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 (NC^1) and polylog-depth circuit families as closures of a set of base functions. (The former is equivalent to ALOGTIME, 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 $|x|$, 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 obliges 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 NC^1, 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 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 NC^1, 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 NC^1 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-

*This work was begun while the author attended the 1991 Conference on Proof Theory, Complexity and Arithmetic, supported by NSF/ČSAV grant INT- 8914569. Further work was supported by grants from the University of Manitoba.

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 $\log^{O(1)}$-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 λ to denote the empty string.

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

$$\begin{array}{rcl} |x| & = & \text{the number of bits in a string } x\text{, or} \\ & & \lceil \log(x+1) \rceil \text{when thinking of } x \\ & & \text{as a number} \\ Bit(i, x) & = & \text{the bit in position } i \text{ of } x, \\ & & \text{treating } i \text{ as a number} \end{array}$$

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

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:*

$$\pi_k^{i,j}(x_1, \ldots, x_i; x_{i+1}, \ldots, x_{i+j}) = x_k,$$

the projection functions,

$$\begin{array}{rcl} \lambda, 0, 1 & & \textit{constants} \\ MSP(; x, y) & = & \textit{the leftmost } |x| \dot{-} |y| \textit{ bits of } x \\ & & \textit{("most significant part")} \\ LSP(; x, y) & = & \textit{the rightmost } |y| \textit{ bits of } x\textit{, with} \\ & & |y| \dot{-} |x| \textit{ leading zeroes} \\ & & \textit{("least significant part")} \\ Cond(; b, u, v) & = & \begin{cases} LSP(; u, \max(u, v)) \textit{ if } b \textit{ is odd} \\ LSP(; v, \max(u, v)) \textit{ if } b \textit{ is even} \end{cases} \\ & & \textit{(treating } b \textit{ as a number)} \\ Conc(; x, y) & = & \textit{the concatenation of } x \textit{ and } y \\ Bh(; x) & = & \textit{the rightmost } \lceil |x|/2 \rceil \textit{ bits of } x \\ & & \textit{("back half")} \\ Fh(; x) & = & \textit{the leftmost } \lfloor |x|/2 \rfloor \textit{ bits of } x \\ & & \textit{("front half")} \\ Not(; x) & = & \textit{the one's complement of } x \\ Or(; x, y) & = & \textit{the bitwise OR of } x \textit{ and } y, \\ & & \textit{with the length of the longer} \\ Ins_0(; x) & = & x \textit{ with a 0 inserted after each bit} \\ Ins_1(; x) & = & x \textit{ with a 1 inserted after each bit} \end{array}$$

Example: $Ins_0(; 0110) = 00101000$. Of course, $Ins_1(; x)$ is redundant: it could be defined as $Not(; Ins_0(; Not(; x)))$.

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 $f(\vec{x}; \vec{y})$ is defined by* safe composition *from functions $g(x_1 \ldots x_r; y_1 \ldots y_s)$, $u_1(\vec{x};) \ldots u_r(\vec{x};)$, and $v_1(\vec{x}; \vec{y}) \ldots v_s(\vec{x}; \vec{y})$ if*

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

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 $f(z, \vec{x}; \vec{y})$ is defined by* safe recursion on notation *from functions $g(\vec{x}; \vec{y})$, $h_0(z, \vec{x}; u, \vec{y})$, and $h_1(z, \vec{x}; u, \vec{y})$ if*

$$\begin{aligned} f(0, \vec{x}; \vec{y}) &= g(\vec{x}; \vec{y}) \\ f(s_i(z), \vec{x}; \vec{y}) &= h_i(z, \vec{x}; f(z, \vec{x}; \vec{y}), \vec{y}) \\ \text{for } s_i(z) &\neq 0 \end{aligned}$$

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 $Conc(x, y;)$, $Conc(x; y)$, or $Conc(; x, y)$. 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 cosmetically different recursion scheme was called "limited recursion" in [4] and "upward tree recursion" in [10].)

Definition 4 (Allen, modified for our notation) *A function $f(y, \vec{x})$ is defined by* polynomial size branching recursion *from functions g and h with polynomial size bound $p(n)$ if*

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

and for all $a, \vec{b}$,

$$|f(a, \vec{b})| \leq p\left(|a| + \sum |\vec{b}|\right)$$

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

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

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 $x \# y = 2^{|x| \cdot |y|}$).

$$\begin{aligned} &COPY(x; y) = \\ &= \begin{cases} \lambda & \text{if } |x| = 0 \\ y & \text{if } |x| = 1 \\ Conc(COPY(Fh(x); y), & \text{if } |x| > 1 \\ \qquad COPY(Bh(x); y)) & \end{cases} \end{aligned}$$

or more formally,

$$\begin{aligned} &COPY(x; y) = aux(x, 1; y) \\ &aux(x, b; y) = \\ &= \begin{cases} g(x; y) & \text{if } |x| \leq \max(|b|, 1) \\ Conc(; aux(Fh(; x); y), & \text{if } |x| > |b| \\ \qquad aux(Bh(; x); y)) & \end{cases} \\ &g(x; y) = Cond(; Fh(; Conc(; 1, x)), y, \lambda) \end{aligned}$$

The $Fh(; Conc(; 1, x))$ 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:

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

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 the $\#_3$ operator defined by $x\#_3 y = 2^{|x|\#|y|}$.

