TREE EVALUATION IS IN SPACE O(LOG N · LOG LOG N)*

JAMES COOK† AND IAN MERTZ‡

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 $\text{poly}(n)$ 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. Tree Evaluation and composition.

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[1] 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 $2^h \cdot \text{poly}(k)$. 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 $\text{poly}(k)$ 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

*A previous version of this work appeared in STOC'24 as [CM24].

†Independent Researcher (falsifian@falsifian.org)

‡University of Warwick (ian.mertz@warwick.ac.uk)

[1]While the case of binary nodes is the most commonly studied, TreeEval was originally defined for trees with a third parameter, typically labeled d, for the fan-in of internal nodes. We discuss this case and its relationship to lower bounds for other computational models in Section 6.

giving an algorithm using only $O(h + \log k)$ bits of memory.

Thus if our intuition is correct, this should be a problem that is in P but not L. This led Cook, McKenzie, Wehr, Braverman, and Santhanam [CMW+12] to define $\mathsf{TreeEval}$ and hypothesize that $\Omega(h \log k)$ space is optimal, leading them to conjecture that $\mathsf{TreeEval}$ separates L—in fact, even NL—from P. The hypothesis was supported by multiple subsequent works, which showed it holds in restricted, but also non-uniform, settings such as *thrifty* algorithms [CMW+12]—a $\mathsf{TreeEval}$-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* programs [EMP18]. Later works extended both of these results to the non-deterministic setting [Liu13, IN19].

This idea, of using what is known as *composition* or *direct product theorems* to prove strong lower bounds from weak lower bounds, 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.

1.2. Known 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, Pyn24, CLMP25, GJST24, FMST25, KMPS25, PSW25, BDRS24, AM25, BFM+25, AFM+25], but while it was in part motivated to challenge the hypothesis 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 straightforward $O(\log^2 n)$ space algorithm.

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 [CMW+12] proposed to separate L from P.

1.3. Our results. We now move to discussing the contributions of this paper. Because there are many different theorems pertaining to specific variants of $\mathsf{TreeEval}$ and implications therein, a full list of theorems appears in Appendix C to help readers compare and keep track.

1.3.1. Main result. Our primary contribution is to give an exponential improvement on the central subroutine of [CM20, CM21], which yields the following result.

THEOREM 1.1. 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 that $\mathsf{TreeEval} \in \mathsf{L}$.

We describe our algorithm as a *register program* over a finite field. We define these in Section 2. In Section 3, we discuss roots of unity over finite fields, which are a fundamental part of our technique (though Goldreich [Gol24a] later showed it can be reframed in terms of polynomial interpolation). We prove Theorem 1.1 in Section 4. In Section 5, we improve and generalize our main subroutine and discuss what that implies.

As observed in [CM20, CM21], 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.

1.3.2. Implications. Our improvement has immediate consequences outside of studying space upper bounds on TreeEval. All models and statements will be formally defined in Sections 5, 6, and 7.

Other parameterizations of TreeEval. As an extension of Theorem 1.1, we study Tree Evaluation in settings besides general $\mathsf{TreeEval}_{k,h}$. First, in Section 5 we show that our results can be improved to logspace for low height instances of TreeEval.

THEOREM 1.2. *Let* $h := h(k)$ *be such that* $h = O(\frac{\log k}{\log\log k})$. *Then* $\mathsf{TreeEval}_{k,h}$ *can be computed in* L.

In Section 6 we move to the more general setting of $\mathsf{TreeEval}_{k,d,h}$, where we additionally vary the *fan-in* d of internal nodes. In this case we extend our results for fan-in 2, showing an $O(\log n \cdot \log\log n)$ space upper bound as before.

THEOREM 1.3. $\mathsf{TreeEval}_{k,d,h}$ *can be computed in space* $O(h\log(d\log k) + d\log k)$.

We also show a similar result to Theorem 1.2 for the case of large d, and in Appendix B (see discussion below) we see a third version for the case of small k. Together these results suggest that the traditionally studied case, i.e. fan-in two trees where both $\log k$ and h are approximately $\Theta(\log n)$, are the most likely to yield space lower bounds.

The KRW conjecture. Moving beyond space bounds for TreeEval, we return to our brief discussion of the KRW conjecture, which implies that $\mathsf{TreeEval} \notin \mathsf{NC}^1$. [CM20, CM21] 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 1.1 makes these two hypotheses incompatible.

THEOREM 1.4. *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 1.4 is quite robust with respect to choosing

weaker versions of the conjecture; any statement that implies TreeEval with alphabet size $k = 2$ requires formula depth $\omega(\log n)$ is sufficient for Theorem 1.4. 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 $\mathsf{P} \not\subseteq \mathsf{NC}^1$. Second, the KRW conjecture would give a superpolynomial size separation between *non-uniform* formulas and *uniform* branching programs; no such 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 e.g. st-connectivity, even in the undirected case. And fourth, and most philosophically, is that any continued belief in the KRW conjecture now represents a bet that the ability to handle composition is the factor that separates space and formulas.

Catalytic branching programs. For our last result, we consider the question of *catalytic* branching program size, also called *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 follows a path which starts at the source and follows edges according to the results of querying one bit of x at each node encountered, until it reaches a sink, which must be labeled with $f(x)$.

Drawing a connection to the catalytic space model as discussed above, Girard, Koucký, and McKenzie [GKM15] 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. In their model, we consider a DAG with m source nodes and $2m$ sink nodes, one for each (source, output) pair, and require that running the program from source i on input x leads to the unique sink labeled $(i, f(x))$.

While one obvious solution is to simply have m disjoint copies of the optimal branching program for f, the question is whether a different program, one which is not so disjoint in the interior of the 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 [Pot17] 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 [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 $m = 2^{2^{\epsilon n}}$ for arbitrarily small constants $\epsilon > 0$.

By improving (a generalization of) the central subroutine of [CM20, CM21] in Theorem 1.1, and applying the framework of [CM22], we show that a slight sacrifice in the length gives a near-optimal improvement in the amount of amortization.

THEOREM 1.5. *Every function $f : \{0,1\}^n \to \{0,1\}$ has m-catalytic branching programs of the following size:*

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

- *size* $O(m \cdot n^2)$ *with* $m = O(2^{(2+\epsilon \log n)n})$*, for any constant* $\epsilon > 0$

Theorem 1.5 may be compared to the best possible[2] amortized size of $\Theta(n)$ (achieved in [Pot17, RZ21, CM22]), and an overall size lower bound of $\Omega(2^n/n)$ by the same counting argument as for ordinary branching programs.

1.4. Follow-up work. Since the original publication of this paper [CM24], there have been two improvements to our main result, one qualitative and one quantitative. We discuss both results, in turn, in the appendices to this paper.

First, Goldreich [Gol24a] observed that our main technique is a specific instance of computing a degree d polynomial at a single point via interpolation given by $d+1$ evaluations along a line. In Appendix A we discuss this framework and how it indicates a barrier to directly improving our main subroutines. They also showed that our approach fits neatly into a model known as *global storage* [Gol08], although we will not pursue this further.

The second result, due to Stoeckl [Sto23] and later Goldreich [Gol24b], improves Theorem 1.1, showing that $\mathsf{TreeEval}$ can be computed in space $O(\log n \cdot \log\log n / \log\log\log n)$. Their proof goes via balancing parameters of the tree, including the fan-in of each node (see Section 6), in order to exploit some slackness in our memory usage; as a byproduct of their proof, they also show that $\mathsf{TreeEval}$ is in logspace whenever the alphabet size k is sufficiently small. In Appendix B, we present this result, and discuss how it circumvents the barrier previously mentioned, giving hope for further improvements.

We also mention a major consequence of our work, which came to light after our original publication. Using a generalization of Theorem 1.3, Williams [Wil25] showed that any multi-tape Turing machine running in time t can be simulated by a machine which only uses space $O(\sqrt{t \log t})$, the first quantitative improvement to the result of Hopcroft, Paul, and Valiant [HPV77] in fifty years. As a consequence, they also obtain a novel separation between space and time, showing that $\mathsf{SPACE}[s] \not\subseteq \mathsf{TIME}[s^2/\operatorname{poly}\log s]$. This work also gives a tigher connection between branching programs and circuits, albeit with some quantitative loss that was removed in subsequent work by Shalunov [Sha25].

1.5. Updates from [CM24]. Besides our two new appendices based on follow-up work to the conference version of this paper [CM24], there were two other changes required, which we briefly discuss now. First, our previous proof of Theorem 1.3 gets the wrong asymptotics when the alphabet size k is negligible compared to the fan-in d. In this work we fix this issue by employing a case analysis, using a different "packed" representation in the aforementioned case. Second, the KRW Conjecture (see Section 6) was stated incorrectly and in too weak a form; it has been updated accordingly.

