise is more powerful than one which cannot. For example, the concatenation function could be defined as $\text{Conc}(z, y)$, $\text{Conc}(x; y)$, or $\text{Conc}(z, x; y)$. Although all three compute exactly the same value, the third is the most flexible, and choosing it over the others to include in BASE will prove crucial in showing size and time bounds.

The major difference between the present system and that of [3] is the replacement of their scheme of “safe recursion on notation” by various forms of “safe divide-and-conquer recursion”. Divide-and-conquer recursion (splitting a large datum into two pieces of roughly equal size, and recursing on both) is familiar to every computer scientist, but to the author’s knowledge its first use to characterize a complexity class functionally was in [1], where the following scheme, which Allen calls “polynomial size branching recursion”, is used to build algebraic characterizations of logspace and NC. (A cosmatically different recursion scheme was called “limited recursion” in [4] and “upward tree recursion” in [10].)

**Definition 4 (Allen, modified for our notation)**

A function $f(y, x)$ is defined by polynomial size branching recursion from functions $g$ and $h$ with polynomial size bound $p(n)$ if

\[
f(y, x) = \begin{cases} 
g(y, x) & \text{if } |y| \leq 1 \\
h(y, x) & \text{if } |y| > 1 \\
f(Fh(y), x) & \text{if } |y| > 1 \\
f(Bh(y), x) & \text{if } |y| > 1 \\
\end{cases}
\]

and for all $a, b$,

$$|f(a, b)| \leq p(|a| + \sum |b|)$$

We shall make similar definitions, but by distinguishing between normal and safe parameters we can dispense with the explicit size bound $p$.

**Definition 5** A function $f(z, b, x; y)$ is said to be defined by safe divide-and-conquer recursion (henceforth safe DCR) from functions $g(z, x; y)$ and $h(z, x; u_1, u_2, y)$ if

\[
f(z, b, x; y) = \begin{cases} 
g(z, x; y) & \text{if } |z| \leq \max(|b|, 1) \\
h(z, x; y) & \text{if } |z| > |b| \\
f(Fh(z), b, x; y), & \text{if } |z| > |b| \\
f(Bh(z), b, x; y)) & \text{if } |z| > |b| \\
\end{cases}
\]

Note: One difficulty in “programming” with DCR is that each leaf of the binary computation tree computes the same function of (for the most part) the same data, distinguished only by the portion of $z$ each is given. If the recursion were required to proceed until $|z| \leq 1$, a more obvious definition, there could be only constantly many (i.e. $|\Sigma| + 1$) distinct leaf values in the whole tree, and we might expect safe DCR to define different classes of functions depending on the size of the alphabet. Specifying $b$, and hence when the recursion is to bottom out, allows leaves to be distinguished by larger data structures than a single bit, which appears necessary in order to prove the main theorems. We maximize $|b|$ with 1 because if $|b| = 0$, the recursion goes into an infinite loop by repeatedly finding $Bh(1) = 1$.

**Example:** Using safe DCR, we can define a function $COPY(x; y)$ which concatenates $|x|$ copies of $y$ and hence has the same growth rate as the “smash” operator $\#$, defined by $x \# y = 2^{2|x| |y|}$. $COPY(x; y)$

\[
\begin{align*}
\lambda & \quad \text{if } |x| = 0 \\
y & \quad \text{if } |x| = 1 \\
\text{COPY}(COPY(Fh(x); y), y) & \quad \text{if } |x| > 1 \\
\text{COPY}(Bh(x); y)) & \quad \text{or more formally,} \\
\end{align*}
\]

\[
\text{COPY}(x; y) = \text{aux}(x, 1; y) \\
\]

\[
\begin{align*}
g(x; y) & \quad \text{if } |x| \leq \max(|b|, 1) \\
\text{Conc}(\text{aux}(Fh(x); y), y, \text{aux}(Fh(x); y)) & \quad \text{if } |x| > |b| \\
\text{aux}(Bh(x); y)) & \quad \text{if } |x| > |b| \\
g(z; y) & = \text{Cond}(Fh(\text{Conc}(1, x)); y, \lambda) \\
\end{align*}
\]

The $Fh(\text{Conc}(1, x))$ in the definition of $g$ allows us to distinguish between the cases $|x| = 0$ and $|x| = 1$. Aside from this, the formal definition is obvious and unilluminating; henceforth we shall usually give such definitions by DCR in the first, less formal manner.
Applying safe DCR to this in turn yields the following:

\[
\#_3(x, y) = \begin{cases} 
1 & \text{if } |z| = 0 \\
\text{COPY}(y; \#_3(Fh(x), y)) & \text{if } |z| > 1
\end{cases}
\]

This function has the property \(|\#_3(x, y)| = |y|^{\|x\|}\). This gives the above function the growth rate \(2^{\|x\|} \cdot (|y| + o(1))\), effectively the same as that of \(#_3\).

Thus \(EXTR'_1(\cdot; COMB_1(a_1 \ldots a_k)) = a_i\) for all \(i\).

**Definition 6**

A function \(f(x, y; \bar{z}; \bar{y})\) is defined by very safe DCR from functions \(g(z, \bar{z}; \bar{y})\) and \(h(z, \bar{z}, y, u_1, u_2)\) if

\[
f(x, b, \bar{z}; \bar{y}) = \begin{cases} 
g(z, b, \bar{z}; \bar{y}) & \text{if } |z| \leq \max(|b|, 1) \\
h(z, \bar{z}, \bar{y}) & \text{if } |z| > |b|
\end{cases}
\]

Note: This differs from safe DCR in that safe DCR allows the iterated function \(h\) to itself be defined by iteration, so long as said iteration does not depend on the results of previous calls to \(h\); in safe DCR, the iterated function takes no normal parameters whatsoever, and thus cannot be defined by iteration at all. For example, the above definition of \(COPY\) is actually very safe, but the definition of \(#_3\) is not (in fact, we prove in Lemma 15 that \(#_3\) cannot be defined by very safe DCR).

**Definition 7** \(sc(BASE)\), the "safe closure of \(BASE\)", denotes the simultaneous closure of the \(BASE\) functions under safe composition and safe DCR.

**Definition 8** \(vsc(BASE)\), the "very safe closure of \(BASE\)", denotes the simultaneous closure of the \(BASE\) functions under safe composition and very safe DCR.

It is often useful to be able to combine a constant number of values into a tuple. If all the values \(\bar{z}\) are guaranteed to be the same length, this can be easily done. Let \(\bar{Z}\) be the string of zeroes of the same length as each of the \(z\)'s, then define a family of combining and extracting functions as follows:

**Definition 9**

\[
\begin{align*}
COMB_1(z) &= z \\
COMB_2(z_1, \ldots, z_{2i}) &= \text{Conc}(COMB_1(z_1, \ldots, z_i), \text{for } i > 0) \\
COMB_i(z_{i+1}, \ldots, z_{2i})
\end{align*}
\]

Thus \(EXTR'_k(\cdot; COMB_k(a_1 \ldots a_k)) = a_i\) for all \(1 \leq i \leq k\).

