Tree Evaluation Is in Space $O(\log n \cdot \log\log n)$

James Cook
Unaffiliated
Toronto, Canada
falsifian@falsifian.org

Ian Mertz
University of Warwick
Coventry, United Kingdom
ian.mertz@warwick.ac.uk

ABSTRACT

The *Tree Evaluation Problem* (TreeEval) (Cook et al. 2009) is a central candidate for separating polynomial time (P) from logarithmic space (L) via *composition*. While space lower bounds of $\Omega(\log^2 n)$ are known for multiple restricted models, it was recently shown by Cook and Mertz (2020) that TreeEval can be solved in space $O(\log^2 n/\log\log n)$. Thus its status as a candidate hard problem for L remains a mystery.

Our main result is to improve the space complexity of TreeEval to $O(\log n \cdot \log\log n)$, thus greatly strengthening the case that Tree Evaluation is in fact in L.

We show two consequences of these results. First, we show that the *KRW conjecture* (Karchmer, Raz, and Wigderson 1995) implies $\mathsf{L} \not\subseteq \mathsf{NC}^1$; this itself would have many implications, such as branching programs not being efficiently simulable by formulas. Our second consequence is to increase our understanding of *amortized branching programs*, also known as *catalytic branching programs*; we show that every function f on n bits can be computed by such a program of length $\mathrm{poly}(n)$ and width $2^{O(n)}$.

CCS CONCEPTS

• **Theory of computation → Complexity theory and logic**; **Complexity classes**.

KEYWORDS

Tree Evaluation Problem, Catalytic Computation, KRW Conjecture, Branching Programs, Logspace, Composition Theorems

ACM Reference Format:
James Cook and Ian Mertz. 2024. Tree Evaluation Is in Space $O(\log n \cdot \log\log n)$. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24), June 24–28, 2024, Vancouver, BC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3618260.3649664

1 INTRODUCTION

In complexity theory, many fundamental questions about time and space remain open, including their relationship to one another. We know that $\mathsf{TIME}(t)$ is sandwiched between $\mathsf{SPACE}(\log t)$ and $\mathsf{SPACE}(t/\log t)$ [18], and both containments are widely considered to be strict, but we have made little progress in proving this fact for any t.

STOC '24, June 24–28, 2024, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0383-6/24/06
https://doi.org/10.1145/3618260.3649664

1.1 Tree Evaluation and Composition

The *Tree Evaluation Problem* [10], henceforth TreeEval, has emerged in recent years as a candidate for a function which is computable in polynomial time ($\mathsf{P} = \mathsf{TIME}(n^{O(1)})$) but not in logarithmic space ($\mathsf{L} = \mathsf{SPACE}(O(\log n))$). This would resolve one of the two fundamental questions of time and space, showing that $\mathsf{TIME}(t)$ strictly contains $\mathsf{SPACE}(\log t)$ in at least one important setting.

TreeEval is parameterized by alphabet size k and height h. The input is a rooted full binary tree of height h, where each leaf is given a value in $[k]$ and each internal node is given a function from $[k] \times [k]$ to $[k]$ represented explicitly as a table of k^2 values. This defines a natural bottom-up way to evaluate the tree: inductively from the leaves, the value of a node is the value its function takes when given the labels from its two children as input. The output of a $\mathsf{TreeEval}_{k,h}$ instance is the value of its root node.

A $\mathsf{TreeEval}_{k,h}$ instance has size $2^h \cdot \mathrm{poly}(k)$. The description of the problem as given defines a polynomial time algorithm for $\mathsf{TreeEval}_{k,h}$: evaluate each node starting from the bottom and going up, spending $\mathrm{poly}(k)$ time at each of the 2^h nodes.

But what about space? Evaluating the output node requires us to have the values of both of its children, which themselves are obtained by computing their respective children, and so on. Now imagine we have computed one of the children of the output node and are moving to the other. This seems to require remembering the value we have computed on one side, using $\log k$ bits of memory, and then on the other side computing a whole new $\mathsf{TreeEval}_{k,h-1}$ instance, for which the same logic applies. This would inductively give a space $\Omega(h \log k)$ algorithm, while $\mathsf{TreeEval}_{k,h} \in \mathsf{L}$ would mean giving an algorithm using only $O(h + \log k)$ bits of memory.

Thus if our intuition is correct, this should be a separating example for L and P. This led Cook, McKenzie, Wehr, Braverman, and Santhanam [10] to define TreeEval and conjecture that $\Omega(h \log k)$ space is optimal. The conjecture was supported by multiple subsequent works, which showed it holds in restricted, but also non-uniform, settings such as *thrifty* algorithms [10]—a TreeEval-specific restriction wherein algorithms are not allowed to read "unnecessary" input bits, i.e. locations in the internal function tables that do not correspond to the true inputs to the node—and *read-once* [14] programs. Later works extended both of these results to the non-deterministic setting [20, 22].

This idea, known as *composition* or *direct product* theorems, is not only studied in the context of space. The *KRW conjecture* of Karchmer, Raz, and Wigderson [21] states that a similar logic holds for formula depth, with the upshot being that TreeEval separates P from the class of logarithmic depth formulas, known as NC^1. Even more so than space, the study of the KRW conjecture has yielded many partial results (see e.g. [6, 13]) as well as encouraging useful parallel lines of work such as *lifting theorems* [16, 26].

Thus the study of composition, and by extension TreeEval, is a very fruitful and well-founded line of study, and it is of great interest as to when this logic holds and when it fails.

1.2 Known Upper Bounds

Nevertheless, the consensus and central composition logic of the space hardness of TreeEval has faced a challenge ever since its inception. Buhrman, Cleve, Koucký, Loff, and Speelman [4] defined a new model of space-bounded computation called *catalytic computing* in order to challenge a crucial assumption in our lower bound strategy: that the space used for *remembering old values* in the tree cannot be useful for *computing new values*. Building on the work of Barrington [1] and Ben-Or and Cleve [2], they show that the presence of full memory can in fact assist in space-bounded computation in a particular setting (unless L can compute log-depth threshold circuits, which would imply many things which are widely disbelieved, e.g. $\mathsf{NL} = \mathsf{L}$).

The catalytic computing model later received attention from a variety of works [3, 5, 12, 17], but while it was in part motivated to challenge the conjecture of [10], it did not immediately lead to any results about TreeEval. However, after a period of quiet on both the upper and lower bound fronts, their objection was validated by Cook and Mertz [7, 8], who showed that the $\Omega(h \log k)$ argument does not hold. They proved that for any k and h, $\mathsf{TreeEval}_{k,h}$ can be computed in space $O(h \log k / \log h)$, which translates to an algorithm using space at most $O(\log^2 n / \log\log n)$, shaving a logarithmic factor off of the trivial algorithm using space $O(\log^2 n)$.

This is a far cry from showing $\mathsf{TreeEval} \in \mathsf{L}$, but both the statement and proof of the result undermine the central compositional logic behind the approach of [10] to separate L from P.

1.3 Main Result

In this work we give an exponential improvement on the central subroutine of [7, 8], which yields the following result.

Theorem 1. TreeEval *can be computed in space* $O(\log n \cdot \log\log n)$.

Compared to having only a logarithmic factor improvement given by [7, 8], we are now only a logarithmic factor improvement away from showing $\mathsf{TreeEval} \in \mathsf{L}$.

Our proof relies on a few fundamental properties of *primitive roots of unity* over finite fields. After defining the main preliminaries in Section 2, we go over these properties in Section 3, with our main proof of Theorem 1 in Section 4. We then improve and generalize our main subroutine, plus a discussion of the implications of these sharper results, in Section 5.

As observed in [7, 8], our techniques avoid the restrictions for which strong lower bounds are known. First, our algorithms avoid the read-once restriction by repeatedly recomputing values throughout the tree. Second, and perhaps more interesting, is that our algorithm avoids the "thrifty" restriction by relying on *every* value in the table of any internal node, not only the one corresponding to the true inputs.

1.4 Implications

Our improvement has immediate consequences outside of studying space upper bounds on TreeEval. We discuss two such results in this paper. All models and statements will be formally defined in Sections 6 and 7 respectively.

1.4.1 The KRW Conjecture. First, we return to our brief discussion of the KRW conjecture, which we recall implies that $\mathsf{TreeEval} \notin \mathsf{NC}^1$. [7, 8] gave a space upper bound of $O(\log^2 n / \log\log n)$ for TreeEval, asymptotically the same as the lower bound on formula depth implied by the KRW conjecture; thus it was possible for the KRW conjecture and $\mathsf{L} \subseteq \mathsf{NC}^1$ to both be true. This is no longer possible, as Theorem 1 makes these two hypotheses incompatible.

Theorem 2. *If the KRW Conjecture holds, then* $\mathsf{L} \not\subseteq \mathsf{NC}^1$.

We have not formally stated the KRW conjecture, and refrain from doing so until Section 6; in fact one can define it in a variety of ways, some stronger than others. We should note, however, that Theorem 2 is quite robust with respect to choosing weaker versions of the conjecture; any statement that implies TreeEval requires formula depth $\omega(\log n)$ is sufficient for Theorem 2. As we show, the strongest (and most widely studied) version implies that L requires formulas of depth $\Omega(\log^2 n / \log\log n)$, which nearly meets the upper bound of $O(\log^2 n)$ given by $\mathsf{L} \subseteq \mathsf{NC}^2$.

