FUNCTION-ALGEBRAIC CHARACTERIZATIONS OF LOG AND POLYLOG PARALLEL TIME

Stephen Bloch

Abstract. The main results of this paper are recursion-theoretic characterizations of two parallel complexity classes: the functions computable by uniform bounded fan-in circuit families of log and polylog depth (or equivalently, the functions bitwise computable by alternating Turing machines in log and polylog time). The present characterizations avoid the complex base functions, function constructors, and *a priori* size or depth bounds typical of previous work on these classes. This simplicity is achieved by extending the "tiered recursion" techniques of Leivant and Bellantoni & Cook.

Key words. Circuit complexity; subrecursion.
Subject classifications. 68Q15, 03D20, 94C99.

1. Introduction

Researchers in computational complexity have tried for decades to describe computational classes without reference to Turing machines or other specific models of computation. One early success, Cobham (1965), characterized the functions computable in polynomial time as the algebra generated from a few base functions by closure under composition and a form of recursion. Cobham's characterization was substantially simplified with new techniques in Bellantoni & Cook (1992). By extending these techniques, I characterize two *parallel* function classes: the functions computable in parallel time proportional to the log, or to a polynomial in the log, of the input length. Here "parallel time" refers to either the alternating Turing machine or the uniform, bounded fan-in circuit family model of computation.

Cobham's characterization inspired many other recursion-theoretic and logical characterizations of complexity classes. The Cobham characterization provided early support for $\mathcal{FP}$ as a natural class, but this support was weakened by the explicit bounds and unfamiliar base functions in the characterization. Such problems—artificial size or depth bounds, base functions introduced only for their growth rates, computationally complex base functions, and untidy collections of composition and recursion schemes—continued to plague later researchers. Lind (1974) gave a recursion-theoretic characterization of logspace, and Allen (1991) gave logical and recursion-theoretic characterizations of NC, both explicitly bounding function growth rates by polynomials. Compton & LaFlamme (1990) presented recursion-theoretic and logical characterizations of uniform NC^1, relying on a polynomial size bound implicit in their finite-models approach. Clote (1989) characterized uniform NC^1, AC^0, and a variety of other parallel classes with a polynomial-bounded recursion scheme, using a complete problem from Buss (1987). A more elegant characterization, in Clote (1993), used the completeness result of Barrington (1989) to simplify the base functions and impose a *constant* bound on recursively-defined functions. Buss (1986), using a variant of Cobham's characterization, showed that the functions at each level of the polynomial-time hierarchy are precisely those provably total in a corresponding weak fragment of Peano arithmetic. Buss's and Clote's characterizations, however, required a language expressly designed to produce polynomial growth rates. Arai (1992) characterized uniform NC^1 with systems of bounded arithmetic based on the "upward tree recursion" of Compton & LaFlamme (1990), while Clote & Takeuti (1992) did the same with a log-depth weak induction scheme; both of these used only the usual language of number theory but explicitly specified $O(\log(n))$ recursion depth (and hence $2^{O(\log(n))} = n^{O(1)}$ size).

By contrast, the results of this paper involve no explicit size or depth bounds, no functions introduced solely to produce desired growth rates, only computationally easy base functions—so easy as to be computable by *constant*-depth, bounded fan-in circuits—and only one composition and one recursion scheme. Essential to these simpler characterizations is the notion (introduced in Leivant 1990 and Bellantoni & Cook 1992) of classifying the parameters of a function into "tiers" by how they are used in the computation. The classification depends on whether a parameter is used only for local, bitwise operations or is used in its entirety to bound a loop or a recursion. The former sort of parameter has much less impact on the resources necessary to compute the function, and is called "safe," or "tier 0;" the latter is called "normal," or "tier 1." (The "safe/normal" terminology is due to Bellantoni & Cook 1992. The "tier" terminology, due to Leivant 1990, generalizes to higher-numbered tiers. In this paper we shall only be concerned with tiers 0 and 1, so we use the two terminologies interchangeably.) By distinguishing tiers of parameters syntactically, one can bound computational resource costs without unduly limiting the system's ability to perform simple operations. This technique allowed Bellantoni & Cook (1992) to characterize polynomial time with a function algebra that made no mention of polynomial growth rates.

The present paper applies the tiering technique for the first time to parallel complexity classes. I replace the scheme of recursion on notation used in Cobham (1965) and Bellantoni & Cook (1992) with a scheme of "divide and conquer recursion." Divide and conquer recursion, without tiering, has been used under various names in Buss (1986), Compton & LaFlamme (1990), and Allen (1991), always with size and/or depth bounds artificially imposed somehow. With tier discipline, however, the desired size and depth bounds follow naturally from the recursion scheme itself.

The main results of this paper are summed up in the following two theorems, whose terminology will be defined later:

THEOREM 1. *For functions* $f : (\{0,1\}^*)^k \mapsto \{0,1\}^*$, *the following are equivalent:*

1. $f(\vec{x};) \in \mathrm{vsc}(\mathrm{BASE})$ *with a length mask;*
2. f *is bitwise computable in alternating log time;*
3. f *is computable with a length mask by a* U_{E^*}*-uniform family of multi-output* NC^1 *circuits.*

THEOREM 2. *For functions* $f : (\{0,1\}^*)^k \mapsto \{0,1\}^*$, *the following are equivalent:*

1. $f(\vec{x};) \in \mathrm{sc}(\mathrm{BASE})$ *with a length mask;*
2. f *is bitwise computable in alternating polylog time;*
3. f *is computable with a length mask by a* U_{E^*}*-uniform family of multi-output, bounded fan-in, polylog depth circuits;*
4. f *is computable in deterministic polylog space, not counting a write-only output tape.*

The "length masks" in these theorems address the sticky question of computing the precise length of a function value. The following special cases, in which function lengths are smooth (*e.g.*, not merely *bounded* by but *equal* to a polynomial), are easier to use and suffice for most practical purposes.

COROLLARY 3. *If there is a polynomial* p *with nonnegative integer coefficients such that for all* $\vec{x}$, $|f(\vec{x};)| = p(|\vec{x}|)$, *then the following are equivalent:*

1. $f(\vec{x};) \in \mathrm{vsc}(\mathrm{BASE})$;
2. f *is bitwise computable in alternating log time;*
3. f *is computable by a* U_{E^*}*-uniform, multi-output,* NC^1 *circuit family.*

COROLLARY 4. *If there is a polynomial* p *with nonnegative integer coefficients such that for all* $\vec{x}$, $|f(\vec{x};)| = 2^{p(||\vec{x}||)}$, *then the following are equivalent:*

1. $f(\vec{x};) \in \mathrm{sc}(\mathrm{BASE})$;
2. f *is bitwise computable in alternating polylog time;*
3. f *is computable by a* U_{E^*}*-uniform, multi-output, bounded fan-in, polylog depth circuit family;*
4. f *is computable in deterministic polylog space, not counting a write-only output tape.*

2. Outline

In Section 3, I define three operations on functions: "safe composition," "safe divide-and-conquer recursion," and "very safe divide-and-conquer recursion;" the last two differ only in the strictness of their tier discipline. I then define a small collection of computationally easy BASE functions. Closing these BASE functions under safe composition and safe divide-and-conquer recursion yields a function algebra sc(BASE); closing under safe composition and *very* safe divide-and-conquer recursion yields a smaller algebra vsc(BASE).

The rather technical Section 4 discusses circuit families and uniformity conditions. While the main theorems characterize log-depth and polylog-depth circuit computation, both of which are well understood, the less common notion of constant-depth (NC^0) circuits seems essential to the proofs. I define this notion, along with suitable uniformity conditions, in this section.

In Section 5, I prove that functions in vsc(BASE) have polynomial growth, and those in sc(BASE) have $2^{polylog}$ growth. I then show that functions in vsc(BASE) are computable in uniform log depth, and those in sc(BASE) are computable in uniform polylog depth. This proves implications ($1 \Rightarrow 3$) of Theorems 1 and 2. Equivalences ($3 \Leftrightarrow 2$) of both theorems follow from Ruzzo (1981), while ($2 \Leftrightarrow 4$) of Theorem 2 follows from Chandra *et al.* (1981).

In Section 6, I prove the implications ($2 \Rightarrow 1$) of both main theorems. This is done by arithmetizing the computation of an alternating Turing machine and showing both that vsc(BASE) can correctly generate and evaluate the computation tree of a log-time alternating machine, and that sc(BASE) can generate and evaluate the computation tree of a polylog-time alternating machine. Thus any function whose bit-graph is in log (respectively polylog) alternating time is bitwise computable by a function in vsc(BASE) (respectively sc(BASE)). By a comprehension lemma, such bitwise computability extends to computability of the whole function within vsc(BASE) (respectively sc(BASE)) so long as the function's growth rate is sufficiently slow and easily computed. Corollaries 3 and 4 follow because the alternating and space-bounded machines cannot compute more than (quasi)polynomially long function values: if they did, by a simple pumping lemma, they would compute *infinitely* long values.

Finally, Section 7 compares these results with others in the field and discusses possible extensions of the technique.

3. Definitions

3.1. Notation. Most functions discussed in this paper operate on bit strings, finite strings over $\{0,1\}$. The length of a string $x \in \{0,1\}^*$ is written $|x|$. The symbol λ stands for the empty string. If $\vec{x}$ is a tuple $x_1, \ldots, x_k$, the notation $|\vec{x}|$ means the tuple $|x_1|, \ldots, |x_k|$. The notation $a \dot{-} b$, where a and b are natural numbers (*e.g.*, lengths of strings), means $\max(a-b, 0)$. The notation $\mathrm{Bit}(i, y)$ denotes the bit i places from the right end of the bit-string y, or 0 if $i > |y|$, treating the bit-string i as a binary representation of a number. Similarly, when we speak of a machine or a circuit computing on natural numbers, we mean binary representations of those numbers unless stated otherwise.