Definition 6
A function $f(z, b, \vec{x}; \vec{y})$ is defined by very safe DCR *from functions $g(z, \vec{x}; \vec{y})$ and $h(; z, \vec{x}, \vec{y}, u_1, u_2)$ if*

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

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 very 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 $\vec{x}$ are guaranteed to be the same length, this can be easily done. Let Z be the string of zeroes of the same length as each of the x's, then define a family of combining and extracting functions as follows:

Definition 9

$$COMB_1(; x) = x$$

$$COMB_{2i}(; x_1, \ldots x_{2i}) = \\ = Conc(COMB_i(; x_1, \ldots x_i), \quad \text{for } i > 0 \\ COMB_i(; x_{i+1}, \ldots x_{2i}))$$

$$COMB_{2i+1}(; x_1, \ldots x_{2i+1}) = \\ = Conc(COMB_i(; x_1, \ldots x_{i+1}), \quad \text{for } i > 0 \\ COMB_i(; x_{i+2}, \ldots x_{2i+1}, Z))$$

$$EXTR_1^1(; y) = y$$

$$EXTR_k^i(; y) = \\ = \begin{cases} EXTR^i_{\lceil k/2 \rceil}(; Fh(y)) & \text{if } i \leq \lceil k/2 \rceil \\ EXTR^{i-\lceil k/2 \rceil}_{\lceil k/2 \rceil}(; Bh(y)) & \text{if } i > \lceil k/2 \rceil \end{cases}$$

Thus $EXTR_k^i(; COMB_k(; a_1 \ldots a_k)) = a_i$ for all $1 \leq i \leq k$.

Definition 10 *A collection of functions $f_1(z, b, \vec{x}; \vec{y}), \ldots f_k(z, b, \vec{x}; \vec{y})$ 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 $i \leq k$,*

$$f_i(z, b, \vec{x}; \vec{y}) = \\ = \begin{cases} g_i(z, \vec{x}; \vec{y}) \\ \quad \text{if } |z| \leq \max(|b|, 1) \\ h_i(z, \vec{x}; \vec{y}, \\ \quad f_1(Fh(z), b, \vec{x}; \vec{y}) \ldots f_k(Fh(z), b, \vec{x}; \vec{y}), \\ \quad f_1(Bh(z), b, \vec{x}; \vec{y}) \ldots f_k(Bh(z), b, \vec{x}; \vec{y})) \\ \quad \text{if } |z| > |b| \end{cases}$$

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 $b, \vec{x}, \vec{y}, i, j$ we have $|g_i(z, \vec{x}; \vec{y})| = |g_j(z, \vec{x}; \vec{y})|$ for $|z| \leq |b|$, and if the equations $(|u_1| = \ldots = |u_k|) \wedge (|v_1| = \ldots = |v_k|)$ imply $|h_i(z, \vec{x}; \vec{y}, \vec{u}, \vec{v})| = |h_j(z, \vec{x}; \vec{y}, \vec{u}, \vec{v})|$, and if $g_1 \ldots g_k, h_1 \ldots h_k \in sc(BASE)$, and if functions $\vec{f}$ are defined by simultaneous, safe DCR from $\vec{g}$ and $\vec{h}$, then*

$$\vec{f} \subseteq sc(BASE)$$

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 $|f_i(z, b, \vec{x}; \vec{y})| = |f_j(z, b, \vec{x}; \vec{y})|$ for all $z, b, \vec{x}, \vec{y}, i, j$. Using the tuple-coding functions $COMB$ and $EXTR$, we can define a single function $F(z, b, \vec{x}; \vec{y})$ as the encoding of the tuple $f_1(z, b, \vec{x}; \vec{y}), \ldots f_k(z, b, \vec{x}; \vec{y})$. That is, given the value of a recursive call to F, we extract each f_i with $EXTR_k^i$, then apply the appropriate h's and combine the results with $COMB_k$. ∎

Many of the results herein concern circuit families. All circuit families in this paper are uniform in the strong, U_E 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^{polylog}-sized 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 $f(\vec{x};\vec{y})$ is a function with normal and safe inputs as specified, and $\vec{a}$ and $\vec{b}$ are suitable tuples of numbers, and A is an alternating algorithm that computes f's bit-graph, then $d_A[\vec{a};\vec{b}]$ is the time, maximized over all $i < |f(\vec{a};\vec{b})|$, necessary to compute bit i of $f(\vec{a};\vec{b})$ by alternating algorithm A.*

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

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

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

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

3 Characterizing Uniform NC^1

Theorem 14 *The following are equivalent:*

1. *$f(\vec{x};) \in vsc(BASE)$*
2. *$f(\vec{x};) \in FNC^1$*
3. *the set $\{(i,b,\vec{x})|Bit(i,f(\vec{x};)) = b\}$ 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 ($3 \Rightarrow 1$) by simulating an arbitrary ALOGTIME 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 $f(\vec{x};\vec{y}) \in vsc(BASE)$, then there is a polynomial p_f with nonnegative coefficients such that*

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

Note: The 1 is included to make things behave properly in case there are no "safe" parameters $\vec{y}$, or they're all zero-length.

Corollary 16 *If $f(;\vec{y}) \in vsc(BASE)$, then f has linear growth rate.*

This corollary is in fact the key to the proof, since the iterated function in very safe DCR must have no normal parameters: when a function of growth rate $n \mapsto kn$ is iterated $\log(|z|)$ times, the result has length $k^{\log(|z|)}n \doteq |z|^{\log(k)}n$, which is polynomial.

