Alternating Function Classes Within $\mathcal{P}$

Stephen Bloch*

Technical Report # 92-16

October 11, 1995

Abstract

We define the notion of adding "small amounts" of nondeterminism to a deterministic function class, and give a machine model; the result is a functional AC^0 closure of the deterministic class. We characterize, by the "safe parameters" technique, the classes of functions computable in linear and in quasilinear time on a multi-tape Turing machine. We then combine these two results by extending the "safe parameters" characterizations to the functions computable in (quasi)linear time with small amounts of nondeterminism, and discuss implications for both sequential and parallel complexity.

*Computer Science Dept, University of Manitoba. This work was sponsored in part by NSERC operating grant OGP0121527.

1 Introduction

Since the 1960's complexity theorists have devoted much of their energy to the class $\mathcal{P}$, the relations computable in time bounded by a polynomial in the length of the input, and to the corresponding function class $\mathcal{FP}$, because these classes seem to capture the notion of "feasible computation." The usual arguments for this thesis are that any sequential complexity class not within $\mathcal{P}$ or $\mathcal{FP}$ is infeasible on realistic-sized inputs, and that smaller complexity classes lack the robustness (closure properties and invariance under a wide variety of machine models) that allows us to prove interesting theorems about $\mathcal{P}$ and $\mathcal{FP}$. Furthermore, although few people would consider an $O(n^{28})$-time algorithm feasible, most real-world problems known to be in polynomial time actually have *small-degree* polynomial-time algorithms and are therefore reasonably practical.

The last point suggests that we look *within* polynomial time to find under what circumstances a problem has a small-degree polynomial-time algorithm. The best we could ask of a sequential algorithm is linear time, but for many purposes this seems too restrictive: it does not contain sorting, and it remains open whether it contains integer multiplication. There are also problems from the theoretician's viewpoint: linear time appears to depend sensitively on such arbitrary details of the machine model as the number of tapes of a Turing machine, not to mention random access.

A class that remedies most of these problems is *quasilinear time* (QL), the problems solvable on a deterministic multi-tape Turing machine in time $O(n \log^{O(1)}(n))$. This class, only slightly slower than linear time, contains multiplication, sorting, and other important tools. It is invariant under change in number of tapes (so long as there are at least two) and closed under simulation techniques such as time-stamping each step of a computation. Furthermore, important theorems involving $\mathcal{P}$ have analogues with QL in place of $\mathcal{P}$, e.g. Schnorr's proof that SAT is complete for nondeterministic QL under deterministic QL reductions [Sch78].

But many everyday problems in $\mathcal{P}$ cannot be expected to have linear or quasilinear-time algorithms. Given a polynomial-time algorithm we may distinguish between two sources of difficulty: is it performing an inherently time-consuming *test*, or merely testing a large number of candidates for problem solution? An example is the problem of determining whether a graph contains a k-clique. A given k-vertex subgraph can be tested for completeness in time $O(k^2 M(n))$, where $M(n)$ is a small polynomial depending on the memory model and the representation of the graph. The naïve algorithm which enumerates k-subgraphs and tests them for completeness takes $O(n^k k^2 M(n))$ time, most of which is a loop that does an existential search and could be replaced by $k \log(n)$ bits of existential nondeterminism. Such problems were first investigated in [BG93], where it is proven that if QL with $O(\log(n))$ bits of existential nondeterminism can simulate all of $\mathcal{P}$, then $\mathcal{P} \neq \mathcal{NP}$. Then in [BG92] we explored the "sharply bounded hierarchy over QL," or $SBH(QL)$, in which there may be constantly many n-bounded quantifiers — existential, universal, or both. This is equivalent to QL with $O(\log(n))$ bits of nondeterminism and $O(1)$ alternations.

This paper expands the scope of [BG92] in several directions. We give machine-independent, recursion-theoretic characterizations of both linear and quasilinear-time *functions*. We then define (by an alternating machine model) what it means to compute a function with the aid of nondeterminism, defining function classes that naturally correspond to $SBH(QL)$ and $SBH(linear)$. We extend our machine-independent characterizations to these latter function classes, and note that they also yield characterizations of the corresponding relation classes.

Our recursion-theoretic characterizations, which do not depend on any explicit linear or quasilinear bounds, argue for the naturalness of both linear and quasilinear time as function classes. The "alternating function machine" model presented here not only allows us to add nondeterminism to these function classes, but may be usefully applied to classes other than linear and quasilinear time. Finally, the structure of $SBH(QL)$ offers valuable information about both sequential and parallel complexity. The sharply bounded hierarchy over QL is within $\mathcal{P}$ and contains many of the known $\mathcal{P}$ algorithms, but the number of quantifiers in an $SBH(QL)$ definition bounds the *degree* of its corresponding polynomial-time algorithm. And the relationships of $SBH(QL)$ and related classes with $\mathcal{P}$ would have important implications for the extent to which polynomial-time computation can be accelerated with polynomially many processors.

2 Definitions

2.1 Constant-time Computation

In section 3 we shall characterize linear and quasilinear time by iterating, (quasi)linearly many times, a constant-time operation (the obvious example being the NEXT function of a TM). But it is not clear what operations, in general, should be considered "constant-time."

Definition 1 *A function $f(x_1, \ldots x_j)$ is* constant-time *if there is a constant-time Turing machine with k tapes and at most k halting states, such that if the machine starts with all its input heads at the right ends of their tapes and terminates in halting state i, then the function value is on tape i, with the head on that tape at the right end of the data.*

This class includes the identity function, as well as such basic operations as $\mathit{Half}(x) = \lfloor \frac{x}{2} \rfloor$, $s_0(x) = 2x$, and $s_1(x) = 2x + 1$, assuming numbers are stored in binary with least significant bit at the right-hand end. It also includes all constant functions as well as the "conditional function"

$$Cond(w, x, y, z) = \begin{cases} x & \text{if } w = 0; \\ y & \text{if } w \text{ is odd;} \\ z & \text{if } w > 0 \text{ is even.} \end{cases}$$

Lemma 2 *The class of constant-time functions is closed under composition.*

PROOF. Suppose a function f is defined by composition as

$$f(\vec{x}) = g(u_1(\vec{x}), u_2(\vec{x}), \ldots u_m(\vec{x})).$$

For simplicity of notation, assume $m = 2$ and the parameter list $\vec{x}$ is a single parameter x, so we actually have

$$f(x) = g(u_1(x), u_2(x)),$$

which is sufficiently general to bring up all the relevant issues.

The intuition is that a constant-time function must return either a constant-time modification of one of its inputs or a constant-time modification of the empty string. We distinguish these cases, and break the input tape x into a "long" part that f doesn't have time to see or modify, and a "short" part which, by being only constant-length, can be readily copied and re-used. (A similar notion, with among other differences the replacement of "constant" by "polynomial," was developed independently in Chapter 6 of [Bel92].)

We construct the machine to compute f as follows. Since the computation of u_1 takes only constant time t_1, it can only reach the t_1 low bits of x; similarly for u_2 and time t_2 and for g and time t_g. Let $t = \max(t_1, t_2) + t_g$. Copy the t low bits of x to two spare tapes x_1 and x_2, delete the t low bits of the original copy of x, and use the two "spare" tapes, plus as many additional blank tapes as needed by the

u_1 and u_2 machines, to compute $u_1(x_1)$ and $u_2(x_2)$. Depending on the final state of the u_1 machine, the function value $u_1(x)$ may be on either a formerly-blank tape or the input tape x_1, and similarly for $u_2(x)$. Denote by y_1 whichever tape u_1's output is on, and similarly y_2 for u_2's output.

If u_1 leaves its output on the same tape as its input, the true function value should be x concatenated with the (possibly modified) tape x_1, which is still at least t_g bits long. If, on the other hand, u_1 leaves its output on a formerly-blank tape, it has discarded the rest of the information on tape x, and the function value should be simply the contents of that formerly-blank tape. All the same holds for u_2.

Now we can compute g on these function values simply by looking at (and modifying) only the tapes y_1 and y_2 and perhaps a few blank tapes. When the g machine terminates, which halting state it reaches will indicate whether the function value is on tape y_1, tape y_2, or a formerly-blank tape. There are several cases.

