Catalytic Computing and Register Programs Beyond Log-Depth

Yaroslav Alekseev*
Technion Israel Institute of Technology
tolstreg@gmail.com

Yuval Filmus*
Technion Israel Institute of Technology
yuvalfi@cs.technion.ac.il

Ian Mertz [†]
Charles University
iwmertz@iuuk.mff.cuni.cz

Alexander Smal
JetBrains Research
avsmal@gmail.com

Antoine Vinciguerra*
Technion Israel Institute of Technology
antoine.v@campus.technion.ac.il

April 24, 2025

Abstract

In a seminal work, Buhrman et al. (STOC 2014) defined the class $\mathsf{CSPACE}(s, c)$ of problems solvable in space s with an additional catalytic tape of size c, which is a tape whose initial content must be restored at the end of the computation. They showed that uniform TC^1 circuits are computable in catalytic logspace, i.e., $\mathsf{CL} = \mathsf{CSPACE}(O(\log n), 2^{O(\log n)})$, thus giving strong evidence that catalytic space gives L strict additional power. Their study focuses on an arithmetic model called register programs, which has been a focal point in development since then.

Understanding CL remains a major open problem, as TC^1 remains the most powerful containment to date. In this work, we study the power of catalytic space and register programs to compute circuits of larger depth. Using register programs, we show that for every $\epsilon > 0$,

$$\mathsf{SAC}^2 \subseteq \mathsf{CSPACE}\left(O\left(\frac{\log^2 n}{\log \log n}\right), 2^{O(\log^{1+\epsilon} n)}\right)$$

This is an $O(\log \log n)$ factor improvement on the free space needed to compute SAC^2, which can be accomplished with near-polynomial catalytic space.

We also exhibit non-trivial register programs for matrix powering, which is a further step towards showing e.g. $\mathsf{NC}^2 \subseteq \mathsf{CL}$.

1 Introduction

1.1 Catalytic Computation

In the realm of space-bounded computation, the *catalytic computing* framework, first introduced by Buhrman et al. [BCK$^+$14], investigates the power of having additional storage which has to be restored to its original value at the end of computation. A catalytic Turing machine is a space-bounded Turing machine with two read-write tapes: a standard work tape of size s and an additional, dedicated catalytic tape of size c. However, the catalytic tape begins completely filled in with some memory τ, and despite being available for use as work space, the catalytic tape must be restored to its initial state τ at the end of the computation.

While $\mathsf{CSPACE}(s, c)$, the class of functions solvable with work space s and catalytic space c, clearly sits between $\mathsf{SPACE}(s)$ and $\mathsf{SPACE}(s+c)$, naïvely it might seem that catalytic space should not improve the power of machines, as was conjectured in many previous works [CMW$^+$12, Liu13, EMP18, IN18]. Unexpectedly, [BCK$^+$14] showed that *catalytic logspace* ($\mathsf{CL} = \mathsf{CSPACE}(\log n, 2^{O(\log n)})$), the catalytic analogue of the complexity class L, contains the circuit class TC^1, which contains non-deterministic and randomized logspace (NL and BPL, respectively) as well as functions, such as determinant, widely believed to be outside both. Thus they gave a strong case for the power of catalytic memory as a resource.

Building on this result, the catalytic approach and methods has seen growing interest. Many follow-up works have sought to understand variants and properties of catalytic space have been studied, such as non-deterministic and randomized analogues [BKLS18, DGJ$^+$20, CLMP24, KMPS25], non-uniform catalytic models [Pot16, RZ22, CM22, CM23], robustness to errors in catalytic resetting [GJST24, FMST24], catalytic communication protocols [PSW25], and other variants [GJST19, BDS22, BDRS24] (see surveys by Koucký [K$^+$16] and Mertz [Mer23] for more discussion of these and other works).

Furthermore, applying catalytic tools to other complexity theoretic questions has seen successful results in recent years. Two of the most successful approaches have been 1) approaches to derandomizating non-catalytic space-bounded classes [Pyn24, DPT24, LPT24]; and 2) space-bounded algorithms for function composition [CM20, CM21, CM23], which recently culminated in a breakthrough simulation by Williams [Wil25] of time with quadratically less space.

Nevertheless, the exact strength of catalytic space remains open. In their first work, [BCK$^+$14] showed $\mathsf{TC}^1 \subseteq \mathsf{CL} \subseteq \mathsf{ZPP}$, with the key open problem being the relationship of CL to P. Further work has shown slight progress on both ends; Cook et al. [CLMP24] showed that CL reduces to the *lossy coding* problem, which itself is in ZPP, while Agarwala and Mertz [AM25] showed that bipartite matching, a function not known to be anywhere in the NC hierarchy, can be solved in CL.

The key open question in this work is whether or not CL contains higher levels of the NC hierarchy. Mertz [Mer23] posed a concrete approach to showing $\mathsf{NC}^2 \subseteq \mathsf{CL}$ via *register programs*, which we turn to now.

1.2 Register Programs

Register programs were first introduced by Coppersmith and Grossmann [CG75], and revived later by Ben-Or and Cleve to generalize Barrington's theorem to arithmetic circuits [Cle88, Cle90]. A register program over a ring $\mathbf{R}$ and a set of inputs $x_1, \ldots, x_n \in \mathbf{R}$ is defined as a sequence of instructions $I_1, \ldots, I_n : \mathbf{R}^s \mapsto \mathbf{R}^s$ applied to a set of registers $\{R_1, \ldots, R_s\}$.

The key technique of Buhrman et al. [BCK$^+$14] was based on register programs, which they showed can be directly simulated on a catalytic machine. They constructed a register program computing the kth power of some value x with k registers and $O(1)$ accesses to the value x; using some extensions and other works, they thus showing that CL contains the circuit class TC^1. This was also the driving force behind the result of Cook and Mertz [CM23], and by extension the work of Williams [Wil25], who showed that the Tree Evaluation problem, which was suggested as a function which would separate L from P [CMW$^+$12], can be computed in space $O(\log n \log \log n)$.