Proof : The fastest-growing functions in BASE are $Conc(;x,y)$, $Ins_0(;x,y)$, and $Ins_1(;x,y)$.

$$\begin{aligned} |Conc(;x,y)| &= |x|+|y| \\ &\leq 2\cdot\max(|x|,|y|) \end{aligned}$$

so they satisfy the statement of the Lemma with $p_f \equiv 2$; Ins_0 and 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 $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$, and suppose p_g, p_u, and p_v are polynomials satisfying Lemma 15 for functions g, u, and v respectively. Then

$$\begin{aligned} |f(\vec{x};\vec{y})| &\leq p_g(|u(\vec{x};)|)\cdot|v(\vec{x};\vec{y})| \\ &\leq p_g(p_u(|\vec{x}|))\cdot p_v(|\vec{x}|)\cdot\max(|\vec{y}|,1) \end{aligned}$$

so it would suffice to let $p_f(|\vec{x}|) = p_g(p_u(|\vec{x}|))\cdot p_v(|\vec{x}|)$. The proof for multiple functions $\vec{u}$ and $\vec{v}$ is similar.

Definition by Very Safe DCR: Let

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

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

$$|g(z,\vec{x};\vec{y})| \leq p_g(|z|,|\vec{x}|)\cdot\max(|\vec{y}|,1)$$

and

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

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

$$\begin{aligned} &|f(z,b,\vec{x};\vec{y})| \leq \\ &\leq \quad k_h^{\lceil\log(|z|)\rceil} \cdot \max(|z|,|b|,|\vec{x}|,p_g(|b|,|\vec{x}|)\cdot\max(|\vec{y}|)) \end{aligned}$$

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

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

$$\begin{aligned} &|f(z,b,\vec{x};\vec{y})| \leq \\ &\leq \quad k_h \cdot \max(|z|,|\vec{x}|,|\vec{y}|, \\ &\qquad\qquad |f(Bh(z),b,\vec{x};\vec{y})|,|f(Fh(z),b,\vec{x};\vec{y})|) \\ &\leq \quad k_h \cdot \max(|z|,|\vec{x}|,|\vec{y}|,k_h^{\lceil\log(\lceil\frac{|z|}{2}\rceil)\rceil} \cdot \\ &\qquad \max(\left\lceil\frac{|z|}{2}\right\rceil,|b|,|\vec{x}|,p_g(|b|,|\vec{x}|)\cdot\max(|\vec{y}|))) \\ &\leq \quad k_h^{1+\lceil\log(\lceil|z|/2\rceil)\rceil} \cdot \\ &\qquad \max(|z|,|b|,|\vec{x}|,p_g(|b|,|\vec{x}|)\cdot\max(|\vec{y}|)) \\ &\qquad \text{assuming } p_g(|b|,|\vec{x}|) \geq 1 \\ &\qquad \text{(Note } |z| \geq 1 \text{ because we're in the induction} \\ &\qquad \text{step, and } k_h \geq 1 \text{ because otherwise } f \text{ has} \\ &\qquad \text{linear growth rate and we're done)} \\ &= \quad k_h^{1+\lceil\log(|z|)\rceil-1} \cdot \\ &\qquad \max(|z|,|b|,|\vec{x}|,p_g(|b|,|\vec{x}|)\cdot\max(|\vec{y}|)) \\ &= \quad k_h^{\lceil\log(|z|)\rceil} \cdot \max(|z|,|b|,|\vec{x}|,p_g(|b|,|\vec{x}|)\cdot\max(|\vec{y}|)) \end{aligned}$$

as desired. ∎

Lemma 17 *If $f(\vec{x};\vec{y}) \in vsc(BASE)$, then there are constants k_f and c_f and a LOGTIME-uniform circuit family F computing the bit-graph of f with depth bound*

$$d_F[\vec{x};\vec{y}] \leq k_f \cdot \max(||\vec{x}||) + c_f$$

Corollary 18 *If $f(;\vec{y}) \in vsc(BASE)$, then f can be computed by a LOGTIME-uniform family of constant-depth circuits; that is, it's in FNC^0.*

Again the corollary is in fact the key to the proof: when a function of constant circuit depth is iterated $\log(|z|)$ times, the result has depth $O(\log(|z|))$.

Proof: The BASE functions all have extremely simple circuits. The circuits for $\lambda, 0$, and 1 are in fact constant-valued, while the functions $\pi_k^{i,j}$, MSP, LSP, $Conc$, Fh, Bh, Ins_0, and Ins_1 can each be computed by wires routing inputs to outputs, with no gates at all. These are functions whose interesting dependency is entirely on the *lengths* of their inputs, which can be computed, added, subtracted, halved, and doubled by a random-access, log-time, deterministic machine. So all these functions have LOGTIME-uniform, constant-depth, bounded-fanin circuits. $Cond$, Or, and Not actually use some gates, but they too are clearly in FNC^0.

Definition by Safe Composition: Suppose $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$. Suppose also that k_g, c_g, k_u, c_u, k_v, and 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 $|u(\vec{x};)| \leq p_u(|\vec{x}|)$. Since $p_u(|\vec{x}|)$ is bounded by a monic polynomial in $\max(|\vec{x}|)$, we assume without loss of generality that it actually is one, i.e. $p_u(|\vec{x}|) = a_u \cdot \max(|\vec{x}|)^{\deg(p_u)}$.

