Theoretical Computer Science 524 (2014) 90–102

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Monomials, multilinearity and identity testing in simple read-restricted circuits [☆]

Meena Mahajan [a], B.V. Raghavendra Rao [b], Karteek Sreenivasaiah [a]

[a] *The Institute of Mathematical Sciences, Chennai, India*
[b] *Indian Institute of Technology Madras, Chennai, India*

ARTICLE INFO

Article history:
Received 10 May 2013
Received in revised form 21 October 2013
Accepted 8 January 2014
Communicated by O. Watanabe

Keywords:
Arithmetic circuits
Computational complexity

ABSTRACT

We study the problem of testing if the polynomial computed by an arithmetic circuit is identically zero. We give a deterministic polynomial time algorithm for this problem when the inputs are read-twice or read-thrice formulas. In the process, these algorithms also test if the input circuit is computing a multilinear polynomial.

We further study three related computational problems on arithmetic circuits. Given an arithmetic circuit C, (1) ZMC: test if a given monomial in C has zero coefficient or not, (2) MonCount: compute the number of monomials in C, and (3) MLIN: test if C computes a multilinear polynomial or not. These problems were introduced by Fournier, Malod and Mengel (2012) [11], and shown to characterise various levels of the counting hierarchy (CH).

We address the above problems on read-restricted arithmetic circuits and branching programs. We prove several complexity characterisations for the above problems on these restricted classes of arithmetic circuits.

1. Introduction

A fundamental question one can ask concerning a given arithmetic circuit is: does the circuit compute the identically zero polynomial? This is the well-known Polynomial Identity Testing problem PIT, that has spurred an enormous amount of research in the last two decades. A complete randomisation of black-box PIT even for the case of depth three arithmetic circuits implies circuit lower bounds [12,16].

Today, there are two frontiers for identity testing. One is based on the (alternation) depth of the circuit. Deterministic identity testing algorithms are known for depth-2 circuits, for depth-3 circuits with restrictions on the top fanin, and for restricted depth-4 circuits. (See [1] and the references therein.) As indicated by [12], improving this to arbitrary depth-3 circuits will be a major breakthrough.

The other frontier is concerned with formulas. Restricting fanout in a circuit to 1 yields formulas; further restricting formulas to allow each variable at no more than k leaves yields read-k formulas. The simplest kind of formulas are read-once formulas ROFs: every variable appears at most once. Deterministic polynomial-time algorithms for PIT on such formulas are trivial. Going beyond these for $k > 1$, one breakthrough in [19] shows how to test k-sums of ROFs: for each $k \in O(1)$, PIT can be efficiently performed on a sum of k ROFs. However, not every read-k formula can be expressed as a sum of k ROFs. Along this thread, the next improvement in [5] shows how to do identity testing on read-k formulas that are known to be multilinear, that is, the polynomials computed at each node are multilinear.

[☆] A preliminary version of this paper appeared in MFCS'12 (see [18]). This research is partially supported by Indo-German Max Planck Center (IMPECS).
E-mail addresses: meena@imsc.res.in (M. Mahajan), bvrr@cse.iitm.ac.in (B.V.R. Rao), karteek@imsc.res.in (K. Sreenivasaiah).

0304-3975/$ – see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.tcs.2014.01.005

To use the algorithm from [5] for a read-k formula, we first need to check whether it is multilinear. Multilinearity testing is as hard as PIT in general [11], but for read-k formulas, it could conceivably be easier. Thus one way to extend the result of [5] to arbitrary read-k formulas is to develop a multilinearity test for such formulas.

Our main results are a multilinearity and identity test for read-twice and read-thrice formulas. Our tests are intertwined with a PIT algorithm for subformulas. We give a deterministic polynomial-time algorithm that simultaneously decides whether an R2F is multilinear and whether it is identically zero (Theorem 4). It uses the sum-of-k-ROFs test from [19] on some subformulas as well as on some formulas obtained by transforming subformulas of the input formula. Thus it is inherently a non-blackbox algorithm; so is the polynomial-time algorithm from [19]. In Theorem 5, we give a multilinearity and identity test for read-thrice formulas. However, this does not gather as much information as the test for read-twice. Hence we feel that the multilinearity test for the read-twice case can potentially be extended to formulas that read variables more than twice.

PIT algorithms check whether the polynomial computed by the circuit has at least one monomial. Natural generalisations/variants of this question are (1) MonCount: compute the number of monomials in the polynomial computed by a given circuit, and (2) ZMC: decide whether a given monomial has zero coefficient in the polynomial computed by a given circuit. ZMC was introduced by Koiran and Perifel [17]. More recently, Fournier, Malod and Mengel [11] showed that ZMC and MonCount characterise certain levels of the counting hierarchy (CH, the hierarchy based on the complexity classes PP and $C_=P$). In fact, MonCount remains hard even if restricted to formulas. They also show that if the circuits compute multilinear polynomials, then these problems become easier (equivalent to PP and PIT respectively), and that multilinearity checking itself is equivalent to PIT. All these results from [11] are in the non-black-box model, where the circuit is given explicitly in the input.

Since PIT on read-k formulas appears easier, naturally one could ask whether MonCount and ZMC become easier as well? We observe that this is not the case: even for monotone (no negative constants) read-twice formulas, MonCount is #P-hard. This further leads us to the investigation: where exactly does hardness for MonCount and ZMC begin? Further, translating the classes between NP and PSPACE down to classes below P, can we show that on restricted circuits, MonCount and ZMC are complete for the translated classes?

Starting with ROFs, we show (Theorem 7) that MonCount for ROFs is in the $GapNC^1$-hierarchy, i.e. the AC^0-closure of $GapNC^1$, where $GapNC^1$ is the class of Boolean problems that can be computed by arithmetic formulas over the integers with constants $0, 1, -1$. The $GapNC^1$-hierarchy is an intriguing class that lies between NC^1 and DLOG and has been studied extensively in the last two decades; see for instance [2]. We also show that ZMC for ROFs is in logspace (Theorem 18). It is straightforward to see that ZMC for ROFs is hard for $C_=NC^1$, so this is almost tight. (The "gap" between Boolean NC^1, $C_=NC^1$, $GapNC^1$ and DLOG is very small.)

Another equally natural and well-studied restriction is when the circuit is an algebraic branching program BP with edges labelled by the allowed constants or by variables. Evaluation of BPs on Boolean-valued inputs is complete for the arithmetic class GapL, the logspace analogue of the class GapP. The GapL hierarchy (the AC^0 closure of GapL) is known to be contained in $\log n$ depth threshold circuits TC^1 and hence in $\log^2 n$ depth Boolean circuits NC^2. Two restrictions on BPs, in order of increasing generality, are: (1) occur-once BPs, or OBPs, where each variable appears at most once anywhere in the BP, these subsume ROFs, and (2) multilinear BPs, or MBPs, where the polynomial computed at every node is multilinear. Again, deterministic algorithms are known for ACIT on OBPs [14]. We show that MonCount for OBPs is in the GapL hierarchy (Theorem 15), while ZMC for OBPs and even MBPs is complete for the complexity class $C_=L$ (Theorem 17). (As a comparison, a well-known complete problem for $C_=L$ is testing singularity of an integer matrix [3].)

A related problem explored in [11] as a tool to solving MonCount is that of checking, given a circuit C and monomial m, whether C computes any monomial that extends m. Denote this problem ExistExtMon. Though our algorithms for MonCount do not need this subroutine, we also show that for OBPs (and hence for ROFs), ExistExtMon lies in the GapL hierarchy (Theorem 19).

