Electronic Colloquium on Computational Complexity, Report No. 41 (2024)

Launching Identity Testing into (Bounded) Space

Pranav Bisht * Nikhil Gupta † Prajakta Nimbhorkar ‡ Ilya Volkovich §

Abstract

In this work, we initiate the study of the space complexity of the Polynomial Identity Testing problem (PIT). First, we observe that the majority of the existing (time-)efficient "blackbox" PIT algorithms already give rise to space-efficient "whitebox" algorithms for the respective classes of arithmetic formulas via a space-efficient arithmetic formula evaluation procedure. Among other things, we observe that the results of Minahan-Volkovich (ACM Transactions on Computation Theory, 2018), Gurjar et. al. (Theory of Computing, 2017) and Agrawal et. al. (SIAM Journal of Computing, 2016) imply logspace PIT algorithms for read-once formulas, constant-width read-once oblivious branching programs, and bounded-transcendence degree depth-3 circuits, respectively.

However, since the best known blackbox PIT algorithms for the class of multilinear read-k formulas[1] are quasi-polynomial time, as shown in Anderson et. al. (Computational Complexity, 2015), our previous observation only yields a $O(\log^2 n)$-space whitebox PIT algorithm. Our main result, thus, is the first $O(\log n)$-space PIT algorithm for multilinear read-twice formulas[2]. We also extend this result to test if a given read-twice formula is equal to a given read-once formula.

Our technical contributions include the development of a space-efficient measure μ_ℓ which is "distilled" from the result of Anderson et. al. (Computational Complexity, 2015) and can be used to reduce PIT for a read-k formula to PIT for a sum of two read-$(k-1)$ formulas, in logarithmic space. In addition, we show how to combine a space-efficient blackbox PIT algorithm for read-$(k-1)$ formulas together with a space-efficient whitebox PIT algorithm for read-k formulas to test if a given read-k formula is equal to a given read-$(k-1)$ formula.

1 Introduction

The algorithmic problem of deciding whether a "given" multivariate polynomial over a field is identically zero holds an important place in theoretical computer science. This problem is popularly known as *Polynomial Identity Testing (PIT)*. There are two ways to give input to a PIT algorithm - either as an *arithmetic circuit* or as a *blackbox* (i.e. via oracle access). An arithmetic circuit F over a field $\mathbb{F}$ is a directed acyclic graph whose leaf nodes are labelled by variables and $\mathbb{F}$-constants, other nodes are labelled by $+$ and $\times$ operations, and the computation in F happens as follows: A leaf

*Department of Computer Science and Engineering, IIT(ISM) Dhanbad, India. Email: pranav@iitism.ac.in. The work was partially done when this author was at Boston College.

†Computer Science Department, Boston College, Chestnut Hill, MA. Email: nikhil.gupta.3@bc.edu

‡Chennai Mathematical Institute, India. Email: prajakta@cmi.ac.in

§Computer Science Department, Boston College, Chestnut Hill, MA. Email: ilya.volkovich@bc.edu

[1] A *read-k* formula is a formula in which every variable labels at most k leaves.

[2] In fact, our algorithm works for non-multilinear read-twice formulas as well.

ISSN 1433-8092

node computes its label and every other node applies the operation in its label to the polynomials computed by its children nodes. For the PIT problem, the input arithmetic circuit is provided either as a *blackbox* or a *whitebox*. In the blackbox PIT, we are only allowed to evaluate the n-variate input polynomial f on any assignment $(a_1, \ldots, a_n) \in \mathbb{F}^n$ of our choice, to get the corresponding evaluation $f(a_1, \ldots, a_n)$. One cannot see the exact arithmetic circuit in the blackbox setting. In contrast, in a whitebox PIT algorithm, one is given the whole arithmetic circuit as input and is therefore allowed to 'look inside' the given circuit. A blackbox PIT algorithm makes its decision by querying the given blackbox on a 'small set' of assignments in $\mathbb{F}^n$ and therefore, designing an efficient blackbox PIT algorithm for a class of polynomials can be harder than designing an efficient whitebox PIT algorithm for the same class. PIT is an important problem in complexity theory. This problem has numerous interesting applications in theoretical computer science like efficient primality testing algorithms [AB03, AKS04], algorithms for finding a perfect matching in graphs [Lov79, KUW86, MVV87, ST17], a proof of $\mathsf{IP} = \mathsf{PSPACE}$ [Sha90] etc.

A polynomial-time randomized algorithm having one-sided error is known for the PIT problem due to the *Schwartz-Zippel-Demillo-Lipton* lemma [DL78, Zip79, Sch80]. This places the PIT problem in the class BPP and, in fact, $\mathsf{co\text{-}RP}$. Derandomizing PIT to obtain a deterministic polynomial-time algorithm is a long-standing open problem. Apart from being a natural and interesting problem in its own right, derandomization of PIT also has deep connections with proving circuit lower bounds in complexity theory. The works of [KI04, HS80, Agr05] showed that a polynomial-time deterministic PIT algorithm implies super-polynomial lower bounds either for Boolean circuits or for arithmetic circuits. In the converse direction, [KI04] showed that a super-polynomial arithmetic circuit lower bound implies a sub-exponential time deterministic PIT algorithm. Derandomization of PIT is well-studied in the literature. We direct the interested reader to [SY10, Sax09] for getting a detailed exposition on PIT.

Although the mystery of derandomization of PIT for general arithmetic circuits remains unresolved, researchers have developed efficient deterministic whitebox and blackbox PIT algorithms in the *time-complexity world* for several natural and important sub-classes of arithmetic circuits. In particular, polynomial-time blackbox PIT algorithms are known for *depth-2 circuits* [BOT88, KS01, LV03], for *depth-3 circuits* with constant top fan-in [DS07, KS08, KS09, SS11, SS12, SS13], for *multilinear depth-4 circuits* [ASSS16, SV18, BSV23], for *arithmetic read-once formulas* (ROFs) [SV15, MV18], for *depth-3* circuits with *bounded transcendence degree* [ASSS16], for *constant-occur constant-depth arithmetic formulas* [ASSS16], etc. Polynomial-time whitebox PIT algorithms are known for the class of *multilinear bounded-read arithmetic formulas* [AvMV15], and for *read-once oblivious arithmetic branching programs* (ROABPs) [RS05].

Since the class of *logspace computable functions*, L, is contained in P, it is natural to ask whether a class of arithmetic circuits having either a deterministic whitebox or blackbox polynomial-time PIT algorithm also admits a deterministic logspace PIT. In this work, we initiate the study of space-efficient PIT. We only talk about space-efficient PIT in the whitebox setting, as it is not clear if space-efficient PIT in the blackbox setting even makes sense, as simply evaluating an arithmetic circuit or even a formula via blackbox access may not be doable in logarithmic space. In contrast, evaluating any circuit in the blackbox setting is always time-efficient, by convention.

1.1 Background on logspace computation

There is a rich and fine hierarchy of classes inside P. A lot of work in the literature has focused on determining the exact complexity of computational problems in this hierarchy. Logspace, also

denoted by L, is an important subclass of P. Several fundamental problems like reachability in undirected graphs [Rei08], isomorphism of trees [Lin92], and planar graphs [DLN+22] are shown to be complete for L. In the arithmetic world, several interesting problems are in L, for example, evaluation of arithmetic formulas [BCGR92], division, iterated multiplication, and powering [HAM02, BCH86] of integers etc. In fact, evaluation of arithmetic formulas lies in $\#\mathsf{NC}^1$, and division, iterated multiplication, and powering lie in TC^0, both of which are subclasses of L. Healy and Viola show iterated multiplication and exponentiation over finite fields of characteristic two is in TC^0 and hence in L [HV06].

Formulas vs. circuits: We would like to highlight the fact that performing several operations on formulas is complexity theoretically simpler than performing the same operations on arbitrary circuits. For instance, counting the number of descendants of a node in an arbitrary circuit is complete for NL [Imm88, Sze87], whereas the same problem is in L for formulas (e.g. [Lin92]). Evaluation of Boolean or arithmetic formulas is in logspace [BCGR92] whereas evaluation of arbitrary Boolean or arithmetic circuits is P-complete [Lad75, GHKL18]. In fact, even deciding whether a given arithmetic circuit evaluates to the zero element on a given assignment is already P-complete [GHKL18].

1.1.1 Facts

Following are some well-known definitions and facts about the class L:

1. We say that a function f is computable in *logspace* if, for each input x of length n, $f(x)$ is computable in space $O(\log n)$. Here, by space, we mean the space on the work-tape of the Turing machine that computes $f(x)$.
2. If a Turing machine uses space at most $s(n)$ on inputs of length n, and halts on all inputs, then it runs in time at most $2^{s(n)}$ and hence the length of its output is also bounded by $2^{s(n)}$.
3. If f and g are computable in logspace, then their composition (i.e. $f \circ g$) is also computable in logspace. In general, a composition of any c functions $f_1, f_2, \ldots, f_c$, that are computable in $O(\log n)$ space, is computable in $O(c \log n)$ space.

Fact 1.1. *A composition of a constant number of logspace computable functions is itself logspace computable.*

1.1.2 Left-heavy Formulas and Generic Traversal

A formula (or a tree) is *left-heavy*, if for each node, the subtree rooted at its left child must be at least as large as the subtree rooted at its right child. We measure the size of a subtree in terms of the number of leaves in it. The results of [Lin92, BCGR92, DLN+22] and others show how to transform any formula into a left-heavy form in logspace. Furthermore, we extend these ideas to a framework that allows us to compute a generic top-down recursive function in a space-efficient way.

Let Ψ be a function described via a triple of "update" functions $\Psi \triangleq (\Psi_{\text{leaf}}, \Psi_+, \Psi_\times)$ and let F be an arithmetic formula of size s. We can think of F as a straight-line program where each of the s instructions is of form:

$$g_a \leftarrow g_b \text{ op } g_c \quad \text{where } b < a \leq s \text{ or } g_b \in \mathbf{x} \cup \mathbb{F} \text{ (and similarly } g_c)$$

The computation of Ψ on F is defined recursively: if g is a leaf of F, then $\Psi(g) \triangleq \Psi_{\text{leaf}}(g)$. If g is a gate (i.e. operation) in F, then $\Psi(g)$ is defined by applying Ψ_+ or $\Psi_\times$ in the case of the $+$ and $\times$ gate, respectively (we assume that Ψ values have already been computed for both of g's children). Formally, suppose we have $g_a \leftarrow g_b$ op g_c for $b, c < a$. Then

$$\Psi(g_a) \triangleq \begin{cases} \Psi_+ \left(\Psi(g_b), \Psi(g_c)\right), & \text{if op} = + \\ \Psi_\times \left(\Psi(g_b), \Psi(g_c)\right), & \text{if op} = \times. \end{cases}$$

The procedure will perform a *post-order* traversal: left-right-center, remembering only the current and the previous positions[3]. After the left child has been processed, the procedure will push the intermediate value of Ψ (of the left child) into a stack and then pop it after processing the right child. While the naïve bound on the space complexity of this procedure is $O(t \cdot \text{depth}(F))$, where t denotes the maximum bit-complexity of Ψ, one could show a better bound of $O(t \cdot \log s)$ due to the fact that F is left-heavy, where s is the size of Ψ. We instantiate this framework in Sections 3 and 4 to obtain logspace procedures to compute partial derivatives, partial assignments, and our main technical contribution, the measure μ_ℓ.

As another example of an instance of the generic traversal, one can obtain a simpler, but less efficient evaluation procedure for arithmetic formulas using $O(\log q \cdot \log s)$ space, where s and q stand for the sizes of the formula and the underlying finite field, respectively. For more details see Section 3.

1.2 Preliminary Results, Motivations and Related Works

In this section, we list some preliminary results which, for most part are based on simple observations. Our main results can be seen as strengthening these observations.

Recall that the *Schwartz-Zippel-Demillo-Lipton* lemma [DL78, Zip79, Sch80] states that, with high probability, a non-zero polynomial $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ evaluates to a non-zero value on a point chosen uniformly at random. Hence, in order to obtain a randomized PIT algorithm it is sufficient to evaluate a given circuit/formula on a random input. As this can be carried out in polynomial time, we obtain a randomized *polynomial-time* PIT algorithm for all arithmetic circuits.

Given the aforementioned space-efficient formula evaluation procedure, one might attempt executing the same strategy on arithmetic formulas with the hope to (potentially) obtain a randomized *logspace* PIT algorithm. Alas, a naïve implementation could be problematic since in the standard randomized bounded space setting, the access to randomness is "read-once". That is, each random bit (or field element) is only given once and must be stored in the working space, if referenced again in the future. One way to address this problem would be to restrict the input formula itself to the class of read-once formulas (ROFs). The class of ROFs is extremely well-studied in the literature in both Boolean [AHK93, KLN+93, BHH95b, BHH95c] and arithmetic settings [HH91, BHH95a, BC98, BB98, SV14, SV15, Vol16, MS21, ST21, GST23]. Unfortunately, the aforementioned space-efficient formula evaluation may reference to the same bits (i.e. field elements) more than once even for the case of ROFs! However, one can observe that the traversal-based evaluation procedure (mentioned in the previous section) operates in a 'read-once fashion' when evaluating a read-once formula. That is, the procedure will require a single access to every bit (and hence field element) when evaluating a read-once formula. This will result in a randomized

[3]Note that remembering the entire traversal history would require linear memory thus rendering the procedure space-inefficient.

$O(\log q \cdot \log s)$-space algorithm, where s and q stand for the sizes of the formula and the underlying field, respectively. Since typically $q = \text{poly}(n)$, we obtain that PIT for ROFs is in the class BPL^2 i.e. in randomized $O(\log^2 n)$ space.

Besides arithmetic circuits and formulas, we also have the algebraic class of arithmetic branching programs (ABPs) and their sub-class ROABPs (read-once oblivious arithmetic branching programs). See Appendix D for the formal definitions of these models. One can observe that evaluating a width-w ROABP of degree d over a field $\mathbb{F}$ of size q can be carried out using $O(w \cdot \log q + \log d)$-space, furthermore, as in the case with read-once formulas, this evaluation can be carried out in a 'read-once' fashion. Hence we obtain that PIT for constant-width ROABPs is in BPL.

In [Nis93], Nisan introduced the class $\mathsf{BP} \cdot \mathsf{L}$ which is an analog of BPL in which the random bits can be read multiple times without having to store them in the working space. By extending the previous observations and using the space-efficient formula evaluation, we obtain that PIT for (all) arithmetic formulas and general (i.e. polynomial-depth) constant-width ABPs is in $\mathsf{BP} \cdot \mathsf{L}$.

Observation 1 (Summary of initial observations on the space complexity of PIT)**.**

- PIT *for the following classes is in* BPL*: constant-width ROABPs.*
- PIT *for the following classes is in* BPL^2*: ROFs.*
- PIT *for the following classes is in* $\mathsf{BP} \cdot \mathsf{L}$*: arithmetic formulas, constant-width ABPs.*

To complement the last bullet of the observation, we remark that if we insist that the random bits can only be accessed once (as in BPL) then the best known PIT algorithm even for formulas will be in $\mathsf{BPSPACE}(n \log s)$ [4] [5]. In addition, we remark that the computational powers of arithmetic formulas and constant-width ABPs are equivalent (see e.g. [BC92, BIZ18]). In fact, the proof of Lemma 2.10 actually goes through by converting an arithmetic formula into a width-3 ABP of polynomial depth. Finally, unlike BPL, the class $\mathsf{BP} \cdot \mathsf{L}$ is not known to be contained in P (see [Nis93] for more details). The only known relation is the following syntactic inclusion:

$$\mathsf{BPL} \subseteq \mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{BPP}$$

and that $\mathsf{L} = \mathsf{P} \implies \mathsf{BP} \cdot \mathsf{L} = \mathsf{BPP}$. And while BPP is *conjectured* to be equal to P, a result of [KI04] shows that any such proof will imply circuit lower bounds. Interestingly, PIT plays a pivotal role of a "BPP-complete" problem in the argument of [KI04]. Formally:

Lemma 1.2 ([KI04])**.** *Suppose* PIT $\in \mathsf{P}$*. Then either* $\mathsf{NEXP} \not\subseteq \mathrm{P/poly}$ *or the Permanent requires super-polynomial arithmetic circuits.*

As was observed in [KI04], this result can be extended and refined to the class of arithmetic formulas.

[4] Let $s : \mathbb{N} \to \mathbb{N}$. The class $\mathsf{BPSPACE}(s(n))$ is similar to BPL except that a probabilistic Turning machine in case of $\mathsf{BPSPACE}(s(n))$ uses $O(s(n))$ space.

[5] In order to invoke the algorithm based on the Schwartz-Zippel lemma, we will need a multiple access to n field elements, which will we need to store in the working memory. In fact, we can carry this out in $\mathsf{DSPACE}(n \log s)$. However, to the best of our knowledge, there is not PIT algorithm for formulas in $\mathsf{DSPACE}(o(n \log s))$ or even $\mathsf{BPSPACE}(o(n \log s))$.

Observation 1.3 ([KI04])**.** *Suppose* PIT *for arithmetic formulas is in* $\mathsf{NP} \cap \mathsf{coNP}$*. Then either* $\mathsf{NEXP} \not\subseteq \mathrm{P/poly}$ *or the Permanent requires super-polynomial arithmetic formulas.*

Combined with Observation 1, we can extend the result of [KI04] to $\mathsf{BP} \cdot \mathsf{L}$: any proof that $\mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{NP}$ will imply formula lower bounds.

Corollary 1.4. *Suppose* $\mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{NP}$*. Then either* $\mathsf{NEXP} \not\subseteq \mathrm{P/poly}$ *or the Permanent requires super-polynomial arithmetic formulas.*

On the other hand, while PIT for arithmetic **formulas** is in $\mathsf{BP} \cdot \mathsf{L}$, we observe that PIT for arithmetic **circuits** is P-hard, i.e., every problem in P reduces in logspace to PIT for circuits. More generally, we make the following observation on the relationships between $\mathsf{BPP}, \mathsf{BP} \cdot \mathsf{L}$ and PIT. These relationships are proved in Observations A.1, A.3, A.2, and A.4 of Section A of the appendix.

Observation 2 (Relationships between $\mathsf{BPP}, \mathsf{BP} \cdot \mathsf{L}$ and PIT)**.**

- PIT *for arithmetic circuits is* P*-hard.*
- $\mathsf{BP} \cdot \mathsf{L}$ *with oracle access to* PIT *for* ***circuits*** *is the same as* BPP*.*
- $\mathsf{BP} \cdot \mathsf{L}$ *with oracle access to* PIT *for* ***formulas*** *is the same as* $\mathsf{BP} \cdot \mathsf{L}$*.*
- $\mathsf{P} \subseteq \mathsf{BP} \cdot \mathsf{L}$ *if and only if* $\mathsf{BPP} = \mathsf{BP} \cdot \mathsf{L}$*.*

BPP and $\mathsf{BP} \cdot \mathsf{L}$ correspond to two important classes of probabilistic algorithms. As noted before, it follows from the definition of $\mathsf{BP} \cdot \mathsf{L}$ that $\mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{BPP}$. In Observation 2, we note some more relationships between these two classes, and the PIT for arithmetic circuits plays an important role in this. It is also not known if $P \subseteq \mathsf{BP} \cdot \mathsf{L}$. We observe that it happens if and only if $\mathsf{BPP} = \mathsf{BP} \cdot \mathsf{L}$. Thus, decoding the exact relationship between one of the $\mathsf{BPP}, \mathsf{BP} \cdot \mathsf{L}$ or $\mathsf{BP} \cdot \mathsf{L}, \mathsf{P}$ will also shed light on the other.

Our next preliminary result shows how to convert certain (time-)efficient "blackbox" PIT algorithms into space-efficient "whitebox" algorithms for the respective classes of arithmetic formulas via the space-efficient arithmetic formula evaluation procedure. The vast majority of the existing blackbox identity testing algorithms were given via a construction of the so-called "hitting-set generators". A (hitting-set) generator $\mathcal{G}$ for a circuit class $\mathcal{C}$ is a polynomial map $\mathcal{G} = (\mathcal{G}^1, \mathcal{G}^2, \ldots, \mathcal{G}^n) : \mathbb{F}^t \to \mathbb{F}^n$ such that for every non-zero $f \in \mathcal{C}$, $f(\mathcal{G}) \not\equiv 0$. Indeed, a generator acts as a variable reduction map that reduces the number of variables from n to t while preserving non-zeroness. Given such a generator, the actual identity testing is performed by evaluating $f(\mathcal{G})$ on a "small" set of points i.e. of size $(d\delta)^t$ where d and δ are the respective degrees of f and $\mathcal{G}$.

Incidentally, one can observe that in the wild, many of the constructed generators are actually *logspace-explicit*. That is, computable in logarithmic space (see Definition 2.11 for a formal treatment). Our first observation is that a logspace-explicit generator gives rise to a logspace PIT algorithm for the respective class of arithmetic formulas.

Theorem 3 (Informal)**.** *Let* $\mathcal{G} : \mathbb{F}^t \to \mathbb{F}^n$ *be a logspace-explicit generator for a class of arithmetic formulas* $\mathcal{C}$ *and let* $F \in \mathcal{C}$ *of size* s*. Then there exists an algorithm that given* F *decides if* $F \equiv 0$*, using* $O(t \log s)$ *space.*

Indeed, we obtain a logspace PIT algorithm when $t = O(1)$. The formal version of the result is given in Theorem 2.14. Subsequently, we observe that the results of [MV18, GKS17, ASSS16] imply logspace PIT algorithms for sums of ROFs, constant-width ROABPs, and bounded-transcendence degree depth-3 circuits, respectively.

Corollary 1.5 (Informal)**.** PIT *for the following classes is in* L*: sum of constantly-many ROFs, constant-width ROABPs, bounded-transcendence degree depth-3 circuits.*

The technical versions of these result can be found in Theorems 4.9, 6.4, 7.1, respectively. We note that these results already improve upon the initial observation made in Observation 1 and that similar observations can be made w.r.t to many other blackbox PIT algorithms in the literature. However, speaking more broadly, the connection observed in Theorem 3 provides another motivation for the study of the space complexity of the PIT problem: since many of the existing and the conjectured generators are logspace-explicit, a space-efficient PIT algorithm can be seen as a prerequisite for a (time-)efficient blackbox PIT algorithm! Hence, designing a space-efficient PIT algorithm for a formula class $\mathcal{C}$ should be considered as an intermediate step between designing time-efficient whitebox and blackbox PIT algorithms for that same class.

As an example, a result of [RS05] provides a polynomial-time whitebox PIT algorithm for the model of ROABPs. Yet, the best known blackbox PIT algorithm is quasi-polynomial time [FS13, AGKS15], even for the case of constant width (unknown variable order). Our main results follow the same line of thought.

