Electronic Colloquium on Computational Complexity, Report No. 174 (2023)

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

James Cook
falsifian@falsifian.org

Ian Mertz
University of Warwick
ian.mertz@warwick.ac.uk

November 15, 2023

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 $\mathsf{L} \neq \mathsf{NC}^1$; 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 $\text{poly}(n)$ and width $2^{O(n)}$.

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 $\mathsf{SPACE}(t/\log t)$ [HPV77], and both containments are widely considered to be strict, but we have made little progress in proving this fact for any t.

1.1 The Tree Evaluation Problem

The *Tree Evaluation Problem* [CMW+12], 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

ISSN 1433-8092

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.

1.2 Hardness through composition

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 [CMW+12] to define $\mathsf{TreeEval}$ and conjecture that $\Omega(h \log k)$ space is optimal, a conjecture which has been backed up by multiple subsequent works. [Liu13, EMP18, IN19]

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 [KRW95] states that a similar logic holds for formula depth, with the upshot being that $\mathsf{TreeEval}$ 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. [dRMN+20, CFK+21]) as well as encouraging useful parallel lines of work such as *lifting theorems* [RM99, GPW18].

Thus the study of composition, and by extension $\mathsf{TreeEval}$, 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.

1.3 Upper bounds

Nevertheless, the consensus and central composition logic of the space hardness of $\mathsf{TreeEval}$ has faced a challenge ever since its inception. Buhrman, Cleve, Koucký, Loff, and Speelman [BCK+14] 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 [Bar89] and Ben-Or and Cleve [BC92], 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 [BKLS18,

GJST19, DGJ+20, BDS22], but while it was in part motivated to challenge the conjecture of [CMW+12], it did not immediately lead to any results about $\mathsf{TreeEval}$. However, after a period of quiet on both the upper and lower bound fronts, their objection was validated by Cook and Mertz [CM20, CM21], 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 [CMW+12] to separate L from P.

1.4 Main result

In this work we give an exponential improvement on the central subroutine of [CM20, CM21], which yields the following result.

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

Compared to having only a logarithmic factor improvement given by [CM20, CM21], 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.

1.5 Implications

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

1.5.1 The KRW Conjecture

First, we return to our brief discussion of the KRW conjecture, which we recall implies that $\mathsf{TreeEval} \notin \mathsf{NC}^1$. The results of [CM20, CM21] gave a space upper bound of $O(\log^2 n / \log\log n)$ for $\mathsf{TreeEval}$, asymptotically the same as the lower bound on formula depth implied by the KRW conjecture; thus it was possible that we could prove both the KRW conjecture and $\mathsf{L} = \mathsf{NC}^1$. This is no longer possible, as Theorem 1 makes these two hypotheses incompatible.

Theorem 2. *If the KRW Conjecture holds, then* $\mathsf{NC}^1 \neq \mathsf{L}$.

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 $\omega(\log n \cdot \log\log n)$ is sufficient for Theorem 2. The strongest (and most widely studied) version implies that L requires formulas of depth $\Omega(\log^2 n/\log^3 \log n)$, 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 gives a superpolynomial size separation between formulas and branching programs, even in the uniform setting. 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.

1.5.2 Amortized branching programs

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

This model, introduced by Girard, Koucký, and McKenzie [GKM15], essentially asks whether we can find smaller branching programs for computing an arbitrary function f if we only want to do so in an *average* sense. Potechin [Pot17] showed that this is possible in the strongest way: every function has amortized branching program size $O(n)$, where the amount of amortization needed is 2^{2^n}.

Reinterpreting and building on work of Potechin [Pot17] and an improvement by Robere and Zuiddam [RZ21], Cook and Mertz [CM22] used the TreeEval argument of [CM20, CM21] in the non-uniform setting to show that the amount of amortization can be reduced to $2^{O(2^{\epsilon n})}$ for arbitrarily small constant $\epsilon > 0$. By improving (a generalization of) the central subroutine of [CM20, CM21] 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, the following amortized branching program size upper bounds exist, where m is the amount of amortization needed:*

- *amortized size $O(n^{2+\epsilon})$ with $m = O(2^{(2+1/\epsilon)n})$, for arbitrarily small $\epsilon > 0$*
- *amortized size $O(n^3/\log^2 n)$ with $m = O(2^{(2+o(1))n})$*
- *amortized size $O(n^2)$ with $m = O(2^{(2+\log n)n})$*

2 Preliminaries

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

Our primary model of space will be that of *register programs*, a somewhat non-standard model formally introduced by Ben-Or and Cleve [BC92] based on work of Coppersmith and Grossman [CG75] and explored in a number of follow-up works [BCK+14, CM20, CM22].

