The Complexity of Composition: New Approaches to Depth and Space

by

Ian Mertz

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2022 by Ian Mertz

Abstract

The Complexity of Composition: New Approaches to Depth and Space

Ian Mertz
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2022

The *composition* of two given functions f and g is a fixed way of combining them into a single new function $f \circ g$. A *composition theorem* for a complexity measure $s(\cdot)$ states that $s(f \circ g) \approx s(f) + s(g)$; in other words, computing the combined function $f \circ g$ is no easier (with respect to s) than computing f and g individually. If true, then we would gain a natural approach towards proving lower bounds on $s(F)$ for an explicit F by repeatedly composing smaller hard functions in such a way that their complexities are additive by the composition theorem. We study the composition problem for two measures: *formula depth* and *space complexity*.

The *KRW conjecture* [KRW95] states that the formula depth required to compute $f \circ g$ is approximately $depth(f) + depth(g)$, where $f \circ g$ is the function given by replacing every input variable of f with a disjoint copy of g. This conjecture is known to imply $\mathsf{NC}^1 \subsetneq \mathsf{P}$. We work towards proving this conjecture by way of proving new *lifting theorems* from query complexity to communication complexity. Our new proof of the classic result of Raz and McKenzie [RM99] allows us to intimately connect lifting to combinatorics, and in doing so we provide a novel improvement to a key parameter called the gadget size. This result also allows us to prove conditional hardness for *automating* the *Cutting Planes* proof system.

Cook et al. [CMW$^+$12] introduced the *tree evaluation problem* as a way of showing $\mathsf{L} \subsetneq \mathsf{P}$; their central conjecture partially relies on showing that the space to compute a function f while remembering the output of another function g is approximately the space to compute f plus the size of g's output. This conjecture, which we call the *z-f conjecture*, was challenged by Buhrman et al. [BCK$^+$14], who defined a new type of space computation called *catalytic computing* and used it to show that composition does not hold for space-bounded computation in some settings. We give further evidence against composition by using the catalytic computing framework to give the first upper bounds on tree evaluation since the problem's definition in [CMW$^+$12], refuting their central conjecture. Using these techniques we also prove new results on *amortized* computation by improving constructions for *catalytic branching programs*.

Acknowledgements

A thesis ends with a bibliography, acknowledging all the works which opened the door to proving the results therein; it only seems right to begin by thanking all the people who helped me walk through it.

I can start nowhere but with Toniann Pitassi. My decision to come to University of Toronto—certainly the luckiest I have made in my career—resulted directly from our meeting on visit day. She has been a model in how to research: broad but with focus, discerning but with passion in every project. She has also been, as any of her many past students and present friends can attest, a blast to spend time with; discussions with Toni are a constant reminder of why you love theoretical computer science. To say my time as her student came at a difficult time for research is a criminal understatement; to say that she was fantastic as a mentor, collaborator, and guiding presence over the past six years, doubly so.

In a similar vein, I have been blessed with collaborators who have continually kept the process of research exciting. To name just those I share authorship with: Eric Allender, James Cook, Anna Gál, Mika Göös, Sajin Koroth, Shachar Lovett, Raghu Meka, Toniann Pitassi, Hao Wei, and Jiapeng Zhang. Distinguished among them is James Cook, who came to the Fields Institute in 2019 with a brilliant piece of research and ended up sharing his time and insights with me for three years since. Thanks to our work together, I finally have an angle to pursue the field of catalytic computing, which has arrested my attention ever since I first encountered it as an undergraduate.

There are many other researchers who, despite being nowhere on my CV (yet), were a vital part of research over the years. Noah Fleming and Morgan Shirley have been my favorite kind of collaborators: the ones you think best with over a beer. Robert Robere was an influence on my work from day one—as he was for most everyone else he shared the lab with—and I'm grateful he entrusted the theory student seminar to me in his absence. The seminar was as intellectually stimulating as procrastinating on research could be, and so my thanks are due to the many participants who joined in.

Few are those who find themselves at a Ph.D program who do not owe an immeasurable debt to their mentors along the way. In elementary school it was Fred Kriesler, who decided that fourth grade is plenty early to love induction and hat puzzles. In high school it was Graciela Elia, who provided a haven for burgeoning computer scientists to learn and experiment. In the summers it was Rajiv Gandhi, who pushed us first to learn, then to research, and finally to teach. In undergraduate it was Eric Allender, who took me seriously right from my first semester and gave me all the opportunities of the best-served graduate students. Finally in my masters it was Toniann Pitassi (surely not again?) and Steve Cook, who, along with Jeff Edmonds and Ben Rossman, gave me all the research angles I'd need for the long years to follow.

While it is often joked that a completed thesis goes unread, the present work benefited enormously from the few who were obliged to scour it in its entirety. Ben Rossman, Henry Yuen, and Shubhangi Saraf are researchers I never truly got the chance to work closely with, and despite this they took on the thankless task of being on my Ph.D committee, which requires not just time and focus but a surprising amount of paperwork. Mike Saks went above and beyond in his role as the external committee member, giving dozens upon dozens of helpful comments which helped reshape many sections. Any remaining errors, of course, are mine alone.

Logistical support is seldom properly acknowledged. Avi Wigderson and Ran Raz made many trips accompanying Toni to Princeton possible by inviting her students, formally and otherwise, to the Institute for Advanced Study. Likewise Steve Cook invited me back to the Heidelberg Laureate Forum in 2019 after a dazzling first experience three years prior. Tom Graves and Becky Saeger opened up their home

in San Francisco to me while I was visiting the Simons Institute, while my parents (more on them later) did the same for my many semesters at the IAS.

Even rarer is it that the endless people whose work makes the world habitable receive their due. The custodial staff in Sanford Fleming kept the office immaculate, and I owe them additional thanks for brightening up every quiet work evening with banter. The administrative and technical staff at the DCS, particularly Ingrid Varga, were invaluable in making the process of being a researcher as seamless as possible. Teaching work was not just a pleasurable learning experience but also an affordable one, in large part thanks to the workers, stewards, and tireless volunteers of CUPE 3902. My six years in Toronto was made possible by hundreds of servers, receptionists, baristas, floor managers, bartenders, property managers, and contract workers; I may never know them, but it is not hard to imagine how difficult life in the city would have been without them.

On the other hand, it is impossible to imagine any kind of life without all the friends who made, and continue to make, every day a joy. Roughly in order of appearance: the high school FC friend group (James, Bri, Rhea, Adam, Martin, Christina, Danny, Izzy, Hex, Michael H., Kristie, Michael W., Lindsay, John); undergraduate game friends (Stephen, Ian, Alan, Morgan, Dan); the engineering group (Brooke, Cate, Tim, Katie, Mary Pat, Guillermo, Jess); Japan friends (Yuhi, Koichiro, Matthias); animation friends (Toadette, Mew, Poppy); Toronto friends (Felipe, Ryan, Eric, Jeremy, Chris, Benett, Ilana, Noah, Alex, Morgan, Greg, Lily, Deeksha, Coby, Adrian, Deepanshu, Yasaman, Yuval, Lawrence); conference friends (Sophie, Xinyu, Rahul); and family here and gone. I feel profoundly lucky to have so many people to thank, and I know that a list twice as long still wouldn't suffice.

It would be easy enough, albeit an understatement, to dedicate this thesis to my parents on the basis of their endless love and support, without which I wouldn't be here today, and consider the matter settled; as one dedication[1] put it: "they are my parents and it'd be f*cked up not to thank them." But in a Ph.D program marred by the COVID pandemic—an incident which made the lifeblood of a healthy research life, namely meeting around the whiteboard, all but impossible—there is much more to be said.

Instead of living in near isolation during the prolonged lockdown, my parents and I spent twenty-two months making and eating good food, mixing up cocktails, experimenting with spice blends, hiking in Taconic, reading together by the fireplace, holding weekly tea and coffee tastings, exercising first thing in the morning, listening to jazz on the turntable at the end of the evening, and gathering every day in between to talk about flavor and quantum mechanics and town politics and everything and anything else at the forefront of our minds. Over half of the present work came from two years in their attic, and far more of the present author came from twenty-eight years at their kitchen table.

This thesis is the culmination of a thousand pieces of fortune, the largest of which being that Herb Mertz and Fran McManus are my parents and friends. It can only be dedicated to them.

[1] Branson Reese, *Hell Was Full.*

Contents

Abstract ii

Acknowledgements iii

Contents vi

1 Introduction 1

1.1 Composition . . . 1
1.2 Computation models and complexity measures . . . 4
1.3 Hardness through composition . . . 5
1.4 Our results . . . 10

I Depth: Communication lower bounds for composed functions 16

2 Query-to-Communication Lifting Theorems 17

2.1 Preliminaries . . . 19
2.2 Main proof: tree-like lifting via sunflowers . . . 22
2.3 Dag-like lifting . . . 32
2.4 Graduated lifting . . . 37
2.5 A note on combinatorics and lifting . . . 39

3 Application: Cutting Planes Proofs are Hard to Find 42

3.1 Proof complexity and lifting . . . 43
3.2 Main proof 1: lifting for tree-like Cutting Planes . . . 51
3.3 Main proof 2: lifting for dag-like Cutting Planes . . . 52

II Space: Algorithms for reusing memory 58

4 Upper Bounds for the Tree Evaluation Problem 59

4.1 Preliminaries . . . 60
4.2 Main proof: recursive TreeEval register programs . . . 63
4.3 A note on general fan-in TreeEval . . . 73

5 **Application: Catalytic/Amortized Algorithms for Every Function** **76**
5.1 Catalytic and amortized computation . . . 76
5.2 Main proof: time/space tradeoffs for catalytic products . . . 79
5.3 Better results for restricted functions . . . 86
5.4 Length for permutation branching programs . . . 89

6 **Conclusion** **98**
6.1 Goals for the potential contrarian . . . 98
6.2 Open problems . . . 99
6.3 Epilogue: Composition as computation . . . 105

Bibliography **107**

A **By the Wayside** **115**
A.1 Full range without Blockwise Robust Sunflower Lemma . . . 115
A.2 Graduated lifting through megacoordinates . . . 120
A.3 Quasipolynomial non-automatability of many systems . . . 125

Chapter 1

Introduction

How much harder are two tasks than one? The answer, inevitably, is that they are either twice as hard, no harder, or somewhere in between. Another way to ask the same question is the following: when is it more efficient to do two tasks together than it is to do them separately? In this thesis we study the phenomenon of *composition*, in which we are first clarifying, then instantiating, and finally answering exactly this question.

1.1 Composition

1.1.1 Computation: a compositional view

Computation, in all its myriad forms, is a method of combining simple atomic steps to efficiently solve a larger problem. To ask a computational question, we must specify three things. First, what is the problem we are looking to solve? Second, what atomic steps are we allowed? And third, what is the resource, utilized by the steps of our procedure, with respect to which we are seeking to solve our problem efficiently?

With respect to the first question, let f be a mapping from the set of n bit strings to m bit strings; our problem will be to compute f on a given input $\alpha \in \{0,1\}^n$, meaning to find the string $\beta \in \{0,1\}^m$ such that $f(\alpha) = \beta$, i.e. f maps α to β. For the second question, let $\mathcal{C}$ be a set of computational devices of interest, which will typically be the infinite family of computers which can be built from a chosen finite set of atomic rules. Finally, let s be a function assigning each device in $\mathcal{C}$ to a number, with the understanding that $s(C)$ is quantitatively conveying how much of some resource the device C is using; our goal then will be to minimize $s(C)$ over all C in $\mathcal{C}$ which successfully compute f, which we henceforth refer to as $s(f)$.

These three elements—*functions* (f), *computation models* ($\mathcal{C}$), and *complexity measures* (s)—are the defining features of theoretical computer science. The field of *algorithms* seeks to show, for a fixed function f, decreasing values of $s(f)$ by way of exhibiting newer and more innovative C computing f, for an arbitrary, or possibly fixed, model $\mathcal{C}$ and measure s. The field of *complexity theory* works against this by showing, for a fixed model $\mathcal{C}$ and measure s, larger and larger values which $s(f)$ must provably exceed, for an arbitrary, or possibly fixed, function f.

To understand these twin pillars of the field, let us return to our initial definition: computation lies in *composing* atomic steps. Imagine each basic operation as being a small vector in space, and by extension

an algorithm as being a combination of these vectors, through various joining operations, which allows us to navigate the larger space in which they lie. The art and beauty intrinsic in studying computation is in seeking out new and clever ways to combine and reorient the elements of this parsimonious basis to explore the furthest reaches of the space, and to bring more and more faraway points, by which I mean increasingly difficult functions, into the fold of computability.