2. Preliminaries

Circuits, formulas, branching programs, polynomials Let $X = \{x_1, \dots, x_n\}$ be a set of variables. An *arithmetic circuit* C over a ring R is a directed acyclic graph with internal nodes labelled $+$ or $\times$ and leaves labeled from $X \cup R$. Every node has in-degree zero or two, and there is exactly one node of out-degree zero, called the output gate. Unless otherwise stated, we consider R to be the ring of integers $\mathbb{Z}$, and we allow only the constants $\{-1, 0, 1\}$ in the circuits. An *arithmetic formula* F is an arithmetic circuit where fan-out for every gate is at most one.

The depth of a circuit is the length of a longest root-to-leaf path. The alternation-depth is the maximum number of alternations between $+$ and $\times$ gates along any root-to-leaf path. In the literature on identity testing, depth is used to mean alternation-depth. However we differentiate between these, as is done in uniform circuit complexity literature, because bounded fanin is crucial to some of our algorithms. Note that converting a circuit to a bounded fanin circuit increases only the depth, not the size or the alternation depth.

Every node in C computes a polynomial in $R[x_1, \dots, x_n]$ in a natural way. Let g be a gate in a circuit (or formula) C. We denote by p_g the polynomial computed at gate g of C. We denote by p_C the polynomial p_r, where r is the output gate of C. Let $\mathrm{var}_g \triangleq \{x_i \mid \text{some descendant of } g \text{ is a leaf labelled } x_i\}$.

A *read-once* arithmetic formula (ROF for short) is an arithmetic formula where each variable occurs at most once as a label. More generally, in a *read-k* arithmetic formula a variable occurs at most k times as a label.

An algebraic branching program (ABP for short) over a ring R is an undirected acyclic graph B with edges labeled from $\{x_1,\dots,x_n\}\cup R$, and with two designated nodes, s with zero in-degree, and t with zero out-degree. For any directed path ρ in B, define

$$\mathsf{weight}(\rho)=\prod_{e\text{: an edge in }\rho}\mathsf{label}(e).$$

Any pair of nodes u, v in B computes a polynomial in $R[x_1\dots x_n]$ defined as follows:

$$p_B(u,v)=\sum_{\rho\text{: }\rho\text{ is a }u\rightsquigarrow v\text{ path in }B}\mathsf{weight}(\rho).$$

The ABP B computes the polynomial $p_B\stackrel{\triangle}{=}p_B(s,t)$. We drop the subscript B from the above when clear from context.

We consider the following restrictions of ABPs in increasing order of generality: (1) occur-once ABPs (OBP for short), where each variable appears at most once anywhere in the ABP (such BPs generalise ROFs), (2) read-once ABPs, or RBPs, where no path has two occurrences of the same variable, and (3) multilinear BPs, or MBPs, where the polynomial computed at every node is multilinear.

Complexity classes For all the standard complexity classes, the reader is referred to [6]. We provide below the definitions of some of the less-familiar complexity classes that are used in the paper. Let $f=(f_n)_{n\geqslant 0}$ be a family of integer valued functions $f_n:\{0,1\}^n\to\mathbb{Z}$. f is in the complexity class GapL exactly when there is some non-deterministic logspace machine M such that for every x, $f(x)$ equals the number of accepting paths of M on x minus the number of rejecting paths of M on x. $\mathsf{C_=L}$ is the class of languages L such that for some $f\in\mathsf{GapL}$, for every x, $x\in L\Leftrightarrow f(x)=0$. The GapL hierarchy, at an intuitive level, can be seen as classes of functions built over bit access to other functions, with a deterministic logspace machine at the base and access to GapL functions at the first level. It is known to be contained in NC^2. For technical complications that arise in the definition of the GapL hierarchy, the reader is referred to [4,3] or [7].

GapNC^1 denotes the class of families of functions $(f_n)_{(n\geqslant 0)}$, $f_n:\{0,1\}\to\mathbb{Z}$, where f_n can be computed by a uniform polynomial size log depth arithmetic circuit. This equals the class of functions computed by uniform polynomial-sized arithmetic formulas [9]. $\mathsf{C_=NC}^1$ is the class of languages L such that for some GapNC^1 function family $(f_n)_{n\geqslant 0}$, and for every x, $x\in L\iff f_{|x|}(x)=0$. The GapNC^1 hierarchy comprises of languages accepted by polynomial-size constant depth unbounded fanin circuits (AC^0) with oracle access to bits of GapNC^1 functions. It follows from the results of [13] that the hierarchy is contained in DLOG.

Miscellaneous notation A monomial is represented by the sequence of degrees of the variables. For instance, over x_1, x_2, x_3, the monomial x_1^2 is represented as $(2,0,0)$, and the monomial x_1x_3 is represented as $(1,0,1)$. For a degree sequence $m=(d_1,\dots,d_n)$ we denote the monomial $\prod_{i=1}^n x_i^{d_i}$ by X^m. For any set $S\subseteq[n]$, we denote by m_S the multilinear monomial $\prod_{i\in S}x_i$. For a monomial m and polynomial p, $\mathsf{coeff}(p,m)$ denotes the coefficient of m in p. [Statement S] is a Boolean 0–1 valued predicate that takes value 1 exactly when the statement S is true.

We now describe the computational problems considered in this paper.

MonCount: Given an arithmetic circuit C over $\mathbb{Z}$, compute the number of monomials in the polynomial computed by C.
MLIN: Given an arithmetic circuit C over $\mathbb{Z}$, test if the polynomial p_C is multilinear or not.
ZMC: Given an arithmetic circuit C over $\mathbb{Z}$, and a monomial m, test if $\mathsf{coeff}(p_C,m)=0$ or not.
ExistExtMon: Given an arithmetic circuit C over Z, and a monomial m, test if there is a monomial M with non-zero coefficient in p_C such that M extends m; that is, $m|M$.

Note that for a polynomial p, when the input monomial is a single variable x_i, ExistExtMon reduces to checking if the partial derivative of p w.r.t. x_i is identically zero.

The following propositions list some of the known results used in the paper.

Proposition 1. *(Follows from [9,13].) Evaluating an arithmetic formula where the leaves are labelled $\{-1,0,1\}$ is in* DLOG *(even* GapNC^1*).*

Proposition 2. *(See [19].) Given k ROFs in n variables, there is a deterministic (non-black-box) algorithm that checks whether they sum to zero or not. The running time of the algorithm is $n^{O(k)}$.*

The following result can be obtained by easy reductions to known log-space complete problems [10].

Proposition 3. *(See [10].) The following problems are in* DLOG:

(1) *Given a formula F, a gate $g \in F$, and a variable x, checking whether $x \in \mathsf{var}_g$.*
(2) *Given a rooted tree T, and two nodes u, v, find lowest common ancestor (LCA) of u and v.*

3. Multilinearity and identity tests

In this section we consider read-twice formulas, and the problems of testing multilinearity MLIN and testing identically zero PIT on read-2 and read-3 formulas. We first look at the read-2 case.

3.1. Read-2 formulas

The individual degree of a variable in a polynomial p computed by read-twice formula F is bounded by two. Thus, multilinearity testing boils down to testing if the second order partial derivative of F with respect to x_i is zero for every variable x_i. Note that the second-order partial derivative of F with respect to x_i is a polynomial in $n-1$ variables; thus MLIN reduces to n instances of PIT on $n-1$ variables. Our approach is to use the inductive structure of a read-twice polynomial to test these partial derivatives for zero, using the knowledge of multilinearity of gates at the lower levels. As an aid in this computation, we also check, for each gate g and each variable x, whether x survives in p_g.

Theorem 4. *For read-twice formulas, the problems* PIT, $\mathsf{ExistExtMon}(\phi, x)$, *and* MLIN *(where ϕ is the input formula and x is a single variable in it) are in P.*

Proof. Let ϕ be the given read-twice formula on variables $x_1, \dots, x_n$, with S internal nodes. Without loss of generality, assume that ϕ is strictly alternating. That is, inputs to a $+$ gate are either leaves or are $\times$ gates, and inputs to a $\times$ gate are either leaves or are $+$ gates.