Then

$$\begin{aligned} &d_F[\vec{x};\vec{y}] \leq \\ &\leq \quad d_G[u(\vec{x};);v(\vec{x};\vec{y})] + \max(d_U[\vec{x};], d_V[\vec{x};\vec{y}]) \\ &\leq \quad k_g \cdot ||u(\vec{x};)|| + c_g + \\ &\qquad \max(k_u \cdot \max(||\vec{x}||) + c_u, k_v \cdot \max(||\vec{x}||) + c_v) \\ &\leq \quad k_g \cdot |a_u \cdot \max(|\vec{x}|)^{\deg(p_u)}| + c_g + \\ &\qquad \max(k_u \cdot \max(||\vec{x}||) + c_u, k_v \cdot \max(||\vec{x}||) + c_v) \\ &\leq \quad k_g \cdot \deg(p_u) \cdot \max(||\vec{x}||) + |a_u| + c_g + \\ &\qquad \max(k_u, k_v) \cdot \max(||\vec{x}||) + \max(c_u, c_v) \\ &\leq \quad (k_g \cdot \deg(p_u) + \max(k_u, k_v)) \cdot \max(||\vec{x}||) + \\ &\qquad \max(c_u, c_v) + |a_u| + c_g \end{aligned}$$

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

Definition by Very Safe DCR: Suppose

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

and that k_g, c_g, and c_h satisfy 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 $|z| > |b|$,

$$\begin{aligned}
& d_f[z, b, \vec{x}; \vec{y}] \leq \\
&\leq \quad d_H[; z, \vec{x}, \vec{y}, f(Fh(z), b, \vec{x}; \vec{y}), f(Bh(z), b, \vec{x}; \vec{y})] + \\
&\qquad d_F[Bh(z), b, \vec{x}; \vec{y}] \\
&\leq \quad c_h + d_F[Bh(z), b, \vec{x}; \vec{y}] \\
&\leq \quad ||z|| \cdot c_h + d_G[z, \vec{x}; \vec{y}] \\
&\leq \quad ||z|| \cdot c_h + k_g \cdot \max(||z||, ||\vec{x}||) + c_g \\
&\leq \quad (c_h + k_g) \cdot \max(||z||, ||\vec{x}||) + c_g
\end{aligned}$$

∎

Thus each bit of $f(\vec{x}; \vec{y})$ 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 $\vec{X}$ and an $O(\log(|\vec{X}|))$ time bound, there is a function $f(\vec{X};) \in vsc(BASE)$ such that*

$$f(\vec{X};) = \begin{cases} 1 \text{ if } M \text{ accepts } \vec{X} \\ 0 \text{ if } M \text{ rejects } \vec{X} \end{cases}$$

In other words, $vsc(BASE)$ 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

$$ZEROES(; x) = LSP(; \lambda, x)$$

which returns a string of zeroes of the same length as x.

Next we define

$$PADRT(; x, y) = Conc(x, ZEROES(; MSP(y, x)))$$

which pads its argument x on the right with zeroes to length $\max(|x|, |y|)$.

We frequently wish to get at the leftmost or rightmost bit of a string.

$$\begin{aligned}
LOBIT(; x) &= LSP(; x, 1) \\
DELLOBIT(; x) &= MSP(x, 1) \\
HIBIT(; x) &= MSP(x, MSP(x, 1)) \\
DELHIBIT(; x) &= LSP(; x, MSP(x, 1))
\end{aligned}$$

We also define a function to shift a number without changing its length.

$$SHL(; x, y) = LSP(Conc(x, y), x)$$

shifts x left by appending y, discarding the high bits so the result is the same length as x.

We would also like to be able to find the length of a string.

$$Len(x;) = \begin{cases} \lambda & \text{if } |x| = 0 \\ 1 & \text{if } |x| = 1 \\ Conc(Len(Fh(x);), 1) & \text{if } |x| > 1 \end{cases}$$

returns a string y such that $|y| = ||x||$. Note that $Len(x;)$ does *not* return a binary representation of $|x|$ 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 $GETBIT(X; i)$ 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 zeroes. We also assume $|X|$ is a power of 2. The function is defined by simultaneous, very safe DCR: let $GETBIT(X; i) = f_1(X; X, LSP(i, DELLOBIT(; Len(X;))))$ where

$$\begin{aligned}
& f_1(Y; X, i) = \\
&= \begin{cases} X \text{ if } Y \leq 1 \\ Fh(f_1(Fh(Y); X, i)) \\ \text{ if } Y > 1 \text{ and } HIBIT(; f_2(Fh(Y); X, i)) = 1 \\ Bh(f_1(Fh(Y); X, i)) \\ \text{ if } Y > 1 \text{ and } HIBIT(; f_2(Fh(Y); X, i)) = 0 \end{cases} \\
& f_2(Y; X, i) = \\
&= \begin{cases} PADRT(; i, X) & \text{if } |Y| \leq 1 \\ SHL(; Fh(f_2(Fh(Y); X, i))), 1) & \text{if } |Y| > 1 \end{cases}
\end{aligned}$$

Note that $|X| = |PADRT(; i, X)|$, so the equal-length requirement is met at the base level. It is also easily proven by induction that $|f_1(Y; X, i)| = \frac{|X|}{|Y|}$ 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 $SHL(; Fh(f_2(Fh(Y); X, i)), 1)$ are all equal. Thus this definition meets all the equal-length criteria of Lemma 11, and so $GETBIT \in vsc(BASE)$.