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

2.1. Register programs. We use *register programs* as a convenient abstraction for describing space-bounded algorithms. Register programs were introduced by Ben-Or and Cleve [BC92] based on work of Coppersmith and Grossman [CG75] and explored in a number of follow-up works [Cle89, BCK+14, CM20, CM22].

[2] One way to obtain a size lower bound of $\Omega(m \cdot n)$ is to consider the parity function. Every computation must query each of the n input bits at least once, and for a fixed input, the m computation paths corresponding to the m source nodes must be disjoint, since they must all arrive at different sink nodes.

Definition 2.1. *A register program over an alphabet Σ consists of a collection of memory locations $R = \{R_1, \ldots, R_s\}$, called registers, each of which can hold one element from Σ, and an ordered list of instructions in the form of updates to some register R_i based on 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 2.2. *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 2.2. So, our programs may include any instruction

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

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

Following [BCK+14], rather than directly writing their output to a set of registers, our programs *add* their outputs to registers using addition over a finite field, a process called *clean* computation. This will be useful for making our algorithms space-efficient.

Definition 2.3. *Let $\mathcal{R}$ be a ring*[3] *and let f be a function whose output can be represented in $\mathcal{R}$. A register program over alphabet $\mathcal{R}$ with s registers* cleanly computes *f into a register R_o if for all possible $x_1, \ldots, x_n \in \{0,1\}^n$ and $\tau_1, \ldots, \tau_s \in \mathcal{R}$, if the program is run after initializing each register $R_i = \tau_i$, then at the end of the execution*

$$R_o = \tau_o + f(x_1, \ldots, x_n)$$

$$R_i = \tau_i \quad \forall i \neq o.$$

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

Definition 2.4. *If P is a register program that cleanly computes $f(x_1, \ldots, x_n)$, an* inverse *to P is any program P^{-1} which cleanly computes $-f(x_1, \ldots, x_n)$.*

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 2.5. *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, with s registers over $\mathcal{R}$ and which has t instructions in total, that cleanly computes f. Then f can be computed in space $O(c + s \log |\mathcal{R}| + \log t)$.*

[3] More generally, the definition could be applied to an Abelian or even non-Abelian group $\mathcal{R}$. In this work, $\mathcal{R}$ will always be a finite field.

Proof. To compute f, allocate $O(s \log |\mathcal{R}|)$ space for registers and initialize them to 0. Using space $\lceil \log t \rceil$ for a program counter, execute the instructions one at a time, temporarily using additional space c to execute each one. The output is the final value of the output register R_o. ∎

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 2.6. *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 representations 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 element of $\mathbb{F}_{p^a}$ 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 sometimes need a smaller field inside a larger finite field:

PROPOSITION 2.7. *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[4]; thus we treat $\mathbb{F}_{p^a}$ as a subset of $\mathbb{F}_{p^{ab}}$ when performing computations.

It can be useful to view elements of a larger field as tuples of elements from a smaller field:

PROPOSITION 2.8. *Let $\mathcal{G}$ be a subfield of a finite field $\mathcal{F}$. Then there exist $b \in \mathbb{N}$ and nonzero elements $e_1, \ldots, e_b \in \mathcal{F}$ such that the mapping $(x_1, \ldots, x_b) \mapsto x_1 e_1 + \cdots + x_b e_b$ is a bijection between $\mathcal{G}^b$ and $\mathcal{F}$.*

Proof. $\mathcal{F}$ is a vector space over $\mathcal{G}$. Since its cardinality is finite, it must have a finite basis $e_1, \ldots, e_b$. ∎

3. Roots of unity. Our work uses *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 3.1. *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 3.2. *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.

[4]For example, one way to do this 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 $\mathbb{F}_{p^{ab}}$ is isomorphic to $\mathbb{F}_{p^a}[y]/(g(y))$, with elements of $\mathbb{F}_{p^a}$ being represented as constant (degree-0) polynomials in $F_{p^a}[y]$.

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

PROPOSITION 3.3. *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 3.4. *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 $0 \le b < m$,*

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

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 3.3. 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 1.1, i.e. that TreeEval can be computed in space $O(\log n \cdot \log\log n)$. More specifically, we prove:

THEOREM 4.1. $\mathsf{TreeEval}_{k,h}$ *can be computed in space $O((h + \log k) \cdot \log\log k)$.*

This implies Theorem 1.1 for any setting of k and h, and is stronger as k gets smaller with respect to the total input size. The dependence on k can be improved; see Theorem 5.2.

Our algorithm is recursive, and treats its memory as a collection of registers over a field $\mathcal{K}$. We encode the value $f_u \in [k]$ at each node u as a string of bits $f_{u,1}, \ldots, f_{u,\lceil \log k \rceil}$. Instead of directly writing this string to memory, the main recursive subroutine *cleanly computes* it in the sense of Definition 2.3: it updates $\lceil \log k \rceil$ designated output registers as

$$\begin{aligned} R_{o,1} &\leftarrow R_{o,1} + f_{u,1} \\ &\vdots \\ R_{o,\lceil \log k \rceil} &\leftarrow R_{o,\lceil \log k \rceil} + f_{u,\lceil \log k \rceil} \end{aligned}$$

where we view the bits $f_{u,i}$ as the elements 0 and 1 of $\mathcal{K}$, and leaves all other register values unchanged. Importantly, we have no control over how any registers are

initialized. This restriction will cause us some trouble, but is the key to using less space.

EXAMPLE 4.2. *Suppose $k = 8$, $\mathcal{K} = \mathbb{F}_5$, the current register values are $R_{o,1} = 3, R_{o,2} = 1, R_{o,3} = 4$, and f_u's encoding is $101 \in \{0,1\}^3$. Then the main subroutine must update the registers to $R_{o,1} = 4, R_{o,2} = 1, R_{o,3} = 0$.*

In order to do this, we think of each bit $f_{u,i}$ as being a polynomial $q_{u,i}$ in the $2\lceil \log k \rceil$ inputs that come from u's two children. Our goal, then, is to compute the polynomials $(q_{u,i})_{i=1}^{\lceil \log k \rceil}$ while only using *clean* access to their inputs, that is, by making recursive calls which add the $2\lceil \log k \rceil$ input values to some other designated registers $R_{\ell,1}, \ldots, R_{\ell,\lceil \log k \rceil}, R_{r,1}, \ldots, R_{r,\lceil \log k \rceil}$. We never directly see the values $f_{\ell,i}, f_{r,i}$ of the child nodes. Instead, each time we make a recursive call, we first see these registers' initial values $\tau_{\ell,1}, \ldots, \tau_{\ell,\lceil \log k \rceil}$ and, later, their updated values $\tau_{\ell,1} + f_{\ell,1}, \ldots, \tau_{\ell,\lceil \log k \rceil} + f_{\ell,\lceil \log k \rceil}$ but not both at once.

To achieve this, we first find a useful equation for evaluating an arbitrary polynomial, given in Lemma 4.4. Then in Lemma 4.5 we turn it into a recursive subroutine, and then we prove Theorem 4.1 by running the subroutine on the root node and accounting for the space used.

Before moving to arbitrary polynomials, we consider a very simple polynomial, namely the product of d inputs.

LEMMA 4.3. *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*

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

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 3.4 we have

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

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

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

The next step is to move from individual products to general polynomials. This is accomplished by a simple corollary to Lemma 4.3.