Definition 1. A *register program* over ring $\mathcal{R}$ is a collection of memory locations $R = \{R_1 \dots R_s\}$, called *registers*, each of which can hold one element from $\mathcal{R}$, plus an ordered list of *instructions* in the form of updates to some register R_i based on the other registers.

Typically we will consider updates of the form

$$R_i \leftarrow R_i + p(R_1, \dots, R_{i-1}, R_{i+1}, \dots, R_s)$$

where p is any polynomial over $\mathcal{R}$.

Following [BCK+14], rather than directly writing their output to a register, our programs will *add* their output to a register while leaving other registers untouched, a process we call *clean* computation. This will be useful for making our algorithms space-efficient.

Definition 2. 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 $\tau_1, \dots, \tau_s \in \mathcal{R}$, if the program is run after initializing each register $R_i = \tau_i$, then at the end of the execution

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

In the register programs we consider, the instructions themselves will sometimes depend on the input. For example, a coefficient of the polynomial may be $[y_j = 1]$, meaning "this coefficient is 1 if input $y_j = 1$; otherwise the coefficient is 0". However, all our register programs, except in Section 7, will satisfy a uniformity condition:

Definition 3. A register program is *space c uniform* if there is an algorithm using space c which, given an instruction index i and the input to the program, performs the i-th instruction.

Thus when we describe an algorithm using a register program, the computation should be thought of as having two parts: (a) an "outer" program which designs a register program based on the input, and (b) an "inner" program which is that register program itself. The space used by our algorithms can be computed by tracking these procedures, although it can also be seen generically.

Proposition 4. *Let $c, s, t \in \mathbb{N}$, and let $\mathcal{R}$ be a ring. Let f be a Boolean function and let P be a space c uniform register program with s registers over $\mathcal{R}$ and which has t instructions in total, such that P cleanly computes f. Then f can be computed in space $O(\log t + s \log |\mathcal{R}| + c)$.*

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 5. *Every element of $\mathbb{F}_{p^a}$ can be represented in space $O(a \log p)$, and all field operations over $\mathbb{F}_{p^a}$ can be carried out in space $O(a \log p)$.*

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 6. *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 represention of $\mathbb{F}_{p^a}$ and $\mathbb{F}_{p^{ab}}$ that agree[1], so that in effect $\mathbb{F}_{p^a} \subseteq \mathbb{F}_{p^{ab}}$, and we treat it as such.

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 4. An element ω of a field $\mathcal{F}$ 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 7. *Every finite field $\mathcal{F}$ has a primitive root of unity of order $|\mathcal{F}| - 1$.*

This follows from the fact that the multiplicative group $\mathcal{F}^\times$ of a finite field is always a cycle. For $\mathcal{F} = \mathbb{F}_{p^a}$, 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 $\sum_{j=1}^{m} \omega_m^j = 0$.

Proposition 8. *Let ω_m be a primitive root of unity of order m. Then for all $0 < b < m$,*

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

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

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

So either $\omega_m^b = 1$ or $s = 0$, but the former is ruled out because ω_m is a *primitive* root of unity and $0 < b < m$. □

[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 $F_{p^a}[y]$.