The functions discussed in this paper are *tiered functions*, functions that distinguish between normal and safe formal parameters. Following Leivant (1990) and Bellantoni & Cook (1992), we write the normal parameters first, separated from the safe parameters with a semicolon. For example, in the function $f(x, z; b, y)$, x and z are normal, b and y safe. A function $f(x_1, \ldots, x_j; y_1, \ldots, y_k)$ represents a map from $(\{0,1\}^*)^{j+k}$ to $\{0,1\}^*$. A function name is upper-case if the function is defined within the systems sc(BASE) or vsc(BASE) from simpler functions; primitive functions have mixed-case names.

A function has polynomial growth if its length is bounded by a polynomial $p(|\vec{x}|)$ in the lengths of its inputs. A function has quasipolynomial growth (see Barrington 1992) if its length is bounded by a function of the form $2^{p(||\vec{x}||)}$, for polynomial p, or equivalently if the length of its length is bounded by a polynomial $p(||\vec{x}||)$ in the lengths of the lengths of its inputs.

3.2. The Function Algebras.

Definition 5. *The set* BASE *comprises constants* $\lambda, 0, 1$, *projection functions* $\pi_k^{i,j}(x_1, \ldots, x_i; x_{i+1}, \ldots, x_{i+j}) = x_k$, *and the following functions:*

$$
\begin{array}{rcl}
\mathrm{Msp}(;x,y) & = & \text{the leftmost } |x| \dot{-} |y| \text{ bits of } x \text{ (“most significant part”)} \\
\mathrm{Lsp}(;x,y) & = & \text{the rightmost } |y| \text{ bits of } x\text{, with } |y| \dot{-} |x| \text{ leading zeroes} \\
 & & \text{(“least significant part”)} \\
\mathrm{Cond}(;b,u,v) & = & \begin{cases} \mathrm{Lsp}(;u,\max(u,v)) \text{ if } b\text{'s rightmost bit is 1} \\ \mathrm{Lsp}(;v,\max(u,v)) \text{ if } b\text{'s rightmost bit is 0, or } b=\lambda \end{cases} \\
\mathrm{Conc}(;x,y) & = & \text{the concatenation of } x \text{ and } y \\
\mathrm{Bh}(;x) & = & \text{the rightmost } \lceil |x|/2 \rceil \text{ bits of } x \text{ (“back half”)} \\
\mathrm{Fh}(;x) & = & \text{the leftmost } \lfloor |x|/2 \rfloor \text{ bits of } x \text{ (“front half”)} \\
\mathrm{Not}(;x) & = & \text{the one's complement of } x \\
\mathrm{Or}(;x,y) & = & \text{the bitwise OR of } x \text{ and } y\text{, first left-extending the} \\
 & & \text{shorter with zeroes to the same length as the longer} \\
\mathrm{And}(;x,y) & = & \text{the bitwise AND of } x \text{ and } y\text{, similarly} \\
\mathrm{Ins}_0(;x) & = & x \text{ with a 0 inserted after each bit, e.g.,} \\
 & & \mathrm{Ins}_0(;01101) = 0010100010 \\
\mathrm{Ins}_1(;x) & = & x \text{ with a 1 inserted after each bit, similarly}
\end{array}
$$

Definition 6. (Bellantoni & Cook) *We say a 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 the safe parameters $\vec{y}$ to f cannot be used as normal, nor can they in any way affect a value which is used as a normal parameter to g. On the other hand, the normal parameters $\vec{x}$ to f can affect the values of the v's, which g treats as safe.

Allen (1991) introduced a scheme of “branching recursion.” I prefer the more descriptive name “divide and conquer recursion,” or DCR for short. By adding tier discipline to this scheme we get the following definition.

Definition 7. *Function* $f(z,b,\vec{x};\vec{y})$ *is defined by* safe divide-and-conquer recursion *(henceforth* safe DCR*) from* $g(z,\vec{x};\vec{y})$ *and* $h(z,\vec{x};\vec{y},u_1,u_2)$ *if*

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

Definition 8. *Function* $f(z, b, \vec{x}; \vec{y})$ *is defined by* very safe divide-and-conquer recursion *(henceforth* very safe DCR*) from* $g(z, \vec{x}; \vec{y})$ *and* $h(; z, \vec{x}, \vec{y}, u_1, u_2)$ *if*

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

Note that safe and very safe DCR differ only in the tiers of the parameters to the iterated function h: in very safe DCR the iterated function takes no normal parameters at all, and therefore cannot itself be defined by recursion (since both recursion schemes require at least one normal parameter z).

Definition 9. *The* safe closure of BASE, *or* sc(BASE)*, is the set of functions definable from* BASE *functions by finitely many applications of safe composition and safe* DCR.

Definition 10. *The* very safe closure of BASE*, or* vsc(BASE)*, is the set of functions definable from* BASE *functions by finitely many applications of safe composition and very safe* DCR.

3.3. Simultaneous Recursion. Many recursion schemes explicitly allow defining a *tuple* of functions simultaneously based on a tuple of their previous values.

Definition 11. *The functions* $f_1(z, b, \vec{x}; \vec{y}), \ldots, f_k(z, b, \vec{x}; \vec{y})$ *are defined by* simultaneous, safe divide-and-conquer recursion *from the functions* $g_1, \ldots, g_k$ *and* $h_1, \ldots, h_k$ *if, for all* $1 \leq i \leq k$*,*

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

Simultaneous, very safe divide-and-conquer recursion is defined analogously.

Simultaneous (very) safe DCR is in fact no more powerful than ordinary (very) safe DCR, in the presence of a recursion-free way to encode and decode k-tuples. With encoding and decoding, we can define a single function H which extracts elements of a tuple, applies the appropriate h functions, and combines the results back into a tuple; (very) safe DCR on this H function produces a function

$$F(z, b, \vec{x}; \vec{y}) = \langle f_1(z, b, \vec{x}; \vec{y}), \ldots, f_k(z, b, \vec{x}; \vec{y}) \rangle,$$

from which one can extract any of the functions f_i.

Such encoding and decoding seem difficult in vsc(BASE) and sc(BASE). But they are easy in the restricted case that all the elements of the tuple are the same length, and this restricted case suffices for our purposes. We define a family of encoding and decoding functions as follows.

$$
\begin{array}{lcl}
\mathrm{COMB}_1(;x) & = & x \\
\mathrm{COMB}_{2i}(;x_1,\ldots,x_{2i}) & = & \\
\quad \mathrm{Conc}(;\mathrm{COMB}_i(;x_1,\ldots,x_i),\mathrm{COMB}_i(;x_{i+1},\ldots,x_{2i})) \quad \text{for } i>0 & & \\
\mathrm{COMB}_{2i-1}(;x_1,\ldots,x_{2i-1}) & = & \\
\quad \mathrm{Conc}(;\mathrm{COMB}_i(;x_1,\ldots,x_i),\mathrm{COMB}_i(;x_{i+1},\ldots,x_{2i-1},x_{2i-1})) \quad \text{for } i>1 & & \\
\mathrm{EXTR}_1^1(;y) & = & y \\
\mathrm{EXTR}_k^i(;y) & = & \begin{cases} \mathrm{EXTR}^i_{\lceil k/2\rceil}(;\mathrm{Fh}(y)) & \text{if } i \le \lceil k/2\rceil \\ \\ \mathrm{EXTR}^{i-\lceil k/2\rceil}_{\lceil k/2\rceil}(;\mathrm{Bh}(y)) & \text{if } \lceil k/2\rceil < i \le k \end{cases}
\end{array}
$$

We leave the reader to verify that for all tuples $\vec{a}$ such that $|a_1| = \cdots = |a_k|$ and for all $1 \le i \le k$, we have $\mathrm{EXTR}_k^i(;\mathrm{COMB}_k(;a_1,\ldots,a_k)) = a_i$.

LEMMA 12. *If $f_1,\ldots,f_k$ are defined by simultaneous, very safe* DCR *from functions in* vsc(BASE)*, and if, for all $z,b,\vec{x},\vec{y}$, $|f_1(z,b,\vec{x};\vec{y})| = \cdots = |f_k(z,b,\vec{x};\vec{y})|$, then $f_1,\ldots,f_k \in$* vsc(BASE).

The proof of this lemma is straightforward, relying on the COMB and EXTR functions taking no normal parameters. With this lemma in hand (and the analogous result with "very safe" replaced by "safe" and vsc(BASE) replaced by sc(BASE)), we may use simultaneous DCR freely whenever all the functions $f_1,\ldots,f_k$ are guaranteed to have the same length.

3.4. Alternating Turing machines. Many of the results herein involve computation by alternating Turing machines, or ATM's, defined in Chandra *et al.* (1981). I omit the complete definition, but point out the conventions particular to our setting. Our machines will have l read-only input tapes, k work tapes, and no output tape (they "output" only an accept/reject decision).

DEFINITION 13. *An ATM configuration is a tuple* $\langle q, L_1, \ldots, L_k, R_1, \ldots, R_k \rangle$ *where*

- q *is the current state, and*
- $L_1, \ldots, L_k, R_1, \ldots, R_k \in \{0,1\}^*$ *represent the contents to the left and right of the head on each of* k *work tapes.*

The symbol under head i *is bit 0 of* L_i*, the symbol to the left is bit 1 of* L_i*, and so on. The symbol to the right of head* i *is bit 0 of* R_i*, the symbol to the right of that is bit 1 of* R_i*, and so on* (i.e., *the* R*'s are stored reversed).*

Note that the input tapes are not an explicit part of a configuration.
We make certain additional assumptions about our ATM's.