We proceed by induction on the structure of the formula ϕ.

We iteratively compute, for each gate g in ϕ and each variable $x \in X$, the following set of 0–1 valued functions:

$$\begin{aligned}
\mathsf{PIT}(g) = 1 \quad &\Leftrightarrow \quad p_g \equiv 0,\\
\mathsf{MLIN}(g) = 1 \quad &\Leftrightarrow \quad p_g \text{ is multilinear},\\
\mathsf{ExistExtMon}(g, x) = 1 \quad &\Leftrightarrow \quad p_g \text{ has a monomial } m \text{ that contains } x.
\end{aligned}$$

We say that x survives in g if $\mathsf{ExistExtMon}(g, x) = 1$.

The base case is when ϕ is a single variable or a constant. That is, ϕ consists of a single gate g that is labelled $L \in \{x_1, \dots, x_n\} \cup \{0, +1, -1\}$. Then $\mathsf{PIT}(g) = 1$ if and only if $L = 0$, $\mathsf{MLIN}(g) = 1$ always, and $\mathsf{ExistExtMon}(g, x) = 1$ if and only if $L = x$.

Now assume that for every gate u below the root gate of ϕ, the above functions have been computed and stored as bits. Let f be the root gate of ϕ. We show how to compute these functions at f. The order in which we compute them depends on whether f is $\times$ or a $+$ gate.

First, consider $f = g \times h$. We compute the functions in the order given below.

1. $\mathsf{PIT}(f)$: f is identically zero if and only if at least one of g, h is. Thus $\mathsf{PIT}(f) = \mathsf{PIT}(g) \vee \mathsf{PIT}(h)$.
2. $\mathsf{MLIN}(f)$: If f is identically zero, then it is vacuously multilinear. Otherwise, for it to be multilinear, it must be the product of two (non-zero) multilinear polynomials in disjoint sets of variables. Thus

$$\mathsf{MLIN}(f) = \mathsf{PIT}(f) \vee \left[\mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h) \wedge \left(\bigwedge_{x \in X} \left[\neg\mathsf{ExistExtMon}(g, x) \vee \neg\mathsf{ExistExtMon}(h, x)\right]\right)\right].$$

   Note that the $\mathsf{PIT}(f)$ term is necessary, since f can be multilinear even if, for instance, g is not, provided $h \equiv 0$.
3. $\mathsf{ExistExtMon}(f, x)$: x appears in p_f if and only if $p_f \not\equiv 0$ and x appears in at least one of p_g, p_h. Thus

$$\mathsf{ExistExtMon}(f, x) = \neg\mathsf{PIT}(f) \wedge \left[\mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x)\right].$$

Next, consider $f = g + h$. We compute the functions in the order given below.

1. $\mathsf{MLIN}(f)$: Since f is read-twice, a non-multilinear monomial in g cannot get cancelled by a non-multilinear monomial in h; that would require at least 4 occurrences of some variable. Thus, f is multilinear only if both g and h are. The converse is trivially true. Thus $\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h)$.
2. $\mathsf{ExistExtMon}(f, x)$: This is the non-trivial part of this proof; we defer the description to a bit later.

3. $\mathsf{PIT}(f)$: Once we compute the functions above, this is straightforward:

$$\mathsf{PIT}(f) = \bigl[f(0) = 0\bigr] \wedge \bigwedge_{x \in X} \neg\mathsf{ExistExtMon}(f, x).$$

Checking if $f(0) = 0$ is feasible; see Proposition 1.

We now complete the description for computing $\mathsf{ExistExtMon}(f, x)$ when $f = g + h$. If x survives in neither g nor h, then it does not survive in f. But if it survives in exactly one of g, h, it cannot get cancelled in the sum, so it survives in f. Thus

$$\begin{aligned} \mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x) = 0 \quad &\Longrightarrow \quad \mathsf{ExistExtMon}(f, x) = 0, \\ \mathsf{ExistExtMon}(g, x) \oplus \mathsf{ExistExtMon}(h, x) = 1 \quad &\Longrightarrow \quad \mathsf{ExistExtMon}(f, x) = 1. \end{aligned}$$

So now assume that x survives in both g and h. We can write the polynomials computed at g, h as $p_g = \alpha x + \alpha'$ and $p_h = \beta x + \beta'$, where α', β' do not involve x; and we know that $\alpha \not\equiv 0$, $\beta \not\equiv 0$. We want to determine whether $\alpha + \beta \equiv 0$.

Since x appears in V_g and V_h, and since f is read-twice, we conclude that x is read exactly once each in g and in h. Hence α, β also do not involve x.

We construct a formula computing α as follows: In the sub-formula rooted at g, let ρ be the unique path from x to g. For each $+$ gate u on the path ρ, let u' be the child of u not on ρ; replace u' by the constant 0. Thus we retain only the parts that multiply x; that is, we compute αx. Setting $x = 1$ gives us a formula G computing α. A similar construction with the formula rooted at h gives a formula H computing β. Set $F = G + H$. Note that F is also a read-twice formula, and it computes $\alpha + \beta$. Thus in this case $\mathsf{ExistExtMon}(f, x) = 1 \Leftrightarrow \mathsf{PIT}(F) = 0$, so we need to determine $\mathsf{PIT}(F)$.

Let Y denote the set of variables appearing in F; $Y \subseteq X \setminus \{x\}$. Partition Y:

A: variables occurring only in G;
B: variables occurring only in H;
C: variables occurring in G and H.

If $A \cup B = \emptyset$, then $Y = C$, and each variable in F appears once in G and once in H. That is, both G and H are read-once formulas. We can now determine $\mathsf{PIT}(F)$ in time polynomial in the size of F using Proposition 2.

If $A \cup B \neq \emptyset$, then let $y \in A$. If y survives in G, it cannot get cancelled by anything in H, so it survives in F and $F \not\equiv 0$. Similarly, if any $y \in B$ survives in H, then $F \not\equiv 0$. We briefly defer how to determine this and complete the high-level argument. If no $y \in A$ survives in G, and no $y \in B$ survives in H, then let $F' = G' + H'$ be the formula obtained from F, G, H by setting variables in $A \cup B$ to 0. Clearly, the polynomial computed remains the same; thus $\alpha + \beta = p_F = p_F|_{A \cup B \leftarrow 0} = p_{F'}$. But F' satisfies the previous case (with respect to F', $A' \cup B' = \emptyset$), and so we can use Proposition 2 as before to determine $\mathsf{PIT}(F') = \mathsf{PIT}(F)$.

What remains is to describe how we determine whether a variable $y \in A$ survives in G. (The situation for $y \in B$ surviving in H is identical.) We exploit the special structure of G: there is a path ρ where all the $+$ gates have one argument 0 and the path ends in a leaf labeled 1. Let $\mathcal{T} = \{T_1, \ldots, T_\ell\}$ be the subtrees hanging off the $\times$ gates on ρ; let u_i be the root of T_i. Note that each $T_i \in \mathcal{T}$ is a sub-formula of our input formula ϕ, and hence by the iterative construction we know the values of the functions PIT, MLIN, $\mathsf{ExistExtMon}$ at gates in these sub-trees. In fact, we already know that $\mathsf{PIT}(u_i) = 0$ for all i, since we are in the situation where $\alpha \not\equiv 0$, and $\alpha = \prod_{i=1}^{\ell} p_{u_i}$. Hence, if y appears in just one sub-tree T_i, then $\mathsf{ExistExtMon}(G, y) = \mathsf{ExistExtMon}(u_i, y)$. If y appears in two sub-trees T_i, T_j, then $\mathsf{ExistExtMon}(G, y) = \mathsf{ExistExtMon}(u_i, y) \vee \mathsf{ExistExtMon}(u_j, y)$. $\square$

A question that arises naturally here is whether this algorithm is optimal, or whether the PIT problem for read-twice formulas is in some class smaller than P. Note that the input formula F can be re-structured into an equivalent log-depth formula F', as described in [8,9]. If the resulting formula is also read-twice, then it appears that the above algorithm can be applied to F', with a careful implementation to keep track of partial values, to yield an upper bound in NC. However, we have not examined the possibility of such an implementation, because it is not at all clear that the depth restructuring does actually preserve the number of times a variable is read.