LEMMA 4.4. *Let $\mathcal{K}$ be a finite field, let $d, m, n \in \mathbb{N}$ be such that $m = |\mathcal{K}| - 1 > d$, and let ω be a primitive root of unity of order m in $\mathcal{K}$. Let $q : \mathcal{K}^n \to \mathcal{K}$ be a degree-d polynomial over $\mathcal{K}$, and let $\tau := (\tau_i)_{i=1}^n \in \mathcal{K}^n, x := (x_i)_{i=1}^n \in \mathcal{K}^n$. Then*

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

Proof. Writing q as a sum of monomials we have

$$q(y_1, \ldots, y_n) = \sum_{\substack{\alpha \in \mathbb{N}^n \\ |\alpha| \le d}} c_\alpha \prod_{i=1}^{n} y_i^{\alpha_i}$$

for some coefficients $c_\alpha \in \mathcal{K}$ and formal variables $y_1, \ldots, y_n$, where $|\alpha| := \sum_{i=1}^n \alpha_i$. Then by substituting $\omega^j \tau_i + x_i$ for each y_i and summing over all j, we have

$$\begin{aligned}
\sum_{j=1}^{m} -1 \cdot q(\omega^j \tau_1 + x_1, \ldots, \omega^j \tau_n + x_n) &= \sum_{j=1}^{m} -1 \cdot \sum_{\substack{\alpha \in \mathbb{N}^n \\ |\alpha| \le d}} c_\alpha \prod_{i=1}^{n} (\omega^j \tau_i + x_i)^{\alpha_i} \\
&= \sum_{\substack{\alpha \in \mathbb{N}^n \\ |\alpha| \le d}} c_\alpha \cdot \left(-1 \cdot \sum_{j=1}^{m} \prod_{i=1}^{n} (\omega^j \tau_i + x_i)^{\alpha_i} \right) \\
&= \sum_{\substack{\alpha \in \mathbb{N}^n \\ |\alpha| \le d}} c_\alpha \prod_{i=1}^{n} x_i^{\alpha_i}.
\end{aligned}$$

The last equality follows from Lemma 4.3, since $\prod_{i=1}^n (\omega^j \tau_i + x_i)^{\alpha_i}$ is a product of at most $|\alpha| \le d$ factors $(\omega^j \tau_i + x_i)$. The last line is $q(x_1, \ldots, x_n)$ by definition. $\square$

Next, we show how to use Lemma 4.4 in a register program to compute our polynomial f_u in the way we described above, given an appropriate choice of $\mathcal{K}$.

LEMMA 4.5. *Let $\mathcal{K}$ be a finite field such that $m := |\mathcal{K}| - 1 > 2\lceil \log k \rceil$. Let u be a non-leaf node in our $\mathsf{TreeEval}_{k,h}$ instance. Let P_ℓ, P_r be register programs which cleanly compute the values $v_\ell, v_r \in \{0,1\}^{\lceil \log k \rceil}$ at u's children 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.*

Then there exists a register program P_u which cleanly computes $v_u = f_u(v_\ell, v_r) \in \{0,1\}^{\lceil \log k \rceil}$ 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} consist of m copies each of P_ℓ, P_r, P_ℓ^{-1}, and P_r^{-1}, plus $5m\lceil \log k \rceil$ other instructions. Additionally, both programs only use the registers R_u plus any additional registers used by P_ℓ, P_r.

Proof. Our goal is to use Lemma 4.4 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 turn this into a polynomial whose $2\lceil \log k \rceil$ variables are the bits of y and z by replacing $[y = \beta]$ and $[z = \gamma]$ with polynomials. $[y = \beta]$ becomes the polynomial

$$e(y,\beta) = \prod_{i=1}^{\lceil \log k \rceil} (1 - y_i + (2y_i - 1)\beta_i)$$

where β_i is the ith bit of the binary representation of β; this equals $[y = \beta]$ when all $y_i \in \{0, 1\}$. Similarly replacing $[z = \gamma]$ with $e(z, \gamma)$, this gives the polynomial

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

We note that $q_{u,i}$ is multilinear, and thus has degree at most $2\lceil \log k \rceil$.

Now that we have converted f_u to polynomials $q_{u,i}$, we use Lemma 4.4 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 $m = |\mathcal{K}| - 1$ in $\mathcal{K}$ (Proposition 3.2). 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 is to compute

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

where $\omega^j \tau_\ell + y$ and $\omega^j \tau_r + z$ are shorthand for tuples of $\lceil \log k \rceil$ values each. We do so using the following register 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: $R_{c,i} \leftarrow \omega^{-j} \cdot R_{c,i}$

The program P_u^{-1} is the same, except line 6 becomes $R_{u,i} \leftarrow R_{u,i} + q_{u,i}(R_\ell, R_r)$, replacing $-$ with $+$ to so that the program instead computes

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

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 $\text{poly}(k)$ 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 4.1 for an account of the space c required for these instructions.

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 2–4, 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, a requirement of clean computation (Definition 2.3). 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 4.4 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 5.1 in Section 5, but even as stated Lemma 4.5 is sufficient to serve as our main TreeEval subroutine.

Proof of Theorem 4.1. We show that our $\mathsf{TreeEval}_{k,h}$ instance can be cleanly computed by a register program of length at most $(9|\mathcal{K}|)^h \lceil \log k \rceil$ 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 2.5, our space usage is ultimately

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

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 $h' \leq h$ such a program of length $(9|\mathcal{K}|)^{h'} \lceil \log k \rceil$ computing f_u exists. For $h' = 0$, 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

$$\lceil \log k \rceil = (9 \cdot |\mathcal{K}|)^0 \lceil \log k \rceil$$

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

Now for a node u at height $h'+1$, we inductively assume we have register programs P_ℓ, P_r for the children ℓ, r of u, each of length $(9 \cdot |\mathcal{K}|)^{h'} \lceil \log k \rceil$ and which use $3\lceil \log k \rceil$ registers. We organize our registers into tuples R_ℓ, R_r, R_u, where P_ℓ computes f_ℓ into R_ℓ and P_r computes f_r into R_r; our goal then is to compute f_u into P_u.

Assuming $|\mathcal{K}| - 1 > 2\lceil \log k \rceil$, we apply Lemma 4.5 to u, inductively giving us a program of length

$$(|\mathcal{K}| - 1) \cdot [4 \cdot (9 \cdot |\mathcal{K}|)^{h'} \lceil \log k \rceil + 5\lceil \log k \rceil] \leq (9 \cdot |\mathcal{K}|)^{h'+1} \lceil \log k \rceil.$$

This completes the induction. We choose

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

which satisfies our two conditions[5]: 1) $\mathcal{K}$ has size $O(\log k)$, ensuring efficiency; and 2) $|\mathcal{K}| - 1 > 2\lceil \log k \rceil$, ensuring correctness.

[5] 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 $4\lceil \log k \rceil + 4$.

It remains only to show that our register program is space $O(h \log |\mathcal{K}| + \log k)$ uniform. Recall this means (Definition 2.2) 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 4.5 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 $t' \leq 2\lceil \log k \rceil$, then we are in 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 $(9|\mathcal{K}|)^h \lceil \log k \rceil$, this requires space $O(h \log |\mathcal{K}| + \log k)$.[6]

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 2.6), and $\log j \leq \log k$ bits suffice to create a loop to compute ω^j.

It remains to compute line 6. We do this using the definition of $q_{u,i}$. 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 $[f_u(\beta, \gamma) = \alpha]$ appears explicitly 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 2.6), the total space usage is at most $O(\log k + h + \log |\mathcal{K}|)$. ∎

5. Improvements and generalizations. In the rest of this paper we adapt the techniques used to other questions in complexity theory. To do so, we first state Lemma 4.5, which is our main subroutine, in a more general and efficient form.

LEMMA 5.1. *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 $g := g(x) \in \mathcal{F}^a$ 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.

The proof is essentially that of Lemma 4.5, and will appear at the end of this section. To see Lemma 4.5 as a special case[7] of Lemma 5.1, 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 5.1 saves some time by avoiding the need to call the inverse program P_g^{-1}.

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

[6] 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 store the values of c and i if we are in one of the loops on lines 2, 5, and 8, for a total of $O(h \log |\mathcal{K}| + \log k)$ space.

[7] Strictly speaking, it is not a special case, since Lemma 4.5 encodes values as bit strings (meaning $\mathcal{F} = \mathbb{F}_2$ in terms of Lemma 5.1) but does not require $\mathbb{F}_2$ to be a subfield of $\mathcal{K}$.

store the memory in the registers. We can improve this to logspace for one of these two aspects, namely the register memory.

THEOREM 5.2. $\mathsf{TreeEval}_{k,h}$ *can be computed in space* $O(h \log\log k + \log k)$.

(Theorem 5.2 is subsumed by Theorem 1.3, proved in Section 6.2.) One consequence is Theorem 1.2, repeated here for convenience:

THEOREM 1.2. *Let* $h := h(k)$ *be such that* $h = O(\frac{\log k}{\log\log k})$. *Then* $\mathsf{TreeEval}_{k,h}$ *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 only a polynomial increase in length. We will not formally state or prove this result.

The proof of Theorem 5.2 is similar to that of Theorem 4.1, except that instead of representing elements of $[k]$ in binary, we represent them as tuples of field elements in a 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 5.1 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 5.2. Let $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$ where r and s will be determined later. By Proposition 2.7 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 $q_{u,i}$ as in the proof of Lemma 4.5, is the same as for Theorem 4.1, except that instead of Lemma 4.5, we invoke Lemma 5.1 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; thus we now have polynomials $q_{u,i}$ over $\mathcal{F}$ for each $i \in [\lceil(\log k)/r\rceil]$. The register program P_g is the concatenation of two register programs for computing the values at the children of u. 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 by P_g in time $2t(h')$, so by Lemma 5.1,

$$t(h'+1) \le |\mathcal{K}|(2t(h') + O(\log k))$$

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

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

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

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

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

space to track our position in the program. Furthermore, our program is space $O(h\log|\mathcal{K}| + \log k) = O(hrs + \log k)$ uniform, which we show at the end of the proof. By Proposition 2.5, in total we need space

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

In order to use Lemma 5.1 we require $a(|\mathcal{F}|-1) < |\mathcal{K}|-1$; that is:

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

For sufficiently large[8] k, choosing $r = \lceil \log\log k\rceil$ gives us

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

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

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

as claimed.

