# Identity Testing, multilinearity testing, and monomials in Read-Once/Twice Formulas and Branching Programs*

Meena Mahajan[1], B V Raghavendra Rao[2], and Karteek Sreenivasaiah[1]

[1] Institute of Mathematical Sciences, Chennai, India. {meena,karteek}@imsc.res.in
[2] Saarland University, Saarbrücken, Germany. bvrr@cs.uni-sb.de

**Abstract.** We study the problem of testing if the polynomial computed by an arithmetic circuit is identically zero (ACIT). We give a deterministic polynomial time algorithm for this problem when the inputs are read-twice formulas. This algorithm also computes the MLIN predicate, testing if the input circuit computes a multilinear polynomial.
We further study two 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, and 2) MonCount: compute the number of monomials in $C$. These problems were introduced by Fournier, Malod and Mengel [STACS 2012], and shown to characterize various levels of the counting hierarchy (CH).
We address the above problems on read-restricted arithmetic circuits and branching programs. We prove several complexity characterizations for the above problems on these restricted classes of arithmetic circuits.

## 1 Introduction

A fundamental question concerning a given arithmetic circuit is: does the circuit compute the identically zero polynomial? This is the well-known problem Arithmetic Circuit Identity Testing ACIT, that has spurred an enormous amount of research in the last two decades. A complete derandomization of black-box ACIT even for depth four arithmetic circuits implies circuit lower bounds [13, 2].

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 crcuits, 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 [2], improving this to arbitrary depth-4 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 ACIT on such formulas are trivial. Going beyond

* partially supported by Indo-German Max Planck Center (IMPECS)

these for $k > 1$, one breakthrough in [15] shows how to test $k$-sums of ROFs: for each $k \in O(1)$, ACIT 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.

To use the algorithm from [5] for a Read-$k$ formula, we first need to check whether it is multilinear. The multilinearity testing predicate MLIN is as hard as ACIT 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 result is a multilinearity test for read-twice formulas R2Fs. Such a test, in conjunction with [5], would give an ACIT test for R2Fs too. But our test is actually intertwined with an ACIT test for subformulas. We give a deterministic polynomial-time algorithm that simultaneously decides whether an R2F is multilinear and whether it is identically zero. It performs identity tests on partial derivatives. It also uses the sum-of-$k$-ROFs test from [15] 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 [15].

ACIT tests check whether the polynomial computed by the crcuit has at least one monomial. Natural generalizations/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 [14]. More recently, Fournier, Malod and Mengel [11] showed that ZMC and MonCount characterize certain levels of the counting hierarchy (CH, the hierarchy based on the complexity classes PP and $\mathsf{C_{=}P}$). 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 ACIT respectively), and that multilinearity checking itself is equivalent to ACIT. All these results from [11] are in the non-black-box model, where the circuit is given explicitly in the input.

Since ACIT 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 2) that MonCount for ROFs is in the $\mathsf{GapNC^1}$-hierarchy, i.e. the $\mathsf{AC^0}$-closure of $\mathsf{GapNC^1}$, where $\mathsf{GapNC^1}$ is the class of Boolean problems that can be computed by arithmetic formulas over the integers with constants 0, 1, -1. The $\mathsf{GapNC^1}$-hierarchy is an intriguing class that lies between $\mathsf{NC^1}$ and DLOG and has been studied extensively in the last

two decades; see for instance [3]. We also show that ZMC for ROFs is in logspace (Theorem 6). It is straightforward to see that ZMC for ROFs is hard for $\mathsf{C}_=\mathsf{NC}^1$, so this is almost tight. (The "gap" between Boolean $\mathsf{NC}^1$, $\mathsf{C}_=\mathsf{NC}^1$, $\mathsf{GapNC}^1$ and DLOG is very small.)

Another natural, well-studied restriction is when the circuit is an algebraic branching program BP with edges labeled 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 $\mathsf{AC}^0$ closure of GapL) is known to be contained in $\log n$ depth threshold circuits $\mathrm{TC}^1$ and hence in $\log^2 n$ depth Boolean circuits $\mathrm{NC}^2$. 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, [12]. We show that MonCount for OBPs is in the GapL hierarchy (Theorem 4), while ZMC for OBPs and even MBPs is complete for the complexity class $\mathsf{C}_=\mathsf{L}$ (Theorem 5). (As a comparison, a well-known complete problem for $\mathsf{C}_=\mathsf{L}$ is testing singularity of an integer matrix [4].)

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 7).