**Definition 10** A collection of functions \(f_i(z, b, \bar{z}; \bar{y})\) is said to be defined by simultaneous, safe divide-and-conquer recursion from functions \(h_1, \ldots, h_k\) and \(g_1, \ldots, g_k\) if for all \(i \leq k\),

\[
f_i(z, b, \bar{z}; \bar{y}) = \begin{cases} 
g_i(z, b, \bar{z}; \bar{y}) & \text{if } |z| \leq \max(|b|, 1) \\
h_i(z, \bar{z}, \bar{y}) & \text{if } |z| > |b|
\end{cases}
\]

The definition of simultaneous, very safe divide-and-conquer recursion is identical, except that all parameters to the \(h_i\)'s are treated as safe.

**Lemma 11** If for all \(b, \bar{z} = \bar{y}, i, j\) we have \(|g_j(z, b, \bar{z}; \bar{y})| \leq |b|\), and if the equations \((|u_1| = \ldots = |u_k|) \land (|v_1| = \ldots = |v_k|)\) imply \(|h_i(z, \bar{z}; \bar{y}, \bar{u}, \bar{v})| = |h_i(z, \bar{z}, \bar{y})|\), and if \(g_1, \ldots, g_k, h_1, \ldots, h_k \in sc(BASE)\), and if functions \(f\) are defined by simultaneous, safe DCR from \(\bar{y}\) and \(\bar{h}\), then \(f \subseteq sc(BASE)\).

That is, under these equal-length conditions, simultaneous safe DCR can be simulated by ordinary safe DCR.

A similar lemma holds for \(vsc(BASE)\).

**Note:** The equal-length hypothesis could be weakened considerably, but this form of the lemma is easy to prove, and suffices for our purposes.

**Proof** for either \(sc(BASE)\) or \(vsc(BASE)\): The hypotheses of equal length, by an obvious induction, imply \(|f_j(z, b, \bar{z}; \bar{y})| = |f_j(z, b, \bar{z}; \bar{y})|\) for all \(z, b, \bar{z}, \bar{y}, i, j\). Using the tuple-coding functions \(COMB\) and \(EXTR\), we can define a single function \(F(z, b, \bar{z}; \bar{y})\) as the encoding of the tuple \(f_j(z, b, \bar{z}; \bar{y})\). That is, given the value of a recursive call to \(F\), we extract each \(f_i\) with \(EXTR_i\), then apply the appropriate \(h_i\)'s.
and combine the results with \( \text{COMB}_h \).\]

Many of the results herein concern circuit families. All circuit families in this paper are uniform in the strong, \( U_B \) sense of [12]: if the circuit family has size \( Z(n) \), the extended connection language of the circuits can be recognized by a deterministic Turing machine in time \( \log(Z(n)) \). That is, paths in polynomial-sized circuits can be recognized in deterministic \( O(\log(n)) \) time, and paths in \( 2^{\log n} \) circuits can be recognized in deterministic polylog time. Our circuits may contain gates computing the constants 0 and 1, negation, two-input AND, and two-input OR. Circuits that compute functions (as opposed to relations) are assumed to be multi-output.

**Definition 12** If \( f(\bar{x}; \bar{y}) \) is a function with normal and safe inputs as specified, and \( \bar{a} \) and \( \bar{b} \) are suitable tuples of numbers, and \( A \) is an alternating algorithm that computes \( f \)'s bit-graph, then \( d_A[\bar{a}; \bar{b}] \) is the time, maximized over all \( i < |f[\bar{a}; \bar{b}]| \), necessary to compute bit \( i \) of \( f[\bar{a}; \bar{b}] \) by alternating algorithm \( A \).

Similarly, if \( F \) is a circuit family that computes \( f \)'s bit-graph, then \( d_F[\bar{a}; \bar{b}] \) is the depth, maximized over all bit positions \( i < |f[\bar{a}; \bar{b}]| \), of the circuit applicable to inputs \( \bar{a} \) and \( \bar{b} \) and bit position \( i \).

Again, if \( F \) is a multi-output circuit family that computes \( f \), then \( d_F[\bar{a}; \bar{b}] \) is the depth of the circuit applicable to inputs \( \bar{a} \) and \( \bar{b} \). Note, in all these cases, that \( d[\bar{a}; \bar{b}] \) depends only on the lengths of \( \bar{a} \) and \( \bar{b} \), not on their values.

**Definition 13** If \( f(\bar{x}; \bar{y}) \) is computable by uniform, polynomial-size, multi-output circuit family \( F \), and there is a constant \( c \) such that for all \( \bar{a}, \bar{b} \) we have \( d_F[\bar{a}; \bar{b}] \leq c \cdot \max(\log(|\bar{a}|), |\bar{b}|) \), then we say \( f \in \text{FNC}^1 \).

Similarly, if there is a constant \( c \) such that for all \( \bar{a}, \bar{b} \) we have \( d_F[\bar{a}; \bar{b}] \leq c \), then we say \( f \in \text{FNC}^0 \).

### 3 Characterizing Uniform NC\(^1\)

**Theorem 14** The following are equivalent:

1. \( f(\bar{x}; \bar{y}) \in \text{vsc}(\text{BASE}) \)
2. \( f(\bar{x}; \bar{y}) \in \text{FNC}^1 \)
3. the set \( \{(i, b, z) | \text{Bit}(i, f(\bar{x}; \bar{y})) = b \} \) is recognized by an alternating \( O(\log) \)-time machine, and \( f \) has polynomial growth rate.

**Proof**: We first prove that (1) implies that \( f \) has polynomial growth rate (Lemma 15) and can be computed bitwise by log-depth, uniform circuits (Lemma 17); (2) follows from these two facts. The equivalence \( (2 \iff 3) \) is a result of Ruzzo [12]. And we prove \( (3 \implies 1) \) by simulating an arbitrary ALGORITHMS machine in vsc(BASE) (Lemma 19), then simulating polynomial-length comprehension (Lemma 23) to build the function \( f \) from its bit-graph.