3.2. Read-3 formulas

The algorithm in the previous section crucially uses the PIT algorithm from [19] for k-sum-of-ROFs. A stronger result due to [5] gives PIT algorithms for read-k formulas that compute multilinear polynomials at each node. Using this algorithm instead of [19], we obtain poly-time PIT and MLIN tests for read-thrice (as opposed to read-twice) formulas. However, we pay a cost: we can no longer check at every node g whether a variable survives at g (the bit $\mathsf{ExistExtMon}(g, x)$). We can compute this information only at nodes g where all descendants compute multilinear formulas. The fact that we can compute $\mathsf{ExistExtMon}(g, x)$ everywhere in the read-twice case may be of independent interest (it seems to be a useful fact where enumerating monomials is concerned). In the following, we prove that for read-3 formulas, PIT and MLIN are in P:

Theorem 5. *Given a read-thrice formula F with leaves labeled by variables from $X = \{x_1, \dots, x_n\}$ or constants from $\{-1, 0, 1\}$ and nodes labeled $+$ or $\times$, there is an efficient deterministic algorithm that decides if F computes the identically zero polynomial, and if not, whether it computes a multilinear polynomial.*

Proof. Algorithm Idea: We proceed bottom-up, processing nodes of the formula, collecting as much information as possible/necessary about the polynomial computed at each node. The type of information collected for a node g could be: $\mathsf{MLIN}(g), \mathsf{PIT}(g), \mathsf{ExistExtMon}(g, x)$ for $x \in X$.

For nodes g computing multilinear polynomials, we will compute all this information.

For nodes where we detect non-multilinearity (and hence know that the polynomial is not identically zero), we will not compute any additional information.

We repeatedly use collected information to prune the formula. For instance, we ensure that no leaf is labeled 0 by moving the zeroes up (replace $g + 0$ by g; $g \times 0$ by 0). We ensure that for each non-leaf node g, $\mathsf{var}(g) \neq \emptyset$ (replace a node adding or multiplying constants by a leaf labeled with the resulting value). Note that the resulting formulas can have any constants from $\mathbb{Z}$ at the leaves.

Further, for nodes g where we determine that the identically zero formula is computed, we will cut away the subformula rooted at g, replacing it by a leaf labelled zero, and then eliminate the zero-leaf as discussed above. Thus a node that is processed and not deleted necessarily computes a non-zero polynomial.

We will also maintain the following property: for nodes g where we determine that a multilinear formula is computed, the subformula rooted at g computes multilinear formulas at each node.

Assume that we have a pruned formula. At a leaf, the required information is trivial to compute. Consider a node f where the information has been computed at the children of f.

Case 1. $f = g \times h$. $\mathsf{PIT}(f) = \mathsf{PIT}(g) \vee \mathsf{PIT}(h)$. However, by the pruning we have described above, we know that $g, h \not\equiv 0$ and so $f \not\equiv 0$, $\mathsf{PIT}(f) = 0$.
Since $f \not\equiv 0$, if either of g, h is non-multilinear then so is f. If both g and h are multilinear, then f is multilinear if and only no variable survives in both g and h. Thus we can compute $\mathsf{MLIN}(f)$ from the information at g, h:

$$\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h) \wedge \bigwedge_{x \in X} \bigl(\neg\mathsf{ExistExtMon}(g, x) \vee \neg\mathsf{ExistExtMon}(h, x)\bigr).$$

When f is multilinear, we need to compute the auxiliary information as well. Note that we have already ensured that all nodes below g and h compute multilinear polynomials, so this property is already true for f. For any $x \in X$, $\mathsf{ExistExtMon}(f, x) = \mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x)$.

Case 2. $f = g + h$. Computing $\mathsf{MLIN}(f)$: Since the formula is read-thrice, if any one of g, h (say g) is not multilinear and hence has an x^2 term for some $x \in X$, then this term cannot be cancelled by the other summand (say h) since h has at most one occurrence of x. So f is not multilinear. If g, h are both multilinear, then so is $g + h$. Thus

$$\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h).$$

Computing $\mathsf{PIT}(f)$: If f is not multilinear, then $f \not\equiv 0$ and so $\mathsf{PIT}(f) = 0$. In this case, we do not compute any further information abut f.
But if f is multilinear, we still need to check if $f \equiv 0$. We have already ensured that all nodes in the sub-formulas rooted at g, h and hence in the sub-formula rooted at f compute multilinear polynomials. And the sub-formula is read-thrice. So using [5], we can test whether $f \equiv 0$.
Computing the remaining information: If we detect that a multilinear f is identically 0, we replace the subformula rooted at f by 0 and move the constants up as far as possible.
If we detect that f is multilinear but $f \not\equiv 0$, then we need to compute the bits $\mathsf{ExistExtMon}(f, x)$. By multilinearity, $f(X) = Ax + B$ where A, B do not use x. We want to know if $A \equiv 0$ (this is equivalent to $\mathsf{ExistExtMon}(f, x) = 0$). A is computed by the formula $f|_{x=1} - f|_{x=0}$. We have already ensured that all nodes in the sub-formulas rooted at g, h and hence in the sub-formula rooted at f compute multilinear polynomials. Thus the formula for A is multilinear and reads every variable at most 6 times. Using [5], we can test whether $A \equiv 0$. $\square$

4. Counting monomials

We now consider the MonCount problem. First we show that it is very hard even for read-twice formulas. Then we consider ROFs and OBPs. In both ROFs and OBPs, a monomial, once generated in a sub-formula/program, can be cancelled only by multiplication with a zero polynomial. We exploit this fact to obtain efficient algorithms for counting monomials in ROFs and OBPs.

4.1. *Hardness of* MonCount

We show that even for formulas that are monotone (no negative constants) and are read-twice, and furthermore, are decomposable as the sum of two read-once formulas, MonCount is as hard as #P.

Theorem 6. MonCount *is* #P *complete for the sum of two monotone read-once formulas.*

Proof. First we show hardness. Valiant showed [21] that the problem of computing the number of perfect matchings in a bipartite graph is #P hard. We reduce this to the problem of computing the number of monomials common to two monotone read-once alternation-depth two formulas. Then we reduce the latter problem to computing the number of monomials in the sum of two monotone alternation-depth two read once formulas.

Let $G = (U, V, E)$ be a bipartite graph with $|U| = |V| = n$. Let $X = \{x_{uv} \mid (u, v) \in E\}$. Define two polynomials

$$f = \prod_{u \in U} \Bigl(\sum_{v:\,(u,v) \in E} x_{uv} \Bigr); \qquad g = \prod_{v \in V} \Bigl(\sum_{u:\,(u,v) \in E} x_{uv} \Bigr).$$

Clearly, both f and g are computable by alternation-depth two read-once formulas. Consider a monomial m common to both f and g. Since m is in f, it contains, for each $u \in U$, exactly one variable of the form x_{uv}. Similarly, since m is in g, it contains, for each $v \in V$, exactly one variable of the form x_{uv}. Thus the set $\{(u, v) \mid x_{uv} \in m\}$ is a perfect matching in G. Conversely, any perfect matching M in G corresponds to a unique monomial $\prod_{(u,v) \in M} x_{uv}$ that is common to both f and g. Therefore, the number of perfect matchings in G is equal to the number of common monomials of f and g. Let $\#(f \cap g)$ denote the later number.