As with catalytic space more generally, the power of register programs are still a mystery, and deserves to be investigated thoroughly. The approach suggested in [Mer23] for CL versus NC^2 is to find a register program computing the kth power of a *matrix* with a register program matching the initial result of [BCK$^+$14] in

the non-commutative setting. More generally, it was proposed that further algebraic extension of these and other works could be the goal to improving the strength of CL.

1.3 Our Results

In this work, we investigate the complexity of computing polynomials in the register program framework, and make the first progress towards catalytic algorithms for circuit classes beyond TC^1. We present register programs for different kinds of polynomials (such as symmetric polynomials and polynomials representing Boolean functions), as well as more efficient programs for evaluating non-constant depth d boolean circuits with a constant number of recursive calls.

From this, we deduce that the class SAC^2 of Boolean circuits of polynomial size and depth $O(\log^2 n)$, with bounded fan-in AND gates and unbounded fan-in OR gates, can be computed with $o(\log^2 n)$ work space when given access to nearly polynomial catalytic space.

Theorem 1.1. *For all $\epsilon > 0$,*

$$\mathsf{SAC}^2 \subseteq \mathsf{CSPACE}\left(O\left(\frac{\log^2 n}{\log \log n}\right), 2^{O(\log^{1+\epsilon} n)}\right)$$

Our technique also gives such an improvement for NC^2 with only polynomial catalytic space, although this can be derived more directly from the results of [BCK$^+$14]. It also extends to $\#\mathsf{SAC}^2$, the arithmetic variant of SAC^2.

Second, we show sublinear register programs to compute matrix powering, thus making initial progress towards the program of [Mer20]:

Theorem 1.2. *Let $d, n, p \in \mathbb{N}$, where p is prime. Let $M \in M_n(\mathbb{F}_p)$. For all $\epsilon > 0$ there is a register program that computes M^d with*

- $O_\epsilon(d^\epsilon \log d)$ *recursive calls to M*

- $O_\epsilon\left(n^{\exp(1/\epsilon)}\right)$ *basic instructions, and*

- $O_\epsilon\left(n^{\exp(1/\epsilon)}\right)$ *registers over $\mathbb{F}_{O\left((np)^{\exp(1/\epsilon)}\right)}$.*

2 Preliminaries

2.1 Circuits

A Boolean circuit is a directed acyclic graph C with $0, 1$-valued inputs and outputs. Internal nodes (gates) are labeled with Boolean operations from a given set B. The circuit size $s(C)$ is the number of nodes, and its depth $d(C)$ is the longest input-output path. While Boolean circuits had been studied earlier [Sha49], their importance for parallel computing gained significant attention in the late 1970s [Bor77,Pip79,Ruz81,BCP83].

Definition 2.1. We define the following circuit families over input literals
$x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$:

- NC circuits: fan-in 2 AND and OR gates

- SAC circuits: fan-in 2 AND gates and unbounded fan-in OR gates[1]

- AC circuits: unbounded fan-in AND and OR gates

- TC circuits: unbounded fan-in threshold gates

---

[1]Since SAC^i is closed under complement for all $i \geqslant 1$ (see [BCD$^+$89]), these fan-ins can also be reversed, but for our argument we specifically use this version.

We denote by NC^i (SAC^i, AC^i, TC^i) the family of functions computable by NC (SAC, AC, TC) circuits of polynomial size and $O(\log^i n)$ depth. By NC we denote $\bigcup_{i \in \mathbb{N}} \mathsf{NC}^i$ and similarly for SAC, AC, and TC.

The relations between these different classes has been extensively studied [Ruz81, Vol99]. For all i, we have the following relations:
$$\mathsf{NC}^i \subseteq \mathsf{SAC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{TC}^i \subseteq \mathsf{NC}^{i+1}$$

As a consequence, we have that
$$\mathsf{NC} = \mathsf{SAC} = \mathsf{AC} = \mathsf{TC}$$

Furthermore, other lines of research [Bor77, Coo83, Ven92] have established other relationships:
$$\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{SAC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{TC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{SAC}^2 \subseteq \cdots \subseteq \mathsf{NC} \subseteq \mathsf{P}$$

While all containments are widely conjectured to be strict, no separations are known.

2.2 Catalytic Computation

Central to this work is the notion, introduced by Buhrman et al. [BCK$^+$14], of catalytic space.

Definition 2.2. A *catalytic Turing machine* with work space $s := s(n)$ and catalytic space $c := c(n)$ is a space-bounded Turing machine M with access to two read-write tapes: a standard work tape of size s, and a *catalytic* tape of size c. The catalytic tape has the additional restriction that for every $\tau \in \{0, 1\}^c$, if we run M on any input x with the catalytic tape in initial configuration τ, the catalytic tape has final configuration τ as well.

This definition gives rise to natural complexity classes:

Definition 2.3. We define $\mathsf{CSPACE}(s, c)$ as the class of problems solvable by a catalytic Turing machine with a work tape of size s and a catalytic tape of size c. Furthermore, we denote *catalytic logspace* to be $\mathsf{CL} = \mathsf{CSPACE}(O(\log n), 2^{O(\log n)})$.

2.3 Register Programs

Inspired by Ben-Or and Cleve's work on straight-line programs [BC92, Cle88], Buhrman et al. [BCK$^+$14] investigated the potential for optimization, examining a more generic type of computational model due to Coppersmith and Grossman [CG75].

Definition 2.4. Let $\mathbf{R}$ be a ring. An $\mathbf{R}$-*register program* over input $x_1, \ldots, x_n$ with space s is defined as a sequence of instructions $I_1, \ldots, I_t$ applied to a set of registers $R_1, \ldots, R_s$, where each instruction I_k has one of the following two forms:

- *Basic instruction*: updating a register R_i with a polynomial p_k over the other registers:
$$I_k \colon R_i \leftarrow R_i + p_k(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n)$$

- *Input access/Recursive call*: adds the input (scaled by some $\lambda \in \mathbf{R}$) to one register R_i:
$$I_k \colon R_i \leftarrow R_i + \lambda x_j$$

The time t of the register program is the number of instructions I_k.