As for its correctness, the $GETBIT$ function first coerces i to length $\log(|X|) = ||X|| - 1$, discarding high bits if i is too long and left-padding with zeroes 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 $t = d \cdot \max(||\vec{X}||)$, 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) $\langle q, IL_1, IR_1, \ldots IL_j, IR_j, WL_1, WR_1, \ldots WL_k, WR_k \rangle$ where*

q represents finite state information,

IL_i 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,

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

IR_i the contents of index tape i from the head rightwards (lowest-significance bits representing the square under the head), and

WR_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 $\vec{X}$ are not an explicit part of a configuration.

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

Define a function $INIT_M(\vec{X};)$ 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 $vsc(BASE)$: we can compute $Len(; BLOWUP(\vec{X};))$ to get something of length t, apply $ZEROES$ to make it an empty tape, then form a tuple of finitely many such things and the start state with $COMB$.

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 $Cond$ and 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 MSP and $Cond$, and tape head movement and writing can be easily done with MSP, $Conc$, and 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 $0, 1, 3, 7, \ldots 2^t - 1$, and the bits in between are ignored. (Each of 2^t paths is inflated to length 2^t, so the tree has length 2^{2t}, which is still polynomial since $t = O(||\vec{X}||)$.) 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 $i < |y|$, $Bit(i, y) = Bit(2^i - 1, x)$. 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:

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

where

$$h(;v) \quad = \quad \begin{cases} Conc(Ins_0(v), Ins_1(v)) \\ \quad \text{if } HIBIT(;v) \neq HIBIT(; Bh(v)) \\ Conc(Ins_0(v), Ins_1(Not(v))) \\ \quad \text{if } HIBIT(;v) = HIBIT(; Bh(v)) \end{cases}$$

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

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

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

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

Note that this last inequality is independent of x.

The less obvious properties of $PATHS$ are presented in the following technical lemma. The notation $\pm x$ denotes "either x or $Not(x)$".

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

1. *If $|x| > 1$, and if $j \geq \lfloor |x|/2^i \rfloor$, then $N_j^{i,x} = Ins_0(N_{j-\lfloor |x|/2^i \rfloor}^{i,Bh(x)})$ while if $j < \lfloor |x|/2^i \rfloor$, then $N_j^{i,x} = Ins_1(\pm N_{j-\lfloor |x|/2^i \rfloor}^{i,Bh(x)})$*
2. *For any node in $PATHS(x;)$ 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 $N_{j_1}^{0,x}$ and $N_{j_2}^{0,x}$ in $PATHS(x;)$:*
   (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 $||x|| - i$ bits.*
4. *If $PATHS(x;)$ is interpreted as a tree, its leaves comprise one full inflation of each path of length $||x||$.*

The proof of this lemma involves a number of straightforward but tedious inductions, and we omit it, but illustrate it by the following **Example:**
$PATHS(\lambda;) = \lambda$, no blocks of length 0
$PATHS(1;) = \hat{0}\hat{1}$, 2 blocks of length 1
$PATHS(11;) = \overbrace{00}\overbrace{10}\overbrace{01}\overbrace{11}$,
4 blocks of length 2
$PATHS(1111;) =$
$= \overbrace{0000}\overbrace{1000}\overbrace{0010}\overbrace{1010}\overbrace{1111}\overbrace{0111}\overbrace{1101}\overbrace{0101}$,
8 blocks of length 4, each of which is the full inflation of a distinct 3-bit path.
$PATHS(11111111;) =$
$= \overbrace{00000000}\overbrace{10000000}\overbrace{00001000}\overbrace{10001000}$
$\overbrace{10101010}\overbrace{00101010}\overbrace{10100010}\overbrace{00100010}$
$\overbrace{01010101}\overbrace{11010101}\overbrace{01011101}\overbrace{11011101}$
$\overbrace{11111111}\overbrace{01111111}\overbrace{11110111}\overbrace{01110111}$,
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 $||BLOWUP(\vec{X};)|| = t$, the time bound of the alternating machine, it follows that $PATHS(BLOWUP(\vec{X};);)$ 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.

$$\begin{aligned} NEXT(;P,C) &= Cond(HIBIT(;P), \\ &\qquad NEXT_L(;C), \\ &\qquad NEXT_R(;C)) \end{aligned}$$

computes $NEXT_L$ if the high bit of P is set, and otherwise $NEXT_R$. Recursion on this function yields

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

The $GETLEAF$ 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 $GETBIT$ on the "normal" input parameters $\vec{X}$). No recursion is required except that to define $GETBIT$, and we can define $EVALLEAF(\vec{X};C)$.