Since f and g are monotone formulas, adding them cannot result in any monomial cancellations. Thus

$$\#(f + g) = \#f + \#g - \#(f \cap g).$$

Since $\#f$ and $\#g$ are ROFs, $\#f$ and $\#g$ can be computed easily in P. (Theorem 7 below shows that in fact it can be computed in DLOG.) Hence, using the above relation, computing $\#(f \cap g)$ reduces to computing $\#(f + g)$, and the reduction is computable in polynomial time (even logspace).

To see the #P upper bound, consider $f + g$, where f and g are monotone ROFs. A monomial m appears in $f + g$ if and only if it appears in at least one of f, g. Now define a non-deterministic machine M as follows: M guesses a monomial m, computes $a = \mathsf{ZMC}(f, m)$ and $b = \mathsf{ZMC}(g, m)$, and accepts if $a \vee b = 1$. The number of accepting paths of M is exactly $\#(f + g)$. Since f and g are ROFs, a and b can be computed in polynomial time (Theorem 18 below shows that in fact it can be computed in DLOG). All potential monomials are multilinear and so can be guessed in polynomial time. Hence M is an NP machine, as required. □

4.2. Counting monomials in read-once formulas

Theorem 7. *Given a read-once formula F,* MonCount(p_F) *can be computed by an* AC^0 *circuit with oracle gates for* GapNC^1 *functions. Hence it can be computed in* DLOG.

We start with some notations. For any gate g in F, let $\#g$ denote the number of monomials in the polynomial p_g computed at g. (The constant term, even if non-zero, does not count as a monomial.) Define a 0–1 valued bit NZ(g), to indicate whether or not the constant term of p_g is zero, as follows:

$$\mathsf{NZ}(g) = \begin{cases} 1 & \text{if } p_g(0) \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 8. *The language L defined below is in* C$_=$NC^1:

$$L = \bigl\{ \langle F, g \rangle \bigm| F \text{ is an arithmetic formula}, \ g \text{ is a gate in } F, \text{ and } \mathsf{NZ}(g) = 0 \bigr\}.$$

Proof. Convert F to formula F' where all variables are set to 0, and g is the output gate. Then F' evaluates to $p_g(0)$, so we need to check if F' evaluates to 0. By Proposition 1, this check can be performed in C$_=$NC^1. □

Proof of Theorem 7. Since F is a read-once formula, we can compute the value of $\#f$ for each gate f inductively, based on the structure of F beneath f. When f is a leaf node, it is labelled 0 or ± 1 or x_i for some i.

$$\#(0) = \#(\pm 1) = 0; \qquad \#(x_i) = 1.$$

Now assume f is not a leaf. Suppose $f = g + h$, then g and h are variable-disjoint read-once formulas. Since the monomials of g and h are distinct,

$$\#f = \#g + \#h. \tag{1}$$

Finally, suppose $f = g \times h$, then again g and h are variable-disjoint. Each pair of monomials m in p_f and m' in p_g gives rise to a monomial mm' in p_f. In addition, if $p_g(0) \neq 0$, then each m' also appears as a monomial in p_f; similarly for $p_h(0)$

and m. Thus

$$\#f = [\#g \times \#h] + \big[\#g \times \mathsf{NZ}(h)\big] + \big[\mathsf{NZ}(g) \times \#h\big]. \tag{2}$$

Using Eqs. (1) and (2), we can transform the given read-once formula F to a new formula F' over $\mathbb{Z}$ that computes $\mathsf{MonCount}(F)$. The transformation is local, and can be done in AC^0 with oracle access to $\mathsf{C}_=\mathsf{NC}^1$. For each gate f in F the local transformation can be described as follows: If f is a leaf gate, then relabel f by $\#f$. If $f = g + h$, then apply Eq. (1). If $f = g \times h$, using Eq. (2) involves using $\#g$ and $\#h$ more than once, and so we do not get a formula. However, we can modify Eq. (2) so that $\#f$ gets the structure of a formula, with oracle access to NZ. We use the identity

$$\#(g \times h) = \big(\#g + \mathsf{NZ}(g)\big) \times \big(\#h + \mathsf{NZ}(h)\big) - \big(\mathsf{NZ}(g) \times \mathsf{NZ}(h)\big).$$

The values $\mathsf{NZ}(g)$ and $\mathsf{NZ}(h)$ can be obtained with oracle access to the language L defined in Lemma 8. Now $\#g$ and $\#h$ are used only once.

Thus, from F we construct a formula F'' where the leaves of F'' are labeled by constants $0, \pm 1$ or by the outputs of $\mathsf{C}_=\mathsf{NC}^1$ oracle gates. Equivalently, in $\mathsf{AC}^0(\mathsf{C}_=\mathsf{NC}^1)$, we can transform F to formula F' whose leaves are labeled by $0, \pm 1$. By construction, F' is variable-free, and $\#p_F = \mathsf{val}(F')$. By Proposition 1, $\mathsf{val}(F')$ can be computed in GapNC^1, completing the proof. $\square$

Remark 9. The AC^0 circuit constructed above needs oracle access mainly to $\mathsf{C}_=\mathsf{NC}^1$ gates, which check whether a GapNC^1 function is zero or not. Only the topmost oracle query requires the entire value of the GapNC^1 function.

For any polynomial p, $p \equiv 0$ if and only if the constant term of p is 0 and $\mathsf{MonCount}(p)$ is 0. Hence, from Theorem 7 and Lemma 8, we have the following:

Corollary 10. *In the non-blackbox setting,* PIT *on ROFs is in the* GapNC *hierarchy and hence in* DLOG.

4.3. Counting monomials in occur-once branching programs

We now show how to count monomials in OBPs. The approach used in Theorem 7 does not directly generalise to OBPs, i.e., knowing MonCount at immediately preceding nodes is not enough to compute MonCount at a given node in an OBP. However, since every variable occurs at most once in an OBP, every path generating a monomial should pass through one of these edges. This allows us to keep track of the monomials at any given node of the OBP, given the monomial count of all of its predecessors.

We begin with some notations. Let B be an occur-once BP on the set of variables X, and u, v be any nodes in B. Let $c(u, v)$ be the constant term in $p(u, v)$. We define the 0–1 valued indicator function that describes whether this term is non-zero:

$$\mathsf{NZ}(u, v) = \begin{cases} 1 & \text{if } c(u, v) \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

We cannot directly use the strategy we used for ROFs, since even in an OBP, there can be cancellations due to the constant terms. For instance, in the figure below, $\#p(s, b) = \#p(s, c) = 1$, but $\#p(s, t) = 0$.

$$\begin{tikzcd}
 & & b \arrow[dr, "1"] & \\
s \arrow[r, "x"] & a \arrow[ur, "1"] \arrow[dr, "-1"'] & & t \\
 & & c \arrow[ur, "1"'] &
\end{tikzcd}$$

We therefore identify edges critical for a polynomial. We say that edge $e = (w, u)$ of B *is critical to* v if

1. $\mathsf{label}((w, u)) \in X$; and
2. B has a directed path ρ from u to v consisting only of edges labeled by $\{-1, 1\}$.

