Arithmetization: A New Method in Structural Complexity Theory.

ARTICLE *in* COMPUTATIONAL COMPLEXITY · MARCH 1991

Impact Factor: 1.08 · DOI: 10.1007/BF01200057 · Source: DBLP

Lance Fortnow
Georgia Institute of Technology
252 PUBLICATIONS **5,832** CITATIONS

ARITHMETIZATION: A NEW METHOD IN STRUCTURAL COMPLEXITY THEORY

László Babai and Lance Fortnow

Abstract. We introduce a technique of arithmetization of the process of computation in order to obtain novel characterizations of certain complexity classes via *multivariate polynomials.* A variety of concepts and tools of elementary algebra, such as the degree of polynomials and interpolation, becomes thereby available for the study of complexity classes.

The theory to be described provides a unified framework from which powerful recent results follow naturally.

The central result is a characterization of $\sharp$P in terms of *arithmetic straight line programs.* The consequences include a simplified proof of Toda's Theorem that $PH \subseteq \mathrm{P}^{\sharp\mathrm{P}}$; and an infinite class of natural and potentially inequivalent functions, checkable in the sense of Blum et al. Similar characterizations of PSPACE are also given.

The arithmetization technique was independently discovered by Adi Shamir. While this simultaneous discovery was driven by applications to interactive proofs, the present paper demonstrates the applicability of this technique to classical complexity classes.

Subject classifications. 68Q15, 68Q60

1. Introduction

We introduce a novel technique in structural complexity theory. We propose to use multivariate polynomials for the description of complexity classes. These descriptions are obtained by arithmetizing the intrinsically Boolean process of computation. By extending the Boolean domain to a field and replacing Boolean operations by arithmetic ones, it becomes possible to "interpolate truth-values" and use concepts like the degree of a polynomial in the study of resource-bounded computation.

This technique was simultaneously and independently discovered by Adi Shamir and has been strikingly successful in determining the power of interactive proof systems [23, 4].

We should mention that in a different context (circuit complexity), A. A. Razborov has previously introduced an arithmetization technique to obtain lower bounds [21, 24].

In this paper we develop a theory of arithmetic straight line programs and demonstrate its applicability to classical complexity classes.

Our central result is a characterization of $\sharp$P in terms of uniform sequences of certain arithmetic straight line programs called "positive retarded arithmetic programs with binary substitution" (PRAB). The proof of this result (Theorem 3.1) indicates the basic arithmetization technique.

We demonstrate the power of the characterization of $\sharp$P by deducing a number of consequences. A similar characterization of the class $\sharp_m$P, the mod m version of $\sharp$P follows directly. A randomized version of the PRAB's is defined analogously. Its mod 2 version is shown to characterize the complexity class BP $\cdot \oplus \cdot$ P. Adapting an idea of Toda, we show that the levels of the polynomial hierarchy also admit a representation by randomized PRAB's mod 2. A simplified proof of Toda's Theorem $PH \subseteq \mathrm{P}^{\sharp \mathrm{P}}$ follows.

As a further consequence, we show that $\sharp_m$P has interactive proofs with $\sharp_m$P-powerful honest provers, thus exhibiting an infinite class of natural and potentially inequivalent functions, checkable in the sense of Blum [8, 9, 10]. (Up to natural equivalence, only a handful of such functions were previously known; unnatural ones could be obtained by padding and other artificial tricks.) Adapting an idea of Beaver and Feigenbaum [6] and Lipton [19], these classes are also shown to have the random self-correcting property in the sense of Blum, Luby and Rubinfeld [10].

We also obtain arithmetic straight line program characterizations of PSPACE functions as well as their multilinear extensions.

The new technique was motivated by an arithmetization of Boolean formulas obtained by the authors in an attempt to remove permanents from the recent proof by Lund, Fortnow, Karloff, and Nisan [20] that $\mathrm{P}^{\sharp \mathrm{P}}$ has interactive proofs. Such a direct proof indeed is a corollary. A similar arithmetization was found independently by Adi Shamir [23].

A preliminary version of this paper was presented at FOCS'90 [3]. Even earlier, the basic technique and some consequences were discussed in the survey paper [2] where the suspense story of the genesis of this and related techniques was also told.

2. Preliminaries

2.1. Complexity classes for counting. By an NP machine we mean a nondeterministic Turing machine that runs in time polynomial in the length of the input. FP will denote the class of polynomial time computable functions.

Following Valiant [27], we say a function $f : \{0,1\}^* \to \mathbf{Z}$ is a $\sharp$P function if there exists an NP machine M such that for all $x \in \{0,1\}^*$, $f(x)$ is exactly the number of accepting paths of $M(x)$. Any nonnegative polynomial-time computable function is also a $\sharp$P function.

Toda [26] showed that every language in the polynomial-time hierarchy [25] reduces to a problem in $\sharp$P. We give a simplified version of Toda's proof in this paper.

Let m be a positive integer. A function $f : \{0,1\}^* \to \mathbf{Z}_m$ belongs to $\sharp_m$P if there exists a function $g \in \sharp$P such that $f(x)$ is the mod m residue class of $g(x)$ for every $x \in \{0,1\}^*$. Beigel, Gill and Hertrampf [7] show that for any constant m we have $\mathrm{P}^{\sharp_m\mathrm{P}} = \mathrm{P}^{\sharp_{m'}\mathrm{P}}$ where m' is the product of prime divisors (the square-free part) of m.

Some comments on reducibility and completeness in these classes will be in order here.

A function f is *polynomially bounded* if for every string x, $|f(x)| \le p(|x|)$ for some polynomial p. We restrict our discussion to such functions. Let PBF denote the class of these functions.

A function $f \in PBF$ is *polynomial time Turing reducible* to a function $g \in PBF$ if $f \in \mathrm{FP}^g$, *i.e.*, a polynomial-time oracle machine with access to values of g can compute f. This machine provides a polynomial-time Turing reduction from f to g. Such a reduction is *nonadaptive* if all queries to g must be computed from the input x (without reference to the outcome of other queries). In other words, f has a nonadaptive polynomial-time Turing reduction to g if there exists a pair of polynomial-time computable functions h_1, h_2 such that for every input string x, $h_1(x) = (y_1(x), \ldots, y_{m(x)}(x))$ is a list of input strings for g, and

$$f(x) = h_2(g(y_1(x))\# \ldots \# g(y_{m(x)}(x))).$$

We say that f has a *polynomial-time many-one reduction* to g if there is a (nonadaptive) polynomial-time Turing-reduction which on each input x, makes a single g-query (i.e. $m(x) = 1$).

Valiant [27] originally defined $\sharp$P-completeness via polynomial-time Turing reductions and has shown that permanent of zero-one matrices is $\sharp$P-complete. In fact, his proof shows that the (0,1)-permanent is $\sharp$P-complete under nonadaptive polynomial-time Turing reductions. Going one step further, Zankó [30] has shown that the $(0,1)$-permanent remains $\sharp$P-complete under polynomial-time many-one reductions.

From the proof it follows that $(0,1)$-permanents mod m are $\sharp_m$P-complete under many-one reductions for *odd* m. (The same is unlikely to hold for even m. Recall in particular that for m a fixed power of 2, the permanent mod m can be computed in polynomial time [27].)