- If the value of g is on tape y_1, and if tape y_1 is really the tape x_1 that was input to u_1, then we concatenate the contents of y_1 to the end of the input tape x and declare the answer to be on tape x. Similarly with tape y_2.
- If the value of g is on tape y_1, and if tape y_1 is really one of the tapes that were given to u_1 as blank, then we declare y_1 to be the correct answer. Similarly with y_2.
- If the value of g is on one of the tapes given to g as blank, then we declare the contents of that (formerly blank) tape to be the correct answer.

2.2 Very Safe Recursion on Notation

In [BC92], Bellantoni and Cook define a scheme called "safe recursion on notation," similar to Cobham's "bounded recursion on notation" (see [Cob65]) but with the growth-rate bounds of Cobham's scheme replaced by syntactic restrictions on which parameters may be used where. Each function is considered to take two kinds of parameters, called "normal" and "safe", the latter of which is subject to only a restricted class of operations. We distinguish them syntactically by separating normal parameters from safe parameters with a semicolon, e.g. $f(x_1, x_2; y, z_1, z_2)$, in which x_1 and x_2 are normal while y, z_1, and z_2 are safe. (A similar scheme was described, more or less independently, in [Lei90].)

"Safe parameters" characterizations depend on a trap-door principle: a safe parameter to a function cannot be used as normal within the computation of that function, nor in any way affect a value used as normal within the computation of that function, although normal parameters may be used as safe. This principle is enforced by the following scheme:

Definition 3 (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*

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

The most important difference between normal and safe parameters is that recursion may not be controlled by a safe parameter. To be more precise,

Definition 4 (Bellantoni & Cook) *A function $f(z,\vec{x};\vec{y})$ is defined by* safe recursion on notation *from functions $g(\vec{x};\vec{y})$ and $h(z,\vec{x};u,\vec{y})$ if*

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(z,\vec{x};\vec{y}) &= h\left(z,\vec{x}; f\left(\lfloor z/2 \rfloor, \vec{x};\vec{y}\right), \vec{y}\right) \quad \textit{for } z > 0 \end{aligned}$$

By combining these two schemes with a simple set of base functions (including, among others, the *Half*, s_0, s_1, and *Cond*functions described above), Bellantoni and Cook gave an elegant characterization of the class of polynomial-time-computable functions.

In [Blo92] and [Blo94] we modified the scheme of safe recursion on notation in two ways: first, "recursion on notation" was replaced by a "divide-and-conquer recursion" akin to that used for the Mergesort; and second, this divide-and-conquer recursion was made "very safe" by requiring the iterated functions to treat all their parameters as safe. As these two modifications were essentially orthogonal, a natural question was "what would very safe recursion on notation produce?" Daniel Leivant [Lei92], Stephen Bellantoni [Bel92], and the author have each independently proposed that it straightforwardly characterizes some form of linear time. We present one proof of this proposition; the author has not yet seen the details of Leivant's or Bellantoni's proofs.,

Definition 5 *A function $f(z,\vec{x};\vec{y})$ is defined by* very safe recursion on notation *from functions $g(\vec{x};\vec{y})$ and $h(;z,\vec{x},u,\vec{y})$ if*

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(z,\vec{x};\vec{y}) &= h\left(;z,\vec{x},f\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right),\vec{y}\right) \quad \text{for } z>0. \end{aligned}$$

Very safe recursion on notation differs from safe recursion on notation only in that the iterated function h must treat *all* its parameters as safe, and therefore cannot itself be defined by recursion at all. We shall define a system in which any function with no normal parameters (hence any function susceptible to iteration) is in fact a constant-time function [1] in the sense of subsection 2.1.

A natural extension of such a recursion scheme is to allow the definition of multiple functions in parallel, e.g.

Definition 6 *A tuple of functions $f_1(z,\vec{x};\vec{y}),\ldots f_k(z,\vec{x};\vec{y})$ is defined by* simultaneous very safe recursion on notation *from functions $g_1(\vec{x};\vec{y}),\ldots g_k(\vec{x};\vec{y})$, and $h_1(;z,\vec{x},u_1,\ldots u_k,\vec{y}),\ldots h_k(;z,\vec{x},u,\vec{y})$ if for all $1\le i\le k$,*

$$\begin{aligned} f_i(0,\vec{x};\vec{y}) &= g_i(\vec{x};\vec{y}) \\ f_i(z,\vec{x};\vec{y}) &= h_i\left(;z,\vec{x},f_1\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right),\ldots f_k\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right),\vec{y}\right) \quad \text{for } z>0 \end{aligned}$$

We shall construct a class of functions by closing a certain base set of functions under the operations of safe composition and simultaneous [2] very safe recursion on notation; we shall then show that the class so constructed contains all and only the functions computable in linear time on a TM.

Definition 7 *The set BASE of basic functions, for purposes of this paper, includes the projection functions $\pi_k^{i,j}(x_1,\ldots,x_i;x_{i+1},\ldots,x_{i+j})=x_k$ as well as*

$$\begin{aligned} Z(;) &= 0, \\ s_0(;x) &= 2x, \\ s_1(;x) &= 2x+1, \end{aligned}$$

[1] We could instead simply declare that only constant-time functions are susceptible to iteration, but this would introduce a semantic notion into an otherwise entirely syntactic characterization.

[2] It is open whether the simultaneous recursion scheme is actually any more powerful than the single-function recursion scheme. We shall not address this question here, as both are realistic models of iterative computation.

$$
\begin{aligned}
Cond(;w,x,y,z) &= \begin{cases} x & \text{if } w=0; \\ y & \text{if } w \text{ is odd;} \\ z & \text{if } w>0 \text{ is even,} \end{cases} \\
Half(;x) &= \left\lfloor \frac{x}{2} \right\rfloor .
\end{aligned}
$$

We prove in Theorem 10 that the linear-time functions (more precisely, the union over $k \in \mathbf{N}$ of the linear-time functions on k-tape Turing machines) are exactly those functions definable from BASE functions with safe composition and very safe recursion on notation.

Bellantoni has speculated ([Bel92]) on the effects of loosening in various ways the prohibition that "no safe parameter may in any way affect a normal parameter." One proposed change allows the *length* of a safe parameter to be used as normal.

Definition 8 *We say that a function $f(\vec{x};\vec{y})$ is defined by* length composition *from the functions $g(x_1 \dots x_r; y_1 \dots y_s)$, $u_1(\vec{x},\vec{Y};) \dots u_r(\vec{x},\vec{Y};)$, and $v_1(\vec{x};\vec{y}) \dots v_s(\vec{x};\vec{y})$ if*

$$f(\vec{x};\vec{y}) = g(u_1(\vec{x},|\vec{y}|;) \dots u_r(\vec{x},|\vec{y}|;); v_1(\vec{x};\vec{y}) \dots v_s(\vec{x};\vec{y}).$$

(An alternate way of viewing the safe-parameters technique is to consider the universe to contain "impredicative numbers," with a subtype called "predicative numbers;" this extension then corresponds to allowing a length function whose domain is the whole universe and whose range is a subset of the predicative numbers.) Clearly length composition is at least as powerful as safe composition. Bellantoni [Bel92] mentions that such a scheme treating the length of a safe parameter as normal can be introduced without affecting his characterization of polynomial-time functions. If, however, we add such a scheme to our characterization of *linear* time, the function class expands to comprise the quasilinear-time functions; this is made precise in Theorem 16.

3 Characterizing Linear and Quasilinear Time

The two main results of this section are characterizations of linear and quasilinear time on multi-tape Turing machines by "safe parameters" methods.

Definition 9 *The function class $\mathcal{D}$ is the closure of the set BASE under the operations of safe composition and simultaneous very safe recursion on notation.*

Theorem 10 *The function $f(\vec{x};)$ is in $\mathcal{D}$ iff there is a multi-tape, linear-time TM that computes $f(\vec{x})$.*

We first prove a lemma about growth rates.