We have the following structural property for the monomials in $p(s, v)$:

Lemma 11. *In an occur-once OBP B with start node s, for any node v in B,*

$$p(s, v) = c(s, v) + \sum_{(w,u) \text{ critical to } v} p(s, w) \cdot \mathsf{label}(w, u) \cdot c(u, v).$$

Proof. First, note that if edges $(w,u) \neq (w',u')$ are both critical to v, then the monomials in $p(s,w) \cdot \mathsf{label}(w,u)$ and $p(s,w') \cdot \mathsf{label}(w',u')$ will be disjoint, because P is occur-once. (The variables labelling (w,u) and (w',u') make the monomials distinct.) Moreover, for any monomial m in $p(s,v)$, there is exactly one critical edge (w,u) such that the monomial m has non-zero coefficient in the polynomial $p(s,w) \times \mathsf{label}(w,u)$. The critical edge corresponds to the last variable of the monomial to be "collected" en route to v from s. This completes the proof. □

For nodes w, u, v in B where (w,u) is an edge, define a 0–1 valued indicator function that specifies whether or not (w,u) is critical to v. That is,

$$\mathsf{critical}\big(\langle w,u\rangle, v\big) = \begin{cases} 1 & \text{if } (w,u) \text{ is critical for } v, \\ 0 & \text{otherwise.} \end{cases}$$

Using this and Lemma 11, we can show:

Lemma 12. *In an occur-once OBP B with start node s, for any node v in B,*

$$\#p(s,v) = \sum_{e=(w,u)} \mathsf{critical}\big(\langle w,u\rangle, v\big) \cdot \big(\#p(s,w) + \mathsf{NZ}(s,w)\big) \cdot \mathsf{NZ}(u,v).$$

Proof. Consider the expression $p(s,w) \times \mathsf{label}(w,u)$, where (w,u) is an edge critical to v. Then $\mathsf{label}(w,u)$ is in X, and multiplies every monomial in $p(s,w)$. Hence every monomial of $p(s,w)$ contributes a monomial to $p(s,w) \times \mathsf{label}(w,u)$. Additionally, if $c(s,w) \neq 0$, then $c(s,w) \times \mathsf{label}(w,u)$ too contributes a monomial. Hence

$$\#\big[p(s,w) \times \mathsf{label}(w,u)\big] = \#p(s,w) + \mathsf{NZ}(s,w).$$

Using this observation along with Lemma 11 completes the proof. □

If w is not in a layer to the left of v, then (w,u) cannot be critical to v, and so $\#p(s,w)$ is not required while computing $\#p(s,v)$. Hence we can sequentially evaluate $\#p(s,v)$ for all nodes v in layers going left to right, provided we have all the values $\mathsf{NZ}(s,w)$ and $\mathsf{critical}(\langle w,u\rangle, v)$.

Lemma 13. *Define languages L_1, L_2 as follows:*

$$L_1 = \big\{\langle B,u,v\rangle \bigm| B \text{ is an OBP, } u,v \text{ are nodes in } B\text{, and } \mathsf{NZ}(u,v) = 0\big\},$$
$$L_2 = \big\{\langle B,u,v,w\rangle \bigm| B \text{ is an OBP, } u,v,w \text{ are nodes in } B\text{, and } \mathsf{critical}\big(\langle w,u\rangle, v\big) = 1\big\}.$$

Then L_1 and L_2 are both in $\mathsf{C_{=}L}$.

Proof. Delete from B all edges with labels from X to get a variable-free BP B'. Then $p_{B'}(u,v) = c_B(u,v)$. Checking whether $p_{B'}(u,v) = 0$ is the canonical complete problem for $\mathsf{C_{=}L}$. Hence L_1 is in $\mathsf{C_{=}L}$. To check membership in L_2, we need to check that $\mathsf{label}(w,u) \in X$ and that v is reachable from u in B'. This can be done in NLOG, which is contained in $\mathsf{C_{=}L}$. □

From Lemma 12, the comment following it, and Lemma 13, we obtain a polynomial time algorithm to count the monomials in p_B.

Theorem 14. *Given an occur-once branching program B, the number of monomials in p_B can be computed in* P.

With a little bit of care, we can obtain the following stronger result:

Theorem 15. *Given an occur-once branching program B, the number of monomials in p_B can be computed in the* GapL *hierarchy and hence in* NC^2.

Proof. Starting from B, we construct another BP B' as follows: B' has a node v' for each node v of B. For each triple w, u, v where (w,u) is an edge in B, we check via oracles for L_1 and L_2 whether (w,u) is critical to v and whether $\mathsf{NZ}(u,v) = 1$. If both checks pass, we add an edge from w' to v'. We also check whether $\mathsf{NZ}(s,w) = 1$, and if so, we add an edge from s' to v'. (We do this for every w, u, so we may end up with multiple parallel edges from s' to v'. To avoid this, we can subdivide each such edge added.) B' thus implements the right-hand-side expression in Lemma 12. It follows that $p_{B'}(s',v')$ equals $\#p_B(s,v)$. Note that B' can be constructed in logspace with oracle access to $\mathsf{C_{=}L}$. Also, since B' is variable-free, it can be evaluated in GapL. Hence $\#p_B$ can be computed in the GapL hierarchy. □

As in Corollary 10, using Theorem 15 and Lemma 13, we have:

Corollary 16. *In the non-blackbox setting,* PIT *on OBPs is in the* GapL *hierarchy and hence in* NC^2.

5. Zero-test on a monomial coefficient (ZMC)

From [11], ZMC is known to be in the second level of CH and hard for the class $\mathsf{C_{=}P}$. For the very restricted case of depth-3 read-3 formulas, ZMC is known to be NP-hard. (In Proposition 13 of [20], hardness is shown for depth-3 degree-3 formulas. It can be verified that the hard formulas there are also read-thrice.) For the case of multilinear BPs (i.e. MBPs), we show that ZMC exactly characterises the complexity class $\mathsf{C_{=}L}$.

Theorem 17. *Given a BP B computing a multilinear polynomial p_B, and given a multilinear monomial m, the coefficient of m in p_B can be computed in* GapL. *Hence* ZMC *for multilinear BPs is complete for* $\mathsf{C_{=}L}$.

Proof. We first show that ZMC, even for OBPs, is hard for $\mathsf{C_{=}L}$. A complete problem for $\mathsf{C_{=}L}$ is: does a BP B with labels from $\{-1,0,1\}$ evaluate to 0? Add a node t' as the new target node, and add edge $t \to t'$ labeled x to get B'. Then B' is an OBP, and $(B', x) \in \mathsf{ZMC}$ if and only if B evaluates to 0.

Now we show the upper bound. We show that given a branching program B computing a multilinear polynomial p_B, and given a multilinear monomial m, the coefficient of m in p_B can be computed in GapL. This will imply that the zero-test is in $\mathsf{C_{=}L}$.

Let $S \subseteq [n]$ be such that $m = m_S$. Let $p_B = \sum_{T\subseteq[n]} \mathsf{coeff}(p_B, m_T)m_T$. We are interested in $\mathsf{coeff}(p_B, m_S)$. The idea is to construct a branching program B' computing a univariate polynomial, and a monomial m', such that $m \in p_B$ if and only if $m' \in p_{B'}$. We obtain B' by relabelling the edges of B as follows:

Label in B	Constant c	x_i for $i \in S$	x_i for $i \notin S$
Label in B'	Constant c	y	0

B' now computes a univariate polynomial $p_{B'}$ in y.