There are multiple important takeaways. First, the KRW conjecture now implies a much sharper separation than $\mathsf{P} \neq \mathsf{NC}^1$. Second, the KRW conjecture would give a superpolynomial size separation between *non-uniform* formulas and *uniform* branching programs; no superpolynomial separation is known even when the uniformity, or lack thereof, is the same for both classes. Third, proving formula lower bounds for TreeEval via KRW is *formally no easier* than proving the same lower bounds for st-connectivity, even in the undirected case. And fourth, and most philosophically, continued belief in the KRW conjecture is a bet that the ability to handle composition is the factor that separates space and formulas.

1.4.2 Catalytic Branching Programs. For our second result, we consider the question of *catalytic* branching program size, or equivalently *amortized* branching program size.

Branching programs are a syntactic model used to analyze space in the non-uniform setting: we have a directed acyclic graph (DAG) with one source node and two sinks, one for each potential output of the function f; computation proceeds by starting at the source and, until we reach a sink labeled with the output $f(x)$, at the current internal node we query a bit of x and proceed down some adjacent edge according to the value read.

Drawing a connection to a model of space known as *catalytic* computation, Girard, Koucký, and McKenzie [15] introduced a model known as *m-catalytic branching programs*, which essentially asks whether we can find smaller branching programs for computing an arbitrary function f if we only want to do so in an *amortized* sense. We now consider a DAG with m source nodes and $2m$ sink nodes, one for each (source, output) pair, and require that restricting attention to any individual source gives us a branching program for f in the usual sense. Nevertheless, we do not require internal nodes to be disjoint; the question becomes whether such a program can have size much less than sm, where s is the size of the optimal single-source branching program for f, and preferably with the smallest value of m, i.e. the least amount of amortization, possible.

Potechin [24] showed that, given enough amortization, this is possible in the strongest way: every function f has m-catalytic

branching programs of size $O(mn)$, regardless of the complexity of f with respect to ordinary branching programs; the only catch is that m must be at least 2^{2^n}. Reinterpreting and building on work of Potechin [24] and an improvement by Robere and Zuiddam [27], Cook and Mertz [9] used the TreeEval argument of [7, 8] in the non-uniform setting to show that the amount of amortization can be reduced to $m = 2^{2^{\epsilon n}}$ for arbitrarily small constants $\epsilon > 0$.

By improving (a generalization of) the central subroutine of [7, 8] in Theorem 1, we show that a slight sacrifice in the length gives a near-optimal improvement in the amount of amortization.

Theorem 3. *For every function f on n bits, f has m-catalytic branching programs of the following size:*

- *size $O(m \cdot n^{2+\epsilon})$ with $m = O(2^{(1+2/\epsilon)n})$*
- *size $O(m \cdot n^3/\log^2 n)$ with $m = O(2^{(2+o(1))n})$*
- *size $O(m \cdot 2^{2/\epsilon} n^2)$ with $m = O(2^{(2+\epsilon\log n)n})$*

where $\epsilon \in (0, 1/2]$ in the first and third points can be made arbitrarily small.

Focusing on the first point, Theorem 3 can be interpreted as saying that every function can be computed in amortized branching program size just above n^2, where the total size of the program is roughly $2^{O(n)}$. By the same counting argument as ordinary branching programs, we can hope for no better than amortized size $O(n)$—as achieved by [9, 24, 27]—and total size $O(2^n/n)$, meaning we are not far from the tightest parameters possible.

2 PRELIMINARIES

In this work the base of logarithms will always be 2: $\log x := \log_2 x$.

2.1 Register Programs

We will use *register programs* as a convenient abstraction for describing space-bounded algorithms. Register programs were introduced by Ben-Or and Cleve [2] based on work of Coppersmith and Grossman [11] and explored in a number of follow-up works [4, 7, 9].

Definition 1. A *register program* over ring $\mathcal{R}$ consists of a collection of memory locations $R = \{R_1 \dots R_s\}$, called *registers*, each of which can hold one element from $\mathcal{R}$, and an ordered list of *instructions* in the form of updates to some register R_i based the current values of the registers and an input $x \in \{0,1\}^n$.

We are primarily interested in register programs which can be simulated by space-bounded algorithms:

Definition 2. A family of register programs $P = \{P_n\}_{n\in\mathbb{N}}$ is *space $c(n)$ uniform* if there is an algorithm using space $c(n)$ which, given (t, x) and access to an array of registers, performs the t-th instruction of P_n on input $x \in \{0,1\}^n$.

Although it is common to restrict register programs to a small vocabulary of instructions, in this work we make no restriction beyond Definition 2. So, our programs may include any instruction

$$R_i \leftarrow R_i + g(x_1 \dots x_n, R_1 \dots R_s)$$

as long as g can be computed in space $c(n)$.

Following [4], rather than directly writing their output to a register, our programs will *add* their output to a register while leaving other registers unchanged, a process we call *clean* computation. This will be useful for making our algorithms space-efficient.

Definition 3. Let $\mathcal{R}$ be a ring and let f be a function whose output can be represented in $\mathcal{R}$. A register program over $\mathcal{R}$ with s registers *cleanly computes* f into a register R_o if for all possible $x_1 \dots x_n \in \{0,1\}^n$ and $\tau_1 \dots \tau_s \in \mathcal{R}$, if the program is run after initializing each register $R_i = \tau_i$, then at the end of the execution

$$R_o = \tau_o + f(x_1 \dots x_n)$$
$$R_i = \tau_i \quad \forall i \neq o$$

We will often want to undo the effect of a register program:

Definition 4. If P is a register program that cleanly computes $f(x_1 \dots x_n)$, an *inverse* to P is any program P^{-1} which computes $-f(x_1 \dots x_n)$.

For example, one way to construct P^{-1} is:

1: $R_o \leftarrow -R_o$
2: P
3: $R_o \leftarrow -R_o$

Notice that running P followed by P^{-1}, or vice versa, leaves every register including R_o unchanged.

We justify our use of uniform register programs and clean computation to describe space-bounded algorithms with the following connection:

Proposition 1. *For $n \in \mathbb{N}$, let $c := c(n), s := s(n), t := t(n) \in \mathbb{N}$, and let $\mathcal{R} := \mathcal{R}_n$ be a ring. Let $f := f_n$ be a Boolean function on n variables, and let $P := P_n$ be a space c uniform register program, which s registers over $\mathcal{R}$ and which has t instructions in total, that cleanly computes f. Then f can be computed in space $O(c + s\log|\mathcal{R}| + \log t)$.*

2.2 Finite Fields

In our programs, the ring $\mathcal{R}$ will always be a *finite field*. For a prime number p and positive integer a, we define $\mathbb{F}_{p^a}$ to be the unique (up to isomorphism) field with p^a elements.

Proposition 2. *Every element $x \in \mathbb{F}_{p^a}$ can be represented by a string of length $O(\log|\mathbb{F}_{p^a}|) = O(a\log p)$, and given any two such strings representing $x, y \in \mathbb{F}_{p^a}$, the representation of $x + y$, $x \times y$, and x/y over $\mathbb{F}_{p^a}$ can be computed in space $O(\log|\mathbb{F}_{p^a}|) = O(a\log p)$.*

Proof. Fix an irreducible degree-a polynomial $f(x) \in \mathbb{F}_p[x]$, so that $\mathbb{F}_{p^a}$ is isomorphic to $\mathbb{F}_p[x]/(f(x))$. Then each field element is represented by a polynomial of degree less than a, which we can store as an a-tuple of coefficients in $\mathbb{F}_p$. It is then straightforward to add, multiply and divide field elements in $O(a\log p)$ space. All this requires finding a suitable $f(x)$ to begin with; this can also be done in $O(a\log p)$ space by exhaustive search. □

We will sometimes need a smaller field inside a larger finite field:

Proposition 3. *For every prime number p and positive integers a, b, the field $\mathbb{F}_{p^a}$ is isomorphic to a subfield of $\mathbb{F}_{p^{ab}}$.*

Again it is computationally possible to find representations of $\mathbb{F}_{p^a}$ and $\mathbb{F}_{p^{ab}}$ that agree[1]; thus we will treat $\mathbb{F}_{p^a}$ as a subset of $\mathbb{F}_{p^{ab}}$ when performing computations.

[1]For example, one way to do that is to first find an irreducible polynomial $f(x) \in \mathbb{F}_p[x]$ such that $\mathbb{F}_{p^a}$ is isomorphic to $\mathbb{F}_p[x]/(f(x))$, and then find $g(y) \in \mathbb{F}_{p^a}[y]$ such that $F_{p^{ab}}$ is isomorphic to $\mathbb{F}_{p^a}[y]/(g(y))$, with elements of $\mathbb{F}_a$ being represented as constant (degree-0) polynomials in $F_{p^a}[y]$.

3 ROOTS OF UNITY

Our work will use *primitive roots of unity*, and so we introduce them and some of their properties before describing our algorithms. All definitions and statements appearing in this section are standard and have been used many times before in the literature, but will be crucial to the proof of our main results.

Definition 5. An element ω of a field $\mathcal{K}$ is a *root of unity* of order m if $\omega^m = 1$. It is a *primitive* root of unity if additionally $\omega^k \neq 1$ for every integer $0 < k < m$.