In [AvMV15], the PIT problem for multilinear formulas of bounded-read was studied. The main results are $\left(s^{k^{O(k)}}\right)$-time whitebox and $\left(s^{k^{O(k)}+k\log n}\right)$-time blackbox PIT algorithms for multilinear read-k formulas. This results in polynomial-time and quasi-polynomial-time whitebox and blackbox PIT algorithms for multilinear bounded-read formulas, respectively.

As in a typical case, the actual blackbox algorithm was obtained via an appropriate hitting-set generator. To state the result more precisely, it was shown that for every n and k, the polynomial map $G_{n,k^{O(k)}+k\log n}$ is a generator for the class of multilinear read-k formulas (for a formal definition see Definition 2.15). And while for any $t \leq n$ the map $G_{n,t}$ is logpsace-explicit (see Lemma 2.20), applying Theorem 3 on $G_{n,k^{O(k)}+k\log n}$ will result in $O(k^{O(k)} \cdot \log n \cdot \log s)$-space PIT algorithm which brings the problem only to L^2 even when $k = O(1)$. At the same time, coming up with a (truly) **polynomial-time** blackbox PIT algorithm for this class of formulas remains an open problem even for $k = 2$ (i.e. read-twice)! Consequently, unlike the case of read-once formulas (see Corollary 1.5), truly logspace PIT algorithms for multilinear read-k formulas do not seem to follow immediately from any previous result for $k \geq 2$.

1.3 Our Results

Our first main result gives the first (truly) logspace whitebox PIT algorithm for multilinear read-twice formulas. As per our previous discussion, this result can be seen as an intermediate step towards a polynomial-time black-box PIT for this class.

Theorem 4 (PIT for read-twice formulas)**.** *Let F be a multilinear read-twice formula of size s over a field $\mathbb{F}$ such that the bit-complexity of the constants appearing in F is b. Then, there exists a deterministic algorithm that, given F as an input, decides whether $F \equiv 0$, in space upper bounded by:*

- $O(\log(s) + \log(q))$ *for any finite field of size* q.
- $O(\log(s) + \log\log(p))$ *for any prime field* $\mathbb{F}_p$.
- $O(\log(s) + \log(b))$ *for* $\mathbb{Z}$ *and* $\mathbb{Q}$.

A more technical version of the result is given in Theorem 4.7 and thus Theorem 4 follows from Theorem 4.7 instantiated with the bound from Lemma 2.10. We present a proof overview of this theorem in Section 1.4.1, and give a proof in Section 4.

The next logical step following a design of a space-efficient PIT algorithm for mulitlinear read-twice formulas would be to tackle multilinear read-thrice formulas and (hopefully) multilinear formulas of a higher read. However, the main obstacle on the way to a read-thrice and, in fact, to any constant-read PIT stems from a requirement to produce a non-zero assignment of a polynomial in logarithmic space. Indeed, the approach taken in [AvMV15] and adapted to the logspace setting in the current paper, goes by first reducing the PIT problem for a read-k formula F into a special case when F can be written as sum of two read-$(k-1)$ formulas $F = F_1 + F_2$. Subsequently, the problem is reduced to the case of a single read-$(k-1)$. While we were able to carry out the first reduction in logspace (see subsequent sections for details) the only known way to perform the second reduction is by finding a "common non-zero assignment" as specified by the "Key Lemma" of [AvMV15] (See Lemma 5.6 for the formal statement).

Computing a non-zero assignment of a given non-zero polynomial is an example of the search-to-decision reduction. This is done by mirroring the algorithm that finds a satisfying assignment of a CNF formula, given an oracle to SAT, in particular, by fixing one variable at a time. Therefore, this task can be carried out easily in polynomial time given a PIT algorithm (the algorithm itself can be whitebox. For more details see e.g. [SV15]). However, due to a sequential nature of the task, it is not clear how one could perform it in logarithmic space (given an appropriate PIT algorithm).

Going back to the case of multilinear read-thrice, it is necessary and sufficient to test if two multilinear read-twice formulas F_1 and F_2 are equivalent (i.e. $F_1 \equiv F_2$) with the additional condition that the overall read of $F_1 - F_2$ is three. That is, while each variable can appear at most twice in each of the F_i-s, each variable must also appear at most three times in F_1 and F_2 **combined**. While we do not solve this case, we make a progress by solving a special case when F_1 is a read-once and F_2 is a multilinear read-twice (or vice-versa).

Theorem 5 (Testing equivalence of a read-once and a read-twice formula)**.** *Let* F_1 *be a read-once formula and* F_2 *be a multilinear read-twice formula both of size (at most)* s *over a field* $\mathbb{F}$ *such that the bit-complexity of the constants appearing in both* F_1 *and* F_2 *is* b*. Then, there exists a deterministic algorithm that given* F_1 *and* F_2 *as inputs decides whether* $F_1 \equiv F_2$*, in space upper bounded by:*

- $O(\log(s) + \log(q))$ *for any finite field of size* q.
- $O(\log(s) + \log\log(p))$ *for any prime field* $\mathbb{F}_p$.
- $O(\log(s) + \log(b))$ *for* $\mathbb{Z}$ *and* $\mathbb{Q}$.

A more technical version of the result is given in Theorem 5.12 and thus Theorem 5 follows from Theorem 5.12 instantiated with the bound from Lemma 2.10. We present a proof overview of this theorem in Section 1.4.2, and give a proof in Section 5.

1.4 Proof Overview and Techniques

1.4.1 Proof overview of Theorem 4

In this section, we present the high-level picture of the proof of Theorem 4. In fact, we prove a more general result in Section 4 (see Theorem 4.6). For the sake of conveying the main ideas, we assume that $\mathbb{F} = \mathbb{F}_q$. Let $\mathcal{C}_k$ be the class of mulitlinear read-k formulas (see Definition 2.4) over the underlying field $\mathbb{F}$. Suppose we are given whitebox access to a size-s formula $F \in \mathcal{C}_2$ over $\mathbb{F}$. We want to show that we can decide whether $F \equiv 0$ using $O(\log s + \log q)$ work-space.

The first ingredient of our algorithm is a $O(\log s + \log q)$-space whitebox PIT algorithm $\mathcal{A}$ for the class $\sum^{[2]} \mathcal{C}_1 \stackrel{\Delta}{=} \{P + Q : P, Q \in \mathcal{C}_1\}$, which is given in Theorem 4.9. We now show how to obtain a $O(\log s + \log q)$-space PIT algorithm for $\mathcal{C}_2$ using $\mathcal{A}$.

To accomplish this logspace reduction of $\mathcal{C}_2$ to a PIT algorithm for $\mathcal{A}$, we introduce a useful concept, which we call as the μ_ℓ-measure, where $\ell \in \mathbb{N}$ is a parameter. For simplicity of exposition, we give an 'intuitive meaning' of this measure. The formal description is given in Section 4.1. The measure μ_ℓ maps every formula in the class $\mathcal{C}_\ell$ to either 0 or 1 based on the following criterion: If F is a *semantically multilinear read-j formula* for some $j < \ell$ then $\mu_\ell(F) = 0$ otherwise $\mu_\ell(F) = 1$ (the converse is not entirely true as pointed in Remark 4.2). A formula P over a field $\mathbb{F}$ is said to be *semantically multilinear read-j formula* if there exists a $Q \in \mathcal{C}_j$ such that P and Q compute the same polynomial over $\mathbb{F}$. For example, 0 is a semantically multilinear read-ℓ polynomial for every $\ell \in \mathbb{N}$. We emphasise once again that this is just an intuitive explanation of the measure μ_ℓ and its exact description is given in Definition 4.1.

Now, let us see the utility of the μ_ℓ-measure in determining whether F is zero or not. We have described the space-efficient computation of the measure μ_ℓ in Lemma 4.5 and here we describe the gist of its proof. For a node v of F, let F_v be the sub-formula of F rooted at v. Since we have whitebox access to F, we show how to compute $\mu_2(F_v)$ for any node v in F. We do the computation of μ_2 of the nodes of F in the bottom-up manner. If v is a leaf node then it is a straightforward task to determine the value of $\mu_2(F_v)$. Suppose v is a non-leaf node and suppose u and w are the children of v. Then, given the values of $\mu_2(F_u)$ and $\mu_2(F_w)$, the value of $\mu_2(F_v)$ can be computed by using one of the following two tables depending on whether v is labelled with $+$ or $\times$.

$F_v = F_u + F_w$	$\mu_2(F_w) = 0$	$\mu_2(F_w) = 1$
$\mu_2(F_u) = 0$	$\mu_2(F_v) = 1$ iff $\exists i \in [n]$ s.t. x_i appears two times in F_v, it also appears in both F_u, F_w, and $\frac{\partial F_u}{\partial x_i} + \frac{\partial F_w}{\partial x_i} \not\equiv 0$	$\mu_2(F_v) = 1$
$\mu_2(F_u) = 1$	$\mu_2(F_v) = 1$	$\mu_2(F_v) = 1$

Table 1: v is a $+$ node

$F_v = F_u \times F_w$	$\mu_2(F_w) = 0$	$\mu_2(F_w) = 1$
$\mu_2(F_u) = 0$	$\mu_2(F_v) = 0$	$\mu_2(F_v) = 1$ iff $F_u \not\equiv 0$
$\mu_2(F_u) = 1$	$\mu_2(F_v) = 1$ iff $F_w \not\equiv 0$	$\mu_2(F_v) = 1$

Table 2: v is a $\times$ node

The correctness of these tables is proven in Section 4.1. Now, we argue why we need whitebox PIT algorithm $\mathcal{A}$ for $\sum^{[2]} \mathcal{C}_1$, which is evident from the first table. For instance, if $F = F_1 + F_2$,

$\mu_2(F_1) = \mu_2(F_2) = 0$, then we check if there exists a variable x_i such that x_i labels two leaves of F, x_i also labels some leaves of both F_1 and F_2, and $\frac{\partial F_1}{\partial x_i} + \frac{\partial F_1}{\partial x_i} \not\equiv 0$. Since $F \in \mathcal{C}_2, \mu_2(F_1) = \mu_2(F_2) = 0$, it follows from Definition 4.1 that $F_1, F_2 \in \mathcal{C}_1$. Then, as every formula in $\mathcal{C}_1$ is multilinear, clearly $\frac{\partial F_1}{\partial x_i}, \frac{\partial F_2}{\partial x_i} \in \mathcal{C}_1$. We first compute $\frac{\partial F_1}{\partial x_i}, \frac{\partial F_2}{\partial x_i}$ in $O(\log s)$ work-space (see Section 3.1 in this regard). It is at this point where we need $\mathcal{A}$ to determine whether $\frac{\partial F_1}{\partial x_i} + \frac{\partial F_2}{\partial x_i} \equiv 0$ or not. At the end, after computing μ_2 recursively for every node, if $\mu_2(F) = 1$ then we output 1, otherwise we know that F is a semantically read-once formula and we test whether $F \equiv 0$. We conclude by noting that PIT algorithm $\mathcal{A}$ takes $O(\log q + \log s)$ space, as observed in Theorem 4.9.

1.4.2 Proof overview of Theorem 5

We prove a more general result in Section 5 (see Theorem 5.1): Let $k \in \mathbb{N}$ be a constant and $\mathcal{C}_k$ be the class of multilinear read-k formulas. Suppose we have an S_k-explicit-generator (Definition 2.11) for the class $\mathcal{C}_k$ and a space-$O(S_{k+1})$ whitebox PIT for $\mathcal{C}_{k+1}$. Then, there exists a whitebox PIT for $\mathcal{C}_k + \mathcal{C}_{k+1}$ having *roughly* the space complexity $O(S_k + S_{k+1})$. In this section, we convey the main ideas of the proof by confining the discussion to $k = 1$. As we have a logspace-explicit hitting set generator for the class of read-once formulas $\mathcal{C}_1$ (see Observation 2.19 and Lemma 2.13), and a logspace whitebox PIT for $\mathcal{C}_2$ (see Theorem 4.7), this general result immediately gives a logspace whitebox PIT algorithm for the class $\mathcal{C}_1 + \mathcal{C}_2$. For simplicity, we assume that $\mathbb{F} = \mathbb{F}_q$.

Now, we discuss the key idea of our proof. Suppose we are given whitebox access to a size-s arithmetic formula $F \triangleq R + Q$ which computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$, where $R \in \mathcal{C}_1$ and $Q \in \mathcal{C}_2$. We want to determine whether $F \equiv 0$ or not in $O(\log q + \log s)$ space. We adapt a crucial result from [AvMV15] to the logspace setting that helps us in doing this PIT. In particular, we show that we can compute a 'special assignment' $\mathbf{a} \in \mathbb{F}^n$ in logspace such that either $F \equiv 0$ or there exists a constant $w \in \mathbb{N}$ such that for every $n > w$, the monomial $x_1 \cdots x_n$ does not divide $F(\mathbf{x} + \mathbf{a})$. Once we have this, then using a result of [SV15], it is not difficult to test whether $F \equiv 0$ in logspace. As mentioned before, this result was proven in [AvMV15] in the polynomial-time setting. In fact, their result holds for the class $\mathcal{C}_k$ for any constant $k \in \mathbb{N}$, and not just for $\mathcal{C}_1 + \mathcal{C}_2$. See the "Key lemma" of [AvMV15] for more details. The "Key lemma" was the backbone of two main theorems of [AvMV15], namely a quasi-polynomial-time blackbox PIT for $\mathcal{C}_k$, and a polynomial-time whitebox PIT for $\mathcal{C}_k$ for any constant $k \in \mathbb{N}$. Now, let us say something about this special assignment $\mathbf{a}$ for the class $\mathcal{C}_k$, where $k \in \mathbb{N}$ is an arbitrary constant. It was shown in [AvMV15] that for any $P \in \mathcal{C}_k$, the special assignment $\mathbf{a}$ for P is a common non-zero assignment of some constant-order partial derivatives of P and they showed how to compute $\mathbf{a}$ in polynomial time. If one could compute this assignment in logspace, then from the polynomial-time whitebox PIT for $\mathcal{C}_k$ given in [AvMV15], we get a logspace whitebox PIT algorithm for $\mathcal{C}_k$. But, it is not clear how to accomplish this. Our main contribution in this theorem is to show how to compute such a special assignment $\mathbf{a} \in \mathbb{F}^n$ for the class $\mathcal{C}_1 + \mathcal{C}_2$ in logspace. This result could be seen as an important step towards obtaining a logspace whitebox PIT for the class $\sum^{[2]} \mathcal{C}_2$. We have shown in Theorem 4.6 that a whitebox PIT for $\sum^{[2]} \mathcal{C}_2$ would give a logspace PIT for the class $\mathcal{C}_3$, which is not known. So, Theorem 5 could be seen as an attempt towards obtaining a logspace whitebox PIT for $\mathcal{C}_k$, where $k \geq 3$ is a constant.

Now, we show how to get hold of a special assignment for $F = R + Q$ in logspace. We prove that this special assignment is in the image of the generator $G_{n,2}$ described in Definition 2.15. Let $u \in \mathbb{N}$ be a constant and E be the set of pairs of the kind (f, I), where f is a sub-formula of either

R or Q and $I \subseteq [n]$ such that $\partial_I f \not\equiv 0$, and either of the following holds:

- f is a sub-formula of R and $|I| \leq u+1$.
- f is a sub-formula of Q, $|I| \leq u$ such that $\mu_2(f) = 0$, where the description of μ_2 is given in Definition 4.1.
- f is a sub-formula of Q such that $\mu_2(f) = 1$, $|I| \leq u$ such that for every $i \in [n]$, $\partial_{I\cup\{i\}} Q \not\equiv 0$.

We now prove that if $\mathbf{a} \in \mathbb{F}^n$ satisfies $(\partial_I f)(\mathbf{a}) \neq 0$ for every (f, I) pair given above then either $F \equiv 0$ or $x_1 \cdots x_n$ divides $F(\mathbf{x} + \mathbf{a})$, which allows us to test in logspace whether $F \equiv 0$. In this regard, we prove the following result: Suppose for every subset $I \subseteq [n], |I| \leq u+1$ that satisfies $\partial_I Q \not\equiv 0$, we have $(\partial_I Q)(G_{n,1}) \not\equiv 0$. If $\mathbf{a} \in \mathbb{F}^n$ is a non-zero assignment of $\partial_I f$ for every $(f, I) \in E$ then $\mathbf{a}$ is a special assignment for F. See Claims 5.7, 5.8, and 5.11 in this regard. This result is the heart of our proof. We also show that if for some $I \subseteq [n], |I| \leq u+1, \partial_I Q \not\equiv 0$ but $(\partial_I Q)(G_{n,1}) \equiv 0$ then $F \not\equiv 0$. It is at this point where we need a logspace whitebox PIT for the class $\mathcal{C}_2$, which is given in Theorem 4.7. Henceforth, we assume that for every subset $I \subseteq [n], |I| \leq u+1$ that satisfies $\partial_I Q \not\equiv 0$, we have $(\partial_I Q)(G_{n,1}) \not\equiv 0$.

Now, we argue how the above mentioned categorization of the elements of the set E gives us a space-efficient computational handle on the required special assignment $\mathbf{a}$. We show that one such assignment $\mathbf{a}$ is in the image of the generator $G_{n,2}$ (Definition 2.15). Let us go over these three categories of (f, I) given above one by one and talk about the first category. Let f be a sub-formula of R and $I \subseteq [n], |I| \leq u+1$. Since $\mathcal{C}_1$ consists of only multilinear formulas, it follows from [MV18] that $G_{n,1}$ hits $\partial_I f$, i.e., $\partial_I f \equiv 0$ if and only if $\partial_I f(G_{n,1}) \equiv 0$, which implies that $G_{n,2}$ also hits $\partial_I f$. Thus, there exists an $\mathbf{a} \in \mathrm{Im}(G_{n,2})$ such that for every (f, I) belonging to the first category such that $\partial_I f \not\equiv 0$, $(\partial_I f)(\mathbf{a}) \neq 0$. Now, we argue that $G_{n,2}$ also hits every non-zero $\partial_I f$, where (f, I) is in the second category. Let f be a sub-formula of Q such that $\mu_2(f) = 0$. Then, it follows from Definition 4.1 that there exists a $P \in \mathcal{C}_1$ such that P and f compute the same polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Then, for every $I \subseteq [n], |I| \leq u, G_{n,1}$ hits $\partial_I f$. Hence, $G_{n,2}$ hits $\partial_I f$. Now, let us come to the third category. Let f be a sub-formula of Q such that $\mu_2(f) = 1$. Then, from Definition 4.1, there exists a $j \in [n]$ such that x_j appears twice in f and $\partial_{x_j} f \not\equiv 0$. Let $I \subseteq [n]$ be such that $|I| \leq u$ and for every $i \in [n], \partial_{I\cup\{i\}} Q \not\equiv 0$. We want to now show that $G_{n,2}$ hits $\partial_I f$. Recall that in this case, as $\partial_I Q \not\equiv 0$, we have $(\partial_I Q)(G_{n,1}) \not\equiv 0$. We prove in Claim 5.5 that $(\partial_{I\cup\{i\}} f)(G_{n,1})$ divides $(\partial_{I\cup\{i\}} Q)(G_{n,1})$. This along with a standard property of a generator implies that as $(\partial_{I\cup\{i\}} Q)(G_{n,1}) \not\equiv 0$, we have $(\partial_{I\cup\{i\}} f)(G_{n,1}) \not\equiv 0$. Then, it follows from Lemma 2.18 that as $\partial_I f \not\equiv 0$, we get $(\partial_I f)(G_{n,2}) \not\equiv 0$. Thus, $G_{n,2}$ hits the polynomials $\partial_I f$, where (f, I) belong to the third category. This proves that there exists an $\mathbf{a} \in \mathrm{Im}(G_{n,2})$ such that either $F \equiv 0$ or for every $n > w$, $F(\mathbf{x} + \mathbf{a})$ is not divisible by the monomial $x_1 \cdots x_n$. Now, using a result from [SV15], we are able to test in logspace whether $F \equiv 0$.

1.5 Organization

The rest of the paper is organized as follows: We present some preliminary definitions and results in Section 2. In Section 3, we present logspace algorithms for some fundamental problems like computing partial derivative and computing partial assignment of an arithmetic formula. Section 4 is devoted to a logspace reduction of whitebox PIT for $\mathcal{C}_{k+1}$ to a whitebox PIT for $\sum^{[2]} \mathcal{C}_k$, using which we prove Theorem 4. In Section 5, we give a result that generalize Theorem 5. After that,

in Sections 6 and 7 we observe that the results of [GKS17] and [ASSS16] yield logspace whitebox PIT algorithms for the classes of *constant-width read-once oblivious arithmetic branching programs* and *depth-3 circuits having constant transcendence degree*. Finally, we conclude with some open questions in Section 8.

2 Preliminaries

The set of natural numbers is represented by $\mathbb{N}$. We use the $\triangleq$ symbol for defining. For an $n \in \mathbb{N}, [n] \triangleq \{1, \ldots, n\}$. We denote the sets of variables by $\mathbf{x}, \mathbf{y}, \mathbf{z}$; vectors over $\mathbb{F}$ by $\mathbf{a}, \mathbf{b}, \boldsymbol{\alpha}$; and circuit classes by upper case calligraphic letters like $\mathcal{C}$. A polynomial $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ is called *multilinear* if its individual degree is at most one. For a polynomial $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ and a vector $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{F}^n$, the shifted polynomial is $f(\mathbf{x} + \mathbf{a}) \triangleq f(x_1 + a_1, \ldots, x_n + a_n)$. We say that $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ *depends* on x_i if there exist a tuple $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{F}^n$ and $b \in \mathbb{F}$ such that

$$f(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n) \neq f(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n).$$

Further, $\mathrm{var}(f) \triangleq \{i \in [n] : f \text{ depends on } x_i\}$ It is not difficult to see that $i \in \mathrm{var}(f)$ if and only if $\frac{\partial f}{\partial x_i} \not\equiv 0$. Let F be an arithmetic formula over a field $\mathbb{F}$ and v be a node of F. Then, the sub-formula of F rooted at v is denoted by F_v or simply v, when it is clear from the context. The following fundamental result will be useful in our proofs.

Lemma 2.1 ([Alo99])**.** *Let $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ be a polynomial. Suppose that for every $i \in [n]$ the individual degree of x_i is bounded by d_i, and let $S_i \subseteq \mathbb{F}$ be such that $|S_i| > d_i$. We denote $S = S_1 \times S_2 \times \cdots \times S_n$. Then, $f \equiv 0$ iff $f|_S \equiv 0$.*

2.1 Partial Derivatives

We have taken the following definition of discrete partial derivatives from [SV15]. It is not difficult to observe that for multilinear polynomials, this definition is same as the analytic definition of partial derivatives.