2.2. Probabilistic and parity operators. Let $<x,y>$ be a standard pairing function mapping $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$. We will use several operators on complexity classes defined in Schöning [22] and Toda [26]. Let $\mathcal{C}$ be any complexity class. We define the classes $\mathrm{P} \cdot \mathcal{C}$, $\mathrm{BP} \cdot \mathcal{C}$ and $\oplus \cdot \mathcal{C}$ as follows:

1. $L \in \mathrm{P} \cdot \mathcal{C}$ if there exists an $L' \in \mathcal{C}$ and a polynomial $p(n)$ such that for all $x \in \{0,1\}^*$, $x \in L$ if and only if for at least half of the strings y of length $p(n)$, $<x,y> \in L'$.

2. $L \in \oplus \cdot \mathcal{C}$ if there exists an $L' \in \mathcal{C}$ and a polynomial $p(n)$ such that for all $x \in \{0,1\}^*$, $x \in L$ if and only if for an odd number of the strings y of length $p(n)$, $<x,y> \in L'$.

3. $L \in \mathrm{BP} \cdot \mathcal{C}$ if there exist an $L' \in \mathcal{C}$ and a polynomial $p(n)$ such that for all $x \in \{0,1\}^*$,

   (a) if $x \in L$ then for at least two-thirds of the y's of length $p(n)$, $<x,y> \in L'$, and

   (b) if $x \notin L$ then for at least two-third of the y's of length $p(n)$, $<x,y> \notin L'$.

Schöning [22] also shows that for any reasonable complexity class $\mathcal{C}$ and any polynomial $q(n)$, we can replace two-thirds by $1 - 2^{q(n)}$ without affecting $\mathrm{BP} \cdot \mathcal{C}$ by repeated testing and majority vote.

With this notation, we have $\mathrm{BPP} = \mathrm{BP} \cdot \mathrm{P}$, $\mathrm{PP} = \mathrm{P} \cdot \mathrm{P}$, and $\sharp_2\mathrm{P} = \oplus \cdot \mathrm{P}$.

2.3. Interactive proofs. An *interactive proof system* is a game between two players, an infinitely powerful prover and a probabilistic polynomial-time verifier. The prover and the verifier have a conversation where the prover tries to convince the verifier that a string x belongs to a language L. They take turns to write messages on a board. In the public-coin version (which we shall use), all messages of the verifier are random strings. After a polynomial number of rounds (each of polynomial length), the conversation ends and a deterministic polynomial time "judge" declares who wins. If indeed $x \in L$ then the prover should be able to win against most random coin flip sequences of the verifier; if $x \notin L$, then the prover should lose against most random coin flip sequences.

IP denotes the class of languages admitting interactive proof of membership. Babai [1, 5] and Goldwasser, Micali and Rackoff [16] invented interactive proof systems and formal definitions can be found in these papers. The two variants defined in these papers were shown equivalent by Goldwasser and Sipser [17].

We say a function $f : \{0,1\}^* \to \{0,1\}^*$ has an interactive proof if $|f(x)| \leq |x|^{O(1)}$ and the graph of f has an interactive proof. Lund, Fortnow, Karloff and Nisan [20] show that every $\sharp$P function has an interactive proof. Extending the ideas of [20], Shamir [23] shows that every PSPACE language has an interactive proof. We give alternate proofs of both these results in this paper.

2.4. Program self-testing. Blum and Kannan [8, 9] describe *program instance checking*. A Monte Carlo program C is a program instance checker for a function f if the following holds. Let P be a program that claims to compute f. The checker C on input x using P as a subroutine will proclaim either "P computes the correct value for $f(x)$" or "P differs from f for some input" and for every x, C will be right with high probability. A language has a program instance checker if its characteristic function does. Blum and Kannan [9] show that if a language L has an interactive proof where the prover only answers questions about membership in L then L has a program instance checker.

Blum, Luby and Rubinfeld [10] developed a concept which they call *self-testing/correcting* pairs for programs. A similar concept encompassing the self-correction feature was developed independently by Lipton [19]. We will give a definition which differs slightly from that of [10]:

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. A *self-testing/correcting pair* for f is a pair (T, C) of probabilistic polynomial-time algorithms with the following properties for every $n \geq 0$. Let P be a program claimed to compute f. We would like the corrector $C(P, x)$ to have the following property:

(C_n): For every $x \in \{0,1\}^n$, $C(P, x)$ computes $f(x)$ with large probability.

We require the tester T to fulfill the following requirements:

1. On input $(P, 1^n)$, T outputs either "Pass" or "Fail".
2. If P correctly computes f on all inputs, then $T(P, 1^n)$ always outputs "Pass".
3. If property (C_n) does not hold then with large probability, $T(P, 1^n)$ outputs "Fail".

Large probability means at least 2/3 but this can be amplified to 2^{-n^c} by repeated application and majority vote. One may wish to allow an error in requirement 2, but in every known example the tester always passes a correct program.

Proposition 2.1. *If a function f has a self-testing/correcting pair then it has a program instance checker.*

Proof. Let (T, C) be the self-testing/correcting pair. The instance checker will work as follows: On input x, simulate $T(P, 1^{|x|})$ and if T says "Fail" then output "P differs from f". Otherwise if $C(P, x) = P(x)$ output "P computes the correct value on input x" otherwise output "P differs from f". □

Up to natural equivalence, the only previously known languages with instance checkers were some number theory related languages from NP∩coNP [16], Graph Isomorphism [18], coset intersection in permutation groups [5], and the languages complete for ♯P [20], PSPACE [23], and EXPTIME [4]. We will show that an infinite set of natural and potentially inequivalent functions, the $\sharp_m$P-complete functions, have self-testing/correcting pairs.

Details of the comparison of the above definition and the [10] definition can be found in [4]. One aspect we should point out is that under our definition, the following holds.

Proposition 2.2. *If the functions f and g are polynomial-time Turing equivalent, and f has a self-testing/correcting pair, then g has a self-testing/correcting pair, too.*

This may be false under the [10] definition.

3. A characterization of ♯P

In this section we introduce the basic arithmetization technique. Variations can be found in later sections of this paper as well as in [23] and [4].

We define a class of straight line programs of multivariate polynomials with integral coefficients over the set $\{x_1, x_2, \ldots\}$ of variables.

A *positive retarded arithmetic program with binary substitutions* (PRAB) will be a sequence $\{p_1, \ldots, p_t\}$ of instructions such that for every k, one of the following holds:

(1) p_k is one of the constant polynomials 0 or 1;

(2) $p_k = x_i$ for some $i \leq k$;

(3) $p_k = 1 - x_i$ for some $i \leq k$;

(4) $p_k = p_i + p_j$ for some $i, j < k$;

(5) $p_k = p_i p_j$ for some i, j such that $i + j \leq k$ *(retarded multiplication);*

(6) $p_k = p_j(x_i = 0)$ or $p_j(x_i = 1)$ for some $i, j < k$ *(binary substitution).* (Here, $p_j(x_i = \varepsilon)$ refers to the polynomial obtained from p_j by replacing the variable x_i by the value ε.)

Such a program defines a sequence $\{\tilde{p}_k\}$ of polynomials in an obvious way.

We note that (by induction) the set of variables involved in $\tilde{p}_k$ is a subset of $\{x_1, \ldots, x_k\}$; and the degree of $\tilde{p}_k$ is $\leq k$. (The latter is a consequence of the retarded multiplication.)

We say that the program P computes the polynomial $\tilde{p}_t$, the last member of the sequence. We use the symbol $\tilde{P}$ to denote $\tilde{p}_t$.

A family $P_1, P_2, \ldots$ of PRAB's is *uniform* if, upon input 1^n, a polynomial time deterministic Turing machine computes the instruction set P_n and the polynomial $\tilde{P}_n$ computed only depends on the variables $\{x_1, \ldots, x_n\}$. (In particular, the length of the straight line program P_n is bounded by $n^{O(1)}$.) A formally stronger condition would be to require the machine to be logspace bounded. We shall see that this distinction is irrelevant from our point of view.