Our algorithm relies on some properties of primitive roots of unity—naturally, first we require that they exist, with the order we need:

Proposition 4. *Every finite field $\mathcal{K}$ has a primitive root of unity of order $|\mathcal{K}| - 1$.*

This follows from the fact that the multiplicative group $\mathcal{K}^\times$ of a finite field is always a cycle. For $\mathcal{K} = \mathbb{F}_{p^a}$, such a primitive root of unity can be found in $O(a \log p)$ space through exhaustive search.

We will use, and for completeness prove, a generalization of the fact that $\sum_{j=1}^{m} \omega^j = 0$.

Proposition 5. *Let $\mathcal{K}$ be a finite field, and let ω be a primitive root of unity of order m in $\mathcal{K}$. Then for all $0 < b < m$,*

$$\sum_{j=1}^{m} \omega^{jb} = 0$$

Proof. Let $s = \sum_{j=1}^{m} \omega^{jb}$. Then

$$\omega^b s = \sum_{j=2}^{m+1} \omega^{jb} = s + \omega^{(m+1)b} - \omega^b = s + \omega^b(\omega^{mb} - 1) = s$$

since $\omega^{mb} = 1^b = 1$. So either $\omega^b = 1$ or $s = 0$, but the former is ruled out because ω is a *primitive* root of unity and $0 < b < m$. □

Corollary 6. *Let $\mathcal{K}$ be a finite field, let $m = |\mathcal{K}| - 1$, and let ω be a primitive root of unity of order m in $\mathcal{K}$. Then for all $0 \leq b < m$,*

$$\sum_{j=1}^{m} \omega^{jb} = -1 \cdot [b = 0]$$

where $[b = 0]$ is the indicator function which takes value 1 if $b = 0$ and 0 otherwise.

Proof. The case of $b \neq 0$ is handled by Proposition 5. For $b = 0$ we have that over $\mathcal{K}$,

$$\sum_{j=1}^{m} \omega^{j0} = \sum_{j=1}^{m} 1 = m = -1$$

where the last equality holds because $m = -1$ in $\mathcal{K}$. □

4 TREE EVALUATION IN LOW SPACE

We now move on to the main goal of our paper, which is to prove Theorem 1. The following is our main result for $\mathsf{TreeEval}_{k,h}$, stated in terms of the two main parameters. It implies Theorem 1 for any setting of k and h, and is stronger as k gets smaller with respect to the total input size.

Theorem 4. *Any $\mathsf{TreeEval}_{k,h}$ instance can be computed in space $O((h + \log k) \cdot \log\log k)$.*

We will build our algorithm from the ground up, first showing how to compute each individual node.

Lemma 7. *Let $\mathcal{K}$ be a finite field, let $m = |\mathcal{K}| - 1$, and let ω be a primitive root of unity of order m in $\mathcal{K}$. Let $d < m$, and let τ_i, x_i be elements of $\mathcal{K}$ for $i \in [d]$. Then*

$$\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega^j \tau_i + x_i) = -1 \cdot \prod_{i=1}^{d} x_i$$

Before going into the proof of Lemma 7, we should stress why it is useful. Our overall goal is to compute the function f_u at node u in our TreeEval instance while only using clean access to its inputs, i.e. we only assume we can add some input bit x_i to whatever τ_i already exists in the target register R_i. Thus, when operating over registers R_i, we need to remove the contributions of the τ_i values themselves when computing f_u. Lemma 7 accomplishes just this for the AND function over d inputs, albeit using τ_i multiplied by m different coefficients. After proving this lemma, we will move to the actual question, which is to compute an arbitrary f_u.

Proof. For a fixed j, expanding the product on the left hand side gives

$$\begin{aligned}\prod_{i=1}^{d} (\omega^j \tau_i + x_i) &= \sum_{S \subseteq [d]} \left(\prod_{i \in S} \omega^j \tau_i\right)\left(\prod_{i \in [d] \setminus S} x_i\right) \\ &= \sum_{S \subseteq [d]} \omega^{j|S|} \left(\prod_{i \in S} \tau_i\right)\left(\prod_{i \in [d] \setminus S} x_i\right)\end{aligned}$$

If we sum over all j and switch the sums we get

$$\begin{aligned}&\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega^j \tau_i + x_i) \\ &= \sum_{j=1}^{m} \sum_{S \subseteq [d]} \omega^{j|S|} \left(\prod_{i \in S} \tau_i\right)\left(\prod_{i \in [d] \setminus S} x_i\right) \\ &= \sum_{S \subseteq [d]} \left(\sum_{j=1}^{m} \omega^{j|S|}\right)\left(\prod_{i \in S} \tau_i\right)\left(\prod_{i \in [d] \setminus S} x_i\right)\end{aligned}$$

By Corollary 6 we have

$$\sum_{j=1}^{m} \omega^{j \cdot |S|} = -1 \cdot [|S| = 0]$$

and thus the outer sum simplifies to the $|S| = 0$ term, which only has $S = \emptyset$:

$$\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega^j \tau_i + x_i) = -1 \cdot \left(\prod_{i \in \emptyset} \tau_i\right)\left(\prod_{i \in [d] \setminus \emptyset} x_i\right) = -1 \cdot \prod_{i \in [d]} x_i \quad \square$$

Thus the next step is to move from individual products to polynomials. This is accomplished by a simple corollary of Lemma 7.

Lemma 8. *Let $\mathcal{K}$ be a finite field, let $m = |\mathcal{K}| - 1$, and let ω be a primitive root of unity of order m in $\mathcal{K}$. Let $p : \mathcal{K}^n \to \mathcal{K}$ be a degree-d polynomial for some $d < m$, and let τ_i, x_i be elements of $\mathcal{K}$ for $i \in [n]$. Then*

$$\sum_{j=1}^{m} -1 \cdot p(\omega^j \tau_1 + x_1, \ldots, \omega^j \tau_n + x_n) = p(x_1 \ldots x_n)$$

Proof. Writing p as a sum of monomials we have

$$p(y_1 \ldots y_n) = \sum_{\substack{I \subseteq [n] \\ |I| \le d}} c_I \prod_{i \in I} y_i$$

for some coefficients $c_I \in \mathcal{K}$ and formal variables $y_1 \ldots y_n$. Then by substituting $\omega^j \tau_i + x_i$ for each y_i and summing over all j, Lemma 7 gives

$$\begin{aligned}
&\sum_{j=1}^{m} -1 \cdot p(\omega^j \tau_1 + x_1, \ldots, \omega^j \tau_n + x_n) \\
&= \sum_{j=1}^{m} -1 \cdot \sum_{\substack{I \subseteq [n] \\ |I| \le d}} c_I \prod_{i \in I} (\omega^j \tau_i + x_i) \\
&= \sum_{\substack{I \subseteq [n] \\ |I| \le d}} c_I \cdot \left(-1 \cdot \sum_{j=1}^{m} \prod_{i \in I} (\omega^j \tau_i + x_i) \right) \\
&= \sum_{\substack{I \subseteq [n] \\ |I| \le d}} c_I \prod_{i \in I} x_i
\end{aligned}$$

and the last line is $p(x_1 \ldots x_n)$ by definition. □

Finally, we show how to use Lemma 8 in a register program to compute our polynomial f_u in the way we described above, given an appropriate choice of $\mathcal{K}$.

Lemma 9. *Let $\mathcal{K}$ be a finite field such that $m := |\mathcal{K}| - 1 > 2\lceil \log k \rceil$. Let P_ℓ, P_r be register programs which cleanly compute values $v_\ell, v_r \in \{0,1\}^{\lceil \log k \rceil}$ into registers $R_\ell, R_r \in \mathcal{K}^{\lceil \log k \rceil}$, respectively, and let P_ℓ^{-1}, P_r^{-1} be their inverses. Let $f_u : \{0,1\}^{2\lceil \log k \rceil} \to \{0,1\}^{\lceil \log k \rceil}$ be the function at node u in our* $\mathsf{TreeEval}_{k,h}$ *instance.*

Then there exists a register program P_u which cleanly computes $f_u(v_\ell, v_r) \in \{0,1\}^{\lceil \log k \rceil}$ into registers $R_u \in \mathcal{K}^{\lceil \log k \rceil}$, as well as an inverse program P_u^{-1}. Both P_u and P_u^{-1} make m recursive calls each to P_ℓ, P_r, P_ℓ^{-1}, and P_r^{-1}, and use $5m\lceil \log k \rceil$ other basic instructions.

Proof. Our goal will be to use Lemma 8 in order to compute the output of f_u using only clean access to the values of its children. In order to do this, we first need to convert f_u into a tuple of polynomials. We can write the i-th bit of f_u as:

$$(f_u(y,z))_i = \sum_{\alpha,\beta,\gamma \in [k]^3} [\alpha_i = 1][f_u(\beta,\gamma) = \alpha][y = \beta][z = \gamma]$$

We will turn this into a polynomial whose $2\lceil \log k \rceil$ variables are the bits of y and z by replacing $[y = \beta]$ with the polynomial $e(y,\beta) = \prod_{i=1}^{\lceil \log k \rceil} (1 - y_i + (2y_i - 1)\beta_i)$, which equals $[y = \beta]$ when all $y_i \in \{0,1\}$, and similarly replacing $[z = \gamma]$ with $e(z,\gamma)$. This gives the polynomial

$$\sum_{\substack{\alpha,\beta,\gamma \in [k]^3 \\ \alpha_i = 1}} [f_u(\beta,\gamma) = \alpha] e(y,\beta) e(z,\gamma) \tag{1}$$

We call this $q_{u,i}(y,z)$ and note that it is multilinear and thus has degree at most $2\lceil \log k \rceil$.