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

$$EVALTREE(T, s, \vec{X}; C) = \\ = \begin{cases} EVALLEAF(\vec{X}; GETLEAF(T; C)) \\ \quad \text{if } |T| \leq |s| \\ EVALTREE(Fh(T), s, \vec{X}; C) \wedge \\ \quad EVALTREE(Bh(T), s, \vec{X}; C) \\ \quad \text{if } |T| > |s| \text{ and the height of } T \text{ is odd} \\ EVALTREE(Fh(T), s, \vec{X}; C) \vee \\ \quad EVALTREE(Bh(T), s, \vec{X}; C) \\ \quad \text{if } |T| > |s| \text{ and the height of } T \text{ is even} \end{cases}$$

or, more formally,

$$EVALTREE(T, s, \vec{X}; C) = \\ = \begin{cases} EVALLEAF(\vec{X}; GETLEAF(T; C)) \\ \quad \text{if } |T| \leq |s| \\ h(; T, \\ \quad EVALTREE(Fh(T), s, \vec{X}; C), \\ \quad EVALTREE(Bh(T), s, \vec{X}; C)) \\ \quad \text{if } |T| > |s| \end{cases}$$

where

$$\begin{aligned} h(; T, u_1, u_2) \quad = \quad & Cond(; HIBIT(; T), \\ & \quad Cond(; HIBIT(; Bh(T)), \\ & \quad\quad Cond(; u_1, 1, u_2), \\ & \quad\quad Cond(; u_1, u_2, 0)), \\ & \quad Cond(; HIBIT(; Bh(T)), \\ & \quad\quad Cond(; u_1, u_2, 0), \\ & \quad\quad Cond(; u_1, 1, u_2))) \end{aligned}$$

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

$$\begin{aligned} & SIMULATE(\vec{X};) = \\ & = \; EVALTREE(PATHS(BLOWUP(\vec{X};);), \\ & \qquad\qquad Bh(BLOWUP(\vec{X};)), \vec{X}; \\ & \qquad\qquad INIT_M(\vec{X};)) \end{aligned}$$

Thus $ALOGTIME \subseteq vsc(BASE)$. This completes the proof of Lemma 19. ∎

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

$$(\forall i < |F(\vec{x};)|)(f(\vec{x}, i;) = Bit(i, F(\vec{x};)))$$

then $F(\vec{x};) \in vsc(BASE)$.

Proof of Lemma 23: We have already noted that any polynomial growth rate can be attained in $vsc(BASE)$, so we can define a function $SIZE_F$ such that $|SIZE_F(\vec{x};)| = |F(\vec{x};)|$. Now we build a binary tree, similar to $PATHS$, each of whose leaves contains a different value, but this time the values are the binary numbers $0 \ldots |SIZE_F(\vec{x};)| - 1$ in reverse order. We can then define F by very safe DCR on such a tree T as follows:

$$F(\vec{x};) = F'(T, Len(SIZE_F(\vec{x};);), \vec{x}; \vec{y})$$

$$F'(T, z, \vec{x};) = \\ = \begin{cases} f(T, \vec{x};) \text{ if } |T| \leq |z| \\ Conc(F'(Fh(T), z, \vec{x};), F'(Bh(T), z, \vec{x};)) \\ \quad \text{if } |T| > |z| \end{cases}$$

The iterated function is $Conc$, which is associative; so the effect is simply to concatenate the bits $f(|z|-1, \vec{x};)$, $f(|z|-2, \vec{x};)$, $\ldots f(1, \vec{x};)$, $f(0, \vec{x};)$, 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, $SIZE_F$ 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:

$$COUNTUP(z;) = f_2(z; Len(z;))$$

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

$$f_2(z; b) \quad = \quad \begin{cases} \lambda \quad \text{if } |z| = 0 \\ ZEROES(; b) \text{ if } |z| = 1 \\ Conc(Or(f_1(Bh(z); b), f_2(Bh(z); b)), \\ \qquad f_2(Bh(z); b)) \\ \quad \text{if } |z| > 1 \end{cases}$$

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 $\frac{|z|}{2}-1,\ldots 1,0$ in blocks of size $|b|$, then $f_2(z;b)$ lists $|z|-1,\ldots \frac{|z|}{2}$ followed by $\frac{|z|}{2}-1,\ldots 1,0$, i.e. $|z|-1,\ldots 1,0$ as desired.

Plugging the resulting $COUNTUP(z;)$ 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. $f(\vec{x};) \in sc(BASE)$
2. *$f(\vec{x})$ can be computed from $\vec{x}$ by a LOGTIME-uniform, polylog depth circuit family, and f has $2^{poly(||\vec{x}||)}$ growth rate*
3. *the i-th bit of $f(\vec{x})$ can be computed from i and $\vec{x}$ in alternating polylog time, and f has $2^{poly(||\vec{x}||)}$ growth rate*
4. *$f(\vec{x})$ 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. ($2 \iff 3$) 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 $f(\vec{x};\vec{y}) \in sc(BASE)$, then there is a polynomial p_f with nonnegative coefficients such that*