We note that we use the term input access when we are given direct access to the input x, while we use the term recursive call when we are designing a subroutine whose input is being received not from the global function but rather from some preceding intermediate computation. We will make this distinction clear where necessary.

With regards to such subroutines, it will be important to restrict our register programs to be of a form amenable to composition:

Definition 2.5. A register program is called *clean* if, assuming all registers are initialized to some initial value, $\tau_i \in \mathbf{R}$, applying the sequence of instructions has the following effect:

- There is a subset of registers S such that for all register $R_i \in S$, $R_i = \tau_i + \delta_i$.

- For every other register R_j, $R_j = \tau_j$.

The total time and space usage of such composed programs can thus be directly analyzed:

Lemma 2.1 (Composition Lemma). *Let $f \colon \mathbf{R} \to \mathbf{R}$ and $g \colon \mathbf{R} \to \mathbf{R}$ be functions. Let R_f and R_g be clean register programs computing f and g with t_f and t_g recursive calls, s_f and s_g basic instructions, and r_f and r_g registers, respectively. Then there exists a register program $R_{f \circ g}$ that computes $f \circ g$ with*

- *$t_f t_g$ recursive calls,*

- *$s_f + t_f s_g$ basic instructions, and*

- *$\max \{r_f, r_g\}$ registers.*

Proof. Let $x \in \mathbf{R}$ and let $y = g(x)$, so $f(g(x)) = f(y)$. We can cleanly compute $f(y)$ using R_f, with t_f recursive calls to y, s_f basic instructions, an input register and $r_f - 1$ additional registers. On the other hand, we can cleanly compute $y = g(x)$ into the input register of R_f using t_g recursive calls and s_g basic instructions for each of the t_f recursive calls made to g.

There are now two cases:

- If $r_g - 1 < r_f - 1$, since R_g cleanly computes g, we can use the additional registers of R_f to compute y.

- If $r_g - 1 > r_f - 1$, we add $r_g - r_f$ registers and compute y using the latter registers as well as the additional registers of R_f.

Thus we can compute $f(g(x))$ with $t_f t_g$ recursive calls to x and $\max \{r_f, r_g\}$ registers. $\qquad\square$

Lastly we connect clean register programs to catalytic computation:

Lemma 2.2 (Lemma 15 in [BCK$^+$14]). *Any (uniform) clean register program of time t, space s, and with n inputs over a finite ring $\mathbf{R}$ can be simulated by a catalytic Turing machine in pure space $O(\log t + \log n + \log |\mathbf{R}|)$ and catalytic space $O(s \log |\mathbf{R}|)$.*

2.4 Polynomial Representation

The question of representing Boolean functions as multivariate polynomials over a ring has been intensively studied and has proven to be a useful tool in circuit complexity (see [Bei93] for a survey). We will consider two kinds of representation: one that we will call the representation of f, and the other the weak representation of f.

Definition 2.6. Let P be a polynomial on n variables over a ring $\mathbf{R}$, and let f be a Boolean function with n inputs. We say P *represents* f if for all inputs $x \in \{0, 1\}^n$, we have $P(x) = 0$ if and only if $f(x) = 0$. We say P *weakly represents* f if there exist two disjoint sets $S_0, S_1 \subset \mathbf{R}$ such that for all inputs $x \in \{0, 1\}^n$, we have $f(x) = b$ if and only if $P(x) \in S_b$, for $b \in \{0, 1\}$.

Let us underline the difference between these two definitions of representation with an example. Let us consider the n-ary AND function. Observe that the polynomial $P(X_1, \ldots, X_n) = \sum_{i=1}^{n} X_i$ over any $\mathbb{Z}_m$, where $m > n$, weakly computes AND. Indeed, we can take $S_0 = \{1, \ldots, n-1\}$ and $S_1 = \{n\}$. On the other hand, P does not represent AND, and in general, it is known that AND cannot be represented by a degree 1 polynomial.

Definition 2.7. We will say that a Boolean function f has a (weak) $(\mathbf{R}, d)$-representation if there is a degree d polynomial which (weakly) represents it. We also define $(\mathbf{R}, d, s)$-representation, where additionally the polynomial is required to have at most s monomials.

3 Register Programs for Polynomials

3.1 Computing Univariate Polynomials

In order to prove $\mathsf{TC}^1 \subseteq \mathsf{CL}$, [BCK$^+$14] design a register program to compute x^n for any element x in a commutative field. We state a straightforward generalization of their lemma and corresponding program to compute arbitrary univariate polynomials:

Lemma 3.1. *Let $p \in \mathbb{N}$ be a prime number, and let $P \in \mathbb{F}_p[X]$ be a univariate polynomial of degree at most n. For all $x \in \mathbb{F}_p$, there is a register program that computes $P(x)$ with*

- *4 recursive calls to x,*

- *$2n + 2$ basic instructions, and*

- *$n + 2$ registers.*

Proof. Let $P = \sum_{i=0}^{n} a_i X^i$ for some coefficients a_i. Let R_{in} be the input register initially equal to τ_{in}. It is straightforward to show, by writing x as $(\tau_{in} + x) - \tau_{in}$, that

$$P(x) = \sum_{i=0}^{n} a_i \sum_{j=0}^{n} \binom{i}{j} (\tau_{in} + x)^j (-\tau_{in})^{i-j}$$

We will use this equation as well as the register program to compute x^n in Lemma 4 of [BCK$^+$14], to construct a register program to compute $P(x)$, given in Figure 1.

$\square$

As discussed in [BCK$^+$14,CM23], this program can be adapted to compute a set of polynomials $\{P_1, \ldots, P_\ell\}$ with similar parameters:

Lemma 3.2. *Let $p \in \mathbb{N}$ be a prime number, and let $P_1 \ldots P_\ell \in \mathbb{F}_p[X]$ be univariates polynomial of degree at most n. For all $x \in \mathbb{F}_p$, there is a register program $\mathbf{UP}_{P_1, \ldots, P_\ell}$ that computes $P_1(x) \ldots P_\ell(x)$ with*

- *4 recursive calls to x,*

- *$2n + 2\ell$ basic instructions, and*

- *$n + 1 + \ell$ registers.*

Proof. The program is the same as that of Lemma 3.1, but with each instruction involving R_{out} replaced by ℓ instructions of the same form, where each involves a different output register $R_{out,j}$ and the corresponding polynomial $P_j(x)$. $\square$

3.2 Computing Multivariate Polynomials

In the last section we showed that computing powers of elements in a commutative ring can be done with a constant number of recursive calls. Moreover, it should be clear that computing the sum of variables is an easy operation: we simply add each variable to a fixed output register in turn (see [BCK$^+$14]).