**Lemma 15** If \( f(\bar{x}; \bar{y}) \in \text{vsc}(\text{BASE}) \), then there is a polynomial \( p_f \) with nonnegative coefficients such that

\[
|f(\bar{x}; \bar{y})| \leq p_f(|\bar{y}|) \cdot \max(|\bar{y}|, 1)
\]

**Proof**: The fastest-growing functions in BASE are \( \text{Conc}(\bar{x}, \bar{y}), \text{Ins}_0(\bar{x}, \bar{y}) \), and \( \text{Ins}_1(\bar{x}, \bar{y}) \).

\[
|\text{Conc}(\bar{x}, \bar{y})| = |\bar{x}| + |\bar{y}|
\]

so they satisfy the statement of the Lemma with \( p_f \equiv 2 \); \( \text{Ins}_0 \) and \( \text{Ins}_1 \) are similar. All the remaining BASE functions have output no longer than their longest (safe) input, so \( p_f \equiv 1 \) will suffice for them.

**Definition by Safe Composition**: Let \( f(\bar{x}; \bar{y}) = g(u(\bar{x}; \bar{v}); v(\bar{z}; \bar{y})) \), and suppose \( p_g, p_u, \) and \( p_v \) are polynomials satisfying Lemma 15 for functions \( g, u, \) and \( v \) respectively. Then

\[
|f(\bar{x}; \bar{y})| \leq p_g(|u(\bar{x}; \bar{y})|) \cdot |v(\bar{z}; \bar{y})|
\]

so it would suffice to let \( p_f(|\bar{x}|) \equiv p_g(|u(\bar{x}; \bar{y})|) \cdot p_v(|\bar{z}|) \cdot \max(|\bar{y}|, 1) \).

The proof for multiple functions \( \bar{u} \) and \( \bar{v} \) is similar.

**Definition by Very Safe DCR**: Let

\[
f(f, \bar{x}; \bar{y}) = \begin{cases} 
  g(z; \bar{x}; \bar{y}) & \text{if } |z| \leq |\bar{b}| \\
  h(z; \bar{x}; \bar{y}) & \text{if } |z| > |\bar{b}| \\
  f(Fh(z; \bar{x}; \bar{y}); g(Fh(z; \bar{x}; \bar{y})) & \text{if } |z| > |\bar{b}|
\end{cases}
\]

and suppose we know, by an inductive assumption, that (for some polynomial \( p_g \) and constant \( k_h \))

\[
|g(z; \bar{x}; \bar{y})| \leq p_g(|z|, |\bar{x}|) \cdot \max(|\bar{y}|, 1)
\]
and

$$|h(z, x, y, u_1, u_2)| \leq k_h \cdot \max(|x|, |y|, |u_1|, |u_2|)$$

Then we prove, by induction on $|x|$, that

$$|f(x, b, z; y)| \leq k_h \cdot \max(|x|, |b|, |z|, p_h(|x|) \cdot \max(|y|))$$

Note that $k_h^{(\log(|x|))} \leq k_h^{\log(|y|)+1} = k_h \cdot |x|^{\log(k_h)}$ is polynomial in $|x|$, so the entire bound is polynomial in $|x|$, $|b|$, and linear in $\max(|y|)$ as desired.

The base case, for $|x| \leq |b|$, is obvious. For larger $|x|$, we know

$$|f(x, b, z; y)| \leq k_h \cdot \max(|x|, |b|, p_h(|x|) \cdot \max(|y|))$$

assumed $p_h(|x|) \geq 1$

(Note $|x| \geq 1$ because we're in the induction step, and $k_h \geq 1$ because otherwise $f$ has linear growth rate and we're done)

$$= k_h^{1+\log(|x|/2)} \cdot \max(|x|, |b|, p_h(|x|) \cdot \max(|y|))$$

which is, as desired, linear in $\max(|y|)$. As before, the proof in the case that there are multiple functions $\overline{u}$ and $\overline{v}$ is similar.

**Definition by Safe Composition:** Suppose $f(\overline{x}; \overline{y}) = g(\overline{u}(\overline{x}); \overline{v}(\overline{x}; \overline{y}))$. Suppose also that $k_g, c_g, k_u, c_u, k_v, c_v$ are constants satisfying Lemma 17 for circuit families $G$, $U$, and $V$. Finally, suppose (by Lemma 15) $p_u$ is a polynomial such that $|u(\overline{x})| \leq p_u(|\overline{x}|)$.

Then

$$d_F(\overline{x}; \overline{y}) \leq$$

$$d_G(\overline{u}(\overline{x}); \overline{v}(\overline{x}; \overline{y})) + d_U(\overline{u}(\overline{x}; \overline{v}(\overline{x}; \overline{y}))$$

where $d_G$ and $d_U$ are the corresponding depth functions for $G$ and $U$.

**Definition by Very Safe DCR:** Suppose $f(\overline{x}; \overline{y}) = g(\overline{u}(\overline{x}); \overline{v}(\overline{x}; \overline{y}))$. Suppose also that $k_g, c_g, k_u, c_u, k_v, c_v$ are constants satisfying the present lemma for circuit families $G$ and $H$ computing $g$ and $h$ respectively. Let $F$ denote the obvious circuit family for $f$: an instance of circuit $H$, replacing each of its inputs by either a smaller instance of $F$ or an instance of $G$,
depending on whether \(|z| > |b|\). Then for \(|z| > |b|\),
\[
d_f(z, b, z, \bar{z}; \bar{y}) \leq \\
\leq d_H([z, z, f(\overline{h}(z), b, \bar{z}; \bar{y}), f(\overline{b}(z), b, \bar{z}; \bar{y})] + \\
d_F(\overline{b}(z), b, \bar{z}; \bar{y}) \\
\leq c_6 + d_F(\overline{b}(z), b, \bar{z}; \bar{y}) \\
\leq |\bar{z}| \cdot c_6 + d_0[\bar{z}, \bar{z}; \bar{y}] \\
\leq |\bar{z}| \cdot c_6 + k_6 \cdot \max(|\bar{z}|, |\bar{z}|) + c_7 \\
\leq (c_6 + k_6) \cdot \max(|\bar{z}|, |\bar{z}|) + c_7
\]

Thus each bit of \(f(\bar{z}; \bar{y})\) can be computed in log depth (and therefore polynomial size). Since there are only polynomially many bits of output, the whole value of \(f\) can be computed by NC^1 circuits. This proves one direction of Theorem 14.

Lemma 19 Given an alternating Turing machine \(M\) with inputs \(\overline{X}\) and an \(O(\log(|\overline{X}|))\) time bound, there is a function \(f(\overline{X} ;) \in \text{vsc(BASE)}\) such that
\[
f(\overline{X} ;) = \begin{cases} 
1 & \text{if } M \text{ accepts } \overline{X} \\
0 & \text{if } M \text{ rejects } \overline{X} 
\end{cases}
\]
In other words, \(\text{vsc(BASE)}\) can simulate the computation of an arbitrary ALOGTIME machine.

The proof of this lemma is algorithmic and fairly technical, and takes up most of the rest of Section 3.

Proof of Lemma 19: We start by defining a number of utility functions to manipulate strings. The first is
\[
\text{ZEROES}(z; X) = \text{LSP} ; X, \lambda, X
\]
which returns a string of zeros of the same length as \(X\).