By extension, when faced with the arduous task of proving complexity lower bounds, we are inevitably building a function that requires many atomic steps to be composed together in order to be computed. It requires us to assert that there exists some quantity, reflected in s, which inherently limits our basic steps, and that when we have exhausted all possibilities of combining them together to push this quantity higher and higher, we may come back with a function just one step out of reach. In so doing, we are finding the outer limit t of our algorithms, and with it we can define a hard function to be one composed out of $t+1$ (or $2t$, t^2, 2^t, $\ldots$) such steps.

By framing computation in this way it may seem that our task has become trivial: all we need to do is understand the number of atomic tasks that compose the function f. However, things are not so simple; just because we have laid out two separate tasks does not mean they take twice as long as one. To reiterate, the beauty of algorithms is in finding new and creative ways to not only combine our basic instructions but also to find new ways of orienting ourselves to the problem at hand, finding strategies where many tasks suddenly become fewer.

Thus we return to our starting question: for a given s, when is the composition of two tasks, say two functions f and g, whose composition we denote by $f \circ g$, more efficiently computable with respect to s than computing both f and g individually? Or to put it more succinctly:

Our central question: Is $s(f \circ g) \approx s(f) + s(g)$?

1.1.2 The question of complexity measures

The main variable in our central question is our choice of s. While our discussion of computation as composition aims to be as general as possible, from the perspective of actually proving lower bounds, some complexity measures may be less amenable to our task of proving results via composition. In fact our choice of s will often dictate our entire approach to proving composition lower bounds, as even the same way of composing functions f and g may work differently for different s.

To take a non-computational example, let us imagine running a set of errands. As the plural form implies, we often run errands at the same time, because a major part of each individual errand is the drive to and from downtown and so there are clear-cut advantages to using the outing to get more than one chore done at a time. Thus we can achieve an obvious gain in time, assuming our stops are not exactly in opposite directions. But how about money? Putting aside the negligible costs of riding the bus, filling up on gas, etc., making two stops in one outing rarely changes the final bill. There may be some exceptions—buy two, get one free—but as a general phenomenon we can say that money scales more or less directly with the number of errands in a way that time does not.

1.1.3 A cautionary tale: parallel repetition

To see how composition both works intuitively and fails mathematically, we will take a classic and early example: computational games. Consider a game where a group of participants are pitted against a lone

host, who will pose them each individual and secret questions which require them to coordinate without communicating. While the host knows everything that the participants may be planning, and thus can easily thwart their strategy, the game will have an element of fairness: while the type of questions will be fixed in advance, and in particular known to the players before the specific questions come, those specific questions themselves will be randomly chosen from the list of all questions available.

Here is a simple example. Two contestants Alice and Bob will each receive a single bit, independently chosen at random, and they will both output a statement of the form "Alice received a 1" or "Bob received a 0" or such. They win if they both submit the *same*, *correct* statement. It should be clear that this task is impossible to get right all the time, and in fact even with their received bits being chosen randomly they cannot do better than winning 50% of the time; no matter how complex their strategy, they are better off by simply both choosing the phrase "Alice received a 0".

A 50-50 shot may not be much, but as the host doling out the prizes we want to crush their hopes much more thoroughly. A classic strategy from the study of randomized algorithms is to just repeat this over and over again, and fail them if they ever make a mistake. If they should happen to get lucky and pass the first round, a coin flip at best, then the next round will give them another chance to fail, and from there another, and another. If we proceed in this way, a simple mathematical argument shows that their odds of winning decay exponentially, from a 1 in 2 chance in one round to a 1 in 2^n chance in n. In other words, the probability of failure—or if we adhere to the letter of our central question, $s(f) = -\log(\max_{str} \mathbf{Pr}_{a,b}(f(a,b; str(a,b)) = 1)$, where f is our game, str is all strategies Alice and Bob can use, and a and b are their respective inputs—obeys composition: $s(f \circ g) = s(f) + s(g)$ when $f \circ g$ is the game of playing f and then g in succession.

This, however, takes a long while; the time allocated to commercials creeps up every year, and all we can guarantee is one round. Naturally, our clever host figures on just giving all the bits up front, and asking for all the statements in one round. With no dependence between the games and total silence between Alice and Bob, surely makes no difference? Alas, even with just two games this fails catastrophically. Alice takes her bit from the first round and dutifully reports that she received it the first round, but oddly reports that *Bob* received *the same bit* in the *second* round! Similarly Bob takes his bit from the *second* round and dutifully reports that he received it in the second round, but also reports that Alice received that same bit in the *first* round. The error is clear: these answers will line up with the truth exactly when Alice's first bit and Bob's second bit match, bringing us back to a 50-50 shot just at the original game had. This error on the part of the host is quite shocking, and consequently quite understandable; in fact the same error appeared in early published work on the topic of games [FRS88].

Thereafter, the study of the *parallel repetition* value of games has remained active field of research. Raz [Raz95a] showed that the value of games with two participants does indeed decay exponentially, although not as good as the exponential of our original separate rounds version. For more participants their chances of winning is known to go to zero, but the speed of this convergence is still open; at current the record is roughly the inverse Ackermann function [Ver96], and results are scarcely better even for reasonably simple cases.[1]

[1] See e.g. [GHM+22] for discussion of known results.

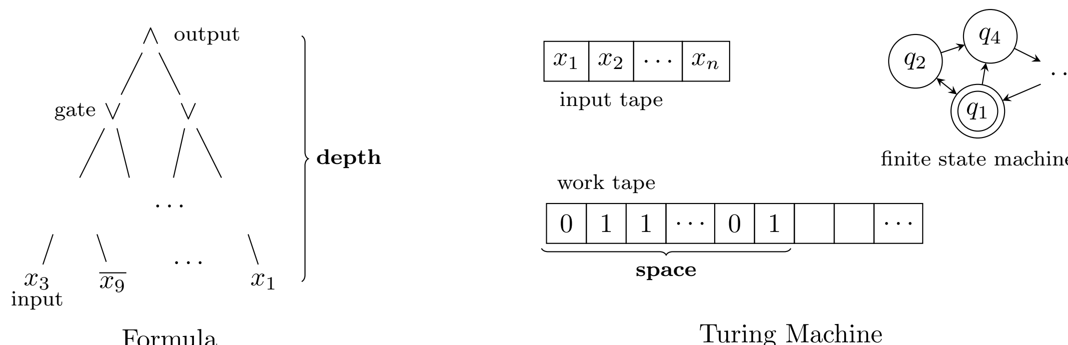

Figure 1.1: Two models of computation and complexity

1.2 Computation models and complexity measures

We now fill in the blanks by defining the $\mathcal{C}$ and s appearing in this work. Two of the earliest and most fundamental models in computer science are *logic circuits* and *Turing machines*. The complexity measures we study in this thesis will be rooted in these two models.

1.2.1 Circuits and formulas

Consider the case of *Boolean* functions f, which is the case where f outputs a single bit. A *Boolean circuit* C is a directed acyclic graph (DAG) with a single sink, where each source node (*input gate* or *leaf*) is labeled with an input variable x_i or a negated input variable $\overline{x_i}$, and where each non-source node (*gate*) is labeled with either the logical AND ($\wedge$) or logical OR ($\vee$) function. The *children* of any node v are the nodes u such that (u, v) is a directed edge in C; we also refer to these as the *inputs* of v, and for simplicity assume that every non-leaf node has exactly two inputs. Now, on input $\alpha \in \{0,1\}^n$, we can compute the *value* of any node $v \in C$, denoted $val(v)$, as follows: 1) if v is an input node labeled x_i ($\overline{x_i}$), then $val(v) = \alpha_i$ ($\overline{\alpha_i}$, respectively); 2) if v is a gate labeled $\wedge$ ($\vee$) with input gates g_1 and g_2, then $val(v) = val(g_1) \wedge val(g_2)$ ($val(g_1) \vee val(g_2)$, respectively). The value of C is the value of the sole sink node, also called the *output gate* or *root* of C, and C computes f iff $C(\alpha) = F(\alpha)$ for all $\alpha \in \{0,1\}^n$.

In this thesis we focus on the special case when C is a tree, which we refer to as a *Boolean formula*. The definition gives us a natural way to view formulas as being split into *layers*, where layer 1 consists of just the output gate, layer 2 consists of the inputs to the output gate, layer 3 consists of all inputs to the gates in layer 2, and so on. The *depth* of C will be the number of layers in C minus one, which is equivalent to the maximum length of any path starting at the root and ending at a leaf. Note that there are many other notions of complexity we could have chosen—for example, the *circuit size*, defined as either the number of nodes or the number of edges, is central to the study of circuits and formulas—but in this thesis we will focus on depth.

1.2.2 Turing Machine time and space

A *Turing Machine* is perhaps the defining computation model for all of theoretical computer science. We consider a machine M which has a read-only *input tape* where the input assignment α is written down, a write-only *output tape* where the machine will write down the final answer, and finally a read-write

work tape where it can do all its intermediate calculations; the machine itself will be a small finite state machine, which simply reads a bit of the input alongside the work tape and using only that information decides how to transition to a new configuration of the machine. This classical definition of Alan Turing is wonderfully general, abstractly capturing the type of reasoning that all "natural" computers are capable of.

There will be two complexity measures we take note of for Turing Machines. First, and most well-known, the *time* of M is defined as the number of intermediate calculations M performs. Second, and more central to this thesis, be the *space* of M is defined as the maximum number of entries on the work tape that are in use at any step in the execution of M.

1.2.3 Uniformity

In discussing syntactic models such as formulas, the question of *uniformity* arises. In short, a uniform object C is one which can be succinctly described, say by a Turing Machine which takes as input the description length $|C|$. Issues of uniformity are largely irrelevant for our discussion, so the less technical (or simply less interested) reader is free to skip this small subsection, as we will be jumping the gun on most of our later discussion. Technical readers who are already familiar with these models and the complexity classes involved may be confused by switching back and forth between uniform and non-uniform models, and so are free to use this discussion as a reference for the rest of the chapter.

When we discuss formulas in the context of separating NC^1 from P (see Question 1 below), we are necessarily considering the uniform version of NC^1, because this is the only case where separating NC^1 from P seems feasible. In particular, $\mathsf{NC}^1 \subseteq \mathsf{L}$ (again see Theorem 2 below) implies we need something more restrictive than logspace-uniformity; $DLOGTIME$-uniformity is certainly sufficient for our purposes.

However, our intuition will come from non-uniform NC^1, and our discussion of depth lower bounds in the body of the thesis will be against non-uniform models, albeit a slightly different one than formulas. What I mean by intuition is that we will be assuming we can find maximally hard functions, i.e. ones that take linear depth, but which are still in exponential time; not only is this all but impossible, but even finding *any* function requiring linear depth is *precisely the question we are trying to solve in the first place*! The answer is that we will be composing functions on a logarithmic number of inputs, meaning that the entire truth table can be written as part of the input. Thus not only is our question no longer trivial, but also we can discuss uniform NC^1 even as we focus on proving lower bounds on non-uniform NC^1, because the formula has access to the entire truth table as part of the input.

While Turing Machines are inherently uniform objects, uniformity concerns will also immediately arise for L, as we will be working with the syntactic definition of space, i.e. *branching programs*. In Chapter 4 we focus on uniform branching programs, while in Chapter 5 we focus on non-uniform branching programs; the latter will be the one time in the thesis where the distinction between the two is truly important, and so we defer further discussion to there. For uniform programs, once again our exact notion of uniformity will be unimportant.

1.3 Hardness through composition

Before studying whether or not depth and space are amenable to composition, let us see if we can make our earlier connection between composition and complexity more concrete. We begin by setting ourselves concrete, albeit extremely challenging, goals to study.

1.3.1 Complexity separations

First, let us establish our benchmarks for efficiency when it comes to depth, space, and time.

Definition 1. NC^1 is the class of all problems computable by formulas of depth $O(\log n)$. L is the class of all problems computable by Turing machines which use space $O(\log n)$. P is the class of all problems computable by Turing machines which take time $\text{poly}(n)$.

These classes are three of the earliest and most fundamental complexity classes in our field, dating back decades. Known for almost equally long is the following relationship between them.