We shall use the term "family" to indicate infinite sequences such as those in the previous paragraph, reserving the term "sequence" for finite ones such as a PRAB.

Let $\mathbf{Z}^+$ denote the set of nonnegative integers.

Theorem 3.1. *For a function $f : \{0,1\}^* \to \mathbf{Z}^+$, the following are equivalent.*

(a) $f \in \sharp$P.

(b) There exists a uniform family of PRAB's P_n such that for every string $x \in \{0,1\}^*$,

$$f(x) = \tilde{P}_{|x|}(x).$$

(By substituting a binary string $\alpha = \alpha_1 \ldots \alpha_n$ into a polynomial P in n variables we mean the substitution $P(\alpha_1, \ldots, \alpha_n)$.)

Remark 3.2. The result is valid with either notion of uniformity.

Proof. I. We first prove the implication (a) $\Longrightarrow$ (b). Let M be an NP-machine which on input $x \in \{0,1\}^*$ has $f(x)$ accepting paths.

Following the standard proof of the Cook-Levin Theorem (cf. [15]), for every n we associate a 3-CNF formula φ_n with M. There are two kinds of variables in φ_n: the *input variables* x_i $(i = 1, \ldots, n)$ and the *guess variables* y_j $(j = 1, \ldots, m = m(n))$. For every input x, the number of strings y such that (x, y) satisfies $\varphi_{|x|}$ is $f(x)$.

We now view the x_i and y_j as arithmetic rather than Boolean variables and associate a polynomial in the same variables with φ_n as follows. Let say $C = u \vee v \vee \neg w$ be one of the clauses in φ_n. We rewrite C as the disjunction of 7 clauses, each clause a 3-term conjunction of the form $(\neg)u \wedge (\neg)v \wedge (\neg)w$. (Out of the 8 possible assignments of $\neg$ in front of the variables, the only one missing will be $\neg C = \neg u \wedge \neg v \wedge w$.) Then we replace each occurrence of a negated variable, say $\neg z$, by $(1 - z)$, each $\wedge$ by multiplication and each $\vee$ by addition. Finally we take the product over all clauses C of the resulting polynomials of degree 3. The resulting polynomial q_n has the property that on $(0, 1)$ substitutions its value is 0 or 1 and agrees with the value of φ_n under the same substitution.

We thus have the identity

$$f(x) = \sum_{y \in \{0,1\}^{m(|x|)}} q_{|x|}(x, y) \tag{3.1}$$

for all $x \in \{0, 1\}^*$.

For every $n = |x|$, the right hand side is easily seen to be represented by a PRAB. Indeed, the polynomial q_n arises using the first 5 instructions only. To obtain the sum, we use, for each variable y_j, both binary substitutions, and add the two results; then proceed to y_{j+1}.

For this PRAB, it is clear that P_n is computable in space $O(\log n)$.

II. For the (b) $\Longrightarrow$ (a) implication, we only assume polynomial time uniformity.

Let $P = \{p_1, \ldots, p_t\}$ be a PRAB such that the set of variables involved in each p_i is contained in the set $\{x_1, \ldots, x_t\}$. For each i, we construct a nondeterministic Turing machine M_i such that on input $x \in \{0, 1\}^t$, the number of accepting paths of M_i is $p_i(x_1, \ldots, x_t)$.

To construct M_k, we distinguish 6 cases according to the type of instruction p_k. Cases (1), (2), (3) are clear. In case (4) (sum), we make a nondeterministic move from START to one of M_i and M_j. In case (5) (product) we first run M_i; and if it accepts, we run M_j. (In this case, the running time is the sum of the running times of M_i and M_j and will therefore be linear in k by the retardation rule.) Case (6) is clear: to obtain M_k, we hard wire the value of x_i in M_j. Note that the description of M_k is only a constant number of instructions added on top of the instructions of earlier machines.

Now the NP-machine we wish to associate with the PRAB $\{P_1, P_2, \ldots\}$ proceeds as follows. On input $\alpha = (\alpha_1, \ldots, \alpha_n) \in \{0, 1\}^n$, it first computes in deterministic polynomial time the program $P_n = \{p_{n,1}, \ldots, p_{n,m(n)}\}$. Then it computes (still in deterministic polynomial time) the description of the nondeterministic Turing machine $M_{n,m(n)}$ associated with $p_{n,m(n)}$. Finally, it simulates this machine on input α.

This completes the proof. □

In an earlier work, Venkateswaran [29] characterized $\sharp P$ functions in terms of certain arithmetic formulas of polynomial depth and polynomial degree. These formulas may have exponential size. We note that his characterization also follows from our characterization of $\sharp P$ by expanding the binary substitution rule.

We can also define a more general class of straight line programs of multivariate polynomials. The difference will be that we permit subtraction as well as addition:

(4') $p_k = p_i \pm p_j$ for some $i, j < k$;

We call the straight line programs composed of instructions (1), (2), (3), (4'), (5), and (6) *retarded arithmetic programs with binary substitution* (RAB).

The class Gap-P consists of all functions that are the difference of two $\sharp$P functions. Fenner, Fortnow and Kurtz [14] show that retarded arithmetic programs characterize Gap-P functions in full analogy with Theorem 3.1:

Theorem 3.3. (Fenner-Fortnow-Kurtz) *For a function $f : \{0,1\}^* \to \mathbf{Z}$, the following are equivalent.*

(a) $f \in$ Gap-P.

(b) There exists a uniform family of RAB's P_n such that for every string $x \in \{0,1\}^*$,

$$f(x) = \tilde{P}_{|x|}(x).$$

Theorem 3.3 follows directly from the following Lemma:

Lemma 3.4. *For a function $f : \{0,1\}^n \to \mathbf{Z}$, the following are equivalent.*

(a) f is the difference of two functions defined by PRAB's.

(b) f is defined by a RAB.

Sketch of Proof. By induction on the instruction index. Sums, differences, products of two differences can be expressed as the differences of sums and sums of products, resp. □

It is natural to also permit general (retarded) substitutions:

(6') $p_k = p_j(p_{i_1}, \ldots, p_{i_l})$ for some $j, i_1, \ldots, i_l$, where $k \geq j(i_1 + \ldots + i_l)$.

We call the straight line programs composed of instructions (1), (2), (3), (4'), (5), and (6') *retarded arithmetic programs* (RA). Suprisingly, adding the extra power of retarded substitutions does not increase the class of functions defined.

THEOREM 3.5. *For a function $f : \{0,1\}^n \to \mathbf{Z}$, the following are equivalent.*

(a) f is defined by a RAB.

(b) f is defined by a RA.

The proof is routine.

We can now categorize the class PP in terms of RAB's.

THEOREM 3.6. *For a language $L \subset \{0,1\}^*$ the following are equivalent.*

(a) $L \in$PP

(b) There exists a uniform family of RAB's P_n such that for every string $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \tilde{P}_{|x|}(x) \geq 0.$$

We omit the routine proof which is based on Theorem 3.3.