## 2 Preliminaries

**Circuits, formulas, branching programs, polynomials.** An *arithmetic circuit* $C$ over a ring $R$ is a directed acyclic graph where every node has in-degree zero or two. The nodes with in-degree zero are called leaves. Internal nodes are labeled $+$ or $\times$ and leaves are labeled from $X \cup R$, where $X = \{x_1, \ldots, x_n\}$, a set of variables. There is a node of out-degree zero, called the root node or the output gate. Unless otherwise stated, $R$ is 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 circut 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 $R[x_1, \ldots, x_n]$ 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 $\mathsf{var}_g \stackrel{\Delta}{=} \{x_i \mid \text{some descendant of } g \text{ is a leaf labelled } x_i\}$.

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 a directed acyclic graph $B$ with edges labeled from $\{x_1, \ldots, x_n\} \cup R$, and with two designated nodes, $s$ with zero in-degree, and $t$ with zero out-degree. For any directed path $\rho$ in $B$, let $\mathsf{weight}(\rho) = \prod_{e:\text{ an edge in }\rho} \mathsf{label}(e)$.

Any pair of nodes $u, v$ in $B$ computes a polynomial in $R[x_1 \ldots x_n]$ defined by: $p_B(u,v) = \sum_{\rho:\ \rho \text{ is a } u \rightsquigarrow v \text{ path in } B} \mathsf{weight}(\rho)$. The ABP $B$ computes the polynomial $p_B \stackrel{\triangle}{=} p_B(s,t)$. We drop the subscript $B$ 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 generalize 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 pair of nodes is multilinear.

**Complexity Classes.** For all the standard complexity classes, the reader is referred to [6]. We define some of the non-standard complexity classes that are used in the paper. Let $f = (f_n)_{n \geq 0}$ be a family of integer valued functions $f_n : \{0,1\}^n \to \mathbb{Z}$. $f$ is in the complexity class $\mathsf{GapL}$ exactly when there is some nondeterministic logspace machine $M$ such that for every $x$, $f(x)$ equals the number of accepting paths of $M$ on $x$ minus the 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 $\mathsf{GapL}$ hierarchy is built over bit access to $\mathsf{GapL}$ functions, with a deterministic logspace machine at the base, and is known to be contained in $\mathsf{NC^2}$. (See [4, 3] for more details.)

$\mathsf{GapNC^1}$ denotes the class of families of functions $(f_n)_{(n \geq 0)}$, $f_n : \{0,1\}^n \to \mathbb{Z}$, where $(f_n)_{n>0}$ can be computed by a uniform polynomial size log depth arithmetic circuit family. This equals the class of functions computed by uniform polynomial-sized arithmetic formulas ([8]). $\mathsf{C_=NC^1}$ is the class of languages $L$ such that for some $\mathsf{GapNC^1}$ function family $(f_n)_{n \geq 0}$, and for every $x$, $x \in L \Leftrightarrow f_{|x|}(x) = 0$. The $\mathsf{GapNC^1}$ hierarchy comprises of languages accepted by polynomial-size constant depth unbounded fanin circuits ($\mathsf{AC^0}$) with oracle access to bits of $\mathsf{GapNC^1}$ functions, and is known to be contained in $\mathsf{DLOG}$. (See [9, 10] for more details.)

**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 $x_1 x_3$ is represented as $(1,0,1)$. For a degree sequence $m = (d_1, \ldots, d_n)$ 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 $S$ is true.

We now describe the computational problems considered in this paper.

**ACIT:** Given an arithmetic circuit $C$ over $\mathbb{Z}$, test if the polynomial computed by $C$ is identically zero.

MonCount: Given an arithmetic circuit $C$ over $\mathbb{Z}$, compute the number of monomials in the polynomial computed by $C$.

MLIN: Given an arithmetic formula $F$ over $\mathbb{Z}$, test if the polynomial $p_F$ is multilinear.

ZMC: Given an arithmetic circuit $C$ over $\mathbb{Z}$, and a monomial $m$, test if $\mathsf{coeff}(p_C, m)$ is zero 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: for a single variable $x_i$, $\mathsf{ExistExtMon}(C, x_i)$ just tests if the partial derivative of $p_C$ with respect to $x_i$ is identically zero.

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

**Proposition 1 ( [7, 8]).** *Evaluating an arithmetic formula where the leaves are labelled* $\{-1, 0, 1\}$ *is in* DLOG *(even* $\mathsf{GapNC}^1$*).*

**Proposition 2 ( [15]).** *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 $n^{O(k)}$.*

**Proposition 3 (folklore).** *The following problems are in* DLOG*:*
*1) Given a formula $F$, a gate $g \in F$, and a variable $x$, check 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 Read-twice Formulas: multilinearity and identity tests

In this section we consider the problem of testing multilinearity (MLIN) and testing identically zero (ACIT) on read-twice 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 $x_i$ is zero for every variable $x_i$. We use the inductive structure of a read-twice formula to test first order partial derivatives for zero, using the knowledge of MLIN and ACIT at gates at the lower levels, and to combine this information to compute MLIN and ACIT at the root.