Thus our goal is to represent our polynomial in such a way that we only have to perform addition and powering operations. Here our discussion of representations in Section 2.4 comes into play:

Theorem 3.3 ([Sch02, BBS08]). *Let $P \in \mathbb{F}_p[x_1, \ldots, x_n]$ be a homogeneous polynomial of degree $d < p$. There exist $m \in \mathbb{N}$, m elements α_i and nm elements $\beta_{i,j}$ such that:*

$$P(x_1, \ldots, x_n) = \sum_{i=1}^{m} \alpha_i \left(\sum_{j=1}^{n} \beta_{i,j} x_j \right)^d.$$

Moreover, $m \leq \binom{n+d-1}{d-1}$.

5

```
 1  Registers:
 2  R_in = τ_in
 3  R_1 = τ_1, ..., R_n = τ_n
 4  R_out = τ_out
 5
 6  R_in ← R_in + x  //  R_in = τ_in + x
 7
 8  For  1 ≤ i ≤ n
 9        R_i ← R_i + R_in^i  //  R_i = τ_i + (τ_in + x)^i
10
11  R_in ← R_in − x  //  R_in = τ_in
12
13  For  1 ≤ i ≤ n
14        R_out ← R_out + a_i ((−1)^i R_in^i + Σ_{j=1}^i (i choose j)(−1)^{i−j} R_j R_in^{i−j})
15  //  R_out = τ_out + P(x) − a_0 + Σ_{i=1}^n a_i Σ_{j=1}^i (i choose j)(−1)^{i−j} τ_j τ_in^{i−j}
16
17  R_in ← R_in + x  //  R_in = τ_in + x
18  R_i ← R_i − R_in^i  //  R_i = τ_i
19  R_in ← R_in − x  //  R_in = τ_in
20
21  For  1 ≤ i ≤ n
22        R_out ← R_out − a_i Σ_{j=1}^i (i choose j)(−1)^{i−j} R_j R_in^{i−j}
23  //  R_out = τ_out + P(x) − a_0
24
25  R_out ← R_out + a_0  //  R_out = τ_out + P(x)
```

Figure 1: Program for computing a polynomial $P(x)$ of degree n using 4 recursive calls to x, $2n + 2$ basic instructions, and $n + 2$ registers

This new representation does not involve any multiplication. We can thus describe a register program that computes a polynomial in $O(1)$ recursive calls:

Lemma 3.4. *Let P be a homogeneous degree $d < p$ polynomial $P(x_1, \ldots, x_n)$ over $\mathbb{F}_p$. There is a register program F_P that cleanly computes P with*

- *4 recursive calls to each x_i,*
- *$O(d\binom{n+d}{d})$ basic instructions, and*
- *$O(d\binom{n+d}{d})$ registers over $\mathbb{F}_p$.*

Proof. We use Theorem 3.3 and write P as:

$$P(x_1, \ldots, x_n) = \sum_{i=1}^{m} \alpha_i \left(\sum_{j=1}^{n} \beta_{i,j} x_j \right)^d.$$

Let $f_i(x_1, \ldots, x_n) = \sum_{j=1}^{n} \beta_{i,j} x_j$ and $g = x^d$. Then by the above discussion:

- f_i can be cleanly computed with 1 recursive call to each x_i, 0 basic instructions, and 1 register.
- g can be cleanly computed with 4 recursive calls, $2d + 2$ basic instructions, and $d + 2$ registers using the register program from Lemma 3.1.

Hence by Lemma 2.1, there exists a register program R_i that cleanly computes

$g(f_i(x_1, \ldots, x_n)) = \left(\sum_{j=1}^{n} \beta_{i,j} x_j \right)^d$ with 4 recursive calls, $2d + 2$ basic instructions, and $d + 2$ registers.

We can compute all these programs R_i in parallel, only sharing the output register R_{out}. This leaves us with a register program that cleanly computes P in R_{out} with:

- 4 recursive calls to each x_i,
- $m(2d + 2) \leqslant (2d + 2)\binom{n+d-1}{d-1} + 1$ basic instructions, and
- $m(d + 1) + 1 \leqslant (d + 1)\binom{n+d-1}{d-1} + 1$ registers,

as required in the statement of the lemma. $\square$

This register program immediately generalizes to a non-homogeneous polynomial P by considering the decomposition $P = \sum_{i=0}^{d} P_i$, where each P_i is a homogeneous degree i polynomial.

Corollary 3.4.1. *Let P be a degree $d < p$ polynomial $P(x_1, \ldots, x_n)$ over $\mathbb{F}_p$. There is a register program F_P that cleanly computes P with*

- *4 recursive calls to each x_i,*
- *$O(d^2 \binom{n+d}{d})$ basic instructions, and*
- *$O(d^2 \binom{n+d}{d})$ registers over $\mathbb{F}_p$.*

This register program has the advantage of employing a constant number of recursive calls. However, this method has two caveats. First, the cost in the number of registers is superpolynomial when d is non-constant. Second, the degree d is upper bounded by the field size.

Concerning the field issue, we will instead consider the field $\mathbb{F}_q$ for some $q > d$, such that $(P \bmod q) \bmod p = P \bmod p$. Observe that for any degree d polynomial over $\mathbb{Z}$ where the coefficients are smaller than p, the polynomial evaluation for $x = (x_1, \ldots, x_n)$, where $0 \leqslant x_i \leqslant p$, is upper bounded by

$$P(x) \leqslant p \sum_{i=0}^{d} \binom{n}{i} p^i \leqslant 2^n p^{d+1}$$

Hence, we can first evaluate $P(x)$ over a field of size $q \geqslant 2^n p^{d+1}$. This yields the most general register program, leaving yet the problem with the number of registers unresolved.