Lemma 11 *If $f(\vec{x};\vec{y}) \in \mathcal{D}$, then for some constants k and c and for all values of $\vec{x}$ and $\vec{y}$ we have $|f(\vec{x};\vec{y})| \leq k \cdot \max(|\vec{x}|) + \max(|\vec{y}|) + c$.*

Proof. This is true for all the BASE functions, as the result of each BASE function has length at most one bit longer than its longest argument. Now suppose a function f is defined by safe composition:

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

For simplicity we'll assume $r = s = 1$, and simply say $f(\vec{x}; \vec{y}) = g(u(\vec{x};); v(\vec{x}; \vec{y}))$. Assuming the existence of constants k_g, c_g, k_u, c_u, k_v, and c_v satisfying the induction hypothesis for functions g, u, and v respectively, we have

$$\begin{aligned} |f(\vec{x}; \vec{y})| &\leq k_g \cdot |u(\vec{x};)| + |v(\vec{x}; \vec{y})| + c_g \\ &\leq k_g \cdot (k_u \cdot \max(|\vec{x}|) + c_u) + (k_v \cdot \max(|\vec{x}|) + \max(|\vec{y}|) + c_v) + c_g \\ &\leq (k_g \cdot k_u + k_v) \cdot \max(|\vec{x}|) + \max(|\vec{y}|) + (k_g \cdot c_u + c_v + c_g) \end{aligned}$$

which is of the desired form.

On the other hand, suppose a tuple of functions $f_1, \dots f_r$ are defined by simultaneous very safe recursion on notation, i.e. for each $1 \leq i \leq r$

$$\begin{aligned} f_i(0, \vec{x}; \vec{y}) &= g_i(\vec{x}; \vec{y}) \\ f_i(z, \vec{x}; \vec{y}) &= h_i\left(; z, \vec{x}, \vec{y}, f_1\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right), \dots f_r\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right)\right) \quad \text{for } z > 0 \end{aligned}$$

By the induction hypothesis, there are constants $k_{g_1}, \dots k_{g_r}$, $c_{g_1}, \dots c_{g_r}$, and $c_{h_1}, \dots c_{h_r}$ such that for all $1 \leq i \leq r$,

$$\begin{aligned} |g_i(\vec{x}; \vec{y})| &\leq k_{g_i} \cdot \max(|\vec{x}|) + \max(|\vec{y}|) + c_{g_i} \\ |h_i(; z, \vec{x}, \vec{y}, \vec{u})| &\leq \max(|z|, |\vec{x}|, |\vec{y}|, |\vec{u}|) + c_{h_i}. \end{aligned}$$

Note that each function h_i takes no normal parameters and therefore adds at most c_{h_i} bits to the length of its longest argument. Now let $c_h = \max(c_{h_1}, \dots c_{h_r})$. For each function f_i we have that

$$|f_i(z, \vec{x}; \vec{y})| \leq c_h + \max\left(|z|, |\vec{x}|, |\vec{y}|, \left|f_1\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right)\right|, \dots \left|f_r\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right)\right|\right).$$

It follows that $\max_i(|f_i(z, \vec{x}; \vec{y})|) \leq c_h \cdot |z| + \max_i(|g_i(\vec{x}; \vec{y})|)$, and so there is a bound of the form $k \cdot \max(|\vec{x}|) + \max(|\vec{y}|) + c$ on the growth of each function $f_i(z, \vec{x}; \vec{y})$, as desired. ∎

Thus every function in $\mathcal{D}$ has linear growth rate. The converse holds, too, in a sense:

Lemma 12 *For any constants $k, c \in \mathbf{N}$, there is a function $f \in \mathcal{D}$ such that for all $x \in \mathbf{N}$,*

$$|f(x;)| = k|x| + c.$$

PROOF. To show this, we define a function

$$\mathit{ReverseConcat}(x; y) = \begin{cases} y & \text{if } x = 0 \\ s_1(; \mathit{ReverseConcat}(\lfloor x/2 \rfloor; y)) & \text{if } x \text{ is odd} \\ s_0(; \mathit{ReverseConcat}(\lfloor x/2 \rfloor; y)) & \text{if } x > 0 \text{ is even} \end{cases}$$

This function produces as output the binary form of y concatenated with the reversal of the binary form of x. Clearly $|\mathit{ReverseConcat}(x; y)| = |x| + |y|$. If *ReverseConcat* is composed suitably many times, we can easily multiply the length of a parameter by any desired constant. We can also add any desired constant to, or subtract it from, the length of a parameter by suitably many s_1 or *Half* functions. ∎

As pointed out earlier, all the base functions are constant-time functions. Since the class of constant-time functions is closed under composition, it follows immediately that any function defined by composition alone from the base functions is a constant-time function. In particular, any function in $\mathcal{D}$ with no normal parameters is a constant-time function.

We next show that a function defined by very safe recursion on notation, iterating a constant-time function, can be evaluated on a multi-tape TM in linear time. This will prove one direction of Theorem 10.

Lemma 13 *If $f(\vec{x};\vec{y}) \in \mathcal{D}$, then for each tuple of values $(\vec{u},\vec{v})$ there is a constant r such that in the computation of $f(\vec{u};\vec{v})$, for at most one of the safe parameters $\vec{y}$ do more than the low r bits affect the result.*

Furthermore, there is a multi-tape TM which, for all $\vec{u},\vec{v}$, evaluates f in time linear in $\max(|\vec{u}|,|\vec{v}|)$ and indicates, by its final state, which (if any) safe parameter has more than r bits that affect the result.

PROOF. The Lemma clearly holds for all the base functions. If f is defined by composition, e.g. $f(\vec{x};\vec{y}) = h(u_1(\vec{x};), u_2(\vec{x};); v_1(\vec{x};\vec{y}), v_2(\vec{x};\vec{y}))$, where h, u_1, u_2, v_1, and v_2 are linear-time, then f can be computed in linear time with the aid of constantly many extra tapes: one for each intermediate result and one to "back up" $\vec{x}$ and $\vec{y}$. The algorithm follows.

algorithm compute_composed_$f(\vec{x};\vec{y})$

Copy $\vec{x}$ and $\vec{y}$ to a backup tape.
Compute $u_1(\vec{x};)$ and put the result on a separate tape.
Restore $\vec{x}$ from backup, compute $u_2(\vec{x};)$, and put the result on another tape.
Restore $\vec{x}$ from backup, compute $v_1(\vec{x};\vec{y})$, and put the result on another tape.
Restore $\vec{x}$ and $\vec{y}$ from backup, compute $v_2(\vec{x};\vec{y})$, and put the result on another tape.
Finally, invoke a machine for h on the four result tapes (plus however many blank tapes the h machine demands).

In all we have used five auxiliary tapes, plus the maximum of the constant numbers of tapes needed by the u_1, u_2, v_1, v_2, and h machines. Each step of the algorithm has taken only linear time.

Now suppose f is defined by very safe recursion on notation, as

$$\begin{aligned} f(0,x_1,\ldots x_r;y_1,\ldots y_s) &= g(\vec{x};\vec{y}) \\ f(z,x_1,\ldots x_r;y_1,\ldots y_s) &= h\left(;z,\vec{x},\vec{y},f\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right)\right) \quad \text{for } z>0. \end{aligned}$$

(We shall deal with the case of simultaneous recursion shortly, after describing how to handle a single recursion.)

By our earlier comments, h is a constant-time function, and by the induction hypothesis, g can be computed in linear time. Then we can evaluate f in linear time, using the maximum number of tapes necessary to compute g and h, plus one for backup purposes and three more named *BaseValue*, *append*, and *counter*. As in the proof of Lemma 11, we distinguish between a "long" part of the intermediate result, which may be on any one of the tapes $\vec{x},\vec{y},z$,*BaseValue* or may be the empty string, and a "short" part which is kept in *append* (we treat *append* rather like a read-through cache: whenever we want to use any data from one of the tapes $\vec{x},\vec{y},z$,*BaseValue* we first make sure the data are in *append*). In order to preserve the values of $\vec{x}$, $\vec{y}$, and z for later use, we never write on these tapes; instead, if the function h ever tries to delete or change some of their low bits, we simply count how many bits have been thus affected in unary on *counter*. At the end of the computation, the function value is one of the tapes, minus its low $|counter|$ bits, plus the contents of *append*.