Theorem 1. *NC^1 and L are both contained in P.*

The above theorem should not be interpreted as saying that time is a more powerful resource than depth or space; Turing Machines in P are allowed time polynomial in n, while formulas in NC^1 and Turing Machines in L are only allowed depth or space (respectively) logarithmic in n, numerically an exponentially stronger limitation. In fact, our inability to resolve the question of whether or not this exponential increase is necessary has plagued theoretical computer science for decades, and poses a fundamental question:

Open Problem 1. *Does NC^1 equal P? Does L equal P?*

It is widely believed, although perhaps somewhat dogmatically at times, that the answer to both of these questions is no. To that end, composition is one of the most, if not *the* most, well-studied method for attempting to prove these two separations in particular. Interestingly, this use of composition on both fronts seems to draw an equivalence between the weaknesses we conjecture to be inherent in both NC^1 and L, even though we have known for a long time that space is at least as powerful as depth:

Theorem 2. *NC^1 is contained in L.*

and as with P it is believed that this containment cannot be made into an equivalence:

Open Problem 2. *Does NC^1 equal L?*

We come back to Question 2 later, and for now consider how one can use composition as a method of attacking Question 1.

Let us pretend we do in fact have a function f which is hard—in fact, we will think of f as being *very* hard, requiring at least formulas with linear depth or Turing Machines with linear space to solve. Such a function f certainly exists—in fact, by a counting argument, one of the most fundamental proof techniques in computer science, almost *every* function is at least this hard—but of course we cannot guarantee that f can be solved in polynomial time, and the entire point of what we want to prove is to find an f that is not just outside of NC^1 or L, but also inside of P. The only thing we can guarantee without issue is that there exists such an f which can be solved in exponential time.

Now since we are looking for superlogarithmic depth and space lower bounds and polynomial time upper bounds, let us scale f down so it only takes $O(\log n)$ inputs. By construction, this new f will require $\Omega(\log n)$ depth and space while being solvable in $\text{poly}(n)$ time. This takes care of our upper bound, but still puts us in the realm of NC^1 and L. One instinct would be to strengthen our assumptions on f in the previous paragraph, maybe assuming that f requires superpolynomial depth or space to compute, which would scale down to superlogarithmic depth or space and complete our task. Alas,

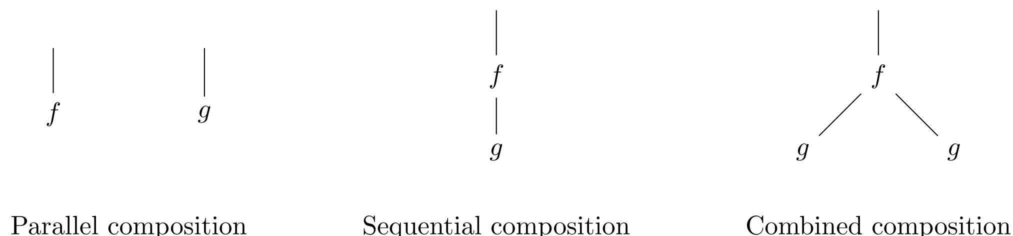

Figure 1.2: $f \circ g$: three methods of composing

assuming such additional hardness of f is a dead end; every function can be computed with formulas of linear depth, and, for reasons that will later become apparent, if we resort to this style of argument where we rely on some non-explicit hard function f, a similar statement holds for space as well.

Finally the utility of composition becomes clear: we will compose f with itself, forming a new larger function $f \circ f$. Our puzzle of whether composition leads to algorithms or complexity results becomes clear. If we can prove that $s(f \circ g) \approx s(f) + s(g)$ for all f and g, where s is either depth or space, this composition of f with itself gives us a function with twice the s-complexity as it had before. And there is no reason to stop here: performing this composition $\omega(1)$ times allows us to add more and more complexity until it finally crosses the superlogarithmic threshold.[2]

1.3.2 Intuition: composing in sequence and in parallel

Now we can return to the actual question: how do formula depth and Turing Machine space act with respect to composition? To frame this discussion, we will informally define two natural ways of composing f and g. These will not be formal definitions, and will only give a sense of how we arrive at our formal composition definition, presented in the next subsection

To compose f and g *in sequence*, we will have the inputs to f come directly from the outputs of g; think of any step-by-step process such as baking a cake, where whatever batter we prepare in one step will be further processed in the next. Is it enough to compose many copies of f in this way? Unfortunately it will not. While it is not obvious, a long skinny formula can be *balanced* to have depth logarithmic in its size, the latter quantity only scaling linearly with the amount of composition we do. For space our total usage will be the space complexity of solving one copy of f plus the space required to store the output of one copy of f, because at every stage in the composition we can throw out all the memory that is not relevant to this particular copy.

By contrast, to compose f and g *in parallel*, we will let f and g be independent of one another but require that we receive the output of both tasks at the end; this is more akin to our errands example, where we are free to order our stops as we choose but need to come home with all our purchases at the end. Alas, this version is even less useful for lower bounds. Formulas are inherently parallel objects, and so composing more functions in parallel adds nothing to the depth. Meanwhile for space we can again solve the functions one at a time, this time not even needing to remember any previous outputs as we can write them down on the output tape and then reset our memory before moving on.

However, when we put these two types of composition together, these obstructions seem to go away.

[2] In all of this our time complexity is no issue; to our point about P having exponentially more resources than NC^1 or L, iterating a polynomial time procedure a slightly superconstant number of times still gives us a polynomial runtime.

Let us compose three functions together, f with two separate copies of g, by composing the gs in parallel and composing this pair of functions with f in sequence. For formula depth, we have now created a situation where the function is already balanced, and of course f cannot appear in parallel with either g as it requires them to be computed beforehand. For space, the program can no longer add the output of either g to the output tape before moving on to the other g, as it will require both on the worktape to solve f; even though it can erase the actual work it needed to solve a function once it is done, it is the requirement to hold on to the outputs themselves that we will exploit.

For many other reasons, this form of composition is the one that has exclusively been studied in the past; neither parallel nor sequential composition has been of much interest. Why do I bother introducing them here? The reason is that while the actual composed function has always been this mixed form, the *composition theorem* which makes this function hard has implicitly focused on either sequential or parallel hardness. Looking at the discussion in the previous paragraph should make this clear; in both cases there is a predominant hard form of composition that we can only avoid on a technicality, a technicality that the other form of composition, however easy on its own, removes. For formulas we add parallel composition in order to make sequential composition nontrivial, whereas for space we add sequential composition in order to make parallel composition nontrivial.[3]

1.3.3 Hard composition problem: Tree Evaluation

We put all our discussion from this section together to propose a concrete approach to Question 1. Let $k, d, h \in \mathbb{N}$. Our problem is known by multiple names, as it appears in many different lower bound contexts; we will follow [CMW+12] and refer to it as the *tree evaluation problem*, denoted $\mathsf{TreeEval}$ in general and $\mathsf{TreeEval}_{k,d,h}$ when refering to a specific instance parameterized by k, d, and h. Our input will be a rooted tree where 1) the height of the tree, defined again as the number of edges traveled in the longest root-to-leaf path, is h; 2) every internal node has exactly d children; 3) each leaf v will be labeled with an element $x_v \in [k]$ and each internal node v will be labeled with a function $f_v : [k]^d \to [k]$. In the same way as our formula model, we inductively define the value of a node v, denoted $val(v)$ to be the label x_v if v is a leaf and $f_v(val(v_1) \dots val(v_d))$ if v is an internal node with children $v_1 \dots v_d$. The output of $\mathsf{TreeEval}_{k,d,h}$ will be the value of the root node. Note that our input has size

$$n := d^h \cdot k + (\sum_{i=0}^{h-1} d^i) \cdot k^d \log k = 2^{O(h \log d + d \log k)}$$

For both NC^1 and L we focus on the quantity $O(\log n) = O(h \log d + d \log k)$.

To start, let us see the most natural algorithm for $\mathsf{TreeEval}$, which will also show that $\mathsf{TreeEval}_{k,d,h} \in \mathsf{P}$ for all k, d, and h. Simply put, we will act in accordance with the inductive definition: starting from the leaves, we compute the value of each node in the tree in a bottom-up fashion until we reach the root. Since computing a leaf only requires reading its label and computing an internal node only requires reading the the appropriate entry in its function table, the runtime is roughly $d^h \cdot \text{poly}\, k = O(n)$.[4]

$\mathsf{TreeEval}$ has appeared in the context of proving hardness in many different literatures under many different names, and particularly for many different settings of k, d, and h; we summarize a few such

[3]This is by no means a formal statement, but in our proofs it will be clear that we implicitly focus on these two different aspects of composition in our two cases.

[4]In fact this is asymptotically smaller than n for large d and k, reflected in the fact that in each function table only one of the k^d entries is relevant to the output of the function.

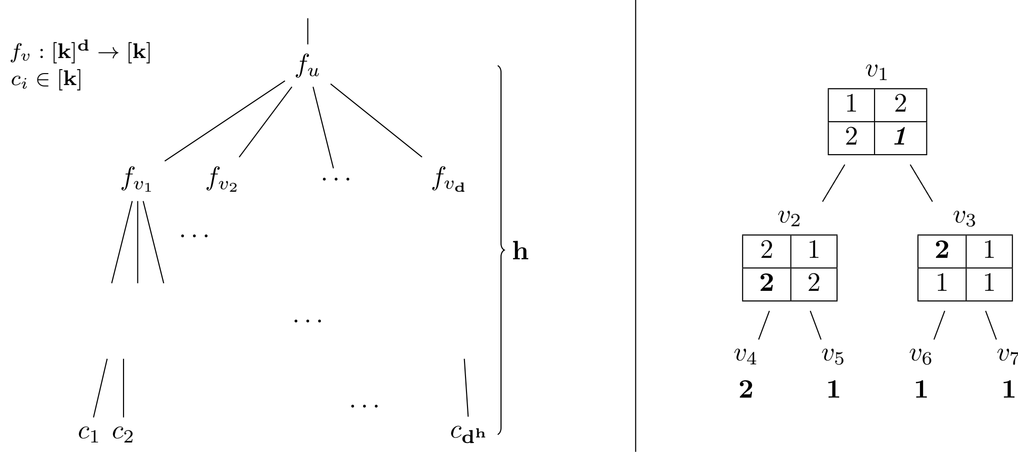

Figure 1.3: a) A general view of $\mathsf{TreeEval}_{k,d,h}$; b) An instance of $\mathsf{TreeEval}_{2,2,2}$ with output 1

settings in Table 1.1. In this thesis we will focus on two natural regimes of TreeEval parameters, one for each of our computation models.

In the realm of formulas, it was conjectured by Karchmer, Raz, and Wigderson [KRW95] that $\mathsf{TreeEval}_{2,d,h}$ requires formulas of depth $\Omega(dh) = \omega(h \log d + d \log 2)$, which would imply that $\mathsf{TreeEval} \notin \mathsf{NC}^1$.[5] Their focus, called the *KRW conjecture*, is to show that for any functions f, g and $f \circ g$ defined as $f(g(x_{1,1} \dots x_{1,m}) \dots g(x_{n,1} \dots x_{n,m}))$, it holds that $depth(f \circ g) \geq depth(f) + depth(g) - O(1)$. If this holds, then we can define a set of functions $f_1 \dots f_h : \{0,1\}^d \to \{0,1\}$, each of which requires depth $\Omega(d)$ to compute, and then for $i = 1 \dots h-1$ we apply our composition lower bound where $f = f_1 \circ \dots \circ f_i$ and $g = f_{i+1}$ to obtain our depth lower bound of $h \cdot \Omega(d) = \Omega(dh)$. The intuition is that of sequential composition: for the formula to utilize any information about some input to f, it must first compute the appropriate copy of g.

For space, Cook et al. [CMW+12] coined the term tree evaluation to refer to $\mathsf{TreeEval}_{k,2,h}$ and conjectured that it requires space $\Omega(h \log k) = \omega(h \log d + 2 \log k)$, which would imply that $\mathsf{TreeEval} \notin \mathsf{L}$.[6] To see this intuitively, let us consider an algorithm known as the *pebbling* algorithm, which will achieve non-trivial upper bounds but also strongly hint at matching lower bounds. Simply put, one can achieve savings over the trivial algorithm by forgetting any node values that are no longer relevant to computing higher nodes in the tree. A quick back-of-the-napkin argument shows that one can solve TreeEval by store at most one value per layer of the tree: starting at the node, we recursively remember the left child while computing the right. This gives us $(h-1) \cdot \log k$ bits to remember, and this basic can take us no further. Thus Cook et al. conjectured that this pebbling algorithm is indeed optimal, with the intuition that storing roughly h values is a bottleneck even if we change our approach.

With the involvement of h, it seems like there is an implicit compositional argument going on here, but it still remains to be spelled out. The hitch is that even if the pebbling *strategy* is optimal, an algorithm could still potentially save space if it was not the case that these bits all had to be remembered in *separate* memory blocks; if for any function f one could compute f while remembering $j \log k$ bits—the

[5] Shifting from [2] to $\{0,1\}$, the focus on $k = 2$ is natural since formulas are generally only defined over $\{0,1\}$. Any instance of larger k would be evaluated by $\log k$ formulas, which together output the binary description of the output. Since these formulas would all be in parallel, we gain nothing in either the upper or lower bound on formula depth.