Corollary 9. *Let $\mathcal{F}$ be a finite field and let $m = |\mathcal{F}| - 1$. Then there exist elements m^{-1}, ω_m in $\mathcal{F}$ such that for all $0 \leq b < m$,*

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

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

Proof. Let ω_m be a primitive root of unity of order m (Proposition 7). The case of $b \neq 0$ is handled by Proposition 8. For $b = 0$ we have that over F,

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

where the last equality holds because $m \equiv -1 \pmod{p}$. To complete the proof, take $m^{-1} = -1$ so $m^{-1}m = 1$. □

4 Main result: Tree Evaluation in low space

We now move on to the main goal of our paper, which is to prove Theorem 1. The following statement 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 10. *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. In order to use the tools from Corollary 9, let $\mathcal{K}$ be a field, the necessary properties of which we will uncover, and let $m = |\mathcal{K}| - 1$.

Lemma 11. *Let $d < m$, and let τ_i, x_i be elements of $\mathcal{K}$ for $i \in [d]$. Then*

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

where m^{-1}, ω_m are given by Corollary 9.

Before going into the proof of Lemma 11, we should stress why it is useful. Our overall goal is to compute the function f_u at node u in our $\mathsf{TreeEval}$ 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 11 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 inner product on the left hand side, we get

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

If we sum over all j and multiply both sides by m^{-1}, then by switching the sums we get

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

By Corollary 9 we have

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

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

$$m^{-1}\sum_{j=1}^{m}\prod_{i=1}^{d}(\omega_m^j\tau_i + x_i) = \left(\prod_{i\in\varnothing}\tau_i\right)\left(\prod_{i\in[d]\setminus\varnothing}x_i\right) = \prod_{i\in[d]}x_i \qquad \square$$

Thus the next step is to move from individual products to polynomials. This is accomplished by a simple corollary of Lemma 11.

Lemma 12. *Let $d, m, \mathcal{K}$ be such that $|\mathcal{K}| - 1 = m > d$, let $p : \mathcal{K}^n \to \mathcal{K}$ be a degree-d polynomial, and let τ_i, x_i be elements of $\mathcal{K}$ for $i \in [n]$. Then*

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

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 $\omega_m^j\tau_i + x_i$ for each y_i and summing over all j, Lemma 11 gives

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

$$= \sum_{\substack{I \subseteq [n] \\ |I| \leq d}} c_I \cdot m^{-1} \sum_{j=1}^{m} \prod_{i \in I} (\omega_m^j \tau_i + x_i)$$
$$= \sum_{\substack{I \subseteq [n] \\ |I| \leq d}} c_I \prod_{i \in I} x_i$$

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

Finally, we show how to use Lemma 12 in a register program to compute our polynomial f_u in the way we described above, given an appropriate choice of $\mathcal{K}$ and m.

Lemma 13. *Let $m, \mathcal{K}$ be such that $|\mathcal{K}| - 1 = m > 2\lceil \log k \rceil$. 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 12 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 $\prod_{i=1}^{\lceil \log k \rceil}(1 - y_i + (2y_i - 1)\beta_i)$, which equals $[y = \beta]$ when all $y_i \in \{0,1\}$, and making a similar substitution for $[z = \gamma]$. This gives the polynomial

$$\sum_{\substack{\alpha,\beta,\gamma \in [k]^3 \\ \alpha_i = 1}} [f_u(\beta,\gamma) = \alpha] \prod_{j=1}^{\lceil \log k \rceil} (1 - \beta_j + (2\beta_j - 1)y_j) \cdot (1 - \gamma_j + (2\gamma_j - 1)z_j)$$

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 $q_{u,i}$, our register program will compute the left hand side of Lemma 12, using one round of recursive calls and basic instructions per term of the outer sum. For any tuple—of registers, polynomials, etc.—we use the subscript i to denote the ith item.

1: **for** $j = 1, \ldots, m$ **do**
2: **for** $c \in \{\ell, r\}, i = 1 \ldots \lceil \log k \rceil$ **do**
3: $R_{c,i} \leftarrow \omega_m^j \cdot R_{c,i}$
4: P_ℓ, P_r
5: **for** $i = 1 \ldots \lceil \log k \rceil$ **do**

6: $R_{v,i} \leftarrow R_{v,i} + m^{-1} \cdot q_{u,i}(R_\ell, R_r)$
7: P_ℓ^{-1}, P_r^{-1}
8: **for** $c \in \{\ell, r\}, i = 1 \dots \lceil \log k \rceil$ **do**
9: $R_{c,i} \leftarrow \omega_m^{-j} \cdot R_{c,i}$

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

In each iteration of the loop we have

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

and m is larger than the degree of each $q_{u,i}$, and so correctness follows from Lemma 12 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 14 in Section 5, but even as stated Lemma 13 is sufficient to serve as our main TreeEval subroutine.