- Each ATM has a time bound t, a function of its input lengths, such that the result of the computation tree on input $\vec{X}$ is determined by the first $t(|\vec{X}|)$ time steps.
- For all input lengths, t is at least the log of the number of states.
- Each R_i and each L_i is initially a t-bit string of zeroes.
- Access to the input tape occurs only at the leaves of the computation tree, at "input configurations." Each input configuration is characterized by three numbers $1 \leq i \leq k$, $1 \leq j \leq l$, and $b \in \{0,1\}$, which are encoded in the state. The input configuration accepts iff the symbol b appears on input tape j in the position whose binary representation is L_i.
- Each non-input configuration has exactly two successor configurations.
- The non-input configurations on each path alternate strictly between universal and existential configurations, starting with an existential.

These assumptions increase the running time of the ATM by only a constant factor, as one can see by standard techniques.

The class of relations decidable by an ATM in $O(\log(\text{max input length}))$ time is called ALOGTIME; the class decidable in $\log^{O(1)}(\text{max input length})$ time is called APLOGTIME.

ATM's as defined above technically compute only relations, not functions. However, it is common in the literature to construct ATM's that compute the bit-graph of a function, and by a slight abuse of terminology say that the ATM computes the corresponding function.

DEFINITION 14. *The function* $f(\vec{x}) : (\{0,1\}^*)^k \mapsto \{0,1\}^*$ *is* bitwise computable *in ALOGTIME (resp. APLOGTIME) iff* both *of the following languages are recognizable in ALOGTIME (resp. APLOGTIME):*

- $\{\langle \vec{x}, i\rangle \ : \ i < |f(\vec{x})|\}$
- $\{\langle \vec{x}, i, b\rangle \ : \ \mathrm{Bit}(i, f(\vec{x})) = b\}$

3.5. Function Length. All the BASE functions have the property that the length of the output is a function of the lengths of the inputs. This is necessary for our model of circuit computation to make sense. However, one may reasonably want to compute functions from $(\{0,1\}^*)^k$ to $\{0,1\}^*$ *without* this property. For such applications we introduce the notion of length masks.

DEFINITION 15. *Given a function class* C, *the function* $f(\vec{x};\vec{y})$ *is* in C with a length mask *if there are two functions* $B_f(\vec{x};\vec{y})$ *(the* bits *of* f*) and* $L_f(\vec{x};\vec{y})$ *(the* length mask *of* f*), both in* C*, such that*

- *For all* $\vec{x}, \vec{y}$, *we have* $L_f(\vec{x};\vec{y}) \in 0^*1^*$, *and*
- *For all* $\vec{x}, \vec{y}$, *the value of* $f(\vec{x};\vec{y})$ *is the value of* $B_f(\vec{x};\vec{y})$ *restricted to the bit positions in which* $L_f(\vec{x};\vec{y})$ *has a 1.*

For example, if C is the class of functions $f(\vec{x})$ such that $\mathrm{Bit}(i, f(\vec{x}))$ is an ALOGTIME or APLOGTIME relation of i and $\vec{x}$, this definition reduces to Definition 14. It differs mainly in being based on a function class, rather than a relation class.

4. Circuits and Uniformity

In this section we define a model of circuit computation and discuss the uniformity conditions necessary to prove the main theorems.

4.1. $\mathbf{NC^0}$ circuits. The smallest complexity class generally studied in circuit complexity is AC^0, the class of relations computable by circuit families of constant depth, using polynomial-fan-in AND and OR gates. If the gates are restricted to *constant* fan-in, the resulting class is called NC^0. When considered as a relation class, NC^0 is trivial because the single bit of output can only depend on constantly many bits of input. But in a *multi-output* circuit model, NC^0 circuits can compute many useful functions, including all the BASE functions.

DEFINITION 16. *A function* $f(x_1, \ldots, x_j; y_1, \ldots, y_k)$ *is in* FNC^0 *if it is computed by a constant depth, linear size, multiple output, bounded fan-in circuit family.*

(For convenience, suppose the gate set includes the constants 0 and 1, the single-input identity and NOT functions, and two-input AND, OR, left and right projections. Any complete basis would suffice, at the cost of longer proofs).

FACT 17. *All the* BASE *functions are in* FNC^0.

We prove this Fact, and elaborate on it, in Lemma 24.

4.2. Uniformity. As usual in circuit complexity theory, uniformity becomes a more crucial and delicate consideration as the circuits themselves become weaker. The most widely accepted notion of uniformity for log- and higher-depth circuits is the following, introduced in Ruzzo (1981):

DEFINITION 18. (RUZZO) *The* extended connection language *of a circuit family of size* $Z(n)$ *is the set of 4-tuples* $\langle n, \gamma, p, \tau\rangle$ *such that* n *is the binary representation of the number of inputs to the circuit,* γ *is a binary number identifying a gate in the circuit,* $p \in \{L, R\}^*$, *and either* $p = \epsilon$ *and* τ *is the type of gate* γ *in some fixed encoding (the type of an internal gate is the Boolean function it computes, while the type of an input gate is a binary number indicating which bit of input it is), or* p *represents a nontrivial path of at most* $\log(Z(n))$ *left/right decisions and* τ *is a binary number identifying the descendant of* γ *reached by following that path.*

DEFINITION 19. (RUZZO) *A family of bounded fan-in circuits of size* $Z(n)$ *and depth* $t(n)$ *is* U_{E^*}*-uniform if its extended connection language is recognizable by an ATM in time* $O(t(n))$ *and space* $O(\log(Z(n)))$.

Unfortunately, this definition makes no sense for constant-depth circuits, as constant time is insufficient even to examine a gate number or an input length. So we separate the uniformity computation into two phases, one taking constant time based on partial information about a gate, and a second combining the first phase with the rest of the information about the circuit. Composing or iterating functions will correspond to composing or iterating the first phase; the second phase is only applied once, and its greater cost amortized.

Before describing these phases, though, we adopt a simple system of numbering the gates in a circuit. First, all circuits henceforth will be treelike, so that each gate γ descends by a unique path $\sigma_\gamma \in \{L, R\}^*$ from a unique output gate OUT_γ. The output bit position OUT_γ and the path σ_γ jointly determine γ. We accordingly identify the "gate number" γ in Ruzzo's definition with the pair $(\mathrm{OUT}_\gamma, \sigma_\gamma)$.

Under this gate numbering scheme, determining the gate descended from γ along a path p is easy (simply concatenate p to σ_γ, in deterministic time $O(|p| + |\sigma_\gamma|)$). Determining the type of an internal gate is also reasonably easy; I come back to this later. The difficult part, which calls for the two-phase approach, is determining which input bit is tied to a given input gate.

In the first phase, a (deterministic, multi-tape) Turing machine translates a path σ into an open term $t_\sigma(x)$, taking time proportional to $|\sigma|$, which is bounded by circuit depth. The term is constructed from the following language, which may be thought of as mapping an output bit position to an input bit position that "affects" it (*i.e.*, there are two inputs, differing only in the specified bit position, that produce different output). In addition to the natural numbers that indicate bit positions, we use a symbol $\perp$ to indicate that *no* input bit affects the given output bit.

For a circuit with k inputs $y_1, \ldots, y_k \in \{0,1\}^*$, the *mapping language* contains the following $3k + 3$ function symbols (the reasons for this choice will become clear in Lemma 24):

$$
\begin{aligned}
\mathrm{Undef}(x) &= \perp, \text{ regardless of } x \\
\mathrm{Zero}(x) &= \begin{cases} \perp & \text{if } x = \perp \\ 0 & \text{otherwise} \end{cases} \\
\mathrm{Halve}(x) &= \begin{cases} \perp & \text{if } x = \perp \\ \left\lfloor \frac{x}{2} \right\rfloor & \text{otherwise} \end{cases} \\
\mathrm{Add}_j(x) &= \begin{cases} \perp & \text{if } x = \perp \\ x + |y_j| & \text{otherwise, for } 1 \le j \le k \end{cases} \\
\mathrm{Sub}_j(x) &= \begin{cases} \perp & \text{if } x = \perp \\ x \dot{-} |y_j| & \text{otherwise, for } 1 \le j \le k \end{cases} \\
\mathrm{AddHalf}_j(x) &= \begin{cases} \perp & \text{if } x = \perp \\ x + \left\lceil \frac{|y_j|}{2} \right\rceil & \text{otherwise, for } 1 \le j \le k \end{cases}
\end{aligned}
$$

In the second phase, an ATM evaluates this term by plugging in the lengths $|\vec{y}|$ and substituting OUT_γ for x, taking time $O(\log(\max(|t_\sigma|, |y_j|, OUT_\gamma)))$.

We are now ready to formally define our uniformity criterion.

DEFINITION 20. *A circuit family is* mapping-uniform *if there is a deterministic, multi-tape Turing machine P and an ATM Q such that for any gate* $(\mathrm{OUT}_\gamma, \sigma_\gamma)$ *in a circuit of the family,*

- *if* $(\mathrm{OUT}_\gamma, \sigma_\gamma)$ *is an input gate tied to bit r of some parameter, then machine P on input σ_γ runs in time $O(|\sigma_\gamma|)$ and outputs a term t_{σ_γ} in the mapping language such that $t_{\sigma_\gamma}(\mathrm{OUT}_\gamma) = r$, and*
- *machine Q on input* $\mathrm{OUT}_\gamma, \sigma_\gamma, |y_1|, \ldots, |y_k|, \tau$ *runs in alternating time* $O(\log(\max(\mathrm{OUT}_\gamma, |\sigma_\gamma|, |y_1|, \ldots, |y_k|)))$ *and accepts iff either* $(\mathrm{OUT}_\gamma, \sigma_\gamma)$ *is an internal gate with type τ, or* $(\mathrm{OUT}_\gamma, \sigma_\gamma)$ *is an input gate tied to bit r of y_j and $\tau = \langle j, r\rangle$ (in some standard encoding).*

LEMMA 21. *Any mapping-uniform circuit family of* $O(\log)$ *or polylog depth is also U_{E^*}-uniform.*

PROOF. Straightforward. □

LEMMA 22. *If f is defined by safe composition from g, $\vec{u}$, and $\vec{v}$, all of which have mapping-uniform circuit families, then f has a mapping-uniform circuit family of depth at most proportional to the maximum depth of the families for g, $\vec{u}$, and $\vec{v}$.*

PROOF. Suppose, for simplicity, that there are only one u and one v function, so $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$. The natural circuit to compute f would be a g circuit with its inputs tied to the outputs of u and v circuits. Unfortunately, machine P would have insufficient time to decide whether a given path went into a u or a v circuit. So we attach each input of g to the output of a projection gate whose left input comes from a u circuit and whose right input comes from a v circuit. Now machine P can tell from σ_γ alone whether gate γ is in a g, a u, or a v subcircuit. Machine Q bears responsibility for deciding whether the projection gate is a left or a right projection.