Lastly we show that our register program is space $O(h\log|\mathcal{K}| + \log k) = O(hrs + \log k)$ uniform. Recall (Definition 2.2) 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 4.1, determining which instruction is the t-th can be done in space $O(\log t(h)) = O(h\log|\mathcal{K}| + \log\log k)$. 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 5.1, lines 2 and 4 are field arithmetic operations which require $O(\log|\mathcal{K}|)$ space (Proposition 2.6). Line 5 requires evaluating the polynomial p_i, which, examining Equation 5.1 (see below), can be done by looping over all $k^{O(1)}$ values of $(z_1,\dots,z_a)$ in the sum and 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. □

The remaining sections of the paper will focus on applications of Lemma 5.1, which is stronger and more flexible than Lemma 4.5 as seen above. To end this section we prove it, with a proof closely mirroring that of Lemma 4.5.

Proof of Lemma 5.1. 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 is 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 the fact that the multiplicative group of $\mathcal{F}$ is cyclic with order $|\mathcal{F}|-1$ we have $q_{z_\ell}(y_\ell) = [y_\ell = z_\ell]$ for any $y_\ell, z_\ell \in \mathcal{F}$. Define

$$p_i(y_1,\dots,y_a) = \sum_{(z_1,\dots,z_a)\in\mathcal{F}^a} f(z_1,\dots,z_a)\prod_{\ell=1}^{a} q_{z_\ell}(y_\ell) \tag{5.1}$$

and so 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 4.4,

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

[8] Any $k \ge 2^{256}$ is sufficiently large, since then $4(\log k)^2/\log\log k + 4\log k \le (\frac{1}{2}+\frac{1}{64})(\log k)^2 < (\log k)^2 - 1 \le 2^{2\log\log k}-1$. If k is smaller than that, take $r = \log\log 2^{256} = 8$.

This leads to the following algorithm. It replaces the inefficient warm-up version presented in the proof of Lemma 4.5 which required an extra m copies of P_g^{-1}.

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

We may assume $m > 1$ (otherwise p_i has degree 0, so is a constant), so $\omega \neq 1$ 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 $\ell = 1, \ldots, a$. 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, we have for $\ell \in [a]$

$$\begin{aligned} R_\ell &= \omega^m \tau'_\ell + g_\ell \\ &= \tau_\ell - g_\ell + g_\ell = \tau_\ell \end{aligned}$$

and Lemma 4.4 tells us that for $i \in [b]$,

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

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 our space-efficient TreeEval algorithms and the techniques they are based on. In this section we discuss their implications in the study of lower bounds against fan-in two formulas with AND, OR, and NOT gates. Our goal is to prove Theorem 1.4, repeated here for reference:

THEOREM 1.4. *If the KRW conjecture holds, then* $\mathsf{L} \not\subseteq \mathsf{NC}^1$.

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

CONJECTURE 6.1 (KRW Conjecture [KRW95]). *For a function f, let* $\mathsf{depth}(f)$ *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 $g_1 \circ g_2$ to be*

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

Then for large enough n_1, n_2, every function g_1, and at least a $2/3$ fraction of functions g_2, it holds that

$$\mathsf{depth}(g_1 \circ g_2) \geq \mathsf{depth}(g_1) + \mathsf{depth}(g_2) - O(1).$$

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 $\mathsf{TreeEval}$.

DEFINITION 6.2. *The d-ary Tree Evaluation Problem, denoted $\mathsf{TreeEval}_{k,d,h}$, is a generalization of $\mathsf{TreeEval}_{k,h}$ where each internal node has d children.*

We can view the KRW conjecture in relation to $\mathsf{TreeEval}_{k,d,h}$ with alphabet size $k = 2$, namely as a tool for "composing" lower bounds against each individual node into a lower bound against instances themselves.

LEMMA 6.3. *Conjecture 6.1 implies* $\mathsf{depth}(\mathsf{TreeEval}_{2,d,h}) = \Omega(dh)$.

Proof. For each layer $\ell \in [h]$, consider a random function $f_\ell : \{0,1\}^d \to \{0,1\}$, and note that with non-zero probability we have both 1) f_ℓ requires formula depth $d/2$ (which holds with probability 99/100 by a counting argument); and 2) $\mathsf{depth}(g_1 \circ g_2) \geq \mathsf{depth}(g_1) + \mathsf{depth}(g_2) - O(1)$, where $g_1 = (f_1 \circ \ldots \circ f_{\ell-1})$ and $g_2 = f_\ell$ (which holds with probability 2/3 by Conjecture 6.1). Fix functions f_ℓ satisfying these two properties for each $\ell \in [h]$ and fix each internal $\mathsf{TreeEval}_{2,d,h}$ node at height ℓ to f_ℓ. Thus our final $\mathsf{TreeEval}$ function is the iterated composition of all $f_1, \ldots, f_\ell$, and by our two assumptions this function requires depth $h \cdot (d/2 - O(1)) = \Omega(dh)$. ∎

Since $\mathsf{TreeEval}_{k,d,h}$ has input size $n = d^h k^d \log k$, fixing $k = 2$ gives us $\log n = O(h \log d + d)$. This implies, for the right setting of parameters, that $\Omega(dh) = \omega(\log n)$, and thus $\mathsf{TreeEval}_{2,d,h} \notin \mathsf{NC}^1$, assuming Conjecture 6.1. We give exact details after establishing the other side of Theorem 1.4, namely the space complexity of $\mathsf{TreeEval}_{2,d,h}$.

6.2. Space bounds for $\mathsf{TreeEval}_{k,d,h}$. Using Lemma 5.1, we can generalize Theorem 1.1, and in fact Theorem 5.2, to degrees d other than 2, proving Theorem 1.3:

THEOREM 1.3. $\mathsf{TreeEval}_{k,d,h}$ *can be computed in space* $O(h \log(d \log k) + d \log k)$.

Proof. Our proof proceeds much like that of Theorem 5.2. Let $\mathcal{F} = \mathbb{F}_{2^r}$ and $\mathcal{K} = \mathbb{F}_{2^{rs}}$ where r and s will be determined later; we treat $\mathcal{F}$ as a subfield of $\mathcal{K}$, and encode data as tuples of elements of $\mathcal{F}$.

Our proof strategy in Theorem 5.2 goes through as before after replacing $2 \log k$ with $d \log k$ in the degree and number of registers. However, this replacement gives rise to an issue: since the size of the larger field $\mathcal{K}$ must be at least the degree, and the degree in turn is at least the number of inputs a, for $a \geq d$ this results in at least $a \log |\mathcal{K}| = \Omega(d \log d)$ space being used to store registers, which could violate our target upper bound when $d \gg k$.

Our solution is to use two different encodings, depending on whether 1) $d < \frac{1}{2}\sqrt{k}$; or 2) $d \geq \frac{1}{2}\sqrt{k}$. Specifically, when $d < \frac{1}{2}\sqrt{k}$ we proceed as in Theorem 5.2; we represent each element of $[k]$ as a tuple of $\lceil (\log k)/r \rceil$ elements of $\mathcal{F}$, and later we choose r to be at most $\lceil \log k \rceil$.

Meanwhile, when $d \geq \frac{1}{2}\sqrt{k}$, we instead use a *packed* representation to group the values from child nodes together: a single element of $\mathcal{F}$ represents a tuple of up to c elements of $[k]$, where $c \leq d$ is an integer to be determined later; we choose $r = c\lceil \log k \rceil$ in this case.

We build this packed representation using Proposition 2.8, as follows. Let $\mathcal{G} = \mathbb{F}_{2^{\lceil \log k \rceil}}$; thus $\mathcal{G}$ is large enough to encode values in $[k]$, and we abuse notation by considering $[k]$ to be a subset of $\mathcal{G}$. We treat $\mathcal{G}$ as a subfield of $\mathcal{F}$ (Proposition 2.7), so Proposition 2.8 then gives us c elements $e_1, \ldots, e_c \in \mathcal{F}$ which allow us to encode a tuple $(v_1, \ldots, v_c) \in \mathcal{G}^c$ as $v_1 e_1 + \cdots + v_c e_c \in \mathcal{F}$.