Corollary 3.4.2. *Let P be a degree d polynomial $P(x_1, \ldots, x_n)$ over $\mathbb{F}_p$. There is a register program that cleanly computes P with*

- *4 recursive calls to each x_i,*

- $O(d^2 \binom{n+d}{d})$ *basic instructions,*

- $O(d^2 \binom{n+d}{d})$ *registers over $\mathbb{F}_q$, where $q = O(2^n p^{d+1})$, and*

- *1 register over $\mathbb{F}_p$.*

We also note one register program of orthogonal strength, namely greater in recursive calls but much lesser in space. In order to prove that Tree Evaluation is in $\mathsf{SPACE}(\log n \log \log n)$ (later improved by Stoeckl, see [Gol24]), Cook and Mertz [CM23] present a register program to compute multivariate polynomials, which we will also use later:

Lemma 3.5. *Let $P(x_1, \ldots, x_n)$ be a polynomial of degree $d < p$ over a prime field $\mathbb{F}_p$. There exists a register program that computes P with:*

- n *input registers*

- *1 output register*

- $O(d)$ *basic instructions*

- $d + 1$ *instructions of the type $I_{\lambda, all}$ or its inverse $I_{\lambda, all}^{-1}$, where:*

$$I_{\lambda, all} \colon \text{For } 1 \leqslant i \leqslant \delta,$$
$$R_{in,\ell} \leftarrow R_{in,\ell} + \lambda x_i.$$

3.3 Computing Boolean functions

In the preceding section we presented a register program to compute polynomials over any finite field. However, for the case of polynomials over $\mathbb{F}_2$, i.e. Boolean functions, there are specific properties that we can exploit to make them simpler to compute, the most important being that any polynomial over $\mathbb{F}_2$ is multilinear since $x_i^2 = x_i$.

Since any Boolean function f is uniquely represented by a polynomial P over $\mathbb{Z}_2$, we will directly consider Boolean functions in this section. Let us present a register program that computes any Boolean function f given a representation of f.

Let us first consider the case of symmetric functions. Given a symmetric function f in n variables, there is some univariate polynomial $g \colon [n] \mapsto \mathbb{N}$ such that $f(x_1, \ldots, x_n) = g(x_1 + \cdots + x_n)$. We can compute g using Lemma 3.1, which yields the following register program:

Lemma 3.6. *Let $n \in \mathbb{N}$, and let f be a symmetric polynomial function. Let $p \in \mathbb{N}$ be a prime number such that $p > n$. There is a register program that computes f with:*

- *4 recursive calls to each x_i,*

- $O(n)$ *basic instructions, and*

- $O(p)$ *registers over $\mathbb{Z}_p$.*

From this we deduce the following lemma for polynomials in general:

Lemma 3.7. *Let $f \colon \{0,1\}^n \mapsto \{0,1\}$ be a boolean function which is $(\mathbb{Z}, d, t)$-represented, and let p be a prime number such that $p > \max\{d, t\}$. There is a register program which cleanly computes f with:*

- *64 recursive calls to each x_i,*

- $O(tp^2 \log p)$ *basic instructions, and*

- $O(tp)$ *registers over* $\mathbb{Z}_p$.

Proof. Let P be the polynomial which represents f, which we write as a sum of terms

$$P = \sum_{j=1}^{t} u_k$$

where each term u_k has the form

$$u_k = \prod_{j=1}^{d} x_{i_j}, \quad i_1, \ldots, i_d \in [n]$$

Observe that each u_k is symmetric, and so using the register program given by Lemma 3.6, we deduce that we can compute all the terms in parallel with

- 4 recursive calls to each x_i,

- $O(tp)$ basic instructions, and

- $O(tp)$ registers over $\mathbb{Z}_p$.

To compute $\sum_{k=1}^{t} u_k$, we can again use the register program of Lemma 3.6. This yields a register program with

- 4 recursive calls to each u_j,

- $O(p)$ basic instructions, and

- $O(p)$ registers over $\mathbb{Z}_p$.

Composing the two register programs using Lemma 2.1, we have a register program R_f which computes P with

- 16 recursive calls to each x_i,

- $O(tp^2)$ basic instructions, and

- $O(tp)$ registers over $\mathbb{Z}_p$.

Lastly, we convert the computation of P into a computation of f. Note that our final output register for P is over $\mathbb{Z}_p$, which we represent in binary with $O(\log p)$ bits. We now apply the register program for the OR function from Lemma 3.1 which uses 4 recursive calls, $O(\log p)$ basic instructions, and $O(\log p)$ registers. Composing this with the rest of the program gives us a final program for computing f with

- 64 recursive calls to each x_i,

- $O(tp^2 \log p)$ basic instructions, and

- $O(tp)$ registers over $\mathbb{Z}_p$.

which completes the lemma. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

3.4 Circuits via Merging Layers

In this section, we will find efficient register programs for circuits, and from it efficient catalytic algorithms, by a strategy of merging layers and directly computing the "super-functions" that emerge. Our starting point is the following lemma, used in [Cle88, BCK+14].

Lemma 3.8. *Let B be a set of Boolean functions such that for any function $g \in B$ we have a register program P_g with at most t recursive calls, b basic instructions, and r registers computing g. Let C be a depth d, size s circuit whose gates are functions in B. Then C can be computed by a register program P_C with*

- *$O(t^d)$ recursive calls,*
- *$O(sb \cdot t^d)$ instructions, and*
- *$O(rs)$ registers.*

Proof. We will perform the operations of each layer of the circuit in parallel. Each layer uses t recursive calls to the previous layer and sb basic instructions; hence for a depth d circuit, applying Lemma 2.1 iteratively gives $O(t^d)$ recursive calls to the last layer and $sb \cdot O(t^d)$ total basic instructions. Since each layer has at most s gates, we will need at most rs registers at each layer, and so again by Lemma 2.1 this gives $2rs$ registers in total. $\qquad\square$

Our strategy will be as follows: instead of computing a height h circuit C with AND and OR gates layer by layer, we will show that we can compute d layers at a time by an efficient register program, and thus consider the circuit C' of height h/d whose gates are themselves depth d circuits. We do this using polynomial representations of such circuits, which we then combine with Lemma 3.7 to obtain the register programs in question. Our main statement is the following:

Lemma 3.9. *Let C be a polynomial size depth d circuit with fan-in 2 AND gates and fan-in ℓ OR gates for some ℓ. Then C can be represented by a degree $k \leqslant 2^d$ polynomial with $t \leqslant \ell^{2^d}$ terms over $\mathbb{Z}$.*

Proof. The proof is by induction. The claim for $d = 0$ follows immediately since this is in this case the circuit computes one of the inputs.