Proof of Theorem 10. We will show that our $\mathsf{TreeEval}_{k,h}$ instance can be cleanly computed by a register program of length $(4|\mathcal{K}|)^h \lceil \log k \rceil$ and using $3\lceil \log k \rceil$ registers over $\mathcal{K}$, where every instruction is computable in space $O(|\mathcal{K}|)$. By Proposition 4, our space usage will ultimately be

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

which is $O((h + \log k) \log \log k)$ by Proposition 5 if we pick $\mathcal{K}$ to be a field of prime power size at most $O(\log k)$.

We build our register program by induction, showing that for every 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, the output can be computed by reading the leaf node directly, which gives a register program of length

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

each instruction of which can be computed in space $O(\log |\mathcal{K}|)$.

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 13 to u, inductively giving us a program of length

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

where each instruction can be computed in space $O(\log k)$ because we assume $|\mathcal{K}| = O(\log k)$.

This completes the recursion. Finally we choose $\mathcal{K} = \mathbb{F}_{2^{\lceil \log(2\lceil \log k \rceil + 2) \rceil}}$. Our two conditions are thus satisfied: 1) $\mathcal{K}$ has size $O(\log k)$, ensuring efficiency; and 2) $|\mathcal{K}| - 1 > 2\lceil \log k \rceil$, ensuring correctness. □

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 13, which is our main subroutine, in a more general and efficient form.

Lemma 14. *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 $R_1, \ldots, R_a$.*

Then there exists a register program P_f which cleanly computes $f(g)$ into registers $R_{a+1}, \ldots, R_{a+b}$. 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 13 as a special case[2] of Lemma 14, 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 14 saves some time by avoiding the need to call the inverse program P_g^{-1}.

The proof is essentially that of Lemma 13, 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 15. *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.

Theorem 16. *Any $\mathsf{TreeEval}_{k,h}$ instance such that $h \leq \log k / \log \log k$ can be computed in L.*

Another consequence is that when converting 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.

Proof of Theorem 15. The proof is similar to the proof of Theorem 10, except that instead of representing elements of $[k]$ in binary, we represent them as tuples of field elements for some field $\mathcal{F} \subseteq \mathcal{K}$. The field $\mathcal{K}$ will be larger than in the proof of Theorem 10, and the register program we produce will be longer, but it will use fewer registers and need less space in total to store those registers.

Let $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$ where r and s will be determined later. By Proposition 6 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 $\mathcal{K}$.

[2]Strictly speaking, it is not a special case, since Lemma 13 encodes values as bit strings (meaning $\mathcal{F} = \mathbb{F}_2$ in terms of Lemma 14) but does not require $\mathbb{F}_2$ to be a subfield of $\mathcal{K}$.

As before, we can show by induction that for any node u of height $h' \leq h$ there is a register program over $\mathcal{K}$ that cleanly computes the encoding of f_u into $\lceil \log k/r \rceil$ registers. The program will have length $(2|\mathcal{K}|)^{h'-1} \operatorname{poly}(k)$ and use $3\lceil \log k/\log r \rceil$ registers.

The induction proof, after converting f_u into polynomials $q_{u,i}$ as in the proof of Lemma 13, is the same as for Theorem 10, except that instead of Lemma 13, we invoke Lemma 14 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,.