Definition 2.2 (Discrete Partial Derivative)**.** *Let $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ and $x \in \{x_1, \ldots, x_n\}$. Then, the* discrete partial derivative *of f with respect to x is defined as:*

$$\frac{\partial f}{\partial x} \triangleq f|_{x=1} - f|_{x=0}.$$

Further, let $I = \{i_1, \ldots, i_r\} \subseteq [n]$ be a non-empty set of size. Then, the iterated partial derivative *of f with respect to I is defined as*

$$\frac{\partial^r f}{\partial x_{i_1} \cdots \partial x_{i_r}} \triangleq \frac{\partial}{\partial x_{i_1}} \left(\frac{\partial}{\partial x_{i_2}} \cdots \left(\frac{\partial f}{\partial x_{i_r}} \right) \cdots \right).$$

We use the following shorthand notation for partial derivatives: Let $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$. For $x \in \{x_1, \ldots, x_n\}, \partial_x f \triangleq \frac{\partial f}{\partial x}$ and for an $I \subseteq [n], I = \{i_1, \ldots, i_r\}, \partial_I f \triangleq \frac{\partial^r f}{\partial x_{i_1} \cdots \partial x_{i_r}}$.

Fact 2.3 (Chain rule of partial derivatives)**.** *Let $\mathbb{F}$ be a field, $f \in \mathbb{F}[x_1, \ldots, x_n, y]$, $h \in \mathbb{F}[x_1, x_2, \ldots, x_n]$, and $g \triangleq f|_{y=h}$. Let $x \in \{x_1, \ldots, x_n\}$ be arbitrary. Then,*

$$\partial_x g = (\partial_x f)|_{y=h} + (\partial_y f)|_{y=h} \cdot \partial_x h.$$

2.2 Multilinear bounded-read arithmetic formulas

Let $\mathbb{F}$ be a field. An arithmetic formula F over $\mathbb{F}$ is a binary tree where every leaf node is labelled by either a variable or a constant from $\mathbb{F}$, every other node is labelled either by $+$ or $\times$ operation. A leaf node of F computes its label and if v is a non-leaf of F labelled by $\circ \in \{+, \times\}$ and has children v_1, v_2 then v computes $F_{v_1} \circ F_{v_2}$, where F_{v_i} is the polynomial computed by the node v_i for every $i \in [2]$. The output of the root of F is said to be the polynomial computed by F. The size of F is defined as the number of nodes in F. We say that an arithmetic formula F is *(syntactically) multilinear* if every node of F computes a multilinear formula. Several restricted versions of arithmetic formulas are well-studied in the literature. One such restriction is given in the following definition.

Definition 2.4 (Multilinear bounded-read formulas)**.** *Let $\mathbb{F}$ be a field and F be a multilinear arithmetic formula over $\mathbb{F}$ that computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Let $k \in \mathbb{N}$. We say that F is a multilinear read-k formula if every variable in $\{x_1, \ldots, x_n\}$ labels at most k leaf nodes of F. We denote the class of mulitlinear read-k arithmetic formulas over $\mathbb{F}$ by $\mathcal{C}_{k,\mathbb{F}}$ and we drop $\mathbb{F}$ from the subscript the whenever the field $\mathbb{F}$ is clear from the context.*

One of the reasons for studying multilinear bounded-read arithmetic formulas is that developing deep understanding of such formulae might give us good insights about the class of multilinear formulas, which is an important class of arithmetic circuits. A deterministic quasi-polynomial-time blackbox PIT algorithm and a polynomial-time whitebox PIT for $\mathcal{C}_k$ were given in [AvMV15] for every constant $k \in \mathbb{N}$. The following observation shows that the class $\mathcal{C}_k$ is closed with respect to partial derivatives (Definition 2.2).

Observation 2.5. *Let $\mathbb{F}$ be a field, $k \in \mathbb{N}$, and $F \in \mathcal{C}_k$ such that F computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Then, for every $i \in [n]$, $\frac{\partial F}{\partial x_i} \in \mathcal{C}_k$.*

2.3 Generators

The problem of blackbox PIT asks for a *hitting set*, which is defined as:

Definition 2.6 (Hitting set)**.** *Let $\mathbb{F}$ be a field and $\mathcal{C}$ be a class of arithmetic circuits over $\mathbb{F}$. Then, $\mathcal{H} \subseteq \mathbb{F}^n$ is a* hitting set *for $\mathcal{C}$, if for every non-zero polynomial $f \in \mathcal{C}$, there exists a point $\mathbf{a} \in \mathcal{H}$ such that $f(\mathbf{a}) \neq 0$.*

A blackbox PIT algorithm for a class of arithmetic circuits $\mathcal{C}$ is *efficient* when $\mathcal{H}$ contains $\text{poly}(s, d)$ many points and can also be constructed in $\text{poly}(s, d)$ time, where s is the size and d is the degree of the circuit given as input to the PIT algorithm. A common convention for polynomials over finite fields is that the size of the field is polynomially-large in the size and the degree of the given circuit or we assume that we have blackbox access to a large enough extension field. Another important notion for blackbox PIT is that of *hitting set generators* (or simply generators). For any class of arithmetic circuits, the power of generators and hitting sets are equivalent (in polynomial-time), and it is 'often' easier to work with generators. We refer the interested reader to a beautiful survey of [SY10] for a detailed exposition of PIT.

Definition 2.7 (Generator)**.** *Let $\mathbb{F}$ be a field and $\mathcal{C}$ be a class of n-variate polynomials over $\mathbb{F}$. Consider the polynomial map $\mathcal{G} = (\mathcal{G}^1, \mathcal{G}^2, \ldots, \mathcal{G}^n) : \mathbb{F}^t \to \mathbb{F}^n$, where for each $i \in [n]$, $\mathcal{G}^i \in \mathbb{F}[y_1, y_2, \ldots, y_t]$. For a polynomial $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$, we define action of $\mathcal{G}$ on polynomial f*

by $f(\mathcal{G}) = f(\mathcal{G}^1, \ldots, \mathcal{G}^n) \in \mathbb{F}[y_1, \ldots, y_k]$. *We call* $\mathcal{G}$ *a* t*-seeded generator for class* $\mathcal{C}$ *if for every non-zero* $f \in \mathcal{C}$, $f(\mathcal{G}) \not\equiv 0$. *Degree of generator* $\mathcal{G}$ *is defined as* $\deg(\mathcal{G}) \triangleq \max\{\deg(\mathcal{G}^i)\}_{i=1}^n$. *Image of generator* $\mathcal{G}$ *is defined as* $\mathrm{Im}(\mathcal{G}) \triangleq \mathcal{G}(\mathbb{F}^t)$.

A generator $\mathcal{G}$ for class $\mathcal{C}$ acts as a variable reduction map that reduces the number of variables from n to t while preserving the non-zeroness. It immediately follows from the above definition that a generator contains a *hitting set* for $\mathcal{C}$ in its image.

Fact 2.8 (Generator $\implies$ hitting-set, [SV15])**.** *Let* $\mathbb{F}$ *be a field and* $\mathcal{G} = (\mathcal{G}^1, \ldots, \mathcal{G}^n) : \mathbb{F}^t \to \mathbb{F}^n$ *be a generator for a circuit class* $\mathcal{C}$ *such that* $\deg(\mathcal{G}) \triangleq \delta$. *Let* $W \subseteq \mathbb{F}$ *be any set of size* $nd\delta$. *Then,* $\mathcal{H} \triangleq \mathcal{G}(W^t)$ *is a hitting set, of size* $|\mathcal{H}| \leq (nd\delta)^t$, *for polynomials* $f \in \mathcal{C}$ *of individual degrees less than* d.

In other words, when the seed-length t and the degree δ of the generator is constant, we get a polynomial-time blackbox PIT algorithm. In the following section we will discuss hitting-set generators through the lens of space complexity.

2.4 From Log-space-Explicit Generator to Log-space PIT

We call an arithmetic formula F over a field $\mathbb{F}$ as size-s, bit-complexity-b formula if the size of F is s and the bit complexity of the field constants appearing in F is (at most) b. Consider the following definition.

Definition 2.9 (The space complexity of formula evaluation)**.** *Let* $\mathbb{F}$ *be a field. Then,* $e_{\mathbb{F}}(s, b)$ *is defined to be the space complexity of computing* $F(\mathbf{a})$, *where* F *is an arbitrary size-*s, *bit-complexity-*b *arithmetic formula over* $\mathbb{F}$ *and* $\mathbf{a} \in \mathbb{F}^n$ *is an arbitrary tuple.*

The following upper bounds on $e_{\mathbb{F}}(s, b)$ follow from [BCGR92] and [BC92]. Also see [CDL01, HAM02, MW07].

Lemma 2.10. *The quantity* $e_{\mathbb{F}}(s, b)$ *is upper bounded by*

- $O(\log(s) + \log(q))$ *for any finite field of size* q.
- $O(\log(s) + \log\log(p))$ *for any prime field* $\mathbb{F}_p$.
- $O(\log(s) + \log(b))$ *for* $\mathbb{Z}$ *and* $\mathbb{Q}$

The second and the third cases of Lemma 2.10 use the Chinese Remainder Theorem, hence it is unclear how to extend the result to arbitrary fields (e.g. arbitrary infinite fields or finite fields of size p^ℓ for $\ell \geq 2$).

In order to formalize the notions of logspace-explicit generators and relate them to some of the existing constructions, we require the following slightly more general definitions which are inspired from the definition of a log-space computable function given in [AB09]. Recall that a (hitting-set) generator (Definition 2.7) is a polynomial map.

Definition 2.11 (Space-explicit polynomial maps)**.** *Let* $\mathbb{F}$ *be a field,* $S : \mathbb{N} \to \mathbb{N}, t : \mathbb{N} \to \mathbb{N}$, $n \in \mathbb{N}$, *and* $P : \mathbb{F}^{t(n)} \to \mathbb{F}^n$ *be a polynomial map. We say that* P *is* $S(n)$-space-explicit *if there exists a Turing machine that given an* $\boldsymbol{\alpha} \in \mathbb{F}^{t(n)}$ *writes* $P(\boldsymbol{\alpha})$ *on the output tape using* $O(S(n))$ *amount of work-space.*

Definition 2.12 (Space-uniformity of a polynomial map)**.** *Let $\mathbb{F}$ be a field, $S : \mathbb{N} \to \mathbb{N}, t : \mathbb{N} \to \mathbb{N}$, $n \in \mathbb{N}$, and $P : \mathbb{F}^{t(n)} \to \mathbb{F}^n$ be a polynomial map. Then, P has an $S(n)$-space-uniform formula if there exists a Turing machine that given the parameters $n, t(n)$ outputs an arithmetic formula over $\mathbb{F}$ that computes P, using the work-space $O(S(n))$.*

The above definitions naturally give rise to the notions of an *$S(n)$-space-explicit generator* and an *$S(n)$-space-uniform generator*, respectively. The following claim relates the two definitions.

Lemma 2.13. *Let $\mathbb{F}$ be a field, $S : \mathbb{N} \to \mathbb{N}, t : \mathbb{N} \to \mathbb{N}$, $n \in \mathbb{N}$, and $P : \mathbb{F}^{t(n)} \to \mathbb{F}^n$ be a polynomial map. If P has an $S(n)$-space-uniform formula of size s and bit-complexity b over $\mathbb{F}$ then P is $(S(n) + \log s + e_{\mathbb{F}}(s, b))$-space-explicit.*

Proof. We want to show that there exists a Turing machine that takes as input $\boldsymbol{\alpha} \in \mathbb{F}^{t(n)}$ and computes $P(\boldsymbol{\alpha})$ using $O(S(n) + \log s + e_{\mathbb{F}}(s, b))$ work-space. As P has a $O(S(n))$-space-uniform formula, we know from Definition 2.12 that there exists a Turing machine T_1 that takes input $n, t(n)$, and outputs a size-s, bit complexity-b arithmetic formula F over the field $\mathbb{F}$ using $O(S(n))$ work-space such that F computes P. We know from Lemma 2.10 that there exists a Turing machine T_2 that takes input F and a $\boldsymbol{\alpha} \in \mathbb{F}^n$ and outputs $F(\boldsymbol{\alpha})$ in work-space $O(\log s + e_{\mathbb{F}}(s, b))$. Let $T_1 \circ T_2$ denote the a Turing machine computing the composition of functions computed by T_1 and T_2. Then, $T_1 \circ T_2$ takes input P and a $\boldsymbol{\alpha} \in \mathbb{F}^{t(n)}$ and outputs $P(\boldsymbol{\alpha})$ using the work-space $O(S(n) + \log s + e_{\mathbb{F}}(s, b))$. □

In the last theorem of this section we show how to connect all the dots obtaining a space-efficient whitebox PIT algorithm from a space-explicit generator.

Theorem 2.14. *Let $S : \mathbb{N} \to \mathbb{N}$ and let F be an arithmetic formula of size s and bit-complexity b computing a non-zero polynomial over $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Let $H : \mathbb{F}^t \to \mathbb{F}^n$ be an $S(n)$-space-explicit polynomial map of degree at most δ such that $F(H) \not\equiv 0$. Then there exists an algorithm, that given F as input, outputs an assignment $\mathbf{a} \in \mathbb{F}^n$ such that $F(\mathbf{a}) \neq 0$, using $O(t \cdot (\log(\delta) + \log(s)) + S(n) + e_{\mathbb{F}}(s, b))$ space.*

Proof. Since F has a formula of size s, $F(H)$ is a non-zero polynomial of degree less than δs. Let $V \subseteq \mathbb{F}$ be a set of size $|V| = \delta s$. We can pick V such that bit-complexity of each element is $O(\log |V|) = O(\log \delta + \log s)$. By Lemma 2.1, there exists $\boldsymbol{\alpha} \in V^t$ such that $F(H(\boldsymbol{\alpha})) \neq 0$. Therefore, to output an $\mathbf{a}$ such that $F(\mathbf{a}) \neq 0$, it suffices to go over each $\boldsymbol{\alpha} \in V^t$, and check if $F(H(\boldsymbol{\alpha})) \neq 0$, where the bit-complexity of each element $\boldsymbol{\alpha}$ is $O(t(\log \delta + \log s))$.

Since H is $O(S(n))$-space-explicit, by Definition 2.11, there is a Turing machine that, given $\boldsymbol{\alpha} \in \mathbb{F}^t$, outputs $H(\boldsymbol{\alpha})$ using $O(S(n))$ work-space. By Lemma 2.10, $F(H(\boldsymbol{\alpha}))$ can be computed using $e_{\mathbb{F}}(s, b)$ work-space given $H(\boldsymbol{\alpha})$. By iterating over all $\boldsymbol{\alpha} \in V^t$ and combining the above steps, a non-zero assignment $\mathbf{a} \in \mathbb{F}^n$ such that $F(\mathbf{a}) \neq 0$ can be computed in space $O(t(\log \delta + \log s) + S(n) + e_{\mathbb{F}}(s, b))$. □

2.5 The Generator $G_{n,k}$ of [SV15]

The generator $G_{n,k}$ was defined in [SV15] for the class of arithmetic read-once formulas. It has been a crucial ingredient in PIT algorithms of various other interesting classes also [KMSV13, FSS14, AvMV15, MV18, BGV23]. We will also be using this generator in our results. We borrow the definition and properties of this generator as presented in [AvMV15, Vol15].

Definition 2.15. *Let $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$ be n distinct elements and for $i \in [n]$, let $L_i(x) \triangleq \prod_{j \in [n] \setminus \{i\}} \frac{x-\alpha_j}{\alpha_i - \alpha_j}$ denote the corresponding Lagrange interpolant. For every $k \in [n]$, let $G_{n,k} : \mathbb{F}^{2k} \to \mathbb{F}^n$ be defined as*

$$G_{n,k}(y_1, \ldots, y_k, z_1, \ldots, z_k) \triangleq \left(\sum_{j=1}^{k} L_1(y_j) z_j, \sum_{j=1}^{k} L_2(y_j) z_j, \ldots, \sum_{j=1}^{k} L_n(y_j) z_j \right)$$

Let $(G_{n,k})_i$ denote the i^{th} component of $G_{n,k}$ and we call α_i as the Lagrange constant *associated with this i^{th} component. We can also define G_k to be the class of generators $\{G_{n,k}\}_{n \in \mathbb{N}}$ for all output lengths.*

For two generators $\mathcal{G}_1, \mathcal{G}_2$ with the same output length, we define their sum $\mathcal{G}_1 + \mathcal{G}_2$ as their component-wise addition, where the seed variables of both generators are implicitly relabelled so as to be disjoint. With this terminology, we can note various useful properties of the generator $G_{n,k}$ from its definition.

Fact 2.16 ([SV15, KMSV13, Vol15])**.** *Let k, k' be positive integers.*

1. $G_{n,k}(\mathbf{y}, \mathbf{0}) \equiv \mathbf{0}$.
2. $G_{n,k}(y_1, \ldots, y_k, z_1, \ldots, z_k)|_{y_k = \alpha_i} = G_{n,k-1}(y_1, \ldots, y_{k-1}, z_1, \ldots, z_{k-1}) + z_k \cdot \mathbf{e}_i$, *where $\mathbf{e}$ is the 0-1 vector with a single 1 in coordinate i and α_i the i^{th} Lagrange constant and $G_{n,0} \triangleq \mathbf{0}$.*
3. $G_{n,k}(y_1, \ldots, y_k, z_1, \ldots, z_k) + G_{n,k'}(y_{k+1}, \ldots, y_{k+k'}, z_{k+1}, \ldots, z_{k+k'})$
   $= G_{n,k+k'}(y_1, \ldots, y_{k+k'}, z_1, \ldots, z_{k+k'})$.
4. *For every $\mathbf{b} \in \mathbb{F}^n$ with at most k non-zero components, $\mathbf{b} \in \mathrm{Im}(G_{n,k})$.*

The observation below immediately follows from Definition 2.15.

Observation 2.17. *Let $\mathcal{C}$ be a class of arithmetic circuits over a field $\mathbb{F}$ and $H = (H_i)_{i \in \mathbb{N}}$ be a hitting-set generator for $\mathcal{C}$. Then, for every $k \in \mathbb{N}$, $H + G_k$ is also a generator for $\mathcal{C}$.*

Lemma 2.18 (Implicit in [SV15], Lemma 2.14 of [AvMV15])**.** *Let $P \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ and H_n be such that there exists a variable x_j such that $(\partial_{x_j} P)(H_n) \not\equiv 0$. Then $P(H_n + G_{n,1}) \not\equiv 0$.*

2.5.1 Log-space Explicitness

Now, let us talk about the space-explicitness and space-uniformity of $G_{n,k}$. Observe that over a finite field $\mathbb{F}_q$ it is clear that for every $k \leq n$, the map is $(\log n + \log q)$-space-explicit. However, it is not immediately clear if this is also the case over $\mathbb{Q}$ or any other field of characteristic zero. The n-fold multiplication in the definition of $G_{n,k}$ (Definition 2.15) may give the impression that the map is *not log-space-explicit*. We, nonetheless, show that $G_{n,k}$ is log-space explicit by first observing that it has a log-space-uniform formula with the right set parameters. The claim then follows immediately from Lemma 2.13.

Observation 2.19. *Let $n, k \in \mathbb{N}$ be such that $k \leq n$ and $\mathbb{F}$ be a field of size $|\mathbb{F}| > n$. Then, $G_{n,k}$ over $\mathbb{F}$ has a $O(\log n)$-space-uniform formula of size $s = O(n^3)$ and bit-complexity $b = O(\log n)$[6].*

[6]Note that definition 2.15 only requires the elements a_i to be distinct. Hence, we can choose the first n lexicographically-smallest elements.

The claim below follows from Observation 2.19 and Lemma 2.13.

Claim 2.20. *Let $n, k \in \mathbb{N}$ be such that $k \leq n$ and $\mathbb{F}$ be a field of size $|\mathbb{F}| > n$. Then, $G_{n,k}$ over $\mathbb{F}$ is $O(\log n + e_{\mathbb{F}}(n^3, \log n))$-space-explicit.*

3 Generic traversal of a formula

In this section, let F be an arithmetic formula over a field $\mathbb{F}$. We can think of F a straight-line program where each instruction is of form:

$$g_a \leftarrow g_b \text{ op } g_c \quad \text{where } b < a \text{ or } g_b \in \mathbf{x} \cup \mathbb{F} \text{ (and similarly } g_c)$$

Since F is a formula, every gate g_i can be used at most once in a later instruction.

We give a meta-procedure in Algorithm 1, whose main objective is to compute a given function $\Psi \triangleq (\Psi_{\text{leaf}}, \Psi_+, \Psi_\times)$ in a space-efficient way. In particular, the computation of Ψ on F will be defined recursively: if g is a leaf of F, then $\Psi(g) \triangleq \Psi_{\text{leaf}}(g)$. If g is a gate (i.e. operation) in F, then $\Psi(g)$ is defined by applying Ψ_+ or $\Psi_\times$ in the case of the $+$ and $\times$ gate, respectively (we assume that Ψ values have already been computed for both g's children). Formally, suppose we have $g_a \leftarrow g_b$ op g_c for $b, c < a$. Then

$$\Psi(g_a) \triangleq \begin{cases} \Psi_+ \left(\Psi(g_b), \Psi(g_c)\right), & \text{if op} = + \\ \Psi_\times \left(\Psi(g_b), \Psi(g_c)\right), & \text{if op} = \times. \end{cases}$$

Our procedure will perform a *post-order* traversal, remembering only the current and the previous positions[7]. In addition to computing a returned value (denoted by γ), Ψ may also output a modified formula F' on the output tape. This will be denoted by OUT. It is to be noted that the main difference between the returned value and the output content, written on the tape, is that the returned value is placed on the work tape of the machine and thus can be used in subsequent computations, whereas, the content of the output tape is, by definition, write-only and thus **cannot** be read again. At the same time, this content is not "charged" to the space complexity of the Turing machine.

If s is the size of the formula F, then each node in the formula can be indexed using $O(\log s)$ bits. Furthermore, observe that given a node, the standard operations of determining its parent, left child or right child can all be carried out in $O(\log s)$ space. To simplify the algorithm and its analysis, we first preprocess the formula by transforming it into a left-heavy form (see Section C of Appendix). As this transformation can be carried out in $O(\log s)$ space, we can assume for our analysis that any given formula is already left-heavy.

Lemma 3.1. *Let F be an arithmetic formula of size s over a field $\mathbb{F}$. Let S_Ψ denote the maximum space-complexity of the procedures $\Psi_{leaf}, \Psi_+, \Psi_\times$ and let $t \geq 1$ denote the maximum bit-complexity of the output values of these functions. Then given F as an input, Algorithm 1 returns the value of $\Psi(F)$ (and might also write on the output tape), using $O(S_\Psi + t \cdot \log s)$ work space.*

The proof follows along the same lines as previous works [BCGR92, Lin92, DLN+22] but we give it in a more general form as we will be using it to perform different functionalities using the same framework. For completeness, we give the proof of Lemma 3.1 in Section B of the Appendix.