Observe that the coefficient c_S of m in p_B is equal to the coefficient of $y^{|S|}$ in $p_{B'}$. To see this, note that

$$p_B = \sum_{T\subseteq[n]} \mathsf{coeff}(p_B, m_T)m_T = \sum_{T\subseteq S} \mathsf{coeff}(p_B, m_T)m_T + \sum_{T\nsubseteq S} \mathsf{coeff}(p_B, m_T)m_T.$$

The substitution described above sends the second sum to zero in B'. Hence,

$$p_{B'}(y) = \sum_{T\subseteq S} \mathsf{coeff}(p_B, m_T)y^{|T|} = \sum_{j=0}^{|S|}\Bigg(\sum_{\substack{T\subseteq S\\|T|=j}} \mathsf{coeff}(p_B, m_T)\Bigg)y^j.$$

The only monomial in p_B that generates $y^{|S|}$ in $p_{B'}$ is $\prod_{i\in S} x_i = m_S$; hence $\mathsf{coeff}(p_{B'}, y^{|S|}) = \mathsf{coeff}(p_B, m_S)$.

(This argument only requires that p_B be multilinear; we do not need B to be occur-once or even read-once.)

Thus the problem now reduces to computing the coefficient of $y^{|S|}$ in B', which is a branching program over just one input variable. A standard construction allows us to explicitly construct all coefficients of $p_{B'}(y)$ in another branching program B''. For completeness, we describe the construction of B''. For each node v in B', B'' has $|S|+1$ nodes $v_0, \ldots, v_{|S|}$, with the intention that v_i should compute the coefficient of y^i in the polynomial $p_{B'}(s, v)$. The start node of B'' is the node s_0, and the final node is $t_{|S|}$. If edge (u, v) has label y in B', we include the edges (u_i, v_{i+1}) with label 1, for $0 \leqslant i < |S|$, in B''. If edge (u, v) has label $\ell \neq y$ in B', we include the edges (u_i, v_i) with label ℓ, for $0 \leqslant i \leqslant |S|$, in B''. By induction on the structure of B', we see that the value computed by B'' at $t_{|S|}$ is the coefficient of $y^{|S|}$ in $p_{B'}(s, v)$.

The above transformation from B' to B'' can be done in DLOG. Since B'' is variable-free, it can be evaluated in GapL. Composing these procedures, we obtain a GapL procedure for computing the coefficient of m in p_B. $\square$

The upper bound above, for ZMC on MBPs, also applies to ROFs, since ROFs can be converted to OBPs by a standard construction. However, with a careful top-down algorithm, we can give a stronger upper bound of DLOG for ZMC on ROFs.

Theorem 18. *Given a read-once formula F computing a polynomial p_F, and given a multilinear monomial m, the coefficient of m in p_F can be computed in* DLOG. *Hence* ZMC *for ROFs is in* DLOG.

Proof. Let $\alpha(g, T)$ denote the coefficient of monomial m_T in p_g. (That is, $\alpha(g, T) = \mathsf{coeff}(p_g, m_T)$.) Let r be the output gate. Let $S \subseteq [n]$ be such that $m = m_S$. The goal is to compute $\alpha(r, S)$. First, we observe some properties of α:

1. For any gate g and any $T \subseteq [n]$, if $T \nsubseteq \mathsf{var}_g$, then $\alpha(g, T) = 0$.
2. For a leaf g labelled x_i, $\alpha(g, T) = 1$ if $T = \{i\}$, 0 otherwise.
3. For a leaf g labelled by a constant c, $\alpha(g, T) = c$ if $T = \emptyset$, 0 otherwise.

4. For an addition gate that computes $g+h$, $\alpha(g+h,T)=\alpha(g,T)+\alpha(h,T)$. And since F is an ROF, at least one of $\alpha(g,T)$, $\alpha(h,T)$ is zero.
5. For a product gate that computes $g\times h$,

$$\alpha(g\times h,T)=\alpha(g,T\cap \mathsf{var}_g)\cdot\alpha(h,T\cap\mathsf{var}_h)\cdot[T\subseteq \mathsf{var}_g\cup\mathsf{var}_h].$$

This is because $\alpha(g\times h,T)=\sum_{Z\subseteq T}\alpha(g,Z)\alpha(h,T\setminus Z)$. But if either $Z\not\subseteq \mathsf{var}_g$ or $T\setminus Z\not\subseteq\mathsf{var}_h$, then $\alpha(g,Z)=0$ or $\alpha(h,T\setminus Z)=0$. Further, F is a read once formula, so $\mathsf{var}_g\cap\mathsf{var}_h=\emptyset$, and var_g and var_h partition $\mathsf{var}_{(g\times h)}$. Hence T must also be similarly partitioned.

Now we construct a formula F' whose evaluation gives us $\alpha(r,S)$. F' will recursively compute $\alpha(g,S\cap\mathsf{var}_g)$ for each gate g. If g is a leaf, we just use properties (2, 3) to compute $\alpha(g,S\cap\mathsf{var}_g)$. We show how to compute $\alpha(f,S\cap\mathsf{var}_f)$ for an internal gate f with children g and h knowing the values for $\alpha(g,S\cap\mathsf{var}_g)$ and $\alpha(h,S\cap\mathsf{var}_h)$:

- Case $f=g+h$:

$$\begin{aligned}\alpha(f,S\cap\mathsf{var}_f)&=\alpha(g,S\cap\mathsf{var}_f)+\alpha(h,S\cap\mathsf{var}_f)\quad\text{from property (4)}\\&=\alpha(g,S\cap\mathsf{var}_g)[S\cap\mathsf{var}_g=S\cap\mathsf{var}_f]\\&\quad+\alpha(h,S\cap\mathsf{var}_h)[S\cap\mathsf{var}_h=S\cap\mathsf{var}_f]\quad\text{from property (1).}\end{aligned}$$

- Case $f=g\times h$:

$$\alpha(f,S\cap\mathsf{var}_f)=\alpha(g,S\cap\mathsf{var}_g)\cdot\alpha(h,S\cap\mathsf{var}_h)\quad\text{from properties (1, 5).}$$

This gives us the formula F' that computes $\alpha(r,S)$ at the topmost gate. By Proposition 1, F' can be evaluated in GapNC^1. Constructing F' from F requires a local transformation at $+$ gates and computation of the predicates $[S\cap\mathsf{var}_f=S\cap\mathsf{var}_g]$. By Proposition 3, these predicates can be computed in DLOG. □

For ROFs, the lower bound proof in Theorem 17 can be modified to show that ZMC on ROFs is hard for $\mathsf{C_=NC}^1$. It is natural to ask whether there is a matching upper bound. In our construction above, we need to compute predicates of the form $[x\in\mathsf{var}_g]$. If these can be computed in NC^1 for ROFs, then the monomial coefficients can be computed in GapNC^1 and hence the upper bound of ZMC can be improved to $\mathsf{C_=NC}^1$. However, this depends on the specific encoding in which the formula is presented. In the standard pointer representation, the problem models reachability in out-degree-1 directed acyclic graphs, and hence is as hard as DLOG. Perhaps, under some other encoding, an upper bound of NC^1 is possible. To see why this may be plausible, consider the problem of testing whether two trees are isomorphic. (And note that the undirected graph underlying a formula is a tree.) For trees encoded as pointer lists, isomorphism testing is DLOG-complete, whereas for trees encoded as strings, the same problem of isomorphism testing is complete for NC^1 [15].

6. Checking existence of monomial extensions

We now address the problem ExistExtMon. Given a monomial m, one wants to check if the polynomial computed by the input arithmetic circuit has a monomial M that extends m (that is, with $m|M$). This problem is seemingly harder than ZMC, and hence the bound of Theorem 17 does not directly apply to ExistExtMon. We show that ExistExtMon for OBPs is in the GapL hierarchy.

Theorem 19. *The following problem lies in the* GapL *hierarchy: Given an occur-once branching program B and a multilinear monomial m, check whether p_B contains any monomial M such that $m|M$.*

Proof. Let $S\subseteq[n]$ be such that $m=m_S$. If $S=\emptyset$, then this amounts to checking if $p_B\not\equiv 0$. By Corollary 16, this is in the GapL hierarchy. So now assume that $S\neq\emptyset$. We call an edge that is labelled by a variable from S a "bridge".