Now given the conversion to polynomials $q_{u,i}$, we use Lemma 8 to compute the values $q_{u,i}(y,z)$ for inputs y, z coming from P_ℓ and P_r respectively. Let ω be a primitive root of unity of order m in $\mathcal{K}$, and for all $c \in \{\ell, r\}$ and $i \in [\lceil \log k \rceil]$, let $\tau_{c,i}$ be the initial value of $R_{c,i}$. Our goal will be to compute

$$R_{u,i} \leftarrow R_{u,i} + \sum_{j=1}^{m} -1 \cdot q_{u,i}(\omega^j \tau_\ell + y, \tau_r + z) \qquad \forall i \in [\lceil \log k \rceil]$$

where $\omega^j \tau_\ell + y$ and $\omega^j \tau_r + z$ are shorthand for $\lceil \log k \rceil$ values each. We do so using the following program P_u:

1: **for** $j = 1 \ldots m$ **do**
2: **for** $c \in \{\ell, r\}, i = 1 \ldots \lceil \log k \rceil$ **do**
3: $R_{c,i} \leftarrow \omega^j \cdot R_{c,i}$
4: P_ℓ, P_r
5: **for** $i = 1 \ldots \lceil \log k \rceil$ **do**
6: $R_{u,i} \leftarrow R_{u,i} - q_{u,i}(R_\ell, R_r)$
7: P_ℓ^{-1}, P_r^{-1}
8: **for** $c \in \{\ell, r\}, i = 1 \ldots \lceil \log k \rceil$ **do**
9: $R_{c,i} \leftarrow \omega^{-j} \cdot R_{c,i}$

We use **for** … **do** as shorthand for concatenating several copies of a block of instructions with varying parameters. So, for example, lines 2–3 describe a sequence of $2\lceil \log k \rceil$ register program instructions with a different pair (c,i) associated to each, and the block from lines 2–9 is repeated m times with different values of j. Lines 4 and 7 are shorthand for inserting complete copies of the register programs $P_\ell, P_r, P_\ell^{-1}, P_r^{-1}$.

On the other hand, each of lines 3, 6, and 9 represents a single instruction (to be repeated several times due to the surrounding **for** loops), even though computing line 6 involves $\mathrm{poly}(k)$ field arithmetic operations. Recall from Section 2 that a single instruction of a space c uniform register program may compute any function computable in space c. See the end of the proof of Theorem 4 for an account of the space c required for these instructions.

To make the inverse program P_u^{-1}, replace the $-$ on line 6 with $+$.

We now analyze the correctness of the program. At the start of an iteration of the loop, we have $R_{c,i} = \tau_{c,i}$, and since lines 7–9 are the inverse of lines 3–4, this invariant is maintained at the end of the iteration; this additionally implies that $R_{c,i} = \tau_{c,i}$ at the end of the program as required. Going into lines 5 and 6, we have that

$$R_{c,i} = \omega^j \tau_{c,i} + v_{c,i} \qquad \forall c \in \{\ell, r\}, i \in [\lceil \log k \rceil]$$

where m is larger than the degree of each $q_{u,i}$, and so correctness follows from Lemma 8 and the fact that $q_{u,i}(y,z) = (f_u(y,z))_i$ when all $y_i, z_i \in \{0,1\}$. □

The above program can be made more efficient, as we will show in Lemma 10 in Section 5, but even as stated Lemma 9 is sufficient to serve as our main TreeEval subroutine.

Proof of Theorem 4. We will show that our $\mathsf{TreeEval}_{k,h}$ instance can be cleanly computed by a register program of length at most $(4|\mathcal{K}|)^h\lceil\log k\rceil$ and using $3\lceil\log k\rceil$ registers over $\mathcal{K}$, and that the register program is space $O(h\log|\mathcal{K}|+\log k)$ uniform. By Proposition 1, our space usage will ultimately be

$$O(h\log|\mathcal{K}|+\log k+\log k\cdot\log|\mathcal{K}|)$$

which is $O((h+\log k)\log\log k)$ if we choose $\mathcal{K}$ to be a field of size $O(\log k)$.

We build our register program by induction, showing that for every node u of height $d\le h$ such a program of length $(4|\mathcal{K}|)^d\lceil\log k\rceil$ computing f_u exists. For $d=0$, i.e. a leaf node, both P_u and P_u^{-1} can be computed by reading the node's value directly from the input, which gives register programs of length

$$\lceil\log k\rceil=(4\cdot|\mathcal{K}|)^0\lceil\log k\rceil$$

since one instruction is needed for each of the $\lceil\log k\rceil$ output registers.

Now for a node u at height $d+1$, we will inductively assume we have register programs P_ℓ,P_r for the children ℓ,r of u, each of length $(4\cdot|\mathcal{K}|)^d\lceil\log k\rceil$ and which use $3\lceil\log k\rceil$ registers. We will organize our registers into tuples R_ℓ,R_r,R_u, where P_ℓ will compute f_ℓ into R_ℓ and P_r will compute f_r into R_r; our goal then will be to compute f_u into P_u.

Assuming $|\mathcal{K}|-1>2\lceil\log k\rceil$, we apply Lemma 9 to u, inductively giving us a program of length

$$(|\mathcal{K}|-1)\cdot[4\cdot(4\cdot|\mathcal{K}|)^d\lceil\log k\rceil+5\lceil\log k\rceil]\le(4\cdot|\mathcal{K}|)^{d+1}\lceil\log k\rceil$$

This completes the recursion. We choose

$$\mathcal{K}=\mathbb{F}_{2^{\lceil\log(2\lceil\log k\rceil+2)\rceil}}$$

which satisfies our two conditions[2]: 1) $\mathcal{K}$ has size $O(\log k)$, ensuring efficiency; and 2) $|\mathcal{K}|-1>2\lceil\log k\rceil$, ensuring correctness.

It remains only to show that our register program is space $O(h\log|\mathcal{K}|+\log k)$ uniform. Recall this means (Definition 2) that on input (t,x), we can perform the t-th step of the program in space $O(h\log|\mathcal{K}|+\log k)$.

The first task is to figure out what the t-th instruction is. The register program given by Lemma 9 has an outer loop with $m=|\mathcal{K}|-1$ iterations, so the first step is to figure out which iteration the instruction lies within—i.e. the value of j—and which instruction number t' it is within that iteration: $t=Tj+t'$ where T is the length of each iteration. Then based on t' we must determine where within the iteration the instruction lies; for example, if $t'\le 2\lceil\log k\rceil$, it's line 3, with the values of $c\in\{\ell,r\}$ and $i\in[\lceil\log k\rceil]$ determined by t'. If the instruction lies within one of the recursive calls to $P_\ell,P_r,P_\ell^{-1},P_r^{-1}$, then we must figure out where within that recursive call the instruction lies, and so on. This can all be done with simple arithmetic; since the length of the program is at most $(4|\mathcal{K}|)^h\lceil\log k\rceil$, this requires space $O(h\log|\mathcal{K}|+\log k)$.[3]

Finally the instruction itself must be performed. Lines 3 and 9 can be performed in space $O(\log k+\log|\mathcal{K}|)$, because field operations can be performed in space $O(\log|\mathcal{K}|)$ (Proposition 2), and $\log j\le\log k$ bits suffice to create a loop to compute ω^j.

It remains to compute line 6. We do this using the definition of $q_{u,i}$ as stated in Equation 1. Taking the outer sum means storing three values in $[k]$, for $3\lceil\log k\rceil$ bits in total, plus $O(\log|\mathcal{K}|)$ bits to keep track of the total thus far. Each coefficient $[f_u(\beta,\gamma)=\alpha]$ is directly given in the input to TreeEval and can be addressed using $O(\log n)=O(h+\log k)$ bits. We can compute the product one value at a time, using one counter for the index and one field element for the product thus far, giving $\lceil\log k\rceil$ and $O(\log|\mathcal{K}|)$ bits, respectively. Lastly, by taking into account the $O(\log|\mathcal{K}|)$ space of computing operations over $\mathcal{K}$ (again by Proposition 2), the total space usage is at most $O(\log k+h+\log|\mathcal{K}|)$. □

5 IMPROVEMENTS AND GENERALIZATIONS

For the rest of this paper we will adapt the techniques used to other questions in complexity theory. To do so, we will first state Lemma 9, which is our main subroutine, in a more general and efficient form.

Lemma 10. *Let $\mathcal{K}$ be a finite field with a subfield $\mathcal{F}\subseteq\mathcal{K}$, let $f:\mathcal{F}^a\to\mathcal{F}^b$ be a function where $a(|\mathcal{F}|-1)<|\mathcal{K}|-1$, and let P_g be a register program with at least $a+b$ registers over $\mathcal{K}$ which cleanly computes a value $g\in\mathcal{F}^a$ into registers $R_1\dots R_a$.*

Then there exists a register program P_f which cleanly computes $f(g)$ into registers $R_{a+1}\dots R_{a+b}$. The length of P_f is $(|\mathcal{K}|-1)(t(P_g)+2a+b)$ where $t(P_g)$ is the length of P_g, and P_f uses the same set of registers as P_g.