Any path σ through the resulting circuit for f can be written as an initial portion σ^g passing through the g subcircuit, an additional branch through the projection gate, and a remaining portion σ^u or σ^v, depending on which path was taken through the projection gate. The term t_σ is simply the composition of t_{σ^g} and either t_{σ^u} or t_{σ^v}; if the terms for all the subcircuits could be produced in linear time (in the lengths of paths), so can the term for the f circuit.

Determining the descendant of a given gate along a given path p, as before, is easy: concatenate p to the path portion of the gate number. To determine the type of a gate in the g subcircuit, simply invoke the uniformity of g. To determine the type of a gate anywhere else in the circuit, note that each such gate descends from a unique input gate to g, so we can use the uniformity of g on σ^g to decide which input it is. This tells us whether the gate is in a u or a v subcircuit, and from which output of the u or v subcircuit our gate descends. Then by the uniformity of u and v we can determine the type of gate γ. $\square$

4.3. Properties of the Mapping Language. The mapping language defined in Section 4.2 has two important properties: terms in it can be efficiently evaluated, and it encapsulates all the possible input-output dependencies for functions in our algebras.

LEMMA 23. *Let* $m = \max(\mathrm{OUT}_\gamma, |y_1|, \ldots, |y_k|)$. *There is an ATM which recognizes the set*

$$\{\langle |y_1|, \ldots, |y_k|, \mathrm{OUT}_\gamma, t_{\sigma_\gamma}, r\rangle \ : \ r = t_{\sigma_\gamma}(\mathrm{OUT}_\gamma)\}$$

in time $O(\log(\max(|t_{\sigma_\gamma}|, m)))$.

For all the uses we shall make of this lemma, time $O(\max(|t_{\sigma_\gamma}|, m))$ would suffice. This gap suggests we could get rid of the alternation, but I have found no deterministic algorithm taking time less than $\theta(\max(|t_{\sigma_\gamma}|, m)^2)$.

PROOF. Suppose the term t_{σ_γ} does not contain the symbols Undef, Zero, or Halve. Evaluating $t_{\sigma_\gamma}(\mathrm{OUT}_\gamma)$ simply requires adding and subtracting a list of binary numbers from among OUT_γ, $|y_1|, \ldots, |y_k|$, and $\lceil |y_1|/2\rceil, \ldots, \lceil |y_k|/2\rceil$. Each symbol in the term corresponds directly to one element of this list, so the list can be bitwise constructed in ALOGTIME. Each of these numbers is at most $\log(m)$ bits long, and there are at most $|t_{\sigma_\gamma}|$ of them, so they can be added (and subtracted, using two's-complement representation) by the carry-save addition algorithm in time $O(\log(\max(m, |t_{\sigma_\gamma}|))$.

Now allow Halve symbols, but no Undef or Zero symbols, in the term. Each non-Halve symbol in the term corresponds to an element of the list. For each such symbol in parallel, count how many Halve symbols appear after it in the term and multiply the corresponding element of the list by two to that power. Add up the list as before, then divide the result by two to the power of the total number of $HALVE$ symbols in the list, rounding down. All this multiplication and division is implemented by bit-shifts, *i.e.*, addition and subtraction of bit positions, which takes $O(\log(m))$ alternating time.

To evaluate an arbitrary term in the language, guess where an Undef symbol appears and confirm it. If so, the answer is $\perp$. If not, for each entry in the list, guess and confirm the position of a Zero before it, and if there is one, replace the entry with 0 before applying the above algorithm. $\Box$

For example, evaluating the term

$$\mathrm{Add}_2(\mathrm{Halve}(\mathrm{Sub}_3(\mathrm{Halve}(\mathrm{AddHalf}_4(\mathrm{OUT}_\gamma)))))$$

corresponds to adding up the list $\left(4|y_2|, -2|y_3|, \left\lceil \frac{|y_4|}{2} \right\rceil, \mathrm{OUT}_\gamma\right)$ and dividing by four, while evaluating the term

$$\mathrm{Halve}(\mathrm{Add}_1(\mathrm{AddHalf}_2(\mathrm{Halve}(\mathrm{Zero}(\mathrm{Add}_3(\mathrm{OUT}_\gamma))))))$$

corresponds to adding up the list $\left(2|y_1|, 2\left\lceil \frac{|y_2|}{2} \right\rceil, 0, 0\right)$ and dividing by four.

LEMMA 24. *All the* BASE *functions have mapping-uniform FNC^0 circuits.*

PROOF.

- For the projection functions $\pi_k^{i,j}$, term $t_{\sigma_\gamma}(x)$ is just x (that is, output bit OUT_γ depends only on bit OUT_γ of some parameter). Let machine Q accept iff σ_γ is empty, $\tau = \langle k, \mathrm{OUT}_\gamma \rangle$, and $\mathrm{OUT}_\gamma < |y_k|$.
- For the constant functions λ, 0, and 1, $t_{\sigma_\gamma}(x)$ is $UNDEF(x)$, since these functions do not depend on input. For λ there are no gates whatsoever, so let machine Q reject unconditionally; for 0 and 1 we let machine Q accept iff τ indicates the appropriate constant-valued gate (and σ_γ is empty).
- For the $\mathrm{Msp}(; y_1, y_2)$ function, output bit OUT_γ is $\mathrm{Bit}(\mathrm{OUT}_\gamma + |y_2|, y_1)$, so $t_{\sigma_\gamma}(x)$ is $ADD_2(x)$. Let machine Q accept iff $\mathrm{OUT}_\gamma + |y_2| < |y_1|$ and τ indicates the appropriate input bit.
- For the $\mathrm{Lsp}(; y_1, y_2)$ function, output bit OUT_γ is $\mathrm{Bit}(\mathrm{OUT}_\gamma, y_1)$, so $t_{\sigma_\gamma}(x)$ is just x. Machine Q accepts iff $\mathrm{OUT}_\gamma < |y_2|$ and τ indicates the appropriate input bit.
- Bit OUT_γ of $\mathrm{Cond}(; y_1, y_2, y_3)$ is

  $$(\mathrm{Bit}(0, y_1)) \wedge \mathrm{Bit}(\mathrm{OUT}_\gamma, y_2)) \vee (\neg\mathrm{Bit}(0, y_1) \wedge \mathrm{Bit}(\mathrm{OUT}_\gamma, y_3)),$$

  which can be evaluated naturally by a circuit of depth 3. So $t_{\sigma_\gamma}(x)$ is either x or $ZERO(x)$, depending on which of the constantly many possibilities σ_γ is. Machine Q also tests these constantly many possibilities, as well as whether $|y_1| = 0$, $\mathrm{OUT}_\gamma < |y_2|$, and $\mathrm{OUT}_\gamma < |y_3|$, and recognizes the appropriate τ.

- For the $\mathrm{Conc}(;y1, y2)$ function, bit OUT_γ is either $\mathrm{Bit}(\mathrm{OUT}_\gamma, y_2)$ or $\mathrm{Bit}(\mathrm{OUT}_\gamma - |y_2|, y_1)$, whichever one exists. Unfortunately, machine P has no access to OUT_γ or $|y_2|$, so we insert a two-input projection gate with left input $\mathrm{Bit}(\mathrm{OUT}_\gamma, y_2)$ and right input $\mathrm{Bit}(\mathrm{OUT}_\gamma - |y_2|, y_1)$, as in the proof of Lemma 22. Machine P can tell from σ_γ whether to output the term x or the term $SUB_2(x)$. Machine Q bears responsibility for testing whether $\mathrm{OUT}_\gamma < |y_2|$ and choosing a left or a right projection gate accordingly. Note that if $x < |y_j|$, it will choose a left projection and the resulting circuit will pay no attention to $\mathrm{Bit}(\mathrm{Sub}_2(\mathrm{OUT}_\gamma), y_1)$, so it makes no difference what bit it is.

- Bit OUT_γ of $\mathrm{Bh}(;y_1)$ is $\mathrm{Bit}(\mathrm{OUT}_\gamma, y_1)$, so P outputs x. Machine Q accepts iff $\mathrm{OUT}_\gamma < \lceil |y_1|/2 \rceil$ and τ indicates the appropriate input bit.

- Bit OUT_γ of $\mathrm{Fh}(;y_1)$ is $\mathrm{Bit}(\mathrm{OUT}_\gamma + \lceil |y_1|/2 \rceil, y_1)$, so P outputs the term $ADDHALF(x)$. Machine Q tests whether the relevant bit actually exists.

- The $\mathrm{Not}(;y_1)$ function is computed by a single layer of NOT gates. Machine P outputs x, since bit OUT_γ of output depends on bit OUT_γ of input. Machine Q is straightforward.

- The $\mathrm{Or}(;y_1, y_2)$ function is computed by a layer of two-input OR gates, whose inputs are tied either to input gates of the circuit or to constant-0 gates. Again P outputs simply x. Machine Q must check whether $\mathrm{OUT}_\gamma < |y_1|$ and $\mathrm{OUT}_\gamma < |y_2|$ in addition to its usual duties.