Let t_h be a bound on the time to compute h (and therefore on the number of low bits of h's parameters that can be examined in the course of computing h).

algorithm compute_recursive_$f(z, \vec{x}, \vec{y})$

Declare a finite variable *WhichTape* to take the values $\{x_1, \ldots x_r, y_1, \ldots y_s, z, \mathit{BaseValue}, \lambda\}$.

Declare the finite variables $\mathit{LocalX}_1, \ldots \mathit{LocalX}_r$, $\mathit{LocalY}_1, \ldots \mathit{LocalY}_s$, *LocalZ* to be at most t_h bits each.

Copy $\vec{x}$ and $\vec{y}$ to a backup tape.

Set *BaseValue* to the result of computing $g(\vec{x}; \vec{y})$.

Restore the values of $\vec{x}$ and $\vec{y}$ from backup.

Set *WhichTape* to indicate *BaseValue*.

Set both *counter* and *append* to be empty.

In linear time, move the head on the z tape to the left-hand end.

While the head on the z tape has not passed the right-hand end,

- Copy the t_h bits from the z head leftward (including blanks if this goes past the left end of the z tape) to *LocalZ*.
- Copy the t_h low bits from each x tape and from each y tape to the corresponding *LocalX*'s and *LocalY*'s.
- **Assertion:** the head on the *append* tape is at the right-hand end.
- **Repeat** t_h times:
  - Move the head on the *append* tape left one square.
  - **If** the *append* head has passed the left end of the tape,
    - Move the head on the tape indicated by *WhichTape* left.
    - Copy the bit from the tape indicated by *WhichTape* to *append* (if *WhichTape* $= \lambda$, copy a blank).
    - Increment *counter*.
- Move the head on the *append* tape back to the right-hand end.
- Compute $h(; \mathit{LocalZ}, \mathit{Local}\vec{X}, \mathit{Local}\vec{Y}, \mathit{append})$, possibly modifying any or all of the arguments.
- Move the head on the *append* tape back to the right-hand end (it can have moved at most t_h squares in the computation of h).
- Branch by cases on the final state of the h machine:
  - **If** h returns its value on tape x_i, y_i, or z,
    - Set *WhichTape* to indicate x_i, y_i, or z respectively.
    - Set *counter* to be exactly t_h bits long.
    - Set *append* to be exactly the (modified) value of LocalX_i, LocalY_i, or *LocalZ* respectively.
  - **Else** /* if h returns its value on the tape corresponding to its final parameter, */ do nothing.
- Move the head on the z tape one square to the right.
- **If** *WhichTape* indicates z, increment *counter*.

Move the head on the tape indicated by *WhichTape* to the right end.

Delete the low $|\mathit{counter}|$ bits of the tape indicated by *WhichTape*.

Concatenate *append* to the end of the tape indicated by *WhichTape*.

If we take our tapes to be delimited by endmarkers at both ends, we can empty a tape by writing two adjacent endmarkers in constant time, so we can set a tape to any bounded-length value in constant time. Copying bounded-length values, and looping t_h times, are likewise constant-time operations, so the entire body of the **while** loop is constant-time and so (by Lemma 11) the **while** loop takes linear time in all. The work done before and after the **while** loop is likewise linear-time, so the whole algorithm takes linear time to compute $f(z, \vec{x}; \vec{y})$.

Now suppose the function in question is one of a tuple defined by *simultaneous*, very safe recursion on notation, e.g. for $1 \leq i \leq k$

$$\begin{aligned} f_i(0, \vec{x}; \vec{y}) &= g_i(\vec{x}; \vec{y}) \\ f_i(z, \vec{x}; \vec{y}) &= h_i\left(; z, \vec{x}, f_1\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right), \ldots f_k\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right), \vec{y}\right) \quad \text{for } z > 0 \end{aligned}$$

We simulate this recursion exactly as before, except that corresponding to each $1 \leq i \leq k$ are a distinct set of *BaseVal*, *append*, *counter*, *WhichTape*, $\mathit{Local}\vec{X}$, $\mathit{Local}\vec{Y}$, and *LocalZ* variables. We let t_h be the maximum of $t_{h_1}, \ldots t_{h_k}$. Inside the main loop, instead of computing $h(; \mathit{LocalZ}, \mathit{Local}\vec{X}, \mathit{Local}\vec{Y}, \mathit{append})$, we compute for each possible $1 \leq i \leq k$ the value $h_i(; \mathit{LocalZ}_i, \mathit{Local}\vec{X}_i, \mathit{Local}\vec{Y}_i, \mathit{append}_1, \ldots \mathit{append}_k)$. ∎

Thus any function in $\mathcal{D}$ can be computed by a linear-time, multi-tape TM. To complete the proof of Theorem 10, we must show that the computation of an arbitrary linear-time, multi-tape TM can be simulated within $\mathcal{D}$.

Lemma 14 *If the function $f(\vec{x})$ is computable in linear time by a multi-tape TM, then $f(\vec{x};) \in \mathcal{D}$.*

PROOF. We represent a configuration of a k-tape Turing machine by a $2k+1$-tuple of the form $\langle L_1, R_1, \ldots L_k, R_k, q\rangle$, where L_i contains the contents of tape i to the left of the head (the closest tape square to the head stored in the least significant bits), R_i contains the contents of tape i to the right of the head (likewise), and q represents the state of the machine. We define $2k+1$ functions $NEXT_{L_1}, \ldots NEXT_q$, each computing the new value of one parameter from the old configuration. Each such function can be defined by composition alone from BASE functions, and hence needs no normal parameters. Since the time bound $t(|\vec{x}|)$ is linear, Lemma 12 assures us there is a function $T(\vec{x};) \in \mathcal{D}$ such that $|T(\vec{x})| \geq t(|\vec{x}|)$ for all $\vec{x}$. We then apply simultaneous very safe recursion on the $2k+1$ functions described above to this value, defining $2k+1$ functions $f_{L_1}, \ldots f_q$ such that, for example, $f_{R_1}(z, \vec{x};)$ computes the contents of tape 1 to the right of the head at time $|z|$. Assuming without loss of generality that the multi-tape TM writes the function value on, say, tape 4, with the head at the right-hand end, we simply let $f(\vec{x};) = f_{L_4}(T(\vec{x};), \vec{x};)$.

This completes the proof of Lemma 14 and therefore of Theorem 10. ∎

We next prove the similar Theorem 16 characterizing functions computable by multi-tape TM in quasilinear time. The results are stated for two-tape machines, since in the quasilinear-time setting two tapes are as good as a thousand.

Definition 15 *The function class $\mathcal{D}'$ is the minimum closure of the set BASE under the operations of length composition and very safe recursion on notation.*

Theorem 16 *A function $f(\vec{x};)$ is in $\mathcal{D}'$ iff it is computable on a two-tape TM in time $O(|\vec{x}| \cdot ||\vec{x}||^{O(1)})$.*

To prove this we begin with the

Lemma 17 *For any natural numbers a, b, c there is a function $f(x;) \in \mathcal{D}'$ such that for all x,*

$$|f(x;)| = a \cdot |x| \cdot ||x||^b + c.$$

Thus any desired quasilinear growth rate is attained by a function in $\mathcal{D}'$.

PROOF. Recall the function *ReverseConcat* from the proof of Lemma 12, which has the property that $|ReverseConcat(x; y)| = |x| + |y|$. By Length Composition we can define the function

$$ReverseLengthConcat(; x, y) = ReverseConcat(|x|; y)$$

which has the property that $|ReverseLengthConcat(; x, y)| = ||x|| + |y|$. We can then perform very safe recursion on notation to define a function

$$MultiplyLength(z; x) = \begin{cases} 0 & \text{if } z = 0 \\ ReverseLengthConcat(; x, MultiplyLength(Half(; z); x)) & \text{otherwise} \end{cases}$$

which has the property that $|MultiplyLength(z; x)| = |z| \cdot ||x||$. We can then define functions

$$\begin{aligned} f_1(x;) &= MultiplyLength(x; x), \\ f_2(x;) &= MultiplyLength(f_1(x;); x), \\ &\vdots \\ f_b(x;) &= MultiplyLength(f_{b-1}(x;); x) \end{aligned}$$

so that $|f_b(x;)| = |x| \cdot ||x||^b$. By further composition with suitable numbers of *ReverseConcat* and constant functions we can produce any desired growth rate of the form $a \cdot |x| \cdot ||x||^b + c$. ∎

The proof that $\mathcal{D}$ can compute the next configuration of a TM by functions with no normal parameters made no use of the linear time bound, so it still applies in the quasilinear setting. Iterating this next-configuration function (or, more properly, tuple of functions) by simultaneous very safe recursion on a quasilinear-length argument allows us to simulate any desired quasilinear-time TM, proving one direction of Theorem 16. For the other direction, we first prove growth rate bounds on functions in $\mathcal{D}'$.