We now assume that the claim holds for depth $d - 1$, and let C be a depth d circuit. We apply the induction hypothesis to the children of the top gate g, and have two cases for g itself:

If the top gate is an OR gate: We can find a polynomial $P = \sum_{i=1}^{\ell} P_i$ representing the circuit C, where P_i is the polynomial for each input of the OR gate. Using the induction hypothesis

$$\deg P = \max_i \deg P_i \leqslant 2^{d-1} < 2^d.$$

Moreover, if we let t_i be the number of terms of each P_i, the number of terms of P is

$$t = \sum_i t_i \leqslant \ell \cdot \ell^{2^{d-1}} \leqslant \ell^{2^d}$$

If the output gate is a binary AND gate: Let P_l and P_r respectively be the polynomials representing the left and right children. The polynomial $P = P_l P_r$ represents the circuit C, and

$$\deg P = \deg P_l + \deg P_r \leqslant 2^{d-1} + 2^{d-1} = 2^d.$$

On the other hand,

$$t = t_l t_r \leqslant \left(\ell^{2^{d-1}}\right)^2 = \ell^{2^d}$$

which completes the proof. $\qquad\square$

Combining Lemma 3.9 for $\ell = n^{O(1)}$ with our register program for representations in Lemma 3.7, we immediately obtain the following corollary.

Corollary 3.9.1. *Let C be a size s depth d circuit on n inputs with unbounded fan-in OR gates and fan-in 2 AND gates, and let $p \in \mathbb{N}$ be a prime number such that $p > s^{2^d}$. There is a register program which cleanly computes C with:*

- *64 recursive calls to each x_i,*

- *$s^{O(2^d)}$ basic instructions, and*

- *s^{2^d} registers over $\mathbb{Z}_p$.*

Our main result follows for the right choice of d.

Proof of Theorem 1.1. Let $d \in \mathbb{N}$ be such that $d \leqslant \epsilon \log \log n$. Corollary 3.9.1 provides a register program cleanly computing any size $n^{O(1)}$ depth d bounded fan-in OR fan-in 2 AND circuit with

- 64 recursive calls to each x_i,

- $2^{O(\log^{1+\epsilon} n)}$ basic instructions, and

- $2^{O(\log^{1+\epsilon} n)}$ registers over $\mathbb{Z}_p$, where $p = 2^{O(\log^{1+\epsilon} n)}$.

Given an SAC^2 circuit C of size $n^{O(1)}$ and depth $O(\log^2 n)$, we will rewrite it as a circuit C' of size at most $n^{O(1)}$ and depth $O(\frac{\log^2 n}{d})$, where each gate is a size $n^{O(1)}$ depth d circuit with unbounded fan-in OR and fan-in 2 AND gates. Hence using Lemma 3.8, for $p = 2^{O(\log^{1+\epsilon} n)}$ we have a register program for C with

- $64^{O\left(\frac{\log^2 n}{\epsilon \log \log n}\right)} = 2^{O\left(\frac{\log^2 n}{\log \log n}\right)}$ recursive calls to each x_i,

- $n^{O(1)} 2^{O(\log^{1+\epsilon} n)} \cdot 2^{O\left(\frac{\log^2 n}{\log \log n}\right)} = 2^{O\left(\frac{\log^2 n}{\log \log n}\right)}$ basic instructions, and

- $n^{O(1)} \cdot 2^{O(\log^{1+\epsilon} n)} = 2^{O(\log^{1+\epsilon} n)}$ registers over $\mathbb{Z}_p$.

At the end these recursive calls translate into basic instructions reading the input, giving a total time of $2^{O\left(\frac{\log^2 n}{\log \log n}\right)}$. We can thus translate this register program to a catalytic machine using Lemma 2.2, and we deduce that:

$$C \in \mathsf{CSPACE}\left(\frac{\log^2 n}{\log \log n}, 2^{O(\log^{1+\epsilon} n)}\right)$$

as claimed. $\qquad\qquad\square$

3.5 Matrix Powering via Decomposition

We now move to register programs for computing powers of a matrix $M \in M_n(\mathbb{Z}_p)$. A first attempt can be given by simply applying Lemma 3.4, as computing M^d is equivalent to computing n^2 degree d polynomial in the n^2 coefficients; namely, if we denote by $m_{i,j}^{(d)}$ the coefficient of M^d, we have:

$$m_{i,j}^{(d)} = \sum_{1 \leqslant k_1, \ldots, k_{d-1} \leqslant n} m_{i,k_1} \left(\prod_{i=1}^{d-2} m_{k_i, k_{i+1}}\right) m_{k_{d-1}, j}.$$

We can therefore use Lemma 3.4 to compute M^d for $d < p$:

Lemma 3.10. *Let $M \in M_n(\mathbb{F}_p)$. There is a register program that computes M^d for $d < p$ with:*

- *4 recursive calls to M,*

11

- $O(dn^2\binom{n+d}{d})$ *basic instructions, and*

- $O\left(dn^2\binom{n^2+d}{d}\right)$ *registers over $\mathbb{F}_p$.*

The register program from Lemma 3.10 is a first step towards a more generic program for matrix powering, but it has two major issues:

- it works only to compute powers up to $p-1$.

- the number of registers grows exponentially with d.

We address these issues independently to get a register program which can handle all the cases.