**Theorem 1.** *For read-twice formulas, the problems* ACIT*,* MLIN*, and* $\mathsf{ExistExtMon}(\phi, x)$ *(where $\phi$ is the input formula and $x$ is a single variable in it) are in P.*

*Proof.* Let $\phi$ be the given read-twice formula on variables $x_1, \ldots, x_n$, with $S$ internal nodes. Without loss of generality, assume that $\phi$ is 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 $\phi$. For each gate $g$ in $\phi$ and each variable $x \in X$, we iteratively compute the value of the constant term of $p_g$, denoted $\mathsf{const}(g)$, and the following set of 0-1 valued functions:

$$\mathsf{ACIT}(g) = 1 \Leftrightarrow p_g \equiv 0; \qquad \mathsf{MLIN}(g) = 1 \Leftrightarrow p_g \text{ is multilinear;}$$
$$\mathsf{ExistExtMon}(g, x) = 1 \Leftrightarrow p_g \text{ has a monomial } m \text{ that contains } x \ .$$

Recall that $\mathsf{ExistExtMon}(g, x) = 1$ exactly when the partial derivative of $p_g$ with respect to $x$ is not identically zero; in this case we say that $x$ survives in $g$. (Note: Since $\phi$ is a formula, the values $\mathsf{const}(g)$ can be represented with $\text{poly}(|\phi|)$ bits.)

The base case is when $\phi$ is a single variable or a constant. That is, $\phi$ consists of a single gate $g$, labelled $L \in X \cup \{0, +1, -1\}$. Then $\mathsf{ACIT}(g) = 1$ if and only if $L = 0$, $\mathsf{MLIN}(g) = 1$ always, and $\mathsf{ExistExtMon}(g, x) = 1$ if and only if $L = x$. Also, $\mathsf{const}(g)$ is $L$ if $L \notin X$, 0 otherwise.

Now assume that for every gate $u$ below the root gate of $\phi$, the above functions have been computed and stored as bits. Let $f$ be the root gate of $\phi$. 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. $\mathsf{const}(f) = \mathsf{const}(g) \times \mathsf{const}(h)$.
2. $\mathsf{ACIT}(f) = \mathsf{ACIT}(g) \vee \mathsf{ACIT}(h)$.
3. $\mathsf{MLIN}(f)$: 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
$$\mathsf{MLIN}(f) = \mathsf{ACIT}(f) \vee \Bigg[\mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h) \wedge \left(\bigwedge_{x \in X} [\neg \mathsf{ExistExtMon}(g, x) \vee \neg \mathsf{ExistExtMon}(h, x)]\right)\Bigg]$$
Note that the $\mathsf{ACIT}(f)$ term is necessary, since $f$ can be multilinear even if, for instance, $g$ is not, provided $h \equiv 0$.
4. $\mathsf{ExistExtMon}(f, x)$: $x$ 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
$$\mathsf{ExistExtMon}(f, x) = \neg \mathsf{ACIT}(f) \wedge [\mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x)]$$

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

1. $\mathsf{const}(f) = \mathsf{const}(g) + \mathsf{const}(h)$.
2. $\mathsf{MLIN}(f)$: 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 $\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h)$.
3. $\mathsf{ExistExtMon}(f, x)$: This is the non-trivial part; we defer it to a bit later.
4. $\mathsf{ACIT}(f)$: Once we compute the functions above, this is straightforward:
$$\mathsf{ACIT}(f) = [\mathsf{const}(f) = 0] \wedge \bigwedge_{x \in X} \neg \mathsf{ExistExtMon}(f, x)$$

We now describe how to compute $\mathsf{ExistExtMon}(f, x)$ 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

$$\mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x) = 0 \Longrightarrow \mathsf{ExistExtMon}(f, x) = 0$$
$$\mathsf{ExistExtMon}(g, x) \oplus \mathsf{ExistExtMon}(h, x) = 1 \Longrightarrow \mathsf{ExistExtMon}(f, x) = 1$$

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 $\alpha', \beta'$ 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 $\mathsf{var}_g$ and $\mathsf{var}_h$, and since $f$ is read-twice, we conclude that $x$ is read exactly once each in $g$ and in $h$. Hence $\alpha, \beta$ also do not involve $x$.