[7]Note that remembering the entire traversal history would require linear memory thus rendering the procedure space-inefficient.

Algorithm 1: Generic traversal for a formula

Input: An arithmetic formula F, functions Ψ_{leaf}, Ψ_+, $\Psi_\times$ and stack Γ.
Output: $\Psi(F)$.

Preprocessing: Transform F into a left-heavy form.
Initialize curr $\stackrel{\Delta}{=}$ root(F), prev $\stackrel{\Delta}{=} \perp$, $\alpha = \beta = \perp$, $\Gamma = \emptyset$.
`/* curr and prev track the current and previous nodes, while` α `will store the popped` Ψ `value from stack and` β `will store the last` Ψ `value returned. */`
while curr $\neq \perp$ **do**
  `/* if curr is leaf, compute` β `and go up */`
  if curr *is a leaf* **then**
    $\beta \leftarrow \Psi_{\text{leaf}}$(curr), prev $\leftarrow$ curr, curr $\leftarrow$ parent(curr)
    continue. `/* Skip to next iteration of loop */`
  end
  `/* curr is not a leaf in the below cases */`
  `/* If coming from up, go left */`
  if prev = parent(curr) **then** prev $\leftarrow$ curr, curr $\leftarrow$ curr.left, **continue**.
  `/* If coming from left, push last value to stack and go right */`
  if prev = curr.left **then**
    Γ.push(β), prev $\leftarrow$ curr, curr $\leftarrow$ curr.right, **continue**.
  end
  `/* If coming from right,` β `has` Ψ `value of right child and top of stack has` Ψ `value of left child */`
  if prev = curr.right **then**
    $\alpha \leftarrow \Gamma$.pop().
    if curr.op = '$+$' **then** $\beta \leftarrow \Psi_+(\alpha, \beta)$ **else** $\beta \leftarrow \Psi_\times(\alpha, \beta)$.
    `/* Go up */`
    prev $\leftarrow$ curr, curr $\leftarrow$ parent(curr).
  end
end
return β. `/* returning root's` Ψ `value */`

3.1 Partial derivative in logspace

Let F be a syntactically multilinear formula that computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. We will describe a logspace algorithm that outputs the partial derivative of F w.r.t to a variable x_i when $i \in [n]$. Towards that end, we use the generic traversal algorithm (Algorithm 1) for computing $\partial_{x_i} F$ by defining an appropriate Ψ function, $\Psi^i = (\Psi^i_{\text{leaf}}, \Psi^i_+, \Psi^i_\times)$. In fact, we will compute a somewhat more general function which will be required for the recursive argument.

For each gate g, the function Ψ^i will simultaneously compute a value $\gamma(g)$ and write on the output tape (denoted by OUT below), based on an appropriate modification of the input formula F.

We first define Ψ^i_{leaf} as follows:

$$\Psi^i_{\text{leaf}}(g) \triangleq \begin{cases} \langle \gamma(g) = 1, \text{OUT} : 1 \rangle, & \text{if } g = x_i, \\ \langle \gamma(g) = 0, \text{OUT} : g \rangle, & \text{otherwise.} \end{cases} \tag{1}$$

We give the following table description (Table 3) for the definition of Ψ^i_+ for an addition gate which is of the form $g_a \leftarrow g_b + g_c$.

$g_a \leftarrow g_b + g_c$	$\gamma(g_c) = 0$	$\gamma(g_c) = 1$
$\gamma(g_b) = 0$	$\langle \gamma(g_a) = 0, \text{OUT} : g_a \leftarrow g_b + g_c \rangle$	$\langle \gamma(g_a) = 1, \text{OUT} : g_a \leftarrow 0 + g_c \rangle$
$\gamma(g_b) = 1$	$\langle \gamma(g_a) = 1, \text{OUT} : g_a \leftarrow g_b + 0 \rangle$	$\langle \gamma(g_a) = 1, \text{OUT} : g_a \leftarrow g_b + g_c \rangle$

Table 3: Definition of Ψ^i_+ to compute $\frac{\partial g}{\partial x_i}$

We give the following table description (Table 4) for definition of $\Psi^i_\times$ for a multiplication gate which is of the form $g_a \leftarrow g_b \times g_c$.

$g_a \leftarrow g_b \times g_c$	$\gamma(g_c) = 0$	$\gamma(g_c) = 1$
$\gamma(g_b) = 0$	$\langle \gamma(g_a) = 0, \text{OUT} : g_a \leftarrow g_b \times g_c \rangle$	$\langle \gamma(g_a) = 1, \text{OUT} : g_a \leftarrow g_b \times g_c \rangle$
$\gamma(g_b) = 1$	$\langle \gamma(g_a) = 1, \text{OUT} : g_a \leftarrow g_b \times g_c \rangle$	Impossible

Table 4: Definition of $\Psi^i_\times$ to compute $\frac{\partial g}{\partial x_i}$

Remark 3.2. *We want to emphasize that in Tables 3 and 4, g_b and g_c are not the gates of the original formula on the input tape, rather the gates written on the output tape after Ψ function has been applied to them. See the proof of Lemma 3.3 for more clarity.*

Lemma 3.3. *Given a multilinear formula F over $\mathbb{F}[x_1, x_2, \ldots, x_n]$ and $i \in [n]$, $\Psi^i(F)$ writes the formula for $\frac{\partial F}{\partial x_i}$ on the output tape, if x_i appears as a leaf in F, otherwise it outputs F itself.*

Proof. For each gate g, by the recursive definition of Ψ^i, the following invariant holds by induction on the structure of g:

$$\Psi^i(g) \triangleq \begin{cases} \langle \gamma = 1, \text{OUT} : \frac{\partial g}{\partial x_i} \rangle, & \text{if } x_i \text{ appears in the subformula } g, \\ \langle \gamma = 0, \text{OUT} : g \rangle, & \text{otherwise.} \end{cases} \tag{2}$$

- If g is a leaf, then the invariant follows directly from Equation (1).
- Suppose g is of the form $g_a = g_b + g_c$. By the sum rule: $\frac{\partial g_a}{\partial x_i} = \frac{\partial g_b}{\partial x_i} + \frac{\partial g_c}{\partial x_i}$. If x_i does not appear in either of the subformulas g_b or g_c, i.e. $\gamma(g_b) = \gamma(g_c) = 0$, then x_i does not appear in the subformula g_a either. Therefore, $\gamma(g_a)$ should be 0, which is indeed the case in Table 3. Moreover, by the induction hypothesis, since γ value for both g_b and g_c is 0, the corresponding gates g_b and g_c on the output tape will compute their original subformulas. Thus, when we write the instruction $g_a \leftarrow g_b + g_c$ on the output tape, as done in Table 3, we maintain the invariant for the current gate g. Now, without loss of generality, suppose x_i appears only in the subformula g_b but not g_c, i.e. $\gamma(g_b) = 1$ and $\gamma(g_c) = 0$. Then, x_i appears in g_a and $\gamma(g_a) = 1$. Moreover, we know that $\frac{\partial g_a}{\partial x_i} = \frac{\partial g_b}{\partial x_i} + 0$. Since $\gamma(g_b) = 1$, by the induction hypothesis, the corresponding gate g_b on the output tape computes the correct subformula for $\frac{\partial g_b}{\partial x_i}$. Therefore, we write the instruction $g_a \leftarrow g_b + 0$ for the current gate g on the output tape. Finally, when x_i appears in both g_b and g_c, i.e. $\gamma(g_b) = \gamma(g_c) = 1$, x_i appears syntactically in g_a also, thus $\gamma(g_a) = 1$. Since $\frac{\partial g_a}{\partial x_i} = \frac{\partial g_b}{\partial x_i} + \frac{\partial g_c}{\partial x_i}$, and by the induction hypothesis, the corresponding gates g_b, g_c on the output tape compute the correct subformulas for $\frac{\partial g_b}{\partial x_i}, \frac{\partial g_c}{\partial x_i}$ respectively, we write the instruction $g_a \leftarrow g_b + g_c$ on the output tape. Thus in all the cases of Ψ^i_+ in Table 3, we maintain the invariant.
- Suppose g is of the form $g_a = g_b \times g_c$. Note that since F is a multilinear formula g_b and g_c are variable disjoint, hence x_i cannot appear in both these subformulas. If x_i does not appear in both g_b, g_c, then $\gamma(g_a) = \gamma(g_b) = \gamma(g_c) = 0$. By the induction hypothesis, the corresponding gates g_b, g_c on the output tape compute their original subformulas, therefore when we write the instruction $g_a \leftarrow g_b \times g_c$ on the output tape, we maintain the invariant for the current gate g. For the case when variable x_i appears in exactly one of g_b or g_c, without loss of generality, say g_b, we have $\gamma(g_b) = 1$ and $\gamma(g_c) = 0$. Thus $\gamma(g_a) = 1$. Moreover, $\frac{\partial g_a}{\partial x_i} = g_c \cdot \frac{\partial g_b}{\partial x_i}$ since x_i appears in g_b but not in g_c. By the induction hypothesis, the corresponding gate g_b on the output tape computes the subformula for $\frac{\partial g_b}{\partial x_i}$ while the corresponding gate g_c on the output tape computes its original subformula g_c. Therefore, when we write $g_a \leftarrow g_b \times g_c$, we maintain the invariant for the current gate g. Thus in all the cases of $\Psi^i_\times$ in Table 4, we maintain the invariant.

Thus, the invariant holds for every gate g in F and in particular, it holds for the root node. Hence, the lemma holds true. □

Remark 3.4. *In the case when x_i does not appear in g, we output g itself instead of 0 on the output tape. This is because, when computing a partial derivative of some multiplication gate $g_a = g_b \times g_c$, where x_i appears in only one of the children gates (say g_b), we know that $\frac{\partial g_a}{\partial x_i} = \frac{\partial g_b}{\partial x_i} \times g_c$. Had we chosen to output 0 previously for g_c, we would lose access to g_c for this computation.*

Remark 3.5. *It could be the case that the variable x_i appears in the formula F but F does not depend on x_i. In that case, the modified formula written on the output tape will actually compute an identically zero polynomial.*

We conclude this section with the desired logspace algorithm to compute partial derivative of a multilinear formula.

Lemma 3.6 (Algorithm A-PARTIAL)**.** *There exists an algorithm that given a multilinear formula F of size s over $\mathbb{F}[x_1, x_2, \ldots, x_n]$ and $i \in [n]$, computes $\frac{\partial F}{\partial x_i}$, using $O(\log s)$ work-space. Moreover, if F is a read-ℓ formula then so is the output of the algorithm.*

Proof. We first compute $\Psi^i(F)$ by plugging the definition of $\Psi = (\Psi^i_{\text{leaf}}, \Psi^i_+, \Psi^i_\times)$ from (1), Table 3 and Table 4 in Algorithm 1. By Lemma 3.3, note that if $\Psi^i(F) = 1$, then we have $\frac{\partial F}{\partial x_i}$ written in the output tape. However, when $\Psi^i(F) = 0$, we have F written in the output tape. Therefore, after computing $\Psi^i(F)$, we post-process it with a simple $O(\log s)$ space procedure as follows: If the (γ) value of $\Psi^i(F)$ is 1, then we copy the entire formula to the output tape, else we write 0 on the output tape.

Correctness: follows from Lemma 3.3 and correctness of Algorithm 1.

Space-complexity: Observe that the space required to execute Ψ^i procedure is $S^i_\Psi = O(\log s)$, which follows from the description of $\Psi^i_{\text{leaf}}, \Psi^i_+, \Psi^i_\times$ in (1) and Tables 3, 4. Note that the returned value γ is just a single bit and hence $t = 1$. Therefore by Lemma 3.1, we use $O(\log s)$-space to compute $\Psi^i(F)$ using Algorithm 1. The additional post-processing step can also be done in $O(\log s)$ space trivially and hence, the total work space used is $O(\log s)$. □

3.2 Partial assignment in logspace

We say that a variable x_i *appears* ℓ times in F, if x_i labels exactly ℓ leaf nodes of F. In this section, we introduce a logspace procedure A-SUBSTITUTE that given a formula F and $\ell \in \mathbb{N}$, as input, will replace every variable that appears at least ℓ times in F, with the field element 0. The outcome of this procedure results in a formula obtained by setting $x_i = 0$ for each variable x_i that appears at least ℓ times in F.

For this procedure, we will invoke a subroutine A-COUNT, which given a formula F and variable x_i, counts the number of appearances of x_i in F. This can be done in logspace by invoking Algorithm 5 with a slight modification: In Line 4, we increment *size* **only** if curr is labelled by x_i.

Given the above, for each leaf, A-SUBSTITUTE will invoke A-COUNT. If the returned value of A-COUNT is at least ℓ, then we will replace this particular leaf with 0, otherwise we keep it as is. We invoke this subroutine for every leaf node that is a variable and reuse the space to run this subroutine for different leaf nodes. We describe the whole procedure formally by defining $\Psi^\ell = (\Psi^\ell_{\text{leaf}}, \Psi^\ell_+, \Psi^\ell_\times)$ below and plugging it into Algorithm 1.

$$\Psi^\ell_{\text{leaf}}(g) \triangleq \begin{cases} \langle \text{OUT} : 0 \rangle,^{8} & \text{if } g \text{ is a variable that appears at least } \ell \text{ times in } F, \\ \langle \text{OUT} : g \rangle, & \text{otherwise.} \end{cases}$$

[8] For the sake of simplicity, we do not mention the returned value γ in the definition of Ψ as we do not care about it but we can always return any dummy bit, say 0.

For the internal computation nodes, we just copy the instructions as it is. We define

$$\Psi^{\ell}_{+}(g) = \Psi^{\ell}_{\times}(g) \triangleq \langle \text{OUT} : g \rangle.$$

Lemma 3.7. *Given a formula F, the function $\Psi^{\ell}(F)$ outputs a modified formula F' that results from F by substituting 0 into every variable that appears at least ℓ times.*

Proof. By definition of $\Psi^{\ell} = (\Psi^{\ell}_{\text{leaf}}, \Psi^{\ell}_{+}, \Psi^{\ell}_{\times})$, we copy every gate unless it is a leaf node corresponding to some variable which appears ℓ or more times in F. In the first case of $\Psi^{\ell}_{\text{leaf}}$, we check whether a variable leaf node appears at least ℓ times, by invoking the subroutine A-COUNT (modified Algorithm 5), to count the number of occurrences of the variable leaf node g. If it does, we relabel it with 0 on the output tape, as stated in $\Psi^{\ell}_{\text{leaf}}$. This is equivalent to substituting a 0 into every variable that appears at least ℓ times in F. □

We now conclude the section with the desired logspace algorithm to compute partial assignment.

Lemma 3.8 (Algorithm A-SUBSTITUTE)**.** *There exist an algorithm that given a formula F of size s, outputs a modified formula F' that results from F by substituting 0 into every variable that appears at least ℓ times, using $O(\log s)$ work-space.*

Proof. We plug $\Psi = (\Psi^{\ell}_{\text{leaf}}, \Psi^{\ell}_{+}, \Psi^{\ell}_{\times})$ in Algorithm 1.

Correctness: Follows from Lemma 3.7 and correctness of Algorithm 1.

Space-complexity: Since $\Psi^{\ell}_{+}, \Psi^{\ell}_{\times}$ merely copy the gate, we only need to analyze space-complexity of $\Psi^{\ell}_{\text{leaf}}$. Observe that it invokes Algorithm A-COUNT (modified Algorithm 5) as a subroutine, which takes only $O(\log s)$ space, which we keep reusing for different leaf nodes. Therefore $S_{\Psi} = O(\log s)$. Moreover, return value is a single dummy bit, thus $t = 1$. Then by Lemma 3.1, the whole algorithm takes $O(\log s)$ space. □

4 Logspace reduction of PIT for $\mathcal{C}_{k+1}$ to PIT for $\sum^{[2]} \mathcal{C}_k$

Let $k \in \mathbb{N}$ and $\mathbb{F}$ be a field. Recall that $\mathcal{C}_k$, be the class of multilinear read-k arithmetic formulas (Definition 2.4) over $\mathbb{F}$. For a class of arithmetic formulas $\mathcal{C}$, we define the associated class $\sum^{[2]} \mathcal{C} \triangleq \{F_1 + F_2 : F_1, F_2 \in \mathcal{C}\}$.

In Section 4.1, we present a tool that will be used in the desired logspace reduction given in Section 4.4. This tool is distilled from the work of [AvMV15], where it was used implicitly and implemented in polynomial time. Indeed, we extract the core concept and show how to implement it in logarithmic space. Subsequently, in Section 4.5 we use the reduction from Section 4.4 to give a logspace whitebox PIT algorithm for arithmetic read-twice formulas.

4.1 The measure μ_{ℓ}

Recall that for a multilinear polynomial $f \in \mathbb{F}[x_1, x_2, \ldots, x_n]$, f depends on a variable x_i if and only if $\frac{\partial f}{\partial x_i} \not\equiv 0$. Below, let F be an arithmetic formula over a field $\mathbb{F}$.

Definition 4.1 (The measure μ_ℓ)**.** *Let $\ell \in \mathbb{N}$. We define the measure μ_ℓ as $\mu_\ell : \mathcal{C}_\ell \to \{0,1\}$, where for any formula $F \in \mathcal{C}_\ell$:*

$$\mu_\ell(F) = \begin{cases} 1, & \text{if there exists an } x_i \in \mathrm{var}(F) \text{ such that } x_i \text{ appears } \ell \text{ times in } F \\ 0, & \text{otherwise.} \end{cases}$$

Equivalently, $\mu_\ell(F) = 0$ if and only if either every variable appears in F at most $\ell - 1$ times or for all x_i that appear in F exactly ℓ times, $x_i \notin \mathrm{var}(F)$.

Remark 4.2. *Note that if $\mu_\ell(F) = 0$ for a read-ℓ formula F, then F is, in fact, a* semantically *read-$(\ell-1)$ formula. That is, F computes a read-$(\ell-1)$ polynomial* [9]*. Note, however, that the converse is not necessarily true. For example, the formula $F = x_1 + x_1$ is a read-twice formula and F computes the semantically read-once polynomial $2x_1$. Yet, $\mu_2(F) = 1$.*

4.2 Computation of μ_ℓ for $\ell \geq 2$

Given a multilinear formula $F \in \mathcal{C}_\ell$, we shall compute $\mu_\ell(F)$, for some $\ell \geq 2$ using the generic framework of Section 3, by defining an appropriate update function Ψ. Formally, we define $\Psi^\ell = (\Psi^\ell_{\text{leaf}}, \Psi^\ell_+, \Psi^\ell_\times)$ as follows:

$\Psi^\ell_{\text{leaf}}(g) \stackrel{\Delta}{=} \langle \gamma = 0 \rangle$. [10]

We define Ψ^ℓ_+ for a gate g of the form $g_a \leftarrow g_b + g_c$, as follows:

$g_a \leftarrow g_b + g_c$	$\gamma(g_c) = 0$	$\gamma(g_c) = 1$
$\gamma(g_b) = 0$	$\langle \gamma(g_a) = 1 \rangle$ iff $\exists i \in [n]$ s.t. x_i appears ℓ times in g_a, it also appears in both g_b, g_c, and $\frac{\partial g_b}{\partial x_i} + \frac{\partial g_c}{\partial x_i} \not\equiv 0$.	$\langle \gamma(g_a) = 1 \rangle$
$\gamma(g_b) = 1$	$\langle \gamma(g_a) = 1 \rangle$	$\langle \gamma(g_a) = 1 \rangle$

Table 5: Definition of Ψ^ℓ_+ to compute $\mu_\ell(g)$

We define $\Psi^\ell_\times$ for a gate g of the form $g_a \leftarrow g_b \times g_c$, as follows:

$g_a \leftarrow g_b \times g_c$	$\gamma(g_c) = 0$	$\gamma(g_c) = 1$
$\gamma(g_b) = 0$	$\gamma(g_a) = 0$	$\gamma(g_a) = 1$ iff $g_b \not\equiv 0$
$\gamma(g_b) = 1$	$\gamma(g_a) = 1$ iff $g_c \not\equiv 0$	$\gamma(g_a) = 1$

Table 6: Definition of $\Psi^\ell_\times$ to compute $\mu_\ell(g)$

[9] In other words, there is a read-$(\ell-1)$ formula that also computes the same polynomial. In Section 3.2, we show how to efficiently convert such a formula into an equivalent read-$(\ell-1)$ formula. This fact will be useful in our PIT reduction.

[10] For the sake of simplicity, we do not mention the OUT value in the definition of Ψ since this function returns a mere bit, and therefore is not required to use the output tape.

Now, we prove that with the above update functions, we actually compute μ_ℓ, i.e. the (γ) value $\Psi^\ell(F)$ is the same as $\mu_\ell(F)$.

Lemma 4.3. *Given a multilinear formula $F \in \mathcal{C}_\ell$, $\Psi^\ell(F) = \mu_\ell(F)$.*

Proof. Observe that any subformula g of F is also in the class $\mathcal{C}_\ell$, therefore we can also define $\mu_\ell(g)$. We show by that the invariant $\Psi^\ell(g) = \mu_\ell(g)$ holds by induction on the structure of g.

1. If g is a leaf, then $\Psi^\ell(g) = 0 = \mu_\ell(g)$ by the definitions of $\Psi^\ell(g)$ and μ_ℓ, since a variable appears only once in a leaf, whereas $\ell \geq 2$.

2. Suppose g is of the form $g_a = g_b + g_c$. Assume $\mu_\ell(g_b) = 1$. Then by the induction hypothesis, $\gamma(g_b) = 1$. Also, it follows from Definition 4.1 that there exists an $i \in \text{var}(g_b)$ such that x_i appears ℓ times in g_b. As F (and hence g_a) is a read-ℓ formula, x_i does not appear in g_c. Thus, irrespective of whether $\mu_\ell(g_c) = 0$ or 1, g_a depends on x_i and x_i appears ℓ times in g_a. Hence, $\mu_\ell(g_a) = 1$ which is exactly $\gamma(g_a)$ from Table 5. Similarly, if $\mu_\ell(g_c) = 1$, we get $\mu_\ell(g_a) = 1 = \gamma(g_a)$ as desired.