- The $\mathrm{Ins}_0(;y_1)$ function involves only input and constant-valued gates, but bit OUT_γ of output may depend on bit $\lfloor \mathrm{OUT}_\gamma/2 \rfloor$ of y_1, so machine P outputs $HALVE(x)$. Machine Q must check whether OUT_γ is even or odd, and decide whether the gate is a constant 0 or an input gate, in addition to its usual duties. □

COROLLARY 25. *Any function in* vsc(BASE) *or* sc(BASE) *which has no normal parameters is in mapping-uniform* FNC^0.

PROOF. Any function with no normal parameters must be defined solely by safe composition from BASE functions. Safe composition preserves both linear growth rate and (by Lemma 22) mapping-uniform constant depth, so any such function has linear-size, constant-depth, mapping-uniform circuits. □

5. Bounding sc(BASE) and vsc(BASE)

In this section, we prove upper bounds on both the growth rate and the necessary circuit depth of functions in vsc(BASE) and sc(BASE).

5.1. Function growth.

LEMMA 26. *If $f(\vec{x};\vec{y}) \in \mathrm{vsc(BASE)}$, then there is a polynomial p_f with non-negative coefficients such that $|f(\vec{x};\vec{y})| \le p_f(|\vec{x}|)\cdot \max(1,|\vec{y}|)$.*

PROOF. By induction on the vsc(BASE) definition of f.

If f has no normal parameters $\vec{x}$, then it has linear growth rate by Corollary 25, so we can simply let p_f be the multiplicative constant. If f is a BASE function, then either it is a projection function (with linear growth) or it has no normal parameters.

If f is defined by safe composition, *e.g.*, $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$, then let p_g, p_u, and p_v satisfy the inductive hypothesis.

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

which has the desired form. The argument for multiple functions $\vec{u}$ and $\vec{v}$ is similar.

If f is defined by very safe DCR, *e.g.*,

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

then let polynomial p_g and constant p_h satisfy the inductive hypothesis.

$$\begin{array}{rl} \multicolumn{2}{l}{|f(z,b,\vec{x};\vec{y})|} \\ \le & \max(|g(b,\vec{x};\vec{y})|, \\ & \qquad p_h\cdot\max(1,|z|,|\vec{x}|,|\vec{y}|,|f(\mathrm{Fh}(;z),b,\vec{x};\vec{y})|,|f(\mathrm{Bh}(;z),b,\vec{x};\vec{y})|)) \\ \le & \max(|g(b,\vec{x};\vec{y})|, p_h^{\log(|z|)}\cdot\max(|\vec{x}|,|\vec{y}|,|g(b,\vec{x};\vec{y})|)) \\ \multicolumn{2}{l}{\text{by a straightforward induction on } ||z||} \\ \le & p_h^{\log(|z|)}\cdot\max(1,|\vec{x}|,|\vec{y}|,|g(b,\vec{x};\vec{y})|)) \\ \le & |z|^{\log(p_h)}\cdot\max(1,|\vec{x}|,|\vec{y}|,|g(b,\vec{x};\vec{y})|)) \\ \le & |z|^{\log(p_h)}\cdot\max(1,|\vec{x}|,|\vec{y}|,p_g(|b|,|\vec{x}|)\cdot\max(1,|\vec{y}|)) \\ \le & |z|^{\log(p_h)}\cdot(\sum|\vec{x}|)\cdot p_g(|b|,|\vec{x}|)\cdot\max(1,|\vec{y}|)), \end{array}$$

which has the desired form. □

LEMMA 27. *If $f(\vec{x};\vec{y}) \in \mathrm{sc}(\mathrm{BASE})$, then there is a polynomial p_f with nonnegative coefficients such that $|f(\vec{x};\vec{y})| \leq 2^{p_f(||\vec{x}||)} \cdot \max(1,|\vec{y}|)$.*

PROOF. The base and composition cases are similar to those in the previous lemma.

If f is defined by safe DCR, *e.g.*,

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

then let p_g and p_h satisfy the inductive hypothesis.

$$\begin{aligned} |f(z,b,\vec{x};\vec{y})| &\leq \max(|g(b,\vec{x};\vec{y})|, \\ &\qquad 2^{p_h(||z||,||b||,||\vec{x}||)} \cdot \max(1,|\vec{y}|, |f(\mathrm{Fh}(;z),b,\vec{x};\vec{y})|, \\ &\qquad\qquad |f(\mathrm{Bh}(;z),b,\vec{x};\vec{y})|)) \\ &\leq \max(|g(b,\vec{x};\vec{y})|, \\ &\qquad 2^{p_h(||z||,||b||,||\vec{x}||)} \cdot \max(1,|\vec{y}|, |f(\mathrm{Bh}(;z),b,\vec{x};\vec{y})|)) \\ &\leq 2^{\log(|z|)\cdot p_h(||z||,||b||,||\vec{x}||)} \cdot \max(1,|\vec{y}|, |g(b,\vec{x};\vec{y})|) \\ &\leq 2^{(||z||+1)\cdot p_h(||z||,||b||,||\vec{x}||)} \cdot \max(1,|\vec{y}|, 2^{p_g(||b||,||\vec{x}||)} \cdot \max(1,|\vec{y}|)) \\ &\leq 2^{(||z||+1)\cdot p_h(||z||,||b||,||\vec{x}||)+p_g(||b||,||\vec{x}||)} \cdot \max(1,|\vec{y}|) \end{aligned}$$

which has the desired form. □

5.2. Circuit depth.

LEMMA 28. *If $f(\vec{x};\vec{y}) \in \mathrm{vsc}(\mathrm{BASE})$, then f has mapping-uniform circuits of depth $O(\log(|\vec{x}|))$, independent of $|\vec{y}|$.*

PROOF. If $f \in \mathrm{BASE}$, then it is in mapping-uniform FNC^0 by Lemma 24.

If f is defined by safe composition, *e.g.*, $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$, then by Lemma 22, f has mapping-uniform circuits of depth

$$O(\max((\text{depth of } g), (\text{depth of } u), (\text{depth of } v))).$$

We know by the induction hypothesis that the depths of u and v circuits are $O(\log(|\vec{x}|))$, and the depth of g circuits is $O(\log(|u(\vec{x};)|))$. By Lemma 26, $|u(\vec{x};)|$ is polynomial in $|\vec{x}|$, so $O(\log(|u(\vec{x};)|))$ is simply $O(\log(|\vec{x}|))$. It follows that the depth of the whole f circuit is $O(\log(|\vec{x}|))$.

If f is defined by very safe DCR, *e.g.*,

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

then let G and H be circuit families satisfying the inductive hypothesis for g and h respectively. A natural circuit for f consists of a binary tree of height $\log(|z|)$ of (constant-depth) H circuits, with a layer of G circuits at the bottom. This clearly has logarithmic depth, since H has constant depth and G logarithmic depth; it remains only to prove it mapping-uniform.

As in the proof of Lemma 22, we glue the layers of the binary tree and the layer of G circuits together with projection gates, so the P machine can construct the mapping term t_σ from σ without knowing which input comes from which parameter. The P machine divides the path σ into subpaths passing through single H circuits, separated one from another by the branch through the projection gate, possibly followed by a sub-path through a G subcircuit. The term t_σ is simply the composition of the terms corresponding to these various subpaths, and since they can each be constructed in linear time (in $|\sigma|$), so can t_σ.

The Q machine may get one of two kinds of queries about the circuit. One type of query, finding the descendant gate of a given gate along a given path, is as usual trivial because of our gate-numbering scheme. The other type requires finding the type of a gate γ. In this case, Q will first measure the length of the path σ_γ and, by comparing it with the depth of H and with $\log(|z|)$, divide σ_γ into two parts: $\sigma^{previous}$ passing from the root, possibly through many layers of H subcircuits, to the output of the G or H subcircuit containing γ, and σ^{local} passing from the output of the local subcircuit down to γ. The Q machine evaluates the term $t_{\sigma^{previous}}$ by Lemma 23 to determine which output of the local subcircuit γ descends from. It can then apply the uniformity of the local subcircuit with σ^{local} to determine the type of gate γ.

Thus, the circuits constructed to evaluate functions defined by very safe DCR are mapping-uniform, and the lemma follows by induction. □

Corollary 29. *Any function in* vsc(BASE) *is computable by multi-output, U_{E^*}-uniform NC^1 circuits.*

The implication $(1 \Rightarrow 3)$ of Theorem 1 follows immediately.

LEMMA 30. *If* $f(\vec{x};\vec{y}) \in \mathrm{sc(BASE)}$, *then there is a polynomial* q_f *such that* f *has mapping-uniform circuits of depth at most* $q_f(||\vec{x}||) \cdot \max(1, ||\vec{y}||)$.

PROOF. The base case is similar to that in the previous lemma.

If f is defined by safe composition, *e.g.*, $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$, then by Lemma 22, f has mapping-uniform circuits of depth

$$O(\max((\text{depth of } g), (\text{depth of } u), (\text{depth of } v))).$$

We know, by the induction hypothesis, that the depths of u and v circuits are polylog in $|\vec{x}|$ and $O(\log)$ in $|\vec{y}|$, while the depth of g circuits is at most $q_g(||u(\vec{x};)||) \cdot \max(1, ||v(\vec{x};\vec{y})||)$. By Lemma 27, $|u(\vec{x};)| \leq 2^{p_u(||\vec{x}||)}$ and $|v(\vec{x};\vec{y})| \leq 2^{p_v(||\vec{x}||)} \cdot \max(1, |\vec{y}|)$ for some polynomials p_u and p_v, so the depth of g circuits is at most $q_g(p_u(||\vec{x}||)) \cdot \max(1, p_v(||\vec{x}||) + \max(1, ||\vec{y}||))$, which is at most $q_g(p_u(||\vec{x}||)) \cdot (p_v(||\vec{x}||) + 1) \cdot \max(1, ||\vec{y}||))$, which has the desired form.