The algorithm is as follows:

```
if $\exists i\in S$ such that $x_i$ does not appear in $B$ at all then
  Output NO and halt.
else if $\exists$ layer $l$ with more than one bridge to layer $l+1$ then
  Output NO and halt.
else
  For each layer $l$ that has a bridge $e$ to layer $l+1$ in $B$, remove all edges except $e$. Call the branching program thus obtained $B'$.
end if
Output $\overline{\mathsf{PIT}(p_{B'})}$ and halt.
```

We now show that m_S has an extended monomial in p_B if and only if the above algorithm outputs YES. If any of the variables of m_S do not appear at all in B, then clearly an extension to m_S cannot exist. So the algorithm rejects correctly. If there is a layer with more than one bridge to the next layer, then any path can go through at most one of these bridges. Since B is occur-once, every path would compute a monomial with at least one variable from m_S missing. So the algorithm correctly rejects. We are only interested in monomial extensions of m_S. So paths that do not go through all the bridges can be ignored. Hence we can safely delete all non-bridge edges in layers which have a bridge to the next layer. Thus $p_{B'}$ is a polynomial where each monomial is an extension of m_S.

By Corollary 16, the above algorithm is in the GapL hierarchy. □

With a little bit of care, the above bound can be brought down to DLOG for the case of ROFs.

Theorem 20. *The following problem is in* DLOG: *Given a read-once formula F computing a polynomial p_F, and given a multilinear monomial m, check whether p_F contains any monomial M such that $m|M$.*

Proof. Let $S \subseteq [n]$ be such that $m = m_S$. If $S = \emptyset$, then this amounts to checking if $p_F \not\equiv 0$. By Corollary 10, this is in DLOG. So now assume that $S \neq \emptyset$. Similar to the case of branching programs, we transform F to a new formula F' as follows:

```
if  $\exists x_i \in S$ such that $x_i$ does not appear in $F$ at all then
  Output NO and halt.
else if  $\exists x_i, x_j \in S$, $i \neq j$, with LCA$(x_i, x_j)$ in $F$ labeled $+$ then
  Output NO and halt.
else
  For every $x_i \in S$, and every $+$ gate $g$ on the unique leaf-to-root path $\gamma$ from $x_i$, replace the input of $g$ not on the path $\gamma$
  by 0.
  Let $F'$ be the resulting formula.
end if
Output $\overline{\mathsf{PIT}(F')}$.
```

We show correctness of the above algorithm. Since F is read-once, if any of the two variables in S have a $+$ gate as their least common ancestor, then m cannot appear as a monomial in F. If the algorithm reaches the **else** statement, then all sub-formulas that are additively related to some variable x_i in S are removed. This implies that every monomial produced by F' has m as a factor. Also, any monomial m' of p_F with $m|m'$ has the same coefficient in $p_{F'}$ as in p_F. Thus, the resulting formula F' computes a polynomial that contains exactly all monomials m' of p_F such that $m|m'$. This proves the correctness.

For the complexity bound, we note that the transformation from F to F' can be done in DLOG (using Proposition 3). Then by Corollary 10, the overall algorithm can be implemented in DLOG. □

7. Conclusion

Our results show that for the restricted case of read-2 and read-3 formulas, PIT and MLIN can indeed be decided efficiently in the non-black box setting. We feel the techniques we use for read-2 should be helpful is attacking PIT for formulas that read variables more often. We leave open the problem of deciding PIT for formulas that read variables $O(1)$, but 4 or more, times.

Although one would expect the complexity of MonCount to reduce drastically for the case of severely restricted circuits, it remains #P hard for even read-twice formulas. We note that the complexity of ZMC, and ExistExtMon does reduces drastically for the case of restricted circuits as expected. Ideally, we would like these problems to characterise complexity classes within P; we have partially succeeded in this. We leave open the question of extending these bounds for formulas and branching programs that are constant-read. It appears that this will require non-trivial modifications of our techniques.

References

[1] M. Agrawal, C. Saha, R. Saptharishi, N. Saxena, Jacobian hits circuits: hitting-sets, lower bounds for depth-d occur-k formulas & depth-3 transcendence degree-k circuits, in: STOC, 2012, pp. 599–614.

[2] E. Allender, Arithmetic circuits and counting complexity classes, in: J. Krajicek (Ed.), Complexity of Computations and Proofs, in: Quad. Mat., vol. 13, Seconda Universita di Napoli, 2004, pp. 33–72.

[3] E. Allender, R. Beals, M. Ogihara, The complexity of matrix rank and feasible systems of linear equations, Comput. Complex. 8 (2) (1999) 99–126.

[4] E. Allender, M. Ogihara, Relationships among PL, #L, and the determinant, ITA 30 (1) (1996) 1–21.

[5] M. Anderson, D. van Melkebeek, I. Volkovich, Derandomizing polynomial identity testing for multilinear constant-read formulae, in: IEEE Conference on Computational Complexity, 2011, pp. 273–282.

[6] S. Arora, B. Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009.

[7] V. Arvind, T.C. Vijayaraghavan, The orbit problem is in the GapL hierarchy, J. Comb. Optim. 21 (1) (2011) 124–137.

[8] S. Buss, The Boolean formula value problem is in ALOGTIME, in: STOC, 1987, pp. 123–131.

[9] S. Buss, S. Cook, A. Gupta, V. Ramachandran, An optimal parallel algorithm for formula evaluation, SIAM J. Comput. 21 (4) (1992) 755–780.

[10] S.A. Cook, P. McKenzie, Problems complete for deterministic logarithmic space, J. Algorithms 8 (3) (1987) 385–394.
[11] H. Fournier, G. Malod, S. Mengel, Monomials in arithmetic circuits: complete problems in the counting hierarchy, in: STACS, 2012, pp. 362–373.
[12] A. Gupta, P. Kamath, N. Kayal, R. Saptharishi, Arithmetic circuits: A chasm at depth three, in: FOCS, 2013, pp. 578–587, http://doi.ieeecomputersociety.org/10.1109/FOCS.2013.68.
[13] W. Hesse, E. Allender, D.A.M. Barrington, Uniform constant-depth threshold circuits for division and iterated multiplication, J. Comput. Syst. Sci. 65 (4) (2002) 695–716.
[14] M.J. Jansen, Y. Qiao, J.M.N. Sarma, Deterministic black-box identity testing π-ordered algebraic branching programs, in: FSTTCS, 2010, pp. 296–307.
[15] B. Jenner, P. McKenzie, J. Torán, A note on the hardness of tree isomorphism, in: IEEE Conference on Computational Complexity, 1998, pp. 101–105.
[16] V. Kabanets, R. Impagliazzo, Derandomizing polynomial identity tests means proving circuit lower bounds, Comput. Complex. 13 (1–2) (2004) 1–46.
[17] P. Koiran, S. Perifel, The complexity of two problems on arithmetic circuits, Theor. Comput. Sci. 389 (1–2) (2007) 172–181.
[18] M. Mahajan, B.R. Rao, K. Sreenivasaiah, Identity testing, multilinearity testing, and monomials in read-once/twice formulas and branching programs, in: MFCS, Springer, Berlin, Heidelberg, 2012, pp. 655–667.
[19] A. Shpilka, I. Volkovich, Read-once polynomial identity testing, in: STOC, 2008, pp. 507–516.
[20] Y. Strozecki, On enumerating monomials and other combinatorial structures by polynomial interpolation, Theory Comput. Syst. 53 (4) (2013) 532–568.
[21] L.G. Valiant, The complexity of computing the permanent, Theor. Comput. Sci. 8 (1979) 189–201.