To see Lemma 9 as a special case[4] of Lemma 10, take $\mathcal{F}=\mathbb{F}_2$, $a=2\lceil\log k\rceil$ and $b=\lceil\log k\rceil$, and let g be the concatenation of the values v_ℓ,v_r, with P_g calling P_ℓ then P_r. Lemma 10 saves some time by avoiding the need to call the inverse program P_g^{-1}.

The proof is essentially that of Lemma 9, and will appear at the end of this section. First, we will use this statement to obtain our results in the next two sections.

To get a sense of the utility of this generalization, as a first application we show how to reduce the space used by our TreeEval algorithm for storing registers. Our algorithm currently uses space $O(\log n\cdot\log\log n)$ both to keep track of time and to store the memory in the registers. We can improve this to logspace for one of these two aspects, namely the register memory.

Theorem 5. *Any $\mathsf{TreeEval}_{k,h}$ instance can be computed in space $O(h\log\log k+\log k)$.*

One consequence of this theorem is that only $\mathsf{TreeEval}_{k,h}$ instances of essentially maximal height can possibly be used to prove space lower bounds.

[2] Any other $\mathcal{K}$ satisfying these constraints would work: for example, $\mathbb{F}_p$ where p is a prime number between $2\lceil\log k\rceil+2$ and $4\lceil\log k\rceil+4$.

[3] Put another way, tracking where we are within the recursive calls requires up to h stack frames, each storing a number $j\in[|\mathcal{K}|-1]$, plus $O(\log k)$ bits to understand which iteration of the loops on lines 2, 5, and 8 we are on if we are not inside a recursive call, for a total of $O(h\log|\mathcal{K}|+\log k)$ space.

[4] Strictly speaking, it is not a special case, since Lemma 9 encodes values as bit strings (meaning $\mathcal{F}=\mathbb{F}_2$ in terms of Lemma 10) but does not require $\mathbb{F}_2$ to be a subfield of $\mathcal{K}$.

Theorem 6. *Any* $\mathsf{TreeEval}_{k,h}$ *instance with* $h = O(\log k/\log\log k)$ *can be computed in* L.

Another consequence is that if we convert our algorithms into *layered branching programs* (see Section 7) computing $\mathsf{TreeEval}_{k,h}$, we can reduce the width to $\mathrm{poly}(n)$ with no asymptotic loss in length. We will not formally state or prove this result.

The proof of Theorem 5 is similar to that of Theorem 4, except that instead of representing elements of $[k]$ in binary, we represent them as tuples of field elements for some larger field $\mathcal{F} \subseteq \mathcal{K}$. The number of registers needed to represent elements of $[k]$ will thus shrink by a factor of $\log|\mathcal{F}|$. Our field $\mathcal{K}$ will be polynomially larger than before (because the degree of the polynomial interpolated by Lemma 10 grows with $|\mathcal{F}|$), but since our space usage was $O((h + \log k) \cdot \log|\mathcal{K}|)$, i.e. our space only depends logarithmically on $|\mathcal{K}|$, this will ultimately not impact our asymptotics.

Proof of Theorem 5. Let $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$ where r and s will be determined later. By Proposition 3 we may assume $\mathcal{F} \subseteq \mathcal{K}$. An element of $[k]$ can be represented using $\lceil(\log k)/r\rceil$ elements of $\mathcal{F}$, but our registers will hold values in the larger field $\mathcal{K}$.

The induction proof, after converting f_u into polynomials $q_{u,i}$ for each $i \in [\lceil\log k\rceil]$ as in the proof of Lemma 9, is the same as for Theorem 4, except that instead of Lemma 9, we invoke Lemma 10 with the two fields $\mathcal{F} \subseteq \mathcal{K}$, and with $f_u : \mathcal{F}^{2\lceil(\log k)/r\rceil} \to \mathcal{F}^{\lceil(\log k)/r\rceil}$ working with encodings as elements of $\mathcal{F}$ instead of binary. Let $t(h')$ be the length of the program for a node at height $h' \le h$. Then the two children of a node at height $h'+1$ can be computed in time $2t(h')$, so by Lemma 10,

$$t(h'+1) \le 2|\mathcal{K}| \cdot t(h') + \mathrm{poly}(k)$$

and thus $t(h)$ is at most $(2|\mathcal{K}|)^{O(h)}\,\mathrm{poly}(k)$.

Now we are ready to choose $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$. Our algorithm uses $3\lceil(\log k)/r\rceil$ registers, each needing rs bits to store, for a total of

$$3\lceil(\log k)/r\rceil \cdot rs = O(s\log k)$$

space devoted to storing registers. As stated above, the register program has length $(2|\mathcal{K}|)^{O(h)}\,\mathrm{poly}(k)$, and so we need

$$\log\left((2\cdot 2^{rs})^{O(h)}\,\mathrm{poly}(k)\right) = O(hrs + \log k)$$

space to track our position in the program.

Our register program is space $O(h\log|\mathcal{K}| + \log k) = O(hrs + \log k)$ uniform. Recall (Definition 2) that to show this, we must show that given (t, x), we can perform the t-th instruction on input x in space $O(h\log|\mathcal{K}| + \log k)$. Similar to the argument in Theorem 4, determining which instruction is the t-th can be done in space $O(\log t(h)) = O(h\log|\mathcal{K}| + \log k)$. Then, each individual instruction can be computed in space $O(\log|\mathcal{K}| + \log k)$. Looking ahead to the program given in the proof of Lemma 10, lines 2 and 4 are field arithmetic operations which require $O(\log|\mathcal{K}|)$ space (Proposition 2). Line 5 requires evaluating the polynomial p_i, which, examining Equation 2, can be done by looping over all $k^{O(1)}$ values of $(z_1 \dots z_a)$ in the sum, all $a = O(\log k)$ values for ℓ in the product, and then looping up to $|\mathcal{F}| - 1$ to compute the exponent in q_{z_ℓ}, plus $O(\log|\mathcal{K}|)$ space to do field arithmetic and store intermediate results, for a total of $O(\log|\mathcal{K}| + \log k)$ space.

By Proposition 1, in total we need space

$$O(hrs + \log k) + O(s\log k) = O(hrs + s\log k)$$

In order to use Lemma 10 we require

$$2\lceil(\log k)/r\rceil(2^r - 1) < 2^{rs} - 1$$

Choosing $r = \lceil\log\log k\rceil$ gives us

$$2\lceil(\log k)/r\rceil(2^r - 1) \le 4(\log k)^2/\log\log k < 2^{2\log\log k} - 1$$

and thus choosing $s = 2$ satisfies our condition, resulting in an algorithm using space

$$O(hrs + s\log k) = O(h\log\log k + \log k) \qquad \square$$

The rest of the paper will focus on applications of Lemma 10, as it will prove to be stronger and more flexible than Lemma 9 as seen above. To end this section we will prove it, with a proof closely mirroring that of Lemma 9.

Proof of Lemma 10. For each $i = 1 \dots b$ we define a polynomial $p_i(y_1 \dots y_a)$ which computes the i-th coordinate of $f(y_1 \dots y_a)$. Our inspiration will be the formula

$$f_i(y_1 \dots y_a) = \sum_{(z_1 \dots z_a) \in \mathcal{F}^a} f_i(z_1 \dots z_a) \prod_{\ell=1}^{a} [y_\ell = z_\ell]$$

To make this a polynomial, we replace each indicator function $[y_\ell = z_\ell]$ with the polynomial

$$q_{z_\ell}(y_\ell) = 1 - (y_\ell - z_\ell)^{|\mathcal{F}|-1}$$

$q_{z_\ell}(y_\ell)$ has degree $|\mathcal{F}| - 1$, and by Fermat's little theorem we have $q_{z_\ell}(y_\ell) = [y_\ell = z_\ell]$ for any $y_\ell, z_\ell \in \mathcal{F}$. Define

$$p_i(y_1 \dots y_a) = \sum_{(z_1 \dots z_a) \in \mathcal{F}^a} f(z_1 \dots z_a) \prod_{\ell=1}^{a} q_{z_\ell}(y_\ell) \tag{2}$$

Thus p_i is a polynomial of degree $a(|\mathcal{F}| - 1)$.

Now let $m = |\mathcal{K}| - 1$ and let ω be a primitive root of unity of order m in $\mathcal{K}$. By assumption, $a(|\mathcal{F}| - 1) < |\mathcal{K}| - 1$, so m is greater than the degree of the polynomials p_i. Let $\tau_\ell \in \mathcal{K}$ be the initial value of each register R_ℓ. By Lemma 8,

$$\sum_{j=1}^{m} -1 \cdot p_i(\omega^j\tau_1 + y_1 \dots \omega^j\tau_a + y_a) = p_i(y_1 \dots y_a)$$

This leads to the following algorithm. It replaces the inefficient warm-up version presented in the proof of Lemma 9 which required an extra m copies of P_g^{-1}.

1: **for** $j = 1 \dots m$ **do**
2: $\quad R_\ell \leftarrow (\omega^{-1} - 1)^{-1} \cdot R_\ell$ for $\ell = 1 \dots a$
3: $\quad P_g$
4: $\quad R_\ell \leftarrow (1 - \omega) \cdot R_\ell$ for $\ell = 1 \dots a$
5: $\quad R_{a+i} \leftarrow R_{a+i} + (-1) \cdot p_i(R_1 \dots R_a)$ for $i = 1 \dots b$

We may assume $m > 1$ (otherwise p_i has degree 0, so is a constant), so $\omega \ne 1$ and $(\omega^{-1} - 1)^{-1}$ exists and can be used on line 2.