Lemma 18 *If $f(\vec{x}; \vec{y}) \in \mathcal{D}'$, then there is a polynomial p_f such that for all values $\vec{x}$ and $\vec{y}$ we have*

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

Corollary 19 *If $f(; \vec{y}) \in \mathcal{D}'$, then there is a polynomial p_f such that for all values $\vec{x}$ and $\vec{y}$ we have*

$$|f(; \vec{y})| \leq p_f(||\vec{y}||) + \max(|\vec{y}|).$$

PROOF. The base functions all clearly satisfy this requirement; indeed, the result of any base function is at most *one* bit longer than its longest argument.

Suppose f is defined by length composition, i.e.

$$f(\vec{x}; \vec{y}) = g(u(\vec{x}, |\vec{y}|;); v(\vec{x}; \vec{y}))$$

(we assume there is only one u and only one v function for simplicity; the general case is more tedious but presents no additional difficulties.) By the induction hypothesis we can assume the existence of polynomials p_g, p_u, and p_v such that

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

Then

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

as desired.

Suppose now that f is one of a tuple of functions $f_1,\dots f_r$ defined by simultaneous very safe recursion on notation, i.e. for each $1\leq i\leq r$

$$\begin{aligned} f_i(0,\vec{x};\vec{y}) &= g_i(\vec{x};\vec{y}) \\ f_i(z,\vec{x};\vec{y}) &= h_i\left(;z,\vec{x},\vec{y},\vec{f}\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right)\right) \quad \text{for } z>0 \end{aligned}$$

By the induction hypothesis, there are polynomials $p_{g_1},\dots p_{g_r}$, and $p_{h_1},\dots p_{h_r}$ such that for all $1\leq i\leq r$,

$$\begin{aligned} |g_i(\vec{x};\vec{y})| &\leq (\max(|\vec{x}|)+1)\cdot p_{g_i}(||\vec{x}||,||\vec{y}||)+\max(|\vec{y}|) \\ |h_i(;z,\vec{x},\vec{y},\vec{u})| &\leq p_{h_i}(||z||,||\vec{x}||,||\vec{y}||,||\vec{u}||)+\max(|z|,|\vec{x}|,|\vec{y}|,|\vec{|}). \end{aligned}$$

Note that each function h_i takes no normal parameters and therefore adds at most polylog-many bits to the length of its longest argument. Now let $p_h=\sum_{i=1}^{r}p_{h_i}$ and $p_g=\sum_{i=1}^{r}p_{g_i}$. For each function f_i, and $z>0$, we have that

$$\begin{aligned} |f_i(z,\vec{x};\vec{y})| &\leq p_h\left(||z||,||\vec{x}||,||\vec{y}||,\left|\left|\vec{f}\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right)\right|\right|\right)+\max\left(|z|,|\vec{x}|,|\vec{y}|,\left|\vec{f}\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right)\right|\right) \\ &\leq p_h\left(||z||,||\vec{x}||,||\vec{y}||,\left|\left|\vec{f}\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right)\right|\right|\right)\cdot|z|+\max(|z|,|\vec{x}|,|\vec{y}|,|\vec{f}(0,\vec{x};\vec{y})|) \end{aligned}$$

We prove by induction the existence of a polynomial p_f such that for all $1\leq i\leq r$ and all $z,\vec{x},\vec{y}$,

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

Since $f_i(0,\vec{x};\vec{y})=g_i(\vec{x};\vec{y})$ for all i, the polynomial p_g works for all $\vec{x},\vec{y}$ so long as $z=0$. Now assume inductively that some polynomial p_f satisfies the lemma for all $\vec{x},\vec{y}$ and all $z<z_0$. We shall find sufficient conditions on p_f to still satisfy the lemma at z_0; then if a particular polynomial p_f meets all the conditions, the induction proof goes through and the polynomial satisfies the requirements of the lemma.

$$\begin{aligned} &|f_i(z_0,\vec{x};\vec{y})|\leq \\ &\quad\leq\ p_h\left(||z_0||,||\vec{x}||,||\vec{y}||,\left|\left|\vec{f}\left(\left\lfloor\frac{z_0}{2}\right\rfloor,\vec{x};\vec{y}\right)\right|\right|\right)\cdot|z_0| \\ &\qquad+\max(|z_0|,|\vec{x}|,|\vec{y}|,|\vec{f}(0,\vec{x};\vec{y})|) \end{aligned}$$

$$
\begin{aligned}
&\leq\; p_h\left(||z_0||,||\vec{x}||,||\vec{y}||,\left|p_f\left(\left|\left|\left\lfloor\frac{z_0}{2}\right\rfloor\right|\right|,||\vec{x}||,||\vec{y}||\right)\cdot\max(|z_0|,|\vec{x}|)+\max(|\vec{y}|)\right|\right)\cdot|z_0| \\
&\quad+\max(|z_0|,|\vec{x}|,|\vec{y}|,|\vec{f}(0,\vec{x};\vec{y})|) \\
&\leq\; p_h\left(||z_0||,||\vec{x}||,||\vec{y}||,\left|\mathrm{tr}(p_f)\cdot||\max(z_0,\vec{x},\vec{y})||^{\deg(p_f)}\cdot\max(|z_0|,|\vec{x}|)+\max(|\vec{y}|)\right|\right)\cdot|z_0| \\
&\quad+\max(|z_0|,|\vec{x}|,|\vec{y}|,|\vec{f}(0,\vec{x};\vec{y})|) \\
&\leq\; p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+\deg(p_f)\cdot|||\max(z_0,\vec{x},\vec{y})|||+||\max(z_0,\vec{x})||+||\max(\vec{y})||)\cdot|z_0| \\
&\quad+\max(|z_0|,|\vec{x}|,|\vec{y}|,|\vec{f}(0,\vec{x};\vec{y})|) \\
&\leq\; p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+(2+\deg(p_f))\cdot||\max(z_0,\vec{x},\vec{y})||)\cdot|z_0| \\
&\quad+\max(|z_0|,|\vec{x}|,|\vec{y}|,|\vec{f}(0,\vec{x};\vec{y})|) \\
&\leq\; p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+(2+\deg(p_f))\cdot||\max(z_0,\vec{x},\vec{y})||)\cdot|z_0| \\
&\quad+\max(|z_0|,|\vec{x}|,|\vec{y}|,(\max(|\vec{x}|)+1)\cdot p_g(||\vec{x}||,||\vec{y}||)+\max(|\vec{y}|)) \\
&\leq\; p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+(2+\deg(p_f))\cdot||\max(z_0,\vec{x},\vec{y})||)\cdot|z_0| \\
&\quad+(\max(|z_0|,|\vec{x}|)+1)\cdot p_g(||\vec{x}||,||\vec{y}||)+\max(|\vec{y}|) \\
&\leq\; (p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+(2+\deg(p_f))\cdot||\max(z_0,\vec{x},\vec{y})||)+p_g(||\vec{x}||,||\vec{y}||)) \\
&\quad\cdot(\max(|z_0|,|\vec{x}|)+1)+\max(|\vec{y}|)
\end{aligned}
$$

It would suffice, therefore, to find a p_f such that, for all z_0, $\vec{x}$, $\vec{y}$,

$$p_f(||z_0||,||\vec{x}||,||\vec{y}||)\geq p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+(2+\deg(p_f))\cdot||\max(z_0,\vec{x},\vec{y})||)+p_g(||\vec{x}||,||\vec{y}||).$$