Next we define
\[
\text{PADRT}(z, y) = \text{Conc}(z, \text{ZEROES}(\text{MSP}(y, z)))
\]
which pads its argument \(z\) on the right with zeros to length \(\max(|z|, |y|)\).

We frequently wish to get at the leftmost or rightmost bit of a string.
\[
\text{LOBIT}(z) = \text{LSP}(z, 1, 1) \\
\text{DELLOBIT}(z) = \text{MSP}(z, 1) \\
\text{HIBIT}(z) = \text{MSP}(z, \text{MSP}(z, 1)) \\
\text{DELHIBIT}(z) = \text{LSP}(z, \text{MSP}(z, 1))
\]

We also define a function to shift a number without changing its length.
\[
\text{SHL}(z, y) = \text{LSP} \left( \text{Conc}(z, y), z \right)
\]
shifts \(z\) left by appending \(y\), discarding the high bits so the result is the same length as \(z\).

We would also like to be able to find the length of a string.
\[
\text{Len}(z) = \begin{cases} 
\lambda & \text{if } |z| = 0 \\
1 & \text{if } |z| = 1 \\
\text{Conc} \left( \text{Len} \left( Fh(z) ; \right) \right) & \text{if } |z| > 1
\end{cases}
\]
returns a string \(y\) such that \(|y| = |\bar{z}|\). Note that \(\text{Len}(z)\) does not return a binary representation of \(|z|\) itself.

Since a log-time Turing machine is defined with random access to its read-only input tape, we need a function to implement random access. We define \(\text{GETBIT}(X; i)\) to return bit \(i\) of the normal parameter \(X\), treating \(i\) as a binary number. By convention, if \(|i| > \log(|X|)\) we shall only pay attention to the low \(\log(|X|)\) bits of \(i\), while if \(|i| < \log(|X|)\) we pad \(i\) on the left with zeros. We also assume \(|X|\) is a power of 2. The function is defined by simultaneous, very safe DCR: let \(\text{GETBIT}(X; i) = f_1(X; X, \text{LSP}(i, \text{DELLOBIT}(\text{Len}(X))))\) where
\[
f_1(Y; X, i) = \\
\begin{cases} 
X \text{ if } Y \leq 1 \\
Fh(f_1(Fh(Y); X, i)) \text{ if } Y > 1 \text{ and } \text{HIBIT}(f_1(Fh(Y); X, i)) = 1 \\
Bh(f_1(Fh(Y); X, i)) \text{ if } Y > 1 \text{ and } \text{HIBIT}(f_1(Fh(Y); X, i)) = 0
\end{cases}
\]
\[
f_2(Y; X, i) = \\
\begin{cases} 
\text{PADRT}(i, X) \text{ if } |Y| \leq 1 \\
\text{SHL}(f_2(Fh(f_2(Fh(Y)); X, i))) \text{ if } |Y| > 1
\end{cases}
\]
Note that \(|X| = |\text{PADRT}(i, X)|\), so the equal-length requirement is met at the base level. It is also easily proven by induction that \(|f_1(Y; X, i)| = |X|/2^i\) is a power of 2 for all \(X, Y\) that arise in the recursion. In particular, \(|f_1(Fh(Y); X, i)|\) is always even, so if \(|f_1(Fh(Y); X, i)| = |f_2(Fh(Y); X, i)|\) then the lengths of \(Fh(f_1(Fh(Y); X, i))\), \(Bh(f_1(Fh(Y); X, i))\), and \(SHL(f_1(Fh(f_2(Fh(Y); X, i))))\) are all equal. Thus this definition meets all the equal-length criteria of Lemma 11, and so \(\text{GETBIT} \in \text{vsc(BASE)}\).

As for its correctness, the \(\text{GETBIT}\) function first coerces \(i\) to length \(\log(|X|)\) = \(|X| - 1\), discarding high bits if \(i\) is too long and left-padding with zeros if it's too short, and then goes into the simultaneous recursion. At each step on the way up, the high bit of \(i\) determines whether to keep the front or back half of \(X\); the other half is discarded, as is the high bit of \(i\). By the time the functions return, \(i\) has been reduced to nothing and \(X\) to a single bit, which is the answer.
Now that these general-purpose utility functions are defined, we turn our attention to the particular machine we wish to simulate. We assume without loss of generality that all computation paths in the machine are the same length, that the configurations of the machine form layers strictly alternating between universal and existential states (say, the last non-leaf layer is existential), that each non-leaf configuration has exactly two successor configurations, and that access to the input tapes is only at leaf configurations. These assumptions increase the alternating time by only a constant factor.

Suppose we're given an ALOGTIME machine $M$ with $j$ random-access input tapes and $k$ work tapes, running in time bounded by $t = d \cdot \max(||X||)$, where $X_1, \ldots, X_j$ are the contents of the input tapes. Then

**Definition 20** A configuration of $M$ is a tuple (in the sense of Definition 9)

$$(q, I_L_1, I_R_1, \ldots, I_L_j, I_R_j, W_L_1, W_R_1, \ldots, W_L_k, W_R_k)$$

where

$q$ represents finite state information,

$I_{Li}$ represents the contents of index tape $i$ to the left of the head, with the lowest-significance bits representing the squares closest to the head,

$W_L_i$ the contents of work tape $i$ to the left of the head (similarly),

$I_{Ri}$ the contents of index tape $i$ from the head rightwards (lowest-significance bits representing the square under the head), and

$W_R_i$ the contents of work tape $i$ from the head rightwards (similarly).

Each element of the tuple is a string of length exactly $t$ (assuming this is sufficient for the number of states).

Since the lengths of the elements are fixed and equal, they can all be encoded as described in Definition 9. Note that the inputs $X$ are not an explicit part of a configuration.

We define $BLOWUP(X;i)$ to yield a string of length $2^i - 1 = 2^{d \cdot \max(||X||)} - 1 \geq \max(2^i)$, so that $||BLOWUP(X;i)|| = d \cdot \max(||X||) = t$, by composing $d$ invocations of the $COPY$ function and suitable numbers of $Conc$ functions.

Define a function $INIT_M(X;i)$ to return the configuration comprising (an encoding of) the start state, $j$ empty index tapes of length $t$, and $k$ empty work tapes of the same size. This is clearly in $\text{sec}(\text{BASE})$: we can compute $	ext{Len}(BLOWUP(X;i))$ to get something of length $t$, apply $\text{ZEROES}$ to make it an empty tape, then form a tuple of finitely many such things and the start state with $\text{COMB}$.