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

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 $f(\vec{x};\vec{y})$ is defined by safe composition as $g(u(\vec{x});v(\vec{x};\vec{y}))$, and if p_g, p_u, and p_v are polynomials satisfying the Lemma for functions g, u, and v respectively, then

$$\begin{aligned} |f(\vec{x};\vec{y})| &\le 2^{p_g(||u(\vec{x})||)} \cdot |v(\vec{x};\vec{y})| \\ &\le 2^{p_g(|2^{p_u(||\vec{x}||)}|)} \cdot 2^{p_v(||\vec{x}||)} \cdot \max(|\vec{y}|,1) \\ &\le 2^{p_g(p_u(||\vec{x}||)+1)} \cdot 2^{p_v(||\vec{x}||)} \cdot \max(|\vec{y}|,1) \\ &\le 2^{p_g(p_u(||\vec{x}||)+1)+p_v(||\vec{x}||)} \cdot \max(|\vec{y}|,1) \end{aligned}$$

so the statement is satisfied by letting

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

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

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

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

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

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

$$\begin{aligned} &|f(z,b,\vec{x};\vec{y})| \le \\ &\le 2^{||z||,||\vec{x}||)} \cdot \\ &\quad \max(|f(Fh(z),b,\vec{x};\vec{y})|,|f(Bh(z),b,\vec{x};\vec{y})|,|\vec{y}|) \\ &\le 2^{p_h(||z||,||\vec{x}||)} \cdot \\ &\quad \max\left(2^{\lceil \log(\lceil \frac{|z|}{2} \rceil) \rceil \cdot p_h(|\lceil \frac{|z|}{2} \rceil|,||\vec{x}||)+p_g(||b||,||\vec{x}||)}, |\vec{y}|\right) \\ &\le 2^{p_h(||z||,||\vec{x}||)} \cdot \\ &\quad \max\left(2^{(\lceil \log(|z|) \rceil - 1) \cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)}, |\vec{y}|\right) \\ &\le 2^{p_h(||z||,||\vec{x}||)} \cdot \\ &\quad 2^{(\lceil \log(|z|) \rceil - 1) \cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)} \cdot \max(|\vec{y}|) \\ &\le 2^{\lceil \log(|z|) \rceil \cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)} \cdot \max(|\vec{y}|) \end{aligned}$$

as desired. ∎

Lemma 26 *If $f(\vec{x};\vec{y}) \in sc(BASE)$, 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 $\vec{x},\vec{y}$,*

$$d_F[\vec{x};\vec{y}] \le q_f(||\vec{x}||) \cdot \max(||\vec{y}||,1)$$

Proof of Lemma 26: All the BASE functions can obviously be computed in such a fashion. For the remaining cases, we assume wolog that $d_F[\vec{a};\vec{b}]$ and the like are monotone in the lengths of their arguments.

Definition by Safe Composition: If $f(\vec{x};\vec{y})$ is defined by safe composition as $g(u(\vec{x};);v(\vec{x};\vec{y}))$, then let p_g, p_u, and p_v be polynomials whose exponentials bound the growth of g, u, and v respectively as in Lemma 25, and let q_g, q_u, and q_v be polynomials bounding the circuit depth required to compute g, u, and v 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.

$$\begin{aligned}
&d_F[\vec{x};\vec{y}] \leq \\
&\quad\leq\ d_G[u(\vec{x};);v(\vec{x};\vec{y})] + \max(d_U[\vec{x};], d_V[\vec{x};\vec{y}]) \\
&\quad\leq\ q_g(||u(\vec{x};)||)\cdot||v(\vec{x};\vec{y})|| + \\
&\quad\qquad \max(q_u(||\vec{x}||), q_v(||\vec{x}||)\cdot\max(||\vec{y}||,1)) \\
&\quad\leq\ q_g(p_u(||\vec{x}||))\cdot\max(p_v(||\vec{x}||)+\max(||\vec{y}||,1)) + \\
&\quad\qquad \max(q_u(||\vec{x}||), q_v(||\vec{x}||)\cdot\max(||\vec{y}||,1)) \\
&\quad\leq\ q_g(p_u(||\vec{x}||))\cdot p_v(||\vec{x}||)\cdot\max(||\vec{y}||,1) + \\
&\quad\qquad (q_u(||\vec{x}||)+q_v(||\vec{x}||))\cdot\max(||\vec{y}||,1) \\
&\quad=\ (q_g(p_u(||\vec{x}||))\cdot p_v(||\vec{x}||)+q_u(||\vec{x}||)+q_v(||\vec{x}||))\cdot \\
&\quad\qquad \max(||\vec{y}||,1)
\end{aligned}$$

so the statement is satisfied by letting

$$q_f(\vec{r}) = q_g(p_u(\vec{r}))\cdot p_v(\vec{r}) + q_u(\vec{r}) + q_v(\vec{r}))$$

Again, the proof in the case that there are multiple functions $\vec{u}$ and $\vec{v}$ is similar.

Definition by Safe DCR: Now suppose $f(z,b,\vec{x};\vec{y})$ is defined by safe DCR on the parameter z, to wit:

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

Let p_f, p_g, and p_h be polynomials satisfying Lemma 25 for functions f, g, and h respectively, and 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 $|z'|\leq|z|$,