REMARK 3.7. *The celebrated proof of Beigel, Reingold and Spielman [11] that PP is closed under union can be expressed simply in terms of RA's. Their main trick involves using a rational function $r(t) = p(t)/q(t)$ of a single variable t which very closely approximates the sign function on the nonzero integers of n^c digits. Then for two RA functions $f(x)$ and $g(x)$ corresponding to the PP languages L_1 and L_2 respectively, the function $h(x) = r(f(x)) + r(g(x)) - 1$ is positive if either $f(x)$ or $g(x)$ is positive and negative otherwise. But $h(x)$ is a rational function, so if we multiply by the square of the products of the denominator we get a RA that is positive if and only if at least one of $f(x)$ or $g(x)$ is positive, i.e. $x \in L_1 \cup L_2$. The completes the sketch of the proof of the Theorem of Beigel et al. [11]*

4. Interactive proofs

We say that a function $f : \{0,1\}^* \to \mathbf{Z}$ has an interactive proof if the number of digits of $f(x)$ is bounded by $|x|^{O(1)}$ and the graph of f belongs to the class IP. Let FIP denote the class of these functions.

THEOREM 4.1. *Let $P_1, P_2, \ldots$ be a (polynomial time) uniform family of retarded arithmetic programs. The function $f : \{0,1\}^* \to \mathbf{Z}$ with*

$$f(x) = \tilde{P}_{|x|}(x)$$

has an interactive proof.

PROOF. The protocol is an adaptation of the interpolation idea of Lund, Fortnow, Karloff, Nisan. Just like their procedure, this one is Arthur-Merlin (only public coins are employed.)

We outline the procedure. There is a trivial $\exp(|x|^{O(1)})$ upper bound to the absolute value of $f(x)$ hence it suffices to verify the value $f(x)$ modulo a prime r, greater than this bound. We shall thus view the arithmetic program as referring to polynomials over the field $\mathbf{F}_r$.

Let the sequence of polynomials in the program be $p_1, \ldots, p_t$. The protocol will run in t rounds, numbered backward (t to 1). In round k, the prover will be requested to state the coefficients of some univariate polynomials of the form $g(z) = p_j(h_1(z), \ldots, h_t(z))$ where $j \le k$ and the univariate polynomials h_i are specified by the verifier. (Clearly, $g(z)$ has polynomially bounded degree.) Subsequently the verifier evaluates g at a randomly selected place $\gamma \in \mathbf{F}_r^t$ and records the result as the stated value of $p_j(\gamma)$.

Round k thus starts by reviewing the recorded stated values of p_k inherited from previous rounds; suppose these are the values at $\gamma_1, \ldots, \gamma_m \in \mathbf{F}_r^t$. (In round t, there is one stated value: the one the protocol intends to verify.) Let $L_i(z)$ denote the Lagrange interpolation polynomial of degree $m-1$ which takes value 1 at $z = i$ and value 0 at $z = 1, \ldots, m$, $z \ne i$. The verifier sets

$$g(z) = p_k(\sum_{i=1}^{m} L_i(z)\gamma_i)$$

and requests the coefficients of $g(z)$. Then he checks for $z = 1, \ldots, m$ that the stated value of $p_k(\gamma_i)$ agrees with the value of the stated polynomial g at i. In case of discrepancy he rejects. Otherwise he picks some $\tau \in \mathbf{F}_r$ at random and records $g(\tau)$ as the stated value of p_k at $\delta_k = \sum_{i=1}^{m} L_i(\tau)\gamma_i$. In distinction from the previous stated values, we call this the stated *core* value of p_k.

Now if p_k arises by rules (1),(2), or (3), the verifier directly checks the core value and rejects if it is found incorrect. If p_k arises by rules (4'), (5), or (6'), the verifier requests the corresponding values of the constituents of p_k and records them as stated values. (In case (6'), e.g., these values are $\zeta_s = p_{i_s}(\delta_k)$, $s = 1, \ldots, l$, and the value $p_j(\zeta_1, \ldots, \zeta_l)$.) End of round k.

Acceptance occurs at the end of round 1 unless the protocol has previously halted with rejection.

It is clear that the procedure is polynomial time. It is also clear that a correct prover will make his case accepted.

We have to argue that a cheating prover has slim chance of acceptance. As in [20], the idea is that if two univariate polynomials of low degree differ, then they differ almost everywhere. So if any one of the stated values present at the beginning of round k is incorrect, then either rejection occurs during round k, or the core value will be misstated with overwhelming probability at least $1 - k(m-1)/r$ since the degree of $g(z)$ is at most $k(m-1)$. But then in cases (4'), (5), or (6'), the prover is forced to state an incorrect value for a lower index member of the sequence. Consequently, the probability of a wrong value being accepted is at most

$$\sum_{k=1}^{t} (kt/r) < t^3/2r$$

noting that in each round, $m < t$. □

A self-contained proof of the [20] result is now immediate.

Corollary 4.2. $\sharp\mathrm{P} \subseteq \mathrm{FIP}$. *Consequently,* $\mathrm{P}^{\sharp\mathrm{P}} \subseteq \mathrm{IP}$.

Proof. Clear. □

We remark that the [20] proof depended on Valiant's celebrated result [27] that the permanent of (0,1)-matrices is $\sharp$P-complete. Valiant's proof (the only proof known of Valiant's Theorem) is substantially more intricate than the proof of the $\mathrm{P}^{\sharp\mathrm{P}} \subseteq \mathrm{IP}$ result given here.

5. A characterization of $\mathrm{BP} \cdot \oplus \cdot \mathrm{P}$

In a recent seminal paper, S. Toda [26] proves that the polynomial hierarchy is contained in $\mathrm{P}^{\sharp\mathrm{P}}$. His proof is a combination of two results, both significant in their own right. He proves that

$$\mathrm{PH} \subseteq \mathrm{BP} \cdot \oplus \cdot \mathrm{P} \subseteq \mathrm{P} \cdot \oplus \cdot \mathrm{P} \subseteq \mathrm{P}^{\sharp\mathrm{P}}.$$

(The inclusion in the middle is trivial.) We show, heavily drawing on details of Toda's proof, that both results tie in neatly with the retarded arithmetic programs, and this connection leads to a modest simplification of Toda's original proof.

First we characterize the classes $\mathrm{BP} \cdot \oplus \cdot \mathrm{P}$ and $\mathrm{P} \cdot \oplus \cdot \mathrm{P}$ in terms of retarded arithmetic programs.

THEOREM 5.1. *For a function $f : \{0,1\}^* \to \mathbf{Z}_2$, the following are equivalent:*

1. *$f \in \mathrm{BP} \cdot \oplus \cdot \mathrm{P}$;*
2. *There exists a uniform family of PRAB's P_n such that the polynomial $\tilde{P}_n$ computed by P_n has $n + m(n)$ variables for some polynomial $m(n)$, and for every $x \in \{0,1\}^*$,*

$$f(x) = \tilde{P}_{|x|}(x, r) \pmod 2$$

for at least a 2/3 fraction of the strings $r \in \{0,1\}^{m(|x|)}$.

The fraction 2/3 can be amplified to $1 - 2^{-n^c}$ for any constant c.

The same holds for membership in $\mathrm{P} \cdot \oplus \cdot \mathrm{P}$, except that marked majority is replaced by simple majority.

PROOF. By definition, $f \in \mathrm{BP} \cdot \oplus \cdot \mathrm{P}$ if and only if there exists $g \in \sharp\mathrm{P}$ such that for every $x \in \{0,1\}^*$,

$$f(x) = g(x, r) \pmod 2$$

for at least a 2/3 fraction of the strings $r \in \{0,1\}^{m(|x|)}$ where $m(n) = n^C$ for some constant C. The fraction 2/3 can be amplified to $1-2^{-n^c}$ for any constant c. [22] The claim is thus immediate from Theorem 3.1.