If f is defined by safe DCR, *e.g.*,

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

then let G and H be circuit families satisfying the inductive hypothesis with polynomials q_g and q_h respectively. The natural circuit family F to compute f is again a binary tree of H circuits, with a layer of G circuits at the bottom. By the induction hypothesis, the depth of each G and H subcircuit is polylog in the lengths of its normal parameters and $O(\log)$ in the lengths of its safe parameters. The depth of the F circuits is therefore $O(\log(|z|))$ times the depth of H circuits plus the depth of G circuits, which (since z is normal) is likewise polylog in the lengths of the normal parameters and $O(\log)$ in the lengths of the safe parameters. The mapping-uniformity of F is proven as in the previous lemma. □

COROLLARY 31. *Any function in* sc(BASE) *is computable by multi-output,* U_{E^*}*-uniform, polylog depth, bounded fan-in circuits.*

The implication $(1 \Rightarrow 3)$ of Theorem 2 follows immediately.

6. Simulating Circuits

It remains to show that functions bitwise computable in ALOGTIME are in vsc(BASE) with a length mask, and that functions bitwise computable in APLOGTIME are in sc(BASE) with a length mask.

6.1. Bootstrapping. We start by defining a number of utility functions in vsc(BASE) and sc(BASE). We shall use definition by cases freely, so long as the choice of cases is decidable without recursion; any such definition expands straightforwardly into a few applications of Cond.

The function $\mathrm{ZEROES}(;x) = \mathrm{Lsp}(;\lambda, x)$ returns a string of zeroes of the same length as x.

The function $\mathrm{TWIN}(;x) = \mathrm{Conc}(;x,x)$ concatenates two copies of x.

The function $\mathrm{PADRT}(;x,y) = \mathrm{Conc}(;x,\mathrm{ZEROES}(;\mathrm{Msp}(;y,x)))$ pads its argument x on the right with zeroes to length $\max(|x|,|y|)$.

Four functions

$$\begin{array}{rcl} \mathrm{LOBIT}(;x) &=& \mathrm{Lsp}(;x,1) \\ \mathrm{DELLOBIT}(;x) &=& \mathrm{Msp}(;x,1) \\ \mathrm{HIBIT}(;x) &=& \mathrm{Msp}(;x,\mathrm{Msp}(;x,1)) \\ \mathrm{DELHIBIT}(;x) &=& \mathrm{Lsp}(;x,\mathrm{Msp}(;x,1)) \end{array}$$

allow access to the rightmost and leftmost bits of a string.

The function $\mathrm{SHL}(;x,y) = \mathrm{Lsp}(;\mathrm{Conc}(;x,y),x)$ shifts x left by appending y, discarding the high bits so the result is the same length as x. The similar function $\mathrm{SHR}(;x,y) = \mathrm{Lsp}(;\mathrm{Msp}(;x,y),x)$ truncates the rightmost $|y|$ bits of x, filling the high bits with zeroes so the result is the same length as x.

The function

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

returns a string y such that $|y| = ||x||$. Note that $\mathrm{LEN}(x;)$ does *not* return a binary representation of $|x|$ itself.

The function

$$\mathrm{ROUNDUP}(x;) = \\ \begin{cases} 1 & \text{if } |x| \leq 1 \\ \mathrm{Conc}(;\mathrm{ROUNDUP}(\mathrm{Bh}(;x);),\mathrm{ROUNDUP}(\mathrm{Bh}(;x);)) & \text{otherwise} \end{cases}$$

returns a string of 1's whose length is the least power of 2 greater than or equal to $|x|$. Then $\mathrm{EXTEND}(x;) = \mathrm{Lsp}(;x,\mathrm{ROUNDUP}(x;))$ returns x padded on the left with zeroes up to power-of-2 length.

Since an ATM has random access to its input tapes, we need a function to implement random access. We define the function $\mathrm{GETBIT}(x;i)$ to return $\mathrm{Bit}(i,x)$, treating i as a binary number of length $\lceil\log(|x|)\rceil$; if i is shorter than that, we left-extend it with zeroes, and if longer, we ignore the leftmost bits.

$$\begin{aligned}\mathrm{GETBIT}(x;i) = f_1(&\mathrm{ROUNDUP}(x;);\mathrm{EXTEND}(x;),\\ &\mathrm{Lsp}(;i,\mathrm{DELLOBIT}(;\mathrm{LEN}(\mathrm{ROUNDUP}(x;);))))\end{aligned}$$

where f_1 is defined by simultaneous, very safe DCR as follows:

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

The reader may check that $|f_1(y;x,i)| = |f_2(y;x,i)|$ for all y, x, i that arise in the computation of GETBIT, so by Lemma 12, GETBIT is in vsc(BASE).

The function

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

returns $|x|$ copies of y. (A different definition of this function was used in Cobham (1965) to provide polynomial growth rates.)

By safe DCR on COPY, we can define

$$\mathrm{GROW}(x,y;) = \begin{cases} 1 & \text{if } |x| = 0 \\ y & \text{if } |x| = 1 \\ \mathrm{COPY}(y;\mathrm{GROW}(\mathrm{Bh}(;x),y;)) & \text{otherwise} \end{cases}$$

This GROW function is in sc(BASE). Its length is $|y|^{||x||}$, so by Lemma 26, it is *not* in vsc(BASE).

LEMMA 32. *If* $t(|\vec{x}|) = O(\log(|\vec{x}|))$, *then there is a function* $\mathrm{BLOWUP}_t(\vec{x};) \in$ vsc(BASE) *such that* $|\mathrm{BLOWUP}_t(\vec{x};)| \geq 2^{t(|\vec{x}|)}$.

If $t(|\vec{x}|) = \log^{O(1)}(|\vec{x}|)$, *then there is a function* $\mathrm{BLOWUP}_t(\vec{x};) \in$ sc(BASE) *such that* $|\mathrm{BLOWUP}_t(\vec{x};)| \geq 2^{t(|\vec{x}|)}$.

PROOF. The Conc function allows us to add the lengths of parameters, and combined with the constant function 1 it produces any desired linear growth rate. The COPY function allows us to multiply the length of a normal parameter by that of a safe one. By composition of these we can construct within vsc(BASE) a function BLOWUP($\vec{x};$) whose length is any desired increasing polynomial of $|\vec{x}|$. This proves the first part of the lemma.

For the second part, we need not only polynomial but quasipolynomial growth rates, which are afforded by suitably many compositions of GROW. □

6.2. Arithmetizing an Alternating Machine. Our goal is to simulate the computation of an ATM within vsc(BASE) or sc(BASE). Throughout this section we shall assume the machine takes l inputs $X_1, \ldots, X_l$ and has k work tapes. Given such a machine and its time bound $t(|\vec{X}|)$, we need to construct and evaluate a computation tree of depth $t(|\vec{X}|)$. This takes several steps:

- Compute $t(|\vec{X}|)$, rounding it up to an even number for technical reasons.
- Construct a full binary tree T_0 of depth $t(|\vec{X}|)$, each of whose leaves represents a distinct path through the computation tree.
- Recurse through T_0, determining whether each nonleaf node accepts by ANDing or ORing whether its children accept.
- At each leaf of T_0, extract the path encoded in that leaf and recurse through it to determine the ATM configuration at that leaf.
- Decide whether this leaf configuration accepts. For input configurations, this requires using GETBIT to read input.

For the sake of presentation, we shall go through these steps in reverse order.

We represent the configuration $\langle q, L_1, \ldots, L_k, R_1, \ldots, R_k \rangle$ of an ATM by a tuple of numbers, possibly defining functions from one configuration to another by simultaneous recursion. This is only valid if all the elements of the tuple are the same length, which is ensured by our assumption that each L_i and R_i have length exactly $t(|\vec{X}|)$, the time bounds of the machine. We also pad the state q with zeroes to the same length; this is why we insist that t be at least the log of the number of states.

With this representation one can easily determine whether the state is an input state, and if so what input tape it reads, what work tape it uses as an index, and what symbol it expects to find on the input tape. Suppose, for example, it uses work tape 7 as an index to input tape 4, and accepts iff it reads a 0. Then the function

$$\mathrm{TEST}_{7,4,0}(\vec{X}; q, \vec{L}, \vec{R}) = \mathrm{Not}(; \mathrm{GETBIT}(X_4; L_7))$$

returns a 1 iff this input configuration accepts. We can in fact define a function $\mathrm{EVALLEAF}(\vec{X}; C) \in \mathrm{vsc}(\mathrm{BASE})$ to determine whether *any* given input configuration C accepts. (Some configurations in the tree may be leaves not because they are input configurations but because their children have been truncated by the time bound. In these cases, it makes no difference what EVALLEAF returns, since whether the tree accepts or rejects has already been determined.)

Similarly, one can test the state and what symbol is under each tape head by composing constantly many Cond and DELLOBIT functions, write and move on the tapes with SHL and SHR functions, and choose states by concatenating constants. By these means, we can define functions (or more properly, tuples of functions) $\mathrm{NEXT}_1(; q, \vec{L}, \vec{R})$ and $\mathrm{NEXT}_2(; q, \vec{L}, \vec{R})$ computing the two successor configurations of any given configuration. If $\langle q, \vec{L}, \vec{R}\rangle$ is an input configuration, we define both its successors to be $\langle q, \vec{L}, \vec{R}\rangle$ itself.