Such a polynomial can be constructed by an iterative algorithm as follows. Start with some polynomial of degree at least $\max(\deg(p_h),\deg(p_g))$, such that each nonzero term of either $p_g(||\vec{x}||,||\vec{y}||)$ or $p_h(||z_0||,||\vec{x}||,||\vec{y}||,||z_0||+||\vec{x}||+||\vec{y}||)$ has a corresponding nonzero term in the new polynomial. Then repeatedly double all its coefficients until the inequality is satisfied (in the sense that each coefficient of the left side is greater than or equal to the corresponding coefficient of the right side). This *will* occur eventually: doubling all the coefficients of p_f m times (i.e. multiplying each coefficient by 2^m) has the effect of multiplying the left hand side by 2^m, while on the right hand side, it has no effect on $\deg(p_f)$ and only adds m to $|\mathrm{tr}(p_f)|$, and therefore it adds to the right hand side a polynomial bounded by a constant times

$$\binom{k}{1}m||\max(z_0,\vec{x},\vec{y})||^{k-1}+\binom{k}{2}m^2||\max(z_0,\vec{x},\vec{y})||^{k-2}+\cdots+\binom{k}{k}m^k$$

(where $k=\deg(p_h)$.) Each coefficient of this polynomial is only polynomial in m, while each coefficient of the polynomial on the left is exponential in m, so there exists an m beyond which each coefficient on the left is greater than or equal to the corresponding coefficient on the right, and hence that the polynomial p_f reached after m doublings has the property that, for all z_0, $\vec{x}$, and $\vec{y}$,

$$p_f(||z_0||,||\vec{x}||,||\vec{y}||)\geq p_h(||z_0||,||\vec{x}||,||\vec{y}||,|\mathrm{tr}(p_f)|+(2+\deg(p_f))\cdot||\max(z_0,\vec{x},\vec{y})||)+p_g(||\vec{x}||,||\vec{y}||),$$

as desired.

Thus simultaneous very safe recursion on notation preserves quasilinear growth rate, and Lemma 18 is proven. ∎

We now define an enhanced sort of TM whose computation corresponds naturally to the functions in $\mathcal{D}'$; later we show that this model is equivalent to the usual model for purposes of defining quasilinear time.

Definition 20 *A* measured Turing machine *is a multi-tape TM with tape alphabet containing at least the symbols* $\{0, 1, \Box\}$ *with the additional ability to, in a single step, replace the entire contents of any given tape by the length of those contents in binary.*

Lemma 21 *If* $f(\vec{x}; \vec{y}) \in \mathcal{D}'$*, then there is a polynomial* p_f *and a measured TM* M *which, for all values* $\vec{x}$ *and* $\vec{y}$*, computes* $f(\vec{x}; \vec{y})$ *in time at most*

$$(\max(|\vec{x}|) + 1) \cdot p_f(||\vec{x}||, ||\vec{y}||).$$

PROOF. The lemma is clearly true for the BASE functions; in fact, each BASE function takes only *constant* time on a measured TM.

Suppose $f(\vec{x}; \vec{y})$ is defined by length composition, e.g.

$$f(\vec{x}; \vec{y}) = g(u(\vec{x}, |\vec{y}|;); v(\vec{x}; \vec{y})).$$

We can compute f on a measured TM as follows.

algorithm simulate_composition_on_MTM $(\vec{x}; \vec{y})$

Make backup copies of $\vec{x}$ and $\vec{y}$ on a spare tape, in linear time.
Compute $v(\vec{x}; \vec{y})$ and put the result on a spare tape, in time
$(\max(|\vec{x}|, |\vec{y}|) + 1) \cdot p_v(||\vec{x}||, ||\vec{y}||)$.
Restore $\vec{x}$ and $\vec{y}$ from backup, in linear time.
Replace each parameter y by its length, in constant time on a measured TM.
Compute $u(\vec{x}, |\vec{y}|;)$ and put the result on a spare tape, in time
$(\max(|\vec{x}|, ||\vec{y}||) + 1) \cdot p_u(||\vec{x}||, |||\vec{y}|||)$.
Compute g of the two result tapes, in time
$(\max(|u(\vec{x}, |\vec{y}|;)|, |v(\vec{x}; \vec{y})|) + 1) \cdot p_g(||u(\vec{x}, |\vec{y}|;)||, ||v(\vec{x}; \vec{y})||)$, which is quasilinear because by Lemma 18, u and v have only quasilinear growth rate themselves.

Now suppose f is one of a tuple of functions $f_1, \dots f_r$ defined by simultaneous very safe recursion on notation, i.e. for each $1 \le i \le r$

$$\begin{aligned} f_i(0, \vec{x}; \vec{y}) &= g_i(\vec{x}; \vec{y}) \\ f_i(z, \vec{x}; \vec{y}) &= h_i\left(; z, \vec{x}, \vec{y}, \vec{f}\left(\left\lfloor \frac{z}{2} \right\rfloor, \vec{x}; \vec{y}\right)\right) \quad \text{for } z > 0. \end{aligned}$$

Then we can compute f with a measured TM in quasilinear time by an algorithm very similar to that of Lemma 13, but with each function h_i allowed to take polylog time and therefore to use and affect the low polylogarithmically-many bits of each argument. The values t_h in the previous algorithm are no longer constant-bounded but rather may range up to a polylog function in the lengths of the inputs, and the variables $Local\vec{X}$, $Local\vec{Y}$, and $LocalZ$ are no longer constant-bounded but rather polylog-length. Note that we can compute bounds for t_h and the lengths of these local variables because polylog functions are time-constructible on a measured TM.

Thus the desired time bounds are preserved by both length composition and simultaneous very safe recursion on notation; the inductive proof of Lemma 21 follows. ∎

Lemma 22 *If* M *is a measured TM which, on input* $\vec{x}$*, always terminates within time* $t(\vec{x})$*, and if* $t(\vec{x}) = \Omega(|\vec{x}|)$*, then there is an ordinary TM* M' *computing the same function or relation in time* $O(t(\vec{x}) \cdot |t(\vec{x})|)$*.*

Corollary 23 *Quasilinear time on an ordinary TM is exactly the same class as quasilinear time on a measured TM.*

PROOF of Corollary from Lemma. Since a measured TM can do anything an ordinary TM can, quasilinear time on a measured TM trivially contains QL. On the other hand, if a measured TM M computes some function in time bound $O(n \log^k(n))$, the lemma tells us we can simulate it on an ordinary TM in time $O\left(n \log^k(n) \left(\left|n \log^k(n)\right|\right)\right)$, which is $O(n \log^{k+2}(n))$ and hence still quasilinear. ∎

PROOF of Lemma 22. Given a k-tape measured TM M, we construct a $2k+1$-tape TM M' to simulate M as follows. We assume without loss of generality that M has left and right end-markers on each of its tapes. For each tape x_i of M there are two tapes y_i and z_i in M'. Machine M' assumes its input to be on the y_i tapes. When machine M' starts, it computes the length of each input tape y_i (in quasilinear time) and writes the result in binary on tape z_i. Thereafter whatever machine M does to tape x_i, machine M' will do to tape y_i, except that whenever machine M moves an endmarker on a tape x_i, lengthening or shortening the contents of that tape, machine M' not only makes the same move on tape y_i but also increments or decrements the binary number on tape z_i, taking logarithmic time. Whenever machine M invokes its ability to compute the length of a tape x_i, machine M' copies the contents of tape z_i to tape y_i, then computes the length of the result and stores it in binary on tape z_i; both of these operations take polylog time.

Thus machine M' simulates the actions of machine M, using an additional quasilinearly-much time at the beginning and at most $O(\log(S))$ time for each step, where S is the space requirement of machine M (which, by Lemma 18, is itself at most quasilinear), and thus machine M' too terminates in quasilinear time. ∎

4 The Sharply Bounded Hierarchy

We start with some definitions of the relational Sharply Bounded Hierarchy discussed more fully in [BG92], then present the corresponding function class.