The proof for $\mathrm{P} \cdot \oplus \cdot \mathrm{P}$ goes analogously. □

COROLLARY 5.2. (TODA) $\mathrm{P} \cdot \oplus \cdot \mathrm{P} \subseteq \mathrm{P}^{\sharp\mathrm{P}}$.

PROOF. We use one of Toda's tricks. Toda shows that the polynomial $T(x) = 4x^3 + 3x^4$ has the property that for every $k \geq 1$ and $\varepsilon \in \{0,1\}$, if $x \equiv -\varepsilon \pmod{2^k}$ then $T(x) \equiv -\varepsilon \pmod{2^{2k}}$. This fact is a consequence of the two decompositions (a) $T(x) = x^2(4x+3x^2)$ and (b) $T(x)+1 = (x+1)^2(3x^2-2x+1)$.

Let now $f \in \mathrm{P} \cdot \oplus \cdot \mathrm{P}$ and let $P_1, P_2, \ldots$ denote the uniform family of PRAB's associated with f by the previous Theorem. Let $T^{(s)} = T \circ \cdots \circ T$ (s times) denote the s-fold iteration of T. For $s = O(\log n)$, the polynomial $\tilde{Q}_n = T^{(s)} \circ \tilde{P}_n$

is still the result of a PRAB. (Each application of T will force us to increase the length of the PRAB by a factor of four, by padding with dummy instructions, to satisfy the retardation rule. Thus $O(\log n)$ applications of T will increase the length of the PRAB by a polynomial factor.) On the other hand, by the congruence property of T, for $n = |x|$ and $s > \log m(n)$, the remainder mod $2^{m(n)}$ of the sum

$$S_n(x) = \sum_{r \in \{0,1\}^{m(n)}} -\tilde{Q}_n(x,r)$$

counts the odd numbers among $\tilde{P}_n(x,r)$. Hence $f(x) = 1$ if and only if this remainder is greater than 2^{s-1}.

Observing that $-S_n$ is described by a PRAB, it follows that the function $-S_{|x|}(x)$ belongs to $\sharp$P, hence f belongs to $\mathrm{P}^{\sharp\mathrm{P}}$. $\square$

6. Retarded arithmetic programs for the polynomial time hierarchy

In this section we reprove Toda's result that PH $\subseteq$ BP $\cdot \oplus \cdot$ P. We use some of his intuition; in particular we, too, use the following Lemma of Rabin (see [28]). Apart from this Lemma, the proof given below is self-contained.

Lemma 6.1. (Rabin[28]) *Let S be a nonempty subset of $\mathbf{F}_2^n$. Let us select n vectors $w_1, \ldots, w_n \in \mathbf{F}_2^n$ at random. Let U_k denote the subspace of $\mathbf{F}_2^n$ defined by the equations $x \cdot w_1 = \ldots = x \cdot w_k = 0$ $(k = 0, \ldots, n)$, where the dot indicates the standard inner product. Then with probability $\geq 1/4$, we have $|U_i \cap S| = 1$ for at least one of the U_i.*

The *characteristic function* of a subset S of a universe X is the function $\chi_S : X \to \{0,1\}$ taking value 1 on S and 0 elsewhere.

Proposition 6.2. *Let U be a subspace of $\mathbf{F}_2^n$ defined by the equations $x \cdot w_1 = \ldots = x \cdot w_k = 0$, where $w_i \in \mathbf{F}_2^n$ and the dot indicates the standard inner product. Then the characteristic function of U can be represented as a polynomial of degree k (over $\mathbf{F}_2$, in $n(k+1)$ variables, namely the components of $x, w_1, \ldots, w_k$).*

Proof. Clearly,

$$\chi_U(x) = \prod_{i=1}^{k} (1 + x \cdot w_i). \quad \square$$

$\square$

We now wish to encode quantified boolean formulas with a bounded number of quantifier alternation as retarded arithmetic programs over $\mathbf{F}_2$. Let us consider a sequence of partially quantified Boolean formulas defined by

$$\psi_i(x_{i+1}, \ldots, x_d) = Q_i x_i \psi_{i-1}(x_i, \ldots, x_d),$$

where the x_i are strings of variables of length n_i and the Q_i are quantifiers. ψ_0 is a 3-CNF formula.

We prove by induction on i that ψ_i has mod 2 PRAB's in the probabilistic sense.

Theorem 6.3. *Partially quantified Boolean formulas with a bounded number of quantifier alternations can be represented probabilistically by PRAB's* mod 2 *in the sense that for any ψ_i there exists a PRAB P^i such that*

$$\tilde{P}^i(x_{i+1}, \ldots, x_d, r_1, \ldots, r_i) = \psi_i(x_{i+1}, \ldots, x_d)$$

for all but an arbitrarily exponentially small fraction of the r_j's where r_j is a string of variables bounded in length by a polynomial in n.

Proof. We prove the Theorem by induction on i. The case $i = 0$ is immediate from Theorem 3.1. Assume a PRAB P^{i-1} represents ψ_{i-1}.

Assume $Q_i = \exists_i$. Let $T_k(x_i, w^k)$ be the characteristic function of the subspace defined in Proposition 6.2 where $w^k = (w_1, \ldots, w_k)$. By Proposition 6.2, this function can be represented by a polynomial of degree k. By Lemma 6.1, if we choose w^{n_i} at random then with probability at least 1/4 we will have $S_k(w^{n_i}) = \Sigma_{x_i} T_k \tilde{P}^{i-1}$ have value one for some k if ψ_{i-1} is satisfiable. Let

$$\tilde{P}^i(w_1^{n_i}, \ldots, w_n^{n_i}) = 1 + \Pi_{j,k}(1 + S_k(w_j^{n_i}))$$

Then with exponentially high probability, $\tilde{P}^i$ will be one if ψ_{i-1} is satisfiable.

Note that we have to create each $S_k(w_j^{n_i})$ separately since they use different variables. Also by the retardation restriction for multiplication we may have to create several dummy polynomials in the PRAB before we can do the required multiplication above. This process may increase the length of the PRAB by a polynomial factor. This increase is the main reason we can only apply this induction a constant number of times.

If $Q_i = \forall_i$ rewrite

$$\psi_i = \neg\exists\neg\psi_{i-1}$$

and proceed as above, adding one for negation.

The probability that at each iteration P^i correctly represents ψ_i is exponentially high and since we are only repeating a constant number of iterations the induction follows. □

The fact that $\mathrm{PH} \subseteq \mathrm{BP} \cdot \oplus \cdot \mathrm{P}$ follows now from Theorems 5.1 and 6.3. Toda's main result that $\mathrm{PH} \subseteq \mathrm{P}^{\sharp \mathrm{P}}$ follows now by Corollary 5.2.

7. A characterization of $\sharp_m$P

Next we give a characterization of the class $\sharp_m$P in terms of retarded arithmetic programs.

THEOREM 7.1. *For a function $f : \{0,1\}^* \to \mathbf{Z}_m$, the following are equivalent:*

1. *$f \in \sharp_m\mathrm{P}$;*
2. *There exists a uniform family of PRAB's P_n such that for every $x \in \{0,1\}^*$,*
$$f(x) \equiv \tilde{P}_{|x|}(x) \pmod m;$$
3. *There exists a uniform family of retarded arithmetic programs Q_n such that for every $x \in \{0,1\}^*$,*
$$f(x) \equiv \tilde{Q}_{|x|}(x) \pmod m.$$

PROOF. The equivalence of (1) and (2) follows from the definition by Theorem 3.1. We only need to prove that for any uniform family of retarded arithmetic programs Q_n there exists a uniform family of PRAB's P_n such that $\tilde{P}_{|x|}(x) \equiv \tilde{Q}_{|x|}(x) \pmod m$.