[6] The focus on $d = 2$ here is less straightforward than the $k = 2$ focus for formulas. We include a general discussion of this choice in Section 4.3, along with a generalization of our results which renders the choice of d fairly moot.

k	d	h	Importance
1	any	any	(trivial)
any	any	1	(trivial)
≥ 2	1	3	*sequential composition*
≥ 2	≥ 2	3	*combined composition*
			canonical form of $f \circ g$
2	2	$\log n$	formula evaluation problem
			canonical NC^1-complete problem
2	$\log n$	2	multiplexor gate
2	≥ 2	3	*KRW conjecture*: *depth* $= d + d$
2	$\log n$	$\frac{\log n}{\log \log n}$	iterated multiplexor
			KRW $\rightarrow$ **IM** $\notin \mathsf{NC}^1$
≥ 2	2	3	*z-f conjecture*: *space* $= \log k + \log k$
poly n	2	$\log n$	tree evaluation problem (original usage)
			z-f $\rightarrow$ **strong evidence that TEP** $\notin \mathsf{L}$

Table 1.1: Some important parameterizations of $\mathsf{TreeEval}_{k,d,h}$

outputs of $j \ll h$ other functions in the tree—in space substantially less than $space(f) + j \log k$, then this would allow us to go below space $O(h \log k)$. This question boils down to one of parallel composition, as we want to show that solving parallel copies of g requires us to save one answer while computing the other, and so a composition theorem for space complexity would remove a major obstacle to separating L from P. In absence of another name, we will term the composition conjecture for space—that is, that the space to compute f while remembering a string z is $space(f) + |z|$—as the *z-f* conjecture.

Our stated aim of complexity theory is to show separations between classes such as NC^1 or L from others such as P, and the goal we have set for ourselves of proving composition theorems is an assertion that of all the functions in P, the one that we will have the easiest time proving separation results with is $\mathsf{TreeEval}$. What about the algorithms side? Clearly an efficient formula or space-bounded Turing Machine solving $\mathsf{TreeEval}$ exactly meets our stated aim, that of finding more and more models which can solve the problem in question. But while algorithms for $\mathsf{TreeEval}$ would not (as far as we know) imply any complexity class equivalencies, they would seem to necessitate novel techniques which could revitalize the study of computation models long considered both weak and well-tread. In particular, because algorithms for $\mathsf{TreeEval}$ will have to strongly and explicitly solve general composed functions with ease, it would give us a new framework for even more surprising algorithms, namely by casting stronger models, such as the NC hierarchy, or harder problems—perhaps the ultimate goal being the Circuit Evaluation Problem, the quintessential P-complete problem—in a framework of composition, and then using our newfound tools to capture them in much lower classes.

1.4 Our results

When studying these two models, of formula depth and Turing Machine space, we push the boundaries on what is known with respect to the composition question and provide guideposts towards future work on Question 1. Perhaps surprisingly, these results go in opposite directions; we will add to nearly two decades of support for a composition lower bound for formulas, but we will also give a space-efficient algorithm which flies in the face of both the composition framework at large and a decade's worth of

partial results supporting it. If these two statements could indeed be carried out to their limits, it would not only resolve Question 1—unexpectedly with different answers to the two questions—but also Question 2, which, while the result would be as expected, would be one of the most significant complexity results in decades.

We close our introduction by giving an overview of these results as well as laying out our roadmap for the rest of the thesis. The thesis will be split into two parts. Each part will itself be split in two: in the first chapter we lead off with our central contribution to the study of composition, while in the second chapter we show how these composition results ripple outwards from our central question and touch other areas in computer science. Our contributions appear in bold, while the central concepts to be explored in the body of the text will be italicized.

Part I Depth: Communication lower bounds for composed functions

Chapter 2 Query-to-Communication Lifting Theorems

While composition lower bounds for formulas have stubbornly eluded us, one of the strongest motivations for believing that they will one day be possible is the recurring appearance of lower bounds for composed functions in an intimately connected model: *communication complexity*, which measures the amount of information that needs to be exchanged between two players who each hold a piece of the partitioned input to f. In fact, "intimately related" is perhaps an understatement; Karchmer and Wigderson [KW90] showed an exact equivalence between formulas computing f and communication protocols computing a related function $\mathcal{S}_f$. Thus, proving composition lower bounds for these related functions would completely resolve the KRW Conjecture.

There are a number of different composition-based lines of work in the context of communication complexity; our focus will be on a very general technique known as *lifting theorems*. These theorems, also widely seen in many other areas, blossomed following a seminal proof of Raz and McKenzie [RM99], which saw new life after being interpreted as a general technique by Göös, Pitassi, and Watson [GPW18]. Together they showed that we can prove strong lower bounds —in fact, quantitatively stronger than our general composition goal in multiple ways—for any f composed with a carefully chosen g.

Query-to-Communication Lifting Theorem. *Let f be a search problem over $\{0,1\}^n$, and let $m = n^{1+\epsilon}$ for any $\epsilon > 0$. Then for $g = \textsc{Ind}_m$,*

$$\mathsf{cc\text{-}tree\text{-}depth}(f \circ g) \;=\; \mathsf{dec\text{-}tree\text{-}depth}(f) \cdot \Theta(\log m)$$

These results are too specialized to prove formula lower bounds through the connection of [KW90], but they *do* carry through for a weaker model called *monotone* formulas, giving essentially optimal lower bounds. Generalizing such results to carry through for broader classes of monotone functions would be enough to get the unrestricted formula depth lower bounds we desire, but more generally the techniques involved in proving composition lower bounds for communication protocols appear to be the right flavor for proving such lower bounds directly, and thus improving the parameters, breadth, and proof structure of query-to-communication lifting theorems is at the forefront of our attempts to settle the KRW Conjecture.

The proof of Query-to-Communication Lifting Theorem has evolved over the past two decades [RM99, GPW18, GPW20, GGKS20, LMM+22]. We can see this most clearly in its most modern

incarnation, which is our central result in this chapter. Our contributions are threefold. In terms of **simplicity**, a central piece of the proof, and the bottleneck of all previous proofs, **immediately follows from a combinatorial lemma** connected to the famous *sunflower conjecture*. In terms of **generality**, our proof **unites the central challenge of many important lifting theorems** beyond Query-to-Communication Lifting Theorem. In terms of **quantitative strength**, we obtain **quasilinear-sized gadgets for every type of lifting we study**, and furthermore **any improvements on the state of the art for the combinatorial lemma involved immediately implies lifting with smaller gadgets**.

We will prove Query-to-Communication Lifting Theorem in detail, making extensive reference to how the proof has evolved to the current state. We also provide sketches of other lifting theorems which follow from our new proof, namely dag-like and graduated lifting.

Chapter 3 Application: Cutting Planes Proofs are Hard to Find

Understanding formula depth is only one of the many ways in which we use communication complexity, possibly one of the most well-connected subfields in complexity theory. As a result, lifting theorems, which are among the strongest and most versatile techniques for proving lower bounds in communication, can also yield important results in other subfields.

One application of query-to-communication lifting is in *proof complexity*, where query complexity and communication complexity are respectively tied to reasoning in the *Resolution* and *Cutting Planes* systems. These systems are well-studied and foundational in proof complexity, but are also highly practical in that their proofs capture the most widely used methods of solving SAT instances and integer programs. Thus efficiently finding small proofs in these systems is of great importance. However, a recent breakthrough result of Atserias and Müller [AM20] strongly rules this out for Resolution, showing that finding short Resolution proofs in polynomial (subexponential) time is hard assuming $\mathsf{P} \neq \mathsf{NP}$ (the *exponential time hypothesis* (ETH), respectively). A version of this result was extended to tree-like Resolution by de Rezende [dR21] using a similar proof.

Using query-to-communication lifting, we extend these results to Cutting Planes, thus ruling out worst-case efficiency for the two most widely used automated prover systems.

Cutting Planes Non-Automatability Theorem. *Let $\mathcal{A}$ be an algorithm which, on input τ which is a unsatisfiable set of m linear equations over n variables which has a Cutting Planes refutation of size s, outputs a Cutting Planes refutation of $\mathcal{A}$. Then assuming $\mathsf{P} \neq \mathsf{NP}$ (assuming the Exponential Time Hypothesis), $\mathcal{A}$ requires time $N^{\omega(1)}$ ($2^{\Omega(N)}$, respectively), where $N = \max(n, m, s)$.*

Let $\mathcal{A}$ be an algorithm which, on input τ which is a unsatisfiable set of m linear equations over n variables which has a tree-like Cutting Planes refutation of size s, outputs a tree-like Cutting Planes refutation of $\mathcal{A}$. Then assuming the Exponential Time Hypothesis, $\mathcal{A}$ requires time $N^{\Omega(\log N/\log^2 \log N)}$, where $N = \max(n, m, s)$.

We use two novel lifting theorems for these respective tasks. The first is the graduated lifting theorem proven in the previous chapter, adapted to the *real* communication setting; this was in fact the original context in which graduated lifting was proven. The second allows us to lift based on the *block-width* of decision-dags, rather than the classic notion of width.

Part II Space: Algorithms for reusing memory

Chapter 4 Upper Bounds for the Tree Evaluation Problem

After introducing TreeEval and conjecturing that the pebbling algorithm is optimal, a number of works have shown that this conjecture indeed holds for various restricted models of space-bounded computation. For nearly as long, however, there has been a counter-current seeking to show that, should composition theorems fail, there may be ways of solving TreeEval that surpass pebbling. This work, based on classical results on reusing space, was codified by an elegant work of Buhrman et al. [BCK+14], where they defined an alternate notion of space-bounded Turing Machines which they dubbed *catalytic computing*.

Putting aside its interest as a model in its own right, their results directly challenged the idea that used memory cannot be repurposed in place of using free space; simply put, catalytic computing is the study of where the z-f conjecture fails. While this was inspired by the potential obstruction to TreeEval lower bounds, however, it did not immediately lead to any new algorithms which quantitatively surpass pebbling, and the algorithms it defined avoided some but not all of the restrictions for which we have already proven TreeEval lower bounds matching pebbling.

Our work finally achieves an algorithm that beats pebbling unconditionally, using catalytic computing to refute the z-f conjecture, or in other words, to refute composition for space-bounded computation, and using that refutation we achieve the first new upper bounds for TreeEval since the problem's inception.

Tree Evaluation Algorithm. *For any k and h, $\mathsf{TreeEval}_{k,2,h}$ can be solved in space $O(h \log k / \log h) = o(h \log k)$.*

Our first challenge was to use the tools of [BCK+14], which were specialized for doing arithmetic operations in low space, for the arbitrary input functions of TreeEval. This goes by a basic *arithmetization* step, i.e. interpolation, which incidentally allows us to avoid a restriction which would immediately imply strong lower bounds against our algorithm. **This also has implications for the use of catalytic tools on more general functions.**

The second challenge is handling the high degree polynomials coming out of our arithmetization using arithmetic tools built for functions with low-degree and nice structure. This requires us to build new tools for the catalytic program, namely *catalytic product lemmas* built both to handle individual large products and to do many such products in parallel. **Our lemmas can be optimized for space or for time**, both of which exactly match the pebbling algorithm, but **by balancing them out we get an algorithm for TreeEval surpassing both**.

Chapter 5 Application: Catalytic/Amortized Algorithms for Every Function

In meeting our first challenge for TreeEval, we built a framework for obtaining space upper bounds against arbitrary functions. We can use this insight to directly consider the catalytic space complexity of *arbitrary* functions, and in particular we can consider the *non-uniform* variant whereby every function, even uncomputable ones, have well-understood upper bounds. Non-uniform space complexity corresponds to the classic syntactic model of *branching programs*, and a natural catalytic variant was introduced by Girard, Koucky, and McKenzie as *m-catalytic branching programs*.

Potechin [Pot17] observed that the definition of m-catalytic branching programs coincides with a natural definition of *amortized* space complexity, again in the non-uniform setting. After drawing this connection, [Pot17] proves an astonishing result: for m sufficiently large, there exist m-catalytic branching

programs of size $O(mn)$ for *every* function! This translates to two facts: 1) every function has a linear time non-uniform algorithm using $\log m$ catalytic bits and *no non-catalytic work space*, aside from keeping track of the current step; 2) every function can be computed with *linear amortized space*.