Computing any power d   Let $M \in M_n(\mathbb{F}_p)$ and let $L \in M_n(\mathbb{Z})$ be the natural extension of M to integers. Observe that the coefficients $\ell_{i,j}^{(d)}$ of L^d are evaluations of degree d polynomials with n^{d-1} terms, and hence $\ell_{i,j}^{(d)} \leqslant p^d n^{d-1}$.

Let us consider the first prime number q greater than $p^d n^{d-1} \geqslant d$. In this case, we have $\left(\ell_{i,j}^{(d)} \bmod q\right) \bmod p = \ell_{i,j}^{(d)} \bmod p = m_{i,j}^{(d)}$. Hence we can use the register program of Lemma 3.10 for $d < q$ and have an output register over $\mathbb{F}_p$ for each coefficient. This yields the following:

Lemma 3.11. *Let $M \in M_n(\mathbb{F}_p)$. There is a register program that computes M^d with:*

- *4 recursive calls to M,*

- $O\left(dn^2\binom{n^2+d}{d}\right)$ *basic instructions,*

- $O\left(dn^2\binom{n^2+d}{d}\right)$ *registers over $\mathbb{F}_{O(p^d n^{d-1})}$, and*

- $O(n^2)$ *registers over $\mathbb{F}_p$.*

Reducing the Number of Registers   The latter register program works for all d and has a constant number of recursive calls. However, the number of registers is still exponential in d, and therefore it is not usable as is. To fix this issue, let us observe that, if we let f_d be the powering function $f_d(M) = M^d$, we have $f_{d^k} = (f_d)^{\circ^k}$, where $\circ^k$ means composing k times the same function. An iterated application of Lemma 2.1 gives the following:

Lemma 3.12. *Let $\mathbf{R}$ be a ring, $x \in \mathbf{R}$ and $\delta \in \mathbb{N}$. Suppose that for all $k \leqslant \delta$ there is a clean register program P_k computing x^k with at most t recursive calls, s basic instructions and r registers. Then, there exists a register program P that computes x^d with*

- *at most $(\lceil \log_\delta d \rceil + 2)\frac{t}{t-1}t^{\lceil \log_\delta d \rceil}$ recursive calls,*

- $O(\log_\delta d)\frac{t+s(t+1)^{\lceil \log_\delta d \rceil}}{t}$ *basic instructions, and*

- $1 + r(\lfloor \log_\delta d \rfloor + 1)$ *registers over $\mathbf{R}$.*

Proof. Let $f_k \colon \mathbf{R} \to \mathbf{R}$ be the functions that computes x^k for all $k \leqslant d$. Observe that

$$d = \sum_{i=0}^{\lfloor \log_\delta d \rfloor} \alpha_i \delta^i$$

where α_i are non-negative integers smaller than δ. Therefore

$$x^d = \prod_{i=0}^{\lfloor \log_\delta d \rfloor} (x^{\delta^i})^{\alpha_i}$$

Note that $(x^{\delta^i})^{\alpha_i} = f_{\alpha_i} \circ (f_\delta)^i(x)$. We can apply Lemma 2.1 to obtain a register program R_i that cleanly computes $(x^{\delta^i})^{\alpha_i}$ with t^{i+1} recursive calls, $(t+1)^i s$ basic instructions, and r registers. Then, using Lemma 3.5 for $P = \prod_i (x^{\delta^i})^{\alpha_i}$, we can compute x^d by calling $\lfloor \log_\delta d \rfloor + 2 \leqslant \lceil \log_\delta d \rceil + 2$ times each program R_i, and using a single additional register. In total, we require

- at most $(\lceil \log_\delta d \rceil + 2) \sum_{i=0}^{\lfloor \log_\delta d \rfloor} t^{i+1} \leqslant (\lceil \log_\delta d \rceil + 2) \frac{t}{t-1} t^{\lceil \log_\delta d \rceil}$ recursive calls to x,

- $O(\log_\delta d)\left(1 + s \sum_{i=0}^{\lfloor \log_\delta d \rfloor}(t+1)^i\right) \leqslant O(\log_\delta d)\frac{t+s(t+1)^{\lceil \log_\delta d \rceil}}{t}$ basic instructions, and

- $1 + \sum_{i=0}^{\lfloor \log_\delta d \rfloor} r = 1 + r(\lfloor \log_\delta d \rfloor + 1)$ registers. $\qquad \square$

Hence, we fix some $\delta \in \mathbb{N}$, and we can compute M^d for all d based on the register program for δ. This yields our register program for Theorem 1.2.

Proof of Theorem 1.2. Let $\delta \in \mathbb{N}$. Combining Lemma 3.11 and Lemma 3.12, we get that for all $d \in \mathbb{N}$, there exists a register program that computes M^d with:

- $O(\frac{\log d}{\log \delta} d^{\frac{3}{\log \delta}})$ recursive calls to M,

- $O\left(\frac{\delta n^2 d^{\frac{3}{\log \delta}} \log d}{\log \delta}\binom{n^2+\delta}{\delta}\right)$ basic instructions,

- $O(\frac{n^2 \log d}{\log \delta})$ registers over $\mathbb{F}_p$.

Replacing by $\epsilon = \frac{3}{\log \delta}$, yields a program with

- $O(\epsilon d^\epsilon \log d)$ recursive calls to M,

- $O\left(\epsilon \frac{n^{2^{\frac{3}{\epsilon}+1}}}{2^{\frac{3}{\epsilon}-1}} d^\epsilon \log d\right)$ basic instructions,

- $O\left(\epsilon \frac{n^{2^{\frac{3}{\epsilon}+1}}}{2^{\frac{3}{\epsilon}-1}} \log d\right)$ registers over $\mathbb{F}_q$ for $q = O\left((np)^{2^{\frac{3}{\epsilon}}}\right)$, and

- $O(\epsilon n^2 \log d)$ registers over $\mathbb{F}_p$.

which completes the proof. $\qquad \square$

References

[AM25]   Aryan Agarwala and Ian Mertz. Bipartite matching is in catalytic logspace. In *Electron. Colloquium Comput. Complex*, volume 48, 2025.

[BBS08]   Andrzej Białynicki-Birula and Andrzej Schinzel. Representations of multivariate polynomials by sums of univariate polynomials in linear forms. In *Colloquium Mathematicum*, volume 2, pages 201–233, 2008.