To simulate a retarded arithmetic program P by a PRAB $Q \pmod m$, we proceed as follows. Q will refer to the same set of variables. Instructions $p_k = p_i - p_j$ will be replaced by $p_k = p_i + (m-1)p_j$; the constant $(m-1)$ is obtained by $O(\log m)$ repeated additions starting from the constant 1. To achieve a general substitution (instruction (6')), we run the program that constructs p_j but use the polynomials p_{i_ℓ} in place of the variables x_ℓ. The retardation condition assures that this simulation will at most square the length of the program. □

REMARK 7.2. *This equivalence remains true even if the integer m depends on the length of the input (in a polynomial time computable way). For a function $m(n)$, computable in polynomial time from 1^n, we can thus define in the natural way the class $\sharp_{m(n)}\mathrm{P}$ of functions f over the domain $\{0,1\}^*$ such that the value $f(x)$ belongs to $\mathbf{Z}_{m(|x|)}$. The above characterization then extends to these classes.*

8. Prover power and checkability of $\sharp_m$P

In this section we will show that $\sharp_m$P has interactive proofs requiring $\sharp_m$P-*powerful honest provers only.* Consequently, $\sharp_m$P-complete functions have a program instance checker in the sense of Blum-Kannan [9].

We will also show that $\sharp_m$P-complete languages have a self-testing/correcting pair in the spirit of Blum, Luby and Rubinfeld [10] (cf. our modified definition and discussion in Section 2.4).

First we solve the problem when m is prime, using extensions of finite fields. An extension to any square-free m is then immediate via the Chinese Remainder Theorem. Finally we will use Beigel-Gill-Hertrampf [7] to extend the result to all m.

For the proof, we have to consider PRAB's over $k \times k$ matrices. In this case, the variables stand for variable matrices; and in rule (6), 0 and 1 represent the zero and the identity matrices, resp.

LEMMA 8.1. *Let $\{p_1, \dots, p_t\}$ be a PRAB where the variables represent $k \times k$ matrices. Then one can construct in polynomial time a PRAB for the entries of these matrices $\{p_1(1,1), \dots, p_t(k,k)\}$ where $p_j(r,s)$ is the entry of p_j in row r and column s. Also $p_j(r,s)$ is defined only in terms of $p_i(r', s')$ with $i < j$.*

PROOF. Rules (1),(2),(3),(4) and (6) are straightforward. For rule (5), note if $p_k = p_i p_j$ then $p_k(r,s) = \sum_{1 \le \ell \le k} p_i(r,\ell) p_j(\ell, s)$. We leave it to the reader to convert this to a PRAB keeping the retarded multiplication requirement. □

THEOREM 8.2. *There exists a single-prover interactive proof system for $\sharp_m$P for any fixed $m > 1$ where the prover need only answer $\sharp_m$P questions.*

PROOF.

I. First we assume m is prime. Let $f \in \sharp_m$P. By Theorem 7.1 we have a uniform family of PRAB's P_n such that $f(x) \equiv \tilde{P}_{|x|}(x) \bmod m$. Suppose we regard $\tilde{P}_n$ as polynomials over the extension field $GF(m^k)$. Then $f(x) = \tilde{P}_{|x|}(x)$ in this field.

The need to extend the field $GF(m)$ arises because in order for the protocol used in Theorem 2 to work, the random choices have to be made from a sufficiently large field.

The field $GF(m^k)$ is a k-dimensional vector space over the field $GF(m)$. Multiplication by an element $a \in GF(m^k)$ is a linear transformation T_a of this vector space. The algebra of the matrices T_a is isomorphic to $GF(m^k)$, hence we can represent $GF(m^k)$ by $k \times k$ matrices over $GF(m)$ (the regular representation). Thus using Lemma 8.1 we have that any PRAB over a field $GF(m^k)$ has an equivalent PRAB over the field $GF(m)$. This holds for any polynomial $k(|x|)$.

If we choose k such that m^k is greater than the square of the length of the given PRAB then the protocol used in Theorem 4.1 will be an interactive protocol for $f(x)$ if we choose the γ randomly from $GF(m^k)$. We can then convert this to a PRAB over the field $GF(m)$ as described above. Since $GF(m)$ is isomorphic to $\mathbf{Z}_m$ for prime m, the prover need only answer $\sharp_m$P questions to give the value of this PRAB.

One comment on how to construct the field $GF(m^k)$. This requires finding an irreducible polynomial of degree k over $GF(m)$. This can be accomplished in Las Vegas time, polynomial in $k \log m$ so the verifier is capable of constructing it even for large k. (Alternatively, the prover could provide it. Checking irreducibility is in P.) In our case even this is not necessary: m^k is so small, the verifier can check all cases and find the desired polynomial deterministically.

This concludes the proof in the case when m is prime.

II. Now we assume m is square free: $m = m_1 \dots m_r$ with each m_i a distinct prime. Then using the Chinese Remainder Theorem we can compute a function $g(x) \bmod m$ from the values $g(x) \bmod m_1, \dots, g(x) \bmod m_r$. Also every function in $\sharp_{m_i}$P is computable (trivially) from a function in $\sharp_m$P.

III. Finally we turn to the case of general $m > 1$.

Recall that for any constant m we have $P^{\sharp_m \mathrm{P}} = P^{\sharp_{m'} \mathrm{P}}$ where m' is the product of prime divisors (the square-free part) of m [7]. We convert the $\sharp_m$P functions into $\sharp_{m'}$P questions and by part II of this proof we have our protocol. $\square$

This result was independently obtained by Peter Shor (oral communication).

THEOREM 8.3. *Every $\sharp_m$P-complete function has a self-testing/correcting pair.*

This theorem refers to our definition of self-testing/correcting pairs given in Section 2.

PROOF. In order to create a self-testing/correcting pair (T, C) for a $\sharp_m$P-complete function f, we have to go through the same phases as the previous proof. According to the remarks in Section 2, it suffices to prove the result for a single $\sharp_m$P-complete function such as the number of satisfying instances modulo m of a Boolean formula. The sole feature of this particular function f we require is that it is $\sharp_m$P-complete under many-one reductions.

I. Assume m is prime. Suppose we have a program P that claims to compute f. For input length n, create the PRAB for n variables which computes f. Let n^c be the length of this PRAB. Select k such that $m^k > 2^n$. The self-testing program T will just check P on n^{2c} randomly chosen instances from $\mathrm{GF}(m^k)^n$, using the instance checker provided by Theorem 8.2. We note that each query to a value over $GF(m^k)$ of a polynomial in the PRAB is a $\sharp_m$P-query according to Lemma 8.1 and can therefore be transformed into a single query to P.

Following the ideas of Beaver-Feigenbaum [6] and Lipton [19], one recognizes that low degree polynomials over fields automatically have the self-correcting feature.

Indeed, if m is prime, we create the correcting program C as follows. Let $f(x_1, \ldots, x_n)$ be the the polynomial computed by the PRAB over $\mathrm{GF}(m^k)^n$. We note that its degree is at most n^c.

Suppose we wish to compute $f(\alpha_1, \ldots, \alpha_n)$. Choose elements $r_1, \ldots, r_n$ at random from $\mathrm{GF}(m^k)^n$ and let $r^i = (\alpha_1 + ir_1, \ldots, \alpha_n + ir_n)$ for $1 \leq i \leq n^c + 1$. Let $g(y) = f(\alpha_1 + yr_1, \ldots, \alpha_n + yr_n)$ for all y. With probability greater than $1 - \frac{n^c+1}{n^{2c}}$, the value of P on r^i is the value of the PRAB since each r^i is random and uniformly distributed over $\mathrm{GF}(m^k)^n$. However, $g(y)$ is a polynomial of degree at most n^c and we have $n^c + 1$ points of its graph, $g(1), \ldots, g(n^c + 1)$. Interpolate this polynomial and compute $g(0) = f(\alpha_1, \ldots, \alpha_n)$. If we repeat this process n times then with extremely high probability a majority of the answers from this process will be the proper value of f.