   Now, suppose that $\mu_\ell(g_b) = \mu_\ell(g_c) = 0$. By the induction hypothesis $\gamma(g_b) = \gamma(g_c) = 0$. We first assume that $\mu_\ell(g_a) = 1$. Then, it follows from Definition 4.1 that there exists a variable x_i such that x_i appears ℓ times in g_a and $\frac{\partial g_a}{\partial x_i} \not\equiv 0$. We claim that x_i should appear in both g_b and g_c. Suppose not and without loss of generality assume that x_i does not appear in g_c. This means x_i appears in g_b exactly ℓ times. Since $\mu_\ell(g_b) = 0$, it follows from Definition 4.1 that $\frac{\partial g_b}{\partial x_i} \equiv 0$. Since $g_a = g_b + g_c$, the linearity of partial derivatives implies that

$$\frac{\partial g_a}{\partial x_i} = \frac{\partial g_b}{\partial x_i} + \frac{\partial g_c}{\partial x_i}.$$

   Thus, we get $\frac{\partial g_a}{\partial x_i} \equiv 0$, which is a contradiction. Hence, x_i appears in both g_b and g_c. We get from this discussion that $\mu_\ell(g_a) = 1$ implies there exists an $i \in [n]$, which appears in both g_b, g_c, appears ℓ times in F, and $\frac{\partial g_a}{\partial x_i} \not\equiv 0$.

   For the converse, suppose there exists an $i \in [n]$, which appears in both g_b, g_c, and, in addition, appears ℓ times in g_a, and $\frac{\partial g_a}{\partial x_i} \not\equiv 0$. Then, Definition 4.1 implies that $\mu_\ell(g_a) \not\equiv 0$. Thus $\gamma(g_a) = \mu_\ell(g_a)$ holds true in this case also.

3. Suppose $g_a = g_b \times g_c$. Since g_a is a syntactic multilinear formula, the sub-formulas g_b and g_c compute variable-disjoint polynomials. First, assume $\mu_\ell(g_b) = \mu_\ell(g_c) = 0$. Then by Definition 4.1, $\mu_\ell(g_a) = 0$. Note that by the induction hypothesis, we had $\gamma(g_b) = \gamma(g_c) = 0$. Therefore by Table 6, $\gamma(g_a) = 0$, which is the same as $\mu_\ell(g_a)$ in this case. Now, assume $\mu_\ell(g_b) = \mu_\ell(g_c) = 1$. Then, it follows from Definition 4.1 that $g_b \not\equiv 0, g_c \not\equiv 0$, and there exists an $i \in [n]$ such that x_i appears ℓ times in g_b and $\frac{\partial g_b}{\partial x_i} \not\equiv 0$. As g_b and g_c compute variable-disjoint polynomials, and $g_a = g_b \times g_c$,

$$\frac{\partial g_a}{\partial x_i} = \frac{\partial g_b}{\partial x_i} \times g_c. \tag{3}$$

   Since $g_c \not\equiv 0$ and $\frac{\partial g_b}{\partial x_i} \not\equiv 0$, we get that $\frac{\partial g_a}{\partial x_i} \not\equiv 0$. This implies that $x_i \in \text{var}(g_a)$ and it appears ℓ times in g_a. Hence, from Definition 4.1, $\mu_\ell(g_a) = 1$. By the induction hypothesis, we had $\gamma(g_b) = \gamma(g_c) = 1$. From Table 6, we get that $\gamma(g_a) = 1$, which is thus the same as $\mu_\ell(g_a)$ in

this case also. Finally, suppose $\mu_\ell(g_b) = 1$ and $\mu_\ell(g_c) = 0$. Then, there exists an $i \in [n]$ such that x_i appears ℓ times in g_b, which implies that x_i appears in g_a also exactly ℓ times. Since $g_b \not\equiv 0$, it follows from Equation (3) that $\frac{\partial g_a}{\partial x_i} \equiv 0$ if and only if $g_c \equiv 0$. Thus, $\mu_\ell(g_a) = 1$ if and only if $g_c \not\equiv 0$. Also by Table 6, $\gamma(g_a) = 1$ if and only if $g_c \not\equiv 0$. Hence, $\mu_\ell(g_a) = \gamma(g_a)$ here also. The case when $\mu_\ell(g_b) = 0$ and $\mu_\ell(g_c) = 1$ follows similarly.

This completes the proof of Lemma 4.3. □

Recall Algorithm A-SUBSTITUTE from the proof of Lemma 3.8. In Remark 4.2, we noted that a read-ℓ formula F with $\mu_\ell(F) = 0$ is actually a semantically read-$(\ell - 1)$ formula. We now show that Algorithm A-SUBSTITUTE realizes an equivalent read-$(\ell - 1)$ formula for F. We use this *cleaning step* multiple times in the next section.

Claim 4.4. *Let $\ell \in \mathbb{N}$ and let F be a read-ℓ formula such that $\mu_\ell(F) = 0$. Then, given ℓ and F, Algorithm* A-SUBSTITUTE *outputs a read-$(\ell - 1)$ formula $\tilde{F}$ such that $\tilde{F} \equiv F$.*

Proof. Let $I = \{i \mid x_i \text{ appears } \ell\text{-times in } F\}$. Observe that since $\mu_\ell(F) = 0$, then by definition, the polynomial f computed by the formula F does not depend on any variable in the set I. Therefore, $f|_{\mathbf{x}_I = \mathbf{0}} = f$. But $f|_{\mathbf{x}_I = \mathbf{0}}$ is exactly the polynomial computed by $\tilde{F}$. Note that $\tilde{F}$ is a read-$(\ell - 1)$ formula since after running Algorithm A-SUBSTITUTE, no variable appears ℓ or more times in $\tilde{F}$. □

4.3 Implementation of μ_ℓ

Let $\ell \in \mathbb{N}$ and $F \in \mathcal{C}_\ell$. Then, for every node v in F, $\mu_\ell(v) \stackrel{\Delta}{=} \mu_\ell(F_v)$, where F_v is the sub-formula of F rooted at v. We will use all the algorithms we have designed in previous sections and a PIT algorithm in implementation of μ_ℓ. We summarize their descriptions below:

- Given a formula F and a variable x_i, the procedure A-COUNT counts the appearances of x_i in F (modified Algorithm 5).
- Given a formula F and $\ell \in \mathbb{N}$, the procedure A-SUBSTITUTE substitutes 0 into all the variables appearing at least ℓ-times in F (Lemma 3.8).
- Given a multilinear formula F and a variable x_i, the procedure A-PARTIAL computes $\frac{\partial F}{\partial x_i}$ (Lemma 3.6).
- Given $\ell \in \mathbb{N}$, we use A-PIT-SUM to denote the PIT algorithm for sum of two multilinear read-$(\ell - 1)$ formulas.

Note that except Algorithm A-PIT-SUM, we already know that every algorithm above uses at most $O(\log s)$ work-space.

4.3.1 Implementation of Ψ^ℓ_+

Observe that in Table 5, we need to invoke various subroutines to determine value of $\gamma(g_a)$ when $\gamma(g_b) = \gamma(g_c) = 0$. Note that both the subformulas $g_b, g_c \in \mathcal{C}_\ell$ as $F \in \mathcal{C}_\ell$.

- We iterate over $i \in [n]$ till we find a variable x_i which appears ℓ times in g_a and also appears in both g_b and g_c. We check this by invoking the subroutine A-COUNT multiple times appropriately and reusing the space for different runs. If we don't find such a variable, we set $\gamma(g_a) = 0$, otherwise we have a variable x_i which appears ℓ times in g_a but at most $(\ell - 1)$ times in either of the subformulas.

- In the latter case, we clean the subformulas g_b, g_c by invoking algorithm A-SUBSTITUTE to get g_b', g_c' respectively. Then by Claim 4.4, both g_b', g_c' are syntactically read-$(\ell - 1)$ formulas that compute the same polynomials, respectively. Note that they are read-$(\ell - 1)$ in every variable, not just x_i, and it was okay to substitute 0 in any variable appearing ℓ times in these subformulas, because their μ_ℓ values were 0 and hence they did not depend on such variables in the first place.

- We now invoke Algorithm A-PARTIAL to compute $\frac{\partial g_a'}{\partial x_i}$, where $g_a' = g_b' + g_c'$. By the sum rule, $\frac{\partial g_a'}{\partial x_i} = \frac{\partial g_b'}{\partial x_i} + \frac{\partial g_c'}{\partial x_i}$.

- Finally, we invoke the PIT algorithm A-PIT-SUM for $\frac{\partial g_a'}{\partial x_i}$, which as observed above, belongs to the required class $\sum^{[2]} \mathcal{C}_{\ell-1}$. We set $\gamma(g_a) = 0$ if and only if the PIT algorithm returns 0.

4.3.2 Implementation of $\Psi^\ell_\times$

Observe that in Table 6, we need to check whether the polynomial $g_c \not\equiv 0$ in the case when $\gamma(g_b) = 1$ and $\gamma(g_c) = 0$. Similarly in the opposite case when $\gamma(g_c) = 1$ and $\gamma(g_b) = 0$, we need to check whether $g_b \not\equiv 0$. We discuss the former case here. The other follows similarly.

- Note that the subformula $g_c \in \mathcal{C}_\ell$. We first clean g_c by invoking the subroutine A-SUBSTITUTE to get g_c'. Since $\gamma(g_c) = 0$ in this case, Claim 4.4 implies that g_c' is a read-$(\ell - 1)$ formula that computes the same polynomial as g_c.

- Although, we only need PIT for a single read-$(\ell - 1)$ formula now but we shall nevertheless use PIT for the more general class of sum of two read-$(\ell - 1)$ formulas, as the implementation of Ψ^ℓ_+ already requires it. Thus, we invoke Algorithm A-PIT-SUM for g_c' to test whether $g_c \equiv 0$. We set $\gamma(g_a) = 0$ if and only if the PIT algorithm returns 0.

4.3.3 Algorithm for μ_ℓ

Putting the implementation of $\Psi^\ell_+, \Psi^\ell_\times$ together, we get the implementation for $\Psi^\ell = (\Psi^\ell_{\text{leaf}}, \Psi^\ell_+, \Psi^\ell_\times)$. The algorithm to compute μ_ℓ is then, simply plugging Ψ^ℓ into Algorithm 1.

Lemma 4.5 (Algorithm A-COMPUTE-MU)**.** *Let $\ell \in \mathbb{N}$. Let* A-PIT-SUM *be the whitebox PIT algorithm for the class $\sum^{[2]} \mathcal{C}_{\ell-1}$ and let $S(\ell-1)$ denote its space-complexity. Then, given a formula $F \in \mathcal{C}_\ell$ of size s, there exists an algorithm to compute $\mu_\ell(F)$ which uses at most $O(\log s + S(\ell - 1))$ work-space.*

Proof. To compute $\mu_\ell(F)$, we plug in $\Psi = (\Psi^\ell_{\text{leaf}}, \Psi^\ell_+, \Psi^\ell_\times)$ in Algorithm 1.

Correctness: Follows from Lemma 4.3, Claim 4.4 and the discussion in Section 4.3.1, Section 4.3.2.

Space-Complexity: For the implementation of Ψ^{ℓ}_{+} (Section 4.3.1), in the worst case, we compose Algorithms A-COUNT, A-SUBSTITUTE, A-PARTIAL, A-PIT-SUM for each $i \in [n]$, to check the condition in the first entry of Table 5. Note that we can reuse space when iterating over $i \in [n]$; we only need extra $O(\log n)$ space needed to keep track of the current variable. Algorithm A-COUNT uses only $O(\log s)$ space (see for e.g. Algorithm 5). Algorithm A-SUBSTITUTE also uses $O(\log s)$ space by Lemma 3.8 Algorithm A-PARTIAL uses $O(\log s)$ space by Lemma 3.6. Finally, Algorithm A-PIT-SUM uses $S(\ell-1)$ space by hypothesis. For implementation of $\Psi^{\ell}_{\times}$ (Section 4.3.2), we only need to compose Algorithms A-SUBSTITUTE and A-PIT-SUM to compute the return value in the non-diagonal entries of Table 6.

We can compose these constantly-many logspace procedures and therefore by Fact 1.1, $S_{\Psi} = O(\log s + S(\ell-1) + \log n) = O(\log s + S(\ell-1))$. Since the return value γ is a single bit, $t = 1$. Therefore by Lemma 3.1, we can compute $\mu_{\ell}(F)$ using Algorithm 1 in $O(\log s + S(\ell-1))$ work-space. □

4.4 The logspace reduction

Having developed all the machinery in this work, we are now ready to present the logspace reduction of whitebox PIT for $\mathcal{C}_{k+1}$ to whitebox PIT for $\sum^{[2]}\mathcal{C}_k$, for some integer $k \geq 0$. Given k as input, we use the whitebox PIT algorithm A-PIT-SUM for the class $\sum^{[2]}\mathcal{C}_k$ as a sub-routine, whose space complexity is denoted by $S(k)$. The following theorem is a corollary of Lemma 4.5.

Theorem 4.6 (Whitebox PIT for $\mathcal{C}_{k+1} \leq_L$ Whitebox PIT for $\sum^{[2]}\mathcal{C}_k$)**.** *Let k be a constant, F be a size-s read-$(k+1)$ arithmetic formula. Let* A-PIT-SUM *be a deterministic whitebox PIT algorithm for the class $\sum^{[2]}\mathcal{C}_k$ and $S(k)$ denote its space-complexity. Then, there exists a deterministic whitebox algorithm that takes input the formula F and decides whether F is identically zero in space $O(\log s + S(k))$. Algorithm 2 provides the outline.*

Proof. Correctness: If $\mu_{k+1}(F) = 1$, then by definition of μ_{k+1}, F depends on a variable and hence is non-zero. If, however $\mu_{k+1}(F) = 0$, we know that F which is a syntactically read-$(k+1)$ computes a semantically read-k polynomial. Then Claim 4.4 allows us to clean F into a syntactically read-k formula by using Algorithm A-SUBSTITUTE with $\ell = k+1$. Therefore, we can invoke PIT algorithm for the class $\mathcal{C}_k$. Note that since we are already using Algorithm A-PIT-SUM in computation of μ_{k+1}, we can invoke it for PIT of this syntactic read-k formula, as $\mathcal{C}_k \subseteq \sum^{[2]}\mathcal{C}_k$.

Space-complexity: Computing μ_{k+1} using Algorithm A-COMPUTE-MU with $\ell = k+1$ in Line 1 takes $O(\log s + S(k))$ space by Lemma 4.5. Invoking subroutine A-SUBSTITUTE in Line 5 for $\ell = k+1$ takes $O(\log s)$ space by Lemma 3.8. Running the PIT algorithm A-PIT-SUM in Line 6 takes $S(k)$ space by the hypothesis. Thus, overall the algorithm takes $O(\log s + S(k))$ space. □

Algorithm 2: Whitebox PIT for $\mathcal{C}_{k+1} \leq_L$ Whitebox PIT for $\sum^{[2]} \mathcal{C}_k$

Input: A multilinear read-$(k+1)$ arithmetic formula F and a whitebox PIT algorithm A-PIT-SUM for the class $\sum^{[2]} \mathcal{C}_k$

Output: If $F \not\equiv 0$ then 1, otherwise 0.

Invoke Algorithm A-COMPUTE-MU in Lemma 4.5 with $\ell = k+1$ to compute $\mu_{k+1}(F)$.
if $(\mu_{k+1}(F)) = 1$ **then**
  output 1.
else
  Invoke Algorithm A-SUBSTITUTE on F with $\ell = k+1$ to get a read-k formula F'.
  Invoke Algorithm A-PIT-SUM on F'. If $F' \equiv 0$ then output 0 else output 1.
end

4.5 Application: logspace PIT for Read-twice Formulas (Proof of Theorem 4)

In this section we prove Theorem 4. To this end, we give a more formal and technical version of the theorem.

Theorem 4.7 (PIT for read-twice)**.** *Let $n \in \mathbb{N}$, $\mathbb{F}$ be a field such that $|\mathbb{F}| > n$, and F be a multilinear read-twice formula of size s over $\mathbb{F}$ such that the bit-complexity of the constants appearing in F is b. Then, there exists a deterministic algorithm that given formula F, decides whether F computes an identically zero polynomial, in space $O(\log s + e_{\mathbb{F}}(n^3+s, b+\log n))$ (see Definition 2.9).*

Remark 4.8. *A more careful description of μ_2 also leads to a logspace whitebox PIT algorithm for non-multilinear read-2 arithmetic formulas. As this paper only deals with multilinear formulas, we omit the details of this PIT algorithm.*

Theorem 4.6 reduces PIT for $\mathcal{C}_2$, the class of multilinear read-twice formulas to PIT for the class $\sum^{[2]} \mathcal{C}_1$. We use the whitebox logspace algorithm of Theorem 4.9 below for the class $\sum^{[2]} \mathcal{C}_1$ $(m = 2)$ as Algorithm A-PIT-SUM in Theorem 4.6. We note that the results of [SV15] and [MV18] together show that the generator $G_{n,3m}$ hits the class $\sum^{[m]} \mathcal{C}_1$ (sum of m-many ROFs). Then by Theorem 2.14 and Claim 2.20, we get the following result. We advise the reader to recall Definition 2.9 and Lemma 2.10 for the following theorem.

Theorem 4.9 (Logspace PIT for sum of read-once)**.** *Let $n \in \mathbb{N}$, $\mathbb{F}$ be a field such that $|\mathbb{F}| > n$, F be a multilinear $\sum^{[m]} \mathcal{C}_1$ formula of size $s \geq n$ over $\mathbb{F}$ where the bit-complexity of the field constants appearing in F is b. Then there exists a deterministic algorithm that given whitebox access to F, decides whether F computes an identically zero polynomial, in space $O(m \cdot \log s + e_{\mathbb{F}}(n^3+s, b+\log n))$.*

As a consequence of the logspace PIT reduction in Theorem 4.6 and the logspace PIT for sum of two read-once formulas, we get logspace whitebox PIT algorithm for the class of read-twice formulas.

Proof of Theorem 4.7. Invoke Algorithm 2 with $k = 1$ and the algorithm of Theorem 4.9 with $m = 2$ as Algorithm A-PIT-SUM. The correctness and space-complexity follow from Theorem 4.6, as $S(1) = O(\log s + e_{\mathbb{F}}(n^3+s, b+\log n))$ where $S(1)$ is the space-complexity of PIT for the class $\sum^{[2]} \mathcal{C}_1$.

4.6 μ_1 and whitebox PIT for read-once formulas (without the need of generator)

In Section 4.2, we showed how to compute μ_ℓ where $\ell \geq 2$. Here, we show how to compute $\mu_1(F)$ for a multilinear read-once formula F. It is along the same lines as computation of μ_ℓ except few minor changes which we highlight below.

The main change is in the definition of Ψ_{leaf}. Here we define it as

$$\Psi^1_{\text{leaf}}(g) \triangleq \begin{cases} \langle \gamma = 1 \rangle, & \text{if } g \text{ is a variable} \\ \langle \gamma = 0 \rangle, & \text{otherwise.} \end{cases}$$

The definition of Ψ^1_+ and $\Psi^1_\times$ is the same as Ψ^ℓ_+ and $\Psi^\ell_\times$ respectively, when $\ell = 1$. We note here that the first entry of Ψ^1_+ is always $\langle \gamma = 0 \rangle$ as the iff condition in Table 5 will be vacuously false, since every $+$ gate in a read-once formula computes sum of two variable-disjoint subformulas. Formally, for a gate g of the form $g_a \leftarrow g_b + g_c$, if $\gamma(g_b) = \gamma(g_c) = 0$ then $\gamma(g_a) = 0$. Thus Table 5 in the case of Ψ^1_+ does not require invoking any algorithm in its implementation.

Table 6 is exactly the same for $\Psi^1_\times$. In its implementation, when $\gamma(g_b) = 0$ and $\gamma(g_c) = 1$, we require testing whether $g_b \equiv 0$. Since $\gamma(g_b) = 0$, we can first clean the subformula g_b by invoking the subroutine A-SUBSTITUTE, with $\ell = 1$. By Claim 4.4, we get a read-zero subformula g'_b computing the same polynomial. Thus we can simply compute the output of g'_b using the logspace evaluation procedure in Lemma 2.10 and test whether it computes 0 or not.

Putting things together, we can compute $\Psi^1(F)$ in $O(e_{\mathbb{F}}(s, b) + \log s)$ space. For PIT of input read-once formula F, we then invoke Algorithm 2 with $k = 0$ to reduce it to PIT for the class $\sum^{[2]} \mathcal{C}_0$, where $\mathcal{C}_0$ is simply the class of read-zero formulas (a formula with field constants in all the leaves). We use A-PIT-SUM with $\ell = 1$ for PIT of the class $\sum^{[2]} \mathcal{C}_0$. Thus we get an $O(e_{\mathbb{F}}(s, b) + \log s)$-space deterministic algorithm whitebox PIT algorithm from Theorem 4.6, by observing that $S(0) = O(\log s + e_{\mathbb{F}}(s, b))$. This is a direct whitebox logspace algorithm for read-once formulas in contrast to the logspace PIT algorithm in Theorem 4.9 with $m = 1$ that uses blackbox PIT: the logspace-explicit blackbox generator G_1. We summarize this formally below.

Theorem 4.10 (Read-once PIT without use of generator)**.** *Let F be a multilinear read-once formula of size s over any field $\mathbb{F}$ where the bit-complexity of the field constants appearing in F is b. Then there exists a deterministic algorithm that given whitebox access to F, decides whether F computes the identically zero polynomial, in space $O(e_{\mathbb{F}}(s, b) + \log s)$, without requiring the use of generator $G_{n,1}$.*

5 Logspace PIT for $\mathcal{C}_1 + \mathcal{C}_2$

So far, we have seen that there are two logspace whitebox PIT algorithms for $\mathcal{C}_1$, the class of arithmetic read-once formulas: One using a logspace-explicit generator (Definition 2.11) for $\mathcal{C}_1$ (see Theorem 4.9), and other using Theorem 4.6, which gives a logspace reduction from whitebox PIT for $\mathcal{C}_{k+1}$ to whitebox PIT for $\sum^{[2]} \mathcal{C}_k$ (see Theorem 4.10). Since we have a logspace whitebox PIT algorithm for $\sum^{[2]} \mathcal{C}_1$ (see Theorem 4.9), following the second route, we also give a logspace whitebox PIT algorithm for $\mathcal{C}_2$ (see Theorem 4.7). In this section, we take a step forward and give a whitebox logspace PIT for the class $\mathcal{C}_1 + \mathcal{C}_2$, which consists of formulas of the kind $R + Q$, where $R \in \mathcal{C}_1$ and $Q \in \mathcal{C}_2$. One might ask what is the need to take such 'baby steps' and why do not we directly attack the problem of obtaining a logspace whitebox PIT algorithm for the class $\mathcal{C}_k$ for

every constant $k \geq 3$. A polynomial-time whitebox PIT for $\mathcal{C}_k$ was given in [AvMV15], whenever $k \in \mathbb{N}$ is a constant. One important reason for our inability to give logspace whitebox PIT for $\mathcal{C}_k$ is the hitting-set generator for $\mathcal{C}_k$ given in [AvMV15] has seed-length $O(\log n)$ and Theorem 2.14 implies that we get *roughly* a $O(\log^2 n)$-space whitebox PIT algorithm for $\mathcal{C}_k$. In this scenario, the only ray of hope emerges from the second route that goes via the logspace reduction given in Theorem 4.6. To exploit the potential of this route, and to obtain a logspace whitebox PIT for $\mathcal{C}_k$ for constant $k \geq 3$, we at least require logspace whitebox PIT for the class $\mathcal{C}_2 + \mathcal{C}_2$. Our result given in this section is a step forward in this direction. We first prove a more general result in Theorem 5.1, and then instantiate this result to prove Theorem 5.12.