To analyse this algorithm, define $\tau'_\ell = \tau_\ell - g_\ell$ for $\ell = 1 \dots a$. At the start of the j-th iteration of the loop, the following invariants hold for $\ell \in [a], i \in [b]$:

$$R_\ell = \omega^{j-1}\tau'_\ell + g_\ell$$

$$R_{a+i} = \tau_{a+i} + \sum_{j'=1}^{j-1} -1 \cdot p_i(\omega^{j'}\tau'_1 + g_1, \ldots, \omega^{j'}\tau'_a + g_a)$$

It is straightforward to verify this invariant holds after each iteration. After the last iteration, Lemma 8 tells us that for $\ell \in [a], i \in [b]$

$$\begin{aligned} R_\ell &= \omega^m \tau'_\ell + g_\ell \\ &= \tau_\ell - g_\ell + g_\ell = \tau_\ell \\ R_{a+i} &= \tau_{a+i} + \sum_{j=1}^{m} -1 \cdot p_i(\omega^j \tau'_1 + g_1, \ldots, \omega^j \tau'_a + g_a) \\ &= \tau_{a+i} + p_i(g_1 \ldots g_a) \end{aligned}$$

This register program includes m copies of P_g and has a total length of $m(2a + b + t(P_g))$. □

6 APPLICATION 1: THE KRW CONJECTURE SEPARATES L AND NC[1]

We now move on to applications of the statement and proof of Theorem 1. In this section we study its implications in the study of formula lower bounds.

6.1 KRW and TEP

To begin, we formally state the KRW conjecture to fit the discussion from Section 1.

Conjecture 1 (KRW Conjecture [21]). *For a function f, let $depth(f)$ denote the smallest depth of any fan-in two formula computing f. For any functions $g_1 : \{0,1\}^{n_1} \to \{0,1\}$ and $g_2 : \{0,1\}^{n_2} \to \{0,1\}$, define their composition to be*

$$g_1 \circ g_2(x_{1,1} \ldots x_{n_1,n_2}) := g_1(g_2(x_{1,1} \ldots x_{1,n_2}) \ldots g_2(x_{n_1,1} \ldots x_{n_1,n_2}))$$

Then for almost all functions g_1, g_2, it holds that

$$depth(g_1 \circ g_2) \geq depth(g_1) + depth(g_2) - O(1)$$

We note that this conjecture can be weakened by increasing the $O(1)$ subtractive term.

To see why this is connected to TreeEval, we need to consider the unbounded fan-in version of TreeEval. A $\mathsf{TreeEval}_{k,d,h}$ instance is as before, a tree of height h and using alphabet size k, but now each internal node has d children rather than 2.

Lemma 11. *Conjecture 1 implies $depth(\mathsf{TreeEval}_{2,d,h}) = \Omega(dh)$.*

Proof. For each layer $\ell \in [h]$, pick a random function $f_\ell : \{0,1\}^d \to \{0,1\}$, and fix each internal $\mathsf{TreeEval}_{2,d,h}$ node at height ℓ to f_ℓ. By a counting argument, each f_ℓ requires formula depth $\Omega(d)$ with high probability. We apply the KRW Conjecture first to $g_1 = f_1$ and $g_2 = f_2$, then $g_1 = f_1 \circ f_2$ and $g_2 = f_3$, and so on $h - 1$ times, until we ultimately get that the composition of all f_ℓ—which is to say, the $\mathsf{TreeEval}_{2,d,h}$ instance in question—requires depth $\Omega(dh)$. □

Since $\mathsf{TreeEval}_{k,d,h}$ has input size $n = d^h k^d \log k$, fixing $k = 2$ gives us $\log n = O(h \log d + d)$, which implies that $\Omega(dh) = \omega(\log n)$—and thus $\mathsf{TreeEval}_{2,d,h} \notin \mathsf{NC}^1$, assuming Conjecture 1—for the right setting of parameters. We give exact details after establishing the other side of Theorem 2, namely the space complexity of $\mathsf{TreeEval}_{2,d,h}$.

6.2 Space Bounds for $\mathsf{TreeEval}_{k,d,h}$

Using Lemma 10, we can generalize Theorem 1, and in fact Theorem 5, to degrees d other than 2:

Theorem 7. *Any $\mathsf{TreeEval}_{k,d,h}$ instance can be computed in space $O(h \log(d \log k) + d \log k)$.*

Proof. The proof is the same as for Theorem 5 but with d inputs instead of 2. Let $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$ where r and s will be determined later. As before, we represent elements of $[k]$ as tuples of $\lceil (\log k)/r \rceil$ field elements, and consider the function at node u as $f_u : \mathcal{F}^{d\lceil (\log k)/r \rceil} \to \mathcal{F}^{\lceil (\log k)/r \rceil}$. Our algorithm uses $(d+1)\lceil (\log k)/r \rceil$ registers, each needing rs bits to store, for a total of

$$(d+1)\lceil (\log k)/r \rceil \cdot rs = O(ds \log k)$$

space devoted to storing registers. Using Lemma 10, we get a register program of length $(d|\mathcal{K}|)^{O(h)} \operatorname{poly}(k)$ (in this case the program P_g in Lemma 10 must evaluate all d children, hence the d in the base of the exponent), and so we need

$$\log\left((d2^{rs})^{O(h)} \operatorname{poly}(k)\right) = O(hrs + h \log d + \log k)$$

space to track our position in the program. Lastly our program is

$$O(hrs + h \log d + \log k + d\lceil (\log k)/r \rceil r) = O(hrs + h \log d + d \log k)$$

uniform by the same argument as in Theorem 4 and 5. By Proposition 1, in total we need space

$$\begin{aligned} &O(hrs + h \log d + \log k) + O(ds \log k) + O(hrs + h \log d + d \log k) \\ =&O(hrs + h \log d + ds \log k) \end{aligned}$$

In order to use Lemma 10 we require

$$d\lceil (\log k)/r \rceil (2^r - 1) < 2^{rs} - 1 \tag{3}$$

Let $r = \lceil \log(d \log k) \rceil$ and $s = 2$. This will result in an algorithm using space

$$O(hrs + h \log d + ds \log k) = O(h \log(d \log k) + d \log k)$$

It remains to show (3), which we do by considering two cases. If $r \leq \log k$, then for sufficiently large $d \log k$,

$$d\lceil (\log k)/r \rceil (2^r - 1) \leq 4(d \log k)^2 / \log(d \log k) < 2^{\lceil 2 \log(d \log k) \rceil} - 1$$

Otherwise ($r > \log k$),

$$\begin{aligned} d\lceil (\log k)/r \rceil (2^r - 1) \leq& d2^r - 1 \\ =& 2^{\lceil \log(d \log k) \rceil + \log d} - 1 \\ \leq& 2^{2\lceil \log(d \log k) \rceil} - 1 \end{aligned}$$

□

6.3 Main Result

The input to $\mathsf{TreeEval}_{k,d,h}$ is of length $d^h \cdot k^d \log k$, and thus Theorem 7 gives us an algorithm using space $O(\log n \cdot \log \log n)$ for every setting of k, d, and h. As with Theorem 6, this also shows that some parameterizations of $\mathsf{TreeEval}_{k,d,h}$ are easy.

Theorem 8. *Any $\mathsf{TreeEval}_{k,d,h}$ instance with $d \geq (\log k)^{\Omega(1)}$ can be computed in* L.

Proof. Theorem 7 gives an algorithm for $\mathsf{TreeEval}_{k,d,h}$ which uses space $O(h \log(d \log k) + d \log k)$, which for $\log k \leq d^{O(1)}$ is at most $O(h \log d + d \log k) = O(\log(d^h \cdot k^d \log k))$. □

This immediately yields Theorem 2, which we state in a more quantitative form.

Theorem 9. *Assume Conjecture 1 holds. Then there exists a function in* L *which requires formulas of depth* $\Omega(\log^2 n/\log\log n)$.

Proof. Let $d = \Theta(\log n)$ and $h = \Theta(\log n/\log\log n)$ be such that $d^h \cdot 2^d = n$. Then by Theorem 8 we have $\mathsf{TreeEval}_{2,d,h} \in \mathsf{L}$, while Lemma 11 states that $\mathsf{TreeEval}_{2,d,h}$ requires depth $\Omega(dh) = \Omega(\log^2 n/\log\log n)$ as claimed. □

Theorem 9 applies to the strongest case of Conjecture 1, but as stated in the introduction, any weakening which implies that $\mathsf{TreeEval}_{2,d,h}$ requires superlogarithmic formula depth is sufficient, with the lower bound derived translating to one of equal asymptotics against L.

7 APPLICATION 2: NEAR-OPTIMAL CATALYTIC BRANCHING PROGRAMS

Our second contribution outside of TreeEval is to the study of catalytic branching programs for computing arbitrary functions.

7.1 Catalytic Branching Programs

7.1.1 Definitions and Motivation. We have thus far avoided discussing any syntactic space-bounded models except in passing. While we assume familiarity on the part of the reader with *branching programs* in the usual sense, to understand our second auxiliary result we must formally define the model of [15] now.