II. If m is square-free, we recreate the value of f from its values modulo the prime factors of m as in the previous proof.

III. By the above quoted polynomial time Turing equivalence of $\sharp_m$P and $\sharp_{m'}$P, where m' is the square-free part of m [7], the general case follows in view of the comments in Section 2. $\square$

Feigenbaum and Fortnow [13] use the ideas of this section to show that $\sharp_m$P-complete languages are random self-reducible.

9. Characterizations of PSPACE functions

In this section we exhibit rules for straight line programs that characterize PSPACE functions.

The first ingredient is a simple characterization of PSPACE languages. We then employ a simplified (linear) version of Shamir's trick of renaming variables, and obtain a modified characterization where each of the polynomials has low degree: at most quadratic in each variable. We remark that this characterization combined with the protocol of Section 4 immediately yields the IP=PSPACE result.

The following set of rules defines what we call *incomplete arithmetic programs with binary substitution* (IAB) (incomplete because addition is not permitted; however, the retardation condition is dropped).

(1) $p_k = 1$

(2) $p_k = x_i$ for some $i \leq k$;

(3) $p_k = 1 - p_i$ for some $i < k$;

(4) $p_k = p_i p_j$ for some i, j such that $i, j < k$ (Note: not retarded);

(5) $p_k = p_j(x_i = 0)$ or $p_j(x_i = 1)$ for some $i, j < k$.

THEOREM 9.1. *For a function $f : \{0,1\}^* \to \mathbf{Z}$, the following are equivalent.*

(a) f is the characteristic function of some language L in PSPACE.

(b) There exists a uniform family of IAB's P_n such that for every string $x \in \{0,1\}^*$,

$$f(x) = \tilde{P}_{|x|}(x).$$

PROOF. The proof is similar to the proof of Theorem 3.1 with the following changes:

I. Assume we have a language in PSPACE. We can reduce this problem to a QBF formula much like in the proof of Theorem 3.1 we reduced to a 3CNF formula. Replace every $a \vee b$ by $\neg(\neg a \wedge \neg b)$. Likewise replace $\exists\phi$ by $\neg\forall\neg\phi$. Now our formula has only variables, $\wedge$, $\neg$ and $\forall$. To arithmetize the formula, we handle variables by rule (2), $\neg$ by rule (3), $\wedge$ by rule (4) and $\forall$ by rules (5) and (4).

II. Suppose we have an arithmetic program P created with the rules above. We can build a polynomial time alternating Turing machine M to decide if the final value is 1. By Chandra-Kozen-Stockmeyer [12] we can simulate this alternating Turing machine by a PSPACE machine.

First we observe that *on $(0,1)$-substitutions, each polynomial in P takes $(0,1)$-values.*

The machine M will proceed backward in stages starting with stage t (the last stage), attempting to verify that $p_t(x) = 1$.

In stage k, M will proceed depending on the rule defining p_k. For rules (1) and (2) M will just verify the correct value. For rules (3) and (5) M will compute what value p_i should have and verifies that value. For rule (4), M will existentially guess the values of p_i and p_j and will universally verify each of them. □

Notice that if we disallow rule (5), we get all the uniform polynomial size circuits and thus exactly the languages in P.

A slight change of the rules will suffice to yield all PSPACE computable functions. (Note that a PSPACE computable function may output an exponential number of bits.) In order to allow addition and subtraction, it suffices to replace rule (3) by (3') below:

(3') $p_k = p_i - p_j$ for some i, j such that $i, j < k$

The rules (1), (2), (3'), (4), and (5) define the *arithmetic programs with binary substitution* (AB).

The price we pay for omitting the retardation requirement of Section 3 is that the degrees of the polynomials involved can be exponentially large.

Before proving that AB's characterize PSPACE-computable functions, we show, adapting and simplifying Shamir's trick, how to reduce the degree of the polynomials to multi-quadratic (degree at most two in each variable).

We note that for any field $\mathbf{F}$, every function $f : \{0,1\}^n \to \mathbf{F}$ has a unique multilinear extension to a function $\mathbf{F}^n \to \mathbf{F}$. We shall see that every AB can be modified so as to yield this unique multilinear extension of f.

Suppose we have a polynomial $p(y)$ in a single variable. We can create a new *linear* polynomial that has the same value as p on $\{0,1\}$ by using interpolation as follows:

$$q(z) = (1 - z)p(0) + zp(1)$$

We can extend this idea to multivariate polynomials.

Lemma 9.2. *Given an AB $\{p_1, \ldots, p_t\}$, one can compute in polynomial time another AB, $\{q_1, \ldots, q_r\}$ such that*

(a) x_j has degree at most two in q_k for all i, j.

(b) $q_r(\alpha_1, \ldots, \alpha_n) = p_t(\alpha_1, \ldots, \alpha_n)$ for all $(\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$.

(c) q_r is the unique multilinear function satisfying (b).

Proof. Polynomials p_k created by rules (1), (2), (3') or (5) do not increase the degree. After each application of rule (4) to obtain a polynomial $p_k(y_1, \ldots, y_k) = q_{k0}(y_1, \ldots, y_k)$, define

$$\begin{aligned} &q_{ki}(z_1, \ldots, z_i, y_{i+1}, \ldots, y_k) = \\ &\qquad (1 - z_i) q_{k,i-1}(z_1, \ldots, z_{i-1}, 0, y_{i+1}, \ldots, y_k) \\ &\qquad + z_i q_{k,i-1}(z_1, \ldots, z_{i-1}, 1, y_{i+1}, \ldots, y_k) \end{aligned}$$

Clearly we can define each of the q_{ki} using the AB rules. Moreover, restricted to $\{0,1\}^k$, p_k agrees with each q_{ki}. (In particular, on $\{0,1\}^n$, the values of all these polynomials lie in $\{0,1\}$.) This process does not increase the degree of any of the variables. Also $q_k := q_{kk}$ is a multilinear function. We then replace all occurrences of the y_i's by z_i's in all p_v's for $v > k$ and have the p_v applied to q_k instead of p_k.

Rule (4) will be applied to multilinear functions only and therefore the degree of any variable will always be at most two. The final polynomial q_r will be multilinear and will agree with p_k on $\{0,1\}^n$. $\Box$

The proof of Theorem 4.1 will now work directly for these modified AB's to yield:

Theorem 9.3. (Shamir) *Every language computable in polynomial space has an interactive proof.*

Note that in the proof of Theorem 4.1 we do the computation modulo a suitable prime (not too large), so we don't need to worry about the blowup of the sizes of numbers involved.

Now we can state the main result of this section.

Theorem 9.4. *For a function $f : \{0,1\}^* \to \mathbf{Z}$, the following are equivalent.*

(a) f is PSPACE-computable.

(b) There exists a uniform family of arithmetic programs with binary substitution P_n such that for every string $x \in \{0,1\}^*$,

$$f(x) = \tilde{P}_{|x|}(x).$$

(c) There exists a uniform family of AB's as described under (b) with the additional properties that each variable has degree at most two in every polynomial in the straight line programs; and the last polynomial of each straight line program, $\tilde{P}_n$, is multilinear.