Now we are ready to choose our fields $\mathcal{F} = \mathbb{F}_{2^r}$ 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. The register program has length $|\mathcal{K}|^{O(h)} \operatorname{poly}(k)$, so we need

$$\log((2^{rs})^{O(h)} \operatorname{poly}(k)) = O(hrs + \log k)$$

space to track our position in the program. In total, then, we need space

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

In order for the algorithm to work, Lemma 14 requires that $a(|\mathcal{F}| - 1) \leq |\mathcal{K}| - 1$ where $a = 2\lceil \log k/r \rceil, |\mathcal{F}| = 2^r, |\mathcal{K}| = 2^{rs}$. In order words, we require $2\lceil \log k/r \rceil (2^r - 1) \leq 2^{rs} - 1$. Choosing $r = \lceil \log \log k \rceil$ and then setting

$$\begin{aligned} s &= \left\lceil \frac{1}{r} \log \left(2 \left\lceil \frac{\log k}{r} \right\rceil (2^r - 1) + 1 \right) \right\rceil \\ &= \left\lceil \frac{1}{\log \log k} \log \left(O\left(\frac{\log k}{\log \log k} \right) \log k \right) \right\rceil \\ &= O\left(\frac{\log \log k}{\log \log k} \right) = O(1) \end{aligned}$$

results 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 14, as it will prove to be stronger and more flexible than Lemma 13 as seen above. To end this section we will prove it, with a proof closely mirroring that of Lemma 13.

Proof of Lemma 14. For each $i = 1, \ldots, b$ we define a polynomial $p_i(y_1, \ldots, y_n)$ which computes the i-th coordinate of $f(y_1, \ldots, y_a)$. Our inspiration will be the formula

$$f_i(y_1, \ldots, y_n) = \sum_{(y'_1, \ldots, y'_a) \in \mathcal{F}^a} f(y'_1, \ldots, y'_a) \prod_{i=1}^{a} [y_i = y'_i]$$

To make this a polynomial, we replace each indicator function $[y_i = z]$ with the polynomial

$$q_{i,z}(y_1, \ldots, y_a) = 1 - (y_i - z)^{|\mathcal{F}|-1}$$

By Fermat's little theorem, $q_{i,z}(y_1, \ldots, y_a) = [y_i = z]$ for any $y_i, z \in \mathcal{F}$. Define

$$p_i(y_1, \ldots, y_n) = \sum_{(y'_1, \ldots, y'_a) \in \mathcal{F}^a} f(y'_1, \ldots, y'_a) \prod_{i=1}^{a} q_{i,y'_i}(y_1, \ldots, y_n) \tag{1}$$

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

Now let $m = |\mathcal{K}| - 1$ and let ω_m be a primitive root of unity of order m (Proposition 7). By assumption, $a(|\mathcal{F}| - 1) < |\mathcal{K}| - 1$, so m is greater than the degree of the polynomials p_i. Let $\tau_i \in \mathcal{K}$ be the initial value of each register R_i. By Lemma 12,

$$\sum_{j=1}^{m} m^{-1} p_i(\omega_m^j \tau_1 + y_1, \ldots, \omega_m^j \tau_a + y_a) = p_i(y_1, \ldots, y_a)$$

This leads to the following algorithm. It replaces the inefficient warm-up version presented in the proof of Lemma 13 which required an extra m copies of P_g^{-1}.

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

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

To analyse this algorithm, define $\tau'_i = \tau_i - g_i$ for $i = 1, \ldots, a$. At the start of the i-th iteration of the loop, the following invariants hold for $j \in [a], k \in [b]$:

$$\begin{aligned} R_j &= \omega_m^{i-1} \tau'_j + g_j \\ R_{a+k} &= \tau_{a+k} + \sum_{i'=1}^{i} m^{-1} p_i(\omega_m^{i'} \tau'_1 + g_1, \ldots, \omega_m^{i'} \tau'_a + g_a) \end{aligned}$$

It is straightforward to verify this invariant holds after each iteration. After the last iteration, Lemma 12 tells us that the output registers $R_{a+1}, \ldots, R_{a+k}$ hold the correct values, and the first a registers are restored to $R_j = \omega_m^m \tau'_j + g_j = \tau_j$.

This register program includes m copies of P_g and has a total length of $m(2a + b + t(P_g))$. Each of the $2a$ instructions on lines 2 and 4 can be executed in $O(\log |\mathcal{K}|)$ space, and using (1), line 5 can be executed in space $c_2 + O(a \log |\mathcal{F}| + \log |\mathcal{K}|)$ where c_2 is the space needed to compute f. □

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 [KRW95])**.** *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*

$$g_1 \circ g_2(x_{11} \dots x_{n_1 n_2}) := g_1(g_2(x_{11} \dots x_{1n_2}) \dots g_2(x_{n_1 1} \dots x_{n_1 n_2}))$$

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 $\mathsf{TreeEval}$, we need to consider the unbounded fan-in version of $\mathsf{TreeEval}$. 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. This version has input size $n = d^h k^d \log k$, and fixing $k = 2$ gives us $\log n = O(h \log d + d)$.