Theorem 5.1 (PIT for $\mathcal{C}_k + \mathcal{C}_{k+1}$)**.** *Let $n \in \mathbb{N}$, $\mathbb{F}$ be a field such that $|\mathbb{F}| > n$, $k \in \mathbb{N}$ be a constant, $R \in \mathcal{C}_k, Q \in \mathcal{C}_{k+1}$, and $F \triangleq R + Q$ be a size s formula that computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$ such that the bit-complexity of the field constants appearing in R and S is b. Suppose we have an $O(S_k)$-space-explicit generator $H = (H_i)_{i \in \mathbb{N}}$ for $\mathcal{C}_k$, where $H_n : \mathbb{F}^{t_n} \to \mathbb{F}^n$ such that the individual degree of H_n is bounded by d_n, and we also have an $O(S_{k+1})$-space whitebox PIT algorithm* PIT- $\mathcal{C}_{k+1}$ *for the class $\mathcal{C}_{k+1}$. Then, given whitebox access to F, we can determine whether $F \equiv 0$ in space $O(t_n(\log s + \log d_n) + S_k + S_{k+1} + e_{\mathbb{F}}(s, b))$, where $e_{\mathbb{F}}(s, b)$ is described in Definition 2.9.*

The proof of Theorem 5.1 essentially follows the proof overview of Theorem 5 given in Section 1.4.2.

5.1 The algorithm

Algorithm 3: Whitebox PIT for $\mathcal{C}_k + \mathcal{C}_{k+1}$

Input: Whitebox access to $F = R + Q \in \mathbb{F}[x_1, x_2, \ldots, x_n]$, where $R \in \mathcal{C}_k, Q \in \mathcal{C}_{k+1}$, $u \triangleq O(k^7)$, $w \triangleq O(k^{10k})$, and Algorithm PIT- $\mathcal{C}_{k+1}$

Output: If $F \not\equiv 0$ then 1, otherwise 0.

```
if n ≤ w then
    Test whether F ≡ 0 or not. If yes, output 0, else output 1.
end
for I ⊆ [n], |I| ≤ u + 1 do
    Compute ∂_I Q using Algorithm A-PARTIAL iteratively |I| times.
    Using Algorithm PIT- C_{k+1}, check if ∂_I Q ≡ 0 or not.
    if ∂_I Q ≢ 0 then
        if (∂_I Q)(H_n) ≡ 0 then
            Output 1.
        end
    end
end
if F(H_n + G_{n,w+1}) ≢ 0 then
    Output 1
else
    Output 0.
end
```

Space Complexity. If $n \leq w$ then Step 2 can be trivially carried out in $O(e_{\mathbb{F}}(s,b) + \log s)$ space, as w is a constant. As we have whitebox access to F and u is a constant, it follows from Lemma 3.6 that for every $I \subseteq [n], |I| \leq u+1$, we can compute the $|I|$-th order partial derivative of Q in space $O(\log s)$. After that, Step 6 can be carried out in space $O(S_{k+1})$. As H is a $O(S_k)$-space-explicit generator, we do Step 8 in space $O(S_k)$. Finally, it follows from Observation 2.19, Lemma 2.13, and Theorem 2.14 that we can decide whether $F(H_n + G_{n,w+1}) \not\equiv 0$ in $O(t_n(\log s + \log d_n) + e_{\mathbb{F}}(s,b))$. Hence, the space complexity of the above algorithm is $O(t_n(\log s + \log d_n) + S_k + S_{k+1} + e_{\mathbb{F}}(s,b))$.

5.2 Some useful results

We first list some important results that would be required to argue the correctness of Algorithm 3. In this section, let $\mathbb{F}$ be an arbitrary field. Recall the definition of the measure μ_ℓ (Definition 4.1). We start with the following observation. This says that the hitting-set generator for $\mathcal{C}_k$ also hits all the sub-formulas f of Q satisfying $\mu_{k+1}(f) = 0$.

Observation 5.2. *Let $k \in \mathbb{N}$ be a constant, $Q \in \mathcal{C}_{k+1}$ be such that it computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$, and f be a sub-formula of Q such that $\mu_{k+1}(f) = 0$. Let $H = (H_i)_{i \in \mathbb{N}}$ be a hitting-set generator for $\mathcal{C}_k$. Then, $f \equiv 0$ if and only if $f(H_n) \equiv 0$.*

Proof. Since $\mu_{k+1}(f) \equiv 0$, we know from Definition 4.1 that either every variable in $\{x_1, \ldots, x_n\}$ appears at most k times in f or for every $i \in [n]$ such that x_i appears $k+1$ times in f, $\partial_{x_i} f \equiv 0$. Then, it follows from Remark 4.2 that there exists an $R \in \mathcal{C}_k$ such that f and R compute the same polynomial. Hence, H hits f, i.e., $f \equiv 0$ if and only if $f(H_n) \equiv 0$. □

The following claim plays an important role in the algorithm.

Claim 5.3. *Let $k \in \mathbb{N}$ be a constant, $R \in \mathcal{C}_k, Q \in \mathcal{C}_{k+1}$ be such that $F \triangleq Q + R$ computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$, and $H = (H_i)_{i \in \mathbb{N}}$ be a hitting-set generator for $\mathcal{C}_k$. If there exists an $I \subseteq [n]$ such that $\partial_I Q \not\equiv 0$ but $(\partial_I Q)(H_n) \equiv 0$, then $F \not\equiv 0$.*

Proof. Suppose for the sake of contradiction that $F \equiv 0$. Let $I \subseteq [n]$ be such that $\partial_I Q \not\equiv 0$ but $\partial_I Q(H_n) \equiv 0$. As $F = Q + R$, we get that

$$\partial_I R = -\partial_I Q,$$

which implies that

$$(\partial_I R)(H_n) = -(\partial_I Q)(H_n).$$

As $\partial_I Q \not\equiv 0$, the first equation implies that $\partial_I R \not\equiv 0$. Since R is a multilinear read-k formula over $\mathbb{F}$, we get from Observation 2.5 that $\partial_I R \in \mathcal{C}_k$. As H is a generator for $\mathcal{C}_k$, $(\partial_I R)(H_n) \not\equiv 0$. Thus, $(\partial_I R)(H_n) \neq -(\partial_I Q)(H_n)$ as $(\partial_I Q)(H_n) \equiv 0$, which is a contradiction. □

In the following two claims, we show that if f is a sub-formula of Q satisfying $\mu_{k+1}(f) = 1$ then for certain specific sets $J \subseteq [n], \partial_J f$ divides $\partial_J Q$. Thus, we get that whenever $\partial_J Q \not\equiv 0$ and $(\partial_J Q)(H_n) \not\equiv 0$, H_n also hits $\partial_J f$.

Claim 5.4. *Let $k \in \mathbb{N}$ be a constant, $Q \in \mathcal{C}_{k+1}$ be such that it computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$, and f be a sub-formula of Q such that $\mu_{k+1}(f) = 1$. Then, there exists an $i \in [n]$ such that $\partial_{x_i} f$ divides $\partial_{x_i} Q$.*

Proof. As Q is a multilinear read-$(k+1)$ formula over $\mathbb{F}$ and f is a sub-formula of Q, it is not difficult to see that Q can be written as

$$Q = A(x_1, \ldots, x_n, y)|_{y=f},$$

where $A \in \mathbb{F}[x_1, \ldots, x_n, y]$ is multilinear in y. As $\mu_{k+1}(f) = 1$, it follows from Definition 4.1 that there exists an $i \in [n]$ such that x_i appears $k+1$ times in f and $\partial_{x_i} f \not\equiv 0$. Then, the chain rule of partial derivatives (see Fact 2.3) implies that

$$\partial_{x_i} Q = (\partial_{x_i} A(x_1, \ldots, x_n, y))|_{y=f} + \partial_y A \cdot \partial_{x_i} f.$$

As Q is a multilinear read-$(k+1)$ formula and as x_i appears $k+1$ times in f, it follows that x_i does not appear in $A(x_1, \ldots, x_n, y)$, which implies that $\partial_{x_i} A \equiv 0$. Then, the above equation becomes

$$\partial_{x_i} Q = \partial_y A \cdot \partial_{x_i} f.$$

This implies that $\partial_{x_i} f$ divides $\partial_{x_i} Q$. □

Claim 5.5. *Let $k \in \mathbb{N}$ be a constant, $Q \in \mathcal{C}_{k+1}$ be such that it computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$, f be a sub-formula of Q such that $\mu_{k+1}(f) = 1$, and $I \subseteq [n]$ be such that $\partial_I f \not\equiv 0$. Then, there exists an $i \in [n]$ such that if $J \stackrel{\Delta}{=} I \cup \{i\}$ then $\partial_J f$ divides $\partial_J Q$.*

Proof. As argued in the proof of Claim 5.4, there exists an $A \in \mathbb{F}[x_1, \ldots, x_n, y]$ such that

$$Q = A(x_1, \ldots, x_n, y)|_{y=f}.$$

As $\mu_{k+1}(f) = 1$, it follows from Definition 4.1 that there exists an $i \in [n]$ such that x_i appears $k+1$ times in f and $\partial_{x_i} f \not\equiv 0$. Let us fix one such an x_i arbitrarily. Let $I \subseteq [n]$ be such that $\partial_I f \not\equiv 0$ and $J \stackrel{\Delta}{=} (I \setminus \{i\}) \cup \{i\}$. Then,

$$\partial_J Q = \partial_{I\setminus\{i\}}(\partial_{x_i} Q) = \partial_{I\setminus\{i\}}\Big((\partial_{x_i} A(x_1, \ldots, x_n, y))|_{y=f} + \partial_y A \cdot \partial_{x_i} f\Big).$$

As argued in the proof of Claim 5.4, $(\partial_{x_i} A(x_1, \ldots, x_n, y))|_{y=f} \equiv 0$. Then,

$$\partial_J Q = \partial_{I\setminus\{i\}}(\partial_{x_i} Q) = \partial_{I\setminus\{i\}}(\partial_y A \cdot \partial_{x_i} f).$$

Since Q computes a multilinear polynomial, so is $\partial_{x_i} Q$ is also multilinear, which implies that $\partial_y A$ and $\partial_{x_i} f$ are variable disjoint polynomials. As $\partial_{I\setminus\{i\}} f \not\equiv 0$, for every $j \in I \setminus \{i\}$, we get that $j \notin \mathrm{var}(\partial_y A)$. This along with the chain rule of partial derivatives implies that

$$\partial_J Q = \partial_y A \cdot \partial_{I\setminus\{i\}} \partial_{x_i} f = \partial_y A \cdot \partial_J f.$$

Thus, $\partial_J f$ divides $\partial_J Q$. □

The following fact is an instance of an important result of [AvMV15]. This result, which was called as the key lemma in [AvMV15], played a crucial role in obtaining the first quasi-polynomial time blackbox PIT algorithm for the class $\mathcal{C}_k$, where k is a constant.

Lemma 5.6 ('Key Lemma' of [AvMV15] [11]). *Let $\mathbb{F}$ be a field, k be a constant, $R \in \mathcal{C}_k, Q \in \mathcal{C}_{k+1}$, and $F \triangleq R+Q$ be such that it computes a non-zero polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Let $u \triangleq O(k^7)$ and $w \triangleq O(k^{10k})$. Suppose $(a_1, \ldots, a_n) \in \mathbb{F}^n$ is a common non-zero assignment of the non-zero formulas of the kind $\partial_I f$, where f is a sub-formula of either R or Q, and $I \subseteq [n], |I| \leq u$. Then, for every $n > w$, the monomial $x_1 \cdots x_n$ does not divide $F(x_1 + a_1, \ldots, x_n + a_n)$.* [12]

The following claim *subsumes* Lemma 5.6.

Claim 5.7. *Let $\mathbb{F}$ be a field, k be a constant, $R \in \mathcal{C}_k, Q \in \mathcal{C}_{k+1}$, and $F \triangleq R+Q$ be such that it computes a non-zero polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Let $u \triangleq O(k^7)$, $w \triangleq O(k^{10k})$, and $(a_1, \ldots, a_n) \in \mathbb{F}^n$ be a common non-zero of all the non-zero formulas of the kind $\partial_I f$, where the set $I \subseteq [n]$ and the sub-formula f of F belong to one of the following categories.*

1. *f is a sub-formula of R and $|I| \leq u + 1$.*
2. *f is a sub-formula of Q such that $\mu_{k+1}(f) = 0$ and $|I| \leq u$.*
3. *f is a sub-formula of Q such that $\mu_{k+1}(f) = 1$ and $|I| \leq u$ such that for every $i \in [n], \partial_{I \cup \{i\}} Q \not\equiv 0$.*
4. *f is a sub-formula of Q such that $\mu_{k+1}(f) = 1$ and $|I| \leq u$ such that there exists an $i \in [n]$, which satisfies $\partial_{I \cup \{i\}} Q \equiv 0$.*

Then, for every $n > w$, the monomial $x_1 \cdots x_n$ does not divide $F(x_1 + a_1, \ldots, x_n + a_n)$.

Proof. Let $E = \{(f, I) \mid f$ and I satisfy Points 1 to 4 and $\partial_I f \not\equiv 0\}$. Then, observe that for any $I \subseteq [n], |I| \leq u$ and a sub-formula f of Q or R such that $\partial_I f \not\equiv 0$, we have that $(f, I) \in E$. Let $n > w$. Then, as $(a_1, \ldots, a_n)$ is a common non-zero assignment of $\partial_I f$ for every $(f, I) \in E$, it follows from Lemma 5.6 that $x_1 \cdots x_n$ does not divide $F(x_1 + a_1, \ldots, x_n + a_n)$. □

This claim forms the base of the proof of Theorem 5.1. Before coming to the algorithm, we show that it is sufficient to consider a common non-zero assignment of non-zero formulas of the kind $\partial_I f$ in Claim 5.7, where sub-formula f and set I belong to the categories given in Points 1 to 3 of the claim.

Claim 5.8. *Let $\mathbb{F}$ be a field, k be a constant, $R \in \mathcal{C}_k, Q \in \mathcal{C}_{k+1}$, and $F \triangleq R+Q$ be such that it computes a non-zero polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Let $u \triangleq O(k^7)$, $w \triangleq O(k^{10k})$, and $(a_1, \ldots, a_n) \in \mathbb{F}^n$ be a common non-zero assignment of the non-zero formulas of the kind $\partial_I f$, where f and I belong to one of the categories mentioned in Points 1 to 3 of Claim 5.7. Then, for every $n > w$, the monomial $x_1 \cdots x_n$ does not divide $F(x_1 + a_1, \ldots, x_n + a_n)$.*

Proof. Let $n > w$. We are given that $\mathbf{a} \triangleq (a_1, \ldots, a_n) \in \mathbb{F}^n$ is a common non-zero assignment of the non-zero formulas of the kind $\partial_I f$, where f and I belong to one of the categories mentioned in Points 1 to 3 of Claim 5.7. Now, there are two cases: Either $\mathbf{a}$ is also a non-zero of all non-zero formulas $\partial_I f$, where f and I belong to the category given in Point 4 of Claim 5.7 or there exists a

[11] The lemma quoted here is taken from the conference version [AvMV11] of the work [AvMV15].

[12] This approach was called as 'hardness of representation' in [SV15] and has been instrumental in obtaining efficient deterministic PIT algorithms for some interesting classes of arithmetic formulas. See [SV15, AvMV15] in this regard.

sub-formula f' of Q such that $\mu_{k+1}(f') = 1$, there exist an $I' \subseteq [n], |I'| \leq u$ and an $i' \in [n]$ such that $\partial_{I'} f' \not\equiv 0$, $\partial_{I' \cup \{i'\}} Q \equiv 0$ but $(\partial_{I'} f')(\mathbf{a}) = 0$. There is nothing to prove in the first case because of Claim 5.7. Let us consider the second case and arbitrarily pick one (f', I', i') mentioned above. We assume for the contradiction that the claim does not hold in this case. Then, as F is multilinear, there exists a non-zero $\alpha \in \mathbb{F}$ such that

$$F(\mathbf{x} + \mathbf{a}) = Q(\mathbf{x} + \mathbf{a}) + R(\mathbf{x} + \mathbf{a}) = \alpha \cdot x_1 \cdots x_n.$$

This implies

$$\partial_{I' \cup \{i'\}} Q(\mathbf{x} + \mathbf{a}) + \partial_{I' \cup \{i'\}} R(\mathbf{x} + \mathbf{a}) = \alpha \prod_{i \in [n] \setminus (I' \cup \{i'\})} x_i.$$

As $\partial_{I' \cup \{i'\}} Q \equiv 0$ by assumption, the above equation can be rewritten as

$$\partial_{I' \cup \{i'\}} R(\mathbf{x} + \mathbf{a}) = \alpha \prod_{i \in [n] \setminus (I' \cup \{i'\})} x_i.$$

As $\alpha \neq 0$, we get that $\partial_{I' \cup \{i'\}} R(\mathbf{x} + \mathbf{a}) \not\equiv 0$, which implies that $\partial_{I' \cup \{i'\}} R(\mathbf{x}) \not\equiv 0$. As $(R, I' \cup \{i'\})$ belongs to the category mentioned in Point 1 of Claim 5.7; and as $\mathbf{a}$ is a common non-zero of $\partial_I f$, where (f, I) belong to the categories mentioned in Points 1 to 3 of Claim 5.7 we get that $\partial_{I' \cup \{i'\}} R(a_1, \ldots, a_n) \neq 0$. On the other hand, as $n > w > u$, on substituting $x_i = 0$ in the above equation for every $i \in [n]$, we get that $\partial_{I' \cup \{i'\}} R(a_1, \ldots, a_n) = 0$. This is a contradiction. Hence, the claim holds in second case as well. □

Consider the following useful result from [SV15].

Fact 5.9 ([SV15])**.** *Let $w, n \in \mathbb{N}, n > w$, $F \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ be a non-zero polynomial, and $\mathbf{a} \in \mathrm{Im}(G_{n,w})$ (see Definition 2.15) such that $F(\mathbf{x} + \mathbf{a})$ is not divisible by $x_1 \cdots x_n$. Then, $F(G_{n,w}) \not\equiv 0$.*

5.3 Correctness of Algorithm 3

The following lemma establishes the correctness of Algorithm 3.

Lemma 5.10. *Algorithm 3 correctly decides whether $F \equiv 0$ or not.*

Proof. If $n \leq w$ then F becomes a constant-variate polynomial, in which case we can trivially check whether $F \equiv 0$ or not in space $O(e_{\mathbb{F}}(s, b) + \log s)$. Now, we assume that $n > w$. Suppose $F \equiv 0$. Then, $F(H_n + G_{n,w+1})$ is also the zero polynomial. In this case, Claim 5.3 ensures that Step 13 is never executed. Hence, the algorithm always outputs 0. Now, suppose $F \not\equiv 0$. If the Algorithm halts in Step 9 then there exists an $I \subseteq [n], |I| \leq u + 1$ such that $\partial_I Q \not\equiv 0$ but $(\partial_I Q)(H_n) \equiv 0$. Then, it follows from Claim 5.3 that $F \not\equiv 0$ and hence the algorithm's output is correct. Thus, it is ensured that if the algorithm does not halt in Step 9, then for every $I \subseteq [n], |I| \leq u + 1$ satisfying $\partial_I Q \not\equiv 0$, we have $(\partial_I Q)(H_n) \not\equiv 0$. Now, we argue that $F(H_n + G_{n,w+1}) \not\equiv 0$. In this direction, we prove the following result.

Claim 5.11. *Suppose $F \not\equiv 0$ and for every $I \subseteq [n], |I| \leq u + 1$ that satisfies $\partial_I Q \not\equiv 0$, we have $(\partial_I Q)(H) \not\equiv 0$. Then, there exists an $\mathbf{a} \in \mathrm{Im}(H_n + G_{n,1})$ such that $F(\mathbf{x} + \mathbf{a})$ is not divisible by $x_1 \cdots x_n$.*

Proof. Let E be the set of pairs of the kind (f, I), where f varies over sub-formulas of R and Q, $I \subseteq [n], |I| \leq u+1$, and $\partial_I f \not\equiv 0$ such that f and I belong to the categories mentioned in Points 1, 2, and 3 of Claim 5.7. We claim that there exists an $\mathbf{a} = (a_1, \ldots, a_n) \in \mathrm{Im}(H_n + G_{n,1})$ such that $\mathbf{a}$ is a common non-zero assignment of the polynomials $\partial_I f$ for every $(f, I) \in E$. It is given to us that H is a generator for $\mathcal{C}_k$. Since $R \in \mathcal{C}_k$ is a multilinear formula, every sub-formula of R is also multilinear, and it follows from Observation 2.17 that for every sub-formula f of R and $I \subseteq [n], |I| \leq u+1$ such that $(\partial_I f)(H_n) \not\equiv 0$. It follows from Observations 5.2 and 2.17 that for every sub-formula f of Q satisfying $\mu_{k+1}(f) = 0$ and for every $I \subseteq [n], |I| \leq u$ such that $\partial_I f \not\equiv 0$, we get that $(\partial_I f)(H) \not\equiv 0$. Now, suppose f is a sub-formula of Q such that $\mu_{k+1}(f) = 1$ and let $I \subseteq [n], |I| \leq u$ such that $\partial_I f \not\equiv 0$ and for every $i \in [n], \partial_{I\cup\{i\}} Q \not\equiv 0$. We know from Claim 5.5 that $(\partial_{I\cup\{i\}} f)(H_n)$ divides $(\partial_{I\cup\{i\}} Q)(H_n)$. As $(\partial_{I\cup\{i\}} Q)(H_n) \not\equiv 0$ and as $\mathbb{F}[x_1, x_2, \ldots, x_n]$ is an integral domain, it follows that $(\partial_{I\cup\{i\}} f)(H_n) \not\equiv 0$. Now, Lemma 2.18 implies that $(\partial_I f)(H_n + G_{n,1}) \not\equiv 0$. This shows the existence of the desired tuple $\mathbf{a} \in \mathrm{Im}(H_n + G_{n,1})$. Now, Claim 5.8 implies that $F(\mathbf{x} + \mathbf{a})$ is not divisible by the monomial $x_1 \cdots x_n$. □

As shown in the claim above, that assuming $F \not\equiv 0$ and $n > w$, $F(\mathbf{x} + \mathbf{a})$ is not divisible by $x_1 \cdots x_n$, where $\mathbf{a} \in \mathrm{Im}(H_n + G_{n,1})$. Then, it follows from Fact 5.9 that $F(G_{n,w}) \not\equiv 0$. We are assuming for every $I \subseteq [n], |I| \leq u+1$ that satisfy $\partial_I Q \not\equiv 0$, we have $(\partial_I Q)(H_n) \not\equiv 0$. Since $\mathbf{a} \in \mathrm{Im}(H_n + G_{n,1})$, it is not difficult to prove that $F \equiv 0$ if and only if $F(H_n + G_{n,1} + G_{n,w}) \equiv 0$ (see [SV15] in this regard). It follows from Fact 2.16 that $G_{n,1} + G_{n,w} = G_{n,w+1}$, which implies that $F \equiv 0$ if and only if $F(H_n + G_{n,w+1}) \equiv 0$. This completes the proof of Lemma 5.10. □

5.4 Proof of Theorem 5

We note a more formal version of Theorem 5 below.