We use induction to build a program P_u to compute the value v_u at each node u. Let $t(h')$ be the length of our programs at height h'.

The value at a leaf node can be read directly from the input, so $t(1) = O(d+\log k)$ as it is either $\lceil d/c \rceil$ or $\lceil (\log k)/r \rceil$.

To compute the value at an internal node we use Lemma 5.1. To do this, we must specify values a, b, a function $f : \mathcal{F}^a \to \mathcal{F}^b$, and an input g computed by a register program P_g.

First we define the function f, which is based on the function $f_u : [k]^d \to [k]$ at node u. In the $d < \frac{1}{2}\sqrt{k}$ case, we treat it as $f : \mathcal{F}^{d\lceil (\log k)/r \rceil} \to \mathcal{F}^{\lceil (\log k)/r \rceil}$, and we have $a = d\lceil (\log k)/r \rceil, b = \lceil (\log k)/r \rceil$, again just as in Theorem 5.2. In the "packed" (i.e. $d \geq \frac{1}{2}\sqrt{k}$) case, we instead treat f_u as

$$f : \mathcal{F}^{\lceil d/c \rceil} \to \mathcal{G}$$

and we have $a = \lceil d/c \rceil, b = 1$. In both cases, $g \in \mathbb{F}^a$ is an encoding of the d input values to the function f_u.

Let $u'_1, \ldots, u'_d$ be the children of u. The register program P_g is a concatenation of d programs $P_{u'_1}, \ldots, P_{u'_d}$ computing $v_{u'_1}, \ldots, v_{u'_d}$. In the $d \geq \frac{1}{2}\sqrt{k}$ case, P_g must include additional instructions to pack each group of c values $v_1, \ldots, v_c$ into a single register using the encoding $v_1 e_1 + \cdots + v_c e_c$. This can be done as follows.

1: **for** $i = 1, \ldots, \lceil d/c \rceil$ **do**
2: **for** $j = 1, \ldots, \min\{c, d-(i-1)c\}$ **do**
3: $R_i \leftarrow R_i / e_j$
4: Run $P_{u'_{ci+j}}$ to add $v_{u'_{ci+j}} \in \mathcal{G} \subseteq \mathcal{F}$ to R_i
5: $R_i \leftarrow R_i \cdot e_j$

The effect of lines 3–5 is to add e_j times the encoding of $v_{u'_{ci+j}}$ to R_i, so the net effect of the program is to add the full encoding $v_1 e_1 + \cdots + v_c e_c$ of each group of c values $v_{u'_i}$ to each of the $\lceil d/c \rceil$ registers. For a node at height $h'+1$, the length of P_g is $t(P_g) = d \cdot t(h')$ for the $d < \frac{1}{2}\sqrt{k}$ case, or $t(P_g) = d \cdot (t(h') + 2)$ for the $d \geq \frac{1}{2}\sqrt{k}$ case.

Finally, invoking Lemma 5.1 produces a program of length

$$(|\mathcal{K}| - 1)(t(P_g) + 2a + b) \leq 2^{rs}(dt(h') + O(d \log k))$$

for computing v_u. Putting all the layers together, we have $t(h) = O((d2^{rs})^h d \log k)$, so we need

$$\log t(h) = O(\log((d2^{rs})^h d \log k)) = O(hrs + h \log d + \log \log k)$$

space to track our position in the program.

We use a total of $a + b$ registers over $\mathcal{K}$, each requiring rs bits to store, thus giving us $(a+b)rs$ space in total. In the $d < \frac{1}{2}\sqrt{k}$ case, we have $a = d\lceil (\log k)/r \rceil$ and $b = \lceil (\log k)/r \rceil$, and we will make sure $r \leq \lceil \log k \rceil$ to ensure that $a+b = O((d \log k)/r)$. In the $d \geq \frac{1}{2}\sqrt{k}$ case, we have $a = \lceil d/c \rceil$ and $b = 1$ for some $c \leq d$, so $a + b = O(d/c)$; as stated above, we make sure that $r = c\lceil \log k \rceil$. Putting this together, the space used by registers is

$$(a+b)rs = O((d \log k)/r) \cdot rs = O(ds \log k)$$

in both cases. Lastly our program is

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

uniform by the same argument as in Theorems 4.1 and 5.2. Putting this all together, by Proposition 2.5, in total we need space

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

All that remains is to choose r and s, or c and s in the $d \geq \frac{1}{2}\sqrt{k}$ case. This is based on the parameter restrictions we have already committed to, as well as the constraint in Lemma 5.1 that $a(|\mathcal{F}| - 1) < |\mathcal{K}| - 1$.

We begin with the $d < \frac{1}{2}\sqrt{k}$ case, where our constraint becomes

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

As long as $k > 1$, setting $r = \log(2d\log k)$ and $s = 2$ satisfies the constraint, and we get an algorithm using space

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

In the $d \geq \frac{1}{2}\sqrt{k}$ case, having set $r = c\lceil\log k\rceil$ our constraint is

$$\lceil d/c\rceil(2^{c\lceil\log k\rceil} - 1) < 2^{sc\lceil\log k\rceil} - 1.$$

Setting $c = (\log d + 1)/(\log k + 1)$ and $s = 2$ satisfies this, and so $r = c\lceil\log k\rceil = O(\log d)$, giving us space

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

for our algorithm, which completes the proof. ∎

6.3. KRW and NC^1 vs L. The input to $\mathsf{TreeEval}_{k,d,h}$ is of length $d^h \cdot k^d \log k$, and thus Theorem 1.3 gives us an algorithm using space $O(\log n \cdot \log\log n)$ for every setting of k, d, and h. This follows from our precise space usage

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

where $h\log d$ and $d\log k$ are both $O(\log n)$. In fact, while $h\log\log k = O(\log n \cdot \log\log n)$ as with $\mathsf{TreeEval}_{k,h}$, this can be tightened for larger values of d. As with Theorem 1.2, this implies that some parameterizations of $\mathsf{TreeEval}_{k,d,h}$ are easy.

THEOREM 6.4. *Let $d := d(k)$ be such that $d \geq (\log k)^{\Omega(1)}$. Then $\mathsf{TreeEval}_{k,d,h}$ can be computed in L.*

Proof. Theorem 1.3 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 $O(h\log d + d\log k)$, which is $O(\log n)$ as $n = O(d^h \cdot k^d \log k)$. ∎

Theorem 6.4 should mostly be interpreted as a statement about large fan-in $\mathsf{TreeEval}$, rather than small alphabet $\mathsf{TreeEval}$, as it gives no new result in the case of $d = O(1)$ for any alphabet size[9]. In any case, applying Theorem 6.4 to $\mathsf{TreeEval}_{2,d,h}$ for large enough d immediately yields Theorem 1.4, which we state in a more quantitative form.

[9] We state a similar statement more geared towards small alphabets, due to [Sto23, Gol24b], in Appendix B.

THEOREM 6.5. *Assume Conjecture 6.1 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 6.4 we have $\mathsf{TreeEval}_{2,d,h} \in \mathsf{L}$, while Lemma 6.3 states that $\mathsf{TreeEval}_{2,d,h}$ requires depth $\Omega(dh) = \Omega(\log^2 n/\log\log n)$ as claimed. ∎

Theorem 6.5 uses the strongest version of Conjecture 6.1, but as stated in the introduction, any weakening which implies that $\mathsf{TreeEval}_{2,d,h}$ requires superlogarithmic formula depth is sufficient, with stronger versions of the conjecture directly translating to stronger separations between NC^1 and L.

7. Application 2: Near-optimal catalytic branching programs. Our second contribution outside of $\mathsf{TreeEval}$ is to the study of catalytic branching programs for computing arbitrary functions.

7.1. Catalytic branching programs. First, we establish the background of catalytic branching programs to give context to our results.

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 [GKM15] now.

DEFINITION 7.1. *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 are 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. Each source node corresponds to an initial setting τ of the registers, with the clean condition exactly giving back the pairing between source and sink nodes.

PROPOSITION 7.2. *Let $f : \{0,1\}^n \to \{0,1\}$ be a function, and 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*[10]*—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 represents a unique setting to all the registers. We execute one instruction of the register program per layer, querying the input bit corresponding to that instruction.

Finally, we 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 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. [CM22] used Proposition 7.2 plus the central TreeEval subroutines of [CM20, CM21] to improve 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 7.2 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 these 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 5.1 gives us a way to cleanly compute arbitrary polynomials, it seems natural to ask whether our techniques can improve the size of m-catalytic branching programs for computing ar-

[10] This is different from our earlier condition, given by Proposition 2.5, that each instruction be computable in small space. In non-uniform branching programs and register programs, 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.

bitrary functions. This requires us to leave behind our strategy of using Lemma 5.1 in a recursive way, and instead apply it directly to the whole function f in question.