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

so if $h(z,\vec{x};u_1,u_2,\vec{y})$ is called in the course of computing $f(z,b,\vec{x};\vec{y})$, we can be assured that

$$|u_1|,|u_2| \leq 2^{p_f(||z||,||b||,||\vec{x}||)}\cdot\max(|\vec{y}|,1)$$

and therefore $||u_1||$ and $||u_2||$ are each at most $p_f(||z||,||b||,||\vec{x}||)+\max(||\vec{y}||,1)$. With this in mind, then,

$$\begin{aligned}
&d_F[z,b,\vec{x};\vec{y}] \leq \\
&\quad\leq\ \max(\ d_G[z,\vec{x};\vec{y}], \\
&\qquad d_H[z,\vec{x};f(Fh(z),b,\vec{x};\vec{y}),f(Bh(z),b,\vec{x};\vec{y})] + \\
&\qquad \max(d_F[Fh(z),b,\vec{x};\vec{y}], d_F[Bh(z),b,\vec{x};\vec{y}])) \\
&\quad\leq\ \max(d_G[z,\vec{x};\vec{y}], \\
&\qquad d_H[z,\vec{x};f(Fh(z),b,\vec{x};\vec{y}),f(Bh(z),b,\vec{x};\vec{y}),\vec{y}] + \\
&\qquad d_F[Bh(z),b,\vec{x};\vec{y}]) \\
&\qquad \text{by monotonicity of } d_F \\
&\quad\leq\ d_G[z,\vec{x};\vec{y}] + \\
&\qquad \left(\sum_{i=||b||+1}^{||z||} q_h(i,||\vec{x}||)\right)\cdot \\
&\qquad \max(p_f(||z||,||b||,||\vec{x}||)+\max(||\vec{y}||,1),||\vec{y}||) \\
&\qquad \text{by the remarks above} \\
&\quad\leq\ d_G[z,\vec{x};\vec{y}] + ||z||\cdot q_h(||z||,||\vec{x}||)\cdot \\
&\qquad (p_f(||z||,||b||,||\vec{x}||)+\max(||\vec{y}||,1)) \\
&\qquad \text{by monotonicity of } q_h \\
&\quad\leq\ q_g(||z||||\vec{x}||)\cdot\max(||\vec{y}||,1) + ||z||\cdot q_h(||z||,||\vec{x}||)\cdot \\
&\qquad (p_f(||z||,||b||,||\vec{x}||)+\max(||\vec{y}||,1)) \\
&\quad\leq\ (||z||\cdot q_h(||z||,||\vec{x}||)\cdot(p_f(||z||,||b||,||\vec{x}||)+1) + \\
&\qquad q_g(||z||,||\vec{x}||))\cdot\max(||\vec{y}||,1)
\end{aligned}$$

and so it would suffice to let $q_f(||z||,||b||,||\vec{x}||)$ be

$$||z||\cdot q_h(||z||,||\vec{x}||)\cdot(p_f(||z||,||b||,||\vec{x}||)+1)+q_g(||z||,||\vec{x}||)$$

∎

Thus each bit of $f(\vec{x};\vec{y})$ can be computed in polylog depth, $2^{polylog}$ size. Since there are at most $2^{polylog}$ bits of output, the whole function value is computable by polylog-depth, $2^{polylog}$-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 $\vec{X}$ and time bound polynomial in $\log(|\vec{X}|)$, there is a function $f\in sc(BASE)$ such that*

$$f(\vec{X};) = \begin{cases} 1 \textit{ if } M \textit{ accepts } \vec{X} \\ 0 \textit{ if } M \textit{ rejects } \vec{X} \end{cases}$$

In other words, $sc(BASE)$ 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 $COPY$ 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 $t = p_M(||\vec{X}||)$, we can redefine $BLOWUP(\vec{X};)$ to have length $2^{p_M(||\vec{X}||)} - 1$, and hence $||BLOWUP(\vec{X};)|| \geq p_M(||\vec{X}||)$, by composing $\deg(p_M)$ invocations of the $\#_3$ function as well as suitable numbers of $COPY$ and $Conc$ functions.

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

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

Proved exactly as Lemma 23, with larger growth rates.

5 Conclusions and Observations

The most unfamiliar functions in BASE are, of course, Ins_0 and Ins_1, although they are in 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 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 NC, circuits of simultaneous polylog depth and polynomial size. The reason Allen's system describes 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 NC, the latter using the theory D^1, which is equivalent to the bounded arithmetic theory R^1_2. Buss, Krajíček, and Takeuti [5], by replacing R^1_2 with R^1_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, NC, 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 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

[1] William Allen. *Divide and Conquer as a Foundation of Arithmetic.* PhD thesis, University of Hawaii at Manoa, 1988.

[2] Stephen Bellantoni. A new recursion-theoretic characterization of the functional polytime hierarchy. Manuscript, 1992.

[3] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *computational complexity*, 2:97–110, Dec 1992.

[4] Samuel R. Buss. *Bounded Arithmetic.* PhD thesis, Princeton University, 1986.

[5] Samuel R. Buss, Jan Krajíček, and Gaisi Takeuti. Provably total functions in bounded arithmetic theories R_3^i, U_2^i and V_2^i. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 116–161. Oxford University Press, 1993.

[6] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[7] Peter Clote. Sequential, machine-independent characterizations of the parallel complexity classes $ALOGTIME$, AC^k, NC^k and NC. In Samuel R. Buss and P. Scott, editors, *Proc. Workshop on Feasible Math.*, pages 49–69. Birkhäuser, 1989.

[8] Peter Clote. Polynomial size Frege proofs of certain combinatorial principles. In Peter Clote and Jan Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 162–184. Oxford University Press, 1993.

[9] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science II*, pages 24–30. North-Holland, 1965.

[10] Kevin Compton and Claude LaFlamme. An algebra and a logic for NC^1. *Information and Computation*, 87:241–263, 1990.

[11] J. Lind. Computing in logarithmic space. Technical Report 52, Project MAC, Massachusetts Inst. of Technology, September 1974.

[12] W. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.