Lemma 17. *The KRW Conjecture implies $depth(\mathsf{TreeEval}_{2,d,h}) = \Omega(dh)$, which implies $\mathsf{TreeEval}_{2,d,h} \notin \mathsf{NC}^1$ for $dh = \omega(\log n)$.*

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 $\Omega(dh)$. □

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

Using Lemma 14, we can generalize Theorem 1 to degrees d other than 2:

Theorem 18. *Any $\mathsf{TreeEval}_{k,d,h}$ instance can be computed in space $O((h+d \log k) \log(d \log k))$.*

Note that the input to $\mathsf{TreeEval}_{k,d,h}$ is of length $d^h \cdot k^d \log k$, and thus Theorem 18 gives us an algorithm using space $O(\log n \cdot \log \log n)$ for every setting of k, d, and h.

Proof. The proof is the same as that of Theorem 1, using Lemma 14 but with $\mathcal{F} = \mathbb{F}_2$ as in Lemma 13. We will use the generalized fan-in to set $a = d\lceil \log k \rceil$, with the input program P_g being the concatenation of all d register programs $P_1 \dots P_d$, where P_j cleanly computes the jth child of v into a separate input register R_j. Furthermore we set $\mathcal{K} = \mathbb{F}_{2^r}$ as before, but now we fix

$$r = \lceil \log(d\lceil \log k \rceil + 2) \rceil$$

The result is a register program of length $(d|\mathcal{K}|)^h \operatorname{poly}(k)$ using $(d+1)\lceil \log k \rceil$ registers over $\mathcal{K}$, which by Proposition 4 puts $\mathsf{TreeEval}_{k,d,h}$ in space

$$h(\log d + r) + \log k + (d+1)(\log k)r$$

which, for $r = O(\log(d \log k))$ is

$$O((h + (d \log k)) \log(d \log k)) \qquad \square$$

This is all we need to prove Theorem 2.

Proof of Theorem 2. Assume for contradiction that $\mathsf{L} = \mathsf{NC}^1$. We will show the KRW conjecture does not hold.

Consider an instance of $\mathsf{TreeEval}_{2,d,h}$ which we pad with $2^{(h+d)\log d}$ zeroes. By Theorem 18 this problem can be solved in logarithmic space, and so by the assumption that $\mathsf{L} = \mathsf{NC}^1$ this gives a formula of depth $O((h+d) \cdot \log d)$ for $\mathsf{TreeEval}_{2,d,h}$, which is $o(dh)$ for $d = \omega(1)$ and $h = \omega(\log d)$.

However, by Lemma 17, the KRW conjecture implies that $\mathsf{TreeEval}_{2,d,h}$ requires depth $\Omega(dh)$ for all values of d and h, which is a contradiction. $\square$

In fact, this proof gives a near-optimal separation between formulas and branching programs, conditioned on the KRW conjecture being true.

Theorem 19. *Assume Conjecture 1 holds. Then there exists a function f on n inputs which can be computed by branching programs of size* $\operatorname{poly}(n)$ *but requires formula depth* $\Omega(\log^2 n / \log^3 \log n)$.

Proof. Consider the padded version of $\mathsf{TreeEval}_{2,d,h}$ in the proof of Theorem 2, and fix $d = \log n$ and $h = \log n / \log \log n$ for n being the original input size of $\mathsf{TreeEval}_{2,d,h}$ before padding. Thus our new input size is

$$N = 2^{(h+d)\log d} = 2^{O(\log n \cdot \log \log n)}$$

and this function is computable in space $\operatorname{poly}(N)$. As shown in Lemma 17, Conjecture 1 implies that our formula depth is at least

$$\Omega(dh) = \Omega\left(\frac{\log^2 n}{\log \log n}\right) = \Omega\left(\frac{(\log N / \log \log N)^2}{\log(\log N / \log \log N)}\right) = \Omega\left(\frac{\log^2 N}{\log^3 \log N}\right) \qquad \square$$

7 Application 2: Near-optimal amortized branching programs

Our second contribution outside of $\mathsf{TreeEval}$ is to the study of amortized/catalytic branching programs for computing arbitrary functions.