Define two functions $NEXT_L$ and $NEXT_R$, each taking a configuration and returning another, to correspond to the "left child" and "right child" of a given configuration. Note that the current state can be tested bit-by-bit by a finite composition of $\text{Cond}$ and $\text{MSP}$ functions, and a new state can be generated by concatenating constant functions. Similarly, the currently visible square of any tape can be extracted and tested with $\text{MSP}$ and $\text{Cond}$, and tape head movement and writing can be easily done with $\text{MSP}$, $\text{Conc}$, and $\text{LSP}$. No recursion is used in the definitions of $NEXT_L$ and $NEXT_R$ so they can accept their configuration parameters as "safe".

Next we'd like to build a binary tree of all the possible bit-strings of length $t$, to be used as paths from root to leaves. But for technical reasons we actually produce a tree not of paths but of "inflated paths", in which the bits of interest appear in bit positions $0, 1, 3, 7, \ldots, 2^i - 1$, and the bits in between are ignored. (Each of $2^i$ paths is inflated to length $2^i$, so the tree has length $2^{2^i}$, which is still polynomial since $t = O(||X||)$.) Producing this binary tree, which must contain a different inflated path at each leaf, with bitwise-similar leaves near one another in the tree and with an easy mechanism for distinguishing universal branches from existential ones, is the most complex part of the proof.

**Definition 21** If $x$ is a bit-string, we define the deflation of $x$ as the bit string $y$, of length $||x||$, such that for all $i < |y|$, $\text{Bit}(i,y) = \text{Bit}(2^i - 1, x)$. Conversely, we say any such $x$ is an inflation of $y$. A string $x$ is a full inflation of $y$ if $x$ is an inflation of $y$ and $|x| = 2^{|y| - 1}$.

The function that produces this binary tree of inflated paths is defined as follows:

$$PATHS(x;) = \begin{cases} \lambda & \text{if } |x| = 0 \\ 01 & \text{if } |x| = 1 \\ 0(h(PATHS(Bh(x;)))) & \text{if } |x| > 1 \end{cases}$$

where

$$h(v) = \begin{cases} \text{Conc}((\text{Ins}_0(v), \text{Ins}_1(v))) & \text{if } \text{HIBIT}(v) \neq \text{HIBIT}(Bh(v)) \\ \text{Conc}((\text{Ins}_0(v), \text{Ins}_1(\text{Not}(v)))) & \text{if } \text{HIBIT}(v) = \text{HIBIT}(Bh(v)) \end{cases}$$

Clearly each iteration of $h$ quadruples the length of the function, so if $|x|$ is a power of 2,

$$|PATHS(x;)| = 2 \cdot 4^{\log(|x|)} = 2|x|^2$$

200
The value of $\text{PATHS}(x;)$ can be usefully interpreted as $2|z|$ blocks of length $|z|$. Indeed, it can be thought of as (the root of) a binary tree: define $N_i^{x,z}$, the $k$-th node of height $i$ in $\text{PATHS}(x;)$, to be bits $2^i|z|$ through $(k + 1) \cdot 2^i|z| - 1$, as long as these numbers are less than $2|z|^2$, the total number of bits. Then the nodes of height 0, the leaves, are the individual blocks of length $|z|$ and the left and right children of any non-leaf node are its front and back halves. Furthermore, the following are all equivalent:

- $N_i^{x,z}$ is a descendant of $N_j^{x,z}$.
- $j'2^i|z| \leq j2^i|z| < (j + 1)2^i|z| \leq (j' + 1)2^i|z|$
- $j'2^i|z| \leq j2^i|z| < (j + 1)2^i|z| \leq (j' + 1)2^i|z|$

Note that this last inequality is independent of $x$.

The less obvious properties of $\text{PATHS}$ are presented in the following technical lemma. The notation $\pm x$ denotes "either $x$ or $\overline{x}$".

**Lemma 22** Let $|x|$ be a power of 2 and $i \leq \log(2|x|) = \lceil \log|x| \rceil$. Then

1. If $|x| > 1$, and if $j \geq \lceil |x|/2^i \rceil$, then $N_j^{x,z} = \text{Ins}_0(N_j^{x,B\overline{z}(x)})$
   while if $j < \lceil |x|/2^i \rceil$, then $N_j^{x,z} = \text{Ins}_1(\pm N_j^{x,B\overline{z}(x)})$

2. For any node in $\text{PATHS}(x;)$ of height $i > 0$, the high bits of its two children are equal iff $i$ is even.

3. The following are equivalent, for any two leaves $N_1^{x,z}$ and $N_2^{x,z}$ in $\text{PATHS}(x;)$:
   a. The leaves have a common ancestor of height $i$.
   b. The leaves have the same rightmost $\lfloor |x|/2^i \rfloor$ bits.
   c. The leaves' deflations have the same rightmost $\lfloor |x|/2^i \rfloor$ bits.

4. If $\text{PATHS}(x;)$ is interpreted as a tree, its leaves comprise one full inflation of each path of length $|x|$.\n
The proof of this lemma involves a number of straightforward but tedious inductions, and we omit it, but illustrate it by the following Example:

$$\text{PATHS}(111;)=
=0000\ 0100\ 0010\ 1010\ 1111\ 1011\ 1011\ 0101,$$
8 blocks of length 4, each of which is the full inflation of a distinct 3-bit path.

$$\text{PATHS}(11111111;)=
=00000000\ 10000000\ 00001000\ 10001000\ \ldots$$

16 blocks of length 8, each of which is the full inflation of a distinct 4-bit path.

The reader may also verify the remaining properties in these examples.

Now, recalling that $||B\text{LOWUP}(x;)|| = t$, the time bound of the alternating machine, it follows that $\text{PATHS}(B\text{LOWUP}(x;))$ produces a binary tree of full inflations of all the possible length-$t$ paths. Given one such path $P$ and a starting configuration $C$, we must find the configuration reached by following that path.

$$\text{NEXT}(P, C)$$

computes $\text{NEXT}_L$ if the high bit of $P$ is set, and otherwise $\text{NEXT}_R$. Recursion on this function yields

$$\text{GETLEAF}(P, C) =
\begin{cases}
\text{NEXT}(P, C) & \text{if } |P| \leq 1 \\
\text{NEXT}(P, \text{GETLEAF}(B\text{h}(P); C)) & \text{otherwise}
\end{cases}$$

The $\text{GETLEAF}$ function chops the inflated path in half repeatedly, each time extracting a single bit of path information (this is why the path was inflated in the first place) and finding a left or a right child according to that bit; when no path remains, i.e. after $t$ choppings, it uses the root configuration $C$.

Next, having found our leaf configuration, we must decide whether it is an accepting configuration. This requires checking whether the state is an accepting state, a rejecting state, or a query state, and if the last, testing the appropriate bit of the appropriate input (using $\text{GETBIT}$ on the "normal" input parameters $\bar{X}$). No recursion is required except that to define $\text{GETBIT}$, and we can define $\text{EVALLEAF}(\bar{X}; C)$.