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

[BCD$^+$89]   Allan Borodin, Stephen A. Cook, Patrick W. Dymond, Walter L. Ruzzo, and Martin Tompa. Two applications of inductive counting for complementation problems. *SIAM J. Comput.*, 18(3):559–578, 1989.

[BCK+14]   Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 857–866, 2014.

[BCP83]   Allan Borodin, Stephen Cook, and Nicholas Pippenger. Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Information and control*, 58(1-3):113–136, 1983.

[BDRS24]   Sagar Bisoyi, Krishnamoorthy Dinesh, Bhabya Deep Rai, and Jayalal Sarma. Almost-catalytic computation. *arXiv preprint arXiv:2409.07208*, 2024.

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

[Bei93]   Richard Beigel. The polynomial method in circuit complexity. In *[1993] Proceedings of the Eigth Annual Structure in Complexity Theory Conference*, pages 82–95. IEEE, 1993.

[BKLS18]   Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic space: Nondeterminism and hierarchy. *Theory of Computing Systems*, 62:116–135, 2018.

[Bor77]   Allan Borodin. On relating time and space to size and depth. *SIAM journal on computing*, 6(4):733–744, 1977.

[CG75]   Don Coppersmith and Edna Grossman. Generators for certain alternating groups with applications to cryptography. *SIAM Journal on Applied Mathematics*, 29(4):624–627, 1975.

[Cle88]   Richard Cleve. Computing algebraic formulas with a constant number of registers. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 254–257, 1988.

[Cle90]   Richard Erwin Cleve. *Methodologies for designing block ciphers and cryptographic protocols*. PhD thesis, University of Toronto, 1990.

[CLMP24]   James Cook, Jiatu Li, Ian Mertz, and Edward Pyne. The structure of catalytic space: Capturing randomness and time via compression. ECCC TR24-106, 2024.

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

[CM21]   James Cook and Ian Mertz. Encodings and the tree evaluation problem. In *Electron. Colloquium Comput. Complex*, volume 54, 2021.

[CM22]   James Cook and Ian Mertz. Trading time and space in catalytic branching programs. In *37th Computational Complexity Conference (CCC 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

[CM23]   James Cook and Ian Mertz. Tree evaluation is in space o (log n· log log n). In *Electron. Colloquium Comput. Complex.*, 2023.

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

[Coo83]   Stephen A Cook. The classification of problems which have fast parallel algorithms. In *International Conference on Fundamentals of Computation Theory*, pages 78–93. Springer, 1983.

[DGJ+20]   Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Randomized and symmetric catalytic computation. In *International Computer Science Symposium in Russia*, pages 211–223. Springer, 2020.

[DPT24]   Dean Doron, Edward Pyne, and Roei Tell. Opening up the distinguisher: A hardness to randomness approach for bpl= l that uses properties of bpl. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 2039–2049, 2024.

[EMP18]   Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. Hardness of function composition for semantic read once branching programs. In *33rd Computational Complexity Conference (CCC 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.

[FMST24] Marten Folkertsma, Ian Mertz, Florian Speelman, and Quinten Tupker. Fully characterizing lossy catalytic computation. *arXiv preprint arXiv:2409.05046*, 2024.

[GJST19] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Unambiguous catalytic computation. In *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.

[GJST24] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Lossy catalytic computation. *arXiv preprint arXiv:2408.14670*, 2024.

[Gol24]   Oded Goldreich. Solving tree evaluation in o (log n· log log n) space. *ECCC, TR24-124*, 2024.

[IN18]   Kazuo Iwama and Atsuki Nagao. Read-once branching programs for tree evaluation problems. *ACM Transactions on Computation Theory (TOCT)*, 11(1):1–12, 2018.

[K+16]   Michal Koucký et al. Catalytic computation. *Bulletin of EATCS*, 1(118), 2016.

[KMPS25] Michal Koucký, Ian Mertz, Ted Pyne, and Sasha Sami. Collapsing catalytic classes. In *Electron. Colloquium Comput. Complex*, volume 19, 2025.

[Liu13]   David Liu. Pebbling arguments for tree evaluation. *arXiv preprint arXiv:1311.0293*, 2013.

[LPT24]   Jiatu Li, Edward Pyne, and Roei Tell. Distinguishing, predicting, and certifying: On the long reach of partial notions of pseudorandomness. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–13. IEEE, 2024.

[Mer20]   Ian Mertz. Catalytic computing, tree evaluation, & clean computation, 2020.

[Mer23]   Ian Mertz. Reusing space: Techniques and open problems. *Bulletin of EATCS*, 141(3), 2023.

[Pip79]   Nicholas Pippenger. On simultaneous resource bounds. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 307–311. IEEE, 1979.

[Pot16]   Aaron Potechin. A note on amortized branching program complexity. *arXiv preprint arXiv:1611.06632*, 2016.

[PSW25]   Edward Pyne, Nathan S. Sheffield, and William Wang. Catalytic Communication. In Raghu Meka, editor, *16th Innovations in Theoretical Computer Science Conference (ITCS 2025)*, volume 325 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 79:1–79:24, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Pyn24]   Edward Pyne. Derandomizing logspace with a small shared hard drive. In *39th Computational Complexity Conference (CCC 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[Ruz81]   Walter L Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981.

[RZ22]   Robert Robere and Jeroen Zuiddam. Amortized circuit complexity, formal complexity measures, and catalytic algorithms. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 759–769. IEEE, 2022.

[Sch02]   Andrzej Schinzel. On a decomposition of polynomials in several variables. *Journal de théorie des nombres de Bordeaux*, 14(2):647–666, 2002.

[Sha49]   Claude E Shannon. The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98, 1949.

[Ven92]   H Venkateswaran. Circuit definitions of nondeterministic complexity classes. *SIAM Journal on Computing*, 21(4):655–670, 1992.

[Vol99]   Heribert Vollmer. *Introduction to circuit complexity: a uniform approach*. Springer Science & Business Media, 1999.

[Wil25]   Ryan Williams. Simulating time with square-root space. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 2025.