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

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 $\mathsf{ACIT}(F)$ 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 $\mathsf{ACIT}(F') = \mathsf{ACIT}(F)$.

Now we describe how to 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 $\rho$ 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 $\rho$; 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 $\phi$, and hence by the iterative construction we know the values of the functions $\mathsf{ACIT}$, $\mathsf{MLIN}$, $\mathsf{ExistExtMon}$ at gates in these sub-trees. In fact, we already know that $\mathsf{ACIT}(u_i) = 0$ 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 $\mathsf{ExistExtMon}(G, y) = \mathsf{ExistExtMon}(u_i, y) \vee \mathsf{ExistExtMon}(u_j, y)$. □

A direct attempt to generalise this to read-$k$ formulas would be to maintain $\mathsf{ExistExtMon}(f, x^i)$ for $1 \leq i \leq k$ at each gate. However, this does not work because in iteratively computing these values, we generate 2-sums of read-$k$ formulas, not $k$-sums of ROFs, and cannot use Proposition 2.

## 4 Counting Monomials

We now consider the MonCount problem. In an ROF, a monomial, once generated in a sub-formula, can be cancelled only by multiplication with a zero polynomial. We exploit this fact to obtain an efficient algorithm for MonCount on ROFs. We then show that even for read-twice formulas, the problem becomes very hard. Since we cannot generalise Theorem 2 to read-twice formulas, we consider generalising it beyond ROFs to read-once BPs. For OBPs, similar properties as for ROFs hold, and again we obtain an efficient algorithm for MonCount.

We start with ROFs:

**Theorem 2.** *Given a read-once formula $F$, $\mathsf{MonCount}(p_F)$ can be computed by an $AC^0$ circuit with oracle gates for $\mathsf{GapNC}^1$ functions, and hence in* DLOG.

The following lemma is used in proving Theorem 2:

**Lemma 1.** *The language $L$ defined below is in $\mathsf{C}_=\mathsf{NC}^1$:*

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

For any polynomial $p$, $p \equiv 0$ if and only if the constant term of $p$ is 0 and $\mathsf{MonCount}(\mathrm{p})$ is 0. Hence, from Theorem 2 and Lemma 1 we have the following:

**Corollary 1.** *In the non-blackbox setting,* ACIT *on ROFs is in the* GapNC *hierarchy and hence in* DLOG.

Our next result shows that extending Theorem 2 to Read-$k$ formulas for $k > 1$ is extremely unlikely. Even for formulas that are monotone (no negative constants) and read-twice, and furthermore, are decomposable as the sum of two read-once formulas, MonCount is at least as hard as #P.

**Theorem 3.** MonCount *is #P complete for the sum of two monotone read-once formulas.*

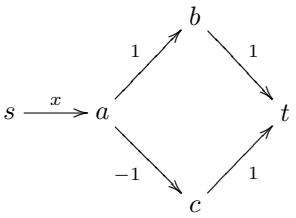
We now show how to count monomials in OBPs. The approach used in Theorem 2 does not directly generalize to OBPs, *i.e.,* knowing MonCount at immediately preceding nodes is not enough to compute MonCount at a given node in an OBP. However, since every variable occurs at most once in an OBP, every path 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 alongside, $\#p(s, b) = \#p(s, c) = 1$, but $\#p(s, t) = 0$. 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 $\rho$ from $u$ to $v$ with all edges labeled by $\{-1, 1\}$.

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

**Lemma 2.** *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.* 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 labeling $(w, u)$ 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 $m$ 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,

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

Using this and Lemma 2, we can show:

**Lemma 3.** *In an occur-once OBP $B$ with start node $s$, for any node $v$ in $B$,*

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

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 4.** *Define languages $L_1, L_2$ as follows:*

$$L_1 = \{\langle B, u, v\rangle \mid B \text{ is an OBP, } u, v \text{ are nodes in } B, \text{ and } \mathsf{NZ}(u, v) = 0.\}$$
$$L_2 = \left\{\langle B, u, v, w\rangle \mid \begin{array}{l} B \text{ is an OBP, } u, v, w \text{ are nodes in } B, \text{ and} \\ \mathsf{critical}(\langle w, u\rangle, v) = 1. \end{array}\right\}$$

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

From Lemma 3, the comment following it, and Lemma 4, we obtain a polynomial time algorithm to count the monomials in $p_B$. However, with a little bit of care, we can obtain the following stronger result:

**Theorem 4.** *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 3. 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 $\mathsf{GapL}$. Hence $\#p_B$ can be computed in the GapL hierarchy. □

As in Corollary 1, using Theorem 4 and Lemma 4, we have:

**Corollary 2.** *In the non-blackbox setting,* ACIT *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 case of multilinear BPs MBPs, we show that ZMC exactly characterizes the complexity class $\mathsf{C_=L}$.