Now we can define the $(0,1)$-valued function $\text{EVALTREE}(T, s, \bar{X}; C)$ as follows ($s$ is intended to
be any fully-inflated path, i.e. \( |s| = 2^{t-1} \):

\[
EVALTREE(T, s, \bar{X}; C) = \begin{cases} 
EVALLEAF(\bar{X}; \text{GETLEAF}(T; C)) \\
EVALTREE(Fh(T), s, \bar{X}; C) \\
EVALTREE(Bh(T), s, \bar{X}; C) \\
\end{cases}
\]

if \( |T| \leq |s| \)

or, more formally,

\[
EVALTREE(T, s, \bar{X}; C) = \begin{cases} 
EVALLEAF(\bar{X}; \text{GETLEAF}(T; C)) \\
h(T; T) \\
EVALTREE(Fh(T), s, \bar{X}; C), \\
EVALTREE(Bh(T), s, \bar{X}; C) \\
\end{cases}
\]

if \( |T| > |s| \)

where

\[
h(T, u_1, u_2) = \begin{cases} 
\text{Cond}(s; \text{HIBIT}(T), \\
\text{Cond}(s; \text{HIBIT}(Bh(T)), \\
\text{Cond}(s; 1, u_2), \\
\text{Cond}(s; u_1, u_2)), \\
\text{Cond}(s; \text{HIBIT}(Bh(T)), \\
\text{Cond}(s; u_1, u_2), \\
\text{Cond}(s; u_1, 1, u_2)) \\
\end{cases}
\]

Finally, \( |\text{BLOWUP}(\bar{X}; s)| = 2^t - 1 \) and so \( |\text{Bh}(\text{BLOWUP}(\bar{X}; s))| = \left\lceil \frac{2^t - 1}{2} \right\rceil = 2^{t-1} \), so the simulation of the ATM can be accomplished by

\[
\text{SIMULATE}(\bar{X}; s) = \text{EVALTREE}(\text{PATHS}(\text{BLOWUP}(\bar{X}; s)), \\
\text{INIT}_{\text{M}}(\bar{X}; s))
\]

Thus ALOGLTIME \( \subseteq \text{vsc}(\text{BASE}) \). This completes the proof of Lemma 19.

Lemma 23: If \( F(\bar{x}; s) \) has exactly polynomial growth rate, and there is a function \( f(\bar{x}; i; s) \in \text{vsc}(\text{BASE}) \) such that

\[
(\forall i < |F(\bar{x}; s)|)(f(\bar{x}; i; s) = Bt(i, F(\bar{x}; s)))
\]

then \( F(\bar{x}; s) \in \text{vsc}(\text{BASE}) \).

Proof of Lemma 23: We have already noted that any polynomial growth rate can be attained in \( \text{vsc}(\text{BASE}) \), so we can define a function \( \text{SIZE}_F \) such that \( |\text{SIZE}_F(\bar{x}; s)| = |F(\bar{x}; s)| \). Now we build a binary tree, similar to \( \text{PATHS} \), each of whose leaves contains a different value, but this time the values are the binary numbers \( 0 \ldots |\text{SIZE}_F(\bar{x}; s)| - 1 \) in reverse order.

We can then define \( F \) by very safe DCR on such a tree \( T \) as follows:

\[
F(\bar{x}; s) = F'(T, \text{Len}(\text{SIZE}_F(\bar{x}; s)); \bar{x}; y)
\]

or

\[
F'(T, z, \bar{x}; z) = \begin{cases} 
F(T, z, \bar{x}; z) & \text{if } |T| \leq |z| \\
\text{Conc}(F(Fh(T), z, \bar{x}; z), F(Bh(T), z, \bar{x}; z)) & \text{if } |T| > |z|
\end{cases}
\]

The iterated function is \( \text{Conc} \), which is associative; so the effect is simply to concatenate the bits \( f([z]-1, \bar{x}), f([z]-2, \bar{x}), \ldots f(1, \bar{x}), f(0, \bar{x}) \), where the numbers \( [z]-1, [z]-2, \ldots, 1,0 \) are given to \( f \) in binary. (Note that if \( F \) were allowed safe parameters, and they affected its length, \( \text{SIZE}_F \) would depend on those parameters, and therefore could not be used in a normal position in \( F' \). This is why the \( F \) in this lemma only takes normal parameters; the lemma is somewhat weaker than we might like, but it suffices to prove Theorem 14.)

All that remains is to build, given a string \( z \), a list of the numbers from \( 0 \ldots |z| - 1 \) in binary, in reverse order. We define it by simultaneous, very safe DCR:

\[
\text{COUNTUP}(z; s) = f_2(z; \text{Len}(z; s))
\]

or

\[
f_1(z; b) = \begin{cases} 
\lambda & \text{if } |z| = 0 \\
\text{SHL}(\text{ZEROES}; b, 0) & \text{if } |z| = 1 \\
\text{Conc}(\text{SHL}(\text{SHL}(f_1(Bh(z); b), 0)); f_1(Bh(z); b)) & \text{if } |z| > 1
\end{cases}
\]

\[
f_2(z; b) = \begin{cases} 
\lambda & \text{if } |z| = 0 \\
\text{ZEROES}(z; b) & \text{if } |z| = 1 \\
\text{Conc}(\text{Or}(f_1(Bh(z); b), f_2(Bh(z); b)), f_3(Bh(z); b)) & \text{if } |z| > 1
\end{cases}
\]

Clearly, \( |f_1| = |f_2| = |b| \) at the base level, and each of \( f_1 \) and \( f_2 \) exactly doubles in length at each recursive step, so this definition meets the requirements of Lemma 11.

As for how it works, \( f_1(z; b) \) is the concatenation of \( |z| \) identical blocks of length \( |b| \), each composed of zeroes except for a 1 in bit position \( |z| - 1 \); shifting such a list of blocks left by one bit has the effect of shifting over the 1 in each block simultaneously. (All this is precisely correct only if \( |z| \) is a power of 2, which we shall assume henceforth.)
$f_2(z; b)$ starts as a block of zeroes (for $|z| = 1$), and at each subsequent step two copies of it are concatenated, the first having a high bit set in each of its blocks by OR-ing with $f_1(z; b)$. Thus if $f_2(Bh(z); b)$ lists the binary numbers $\frac{|b|}{2} - 1, \ldots, 1, 0$ followed by $\frac{|b|}{2} - 1, \ldots, 1, 0$, i.e. $|z| = 1, \ldots, 0$ as desired.

Plugging the resulting $COUNTUP(z;\cdot)$ in place of the $T$ in the above definition of $F$ completes the proof of the lemma. This comprehension capability allows us to pass from a function defined by its bit-graph (as produced by Lemma 19) to the function itself. In combination with Lemmas 15 and 17, this completes the proof of Theorem 14.