The only catch in the result of [Pot17] is that the value of m is enormous: $m = 2^{2^n - 1}$. Reducing m while maintaining $O(mn)$ size would improve on both interpretations, showing that less catalytic space and amortized copies are necessary. Taking up this challenge, Robere and Zuiddam [RZ21] show that m can be reduced if the function in question has small degree when arithmetized over $\mathbb{F}_2$. Unfortunately this result fails to improve the state of affairs for most functions.

Our TreeEval techniques allow us to reduce this value significantly while maintaining the minimal amortized/catalytic space complexity.

General Catalytic/Amortized Algorithm. *For any $\epsilon \geq 2/n$ and any function $f : \{0,1\}^n \to \{0,1\}$ there is an m-catalytic branching program with length $2^{1/\epsilon} \cdot 2\epsilon n$ and width $2m$ that computes f, where $m = 2^{n + \epsilon^{-1} \cdot 2^{\epsilon n}}$.*

Furthermore, if f is an $\mathbb{F}_2$ polynomial of degree at most d and $\epsilon \geq 2/d$, there is an m-catalytic branching program with length $2^{1/\epsilon} \cdot 2n$ and width $2m$ that computes f, where $m = 2^{n + \epsilon^{-1}\binom{n}{\leq \epsilon d}}$.

In particular, for $\epsilon = \Theta(1)$, General Catalytic/Amortized Algorithm gives us an m-catalytic branching program of length $O(n)$ where $m \leq 2^{2^{\delta n}}$ $(\leq 2^{\binom{n}{\leq \delta d}})$ for as small a constant δ as we choose. We also note that for $\epsilon = \Theta(1) \cdot \log^{-1} n$ this gives us an m-catalytic branching program of length $\mathrm{poly}(n)$ where $m = 2^{2^{n/K \log n}}$ $(\leq 2^{\binom{n}{\leq d/K \log n}}$, respectively) for as large a constant K as we choose.

Using [RZ21] as a starting point, we return to [Pot17] with the view of $\mathbb{F}_2$ polynomials in mind, and recreate their result using catalytic product lemmas of the same form as our previous results. Then, **using the same time-space tradeoff as before, we can reduce the exponent of $\log m$—i.e. the double exponent of m—while only paying in the constant in front of the runtime.** Returning to [RZ21], **we obtain the exact same improvement to their smoothed result with very little change to our algorithm**.

We conclude the thesis in **Chapter 6: Conclusion** by discussing the next steps for understanding composition and its applications. We touch on all the ways that these results and any future progress are connected to open problems all around theoretical computer science.

Works used

The results of this thesis previously appeared in the following works:

- Chapter 2

  1. Shachar Lovett, Raghu Meka, Ian Mertz, Toniann Pitassi, Jiapeng Zhang. *Lifting with Sunflowers*. In Proc. 19th Innovations in Theoretical Computer Science (ITCS'22), pp. 104:1-24, 2022. [LMM+22]

- Chapter 3

  1. Mika Göös, Sajin Koroth, Ian Mertz, Toniann Pitassi. *Automating Cutting Planes is NP-Hard*. In Proc. 52nd Symposium on Theory of Computing (STOC'20), pp. 68-77, 2020.

2. Ian Mertz, Toniann Pitassi, Yuanhao Wei. *Short Proofs Are Hard to Find.* In Proc. 46th International Colloquium on Automata, Languages and Programming (ICALP'19), pp. 1-16, 2019. [GKMP20]

- Chapter 4

1. James Cook, Ian Mertz. *Encodings and the Tree Evaluation Problem.* Technical note, 2021. [CM21]
2. James Cook, Ian Mertz. *Catalytic Approaches to the Tree Evaluation Problem.* In Proc. 52nd Symposium on Theory of Computing (STOC'20), pp. 752-760, 2020. [CM20]

- Chapter 5

1. James Cook, Ian Mertz. *Trading Time and Space in Catalytic Branching Programs.* In 37th Computational Complexity Conference (CCC'22), pp. 8:1-21, 2022. [CM22]

Part I

Depth: Communication lower bounds for composed functions

Chapter 2

Query-to-Communication Lifting Theorems

In the first technical part of the thesis, we will make progress towards showing $depth(f \circ g) \approx depth(f) + depth(g)$, where $f \circ g = f(g_1 \dots g_n)$.

As discussed in Chapter 1, our intuition as to the power of composition theorems for formula depth comes from an important connection to another model called *communication complexity* [Yao79]. In this model, all functions F have their inputs split between computationally unbounded players Alice and Bob, and the communication complexity of F is the number of bits they need to communicate to one another about their respective inputs to solve the function. A beautiful connection by Karchmer and Wigderson [KW90] (see Definition 3 and Theorem 3 below) shows that the formula depth of any function f is precisely characterized by the communication complexity of a related two-party relation $\mathcal{S}_f$, where Alice and Bob are respectively given a 0 and 1 input to f and are tasked with finding a bit where their inputs differ.

While we do not have a composition theorem for $\mathcal{S}_{f \circ g}$—this would imply the KRW conjecture and thus resolve $\mathsf{TreeEval} \notin \mathsf{NC}^1$—other forms of composition results have a long and storied history in communication complexity. For example, for a given function $f : \{0,1\}^n \to \{0,1\}$, instead of composing with another function $g : \{0,1\}^m \to \{0,1\}$ and considering $\mathcal{S}_{f \circ g}$, we can consider a two-party function $g : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$, and thus $f \circ g$ immediately constitutes a communication problem in the natural way. A *lifting theorem* gives us a composition theorem in this setting, but with two small differences: first, rather than focusing on the case of arbitrary inner two-party functions g, lifting theorems focus on a carefully chosen hard function, often called the *gadget*; and second, we will not show $\mathsf{cc\text{-}tree\text{-}depth}(f \circ g) \approx \mathsf{cc\text{-}tree\text{-}depth}(f) + \mathsf{cc\text{-}tree\text{-}depth}(g)$—in fact $\mathsf{cc\text{-}tree\text{-}depth}(f)$ is not well-defined in this context—but rather than $\mathsf{cc\text{-}tree\text{-}depth}(f \circ g) \approx t(f) \cdot \mathsf{cc\text{-}tree\text{-}depth}(g)$, where $t(\cdot)$ is an appropriate complexity model for the one-party function f.

Before moving into discussing lifting, we address the connection between this and the KRW conjecture. In many ways the multiplicative form given above seems too good to hope for for KRW; even beyond the issue of focusing on a specific gadget rather than general g, it would go against the subaddititivity of $\mathsf{cc\text{-}tree\text{-}depth}$ to hope for a multiplicative composition theorem for $\mathcal{S}_{f \circ g}$. In fact, the nature of $\mathcal{S}_f$ in the Karchmer-Wigderson connection means that lifting can only ever give us *monotone* formula lower bounds. Thus, lifting should be viewed more as a prolific line of work which informs our attempts to resolve the

KRW conjecture; in this capacity it is useful for two reasons. First, strong lower bounds for a sufficiently broad class of monotone functions, i.e. *slice functions*, would be enough to prove lower bounds in the non-monotone case as well. And second, most modern work on KRW uses arguments derived from lifting, as these proofs have been very successful at showing compositional lower bounds.

Thus we now turn to see what lifting results are known and what they can accomplish on their own terms. The first result was the query-to-communication lifting theorem of Raz and McKenzie [RM99, GPW18], which states that that for any function $f : \{0,1\}^n \to \{0,1\}$ (or more generally any search problem) and a gadget function called the *index gadget* $IND_m : [m] \times \{0,1\}^m \to \{0,1\}$, the communication complexity of the composed function $f \circ IND_m$ is roughly the *decision tree complexity* of f times the decision tree complexity of IND_m, where decision trees are a very weak model of computation which can only query variables one at a time.

The multiplicative lower bound on $\mathsf{cc\text{-}tree\text{-}depth}(f \circ g)$, plus the fact that t is a much weaker model of complexity than $\mathsf{cc\text{-}tree\text{-}depth}$, allows us to prove very strong lower bounds even for relatively simple functions f. These lifting theorems have yielded many fantastic results; in particular, since lifting gives us optimal communication lower bounds, using Karchmer-Wigderson they also give us the strongest known monotone formula lower bounds. Beyond formula lower bounds, there is a substantial body of work proving lifting theorems for a variety of flavors of query-to-communication, including but not limited to: deterministic [RM99, GPW18, dRNV16, WYY17, CKLM19, CFK$^+$21], nondeterministic [GLM$^+$16, Göö15], randomized [GPW20, CFK$^+$21], degree-to-rank [She11, PR17, PR18, RPRC16], and non-negative degree to nonnegative rank [CLRS16, KMR17]. In these papers and others, lifting theorems have been applied to simplify and resolve longstanding open problems, including new separations in (again including but not limited to): communication complexity [GP18, GPW18, GPW20, CKLM19, CFK$^+$21], proof complexity [GLM$^+$16, HN12, GP18, dRNV16, dRGN$^+$21] monotone circuit complexity [GGKS20], monotone span programs and linear secret sharing schemes [RPRC16, PR17, PR18], and lower bounds on the extension complexity of linear and semi-definite programs [CLRS16, KMR17, LRS15]. Furthermore within communication complexity many natural functions of interest—e.g. equality, set disjointness, inner product, gap-hamming (c.f. [Kus97, Juk12])—are also lifted functions themselves.

In this first part of the thesis, we will study lifting theorems, and seek to push this enormous body of work further, both in progressing towards an optimal lifting theorem in hopes that such a proof could one day help us reach our true goal of resolving KRW, as well as proving quantitatively similar lifting theorems for more models with useful applications. Our main focus will be to prove Query-to-Communication Lifting Theorem, itself a strengthening of the results of [RM99, GPW18], which builds off the existing proofs which have developed over the past two decades plus a novel connection between lifting to combinatorics.

In particular, we use a key lemma developed by Alweiss et al. [ALWZ20] in the process of improving the state of the art of the famous *sunflower lemma*. We emphasize here, however, that earlier iterations of the same lemma also work and give similar, albeit weaker, improvements; this technique is novel to our work, but the pieces have been around for at least a decade. In fact, connecting lifting to sunflowers has much deeper implications than just the possibility of future progress, a point we discuss at the end of the chapter.

In terms of concrete benefits to our work, as discussed in the intro this proof has three major advantages:

Simplicity. To simulate a communication protocol by a decision tree, we use a basic *round-by-round simulation* while using the deficiency of *min-entropy* as our potential function. This simulation has two basic invariants connecting the current state of the protocol to the variables of our decision tree: 1) the protocol is consistent with the values of all variables queried thus far; and 2) the protocol has very high entropy on all variables not queried thus far. Maintaining this invariant goes by a *density-restoring partition* argument, while the central lemma, stating that a good partition can be found, will follow directly from the aforementioned combinatorial lemma.

Generality. Our new proof of the central lemma actually proves a much stronger lemma, which becomes necessary for stronger lifting theorems such as the *dag-like* version [GGKS20]. It also changes the reliance on the *gadget size* in such a way that *graduated* lifting [GKMP20], i.e. lifting whose gadget size scales with the strength of the lower bound, follows with no changes in the proof.

Quantitative strength. A central goal of modern lifting is to improve on the gadget size of the argument, as this parameter immediately impacts, and indeed is the bottleneck to, most applications of lifting to other fields. Our contribution with respect to the gadget size is twofold. First, we improve on the previous upper bound, which was quadratic for basic lifting and an enormous polynomial for most other types. We obtain quasilinear-sized gadgets for every type of lifting we study; furthermore the size can be further improved if one is willing to sacrifice in the upper bound, up to $n \log n$. Second, we consolidate all reliance on the gadget size into our application of the combinatorial lemma, meaning that any improvements on the state of the art for sunflowers immediately implies lifting with smaller gadgets.

2.1 Preliminaries

2.1.1 Communication complexity

We begin by formally introducing communication complexity, and draw a connection between communication and formulas. A *search problem* is a relation $f \subseteq \mathcal{Z} \times \mathcal{O}$, and we let denote $f(z)$ denote the set of all $o \in \mathcal{O}$ such that $(z, o) \in f$. Likewise a *bipartite search problem* is a relation $F \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{O}$ and $F(x, y)$ is defined analogously.