Definition 24 *Let p be a polynomial and $c \geq 0$ a constant. A relation $R(\vec{x}) \subseteq \mathbf{N}^k$ is defined by* quasilinear existential quantification *from $S(\vec{x}, y) \subseteq \mathbf{N}^{k+1}$ if for all $\vec{x}$,*

$$R(\vec{x}) \iff (\exists y < \max(|\vec{x}|) \cdot p(\max(||\vec{x}||)) + c) \; S(\vec{x}, y).$$

The definition of quasilinear universal quantification *is similar.*

Definition 25 *A relation $R(\vec{x}) \subseteq \mathbf{N}^k$ is in the* sharply bounded hierarchy over QL *(written $SBH(QL)$ for short) if R is defined by finitely many applications of quasilinear quantification from a relation $S(\vec{x}, \vec{y}) \in QL$.*

One can similarly define $SBH(lin)$ by linear quantification over linear time, $SBH(\mathcal{P})$ by polynomial quantification over $\mathcal{P}$ (although this is uninformative, since $SBH(\mathcal{P}) = \mathcal{P}$), and so on. So long as the underlying time class is at least QL, whether the quantifiers are linear, quasilinear, or polynomial length makes no difference: since multiplication and addition are QL functions, quantification up to a d-degree polynomial can be simulated by treating d quantified variables as indices into a d-dimensional row-major array.

In this section, however, we shall discuss *function* classes that naturally correspond to $SBH(QL)$ and $SBH(lin)$. We present an alternating Turing machine model that computes functions, such that the 0/1-valued functions computable in the model are exactly the $SBH(lin)$ or $SBH(QL)$ relations described above. We then modify the safe-parameters characterizations $\mathcal{D}$ and $\mathcal{D}'$ to characterize exactly these functions.

4.1 A Machine Model

Traditional forms of alternating TM's compute relations, not functions. In parallel complexity with polynomial size bounds, this is no handicap because a function is easily computable from its bit-graph. In our setting, however, there is no reason to believe that a function whose bit-graph is in QL is itself in QL. So we modify the ATM model to compute functions in a natural way. An additional modification allows each guess to range over (quasi)linearly-many values rather than 0/1, so we need only count guesses rather than alternations.

Definition 26 *A* length-branching function machine *is a multi-tape TM whose states are partitioned into universal, existential, and deterministic states. Universal and existential states are collectively referred to as nondeterministic states. Each deterministic state has at most one next state; if it has none, it is called a halting state. Associated with each nondeterministic state q are three next states, referred to as $TEST(q)$, $YES(q)$, and $NO(q)$.*

We define a mapping F from machine configurations to strings as follows:

- *If configuration C is in a halting state, $F(C)$ is the contents of tape 1.*
- *If configuration C is in a deterministic but non-halting state, $F(C)$ is F of the next configuration.*
- *If configuration C is in existential state q, let $TEST(C,m)$ denote the configuration identical to C but in state $TEST(q)$ and with the head on tape 1 m squares from the right-hand end, let $YES(C)$ be the configuration identical to C but in state $YES(q)$, and let $NO(C)$ be the configuration identical to C but in state $NO(q)$. If there is a value m such that square m of tape 1 in configuration C is nonblank and $F(TEST(C,m)) = 1$, then $F(C) = F(YES(C))$; otherwise $F(C) = F(NO(C))$.*
- *If configuration C is in universal state q, define $TEST(C,m)$, $YES(C)$, and $NO(C)$ as above. If all values m such that square m of tape 1 in configuration C is nonblank satisfy $F(TEST(C,m)) = 1$, then $F(C) = F(YES(C))$; otherwise $F(C) = F(NO(C))$.*

The function computed by the machine on input $\vec{x}$ is F of its starting configuration on that input.

A computation of a length-branching function machine may be viewed as a tree in which each nondeterministic configuration has $2 + |y_1|$ children (where y_1 is the contents of tape 1 in that configuration) and each deterministic configuration has at most one. The *quantifier depth* of such a computation is the maximum number of nondeterministic configurations along any branch of the tree, while the *time usage* of such a computation is the maximum distance from the root to any leaf. (A somewhat similar scheme was used in [Bus88], with an arbitrary oracle in place of the $TEST$ subtree, and with the intention of controlling space usage rather than counting nondeterministic guesses.)

Definition 27 *An $SBH(lin)$-time function is any function computed by a length-branching function machine with constant quantifier depth and linear time usage.*

An $SBH(QL)$-time function is any function computed by a length-branching function machine with constant quantifier depth and quasilinear time usage.

Lemma 28 *A 0/1-valued function is in $SBH(lin)$ iff it is the characteristic function of an $SBH(lin)$ relation, and similarly for $SBH(QL)$.*

PROOF. We give a proof for $SBH(lin)$; the proof for $SBH(QL)$ is similar.

Suppose relation $R(\vec{x})$ is an $SBH(lin)$ relation. Then it can be written in the form

$$(Q_1 y_1 < l_1(|\vec{x}|)) \cdots (Q_k y_k < l_k(|\vec{x}|)) S(\vec{x}, \vec{y})$$

where each Q_i is either $\exists$ or $\forall$, each l_i is a linear function, and S is a linear-time relation. We can compute its characteristic function with a length-branching function machine as follows: for each quantifier $(Qy_i < l_i(|\vec{x}|))$ in turn, construct a value of length $l_i(|\vec{x}|)$ on an additional tape, guess a head position on that tape, and in time $O(l_i(|\vec{x}|))$ count how far the head is from the right-hand end, writing the result in binary on another tape henceforth known as y_i. Then invoke the deterministic, linear-time machine for S, replacing "accept" states by "write 1 and halt," and "reject" states by "write 0 and halt."

Conversely, suppose the 0/1-valued function $f(\vec{x})$ is computed by a length-branching function machine M with quantifier depth c and linear time bound. The computation tree of M has quantifier depth c and branching factor three, so it contains at most 3^c nondeterministic configurations in all (for any given input). A nondeterministic configuration in an alternating function machine behaves like an "if ... then ... else ... " and can be replaced by the DNF equivalent, which comprises a constant number of quantifiers followed by deterministic work. We then move all the quantifiers to the beginning of the computation to put the entire computation in the form of an $SBH(lin)$ relation. Whenever machine M enters a nondeterministic state, it essentially makes a nondeterministic choice $1 \leq m \leq |t_1|$, where t_1 is the contents of tape 1. Since machine M has only linear time in all, the length of tape 1 is a linear-bounded, time-constructible function $l(\vec{x})$ of the lengths of the inputs. Each existential configuration then computes something of the form "if $(\exists y < l(|\vec{x}|)) t(\vec{x}, y)$ then $f(\vec{x})$ else $g(\vec{x})$," where the functions t, f, and g are all 0/1-valued. This construct can be replaced by the constant-depth formula

$$(((\exists y < l(|\vec{x}|)) t(\vec{x}, y)) \wedge f(\vec{x})) \vee ((\neg(\exists y < l(|\vec{x}|)) t(\vec{x}, y)) \wedge g(\vec{x})),$$

or equivalently by

$$(\exists a \leq 1)(\forall b \leq 1)(\exists y < l(|\vec{x}|)) \begin{cases} t(\vec{x}, y) & \text{if } a = b = 0 \\ f(\vec{x}) & \text{if } a = 0 \wedge b = 1 \\ \neg t(\vec{x}, y) & \text{if } a = 1 \wedge b = 0 \\ g(\vec{x}) & \text{if } a = b = 1. \end{cases}$$

A similar construction can be carried out for universal configurations. Thus we have replaced each nondeterministic branch in the computation tree of M by three nested quantifiers (two of which are constant-bounded, incidentally) followed by deterministic work. Since the computation tree of M has only quantifier depth c, and a branching factor of three, it contains at most 3^c nondeterministic branches in all; after the above transformation we have quantifier depth at most $3 \cdot 3^c = 3^{c+1}$. Furthermore, the constantly-many quantifiers can now be moved ahead of all the deterministic work, yielding a string of constantly many quantifiers followed by a linear-time deterministic relation, i.e. a relation in $SBH(lin)$.
∎

4.2 A Function-Algebra Model

We now present modifications of the function algebras $\mathcal{D}$ and $\mathcal{D}'$ that characterize the functions in $SBH(lin)$ and $SBH(QL)$ respectively.