7.1 Amortized (or 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 define the model of [GKM15] now.

Definition 5. 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 [Pot17] 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 [Pot17] showed that, given enough amortization, *every* function can be computed by branching programs of amortized linear size. Robere and Zuiddam [RZ21]

studied two different amortized branching program models, with one being catalytic branching programs, and concluded along with [Pot17] 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 [CM22] showed the results of [Pot17, RZ21] 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 20. *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*[3]*—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 $\{e_1, \ldots, e_r\}$ 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 $f(x_1, \ldots, x_n) = 1$, then 1 is added to the output register, so that the its first coordinate is 1 instead of 0. □

In [Pot17, RZ21], the amount of amortization required to achieve linear upper bounds was 2^{2^n} in the worst case. Using Proposition 20 plus the central TreeEval subroutines of [CM20, CM21], [CM22] 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 20 can be made into *permutation 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 [HPV21] and Pyne and Vadhan [PV21] showed a lower bound against the read-once version of such programs for *infinite* width. See [CM22] 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 14 gives us a way to cleanly compute arbitrary polynomials, it seems

[3]This is different from our earlier condition that each instruction be computable in small space. In non-uniform models we can compute any function of the current space in one step.

natural to ask whether our techniques can improve the parameters of computing arbitrary functions using m-catalytic branching programs. 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 21. *Let f be any function on n bits, and let r, s be positive integers such that*

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

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. Like in the proof of Theorem 15, 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 14, 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 14 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 20, 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 14 requires $a(|\mathcal{F}| - 1) < |\mathcal{K}| - 1$; that is,

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

which completes the proof. □

Proof of Theorem 3. We analyze three ways to choose r and s to satisfy (2) corresponding to the claims of the theorem.[4]

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

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

Then (2) is satisfied for sufficiently large n. Our length is less than

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

[4]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.

$$= (2 + o(1))n^{\frac{2s-1}{s-1}}$$

and for m have

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

We consider two settings, $s = 2$ and $s \geq 3$. In the latter case, we can set $\epsilon = \frac{1}{s-1} \in (0, 1]$ so $s = 1 + 1/\epsilon$, which gives us length at most $O(n^{2+\epsilon})$ and m at most

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

which gives us the first program of Theorem 3.

For the second program, we move to the $s = 2$ case. Fix $s = 2$ and $r = \lceil \log n - \log\log n + 1 \rceil < \log n - \log\log n + 2$. Then (2) is satisfied for sufficiently large n. Our length is less than

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

while for m we have

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

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

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

Then (2) is satisfied for sufficiently large n. Our length is less than

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

and for $r > 1$ this leaves us with

$$\begin{aligned} m &< 2^{(n+2r)((\log n - \log r)/r + 1/n + 2)} \\ &\leq \left(\frac{n}{r}\right)^{n/r} \cdot 2 \cdot 2^{2n} \cdot \left(\frac{n}{r}\right)^2 \cdot 2^{2r/n} \cdot 2^{4r} \\ &= O\left(4^{n+r}\left(\frac{n}{r}\right)^{n/r+2}\right) \\ &< O(4^n n^n) \end{aligned}$$

which gives us our third program and thus completes the proof. □

8 Conclusion

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

There is also a broader question of how to apply our techniques to other problems in space-bounded complexity. The result of Lemma 14, 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 [Mer23] 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 13 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.

Acknowledgements

The authors would like to thank Robert Robere and Bruno Loff for many insightful discussions, as well as Igor Oliveira, Ninad Rajgopal, Pierre McKenzie, and the editors of ECCC for feedback on the initial draft. 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

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1. *J. Comput. Syst. Sci.*, 38(1):150–164, 1989.

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

[BCK+14] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014*, pages 857–866. ACM, 2014.