To find the configuration at a leaf of the computation tree, we must know the starting configuration of the ATM as well as the sequence of left/right decisions (henceforth called a "path") it made on the way. The starting configuration is relatively easy to handle. The starting state q_0 can be constructed by composing finitely many Conc and constant functions, and we can initialize each L_i and R_i to a blank work tape with $\mathrm{ZEROES}(; \mathrm{LEN}(\mathrm{BLOWUP}_t(\vec{X};);))$. Thus, we can define a function (actually a tuple of functions) $\mathrm{INIT}(\vec{X};)$ to return the starting configuration of the machine.

As for the path, a natural representation is a string of 0's and 1's, with (say) the 0's representing left branches and the 1's right branches. For technical reasons, we shall "inflate" our paths, embedding the bits of interest in positions $0, 1, 3, 7, 15, \ldots$ of an exponentially longer bit string.

Definition 33. *The* deflation *of a bit-string x is the string y of length* $\log(|x|)$ *such that for all i,* $\mathrm{Bit}(i, y) = \mathrm{Bit}(2^i - 1, x)$.

Definition 34. *An* inflated path *is a bit-string whose deflation is intended to be interpreted as a path.*

Given an inflated path P and a starting configuration C, we find the input configuration at the end of the path as follows (still writing as one function what is really a tuple defined by simultaneous recursion):

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

where $\mathrm{NEXT}(;P,C) = \mathrm{Cond}(;\mathrm{HIBIT}(;P), \mathrm{NEXT}_1(;C), \mathrm{NEXT}_2(;C))$. At each step, NEXT examines the leftmost bit of the path and invokes NEXT_1 or NEXT_2 accordingly. GETLEAF then discards the left half of the inflated path (*i.e.*, the leftmost bit of the path) and continues until the path is exhausted, at which point it returns the starting configuration.

Thus, if inputs $\vec{X}$ and an inflation P of path p are available as normal parameters, then $\mathrm{EVALLEAF}(\vec{X};\mathrm{GETLEAF}(P;\mathrm{INIT}(\vec{X};)))$ returns 0 or 1, depending on whether the final state reached by the ATM along path p accepts or rejects. In order to determine whether the *whole* ATM accepts or rejects, we must apply this to each possible path p and combine the results with AND and OR as appropriate. We do this by constructing a binary tree T_0 with an inflated path at each leaf, and performing very safe DCR on it: at each leaf of the binary tree, we call $\mathrm{EVALLEAF}(\vec{X};\mathrm{GETLEAF}(P;\mathrm{INIT}(\vec{X};)))$ as above, and at each non-leaf stage, we either AND or OR the results of the two recursive calls, depending on whether the state is universal or existential.

To meet the requirements of very safe DCR, we must define a function $H(;T,u_1,u_2)$, taking no normal parameters, which determines from a subtree T of T_0 whether the corresponding state is universal or existential and thus whether to AND or OR the recursive results u_1 and u_2. This is why we imposed the requirement in section 3.4 that the states of an ATM alternate strictly between universal and existential and start with an existential: the type of a non-leaf node in the computation tree is existential iff its distance from the root is even. We round $t(|\vec{X}|)$ up to an even number, so the distance from the root is even iff the distance from the leaves, *i.e.*, the height of the subtree T, is even. The height of the subtree is the log of the number of leaves in it, but this seems impossible to compute without recursion. Instead, we construct T_0 so as to make the parity of the height of each subtree evident: for each subtree T, the height of T will be even iff the leftmost bits of its two children are equal.

Assuming we can construct a binary tree with all these properties, we define the $\{0,1\}$-valued function $\mathrm{EVALTREE}(T,b,\vec{X};)$ by very safe DCR on a binary tree T of inflated paths as follows. The parameter b supplied to EVALTREE will be anything of the same length as an inflated path.

$$\mathrm{EVALTREE}(T,b,\vec{X};) = \begin{cases} \mathrm{EVALLEAF}(\vec{X};\mathrm{GETLEAF}(T;\mathrm{INIT}(\vec{X};))) & \text{if } |T| \le |b|; \\ \mathrm{EVALTREE}(\mathrm{Fh}(T),b,\vec{X};) \wedge \mathrm{EVALTREE}(\mathrm{Bh}(T),b,\vec{X};) & \\ \qquad \text{if } |T| > |b| \text{ and the height of } T \text{ is odd;} & \\ \mathrm{EVALTREE}(\mathrm{Fh}(T),b,\vec{X};) \vee \mathrm{EVALTREE}(\mathrm{Bh}(T),b,\vec{X};) & \\ \qquad \text{if } |T| > |b| \text{ and the height of } T \text{ is even.} & \end{cases}$$

To put this more formally,

$$\mathrm{EVALTREE}(T, b, \vec{X};) =$$
$$\begin{cases} \mathrm{EVALLEAF}(\vec{X}; \mathrm{GETLEAF}(T; \mathrm{INIT}(\vec{X};))) & \text{if } |T| \leq |b|; \\ H(; T, \mathrm{EVALTREE}(\mathrm{Fh}(T), b, \vec{X};), \mathrm{EVALTREE}(\mathrm{Bh}(T), b, \vec{X};)) & \text{if } |T| > |b|, \end{cases}$$

where

$$\begin{aligned} &H(; T, u_1, u_2) = \\ &\mathrm{Cond}(; \mathrm{HIBIT}(; T), \\ &\qquad \mathrm{Cond}(; \mathrm{HIBIT}(; \mathrm{Bh}(T)), \mathrm{Or}(; u_1, u_2), \mathrm{And}(; u_1, u_2)), \\ &\qquad \mathrm{Cond}(; \mathrm{HIBIT}(; \mathrm{Bh}(T)), \mathrm{And}(; u_1, u_2), \mathrm{Or}(; u_1, u_2))). \end{aligned}$$

The question remains: how do we get this binary tree of inflated paths? We need one inflation of each possible path through the ATM's computation tree, and they must obey the right relationships: two siblings must represent paths differing only in the first bit, first cousins must represent paths differing in the first two bits, and so on. Furthermore, at each level of the tree, the two children must have equal high bits iff the height is even. All this is accomplished by the following function:

$$\mathrm{PATHS}(Y;) = \begin{cases} \lambda & \text{if } |Y| = 0; \\ \mathrm{Conc}(; 0, 1) & \text{if } |Y| = 1; \\ H(; \mathrm{PATHS}(\mathrm{Bh}(; Y);)) & \text{if } |Y| > 1, \end{cases}$$

where

$$\begin{aligned} &H(; v) = \\ &\quad \begin{cases} \mathrm{Conc}(; \mathrm{Ins}_0(; v), \mathrm{Ins}_1(; v)) & \text{if } \mathrm{HIBIT}(; v) \neq \mathrm{HIBIT}(; \mathrm{Bh}(; v)); \\ \mathrm{Conc}(; \mathrm{Ins}_0(; v), \mathrm{Ins}_1(; \mathrm{Not}(; v))) & \text{if } \mathrm{HIBIT}(; v) = \mathrm{HIBIT}(; \mathrm{Bh}(; v)). \end{cases} \end{aligned}$$

This PATHS function has many useful properties. First, $\mathrm{PATHS}(Y;) = \mathrm{PATHS}(\mathrm{ROUNDUP}(Y;);)$, so in discussing it we can assume without loss of generality that $|Y|$ is a power of two. Since $|H(; v)| = 4|v|$, we have $|\mathrm{PATHS}(Y;)| = 2 \cdot 4^{\log(|Y|)} = 2|Y|^2$. We shall view $\mathrm{PATHS}(Y;)$ as (the root of) a binary tree of depth $||Y|| = 1 + \log(|Y|)$, each node of which is the concatenation of its children, and each leaf of which is a string of length $|Y|$.

The less obvious properties of PATHS are presented in the following technical lemma.

Lemma 35. *The binary tree represented by* PATHS$(Y;)$ *has the following properties:*

- *For any non-leaf node, the leftmost bits of its two children are equal iff the node's height is even.*
- *The deflations of the leaf nodes comprise one copy of each of the possible bit-strings of length* $||Y|| = 1 + \log(|Y|)$.
- *Two leaves have a common ancestor of height* i *iff the rightmost* $||Y|| - i$ *bits of their deflations are equal.*

The proof of this lemma involves a number of straightforward but tedious inductions, and we omit it, but illustrate it with the first few values of PATHS, in which overbraces delimit the leaves:
$\mathrm{PATHS}(\lambda;) = \lambda$, one leaf of length 0.
$\mathrm{PATHS}(1;) = \hat{0}\hat{1}$, two leaves of length 1.
$\mathrm{PATHS}(11;) = \overbrace{00}\,\overbrace{10}\,\overbrace{01}\,\overbrace{11}$, four leaves of length 2.
$\mathrm{PATHS}(1111;) = \overbrace{0000}\overbrace{1000}\overbrace{0010}\overbrace{1010}\overbrace{1111}\overbrace{0111}\overbrace{1101}\overbrace{0101}$, eight leaves of length 4, each of which deflates to a distinct 3-bit path.
$\mathrm{PATHS}(11111111;) =$
$\overbrace{00000000}\overbrace{10000000}\overbrace{00001000}\overbrace{10001000}\overbrace{10101010}\overbrace{00101010}\overbrace{10100010}\overbrace{00100010}$
$\overbrace{01010101}\overbrace{11010101}\overbrace{01011101}\overbrace{11011101}\overbrace{11111111}\overbrace{01111111}\overbrace{11110111}\overbrace{01110111}$,
16 leaves of length 8, each of which deflates to a distinct 4-bit path. The reader is invited to verify the remaining properties in this example.