**Theorem 5.** ZMC *for multilinear BPs is complete for $\mathsf{C_=L}$. More precisely,*

1. ZMC *for OBPs is hard for $\mathsf{C_=L}$.*
2. *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.

*Proof.* (Sketch) Hardness: 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 $(B', x) \in$ ZMC if and only if $B$ evaluates to 0.

Upper bound: The idea is to construct (by relabelling the edges of $B$) a branching program $B'$ computing a univariate polynomial, and a monomial $m'$, such that the coefficients of $m$ in $p_B$ and of $m'$ in $p_{B'}$ are the same. The coefficients of $p_{B'}$ can be computed in GapL, establishing the second statement. This will imply that the zero-test is in $\mathsf{C}_=\mathsf{L}$. □

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 6.** *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.

The lower bound proof in Theorem 5 can be modified to show that ZMC on ROFs is hard for $\mathsf{C}_=\mathsf{NC}^1$. 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 $\mathsf{NC}^1$ for ROFs, then the monomial coefficients can be computed in $\mathsf{GapNC}^1$, improving the upper bound of ZMC to $\mathsf{C}_=\mathsf{NC}^1$. 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.

## 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 5 does not directly apply to ExistExtMon. We show that ExistExtMon for OBPs is in the GapL hierarchy.

**Theorem 7.** *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 $m|M$.*

The above bound can be brought down to DLOG for the case of ROFs.

**Theorem 8.** *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 $m|M$.*

## 7 Conclusion

In this paper, we studied the complexity of certain natural problems on severely restricted circuits.

We have shown that ACIT and MLIN are easy on read-twice formulas. In a recent extension, we have shown that using [5] instead of [15] yields a simpler algorithm that works even for read-3 formulas. Extending this to Read-$k$ formulas for any constant $k > 3$ remains open.

We have shown that MonCount remains #P-hard for read-twice formulas.

We have shown that on read-once formulas and occur-once branching programs, the complexity of ZMC and ExistExtMon does reduce drastically. 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.

## References


1. M. Agrawal, C. Saha, R. Saptharishi, and N. Saxena. Jacobian hits circuits: Hitting-sets, lower bounds for depth-d occur-k formulas & depth-3 transcendence degree-k circuits. In *STOC*, 2012. To Appear.
2. M. Agrawal and V. Vinay. Arithmetic circuits: A chasm at depth four. In *FOCS*, pages 67–75, 2008.
3. E. Allender. Arithmetic circuits and counting complexity classes. In J. Krajicek, editor, *Complexity of Computations and Proofs*, Quaderni di Matematica Vol. 13, pages 33–72. Seconda Universita di Napoli, 2004.
4. E. Allender, R. Beals, and M. Ogihara. The complexity of matrix rank and feasible systems of linear equations. *Computational Complexity*, 8(2):99–126, 1999.
5. M. Anderson, D. van Melkebeek, and I. Volkovich. Derandomizing polynomial identity testing for multilinear constant-read formulae. In *CCC*, pages 273–282, 2011.
6. S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
7. S. Buss. The Boolean formula value problem is in ALOGTIME. In *STOC*, pages 123–131, 1987.
8. S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal of Computation*, 21(4):755–780, 1992.
9. H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic $NC^1$ computation. *Journal of Computer and System Sciences*, 57:200–212, 1998.
10. S. Datta, M. Mahajan, B. V. R. Rao, M. Thomas, and H. Vollmer. Counting classes and the fine structure between $NC^1$ and L. *Theoretical Computer Science*, 417:36–49, 2012.
11. H. Fournier, G. Malod, and S. Mengel. Monomials in arithmetic circuits: Complete problems in the counting hierarchy. In *STACS*, 2012.
12. M. J. Jansen, Y. Qiao, and J. M. N. Sarma. Deterministic black-box identity testing $\pi$-ordered algebraic branching programs. In *FSTTCS*, pages 296–307, 2010.
13. V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1-2):1–46, 2004.
14. P. Koiran and S. Perifel. The complexity of two problems on arithmetic circuits. *Theoretical Computer Science*, 389(1-2):172–181, 2007.
15. A. Shpilka and I. Volkovich. Read-once polynomial identity testing. In *STOC*, pages 507–516, 2008.