[BDS22] Sagar Bisoyi, Krishnamoorthy Dinesh, and Jayalal Sarma. On pure space vs catalytic space. *Theor. Comput. Sci.*, 921:112–126, 2022.

[BKLS18] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic space: Non-determinism and hierarchy. *Theory Comput. Syst.*, 62(1):116–135, 2018.

[CFK+21] Arkadev Chattopadhyay, Yuval Filmus, Sajin Koroth, Or Meir, and Toniann Pitassi. Query-to-communication lifting using low-discrepancy gadgets. *SIAM J. Comput.*, 50(1):171–210, 2021.

[CG75] Don Coppersmith and Edna K. Grossman. Generators for certain alternating groups with applications to cryptography. *Siam Journal on Applied Mathematics*, 29:624–627, 1975.

[CM20] James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing, STOC 2020*, pages 752–760. ACM, 2020.

[CM21] James Cook and Ian Mertz. Encodings and the tree evaluation problem. *Electron. Colloquium Comput. Complex.*, page 54, 2021. URL: https://eccc.weizmann.ac.il/report/2021/054.

[CM22] James Cook and Ian Mertz. Trading time and space in catalytic branching programs. In *37th Computational Complexity Conference, CCC 2022*, volume 234 of *LIPIcs*, pages 8:1–8:21, 2022.

[CMW+12] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, 2012.

[DGJ+20] Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Randomized and symmetric catalytic computation. In *CSR*, volume 12159 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2020.

[dRMN+20] Susanna F. de Rezende, Or Meir, Jakob Nordström, Toniann Pitassi, and Robert Robere. KRW composition theorems via lifting. In *FOCS*, pages 43–49. IEEE, 2020.

[EMP18] Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. Hardness of function composition for semantic read once branching programs. In *33rd Computational Complexity Conference, CCC 2018*, volume 102 of *LIPIcs*, pages 15:1–15:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[GJST19] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Unambiguous catalytic computation. In *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2019*, volume 150 of *LIPIcs*, pages 16:1–16:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[GKM15] Vincent Girard, Michal Koucký, and Pierre McKenzie. Nonuniform catalytic space and the direct sum for space. *Electronic Colloquium on Computational Complexity (ECCC)*, 138, 2015.

[GPW18] Mika Göös, Toniann Pitassi, and Thomas Watson. Deterministic communication vs. partition number. *SIAM J. Comput.*, 47(6):2435–2450, 2018.

[HPV77] John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space. *J. ACM*, 24(2):332–337, 1977.

[HPV21] William Hoza, Edward Pyne, and Salil Vadhan. Pseudorandom generators for unbounded-width permutation branching programs. In *12th Innovations in Theoretical Computer Science (ITCS'21)*, LIPIcs, 2021.

[IN19] Kazuo Iwama and Atsuki Nagao. Read-once branching programs for tree evaluation problems. *ACM Trans. Comput. Theory*, 11(1):5:1–5:12, 2019.

[KRW95] Mauricio Karchmer, Ran Raz, and Avi Wigderson. Super-logarithmic depth lower bounds via the direct sum in communication complexity. *Comput. Complex.*, 5(3/4):191–204, 1995.

[Liu13] David Liu. Pebbling arguments for tree evaluation. *CoRR*, abs/1311.0293, 2013.

[Mer23] Ian Mertz. Reusing space: Techniques and open problems. *B.EATCS*, 141:57–106, 2023.

[Pot17] Aaron Potechin. A note on amortized branching program complexity. In *Computational Complexity Conference*, volume 79 of *LIPIcs*, pages 4:1–4:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[PV21] Edward Pyne and Salil Vadhan. Pseudodistributions that beat all pseudorandom generators (extended abstract). In *36th Computational Complexity Conference (CCC'21)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[RM99] Ran Raz and Pierre McKenzie. Separation of the monotone NC hierarchy. *Comb.*, 19(3):403–435, 1999.

[RZ21] Robert Robere and Jeroen Zuiddam. Amortized circuit complexity, formal complexity measures, and catalytic algorithms. In *FOCS*, pages 759–769. IEEE, 2021.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il