Theorem 5.12 (PIT for $\mathcal{C}_1 + \mathcal{C}_2$)**.** *Let $n \in \mathbb{N}$, $\mathbb{F}$ be a field such that $|\mathbb{F}| > n$, $R \in \mathcal{C}_1, Q \in \mathcal{C}_2$, and $F \triangleq R + Q$ be formula of size s that computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$ such that the bit complexity of the constants appearing in R and Q is b. Then, given formula F, we can determine in $O(\log s + e_{\mathbb{F}}(n^3 + s, b + \log n))$-space whether F computes the zero polynomial or not.*

Let us instantiate Theorem 4.6 to obtain the proof of the above theorem. It was shown in [MV18] that $G_{n,1}$ hits $\mathcal{C}_1$ and we know from Claim 2.20 that $G_{n,1}$ is $O(\log n, e_{\mathbb{F}}(n^3, \log n))$-space-explicit. We showed in Theorem 4.7 that $\mathcal{C}_2$ admits a $O(\log s + e_{\mathbb{F}}(n^3 + s, b + \log n))$-space-explicit whitebox PIT algorithm. On setting $k = 1, H_n = G_{n,1}, t_n = 2, d_n = n$, $S_1 = \log n + e_{\mathbb{F}}(n^3, \log n)$, and $S_2 = \log s + e_{\mathbb{F}}(n^3 + s, b + \log n)$, and using the fact that $s \geq n$, we get a $O(\log s + e_{\mathbb{F}}(n^3 + s, b + \log n))$-space whitebox PIT algorithm for the class $\mathcal{C}_1 + \mathcal{C}_2$.

6 Logspace PIT for constant-width ROABPs

In this section, we give a logspace whitebox PIT algorithm for constant-width *read-once oblivious arithmetic branching programs* (ROABPs) by observing that the hitting set generator of [GKS17] for this class is computable in logspace. See Appendix D for formal definition of ROABPs.

Gurjar et al. show that for a non-zero bivariate polynomial $f(x_1, x_2)$ computable by a width-w ROABP, $f(x_1, x_2) \equiv 0 \Leftrightarrow f(t^w, t^w + t^{w-1}) \equiv 0$ [GKS17, Lemma 3.2]. They use this to give a 'recursive' generator construction for a general n-variate ROABP, such that you halve the number

of variables in each round and still maintain the ROABP structure. After applying this map $\log n$ times, one gets a univariate polynomial which is non-zero if and only if the input polynomial was non-zero. This generator works for a known variable order ROABP. For the sake of completeness, we state their result below. Without loss of generality, they assume that the number of variables n is a perfect power of 2.

Lemma 6.1 (Lemma 3.5 in [GKS17]). *Suppose* $\mathrm{char}(\mathbb{F}) = 0$, *or* $\mathrm{char}(\mathbb{F}) \geq ndw^{\log n}$. *Let* $f \in \mathbb{F}[\mathbf{x}]$ *be a non-zero polynomial of individual degree* d *and computed by a width-*w *ROABP in variable order* $(x_0, x_1, \ldots, x_{n-1})$. *Let* $p_0(t) = t^w$ *and* $p_1(t) = t^w + t^{w-1}$ *be two polynomials and* $\phi : \{x_0, x_1, \ldots, x_{n-1}\} \to \mathbb{F}[t]$ *be a map such that for any index* $0 \leq i \leq n-1$,

$$\phi(x_i) = p_{i_1}(p_{i_2} \cdots (p_{i_{\log n}}(t))),$$

where $i_{\log n} i_{\log n - 1} \cdots i_1$ *is the binary representation of* i.
Then $f(\phi(x))$ *is a non-zero univariate polynomial of degree* $ndw^{\log n}$.

The above lemma yields a hitting set of size $O(ndw^{\log n})$ which is $\mathrm{poly}(n, d)$ when the width w is constant. Since, there are $\log n$ applications of either $p_0(t)$ or $p_1(t)$ in Lemma 6.1, the map ϕ is logspace computable. For the sake of our exposition, we reformulate the above lemma in terms of evaluation below. Since $f(\phi(x)) \in \mathbb{F}[t]$ is a univariate polynomial, we substitute field elements from a set of size $> ndw^{\log n}$ in the t-variable of the final polynomial $f(\phi(x))$. Equivalently, given a field element $\alpha \in F$, we can compute the exact substitution to be made into each variable x_i, using the logspace procedure shown below.

Observation 6.2. *Let* $\mathbb{F}$ *be any field. Let* $w, n \in N$ *where* $n = 2^k$, *for some integer* $k \geq 1$. *Then given a variable* x_i, *for some* $0 \leq i \leq n-1$, *and a field constant* $\alpha \in \mathbb{F}$, *let* $\alpha_0 \stackrel{\Delta}{=} \alpha$ *and for* $1 \leq j \leq k$,

$$\alpha_j = \alpha_{j-1}^w + b_j \alpha_{j-1}^{w-1},$$

where $i = b_1 b_2 \cdots b_k$ *is the binary representation of* i. *Then,* α_k *is the substitution into the variable* x_i, *based on Lemma 6.1, that is computable in* $O(e_{\mathbb{F}}(w, b) + \log n)$ *space, where* b *is the bit-complexity of the field constant* α.

Proof. Note that $k = \log n$ and $\alpha_k = \phi(x_i)|_{t=\alpha}$ from Lemma 6.1. Further, observe that given α_{j-1}, there is a size-w formula to compute α_j and thus we only need $O(e_{\mathbb{F}}(w, b))$ space to compute α_j (Recall Definition 2.9 and Lemma 2.10). Since there are $k = \log n$ iterations only and we can reuse the space for each iteration, we can compute the final α_k in $O(e_{\mathbb{F}}(w, b) + \log n)$ space. □

We now discuss the second component of our logspace whitebox PIT algorithm for a constant-width ROABP. Note that since, we are in the whitebox setting we know the variable order. Suppose, without loss of generality, that the variable order is $(x_1, \ldots, x_n)$. We note below that we can evaluate a constant-width ROABP in logarithmic space.

Observation 6.3. *Let* $f(x_1, \ldots, x_n)$ *be a polynomial computed by a width-*w *ROABP of individual degree* d *over a field* $\mathbb{F}$ *such that the bit-complexity of the field constants appearing in ROABP of* f *is* b. *Then, given an assignment* $(\beta_1, \ldots, \beta_n) \in \mathbb{F}^n$, *we can compute* $f(\beta_1, \ldots, \beta_n)$ *in* $O(w \cdot e_{\mathbb{F}}(d, b) + \log n)$*-space.*

Proof. We have n layers in the ROABP. For each layer, we compute the partial evaluations (till that layer) for every node of the layer. Since the computation is sequential, we only need to remember partial computations of the previous layer to compute the values for the current layer and thus, we can reuse this space for different layers. There are at most w nodes in each layer and we need to store the values of $2w$-many registers/nodes (previous and current layer). Since, at each node we need to evaluate a univariate polynomial of degree d, we thus need $O(w \cdot e_{\mathbb{F}}(d, b))$ space, as any univariate polynomial of degree d has a formula of size $O(d)$. We need an additional $O(\log n)$ space to keep track of the current layer. □

Putting Observation 6.2 and Observation 6.3 together, we get a logspace whitebox PIT algorithm for constant-width ROABPs. We compute the correct hitting set of size $ndw^{\log n}$ using Lemma 6.1 and Observation 6.2 in logspace. Then, we compose it with the logspace evaluation procedure of Observation 6.3 by iterating over each assignment in the hitting set, till we find a non-zero evaluation. If all the evaluations are 0, we say that the input polynomial is zero and declare it non-zero, otherwise. Since both the procedures are logspace, then so is the PIT algorithm, by Fact 1.1.

Theorem 6.4. *Let F be an n-variate, w-width ROABP of individual degree d over a field $\mathbb{F}$ of characteristic 0 or $\mathrm{char}(\mathbb{F}) \geq ndw^{\log n}$. Further, let b be the bit-complexity of the field constants appearing in F. Then, there is a deterministic algorithm that, given whitebox access to F, decides whether F computes an identically zero polynomial, in space $O(w \cdot e_{\mathbb{F}}(d, b) + \log n)$.*

7 Logspace PIT for log-depth circuits having constant transcendence degree

Let $\boldsymbol{f} \triangleq \{f_1, \ldots, f_m\} \subseteq \mathbb{F}[x_1, x_2, \ldots, x_n]$. We say that $\boldsymbol{f}$ is *algebraically independent over* $\mathbb{F}$ if there does not exist a non-zero polynomial $P(y_1, \ldots, y_m) \in \mathbb{F}[y_1, \ldots, y_m]$ such that $P(f_1, \ldots, f_m) \equiv 0$. Otherwise, $\boldsymbol{f}$ is called *algebraic dependent over* $\mathbb{F}$. A maximal algebraic independent subset of $\boldsymbol{f}$ is known as a *transcendence basis* of $\boldsymbol{f}$ and the size of such a transcendence basis is called as the *transcendental degree* of $\boldsymbol{f}$, denoted $\mathrm{trdeg}_{\mathbb{F}}\boldsymbol{f}$. Now, we state the main theorem.

Theorem 7.1. *Let $d, r, n, m, r \in \mathbb{N}$ and $\mathbb{F}$ be a field satisfying either $char(\mathbb{F}) > d^r$ or $char(\mathbb{F}) = 0$ and $|\mathbb{F}| \geq drn(r+1)^2$. Let $T_1, \ldots, T_m \in \mathbb{F}[x_1, x_2, \ldots, x_n]$ be products of linear polynomials such that $\deg(T_i) \leq d$ for every $i \in [n]$ and $trdeg_{\mathbb{F}}\{T_1, \ldots, T_m\} \leq r$. Let C be an m-variate, size-s, $O(\log n)$-depth, and poly-degree circuit over $\mathbb{F}$ and let b be the bit-complexity of constants appearing in $C(T_1, \ldots, T_m)$ [13]. Then, given the circuit $C(T_1, \ldots, T_m)$, we can decide whether $C(T_1, \ldots, T_m) \equiv 0$ in space $O(r \cdot (e_{\mathbb{F}}(s, b) + \log d))$, where $e_{\mathbb{F}}(s, b)$ is described in Definition 2.9.*

Remark 7.2. *The model given in the above theorem was studied in [ASSS16]. They gave a polynomial-time deterministic algorithm for the circuits of the kind $C(T_1, \ldots, T_m)$, where C is an m-variate, polynomial degree circuit of arbitrary depth, and $T_1, \ldots, T_m$ are products of linear polynomials such that $trdeg_{\mathbb{F}}\{T_1, \ldots, T_m\}$ is bounded. The main reason for considering $O(\log n)$-depth restriction on the circuit C is that at some point our algorithm evaluates the given circuit on some points in $\mathbb{F}^n$ and we do not know if the problem of evaluating an arbitrary arithmetic circuit is in L (see Section 1.1 in this regard). It is a well-known fact that a $O(\log n)$-depth arithmetic*

[13] The circuit $C(T_1, \ldots, T_m)$ is obtained by replacing y_i with T_i for every $i \in [m]$.

circuit can be converted to an arithmetic formula with only a polynomial blow-up in the size, and thus we can evaluate $C(T_1, \ldots, T_m)$ in logspace.

Remark 7.3. *Let $\mathcal{C}$ be the class of circuits of the kind $C(T_1, \ldots, T_k)$, where C computes a linear form in $\mathbb{F}[y_1, \ldots, y_m]$, and $T_1, \ldots, T_m$ are products of linear polynomials in $\mathbb{F}[x_1, x_2, \ldots, x_n]$ such that $trdeg_{\mathbb{F}}\{T_1, \ldots, T_m\}$ is constant. Let $\sum^{[k]} \prod \sum$ be a class of bounded-top fan-in depth 3 arithmetic circuits over $\mathbb{F}$. Then, trivially, for every constant $k \in \mathbb{N}$, $\sum^{[k]} \prod \sum \subseteq \mathcal{C}$. Thus, a result of [ASSS16] gives a polynomial-time blackbox PIT algorithm for depth 3 circuits with bounded top fan-in whenever satisfying some constraints on the characteristic of $\mathbb{F}$ and over such fields, Theorem 7.1 gives a whitebox logspace PIT algorithm for depth 3 circuits with bounded top fan-in.*

The logspace algorithm in the above theorem follows from the polynomial-time blackbox algorithm given in [ASSS16] and an $O(e_{\mathbb{F}}(s, b))$-space arithmetic formula evaluation algorithm over $\mathbb{F}$. Hence, we omit the details of the proof.

8 Discussion and Future Work

In this section, we discuss some future directions. We gave a logspace reduction from whitebox PIT for $\mathcal{C}_{k+1}$ to whitebox PIT $\sum^{[2]} \mathcal{C}_k$. As we have a logspace whitebox PIT for $\sum^{[2]} \mathcal{C}_1$ (see Theorem 4.9), this result allowed us to obtain a logspace whitebox PIT algorithm for $\mathcal{C}_2$, the class of multilinear read-twice formulas. We extended this result further to obtain a whitebox logspace PIT algorithm for the class $\mathcal{C}_1 + \mathcal{C}_2$. As we have a polynomial-time whitebox PIT algorithm for $\mathcal{C}_k$ whenever $k \in \mathbb{N}$ is a constant, we wonder if there is also a logspace whitebox PIT for the same class. We note this as an open question below.

- **Design a logspace whitebox PIT algorithm for the class of multilinear constant-read formulas**: The first step in this direction would be to obtain a logspace whitebox PIT for the class of multilinear read-thrice formulas, $\mathcal{C}_3$. It follows from Theorem 4.6 that for achieving this, it suffices to obtain a logspace PIT algorithm for $\sum^{[2]} \mathcal{C}_2$.
- **Search-to-decision in Logspace**: Let $\mathcal{C}$ a class of formulas that has a whitebox logspace PIT algorithm. Can one design (another) logspace algorithm that given a formula $F \in \mathcal{C}$ outputs a non-zero assignment of F (if one exists, of course)? Clearly, having a logspace PIT algorithm is a perquisite for such a task. Is it also sufficient? While this question is interesting in its own right, obtaining such an algorithm would in particular imply logspace PIT algorithms for all multilinear read-k formulas with bounded k. In [SV09] the equivalence between these two tasks was shown in the polynomial-time setting for any circuit class.
- **The one-way space complexity of PIT**: A naïve PIT algorithm in this setting can be carried out in $\mathsf{DSPACE}(n \log s)$. Can one show better space bounds? $\mathsf{DSPACE}(o(n \log s))$ or even $\mathsf{BPSPACE}(o(n \log s))$?

9 Acknowledgement

The authors would like to thank Howard Straubing for being a part of the discussion for understanding that arithmetic formula evaluation problem can be carried out in logarithmic space.

References

[AB03] M. Agrawal and S. Biswas. Primality and identity testing via chinese remaindering. *JACM*, 50(4):429–443, 2003. 2

[AB09] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009. 14

[AGKS15] M. Agrawal, R. Gurjar, A. Korwar, and N. Saxena. Hitting-sets for ROABP and sum of set-multilinear circuits. *SIAM J. Comput.*, 44(3):669–697, 2015. 7

[Agr05] M. Agrawal. Proving lower bounds via pseudo-random generators. In *Proceedings of the 25th FSTTCS*, volume 3821 of *LNCS*, pages 92–105, 2005. 2

[AHK93] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. *J. ACM*, 40(1):185–210, jan 1993. 4

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004. 2

[All98] 1998. Lecture notes on Complexity of Computation by Eric Allender, Link - https://people.cs.rutgers.edu/~allender/538/murata3.pdf. 44

[Alo99] N. Alon. Combinatorial nullstellensatz. *Combinatorics, Probability and Computing*, 8:7–29, 1999. 12

[ASSS16] M. Agrawal, C. Saha, R. Saptharishi, and N. Saxena. Jacobian hits circuits: Hitting sets, lower bounds for depth-d occur-k formulas and depth-3 transcendence degree-k circuits. *SIAM J. Comput.*, 45(4):1533–1562, 2016. 2, 7, 12, 37, 38

[AvMV11] M. Anderson, D. van Melkebeek, and I. Volkovich. Derandomizing polynomial identity testing for multilinear constant-read formulae. In *Proceedings of the 26th Annual IEEE Conference on Computational Complexity, (CCC)*, pages 273–282, 2011. Full version at https://eccc.weizmann.ac.il/report/2010/188. 33

[AvMV15] M. Anderson, D. van Melkebeek, and I. Volkovich. Derandomizing polynomial identity testing for multilinear constant-read formulae. *Computational Complexity*, 24(4):695–776, 2015. 2, 7, 8, 10, 13, 15, 16, 22, 30, 32, 33

[BB98] D. Bshouty and N. H. Bshouty. On interpolating arithmetic read-once formulas with exponentiation. *JCSS*, 56(1):112–124, 1998. 4

[BC92] M. Ben-Or and R. Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, 1992. 5, 14

[BC98] N. H. Bshouty and R. Cleve. Interpolating arithmetic read-once formulas in parallel. *SIAM J. on Computing*, 27(2):401–413, 1998. 4

[BCGR92] Samuel R. Buss, Stephen A. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM J. Comput.*, 21(4):755–780, 1992. 3, 14, 17, 47

[BCH86] P. Beame, S. A. Cook, and H. J. Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15(4):994–1003, 1986. 3

[BGV23] Pranav Bisht, Nikhil Gupta, and Ilya Volkovich. Towards Identity Testing for Sums of Products of Read-Once and Multilinear Bounded-Read Formulae. In *43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2023)*, volume 284 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. 15

[BHH95a] N. H. Bshouty, T. R. Hancock, and L. Hellerstein. Learning arithmetic read-once formulas. *SIAM J. on Computing*, 24(4):706–735, 1995. 4

[BHH95b] N. H. Bshouty, T. R. Hancock, and L. Hellerstein. Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates. *JCSS*, 50:521–542, 1995. 4

[BHH95c] N.H. Bshouty, T.R. Hancock, and L. Hellerstein. Learning boolean read-once formulas over generalized bases. *J. Comput. Syst. Sci.*, 50(3):521–542, jun 1995. 4

[BIZ18] K. Bringmann, C. Ikenmeyer, and J. Zuiddam. On algebraic branching programs of small width. *J. ACM*, 65(5):32:1–32:29, 2018. 5

[BOT88] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 301–309, 1988. 2

[BSV23] V. Bhargava, S. Saraf, and I. Volkovich. Linear independence, alternants and applications. In *STOC '23: 55th Annual ACM SIGACT Symposium on Theory of Computing, Orlando, Florida, June 20-23, 2023*. ACM, 2023. 2

[CDL01] A. Chiu, G. I. Davida, and B. E. Litow. Division in logspace-uniform nc1. *RAIRO Theor. Informatics Appl.*, 35:259–275, 2001. 14

[DL78] R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978. 2, 4

[DLN+22] S. Datta, N. Limaye, P. Nimbhorkar, T. Thierauf, and F. Wagner. Planar graph isomorphism is in log-space. *ACM Trans. Comput. Theory*, 14(2):8:1–8:33, 2022. 3, 17, 47

[DS07] Z. Dvir and A. Shpilka. Locally decodable codes with 2 queries and polynomial identity testing for depth 3 circuits. *SIAM J. on Computing*, 36(5):1404–1434, 2007. 2

[FS13] M. A. Forbes and A. Shpilka. Quasipolynomial-time identity testing of non-commutative and read-once oblivious algebraic branching programs. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 243–252, 2013. Full version at https://eccc.weizmann.ac.il/report/2012/115. 7

[FSS14] M. A. Forbes, R. Saptharishi, and A. Shpilka. Pseudorandomness for multilinear read-once algebraic branching programs, in any order. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 867–875, 2014. Full version at https://eccc.weizmann.ac.il/report/2013/132. 15

[GHKL18] M. Ganardi, D. Hucke, D. König, and M. Lohrey. Circuits and expressions over finite semirings. *ACM Trans. Comput. Theory*, 10(4), aug 2018. 3, 44

[GKS17] R. Gurjar, A. Korwar, and N. Saxena. Identity testing for constant-width, and any-order, read-once oblivious arithmetic branching programs. *Theory of Computing*, 13(2):1–21, 2017. 7, 12, 35, 36

[GST23] N. Gupta, C. Saha, and B. Thankey. Equivalence test for read-once arithmetic formulas. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 4205–4272. SIAM, 2023. 4

[HAM02] W. Hesse, E. Allender, and D. A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002. Special Issue on Complexity 2001. 3, 14

[HH91] T. R. Hancock and L. Hellerstein. Learning read-once formulas over fields and extended bases. In *Proceedings of the 4th Annual Workshop on Computational Learning Theory (COLT)*, pages 326–336, 1991. 4

[HS80] J. Heintz and C. P. Schnorr. Testing polynomials which are easy to compute (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC)*, pages 262–272, 1980. 2

[HV06] A. Healy and E. Viola. Constant-depth circuits for arithmetic in finite fields of characteristic two. In *STACS*, volume 3884 of *Lecture Notes in Computer Science*, pages 672–683. Springer, 2006. 3

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988. 3

[KI04] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1-2):1–46, 2004. 2, 5, 6

[KLN+93] M. Karchmer, N. Linial, I. Newman, M. Saks, and A. Wigderson. Combinatorial characterization of read-once formulae. *Discrete Math.*, 114(1–3):275–282, April 1993. 4

[KMSV13] Z. S. Karnin, P. Mukhopadhyay, A. Shpilka, and I. Volkovich. Deterministic identity testing of depth 4 multilinear circuits with bounded top fan-in. *SIAM J. on Computing*, 42(6):2114–2131, 2013. 15, 16

[KS01] A. Klivans and D. Spielman. Randomness efficient identity testing of multivariate polynomials. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 216–223, 2001. 2

[KS08] Z. S. Karnin and A. Shpilka. Deterministic black box polynomial identity testing of depth-3 arithmetic circuits with bounded top fan-in. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity (CCC)*, pages 280–291, 2008. 2