Definition 2 (Communication)**.** Consider a bipartite search problem F. A *communication protocol* Π is a binary tree where now each non-leaf node v is labeled with a binary function g_v which takes its input either from $\mathcal{X}$ or $\mathcal{Y}$. This is informally viewed as two players Alice and Bob jointly computing a function, where Alice receives $x \in \mathcal{X}$ and Bob receives $y \in \mathcal{Y}$, and where at each node in the protocol, depending on whose turn it is, either Alice computes $g_v(x)$ or Bob computes $g_v(y)$, and whoever does "speaks" as to which child to go to. The protocol Π solves F if, for any input $(x, y) \in \mathcal{X} \times \mathcal{Y}$, the unique root-to-leaf path, generated by walking left at node v if $g_v(x, y) = 0$ (and right otherwise), terminates at a leaf u with $o_u \in F(x, y)$. We define

$$\textsf{cc-tree-depth}(F) \ := \ \text{least } \textit{depth} \text{ of a communication protocol solving } F.$$

An alternative characterization of communication protocols, which will be useful for proving our main theorem, is as follows. Each non-leaf node v is labeled with a *combinatorial rectangle* (henceforth *rectangle*) $R_v = X_v \times Y_v \subseteq \mathcal{X} \times \mathcal{Y}$, such that if v_ℓ and v_r are the children of v, R_{v_ℓ} and R_{v_r} partition R_v.

Furthermore this partition is either of the form $X_{v_\ell} \times Y_v \sqcup X_{v_r} \times Y_v$ (if Alice speaks) or $X_v \times Y_{v_\ell} \sqcup X_v \times Y_{v_r}$ (if Bob speaks). The unique root-to-leaf path on input (x, y) is generated by walking to whichever child v of the current node satisfies $(x, y) \in R_v$.

In order to connect (one-party) formulas to (two-party) communication, we need a connection between functions and bipartite search problems.

Definition 3 (Canonical search problem)**.** Let $f \subseteq \{0,1\}^n \to \{0,1\}$ be Boolean function. The *canonical search problem* associated with f, denoted $\mathcal{S}_f \subseteq \{0,1\}^n \times \{0,1\}^n \times [n]$, is the set of all pairs (x, y, i) such that 1) $x \in f^{-1}(1)$; 2) $y \in f^{-1}(0)$; and 3) $x_i \neq y_i$.

Thus a communication protocol for $\mathcal{S}_f$ can be described as Alice receiving a 1-input to f, Bob receiving a 0-input to f, and their goal is to find a spot where their respective inputs differ. It turns out that such protocols are completely isomorphic to formulas solving f itself; by extension their complexities match.

Theorem 3 (Main theorem, [KW90])**.** $\mathsf{cc\text{-}tree\text{-}depth}(\mathcal{S}_f) = \mathit{depth}(f)$

We note that as mentioned in Chapter 1, formulas can be *balanced* to have depth at most logarithmic in the optimal size; Theorem 3 implies the same for communication complexity.

2.1.2 Lifting

In this work we focus on a different way of connecting search problems to bipartite search problems, namely by composition. As stated in the introduction, we will not consider the composition of general f and g, but rather the composition of general f with a specialized gadget function g.

Definition 4. Let $m \in \mathbb{N}$. The *index gadget*, denoted $\textsc{Ind}_m$, is a Boolean function which takes two inputs $x \in [m]$ and $y \in \{0,1\}^m$, and outputs $y[x]$. For a search problem $f : \{0,1\}^n \to \mathcal{O}$, the *lifted search problem* $f \circ \textsc{Ind}_m^n$ is a bipartite search problem defined by $\mathcal{X} := [m]^n$, $\mathcal{Y} := (\{0,1\}^m)^n$, and $f \circ \textsc{Ind}_m^n(x, y) = \{o \in \mathcal{O} : o \in f(\textsc{Ind}_m^n(x, y))\}$, where $\textsc{Ind}_m^n$ refers to n separate instances of $\textsc{Ind}_m$.

Our goal in lifting will be to turn lower bounds for the original search problem f into communication lower bounds for the lifted search problem $f \circ \textsc{Ind}_m^n$. The lower bounds for f will be for a model of computation called *query complexity*.

Definition 5. Consider a search problem $f \subseteq \{0,1\}^n \times \mathcal{O}$. A *decision tree* T is a binary tree such that each non-leaf node v is labeled with an input variable z_i, and each leaf v is labeled with a solution $o_v \in \mathcal{O}$. The tree T solves f if, for any input $z \in \{0,1\}^n$, the unique root-to-leaf path, generated by walking left at node v if the variable z_i that v is labeled with is 0 (and right otherwise), terminates at a leaf u with $o_u \in f(z)$. We define

$$\mathsf{dec\text{-}tree\text{-}depth}(f) \;:=\; \text{least } \mathit{depth} \text{ of a decision tree solving } f.$$

As far as complexity measures go, the parameter dec-tree-depth is typically the one that we care about, and certainly for lifting our goal will be to connect dec-tree-depth to cc-tree-depth, but in Chapter 3 we will also draw attention to dec-tree in the context of proof complexity. One important point here is that unlike communication protocols, decision trees can *not* be balanced,[1] which means that in lifting

[1] A trivial example of this is the OR function with n inputs, which can be computed by decision trees of size n but also requires depth n.

the *depth* of decision trees to communication, we may gain exponentially in the lower bound on their respective *sizes* as well.

However, while lifting allows us to obtain strong communication lower bounds against lifted search problems, we are not guaranteed that $f \circ \textsc{Ind}_m^n = \mathcal{S}_{f'}$ for some function f', which is why these lower bounds do not automatically give us results for formulas. However, it turns out that there is a generic transformation that can turn any bipartite search problem F into a new problem F' which is indeed a *monotone* canonical search problem for some function f', and composing with a monotone analogue of Theorem 3 [KW90]—we get the following[2]:

Theorem 4 (Lifted functions to monotone formulas)**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a function and $g : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$ be a bipartite function. Then there exists a function $f' : \{0,1\}^{n'} \to \{0,1\}$ such that*

$$\mathsf{cc\text{-}tree\text{-}depth}(f \circ g) = \mathsf{cc\text{-}tree\text{-}depth}^+(\mathcal{S}_{f'}) = depth^+(f')$$

where the $+$ subscript indicates monotonicity.

The KRW conjecture, and by extension $\mathsf{NC}^1 \neq \mathsf{P}$, would follow from two improvements: 1) generalizing our lifting results to work for any g instead of just $\textsc{Ind}_m$; and 2) removing the monotonicity in Theorem 4.

2.1.3 Abstract and dag-like query/communication models

The above definitions are the most recognizable form of query and communication models respectively, but we can consider these models in a more general form as well. We consider a search problem $f \subseteq \mathcal{Z} \times \mathcal{O}$, and let $\mathcal{Q}$ be a family of subsets of $\mathcal{Z}^n$. A $\mathcal{Q}$-tree T for f is a tree where each internal vertex v of T is labeled with a function $q_v \in \mathcal{Q}$, each leaf vertex of T is labelled with some $o \in \mathcal{O}$, and all labels satisfy the following properties:

- $q_v = \mathcal{Z}^n$ when v is the root of T
- $q_v \subseteq q_u \cup q_w$ for any node v with children u and w
- $q_v \subseteq f^{-1}(o)$ for any leaf node v labeled with $o \in \mathcal{O}$

We can see that for $\mathcal{Z} = \{0,1\}^n$, ordinary decision trees are $\mathcal{Q}$-trees where $\mathcal{Q}$ is the set of strings accepted by juntas (i.e. conjunctions of literals).

We can also use $\mathcal{Q}$-trees to generalize communication complexity search problems $F \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{O}$, where now $\mathcal{Q}$ is the family of functions from $\mathcal{X} \times \mathcal{Y}$ to $\{0,1\}$ corresponding to combinatorial rectangles $X \times Y \subseteq \mathcal{X} \times \mathcal{Y}$; more specifically $q^{X \times Y}(x,y) = 1$ iff $(x,y) \in X \times Y$.

Our abstract models will have the added benefit of allowing us to properly define a dag-like model for both query and communication [Raz95b, Pud10, Sok17]. For a search problem $f \subseteq \mathcal{Z} \times \mathcal{O}$ and a family of functions $\mathcal{Q}$ from $\mathcal{Z}$ to $\{0,1\}$, a *$\mathcal{Q}$-dag* is a directed acyclic graph D where each internal vertex v of the dag is labeled with a function $q_v(z) \in \mathcal{Q}$ and each leaf vertex is labeled with some $o \in \mathcal{O}$ and satisfying the following properties:

- $q_v^{-1}(1) = \mathcal{Z}$ when v is the root of D
- $q_v^{-1}(1) \subseteq q_u^{-1}(1) \cup q_w^{-1}(1)$ for any node v with children u and w

[2]We could not find the first reference to the connection between lifted problems and monotone canonical search problems, and so we assume Theorem 4 is essentially folklore given the monotone results of [KW90].

- $q_v^{-1}(1) \subseteq f^{-1}(o)$ for any leaf node v labeled with $o \in \mathcal{O}$

For $\mathcal{Z} = \{0,1\}^n$ a *conjunction dag* D solving f is a $\mathcal{Q}$-dag where $\mathcal{Q}$ is the set of all juntas over $\mathcal{Z}$. [3] For conjunction dags our measure of complexity will be a bit different than depth. The *width* of Π is the maximum number of variables occurring in any junta $v(z)$. We define

$$\begin{aligned} \textsf{dec-dag}(S) &\coloneqq \text{ least } \textit{size} \text{ of a decision-dag solving } S, \\ \textsf{dec-dag-width}(S) &\coloneqq \text{ least } \textit{width} \text{ of a decision-dag solving } S. \end{aligned}$$

For a bipartite search problem $F \subseteq \mathcal{X} \times \mathcal{Y} \times \mathcal{O}$ and a family of functions $\mathcal{Q}$ from $\mathcal{X} \times \mathcal{Y}$ to $\{0,1\}$, we define a $\mathcal{Q}$-dag solving F analogously. A *rectangle dag* Π solving F is a $\mathcal{Q}$-dag where $\mathcal{Q}$ is the set of all indicator vectors of rectangles $X \times Y \subseteq \mathcal{X} \times \mathcal{Y}$. We define

$$\textsf{rect-dag}(F) \coloneqq \text{ least } \textit{size} \text{ of a rectangle dag solving } F.$$

2.2 Main proof: tree-like lifting via sunflowers

Our main goal will be to give a novel proof of the deterministic query-to-communication lifting theorem of [RM99, GPW18]:

Query-to-Communication Lifting Theorem. *Let f be a search problem over $\{0,1\}^n$, and let $m = n^{1+\epsilon}$ for any $\epsilon > 0$. Then for $g = \textsc{Ind}_m$,*

$$\textsf{cc-tree-depth}(f \circ g) = \textsf{dec-tree-depth}(f) \cdot \Theta(\log m)$$

To prove Query-to-Communication Lifting Theorem, we prove that a) a decision tree of depth d for f can be simulated by a communication protocol of depth $O(d \log m)$ for the composed problem $f \circ \textsc{Ind}_m^n$, and b) a communication protocol of depth $d \log m$ for the composed problem $f \circ \textsc{Ind}_m^n$ can be simulated by a decision-tree of depth $O(d)$ for f. Let $\{z_i\}_i$ be the variables of f and let $\{x_i\}_i, \{y_i\}_i$ be the variables of $f \circ \textsc{Ind}_m^n$; recall that each z_i takes values in $\{0,1\}$, x_i takes values in $[m]$, and y_i takes values in $\{0,1\}^m$. The forward direction of the theorem is obvious: given a decision tree T for f, Alice and Bob can simply trace down T and compute the appropriate variable z_i at each node $v \in T$ visited, spending $\log m$ bits to compute $\textsc{Ind}_m(x_i, y_i)$ to do so.

Thus our goal is to prove that if there exists a communication protocol Π of depth $d \log m$ for the composed problem $f \circ \textsc{Ind}_m^n$, then there exists a decision tree of depth $O(d)$ for f. Our proof will follow the basic structure of previous works [GPW20, GGKS20]. At the heart of lifting proofs is a *simulation theorem*,[4] which shows that for large enough $m = n^{O(1)}$ the simulation goes the other way as well: we can build a decision tree for f by mimicking a communication protocol for $f \circ \textsc{Ind}_m^n$ while only querying an $O(1/\log m)$ fraction of the original variables.

The proof of this simulation theorem has evolved considerably since [RM99], applying to a wider range of gadgets [WYY17, CKLM19, CFK$^+$21], and with more sharpened results giving somewhat improved

[3] As noted above the terms "conjunction" and "junta" are closely related, but conjunctions are usually thought of as syntactic objects while juntas are functions. We keep the term conjunction dag from [GGKS20] for consistency even though we switch to using junta for the functions in $\mathcal{Q}$.