Definition 6. Let $n \in \mathbb{N}$ and let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary function. An *m-catalytic branching program* is a directed acyclic graph G with the following properties:

- There are m source nodes and $2m$ sink nodes.
- Every non-sink node is labeled with an input variable x_i for $i \in [n]$, and has two outgoing edges labeled 0 and 1.
- For every source node v there is one sink node labeled with $(v, 0)$ and one with $(v, 1)$.

We say that G *computes* f if for every $x \in \{0,1\}^n$ and source node v, the path defined by starting at v and following the edges labeled by the value of the x_i labeling each node ends at the sink labeled by $(v, f(x))$.

The *size* of G is the number of nodes in G. For this paper all branching programs will be *layered*, meaning all nodes are organized into groups, called layers, where all edges from layer i go to nodes in layer $i+1$ for all i. The *width* of G is the largest size of any layer, while the *length* of G is the number of layers.

The (logarithm of the) size of an ordinary branching program computing f non-uniformly corresponds to the space needed to compute f, as we need only remember where in the program we currently are. By contrast, the reader should think of the m-catalytic branching program model as providing some initial memory τ in the form of the label of some start node, and the (logarithm of the) size of the program is the space required to compute f while remembering this string τ.

Clearly this can be done with sm nodes, where s is the size of the smallest branching program for f, by simply taking m disjoint copies of an optimal branching program for f; we are interested in when this value can be reduced. This corresponds to using the space needed to store τ in a non-trivial way during the computation of f. This view also motivated Potechin [24] to alternately view catalytic branching programs as *amortized* branching programs, as we can think of taking these m disjoint branching programs for f and letting them share memory states, i.e. internal nodes, while still preserving the same disjoint source-sink behavior.

7.1.2 Past Results. In addition to characterizing m-catalytic branching programs as amortized branching programs, Potechin [24] showed that, given enough amortization, *every* function can be computed by branching programs of amortized linear size. Robere and Zuiddam [27] studied two different amortized branching program models, with one being catalytic branching programs, and concluded along with [24] that a linear upper bound holds; they also improved the amount of amortization needed for functions f that can be represented as low-degree $\mathbb{F}_2$ polynomials.

Later, Cook and Mertz [9] showed the results of [24, 27] can be captured by clean register programs. As with traditional space, clean register programs can utilize this initial memory τ as the setting of its registers at the beginning of the program, with the clean condition exactly giving back the pairing between source and sink nodes.

Proposition 12. *Let $f : \{0,1\}^n \to \{0,1\}$ be a function, let $\mathcal{F}$ be a finite field of characteristic p. Assume that there exists a register program P using t instructions—each of which only reads one input bit*[5]*—and s registers over $\mathcal{F}$, whose net result is to cleanly compute f into some register. Then f can be computed by an m-catalytic branching program of width $m \cdot p$ and length t, where $m = |\mathcal{F}|^s/p$.*

Proof. Each of the $|\mathcal{F}|^s$ nodes in a given layer will represent a unique setting to all the registers. We will execute one instruction of the register program per layer, querying the input bit corresponding to that instruction.

Finally, we will consider, for each source and sink node, the corresponding assignment to the designated output register. Find a basis $\{e_1 \dots e_r\}$ for $\mathcal{F}$ considered as a vector space over $\mathbb{F}_p$ such that e_1 is the field element $1 \in \mathcal{F}$. We delete all source nodes except those whose first coordinate is 0—leaving us with $|\mathcal{F}|^s/p$ source nodes as claimed—and similarly we delete all sink nodes except those whose corresponding assignment to the first coordinate is either 0 or 1. By construction, each source whose assignment is τ will reach the sink node labeled by the same τ, except that if $f(x_1, \dots, x_n) = 1$, then 1 is added to the output register, so that the its first coordinate is 1 instead of 0. □

In [24, 27], the amount of amortization required to achieve linear upper bounds was 2^{2^n} in the worst case. Using Proposition 12 plus the central TreeEval subroutines of [7, 8], [9] improved this to $2^{2^{\epsilon n}}$ for any $\epsilon > 0$. This is still the best known result for achieving linear amortized braching program size.

We also mention in passing that the m-catalytic branching programs produced by Proposition 12 can be made into *permutation*

[5]This is different from our earlier condition, given by Proposition 1, that each instruction be computable in small space. In non-uniform models we can compute any function of the current space in one step, but need to take careful account of the length as the exact number of variable reads.

branching programs—a classic and much more well-studied model—of the same width and length. In fact they are more restricted, and for example only have one accepting vertex; recently, Hoza, Pyne, and Vadhan [19] and Pyne and Vadhan [25] showed a lower bound against the read-once version of such programs for *infinite* width. See [9] for more discussion of the connections between these models and of how close to read-once our programs can be made.

7.2 One-Shot Clean Polynomials

Given our connection between register programs and m-catalytic branching programs, and the fact that Lemma 10 gives us a way to cleanly compute arbitrary polynomials, it seems natural to ask whether our techniques can improve the parameters of computing arbitrary functions using m-catalytic branching programs. This will require us to leave behind our strategy of using Lemma 10 in a recursive way, and instead apply it directly to the whole function f in question.

Using this idea to prove Theorem 3 will be the subject of the rest of the section; we will prove a more general, fine-grained version.

Theorem 10. *Let f be any function on n bits, and let r, s be positive integers such that*

$$\lceil n/r \rceil (2^r - 1) < 2^{rs} - 1$$

Then there exists an m-catalytic branching program of width $2m$ and length $2^{rs}n(1 + 2/r + 3/n)$ computing f, where $m \leq 2^{(n+2r)s}$.

Proof. Let $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$. We will group the input into groups of r bits, and encode each group of bits as an element of $\mathcal{F} = \mathbb{F}_{2^r}$. This grouping and encoding together define a function $g : \{0,1\}^n \to \mathcal{F}^{\lceil n/r \rceil}$, which will play the role of g in the statement of Lemma 10, with $a = \lceil n/r \rceil$. The program P_g (which cleanly computes g) can be implemented as a sequence of n instructions, reading each input once.

Applying Lemma 10 gives a register program of length

$$\begin{aligned}(|\mathcal{K}| - 1)(t(P_g) + 2a + b) &= (2^{rs} - 1)(n + 2\lceil n/r \rceil + 1) \\ &< 2^{rs} n(1 + 2/r + 3/n)\end{aligned}$$

which uses

$$a + b = \lceil n/r \rceil + 1$$

registers over $\mathcal{K}$. By Proposition 12, this gives us an m-catalytic branching program of length $2^{rs}n(1 + 2/r + 3/n)$ and width $2m$, where

$$m = |\mathcal{K}|^{\lceil n/r \rceil + 1}/2 = (2^{rs})^{\lceil n/r \rceil + 1}/2 < 2^{(n+2r)s}$$

Finally Lemma 10 requires $a(|\mathcal{F}| - 1) < |\mathcal{K}| - 1$; that is,

$$\lceil n/r \rceil (2^r - 1) < 2^{rs} - 1$$

which completes the proof. □

7.3 Main Result

Theorem 3 will follow by analyzing various parameter regimes from Theorem 10.

Proof of Theorem 3. We analyze three ways to choose r and s to satisfy the precondition of Theorem 10, each corresponding to one claim of the theorem.[6]

Constant s. Let s be any integer greater than 1. We consider two settings, $s = 2$ and $s \geq 3$. In the latter case, let

$$r = \left\lceil \frac{1}{s-1} \log n \right\rceil < \frac{1}{s-1} \log n + 1$$

Then the length of the program given by Theorem 10 is at most

$$\begin{aligned}2^{rs} n(1 + 2/r + 3/n) &\leq 2^s \cdot 2^{(s/(s-1)) \log n} \cdot n \cdot (1 + o(1)) \\ &= (2^s + o(1)) n^{(2s-1)/(s-1)}\end{aligned}$$

and for m we have

$$\begin{aligned}m &\leq 2^{(n+2r)s} \\ &< 2^{(n+2)s} \cdot n^{2s/(s-1)}\end{aligned}$$

Let $\epsilon = \frac{1}{s-1} \in (0, 1/2]$, so $s = 1 + 1/\epsilon$. This gives us length at most $(2^{1+1/\epsilon} + o(1)) \cdot n^{2+\epsilon} = O(n^{2+\epsilon})$ and m at most

$$2^{(n+2)(1+1/\epsilon)} n^{2(1+\epsilon)} < 2^{(1+1/\epsilon+o(1))n}$$

which gives us the first program of Theorem 3. Note that ϵ can be made arbitrarily small by increasing s.