[KS09] N. Kayal and S. Saraf. Blackbox polynomial identity testing for depth 3 circuits. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2009. Full version at https://eccc.weizmann.ac.il/report/2009/032. 2

[KUW86] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random nc. *Combinatorica*, 6:35–48, 1 1986. 2

[Lad75] R. E. Ladner. The circuit value problem is log space complete for *P*. *SIGACT News*, 7(1):18–20, 1975. 3

[Lin92] S. Lindell. A logspace algorithm for tree canonization (extended abstract). In *STOC*, pages 400–404. ACM, 1992. 3, 17, 47

[Lov79] L. Lovasz. On determinants, matchings, and random algorithms. In L. Budach, editor, *Fundamentals of Computing Theory*. Akademia-Verlag, 1979. 2

[LV03] R. J. Lipton and N. K. Vishnoi. Deterministic identity testing for multivariate polynomials. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 756–760, 2003. 2

[MS21] D. Medini and A. Shpilka. Hitting sets and reconstruction for dense orbits in vp_{e} and ΣΠΣ circuits. In Valentine Kabanets, editor, *36th Computational Complexity Conference, CCC 2021, July 20-23, 2021, Toronto, Ontario, Canada (Virtual Conference)*, volume 200 of *LIPIcs*, pages 19:1–19:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 4

[MV18] D. Minahan and I. Volkovich. Complete derandomization of identity testing and reconstruction of read-once formulas. *TOCT*, 10(3):10:1–10:11, 2018. 2, 7, 11, 15, 28, 35

[MVV87] K. Mulmuley, U. Vazirani, and V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987. 2

[MW07] Pierre McKenzie and Klaus W. Wagner. The complexity of membership problems for circuits over sets of natural numbers. *Comput. Complex.*, 16(3):211–244, 2007. 14

[Nis93] N. Nisan. On read-once vs. multiple access to randomness in logspace. *Theor. Comput. Sci.*, 107(1):135–144, 1993. 5

[Rei08] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), sep 2008. 3

[RS05] R. Raz and A. Shpilka. Deterministic polynomial identity testing in non commutative models. *Computational Complexity*, 14(1):1–19, 2005. 2, 7

[Sap16] R. Saptharishi. A survey of lower bounds in arithmetic circuit complexity. Technical report, https://github.com/dasarpmar/lowerbounds-survey/, 2016. 48

[Sax09] N. Saxena. Progress on polynomial identity testing. *Bulletin of EATCS*, 99:49–79, 2009. 2

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980. 2, 4

[Sha90] A. Shamir. IP=PSPACE. In *Proceedings of the Thirty First Annual Symposium on Foundations of Computer Science*, pages 11–15, 1990. 2

[SS11] N. Saxena and C. Seshadhri. Blackbox identity testing for bounded top fanin depth-3 circuits: the field doesn't matter. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 431–440, 2011. 2

[SS12] N. Saxena and C. Seshadhri. Blackbox identity testing for bounded top-fanin depth-3 circuits: The field doesn't matter. *SIAM J. Comput.*, 41(5):1285–1298, 2012. 2

[SS13] N. Saxena and C. Seshadhri. From sylvester-gallai configurations to rank bounds: Improved blackbox identity test for depth-3 circuits. *J. ACM*, 60(5):33, 2013. 2

[ST17] O. Svensson and J. Tarnawski. The matching problem in general graphs is in quasi-NC. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 696–707, 2017. 2

[ST21] C. Saha and B. Thankey. Hitting sets for orbits of circuit classes and polynomial families. In Mary Wootters and Laura Sanità, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2021, August 16-18, 2021, University of Washington, Seattle, Washington, USA (Virtual Conference)*, volume 207 of *LIPIcs*, pages 50:1–50:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 4

[SV09] A. Shpilka and I. Volkovich. Improved polynomial identity testing for read-once formulas. In *APPROX-RANDOM*, pages 700–713, 2009. Full version at https://eccc.weizmann.ac.il/report/2010/011. 38

[SV14] A. Shpilka and I. Volkovich. On reconstruction and testing of read-once formulas. *Theory of Computing*, 10:465–514, 2014. 4

[SV15] A. Shpilka and I. Volkovich. Read-once polynomial identity testing. *Computational Complexity*, 24(3):477–532, 2015. 2, 4, 8, 10, 11, 12, 14, 15, 16, 28, 33, 34, 35

[SV18] S. Saraf and I. Volkovich. Blackbox identity testing for depth-4 multilinear circuits. *Combinatorica*, 38(5):1205–1238, 2018. 2

[SY10] A. Shpilka and A. Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010. 2, 13, 48

[Sze87] Róbert Szelepcsényi. The moethod of focing for nondeterministic automata. *Bull. EATCS*, 33:96–99, 1987. 3

[Vol15] I. Volkovich. Deterministically factoring sparse polynomials into multilinear factors and sums of univariate polynomials. In *APPROX-RANDOM*, pages 943–958, 2015. 15, 16

[Vol16] I. Volkovich. Characterizing arithmetic read-once formulae. *ACM Transactions on Computation Theory (ToCT)*, 8(1):2, 2016. 4

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 216–226, 1979. 2, 4

A Missing proofs from Section 1

Observation A.1. *Let $\mathbb{F}$ be a field. Then,* PIT *for arithmetic circuits over $\mathbb{F}$ is* P*-hard.*

Proof. Let C-eval be the following problem: given an arithmetic circuit $\mathtt{C}$ over $\mathbb{F}$ and an assignment $\mathbf{a} \in \mathbb{F}^n$, decide whether $\mathtt{C}(\mathbf{a}) = 0$. As noted in Section 1.1, C-eval is P-complete [GHKL18], i.e., C-eval $\in \mathsf{P}$ and every problem in P reduces in logspace to C-eval. To show that PIT for circuits is P-hard over $\mathbb{F}$, it suffices to prove that C-eval reduces in logspace to PIT circuits. Let $\mathtt{C}$ be an arbitrary arithmetic circuit that computes a polynomial in $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Let y be a fresh variable and $\widehat{C} := Cy$ be an arithmetic circuit. Let $\mathbf{a} \in \mathbb{F}^n$ be an arbitrary assignment. Then, $\mathtt{C}(\mathbf{a}) = 0$ if and only if $y \cdot \mathtt{C}(\mathbf{a})$ is the identically zero polynomial. This immediately implies that we can reduce C-eval to PIT for arithmetic circuits over $\mathbb{F}$ in logspace. □

Observation A.2. *Let $\mathbb{F}$ be a field and* PIT*-Formulas be the PIT on arithmetic formulas on $\mathbb{F}$. Then,*

$$\mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Formulas}} = \mathsf{BP} \cdot \mathsf{L}.$$

Proof. Recall that $\mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Formulas}}$ is the class of problems in $\mathsf{BP} \cdot \mathsf{L}$ with oracle access to PIT-Formulas. As shown in Observation 1, PIT-Formulas $\in \mathsf{BP} \cdot \mathsf{L}$. We observe that $\mathsf{BP} \cdot \mathsf{L}$ is *self-low*, i.e., $\mathsf{BP} \cdot \mathsf{L}^{\mathsf{BP} \cdot \mathsf{L}} = \mathsf{BP} \cdot \mathsf{L}$. Its proof can be shown using a proof that BPP is self-low (see [All98]). As $\mathsf{BP} \cdot \mathsf{L}$ is self-low, we get

$$\mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Formulas}} \subseteq \mathsf{BP} \cdot \mathsf{L}^{\mathsf{BP} \cdot \mathsf{L}} \subseteq \mathsf{BP} \cdot \mathsf{L}.$$

Hence, $\mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Formulas}} = \mathsf{BP} \cdot \mathsf{L}$. □

Observation A.3. *Let $\mathbb{F}$ be a field and* PIT*-Circuits be the PIT on arithmetic formulas on $\mathbb{F}$. Then,*

$$\mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Circuits}} = \mathsf{BP} \cdot \mathsf{L}^{\mathsf{P}} = \mathsf{BPP}.$$

Proof. As noted in the proof of Observation A.2, BPP is self-low, i.e., $\mathsf{BPP}^{\mathsf{BPP}} = \mathsf{BPP}$. We know due to the Schwartz-Zippel lemma that PIT-circuits$\in \mathsf{BPP}$. We also know that $\mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{BPP}$. Then, from Observation A.1,

$$\mathsf{BP} \cdot \mathsf{L}^{\mathsf{P}} \subseteq \mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Circuits}} \subseteq \mathsf{BPP}^{\text{PIT-Circuits}} \subseteq \mathsf{BPP}^{\mathsf{BPP}} = \mathsf{BPP}.$$

Now, lets see a proof in the other direction. Recall that a language $L \in \mathsf{BPP}$ if and only if there exists a deterministic Turing Machine M such that for every string x, M runs in time poly$(|x|)$ and

$$\forall x \in L : \Pr_r[M(x, r) = 1] \geq \frac{2}{3},$$

$$\forall x \notin L : \Pr_r[M(x,r) = 1] \leq \frac{1}{3}.$$

Let us fix an input string x arbitrarily. Let $A_x := \{r : M(x,r) = 1\}$. It is not difficult to verify that $A_x \in \mathsf{P}$. Then, it can be easily shown that there exists a deterministic Turing machine M' that takes oracle access to P such that for every string x, M' has two way access to a random string, it runs in space $O(\log(|x|))$ and

$$\forall x \in L : \Pr_r[M'(x,r) = 1] \geq \frac{2}{3},$$

$$\forall x \notin L : \Pr_r[M'(x,r) = 1] \leq \frac{1}{3}.$$

This implies that $L \in \mathsf{BP} \cdot \mathsf{L}^{\mathsf{P}}$. We know from Observation A.1 that every problem in P reduces in log space to PIT-circuits. This implies $L \in \mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Circuits}}$. So, we have shown that hence $\mathsf{BPP} \subseteq \mathsf{BP} \cdot \mathsf{L}^{\mathsf{P}} \subseteq \mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Circuits}}$. This proves that

$$\mathsf{BP} \cdot \mathsf{L}^{\text{PIT-Circuits}} = \mathsf{BP} \cdot \mathsf{L}^{\mathsf{P}} = \mathsf{BPP}.$$

□

Observation A.4. $\mathsf{P} \subseteq \mathsf{BP} \cdot \mathsf{L}$ *if and only if* $\mathsf{BPP} = \mathsf{BP} \cdot \mathsf{L}$.

Proof. Suppose $\mathsf{P} \subseteq \mathsf{BP} \cdot \mathsf{L}$. We know from Observation A.3 that $\mathsf{BPP} = \mathsf{BP} \cdot \mathsf{L}^{\mathsf{P}}$. Then, the self-low property of $\mathsf{BP} \cdot \mathsf{L}$ mentioned in the proof of Observation A.2 implies $\mathsf{BPP} \subseteq \mathsf{BP} \cdot \mathsf{L}^{\mathsf{BP} \cdot \mathsf{L}} = \mathsf{BP} \cdot \mathsf{L}$. We know that trivially $\mathsf{BP} \cdot \mathsf{L} \subseteq \mathsf{BPP}$. Thus, $\mathsf{BPP} = \mathsf{BP} \cdot \mathsf{L}$.

Now, suppose $\mathsf{BPP} = \mathsf{BP} \cdot \mathsf{L}$. As $\mathsf{P} \subseteq \mathsf{BPP}$, we get $\mathsf{P} \subseteq \mathsf{BP} \cdot \mathsf{L}$. □

B Analysis of Generic Traversal

To prove Lemma 3.1, we will require the following definitions.

Definition B.1. *Consider the following definitions with respect to a node v in formula F.*

- *We say that a node v in the formula has been* processed *by Algorithm 1, if v will not be visited again. In other words, $\Psi(v)$ has been computed, and pushed to stack if necessary.*
- *For a node v in F, let $\pi_F(v)$ denote the path from root(F) to the node v. It is well defined since F is a tree and there is a unique path between any two nodes.*
- *For nodes* curr *and* prev *in Algorithm 1, we define a modification of path $\pi_F(\text{curr})$:*

$$\pi'_F(\text{curr}) \triangleq \begin{cases} \pi_F(\text{curr}), & \text{if prev} \in \pi_F(\text{curr}) \\ \pi_F(\text{curr}) \cup \{\text{prev}\}, & \text{otherwise.} \end{cases}$$

- *For a path π in F, we define right turns in π as the sequence of edges in π that go from a node to its right child.*

We prove the following invariant that holds for Algorithm 1.

Claim B.2. *Let* curr *be the current node in Algorithm 1. Then,*

1. $\beta = \Psi(v)$, *where* v *is the last processed node in* F.

2. Γ *corresponds to the reverse sequence of right turns in* $\pi'_F(\text{curr})$. *That is, for every right turn:* $v \to v.\text{right}$ *in* π', Γ *stores* $\Psi(v.\text{left})$. *By reverse, we mean that the top* Ψ *value in* Γ *corresponds to the latest right turn and the bottom* Ψ *value corresponds to the first right turn in* $\pi'_F(\text{curr})$.

Proof. Initialization: We have curr = root(F). No vertex has been processed and there is no right turn. In Line 2, indeed $\beta = \perp$ and $\Gamma = \emptyset$, as desired.

Maintenance: We show that the invariant is maintained in each case of the while loop in Algorithm 1.

- If curr is a leaf, then we process it and we update $\beta = \Psi(\text{curr})$ in Line 5. We also go up in Line 5. After executing it, we did not change $\pi'_F(curr)$ due to the second case in definition of π' which is consistent with the fact that we did not push or pop from the stack Γ.

- If prev = parent(curr) (coming from up), then we go left. In this case, in Line 8, we neither processed any node, nor made or removed any right turns, thus invariant is maintained because we also don't update β or Γ.

- If prev = curr.left (coming from left), then we push β on top of stack Γ and move right. Note that at Line 9, by the invariant, β was holding the Ψ value of last processed node, i.e. curr.left. Since, we did not process any new node in this case, we don't update β as desired. We made a right turn in this step, and indeed after executing Line 10, $\Gamma.\text{top} \leftarrow \Psi(\text{curr.left}) = \beta$, thus maintaining the invariant.

- If prev = curr.right (coming from right), then observe that by definition of π' and the loop invariant, $\Gamma.\text{top} = \Psi(\text{curr.left})$ at Line 12, corresponding to the last right turn in $\pi'_F(\text{curr})$. We also have $\beta = \Psi(\text{curr.right})$ by the loop invariant. Thus, after executing Lines 14-16, we process curr and indeed store Ψ value of the last processed node in β. We also removed a right turn from π', but to match it, we did pop from Γ also, thus maintaining the invariant.

Termination: At the end of the while loop curr = $\perp$ and prev = root(F). The last processed node is root(F) and $\pi'_F(\text{curr}) = \{\text{root}(F)\}$. Because we maintained the loop invariant, we then get $\beta = \Psi(\text{root}(F))$ and $\Gamma = \emptyset$, as desired. □

By the loop invariant in Claim B.2, we now give the proof of Lemma 3.1.

Proof. Correctness: The algorithm follows from the invariant in Claim B.2. Therefore, when the algorithm terminates β holds the correct Ψ-value for root(F), i.e. $\beta = \Psi(F)$ and the stack Γ is empty.

Space-complexity: To track the two pointers curr, prev we need only $2 \cdot \log s$ space. For different nodes v, to execute the procedure $\Psi(v)$, we keep reusing the space, therefore we only need total $O(S_\Psi)$ space for the whole formula. However, to compute the value $\Psi(v)$, we use additional memory from the stack Γ by storing Ψ values corresponding to the right turns in $\pi'_F(v)$. We claim that for the *deepest* leaf v in F (the leaf at maximum depth), the number of right turns in $\pi'_F(v)$ is given by the recurrence relation $L(s) \leq L(s/2) + 1$. This is because, whenever we make a right turn we at least halve the size of the formula, as F is left-heavy (after preprocessing in Line 1).

Hence, $L(s) \leq O(\log s)$. Each entry in stack Γ stores a Ψ value of bit-complexity t and maximum length of the stack used is $L(s)$. Therefore, the total work space used by Algorithm 1 is given by $2\log s + O(S_\Psi) + t \cdot L(s) \leq O(S_\Psi + t \cdot \log s)$. □

C Log-space algorithm for making a formula left-heavy

For the sake of completeness, we give a log-space algorithm to make a formula left-heavy. This has been used in several papers (e.g. [BCGR92, Lin92, DLN+22] etc.).

Algorithm 4: Making a formula left-heavy

Input: An arithmetic formula F
Output: Left-heavy form of F

Initialize curr $\triangleq$ root(F), prev $\triangleq \perp$.
`/* curr and prev track the current and previous nodes */`
while curr $\neq \perp$ **do**
  `/* if curr is leaf, go to its parent. */`
  if curr *is a leaf* **then**
    prev $\leftarrow$ curr, curr $\leftarrow$ parent(curr). **continue**
  end
  `/* curr is not a leaf in the below cases */`
  `/* If coming from up, go left */`
  if prev = parent(curr) **then** prev $\leftarrow$ curr, curr $\leftarrow$ curr.left
  `/* If coming from left, go right */`
  if prev = curr.left **then**
    prev $\leftarrow$ curr, curr $\leftarrow$ curr.right
  end
  `/* If coming from right, compare the sizes of left and right subtree,`
  `interchange if necessary */`
  if prev = curr.right **then**
    if *size*(curr.left) $<$ *size*(curr.right) **then**
      $tmp \leftarrow$ curr.left, curr.left $\leftarrow$ curr.right, curr.right $\leftarrow tmp$
    end
    `/* Go up */`
    prev $\leftarrow$ curr, curr $\leftarrow$ parent(curr).
  end
end

Now we describe the size function.

Algorithm 5: Computing the size of the subtree rooted at a given node

Input: An arithmetic formula F, a node r
Output: Size of the subtree rooted at r

Initialize curr $\triangleq r$, prev $\triangleq \perp$, $size = 0$.
`/* curr and prev track the current and previous nodes */`
while curr $\neq$ parent(r) **do**
  `/* if curr is a leaf, increment size and go to its parent. */`
  if curr *is a leaf* **then**
    $size \leftarrow size + 1$, prev $\leftarrow$ curr, curr $\leftarrow$ parent(curr).
  end
  `/* curr is not a leaf in the below cases */`
  `/* If coming from up, go left */`
  if prev = parent(curr) **then** prev $\leftarrow$ curr, curr $\leftarrow$ curr.left
  `/* If coming from left, go right */`
  if prev = curr.left **then**
    prev $\leftarrow$ curr, curr $\leftarrow$ curr.right
  end
  `/* If coming from right, go up */`
  if prev = curr.right **then**
    prev $\leftarrow$ curr, curr $\leftarrow$ parent(curr).
  end
end
Return($size$)

D Arithmetic circuits, formulas and ROABPs

In this section, we give the formal definitions of the various algebraic models of computation discussed in this work. For a detailed exposition, we refer the reader to the excellent surveys of [SY10] and [Sap16].

An *arithmetic circuit* is defined as a directed acyclic graph, where the leaves are labelled with input variables or field constants and the internal nodes are either + (addition) or × (multiplication). An addition node adds all the polynomials on its incoming edges, while a multiplication node multiplies. We have a single root node as the output node at top. The edges can also be labeled with field constants which get multiplied to the polynomial computed at the node below. An unlabelled edge can be thought to be labelled with the constant 1. The computation happens in a natural bottom-up fashion. The in-degree of a node is called it *fan-in* and out-degree is called *fan-out*. An arithmetic circuit has two resources: size and depth. *Size* of the circuit is size of the underlying graph, given by the number of edges and nodes. *Depth* of the circuit is the length of the longest path from some leaf node to the output node. The class VP is defined as the class of poly(n)-sized circuits computing polynomials of poly(n)-degree.

An *arithmetic formula* or simply formula in short is defined as an arithmetic circuit where every node has at most one outgoing edge. The underlying graph for a formula has a tree structure.

An *algebraic branching program* (ABP) is a layered directed graph with a unique source vertex s and sink vertex t. The ABP of *depth-d* has $d+1$ layers– $V_0, V_1, \ldots, V_d$, where the first layer

$V_0 \triangleq \{s\}$, and the last layer $V_d \triangleq \{t\}$. The directed edges go from layer i to $i+1$, for $0 \leq i \leq d-1$ and are labeled with *linear* polynomials. The *weight of a path* p is $W(p) := \prod_{e \in p} W(e)$, where $W(e)$ denotes the weight (or label) of an edge. The final polynomial $f(\mathbf{x})$ *computed by the ABP* is then simply the weighted sum of all the paths from source to sink: $f(\mathbf{x}) := \sum_{\text{path } p\,:s \rightsquigarrow t} W(p)$. The *length* of the ABP is the number of layers. We say that the ABP has *width* w, if for $0 \leq i \leq d$, $|V_i| \leq w$. *Size* of the ABP is the product of its length and width. VBP is the class of all polynomial-sized ABPs.

The ABP can also be viewed as a product of matrices with polynomial entries. Let $V_i = \{v_{i,j} \mid j \in [w]\}$. Then, $f(\mathbf{x}) = \prod_{i=1}^{d} D_i$, where $D_1 \in \mathbb{F}^{1 \times w}[\mathbf{x}]$, $D_i \in \mathbb{F}^{w \times w}[\mathbf{x}]$ (for $2 \leq i \leq d-1$), and $D_d \in \mathbb{F}^{w \times 1}[\mathbf{x}]$ such that the entries are:

$$\begin{aligned} D_1(j) &= W(s, v_{1,j}) \text{ , for } j \in [w] \\ D_i(j,k) &= W(v_{i-1,j}, v_{i,k}) \text{ , for } j,k \in [w] \text{ and } 2 \leq i \leq d-1 \\ D_d(k) &= W(v_{d-1,k}, t) \text{ , for } k \in [w] \text{ .} \end{aligned}$$

If there is no edge (u, v) in the ABP, we set $W(u, v) \triangleq 0$.

An ABP is called *read-once oblivious* ABP (ROABP) if each variable appears in only one layer and instead of linear polynomials, edge weights are univariate polynomials. Thus an ROABP has n-many layers, one for each variable. The *variable order* $(x_{\pi(1)}, \ldots, x_{\pi(n)})$ of ROABP is the sequence in which ROABP reads the variables in each layer. *Size* of an ROABP is given by three parameters - width, individual degree and number of variables.

In the matrix product form, ROABP $D(\mathbf{x}) = \prod_{i=1}^{n} D_i$, where $D_1 \in \mathbb{F}^{1 \times w}[x_{\pi(1)}]$, $D_i \in \mathbb{F}^{w \times w}[x_{\pi(i)}]$ for $2 \leq i \leq n-1$, and $D_n \in \mathbb{F}^{w \times 1}[x_{\pi(n)}]$.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il