Using this idea to prove Theorem 1.5 is the subject of the rest of the section. We prove a more general, fine-grained version of the theorem first.

THEOREM 7.3. *For any $f : \{0,1\}^n \to \{0,1\}$ and positive integers r, s such that*

$$\lceil n/r \rceil (2^r - 1) < 2^{rs} - 1 \tag{7.1}$$

there exists an m-catalytic branching program of width $2m$ and length less than $2^{rs} n \cdot (1 + 2/r + 3/n)$ 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 partition the input into groups of up to 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 plays the role of g in the statement of Lemma 5.1, 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 5.1 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 7.2, this gives us an m-catalytic branching program of length less than $2^{rs} n(1 + 2/r + 3/n)$ 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 5.1 requires $a(|\mathcal{F}| - 1) < |\mathcal{K}| - 1$, which is exactly our requirement

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

7.3. Main result. Theorem 1.5 follows from an analysis of various parameter regimes from Theorem 7.3.

Proof of Theorem 1.5. We analyze three ways to choose r and s to satisfy the precondition of Theorem 7.3, each corresponding to one claim of the theorem. In what follows, all asymptotics $(O(), o())$ take n as the growing variable, with one of r or s fixed and the other a function of n.

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

In the $s = 2$ setting, fix $r = \lceil \log n - \log\log n + 1 \rceil$. Let us verify the prerequsiite (7.1) of Theorem 7.3. For sufficiently large n,

$$\begin{aligned}\left\lceil \frac{n}{r} \right\rceil (2^r - 1) &< \frac{n}{\frac{2}{3}\log n}(2^r - 1) \\ &< 2^r (2^r - 1) \\ &< 2^{2r} - 1 = 2^{rs} - 1.\end{aligned}$$

So, we may apply Theorem 7.3. The length of the resulting program is at most

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

and for m we have

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

This proves the first claim of Theorem 1.5 (size $O(m \cdot n^3/\log^2 n)$, $m = 2^{(2+o(1))n}$).

For the second claim (size $O(m \cdot n^{2+\epsilon})$, $m = 2^{O(n)}$), we move to the $s \ge 3$ setting. Let

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

The prerequisite (7.1) is satisfied since for sufficiently large n, the left side is less than $2(s-1)n^{s/(s-1)}/\log n$ and the right side is at least $n^{s/(s-1)} - 1$. The length of the program given by Theorem 7.3 is at most

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

and for m we have

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

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

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

Note that ϵ can be made arbitrarily small by increasing s.

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

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

The prerequisite (7.1) is satisfied for large enough n since $2^{rs} - 1 > 2^r \cdot \frac{n}{r} - 1 > 2^r - 1$. The length of the program produced by Theorem 7.3 is at most

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

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 the third claim of Theorem 1.5 (size $O(m \cdot n^2)$, $m = O(2^{(2+\epsilon \log n)n})$). ϵ can be made arbitrarily small by increasing r. This 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.

Similarly, we may take the chance to consider what answer we might expect on the KRW conjecture. We have stated Theorem 1.4 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 1.4 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 below the bound given by Savitch's Theorem [Sav70] 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 5.1, 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, [Mer23] conjectured that an optimal improvement to Lemma 4.5 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.

Appendix A. Evaluating polynomials on a line. In this section we discuss the characterization of Lemmas 4.5 and 5.1 given by Goldreich [Gol24a]. They noted that the use of primitive roots of unity is not necessary; our key equation in Lemma 4.4, extrapolating from Lemma 4.3, is only one instantiation of a broader class of equations, stating that any $d + 1$ (or more) evaluations of a degree d polynomial p along a line is enough to determine its value anywhere else on the line.

LEMMA A.1. *Let $\mathcal{K}$ be a finite field, let $d, m, n \in \mathbb{N}$ be such that $d < m \leq |\mathcal{K}| - 1$, and let $p : \mathcal{K}^n \to \mathcal{K}$ be a degree-d polynomial over $\mathcal{K}$. Then for every distinct $\ell_1, \ldots, \ell_m \in \mathcal{K}$, there exist coefficients $c_1, \ldots, c_m \in \mathcal{K}$ such that the following interpolation equation is true for all choices of $\tau := (\tau_i)_{i=1}^n \in \mathcal{K}^n, x := (x_i)_{i=1}^n \in \mathcal{K}^n$.*

$$\sum_{j=1}^{m} c_j p(\ell_j \tau + x) = p(x).$$

Here, $\ell_j \tau + x$ represents the tuple $(\ell_j \tau_i + x_i)_{i=1}^n$. Each coefficient c_i can be computed in $O(\log |\mathcal{K}| + \log m)$ space given access to $\ell_1, \ldots, \ell_m$.

Proof. Fixing $\tau_1, \ldots, \tau_n, x_1, \ldots, x_n$, define the univariate polynomial

$$q(\ell) = p(\ell\tau + x) = p(\ell\tau_1 + x_1, \ldots, \ell\tau_n + x_n).$$

In other words, q is p restricted to the line $\ell\tau + x$.

Because q has degree less than m, we can use Lagrange interpolation to reconstruct the polynomial $q(\ell)$ by evaluating it at any m points. First, define

$$r_i(\ell) = \frac{\prod_{j \neq i}(\ell_j - \ell)}{\prod_{j \neq i}(\ell_j - \ell_i)}$$

so that $r_i(\ell_i) = 1$ and $r_i(\ell_j) = 0$ for $j \neq i$. Then $q(\ell) = \sum_{i=1}^{m} q(\ell_i) r_i(\ell)$ and so

$$p(x) = q(0) = \sum_{i=1}^{m} c_i q(\ell_i)$$

where each coefficient

$$c_i = \frac{\prod_{j \neq i} \ell_j}{\prod_{j \neq i}(\ell_j - \ell_i)}$$

can be computed from the values ℓ_i with arithmetic over $\mathcal{K}$ and a loop variable with m values. ∎

Lemma 4.4 is a special case: for a careful choice of ℓ_i, the coefficients are all $c_i = -1$. This has the advantage of simplicity—it is perhaps easier to see that the interpolation can be carried out without using much space. On the other hand, Goldreich went on to show that the view given by Lemma A.1 makes the later embedding argument given in Lemma 5.1 simpler by not relying on a field extension. Furthermore, it leads to the question of where our work can be generalized, and whether other choices of coefficients—perhaps based on properties of the specific polynomials coming from TreeEval—can yield nicer properties, or even improvements, of results such as Theorem 1.1.

To close this section, we remark that this generalization points to a barrier in improving Lemmas 4.5 and 5.1. From Lemma A.1 we know that $d + 1$ evaluations are *sufficient* to determine $p(x)$; however, for a general degree d polynomial, $d + 1$ evaluations are *necessary* as well.[11] The results on register programs for TreeEval given by [CM20, CM21] as well as this work only compute one evaluation $p(\alpha\tau + \beta x)$ per recursive call, as rebalancing the coefficients α and β requires accessing x; thus it would seem that we cannot decrease the amount of recursion at each node below $\Omega(\log k)$ calls without changing our approach.

Appendix B. Improving our result by an $O(\log\log\log n)$ factor. In this section we give an exposition of the results of Stoeckl [Sto23] and Goldreich [Gol24b], which give an improvement to Theorem 1.1.

THEOREM B.1 ([Sto23, Gol24b]). TreeEval *can be computed in space* $O(\log n \cdot \log\log n / \log\log\log n)$.

[11] As stated the lemma technically would work for just one evaluation, i.e. when $\ell = 0$ and $c = 1$. However, using $\ell = 0$ in our register program is akin to erasing the memory where we are adding x, which prevents us from cleanly computing our value. Thus what we mean here is that $d+1$ non-zero values ℓ are necessary.

In contrast to our techniques, which solely rely on efficient computation at the given nodes of the $\mathsf{TreeEval}$ instance, they alter the structure of the $\mathsf{TreeEval}$ instance itself to balance the gap in the space used for the register memory and the space used for the program counter in Theorem 1.2. In particular, *merging* levels of the tree allows us to reduce the height h at the expense of growing the fan-in d.

LEMMA B.2 ([Sto23, Gol24b]). *$\mathsf{TreeEval}_{k,d,h}$ can be computed in space*

$$O\left(h \log d + \frac{h}{t} \log\log k + d^t \log k\right)$$

for any integer $t = t(k,d,h) \in \{1,\dots,h\}$.