[4] Here we restrict ourselves to lifting theorems in the setting of Boolean models of query complexity (e.g., decision trees, randomized decision trees). Interestingly *algebraic* lifting theorems which lift polynomial degree to an associated communication measure, exploit duality in order to give nonconstructive proofs of lifting (see e.g. [She11, PR18, Rob18])

parameters and simulation theorems for the more difficult settings of randomized and dag-like lifting. The original proof of [RM99] used the notion of min-degree for the central invariant used to prove the simulation theorem; later [GLM+16] introduced the notion of blockwise min-entropy, which has since been used for a variety of lifting theorems, including randomized [GPW20] and dag-like [GGKS20]. Nearly all of these proofs used either intricate combinatorial arguments or tools from Fourier analysis.

After proving Query-to-Communication Lifting Theorem we will move on to other lifting theorems which follow by similar proofs. In particular we prove a lifting theorem for *dag-like* protocols as well as a *graduated* lifting theorem whose gadget size scales with $\mathsf{dec\text{-}tree\text{-}depth}(f)$. Almost all our proofs will generalize to the *real communication* setting, which will be the focus of Chapter 3.

2.2.1 High level idea: Tracing the "important" coordinates.

What does it mean to "simulate" a communication protocol for $f \circ \text{IND}_m^n$ by a decision tree for f? When we look at the communication matrix for $f \circ \text{IND}_m^n$, we label the (x, y) entry with the solutions $o \in \mathcal{O}$ satisfying $(x, y) \in (f \circ \text{IND}_m^n)^{-1}(o)$. However we have no control over f, and so in some sense what we really care about is the z variables. So instead we will think of the (x, y) entry as storing $z = \text{IND}_m^n(x, y)$, and then instead of having to reason about f we can ask "what does the set of z values that make it to any given leaf of Π look like?"

For each leaf we want to split the coordinates into two categories: the "important" coordinates where the z values are (jointly) nearly fixed, and the rest where every possibility is still open. Hopefully this means that knowing the important coordinates is enough to declare the answer. Applying the same logic to the internal nodes we can query variables as they cross the threshold from unfixed to important, which leads us down to the leaves in a natural way. To do this efficiently, we have to define "importance" in a way that satisfies all these conditions while also ensuring that no leaf contains more than $O(d)$ important variables.

In order to prove this formally, we will trace down the communication protocol node by node, at each step looking for the z variables that are fairly "well determined" by the current rectangle. We focus exclusively on the X side of the current rectangle, since Y is so large that it would take more than $d \log m$ rounds just to fix a single y_i. Our measure of coordinate i being well-determined will be the *min-entropy* of the uniform distribution on X marginalized to the coordinate i.

Definition 6. Let S be a set. For a random variable $\boldsymbol{s} \in S$ we define its *min-entropy* by $\mathbf{H}_\infty(\boldsymbol{s}) := \min_s \log(1/\mathbf{Pr}[\,\boldsymbol{s} = s\,])$. We also define the *deficiency* of $\boldsymbol{s}$ by $\mathbf{D}_\infty(\boldsymbol{s}) := \log |S| - \mathbf{H}_\infty(\boldsymbol{s}) \geq 0$.

In this paper we will use the convention that bolding the name of a set means the random variable which is uniform over the set. Thus our focus will be on $\mathbf{H}_\infty(\mathbf{X}_i)$.

Let X_v be the X set associated with node v in the communication protocol. By definition of a communication protocol, if v has children u and w, then $X_v \subseteq X_u \cup X_w$, and so by averaging one of the two children—let us say u without loss of generality—obeys $|X_u| \geq |X_v|/2$. An obvious but crucial fact about min-entropy is that the min-entropy of every coordinate in X_u at least its entropy in X_v minus 1. Thus, at the start of the protocol every coordinate will have min-entropy exactly $\log m$, while moving to the child with the larger X_u set guarantees that in each round the min-entropy of every coordinate drops by at most 1.

This immediately leads us to a suitable definition of a coordinate being important: once a coordinate i falls below a certain min-entropy threshold, say $(1 - \delta) \log m$ for a very small constant δ, we can consider

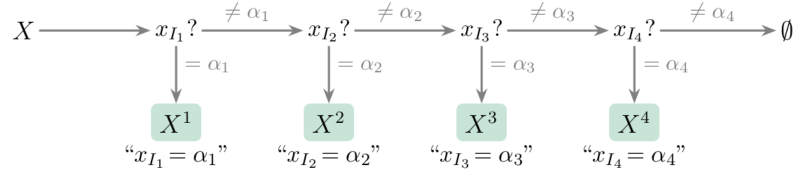

Figure 2.1: Rectangle Partition procedure (figure from [GPW20]).

the coordinate important enough to query in the decision tree. We can think of Π as having "paid" for the coordinate i; since min-entropy can only drop by 1 each round, it took $\delta \log m$ rounds to reduce the entropy of X_i to below the threshold. Since we ultimately want to shave an $\Omega(\log m)$ factor off the height of the communication protocol in our decision tree, once Π has spent $\Omega(\log m)$ steps transmitting information about coordinate i we can feel satisfied giving up the rest of the information about X_i and Y_i for free.

In fact we will use a generalization of min-entropy so that instead of tracking individual coordinates we stop whenever a *set* of coordinates I has a joint assignment $x[I] = \alpha$ which violates $(1 - \delta) \log m$ blockwise min-entropy. For the rest of the proof δ will be some small constant whose value will be fixed later; the reader is free to think of it as $1/100$ for clarity if they are so inclined.

Definition 7. Let S be a set. For a random variable $\boldsymbol{s} \in S^N$, we define its *blockwise min-entropy* by $\mathbf{H}_\infty^\square(\boldsymbol{s}) := \min_{\emptyset \neq I \subseteq [N]} \frac{1}{|I|} \mathbf{H}_\infty(\boldsymbol{s}_I)$, or in other words the least (normalized) marginal min-entropy over all subsets I of the coordinates $[N]$.

Let us use blockwise min-entropy to define our main invariant; as stated, we want to show that the important coordinates, i.e. the ones we have already queried in T, are fixed with the right assignment, while the unimportant coordinates, i.e. the ones we have not already queried in T, are "very unfixed". We denote by $\text{free}(\rho) \subseteq [n]$ the variables assigned a star, and define $\text{fix}(\rho) := [n] \setminus \text{free}(\rho)$.

Definition 8. Let $\rho \in \{0, 1, *\}^n$ be a partial assignment with $J := \text{fix}(\rho) \subseteq [n]$. A rectangle $R = X \times Y \subseteq [m]^n \times (\{0,1\}^m)^n$ is *ρ-structured* if the following conditions hold:

- the gadget is fixed according to ρ: $\text{IND}_m^J(X_J, Y_J) = \{\rho[J]\}$
- X is fixed on fixed blocks and is free on free blocks: X_J is fixed to a single value α, and $\mathbf{H}_\infty^\square(\boldsymbol{X}_{\bar{J}}) \geq (1 - \delta) \log m$
- Y is large: $|Y| \geq 2^{mn - d \log m - |J| \log m}$

If the second condition only holds for $(1 - \delta) \log m - O(1)$, we say R is *ρ-almost structured.*

2.2.2 Density-restoring partition procedure

In the procedure described above, our goal is to maintain a ρ-structured rectangle R, where ρ is the restriction fixed by our path in the decision tree. In each step our blockwise min-entropy drops by 1, and so at some step we will no longer be ρ-structured, but rather ρ-almost structured. Thus a key piece of our algorithm will be to use an entropy-restoring procedure called the *rectangle partition*, which will break our ρ-almost-structured rectangle R into a group of smaller rectangles, from among which we can find a ρ'-structured rectangle R' for some ρ' extending ρ.

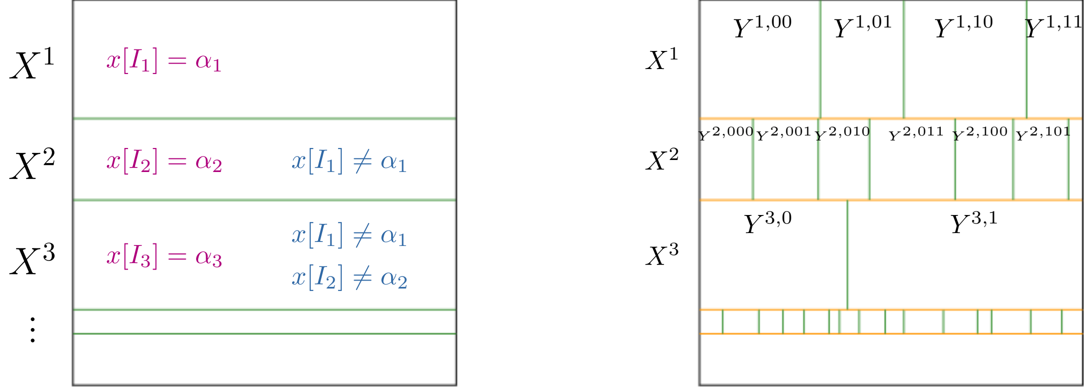

Figure 2.2: Phases I and II of Rectangle Partition. In each $X^j \times Y^{j,\beta}$, $x[I_j]$ is fixed to α_j and $y[I_j]$ is fixed so that $\text{IND}_m^{I_j}(X^j_{I_j}, Y^{j,\beta}_{I_j}) = \beta$.

To perform the partition we will need to find sets $X^j \times Y^{j,\beta}$ along with a corresponding assignment $\rho^{j,\beta}$ for which they are $\rho^{j,\beta}$-structured. This is done in two phases. Our goal in Phase I will be to break up X into disjoint parts X^j, such that each X^j is fixed on some set $I_j \subseteq \bar{J}$ (as well as J, of course) and has blockwise min-entropy $(1-\delta)\log m$ on $\bar{J} \smallsetminus I_j$—hence this partition is “density-restoring” when $\boldsymbol{X}_{\bar{J}}$ starts off with blockwise min-entropy below $(1-\delta)\log m$. To do this, the procedure iteratively finds a maximal partial assignment (I_j, α_j) such that the assignment $x[I_j] = \alpha_j$ violates $(1-\delta)\log m$ blockwise min-entropy in $\boldsymbol{X}_{\bar{J}}$, splits the remaining X into the part X^j satisfying this assignment and the part $X \smallsetminus X^j$ not satisfying it, and recurses on the latter part. We do this until we've covered at least half of X by X^j subsets.

Our goal in Phase II will be to break up Y into disjoint parts $Y^{j,\beta}$ for each X^j from Phase I, such that each $X^j \times Y^{j,\beta}$ is $\rho^{j,\beta}$-structured for some restriction $\rho^{j,\beta}$. We already have the blockwise min-entropy of X^j in the coordinates $\bar{J} \smallsetminus I_j$ by our first goal, and clearly we will choose $\rho^{j,\beta}$ such that $\text{fix}(\rho^{j,\beta}) = J \cup I_j$ for each j. Thus we need to fix the coordinates of Y within the blocks I_j, and within each $Y^{j,\beta}$ it should be the case that $y[I_j, \alpha_j] = \beta$ for all $y \in Y^{j,\beta}$, at which point $\rho^{j,\beta}$ can be fixed to β on I_j and left free everywhere else in $\bar{J}$ (with the coordinates of J being fixed by assumption).

Algorithm Rectangle Partition

1: Initialize $\mathcal{F} = \emptyset$, $j = 1$, and $X^{\geq 1} := X$
2: **PHASE I** (X^j)**:**
3: **while** $|X^{\geq j}| \geq |X|/2$ **do**
4: Let I_j be a maximal subset of $\bar{J}$ such that $\boldsymbol{X}^{\geq j}$ violates $(1-\delta)\log m$-blockwise min-entropy on I_j, or let $I_j = \emptyset$ if no such subset exists
5: Let $\alpha_j \in [m]^{I_j}$ be an outcome such that $\mathbf{Pr}_{x \sim \boldsymbol{X}^{\geq j}}(x[I_j] = \alpha_j) > 2^{-(1-\delta)|I_j|\log m}$
6: Define $X^j := \{x \in X^{\geq j} : x[I_j] = \alpha_j\}$
7: Update $\mathcal{F} \leftarrow \mathcal{F} \cup \{(I_j, \alpha_j)\}$, $X^{\geq j+1} := X^{\geq j} \smallsetminus X^j$, and $j \leftarrow j+1$ [5]
8: **PHASE II** $(Y^{j,\beta})$**:**
9: **for** $(I_j, \alpha_j) \in \mathcal{F}$, $\beta \in \{0,1\}^{I_j}$ **do**
10: Define $Y^{j,\beta} := \{y \in Y : y[I_j, \alpha_j] = \beta\}$
11: Return $\mathcal{F}$, $\{X^j\}_j$, $\{Y^{j,\beta}\}_{j,\beta}$

Our algorithm is formally described in Rectangle Partition. Let $X \times Y$ be ρ-almost structured for

some ρ with $\text{fix}(\rho) = J$ where $|J| = O(d)$, and let $\mathcal{F}$, $\{X^j\}_j$, $\{Y^{j,\beta}\}_{j,\beta}$ be the result of Rectangle Partition on $X \times Y$. Recall that our goal was to break $X \times Y$ up into $\rho^{j,\beta}$-structured rectangles $X^j \times Y^{j,\beta}$; the following simple claims show that the obvious choice of $\rho^{j,\beta}$ achieves two of the three conditions needed (outside of the part of X that we never touch before the procedure ends).