Definition 29 *A function f is defined by* sharply-bounded zero-detection *from g if*

$$f(z,\vec{x};\vec{y}) = \begin{cases} 0 & \textit{if } (\forall w < |z|)(g(w,\vec{x};\vec{y}) = 0) \\ 1 & \textit{if } (\exists w < |z|)(g(w,\vec{x};\vec{y}) > 0) \end{cases}.$$

Definition 30 *The function class $\mathcal{D}^{zd}$ is the closure of the BASE functions under safe composition, very safe recursion on notation, and sharply-bounded zero-detection.*

The function class $\mathcal{D}'^{zd}$ is the closure of the BASE functions under length composition, very safe recursion on notation, and sharply-bounded zero-detection.

Theorem 31 *A function $f(\vec{x};\vec{y})$ is in $\mathcal{D}^{zd}$ iff $f(\vec{x})$ is in $SBH(lin)$.*

Theorem 32 *A function $f(\vec{x};\vec{y})$ is in $\mathcal{D}'^{zd}$ iff $f(\vec{x})$ is in $SBH(QL)$.*

Proof. We prove the linear version of the theorem; the quasilinear version is similar.

The containment $SBH(lin) \subseteq \mathcal{D}^{zd}$ is easy. If q is a existential state in an $SBH(lin)$ machine, let $g(w,\vec{x},\vec{y})$ be the function computed by its $TEST$ branch, and assume without loss of generality that this function is 0/1-valued (since all that matters is whether its value is 1 or not). Define

$$h(z,\vec{x},\vec{y}) = \begin{cases} 0 & \text{if } (\forall w < |z|)(g(w,\vec{x},\vec{y}) = 0) \\ 1 & \text{if } (\exists w < |z|)(g(w,\vec{x},\vec{y}) > 0) \end{cases}.$$

Clearly if we plug in the contents of tape 1 for z, h will compute the value of the existential test; we can then compose this with a *Cond*function to return either the result of the YES branch or that of the NO branch. With the aid of a few *Cond*and constant functions we can negate any 0/1-valued function, so the ability to simulate existential tests also allows us to simulate universal tests.

We have used one application of sharply-bounded zero-detection for each of the finitely many nondeterministic configurations in a computation tree of the $SBH(lin)$ machine. Thus any function computable in $SBH(lin)$ has a finite definition in $\mathcal{D}^{zd}$.

Conversely, let $f \in \mathcal{D}^{zd}$. If $f \in \mathcal{D}$, Theorem 10 tells us f can be computed in linear time, and hence certainly in $SBH(lin)$. If the last step of f's definition is a composition, we're done because $SBH(lin)$ is closed under composition.

If f is one of a tuple of functions defined by simultaneous very safe recursion on notation, e.g.

$$\begin{aligned} f_i(0,\vec{x};\vec{y}) &= g_i(\vec{x};\vec{y}) \\ f_i(z,\vec{x};\vec{y}) &= h_i\left(;z,\vec{x},f_1\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right),\dots f_k\left(\left\lfloor\frac{z}{2}\right\rfloor,\vec{x};\vec{y}\right),\vec{y}\right) \quad \text{for } z > 0, \end{aligned}$$

we note that since each function h_i contains no normal parameters, its definition cannot have involved zero-detection. Thus f is computable with no more nondeterminism than is g.

Finally, suppose f is defined by sharply-bounded zero-detection:

$$f(z,\vec{x};\vec{y}) = \begin{cases} 0 & \text{if } (\forall w < |z|)(g(w,\vec{x};\vec{y}) = 0) \\ 1 & \text{if } (\exists w < |z|)(g(w,\vec{x};\vec{y}) > 0) \end{cases}.$$

Assume without loss of generality that g is 0/1-valued (if not, compute g, examine the result, and if it's not zero erase the whole result tape and write a 1). The induction hypothesis assures us there is an $SBH(lin)$ machine $M(w,\vec{x},\vec{y})$ computing g. We can then compute f with an $SBH(lin)$ machine by copying z onto tape 1 (if tape 1 contains something important, we can first copy it somewhere else and then restore it), then performing an existential branch to test whether $(\exists w < |z|)(g(w,\vec{x};\vec{y}) > 0)$ and writing a 0 or a 1 as appropriate. ∎

5 Conclusions and Open Problems

The safe-parameters characterizations of (quasi)linear-time functions and their SBH closures make no explicit mention of linear or quasilinear bounds. This argues for the naturalness of the classes by comparison with (say) quadratic time, which appears to have no such clean characterization.

A closely-related, and equally interesting, class is $QL[O(\log(n))]$, the class of relations computable in quasilinear time with $O(\log(n))$ bits of nondeterminism but no bound on alternations. (We could of course define a corresponding function class as well.) This class is still within polynomial time, and it is easy to show that it contains ALOGTIME, so if $QL[O(\log(n))] \subset \mathcal{P}$ it implies *a fortiori* that uniform $NC^1 \neq \mathcal{P}$. On the other hand, if $QL[O(\log(n))] = \mathcal{P}$, it implies that all polynomial-time relation computations can be parallelized, perhaps not down to log or polylog time, but down to quasilinear time with polynomially many processors. This would answer a question of [Con94] by showing that essentially all problems in $\mathcal{P}$ have polynomial parallel speedup, and would have tremendous practical implications.

Another class, apparently between $SBH(QL)$ and $QL[O(\log(n))]$, would allow a deterministic algorithm to invoke a sharply-bounded-quantifier test inside a loop. For example, if u denotes the bit string of length $|x|$ defined by $u_i = 1 \iff (\exists y < |x|)\phi(x, y, u_1u_2 \cdots u_{i-1})$, with ϕ a QL relation, such an algorithm could compute u and test whether u has some complex property, such as encoding a graph with a unique 7-clique — which is not obviously possible in $SBH(QL)$. This class, like $SBH(QL)$, has the property that each leaf of the computation tree has access to only constantly many $\log(n)$-bit blocks of nondeterminism, but there may be quasilinearly many quantifiers in the whole tree. We could characterize it with the same machine model as in subsection 4.1, requiring not that the number of nondeterministic configurations along any branch of the computation tree be constant-bounded, but rather that the number of edges $q \mapsto TEST(q)$ along any branch of the computation tree be constant-bounded.

6 Acknowledgments

The author wishes to thank Judy Goldsmith and Jon Buss for valuable discussions.

References

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *computational complexity*, 2:97–110, Dec 1992.

[Bel92] Stephen Bellantoni. *Predicative Recursion and Computational Complexity*. PhD thesis, University of Toronto, 1992.

[BG92] Stephen Bloch and Judy Goldsmith. Sharply bounded alternation within $\mathcal{P}$. Technical Report 92-04, University of Manitoba Computer Science Dept, Apr 1992.

[BG93] Jonathan F. Buss and Judy Goldsmith. Nondeterminism within $\mathcal{P}$. 22(3):560–572, 1993. See also Proceedings, STACS 1991, 348–359.

[Blo92] Stephen Bloch. *Divide and Conquer in Parallel Complexity and Proof Theory*. PhD thesis, University of California, San Diego, 1992.

[Blo94] Stephen Bloch. Function-algebraic characterizations of log and polylog parallel time. *computational complexity*, 1994. To appear; see also "Functional characterizations of uniform log-depth and polylog-depth circuit families" in *Proc. Seventh Annual Structure in Complexity Theory Conference*, IEEE Computer Society Press, 1992, 193–206.

[Bus88] Jonathan F. Buss. Relativized alternation and space-bounded computation. *Journal of Computer and System Sciences*, 36(3):351–378, June 1988.

[Cob65] 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.

[Con94] Anne Condon. A theory of strict P-completeness. *computational complexity*, 1994. To appear.

[Lei90] Daniel Leivant. Subrecursion and lambda representation over free algebras. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, *Perspectives in Computer Science*, pages 281–291. Birkhäuser, 1990.

[Lei92] Daniel Leivant. Personal correspondence, 1992.

[Sch78] C.P. Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25:136–145, 1978.