Proof. We can transform the $\mathsf{TreeEval}_{k,d,h}$ instance into a $\mathsf{TreeEval}_{k,d^t,\lceil h/t \rceil}$ instance as follows: starting at the top level, i.e. the level containing only the root of $\mathsf{TreeEval}$, we take each node at the current level and merge it with its subtree of depth t, with the function at the new node being the composite function of all nodes merged this way. This subtree has d^t leaves, as each internal node has branching factor d, and thus we are left with every node at the current level having d^t children, while we have removed t levels from the tree. We now move to the level containing these children and repeat. In the end we are left with a tree of height $\lceil h/t \rceil$ and each node having fan-in d^t.

We complete the proof with two applications of Theorem 1.3. First, the function $f_u : [k]^{d^t} \to [k]$ associated with each internal node u of the new tree is an instance of $\mathsf{TreeEval}_{k,d,t}$, and so can be computed in space

$$(\#) \qquad O(t \log(d \log k) + d \log k).$$

Setting this space aside allows us to access the input to the transformed instance of $\mathsf{TreeEval}_{k,d^t,\lceil h/t \rceil}$, which can then be computed in space

$$(*) \qquad O\left(\left\lceil \frac{h}{t} \right\rceil \log(d^t \log k) + d^t \log k\right) = O\left(h \log d + \frac{h}{t} \log\log k + d^t \log k\right).$$

The total space used is $(*)$ plus $(\#)$, which is dominated by $(*)$. ∎

We now use Lemma B.2 prove Theorem B.1. Note that in Section 1, we used $\mathsf{TreeEval}$ without parameters to refer to $\mathsf{TreeEval}_{k,h}$ rather than $\mathsf{TreeEval}_{k,d,h}$, and in fact [Sto23, Gol24b] only state Theorem B.1 for the case of $\mathsf{TreeEval}_{k,h}$ as well. However, extending their proof to general fan-in only requires a minor modification, and so we show their improvement in full generality.

THEOREM B.3. *Define $n := n(k,d,h) = \Theta(d^h k^d \log k)$ to be the size of inputs to $\mathsf{TreeEval}_{k,d,h}$. Then $\mathsf{TreeEval}_{k,d,h}$ can be computed in space $O(\log n \cdot \log\log n / \log\log\log n)$.*

Proof. By definition we have $h \log d = O(\log n)$ and $d \log k = O(\log n)$. Set $t = \lceil (\log\log\log k)/(2 \log d) \rceil$. If $t > h$ then we have $h < \log\log\log k$, so Theorem 1.3 gives space

$$O(\log n + \log\log n \log\log\log n) = O(\log n).$$

So assume instead that $t \le h$. Note that $d^t \le d \cdot d^{(\log\log\log k)/(2\log d)} = d\sqrt{\log\log k}$, so Lemma B.2 gives space

$$O\left(h \log d + \frac{h \log d \log\log k}{\log\log\log k} + d \log k \sqrt{\log\log k}\right) \le O\left(\frac{\log n \log\log n}{\log\log\log n}\right).$$ ∎

Theorems 1.2 and 6.4 show that $\mathsf{TreeEval}_{k,d,h}$ is in fact in logspace whenever d is too large or h is too small. We remark here that Lemma B.2 can be used to give a logspace upper bound for the last remaining parameter, namely whenever k is small.

THEOREM B.4 ([Sto23, Gol24b]). *Let $k := k(d,h)$ be such that*

$$\log k \leq (h \log d)^{1-\Omega(1)}.$$

Then $\mathsf{TreeEval}_{k,d,h}$ can be computed in L.

Proof. We again recall that $h \log d = O(\log n)$ and $d \log k = O(\log n)$. Let $\epsilon > 0$ be such that $\log k \leq (h \log d)^{1-\epsilon}$, let $\alpha = \frac{\epsilon}{2(1-\epsilon)}$—note that $\alpha = \Theta(1)$ and $1 + 2\alpha = (1-\epsilon)^{-1}$—and set $t = \lceil (\alpha \log\log k)/(\log d) \rceil$. We consider three cases.

Case 1: $t = 1$. Then $\log\log k \leq (\log d)/\alpha$, so Theorem 1.3 gives space

$$\begin{aligned} O(h \log(d \log k) + d \log k) &= O(h \log d + h \log\log k + d \log k) \\ &\leq O\left(h \log d + \frac{h \log d}{\alpha} + d \log k\right) \\ &\leq O(\log n). \end{aligned}$$

Case 2: $t > h$. Then $h < 2\alpha \log\log k / \log d$, and Theorem 1.3 gives space

$$\begin{aligned} O(h \log d + h \log\log k + d \log k) &\leq O(\alpha \log\log k + (\log\log k)^2 + d \log k) \\ &\leq O(\log n). \end{aligned}$$

Case 3: $2 \leq t \leq h$. Then $t \leq 2\alpha \log\log k / \log d$, so $d^t \leq (\log k)^{2\alpha}$, and applying Lemma B.2 gives space

$$\begin{aligned} O\left(h \log d + \frac{h}{t} \log\log k + d^t \log k\right) &\leq O(h \log d + h \log d + (\log k)^{1+2\alpha}) \\ &\leq O(h \log d + (h \log d)^{\frac{1-\epsilon}{1-\epsilon}}) \\ &\leq O(\log n). \end{aligned}$$

∎

At the end of Appendix A, we saw that $\Omega(\log k)$ recursive calls at each node could be inherently necessary, given our register program approach. However, the proof of Lemma B.2 circumvents this issue by taking the same subroutine at each node as before, but preprocessing the tree to alter its structure. This indicates that rather than trying to improve Lemma 4.5 against the backdrop of the $d + 1$ evaluations barrier, we should instead be looking for "external" approaches to shaving off the last factors in order to show $\mathsf{TreeEval} \in \mathsf{L}$.

Appendix C. List of theorems.

C.1. Space bounds for TreeEval.

THEOREM 1.1. $\mathsf{TreeEval}$ *can be computed in space $O(\log n \cdot \log\log n)$.*

THEOREM 4.1. $\mathsf{TreeEval}_{k,h}$ *can be computed in space $O((h + \log k) \cdot \log\log k)$.*

THEOREM 5.2. $\mathsf{TreeEval}_{k,h}$ *can be computed in space $O(h \log\log k + \log k)$.*

THEOREM 1.3. $\mathsf{TreeEval}_{k,d,h}$ *can be computed in space $O(h \log(d \log k) + d \log k)$.*

THEOREM B.1 ([Sto23, Gol24b]). $\mathsf{TreeEval}$ *can be computed in space $O(\log n \cdot \log\log n / \log\log\log n)$.*

THEOREM B.3. *Define $n := n(k,d,h) = \Theta(d^h k^d \log k)$ to be the size of inputs to $\mathsf{TreeEval}_{k,d,h}$. Then $\mathsf{TreeEval}_{k,d,h}$ can be computed in space $O(\log n \cdot \log\log n / \log\log\log n)$.*

C.2. Conditions under which TreeEval $\in$ L.

THEOREM 1.2. *Let $h := h(k)$ be such that $h = O(\frac{\log k}{\log\log k})$. Then $\mathsf{TreeEval}_{k,h}$ can be computed in L.*

THEOREM 6.4. *Let $d := d(k)$ be such that $d \geq (\log k)^{\Omega(1)}$. Then $\mathsf{TreeEval}_{k,d,h}$ can be computed in L.*

THEOREM B.4 ([Sto23, Gol24b]). *Let $k := k(d,h)$ be such that*

$$\log k \leq (h \log d)^{1-\Omega(1)}.$$

Then $\mathsf{TreeEval}_{k,d,h}$ can be computed in L.

C.3. The KRW conjecture.

THEOREM 1.4. *If the KRW conjecture holds, then $\mathsf{L} \not\subseteq \mathsf{NC}^1$.*

THEOREM 6.5. *Assume Conjecture 6.1 holds. Then there exists a function in L which requires formulas of depth $\Omega(\log^2 n/\log\log n)$.*

C.4. Catalytic branching programs.

THEOREM 1.5. *Every function $f : \{0,1\}^n \to \{0,1\}$ has m-catalytic branching programs of the following size:*

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

THEOREM 7.3. *For any $f : \{0,1\}^n \to \{0,1\}$ and positive integers r, s such that*

$$\lceil n/r \rceil (2^r - 1) < 2^{rs} - 1 \tag{7.1}$$

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

Acknowledgements. 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, STOC, and SICOMP for feedback and careful edits 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

[AFM+25] Yaroslav Alekseev, Yuval Filmus, Ian Mertz, Alexander Smal, and Antoine Vinciguerra. Catalytic computing and register programs beyond log-depth. *Electron. Colloquium Comput. Complex.*, TR25-055, 2025. URL: https://eccc.weizmann.ac.il/report/2025/055/.