Sketch of Proof. The equivalence of (b) and (c) follows from the Lemma. We only need to prove the equivalence of (a) and (b).
I. Suppose f is a PSPACE-computable function and $|f(x)| \leq 2^{q(|x|)}$ for some polynomial $q(|x|)$. Let $h(x,i)$ be the ith lowest order bit of $f(x)$. Let $n(x) = 0$ if $f(x) < 0$ and $n(x) = 1$ otherwise. Since h and n are PSPACE-computable and zero-one valued, by Theorem 9.1 we have IAB's and thus AB's computing h and n. We define a set of functions $g_0, \ldots, g_{p(|x|)}$ by the following top-to-bottom recurrence:

$$\begin{aligned} g_k(y_1,\ldots,y_k) &= g_{k+1}(y_1,\ldots,y_k,0) \\ &+ 2^{2^k} g_{k+1}(y_1,\ldots,y_k,1) \\ g_{p(|x|)}(y_1,\ldots,y_{p(|x|)}) &= h(x, y_1 \ldots y_{p(|x|)}) \end{aligned}$$

Note that we can create each g_k using an AB and $f(x) = (2n(x)-1)g_0$.
II. We can compute the sum and product of two n-bit numbers in $O(\log n)$ space. We can adapt these algorithms to give PSPACE algorithms to compute the values of AB's.

Suppose we have an AB of polynomials $\{p_1,\ldots,p_t\}$. Let $g(k,i)$ be the ith bit of the value of p_k. We can compute $g(k,i)$ recursively in terms of $g(j,r)$ for $j < k$ using the well known logspace algorithms. Since we have a potentially exponential number of bits we will need polynomial space to compute these values. A little care will show we can compute all the bits of $P_{|x|}(x)$ in polynomial space. □

Acknowledgments

The first author was partially supported by NSF Grant CCR-8710078. The second author is partially supported by NSF Grant CCR-9009936.

References

[1] L. Babai, Trading group theory for randomness, in *Proc. 17th Ann. ACM Symp. Theory of Computing*, 1985, 421-429.

[2] L. Babai, E-mail and the unexpected power of interaction, in *Proc. 5th Ann. IEEE Structures in Complexity Theory Conf.*, 1990, 30-44.

[3] L. Babai and L. Fortnow, A characterization of #P by arithmetic straight line programs, in *Proc. 31st Ann. IEEE Symp. Foundations of Comp. Sci.*, 1990, 26-34.

[4] L. Babai, L. Fortnow, and C. Lund, Nondeterministic exponential time has two-prover interactive protocols, in *Proc. 31st Ann. IEEE Symp. Foundations of Comp. Sci.*, 1990, 16-25.

[5] L. Babai and S. Moran, Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes, *Journal Comp. Sys. Sci.* **36** (1988), 254-276.

[6] D. Beaver and J. Feigenbaum, Hiding instances in multioracle queries, in *Proc. 7th Symp. on Theoretical Aspects of Comp. Sci.*, *Lecture Notes in Comp. Sci.* **415** (1990), 37-48.

[7] R. Beigel, J. Gill, and U. Hertrampf, Counting classes: thresholds, parity, mods, and fewness, in *Proc. 7th Symp. on Theoretical Aspects of Comp. Sci.*, *Lecture Notes in Comp. Sci.* **415** (1990), 49-57.

[8] M. Blum, Designing programs that check their work, submitted to *Comm. of ACM.*

[9] M. Blum and S. Kannan, Designing programs that check their work, in *Proc. 21st Ann. ACM Symp. Theory of Computing*, 1989, 86-97.

[10] M. Blum, M. Luby, and R. Rubinfeld, Self-testing and self-correcting programs, with applications to numerical programs, in *Proc. 22nd Ann. ACM Symp. Theory of Computing*, 1990, 73-83.

[11] R. Beigel, N. Reingold and D. Spielman, PP is Closed under Intersection, in *Proc. 23rd Ann. ACM Symp. Theory of Computing*, 1991, to appear.

[12] A. Chandra, D. Kozen, and L. Stockmeyer, Alternation, *J. Assoc. Comput. Mach.* **28** (1981), 114-133.

[13] J. Feigenbaum and L. Fortnow, On the random-self-reducibility of complete sets, *University of Chicago Technical Report* **90-22**, 1990.

[14] S. Fenner, L. Fortnow, and S. Kurtz, Gap-definable counting classes, *University of Chicago Technical Report* **90-32**, 1990.

[15] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., 1979

[16] S. Goldwasser, S. Micali, and C. Rackoff, The knowledge complexity of interactive proofs, *SIAM Journal on Computing* **18** (1989), 186-208. (Preliminary version appeared in *Proc. 18th Ann. ACM Symp. Theory of Computing*, 1985, 291-304.)

[17] S. Goldwasser and M. Sipser, Private coins versus public coins in interactive proof systems, in *Randomness in Computation*, S. Micali, ed., *Advances in Computing Research* **5**, JAI Press, 1989, 73-90.

[18] O. Goldreich, S. Micali, and A. Wigderson, Proofs that yield nothing but their validity and a methodology of cryptographic protocol design, in *Proc. 27th Ann. IEEE Symp. Foundations of Comp. Sci.*, 1986, 174-187.

[19] R. Lipton, New directions in testing, in *Proceedings of the DIMACS Workshop on Distributed Computing and Cryptography*, 1989, to appear.

[20] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, Algebraic methods for interactive proof systems, in *Proc. 31st Ann. IEEE Symp. Foundations of Comp. Sci.*, 1990, 1-10.

[21] A. A. Razborov, Lower bounds for the size of circuits of bounded depth with a complete basis including the logical addition function (in Russian), *Matem. Zametki* **41** (1981), 598-607. (English translation in *Math. Notes of the Acad. Sci. USSR* **41:4**, 333-338.)

[22] U. Schöning, Probabilistic complexity classes and lowness, in *Proc. 2nd Ann. IEEE Structure in Complexity Theory Conf.*, 1987, 2-8.

[23] A. Shamir, IP = PSPACE, in *Proc. 31st Ann. IEEE Symp. Foundations of Comp. Sci.,* 1990, 11-15.

[24] R. Smolensky, Algebraic methods in the theory of lower bounds for Boolean circuit complexity, in *Proc. 19th Ann. ACM Symp. Theory of Computing,* 1987, 77-82.

[25] L. Stockmeyer, The Polynomial-time hierarchy, *Theoretical Computer Science* **3** (1977), 1-22.

[26] S. Toda, On the computational power of PP and $\oplus$P, in *Proc. 30th Ann. IEEE Symp. Foundations of Comp. Sci.*, 1989, 514–519.

[27] L. Valiant, The complexity of computing the permanent, *Theoretical Computer Science* **8** (1979), 189-201.

[28] L. Valiant and V. Vazirani, NP is as easy as detecting unique solutions, *Theoretical Computer Science* **47** (1986), 85-93.

[29] H. Venkateswaran, Circuit definitions of nondeterministic complexity classes, in *Proc. 8th FST & TCS*, *Lecture Notes in Comp. Sci.* **338** (1988), 175-192.

[30] V. Zankó, ♯P-completeness via many-one reductions, *University of Chicago Tech. Report* **90-18**, 1990.

Manuscript received 30 March 1990

László Babai
University of Chicago
Chicago, IL 60637
and
Eötvös University, Budapest, Hungary
laci@cs.uchicago.edu

Lance Fortnow
Department of Computer Science
University of Chicago
1100 E. 58th St.
Chicago, IL 60637
fortnow@cs.uchicago.edu