### 4 Characterizing Uniform Polylog Depth

**Theorem 24** The following are equivalent:

1. $f(\vec{z}; i) \in sce(BASE)$
2. $f(\vec{z})$ can be computed from $\vec{z}$ by a LOGTIME-uniform, polylog depth circuit family, and $f$ has $2^{p_2(|\vec{z}|)}$ growth rate
3. the $i$-th bit of $f(\vec{z})$ can be computed from $i$ and $\vec{z}$ in alternating polylog time, and $f$ has $2^{p_2(|\vec{z}|)}$ growth rate
4. $f(\vec{z})$ can be computed by a deterministic Turing machine in polylogarithmic space (not counting a write-only output tape)

The proof is similar to that of Theorem 14. We prove (1 $\Rightarrow$ 2) and (3 $\Rightarrow$ 1) directly. (2 $\Leftrightarrow$ 3) is a result of Ruzzo [12]. (3 $\Rightarrow$ 4) follows from a result of Borodin, quoted in [6] as Theorem 4.2. And (4 $\Rightarrow$ 3) is easily proven by a technique similar to that of Savitch's Theorem.

**Lemma 25** If $f(\vec{z}; \vec{y}) \in sce(BASE)$, then there is a polynomial $p_f$ with nonnegative coefficients such that

$$|f(\vec{z}; \vec{y})| \leq 2^{p_f(|\vec{z}|)} \cdot \max(|\vec{y}|, 1)$$

**Proof of Lemma 25:** As pointed out before, all the BASE functions have linear growth rate, easily satisfying this criterion.

**Definition by Safe Composition:** If $f(\vec{z}; \vec{y})$ is defined by safe composition as $g(u(\vec{z}); v(\vec{z}; \vec{y}))$, and if $p_u$ and $p_v$ are polynomials satisfying the Lemma for functions $g$, $u$, and $v$ respectively, then

$$|f(\vec{z}; \vec{y})| \leq 2^{p_u(|\vec{z}|)} \cdot |v(\vec{z}; \vec{y})| \leq 2^{p_u(2^{p_v(|\vec{z}|)})} \cdot 2^{p_v(|\vec{y}|)} \cdot \max(|\vec{y}|, 1) \leq 2^{p_u(2^{p_v(|\vec{y}|)+1})} \cdot 2^{p_v(|\vec{y}|)} \cdot \max(|\vec{y}|, 1) \leq 2^{p_u(2^{p_v(|\vec{y}|)+1})} \cdot \max(|\vec{y}|, 1)$$

so the statement is satisfied by letting

$$p_f(|\vec{z}|) = p_u(|\vec{z}|) + 1 + p_v(|\vec{z}|)$$

**Definition by Safe DCR:** On the other hand, if $f(z, b; \vec{y})$ is defined by safe DCR on the parameter $z$, to wit:

$$f(z, b, \vec{z}; \vec{y}) = \begin{cases} g(z, \vec{z}; \vec{y}) & \text{if } |z| \leq \max(|b|, 1) \\ h(z, \vec{z}; f(Fh(z), b, \vec{z}; \vec{y}), f(Bh(z), b, \vec{z}; \vec{y}), \vec{y}) & \text{if } |z| > |b| \end{cases}$$

and $p_g$ and $p_h$ are polynomials satisfying the Lemma for functions $g$ and $h$ respectively, then we prove by induction on $|z|$ that

$$|f(z, b, \vec{z}; \vec{y})| \leq 2^{\log(|z|)+1} \cdot 2^{p_g(|\vec{y}|)+p_h(|\vec{y}|)} \cdot \max(|\vec{y}|, 1)$$

The base case, for $|z| \leq |b|$, is obvious. For larger $|z|$, we know

$$|f(z, b, \vec{z}; \vec{y})| \leq 2^{\log(|z|)+1} \cdot \max(|f(Fh(z), b, \vec{z}; \vec{y})|, |f(Bh(z), b, \vec{z}; \vec{y})|, |\vec{y}|) \leq 2^{p_g(|\vec{y}|)+1} \cdot \max \left(2^{\log(|z|)+1} \cdot 2^{p_h(|\vec{y}|)+1}, 2^{p_h(|\vec{y}|)+1} \cdot \max(|\vec{y}|, 1) \right) \leq 2^{p_g(|\vec{y}|)+1} \cdot \max \left(2^{\log(|z|)+1} \cdot 2^{p_h(|\vec{y}|)+1}, 2^{\log(|z|)+1} \cdot 2^{p_h(|\vec{y}|)+1} \cdot \max(|\vec{y}|), 2^{p_h(|\vec{y}|)+1} \right) \leq 2^{\log(|z|)+1} \cdot 2^{p_h(|\vec{y}|)+1} \cdot \max(|\vec{y}|)$$

as desired.

**Lemma 26** If $f(\vec{z}; \vec{y}) \in sce(BASE)$, then there is a polynomial $q_f$ with nonnegative coefficients and a uniform circuit family $F$ computing $f$'s bit-graph such that for all $\vec{z}, \vec{y}$,

$$d_F[\vec{z}; \vec{y}] \leq q_f(|\vec{z}|) \cdot \max(|\vec{y}|, 1)$$
Proof of Lemma 26: All the BASE functions can obviously be computed in such a fashion. For the remaining cases, we assume wolog that \(d_F[z, u, w, g] \) and the like are monotone in the lengths of their arguments.

Definition by Safe Composition: If \( f(z, g) \) is defined by safe composition as \( g(u(z), z, g) \), then let \( p_u, p_v, \) and \( p_w \) be polynomials whose exponents bound the growth of \( u, v, \) and \( w \) respectively as in Lemma 25, and let \( q_u, q_v, \) and \( q_w \) be polynomials bounding the circuit depth required to compute \( u, v, \) and \( w \) with circuit families \( G, U, \) and \( V \) respectively, by an inductive hypothesis on the present lemma. And let \( F \) be the obvious circuits for \( f: a \) copy of circuit \( G, \) with each input replaced by a copy of circuit \( U \) or \( V \) as appropriate.

\[
d_F[z, w, g] \leq \max \left( d_G[w, z, g], d_F[z, w, g] \right)
\]

so if \( h(z, x, u, w, g) \) is called in the course of computing \( f(z, x, u, w, g) \), we can be assured that

\[
|u|, |w| \leq 2^n(|z|, |b|, |x|, |g|) \cdot \max(|g|, 1)
\]

and therefore \( |u| \) and \( |w| \) are each at most \( p_f(|z|, |b|, |x|, |g|) + \max(|g|, 1) \). With this in mind, then,

\[
d_F[z, w, g] \leq \max \left( d_G[z, w, g], d_H[z, w, g, f(Bh(z), b, z, g)], d_F[Bh(z), b, z, g] \right)
\]

by monotonicity of \( d_F \)

\[
d_G[z, w, g] \leq \max \left( q_u(|z|), q_v(|z|), q_w(|z|) \right) \cdot \max(|g|, 1)
\]

so the statement is satisfied by letting

\[
q_f(z, w, g) = q_u(|z|) \cdot q_v(|z|) + q_w(|z|)
\]

Again, the proof in the case that there are multiple functions \( u' \) and \( v' \) is similar.