[AM25] Aryan Agarwala and Ian Mertz. Bipartite matching is in catalytic logspace. *Electron. Colloquium Comput. Complex.*, TR25-048, 2025. URL: https://eccc.weizmann.ac.il/report/2025/048/.

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences (J.CSS)*, 38(1):150–164, 1989. doi:10.1016/0022-0000(89)90037-8.

[BC92] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM Journal on Computing (SICOMP)*, 21(1):54–58, 1992. doi:10.1137/0221006.

[BCK+14] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *ACM Symposium on Theory of Computing (STOC)*, pages 857–866, 2014. doi:10.1145/2591796.2591874.

[BDRS24] Sagar Bisoyi, Krishnamoorthy Dinesh, Bhabya Rai, and Jayalal Sarma. Almost-catalytic computation. *Electronic Colloquium on Computational Complexity (ECCC)*, TR24-140, 2024. URL: https://eccc.weizmann.ac.il/report/2024/140.

[BDS22] Sagar Bisoyi, Krishnamoorthy Dinesh, and Jayalal Sarma. On pure space vs catalytic space. *Theoretical Computer Science (TCS)*, 921:112–126, 2022. doi:10.1016/J.TCS.2022.04.005.

[BFM+25] Harry Buhrman, Marten Folkertsma, Ian Mertz, Florian Speelman, Sergii Strelchuk, Sathya Subramanian, and Quinten Tupker. Quantum catalytic space. In *Theory of Quantum Computation, Communication and Cryptography*, 2025.

[BKLS18] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic space: Non-determinism and hierarchy. *Theory of Computing Systems (TOCS)*, 62(1):116–135, 2018. doi:10.1007/S00224-017-9784-7.

[CFK+21] Arkadev Chattopadhyay, Yuval Filmus, Sajin Koroth, Or Meir, and Toniann Pitassi. Query-to-communication lifting using low-discrepancy gadgets. *SIAM Journal on Computing (SICOMP)*, 50(1):171–210, 2021. doi:10.1137/19M1310153.

[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. doi:10.1137/0129051.

[Cle89] Richard Cleve. *Methodologies for designing block ciphers and cryptographic protocols*. PhD thesis, University of Toronto, Canada, 1989.

[CLMP25] James Cook, Jiatu Li, Ian Mertz, and Edward Pyne. The structure of catalytic space: Capturing randomness and time via compression. In *ACM Symposium on Theory of Computing (STOC)*, 2025.

[CM20] James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *ACM Symposium on Theory of Computing (STOC)*, pages 752–760. ACM, 2020. doi:10.1145/3357713.3384316.

[CM21] James Cook and Ian Mertz. Encodings and the tree evaluation problem. *Electronic Colloquium on Computational Complexity (ECCC)*, TR21-054, 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 *IEEE Conference on Computational Complexity (CCC)*, volume 234 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:21, 2022. doi:10.4230/LIPIcs.CCC.2022.8.

[CM24] James Cook and Ian Mertz. Tree evaluation is in space O(log n · log log n). In *ACM Symposium on Theory of Computing (STOC)*, pages 1268–1278. ACM, 2024. doi:10.1145/3618260.3649664.

[CMW+12] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Transactions on Computational Theory (TOCT)*, 3(2):4:1–4:43, 2012. doi:10.1145/2077336.2077337.

[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 (LNCS)*, pages 211–223. Springer, 2020. doi:10.1007/978-3-030-50026-9_15.

[dRMN+20] Susanna F. de Rezende, Or Meir, Jakob Nordström, Toniann Pitassi, and Robert Robere. KRW composition theorems via lifting. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 43–49. IEEE, 2020. doi:10.1109/FOCS46700.2020.00013.

[EMP18] Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. Hardness of function composition for semantic read once branching programs. In *IEEE Conference on Computational Complexity (CCC)*, volume 102 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.ICALP.2016.36.

[FMST25] Marten Folkertsma, Ian Mertz, Florian Speelman, and Quinten Tupker. Fully characterizing lossy catalytic computation. In *Innovations in Theoretical Computer Science Conference (ITCS)*, volume 325 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:13, 2025.

[GJST19] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Unambiguous catalytic computation. In *Conference on Foundations of Software Technology*

and Theoretical Computer Science (FSTTCS), volume 150 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSTTCS.2019.16.

[GJST24] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Lossy catalytic computation. *Computing Research Repository (CoRR)*, abs/2408.14670, 2024.

[GKM15] Vincent Girard, Michal Koucký, and Pierre McKenzie. Nonuniform catalytic space and the direct sum for space. *Electronic Colloquium on Computational Complexity (ECCC)*, TR15-138, 2015.

[Gol08] Oded Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008. doi:10.1017/CBO9780511804106.

[Gol24a] Oded Goldreich. On the cook-mertz tree evaluation procedure. *Electronic Colloquium on Computational Complexity (ECCC)*, TR24-109, 2024. URL: https://eccc.weizmann.ac.il/report/2024/109.

[Gol24b] Oded Goldreich. Solving tree evaluation in $o(\log n \cdot \log \log n)$ space. *Electronic Colloquium on Computational Complexity (ECCC)*, TR24-124, 2024. URL: https://eccc.weizmann.ac.il/report/2024/124.

[GPW18] Mika Göös, Toniann Pitassi, and Thomas Watson. Deterministic communication vs. partition number. *SIAM Journal on Computing (SICOMP)*, 47(6):2435–2450, 2018. doi:10.1137/16M1059369.

[HPV77] John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space. *Journal of the ACM (J.ACM)*, 24(2):332–337, 1977. doi:10.1145/322003.322015.

[HPV21] William Hoza, Edward Pyne, and Salil Vadhan. Pseudorandom generators for unbounded-width permutation branching programs. In *Innovations in Theoretical Computer Science Conference (ITCS)*, Leibniz International Proceedings in Informatics (LIPIcs), 2021. doi:10.4230/LIPIcs.ITCS.2021.7.

[IN19] Kazuo Iwama and Atsuki Nagao. Read-once branching programs for tree evaluation problems. *ACM Transactions on Computational Theory (TOCT)*, 11(1):5:1–5:12, 2019. doi:10.1145/3282433.

[KMPS25] Michal Koucký, Ian Mertz, Ted Pyne, and Sasha Sami. Collapsing catalytic classes. *Electronic Colloquium on Computational Complexity (ECCC)*, TR25-018, 2025. URL: https://eccc.weizmann.ac.il/report/2025/018.

[KRW95] Mauricio Karchmer, Ran Raz, and Avi Wigderson. Super-logarithmic depth lower bounds via the direct sum in communication complexity. *Computational Complexity (CC)*, 5(3/4):191–204, 1995. doi:10.1007/BF01206317.

[Liu13] David Liu. Pebbling arguments for tree evaluation. *Computing Research Repository (CoRR)*, abs/1311.0293, 2013. doi:10.48550/arXiv.1311.0293.

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

[Pot17] Aaron Potechin. A note on amortized branching program complexity. In *IEEE Conference on Computational Complexity (CCC)*, volume 79 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CCC.2017.4.

[PSW25] Edward Pyne, Nathan S. Sheffield, and William Wang. Catalytic communication. In *Innovations in Theoretical Computer Science Conference (ITCS)*, volume 325 of *LIPIcs*, pages 79:1–79:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ITCS.2025.79.

[PV21] Edward Pyne and Salil Vadhan. Pseudodistributions that beat all pseudorandom generators (extended abstract). In *IEEE Conference on Computational Complexity (CCC)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CCC.2021.33.

[Pyn24] Edward Pyne. Derandomizing logspace with a small shared hard drive. In *IEEE Conference on Computational Complexity (CCC)*, volume 300 of *LIPIcs*, pages 4:1–4:20, 2024.

[RM99] Ran Raz and Pierre McKenzie. Separation of the monotone NC hierarchy. *Comb.*, 19(3):403–435, 1999. doi:10.1007/S004930050062.

[RZ21] Robert Robere and Jeroen Zuiddam. Amortized circuit complexity, formal complexity measures, and catalytic algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 759–769. IEEE, 2021. doi:10.1109/FOCS52979.2021.00079.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape

complexities. *Journal of Computer and System Sciences (J.CSS)*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

[Sha25] Yakov Shalunov. Improved bounds on the space complexity of circuit evaluation, 2025. URL: https://arxiv.org/abs/2504.20950, arXiv:2504.20950.

[Sto23] Manuel Stoeckl. Private correspondence, 2023.

[Wil25] Ryan Williams. Simulating time in square-root space. In *ACM Symposium on Theory of Computing (STOC)*, 2025.