We put all this together with the following function:

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

This function determines whether the ATM, given input $\vec{X}$, accepts or rejects. Every part of its definition, with the possible exception of BLOWUP_t, has been by safe composition or very safe DCR. If $t(|\vec{X}|) = O(\log(|\vec{X}|))$, then BLOWUP_t is in vsc(BASE), and therefore so is SIMULATE. If, on the other hand, $t(|\vec{X}|) = \log^{O(1)}(|\vec{X}|)$, then BLOWUP_t cannot be defined in vsc(BASE) but can be in sc(BASE), so SIMULATE is also in sc(BASE). We have proven the following lemma:

LEMMA 36. *For any log-time ATM with inputs $\vec{X}$, there is a function $f(\vec{X};) \in$ vsc(BASE) that returns 1 if the machine accepts and 0 otherwise.*

For any polylog-time ATM with inputs $\vec{X}$, there is a function $f(\vec{X};) \in$ sc(BASE) that returns 1 if the machine accepts and 0 otherwise.

6.3. Comprehension. We have shown that any relation, *i.e.*, any $\{0,1\}$-valued function, in ALOGTIME (respectively APLOGTIME) is in vsc(BASE) (respectively sc(BASE)). But what of functions with values in $\{0,1\}^*$? The following lemma demonstrates that, so long as the length of the function value is a sufficiently slow-growing and easily computed function, the ability to compute *each bit* of a function enables us to compute its whole value at once.

LEMMA 37. *Let $F(\vec{x};)$ be a tiered function. If there are functions $f(i,\vec{x};)$ and $\mathrm{SAME\_LENGTH}_F(\vec{x};)$ in vsc(BASE) such that, for all $\vec{x}$,*

- $|F(\vec{x};)| = |\mathrm{SAME\_LENGTH}_F(\vec{x};)|$,
- *for all* $i < |F(\vec{x};)|$, $(f(i,\vec{x};) = \mathrm{Bit}(i, F(\vec{x};)))$, *and*
- *for all* $i \in \{0,1\}^*$, $f(i,\vec{x};) = f(\mathrm{Conc}(;0,i),\vec{x};)$,

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

The same holds with vsc(BASE) *replaced throughout by* sc(BASE).

PROOF. We prove this by a simpler version of the "paths" technique used in section 6.2. We compute $\mathrm{SAME\_LENGTH}_F(\vec{x};)$, then generate a binary tree T whose leaves are the binary numbers $(|\mathrm{SAME\_LENGTH}_F(\vec{x};)|-1),\ldots,2,1,0$, each embedded in a block of length $||\mathrm{SAME\_LENGTH}_F(\vec{x};)|-1|$.

$\mathrm{COUNTDN}(z;) = f_2(z;\mathrm{LEN}(z;))$

$$f_1(z;s) = \begin{cases} \lambda & \text{if } |z|=0 \\ \mathrm{SHL}(;\mathrm{ZEROES}(;s),1) & \text{if } |z|=1 \\ \mathrm{TWIN}(;\mathrm{SHL}(;f_1(\mathrm{Bh}(z);s),0)) & \text{if } |z|>1 \end{cases}$$
$$f_2(z;s) = \begin{cases} \lambda & \text{if } |z|=0 \\ \mathrm{ZEROES}(;s) & \text{if } |z|=1 \\ \mathrm{Conc}(;\mathrm{Or}(;f_1(\mathrm{Bh}(z);s), f_2(\mathrm{Bh}(z);s)), f_2(\mathrm{Bh}(z);s)) & \text{if } |z|>1 \end{cases}$$

We can then define F by very safe DCR as follows:

$$\begin{aligned} F(\vec{x};) &= F'(\mathrm{COUNTDN}(\mathrm{SAME\_LENGTH}_F(\vec{x};);), \\ &\qquad \mathrm{LEN}(\mathrm{SAME\_LENGTH}_F(\vec{x};);),\vec{x};) \\ F'(T,b,\vec{x};) &= \begin{cases} f(T,\vec{x};) & \text{if } |T| \le |b| \\ \mathrm{Conc}(;F'(\mathrm{Fh}(;T),b,\vec{x};), F'(\mathrm{Bh}(;T),b,\vec{x};)) & \text{if } |T| > |b| \end{cases} \end{aligned}$$

Since f is insensitive to leading zeroes, if T is the i-th leaf from the right then $f(T, \vec{x}) = f(i, \vec{x})$. The iterated function, Conc, is associative, so the effect is simply to concatenate the bits $f(|b|-1, \vec{x};), f(|b|-2, \vec{x};), \ldots, f(1, \vec{x};), f(0, \vec{x};)$. By the known properties of SAME_LENGTH$_F$ and f, this concatenation is precisely $F(\vec{x};)$. $\square$

7. Conclusions and Observations

The notion of classifying formal parameters into tiers by how they are used in computation has produced a number of simple characterizations of complexity classes. The project began with the introduction of tiering in Leivant (1990). The related second-order logical characterization of $\mathcal{P}$ in Leivant (1991) inspired the recursive-functions characterization of $\mathcal{FP}$ in Bellantoni & Cook (1992), which in turn inspired work on fast parallel classes in the present paper, and on the polynomial hierarchy and a form of linear space in Bellantoni (1992) (the latter result was independently proven in Handley 1992). Bellantoni, Leivant, and I all independently conjectured a "very safe" characterization of linear time; for one proof, see Bloch (1992).

Leivant has pointed out that all these characterizations depend on viewing the universe of data as a free (or nearly so) algebra, and that the differences among the characterizations of APLOGTIME, $\mathcal{P}$, and linear space are primarily differences in the choice of this algebra. Since the three algebras involved could be called the three simplest algebras suitable for representing computational data, we conclude that these three complexity classes are foundationally extremely natural. Indeed, Otto (1992) has recast the results of Bellantoni & Cook into category theory, suggesting that the class is natural even in an extremely abstract setting with no notion of computation.

An obvious direction for further research is to find other complexity classes that are characterized by tiered recursion over a simple algebra, and eventually to determine in general what classes have such characterizations and thus may be considered foundationally "natural classes." Similarly, one may ask how properties of the algebra relate to the complexity classes they characterize.

An alternate interpretation of very safe recursion is as safe recursion with a prohibition on *nested* applications. A natural question, then, is what complexity classes arise from allowing a fixed nesting depth greater than one. Bellantoni (1992) counted nestings of minimization to produce characterizations of the levels of the Polynomial Hierarchy, and it seems that similar techniques could distinguish infinite hierarchies within polynomial time or polylog space.

Acknowledgements

This work began at the 1991 Conference on Proof Theory, Complexity and Arithmetic, which I attended on NSF/ČSAV grant INT-8914569 through the efforts of Sam Buss and Gaisi Takeuti. Further work was supported by the University of Manitoba and Judy Goldsmith's NSERC grant OGP0121527.

I would like to thank Sam Buss, Judy Goldsmith, and an anonymous referee for many helpful suggestions, both technical and expository. Discussions with Stephen Bellantoni and Daniel Leivant also helped illuminate the subject.

References

William Allen, Arithmetizing uniform NC. *Annals of Pure and Applied Logic* **53**(1) (1991), 1–50. See also *Divide and Conquer as a Foundation of Arithmetic*, Ph.D. thesis, University of Hawaii at Manoa, 1988.

Toshiyasu Arai, Frege systems, ALOGTIME and bounded arithmetic. Manuscript, 1992.

D.A. Barrington, Bounded-width polynomial-size branch programs recognize exactly those languages in NC^1. *J. Comput. Sys. Sci.* **38**(1) (1989), 150–164. See also *Proc. Eighteenth Ann. ACM Symp. Theor. Comput.*, ACM Press 1986, 1–5.

David A. Mix Barrington, Quasipolynomial size circuit classes. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*. IEEE Computer Society Press, 1992, 86–93.

Stephen Bellantoni, *Predicative Recursion and Computational Complexity*. PhD thesis, University of Toronto, 1992.

Stephen Bellantoni and Stephen Cook, A new recursion-theoretic characterization of the polytime functions. *computational complexity* **2** (1992), 97–110.

Stephen Bloch, Alternating function classes within $\mathcal{P}$. Technical Report 92-16, University of Manitoba Computer Science Dept, 1992.

Samuel R. Buss, *Bounded Arithmetic*. Number 3 in Studies in Proof Theory. Bibliopolis (Naples), 1986. See also "The polynomial hierarchy and bounded arithmetic" in *Proc. Seventeenth Ann. ACM Symp. Theor. Comput.*, ACM Press 1985, 285–290.

Samuel R. Buss, The boolean formula value problem is in ALOGTIME. In *Proc. Nineteenth Ann. ACM Symp. Theor. Comput.*, 1987, 123–131.

A. Chandra, D. Kozen, and L. Stockmeyer, Alternation. *J. Assoc. Comput. Mach.* **28**(1) (1981), 114–133.

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

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

Peter Clote and Gaisi Takeuti, Bounded arithmetic for NC, $ALogTIME$, L and NL. *Annals of Pure and Applied Logic* **56** (1992), 73–117.

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

Kevin Compton and Claude LaFlamme, An algebra and a logic for NC^1. *Information and Computation* **87** (1990), 241–263.

W.G. Handley, Bellantoni and Cook's characterization of polynomial time functions. Typescript, 1992.

Daniel Leivant, Subrecursion and lambda representation over free algebras. In *Feasible Mathematics*, ed. Samuel Buss and Philip Scott, *Perspectives in Computer Science*, 281–291. Birkhäuser, 1990.

Daniel Leivant, A foundational delineation of computational feasibility. In *Proc. Sixth IEEE Conf. Logic in Computer Science*. IEEE Computer Society Press, 1991.

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

James Otto, A category-theoretic characterization of polytime. Contributed talk at MSI Feasible Mathematics Conference, 1992.

W. Ruzzo, On uniform circuit complexity. *J. Comput. Sys. Sci.* **22** (1981), 365–383.

Manuscript received 16 December 1992

Stephen Bloch
Department of Computer Science
915 Patterson Office Tower
University of Kentucky
Lexington, KY 40506–0027
sbloch@s.ms.uky.edu