For the second program, we move to the $s = 2$ case. Fix $r = \lceil \log n - \log\log n + 1 \rceil < \log n - \log\log n + 2$. Our length is at most

$$\begin{aligned}2^{rs} n(1 + 2/r + 3/n) &\leq 2^{2(\log n - \log\log n + 2)} n(1 + o(1)) \\ &= (16 + o(1)) \left(\frac{n^3}{\log^2 n} \right)\end{aligned}$$

while for m we have

$$\begin{aligned}m &\leq 2^{2(n+2r)} \\ &< 2^{2(n + 2\log n - 2\log\log n + 4)} \\ &= 2^8 \cdot 2^{2n} \left(\frac{n}{\log n} \right)^4\end{aligned}$$

Constant r. Let r be any integer greater than 1, and set

$$s = \left\lceil \frac{\log n - \log r}{r} + \frac{1}{n} \right\rceil + 1 < \frac{\log n - \log r}{r} + \frac{1}{n} + 2$$

Thus our length is at most

$$\begin{aligned}2^{rs} n(1 + 2/r + 3/n) &< 2^{r((\log n - \log r)/r + (1/n) + 2)} n(1 + 2/r + 3/n) \\ &= \frac{n}{r} \cdot 2^{r/n} \cdot 2^{2r} \cdot n \cdot (1 + 2/r + 3/n) \\ &\leq (1 + o(1)) \left(2^{2r} \left(\frac{1}{r} + \frac{2}{r^2} \right) n^2 \right) \\ &\leq (1 + o(1)) 2^{2r} n^2\end{aligned}$$

and for the width we get

$$\begin{aligned}m &< 2^{(n+2r)((\log n - \log r)/r + 1/n + 2)} \\ &\leq 2^{(n \log n)/r + n(2 - (\log r)/r + o(1))}\end{aligned}$$

Setting $\epsilon = 1/r$ gives us our third program—ϵ can be made arbitrarily small by increasing r—which completes the proof. □

[6] In what follows, all asymptotics $(O(), o())$ take n as the growing variable, with either r or s fixed and the other a function of n.

8 CONCLUSION

The most immediate question left open by this work is whether or not TreeEval ∈ L. Both answers are entirely possible, and it is no longer clear why one should be wholly convinced of either.

Similarly, we may take the chance to consider what answer we might expect on the KRW conjecture. We have stated Theorem 2 about the implications of composition theorems for formulas, but since our main theorem can and should be read as a failure of composition theorems in the space-bounded case, it is natural, possibly more so than before, to also believe that they could fail for formulas as well. Here one should read the contrapositive of Theorem 2 as giving a different angle: if one can show that deterministic uniform logspace has formulas of depth $o(\log^2 n/\log\log n)$—barely above the bound given by Savitch's Theorem [28] for non-deterministic non-uniform space—then the KRW conjecture falls in tandem.

There is also a broader question of how to apply our techniques to other problems in space-bounded complexity. The result of Lemma 10, of cleanly and efficiently computing arbitrary polynomials, seems to be a heavy hammer, but thus far it has only found a few nails.

Recently, Mertz [23] surveyed a number of techniques for space-bounded complexity, including the use of clean register programs seen in this and previous papers. The survey posed a host of open questions of how they can be further strengthened and applied, such as showing the power of *catalytic computing*. To take one example where our results may be relevant, they conjecture that an optimal improvement to Lemma 9 could also show that *catalytic logspace* contains NC^2. However, whether our more modest improvement in this paper can be useful in making progress on this or any other questions posed remains unknown.

ACKNOWLEDGMENTS

The authors would like to thank Robert Robere, Bruno Loff, and Manuel Stoeckl for many insightful discussions, as well as Igor Oliveira, Ninad Rajgopal, Pierre McKenzie, and the reviewers of ECCC and STOC for feedback on earlier drafts. The second author received support from the Royal Society University Research Fellowship URF\R1\191059 and from the Centre for Discrete Mathematics and its Applications (DIMAP) at the University of Warwick.

REFERENCES

[1] David A. Mix Barrington. 1989. Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1. *J. Comput. Syst. Sci.* 38, 1 (1989), 150–164. https://doi.org/10.1016/0022-0000(89)90037-8

[2] Michael Ben-Or and Richard Cleve. 1992. Computing Algebraic Formulas Using a Constant Number of Registers. *SIAM J. Comput.* 21, 1 (1992), 54–58. https://doi.org/10.1137/0221006

[3] Sagar Bisoyi, Krishnamoorthy Dinesh, and Jayalal Sarma. 2022. On pure space vs catalytic space. *Theor. Comput. Sci.* 921 (2022), 112–126. https://doi.org/10.1016/J.TCS.2022.04.005

[4] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. 2014. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014*. ACM, 857–866. https://doi.org/10.1145/2591796.2591874

[5] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. 2018. Catalytic Space: Non-determinism and Hierarchy. *Theory Comput. Syst.* 62, 1 (2018), 116–135. https://doi.org/10.1007/S00224-017-9784-7

[6] Arkadev Chattopadhyay, Yuval Filmus, Sajin Koroth, Or Meir, and Toniann Pitassi. 2021. Query-to-Communication Lifting Using Low-Discrepancy Gadgets. *SIAM J. Comput.* 50, 1 (2021), 171–210. https://doi.org/10.1137/19M1310153

[7] James Cook and Ian Mertz. 2020. Catalytic approaches to the tree evaluation problem. In *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing, STOC 2020*. ACM, 752–760. https://doi.org/10.1145/3357713.3384316

[8] James Cook and Ian Mertz. 2021. Encodings and the Tree Evaluation Problem. *Electron. Colloquium Comput. Complex.* (2021), 54. https://eccc.weizmann.ac.il/report/2021/054

[9] James Cook and Ian Mertz. 2022. Trading Time and Space in Catalytic Branching Programs. In *37th Computational Complexity Conference, CCC 2022 (LIPIcs, Vol. 234)*. 8:1–8:21. https://doi.org/10.4230/LIPIcs.CCC.2022.8

[10] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. 2012. Pebbles and Branching Programs for Tree Evaluation. *ACM Trans. Comput. Theory* 3, 2 (2012), 4:1–4:43. https://doi.org/10.1145/2077336.2077337

[11] Don Coppersmith and Edna K. Grossman. 1975. Generators for Certain Alternating Groups with Applications to Cryptography. *Siam Journal on Applied Mathematics* 29 (1975), 624–627. https://doi.org/10.1137/0129051

[12] Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. 2020. Randomized and Symmetric Catalytic Computation. In *CSR (Lecture Notes in Computer Science, Vol. 12159)*. Springer, 211–223. https://doi.org/10.1007/978-3-030-50026-9_15

[13] Susanna F. de Rezende, Or Meir, Jakob Nordström, Toniann Pitassi, and Robert Robere. 2020. KRW Composition Theorems via Lifting. In *FOCS*. IEEE, 43–49. https://doi.org/10.1109/FOCS46700.2020.00013

[14] Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. 2018. Hardness of Function Composition for Semantic Read once Branching Programs. In *33rd Computational Complexity Conference, CCC 2018 (LIPIcs, Vol. 102)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:22. https://doi.org/10.4230/LIPICS.ICALP.2016.36

[15] Vincent Girard, Michal Koucký, and Pierre McKenzie. 2015. Nonuniform catalytic space and the direct sum for space. *Electronic Colloquium on Computational Complexity (ECCC)* 138 (2015).

[16] Mika Göös, Toniann Pitassi, and Thomas Watson. 2018. Deterministic Communication vs. Partition Number. *SIAM J. Comput.* 47, 6 (2018), 2435–2450. https://doi.org/10.1137/16M1059369

[17] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. 2019. Unambiguous Catalytic Computation. In *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2019 (LIPIcs, Vol. 150)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:13. https://doi.org/10.4230/LIPIcs.FSTTCS.2019.16

[18] John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. 1977. On Time Versus Space. *J. ACM* 24, 2 (1977), 332–337. https://doi.org/10.1145/322003.322015

[19] William Hoza, Edward Pyne, and Salil Vadhan. 2021. Pseudorandom generators for unbounded-width permutation branching programs. In *12th Innovations in Theoretical Computer Science (ITCS'21) (LIPIcs)*. https://doi.org/10.4230/LIPIcs.ITCS.2021.7

[20] Kazuo Iwama and Atsuki Nagao. 2019. Read-Once Branching Programs for Tree Evaluation Problems. *ACM Trans. Comput. Theory* 11, 1 (2019), 5:1–5:12. https://doi.org/10.1145/3282433

[21] Mauricio Karchmer, Ran Raz, and Avi Wigderson. 1995. Super-Logarithmic Depth Lower Bounds Via the Direct Sum in Communication Complexity. *Comput. Complex.* 5, 3/4 (1995), 191–204. https://doi.org/10.1007/BF01206317

[22] David Liu. 2013. Pebbling Arguments for Tree Evaluation. *CoRR* abs/1311.0293 (2013). https://doi.org/10.48550/arXiv.1311.0293

[23] Ian Mertz. 2023. Reusing Space: Techniques and Open Problems. *B.EATCS* 141 (2023), 57–106.

[24] Aaron Potechin. 2017. A Note on Amortized Branching Program Complexity. In *Computational Complexity Conference (LIPIcs, Vol. 79)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:12. https://doi.org/10.4230/LIPIcs.CCC.2017.4

[25] Edward Pyne and Salil Vadhan. 2021. Pseudodistributions That Beat All Pseudorandom Generators (Extended Abstract). In *36th Computational Complexity Conference (CCC'21)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CCC.2021.33

[26] Ran Raz and Pierre McKenzie. 1999. Separation of the Monotone NC Hierarchy. *Comb.* 19, 3 (1999), 403–435. https://doi.org/10.1007/S004930050062

[27] Robert Robere and Jeroen Zuiddam. 2021. Amortized Circuit Complexity, Formal Complexity Measures, and Catalytic Algorithms. In *FOCS*. IEEE, 759–769. https://doi.org/10.1109/FOCS52979.2021.00079

[28] Walter J. Savitch. 1970. Relationships Between Nondeterministic and Deterministic Tape Complexities. *J. Comput. Syst. Sci.* 4, 2 (1970), 177–192. https://doi.org/10.1016/S0022-0000(70)80006-X

Received 12-NOV-2023; accepted 2024-02-11