Definition by Safe DCR: Now suppose \( f(z, b, x, g) \) is defined by safe DCR on the parameter \( z \), to wit:

\[
f(z, b, x, g) = \begin{cases} f(z, b, x, g) \quad & \text{if } |z| \leq |b| \\ f(z, b, x, g) \quad & \text{if } |z| > |b| \end{cases}
\]

Let \( p_f, p_g, \) and \( p_h \) be polynomials satisfying Lemma 25 for functions \( f, g, \) and \( h \) respectively, and let \( q_f, q_g, \) and \( q_h \) be polynomials satisfying Lemma 26 for circuit families \( G \) and \( H \). And let \( F \) be the obvious circuits for \( f: \) an instance of circuit \( H, \) but replacing each of its inputs by a smaller instance of \( F \) or an instance of \( G, \) depending on whether \( |z| > |b| \).

We know that, for all \( |z'| \leq |z|, \)

\[
|f(z', x, g)| \leq 2^n(|z'|, |b|, |x|, |g|) \cdot \max(|g|, 1)
\]

Thus each bit of \( f(z, g) \) can be computed in polylog depth, \( 2^n \cdot \text{polylog} \) size. Since there are at most \( 2^n \cdot \text{polylog} \) bits of output, the whole function value is computable by polylog-depth, \( 2^n \cdot \text{polylog} \)-size circuits. This is one direction of Theorem 24.

For the remaining direction, we shall show that

Lemma 27: Given an alternating Turing machine \( M \) with inputs \( X \) and time bound polynomial in \( \log(|X|) \), there is a function \( f \in \text{sc}(\text{BASE}) \) such that

\[
f(X) = \begin{cases} 1 \quad & \text{if } M \text{ accepts } X \\ 0 \quad & \text{if } M \text{ rejects } X \end{cases}
\]

In other words, \( \text{sc}(\text{BASE}) \) can simulate the computation of an arbitrary polylog-time alternating machine.
The proof of this is almost identical to that of Lemma 19. The difference is that with merely-safe DCR we can define not only $\text{COPY}$ but $\#_3$, and with this growth rate we can build and evaluate a binary tree of polylog rather than merely $O(\log)$ depth.

With $\#_3$ in hand, then, and given an alternating machine with time bound $t = \text{pm}(||X||)$, we can redefine $\text{BLOWUP}(X;\cdot)$ to have length $2^{\text{pm}(||X||)} - 1$, and hence $||\text{BLOWUP}(X;\cdot)|| \geq \text{pm}(||X||)$, by composing $	ext{deg}(<\text{pm}>)$ invocations of the $\#_3$ function as well as suitable numbers of $\text{COPY}$ and $\text{Conc}$ functions.

We define the functions $\text{ZEROES, INITM, GETBIT, NEXT_L, NEXT_R, NEXT, PATHS, EVALLEAF, GETLEAF, EVALTREE,}$ and SIMULATE exactly as before; the result is that $\text{EVALTREE}$ now evaluates an alternating and/or tree of depth $\text{pm}(||X||)$ rather than $d \cdot \text{max}(||X||)$, and thus correctly simulates the alternating polylog-time Turing machine.

**Lemma 28:** If $F(\vec{x};\cdot)$ has growth rate $2^{\text{pm}(||\vec{x}||)}$ and there is a function $f(\vec{x};\cdot) \in \text{sc}(\text{BASE})$ such that $(\forall i < ||F(\vec{x};\cdot)||)(f(\vec{x};i) = \text{Bit}(i,F(\vec{x};\cdot)))$, then $F(\vec{x};\cdot) \in \text{sc}(\text{BASE})$.

Proved exactly as Lemma 23, with larger growth rates.

## 5 Conclusions and Observations

The most unfamiliar functions in BASE are, of course, $\text{Ins}_0$ and $\text{Ins}_1$, although they are in $\text{FNC}^0$ and hence not computationally difficult. An alternate way to prove the theorems in this paper would be to omit them but add some form of concatenation recursion, analogous to Clote's Concatenation Recursion on Notation (see [7]).

Our Theorem 14 bears a resemblance to Theorem 3.3 of [10], which first defines the "simple functions" as the closure of a few specific functions (similar to our BASE functions) under composition, and then characterizes $\text{ALOGTIME}$ as the closure of the simple functions, together with an equality test, under composition and "upward tree recursion", a scheme that allows branching iteration of any simple function. Our analogue of Compton & LaFlamme's simple functions are all those functions with no normal parameters, a purely syntactic definition which we feel makes the recursion scheme's definition cleaner. In addition, we need no $a$ priori polynomial size bound, need not treat the equality test as a special case, and our bit-string formulation seems more natural than the finite-models formulation in that paper.

Theorem 24 can be seen as analogous to Theorem 2.1 of [1], which characterizes $\text{NC}$, circuits of simultaneous polylog depth and polynomial size. The reason Allen's system describes $\text{NC}$ while Theorem 24 describes polylog-depth circuits is simply that Allen's recursion schemes involve an explicit polynomial size bound. ([1] actually gives both a recursion-theoretic and a proof-theoretic characterization of $\text{NC}$, the latter using the theory $D^1$, which is equivalent to the bounded arithmetic theory $R^2_1$. Buss, Krajíček, and Takeuti [5], by replacing $R^2_1$ with $R^2_3$, have characterized polylog space proof-theoretically.) The present result also requires only two function-combination schemes (safe composition and safe DCR) rather than the four (composition, p-comprehension, log-bounded recursion on notation, and polynomial-bounded branching recursion) in [1], and uses simpler base functions.

The technique of distinguishing safe parameters pioneered in [3] has now been extended to characterize both polylog space and alternating log time, and has proven itself to be a clean, general idea. We speculate that further variants thereon may yield useful characterizations of log space, $\text{NC}$, $\text{NC}^k$, and other complexity classes. (Bellantoni, in [2], extends the characterization of $\mathcal{P}$ to the polynomial hierarchy. In recent personal communication he claims to have characterized logspace and $\text{NC}$ in a similar manner, but the author has not seen the proofs.) In addition, these clean functional characterizations may provide techniques for developing new proof-theoretic characterizations in the style of [4].

## References