Lemma 5. *For all j and for all $\beta \in \{0,1\}^{I_j}$, define $\rho^{j,\beta} \in \{0,1,*\}^n$ to be the restriction extending ρ by $\rho^{j,\beta}[I_j] = \beta$. Then $X^j_{I_j} = \{\alpha_j\}$ and $\text{IND}_m^{J \cup I_j}(X^j, Y^{j,\beta}) = \rho^{j,\beta}[J \cup I_j]$.*

Proof. By definition X^j is fixed to α_j on the coordinates I_j, while $Y^{j,\beta}$ only contains values y such that $y[\alpha_j] = \beta$. □

Lemma 6. *For all j, $\mathbf{H}_\infty^\square(\boldsymbol{X}^j_{\overline{J \cup I_j}}) \geq (1-\delta)\log m$.*

Proof. Assume for contradiction that $I^* \subseteq \bar{J} \smallsetminus I_j$ such that $\boldsymbol{X}^j$ violates $(1-\delta)\log m$-blockwise min-entropy on I^*, and let α^* be an outcome witnessing this. Then

$$\begin{aligned}\mathbf{Pr}_{x \sim \boldsymbol{X}^{\geq j}}(x[I_j] = \alpha_j \wedge x[I^*] = \alpha^*) &> 2^{-(1-\delta)|I_j|\log m} \cdot \mathbf{Pr}_{x \sim \boldsymbol{X}^j}(x[I^*] = \alpha^*) \\ &> 2^{-(1-\delta)|I_j|\log m - (1-\delta)|I^*|\log m} = 2^{-(1-\delta)|I_j \cup I^*|\log m}\end{aligned}$$

which contradicts the maximality of I_j. □

So far in Lemmas 5 and 6 we have not used the fact that $X \times Y$ was ρ-almost structured. For our third condition, instead of showing that $|Y^{j,\beta}|$ is large for *every* j and every β, we want to show that $|Y^{j,\beta}|$ is large for *some* j and every β. If every β were equally likely then $|Y^{j,\beta}| \approx |Y|/2^{|I_j|}$; for us it is enough that the smallest $Y^{j,\beta}$ be has size at least $|Y|/2^{|I_j|\log m}$. For convenience we redefine X to only be the union of the X^j parts—since we terminate after $|X^{\geq j}| < |X|/2$ we can do this and only decrease the blockwise min-entropy of $\boldsymbol{X}$ by 1—and furthermore we restrict down to the free coordinates $\bar{J}$.

Lemma 7. *Let $X \times Y$ be ρ-almost structured for some ρ with $\text{fix}(\rho) = J \subseteq [n]$ and let $\mathcal{F}$, $\{X^j\}_j$, $\{Y^{j,\beta}\}_{j,\beta}$ be the result of Rectangle Partition on $X \times Y$. Let $X' := (\cup_j X^j)_{\bar{J}}$ be such that $\mathbf{H}_\infty^\square(\boldsymbol{X}') \geq (1-\delta)\log m - O(1)$, and let Y be such that $|Y| \geq 2^{mn - d\log m - |J|\log m}$. Then there is a j such that for all $\beta \in \{0,1\}^{I_j}$,*

$$|Y^{j,\beta}| \geq 2^{mn - d\log m - |J \cup I_j|\log m}$$

Proof. We will show that there is a j such that for all $\beta \in \{0,1\}^{I_j}$, $|Y^{j,\beta}| \geq |Y|/2^{|I_j|\log m}$, which is sufficient by our bound on $|Y|$. Assume for contradiction that for every j there exists a β_j such that $|Y^{j,\beta_j}| < |Y|/2^{|I_j|\log m}$. Define $Y_= := \{y \in Y : \exists j, y[I_j, \alpha_j] = \beta_j\}$ and $Y_{\neq} := Y \smallsetminus Y_= = \{y \in Y : \forall j, y[I_j, \alpha_j] \neq \beta_j\}$.

We first show that $|Y_=| < |Y|/2$. Define $\mathcal{F}(k) := \{(I_j, \alpha_j) \in \mathcal{F} : |I_j| = k\}$. Assume that there exists some k such that $|\mathcal{F}(k)| > 2m^{0.95k}$. Note that every set $(I_j, \alpha_j) \in \mathcal{F}(k)$ corresponds to an assignment to X which occurs with probability greater than $2^{-(1-\delta)k\log m}$ in $X^{\geq j}$, which has size at least $|X|/2$ by construction, and so by a union bound we get that

$$|X'| > |\mathcal{F}(k)| \cdot (2^{-(1-\delta)k\log m}\frac{|X|}{2}) > (\frac{1}{2} \cdot 2^{(1-\delta)\cdot k\log m + 1} \cdot 2^{-(1-\delta)\cdot k\log m})|X| = |X|$$

which is clearly a contradiction. Thus we can assume that $|\mathcal{F}(k)| \leq 2m^{0.95k}$ for all k, then because we

assumed $|Y^{j,\beta_j}| < |Y|/2^{|I_j|\log m}$ we get that

$$\begin{aligned}|Y_=| &< \sum_{k=1}^{n}(2m^{(1-\delta)k} \cdot \frac{|Y|}{2^{k\log m}}) \\ &< \sum_{k=1}^{n}(2^{(1-\delta/2)k\log m - 1} \cdot \frac{|Y|}{2^{k\log m}}) \\ &= \frac{|Y|}{2} \cdot \sum_{k=1}^{n}(2^{(\delta/2)\log m})^{-k} \\ &< \frac{|Y|}{2} \cdot \sum_{k=1}^{\infty} 2^{-k} = \frac{|Y|}{2}\end{aligned}$$

Now because $|Y_=| < |Y|/2$, it must be the case that $|Y_{\neq}| \geq |Y|/2 \geq 2^{mn-d\log m-|J|\log m} > 2^{mn-2n\log m}$. The following lemma, whose proof we defer to the end of the section, is our central contribution; it shows that if $\boldsymbol{X}'$ has high blockwise min-entropy outside some set of coordinates J, and furthermore Y is large, then it's possible to find an $x^* \in X'$ such that the full image of the index gadget is available to x^* outside J, or in other words $\text{Ind}_m^{\bar{J}}(x^*, Y) = \{0,1\}^{\bar{J}}$. We also emphasize that this is the only place in the proof of Query-to-Communication Lifting Theorem where we use the size of the gadget, and it will also be the only place where we are restricted in our choice of δ.

Full Range Lemma. *Let $m^{1-\delta} \geq O(n\log n)$ and let $J \subseteq [n]$. Let $X \times Y \subseteq [m]^{\bar{J}} \times (\{0,1\}^m)^n$ be such that $\mathbf{H}_\infty^{\square}(\boldsymbol{X}) \geq (1-\delta)\log m - O(1)$ and $|Y| > 2^{mn-2n\log m}$. Then there exists an $x^* \in X$ such that for every $\beta \in \{0,1\}^{\bar{J}}$, there exists a $y_\beta \in Y$ such that $\text{Ind}_m^{\bar{J}}(x^*, y_\beta) = \beta$.*

By Full Range Lemma on $X' \times Y_{\neq}$, there must exist some $x^* \in X'$ such that for every $\beta \in \{0,1\}^{\bar{J}}$ there exists $y_\beta \in Y_{\neq}$ such that $y_\beta[\bar{J}, x^*] = \beta$. Since $x^* \in X'$, there exists some j such that $x^* \in X^j_{\bar{J}}$, and thus for any $\beta \in \{0,1\}^{\bar{J}}$ such that $\beta[I_j] = \beta_j$, there exists a $y_\beta \in Y_{\neq}$ such that $y_\beta[\bar{J}, x^*[\bar{J}]] = \beta$. But since $x^* \in X^j_{\bar{J}}$, $x^*[I_j] = \alpha_j$, so $y_\beta[I_j, \alpha_j] = \beta_j$ which is a contradiction since $Y_{\neq} = \{y \in Y : \forall j, y[I_j, \alpha_j] \neq \beta_j\}$. □

Lemmas 5, 6, and 7 together imply that for the choice of j given by Lemma 7, every rectangle $R^{j,\beta}$ is $\rho^{j,\beta}$-structured, where $\rho^{j,\beta}$ extends ρ by fixing $x[I_j] = \beta_j$. Before moving into the simulation theorem, we make a last observation, showing that the deficiency of each $\boldsymbol{X}^j$ drops by $\Omega(|I_j|\log m)$. This will be used later to show the efficiency of our simulation.

Lemma 8. *For all $(I_j, \alpha_j) \in \mathcal{F}$, $\mathbf{D}_\infty(\boldsymbol{X}^j_{\overline{J \cup I_j}}) \leq \mathbf{D}_\infty(\boldsymbol{X}_{\bar{J}}) - \delta|I_j|\log m + 1$.*

Proof. By our choice of (I_j, α_j) it must be that $|X^j| = |X^{\geq j}| \cdot \mathbf{Pr}_{x \sim \boldsymbol{X}^{\geq j}}(x[I_j] = \alpha_j) \geq |X^{\geq j}| \cdot 2^{-(1-\delta)\log m}$. Then by the fact that X^j is fixed on $J \cup I_j$ and and X is fixed on J,

$$\begin{aligned}\mathbf{D}_\infty(\boldsymbol{X}^j_{\overline{J \cup I_j}}) &= |\overline{J \cup I_j}|\log m - \log|X^j| \\ &\leq (n - |J \cup I_j|)\log m - \log(|X^{\geq j}| \cdot 2^{-(1-\delta)|I_j|\log m}) \\ &\leq ((n\log m - |J|\log m) - |I_j|\log m) - \log|X^{\geq j}| + (1-\delta)|I_j|\log m - \log|X| + \log|X| \\ &= (|\bar{J}|\log m - \log|X|) - \delta|I_j|\log m + \log(|X|/|X^{\geq j}|) \\ &\leq \mathbf{D}_\infty(\boldsymbol{X}_{\bar{J}}) - \delta|I_j|\log m + 1\end{aligned}$$

where the last step used the fact that $|X^{\geq j}| \geq |X|/2$, since we terminate as soon as $|X^{\geq j}| < |X|/2$ at the start of the j-th iteration. □

2.2.3 Simulation

We now describe our high level procedure using this partitioning subroutine. Fix δ such that $(1-\delta)(1+\epsilon) > 1 + \Omega(1)$, and let $m = n^{1+\epsilon}$; note that $m^{1-\delta} = n^{(1-\delta)(1+\epsilon)} \geq O(n \log n)$ as required. In addition to the rectangles R_v corresponding to each node v of Π, we maintain a ρ-structure subrectangle $R = X \times Y$, which will be our guide for how to proceed down Π. We start at the root, where $R = R_v = \mathcal{X} \times \mathcal{Y}$ and $\rho = *^n$. At each step we go down to the child v with the larger rectangle $R \cap R_v$—which guarantees that the blockwise min-entropy of $\boldsymbol{X} \cap \boldsymbol{X}_v$ (in the free coordinates) goes down by at most 1 from $\boldsymbol{X}$, as required—and update R to be $R \cap R_v$ for whichever child v we picked. We continue going down the protocol and taking the child with the larger intersection with R until we find that a set of coordinates has blockwise min-entropy less than $(1-\delta)\log m$ in R. After running the rectangle partition, we will decide which set of variables $z[I_j]$ to query using Lemma 7, with the resulting rectangle $R^{j,\beta}$ corresponding to query answer β is $\rho^{j,\beta}$-structured as observed. Lastly, the invariant that $|J|$—i.e. the number of fixed coordinates, and by extension the depth of our tree T—is at most $O(d)$ follows from three facts about the deficiency of $\boldsymbol{X}$: 1) $\mathbf{D}_\infty(\boldsymbol{X}) = 0$ at the start of the protocol; 2) $\mathbf{D}_\infty(\boldsymbol{X}_{\bar{J}}) \leq d \log m - \delta |J| \log m$ by Lemma 8; 3) $\mathbf{D}_\infty(\boldsymbol{s}) \geq 0$ for any random variable $\boldsymbol{s}$ by definition of deficiency.

We describe our query simulation of the communication protocol Π in Simulation Protocol. For all $v \in \Pi$ let $R_v = X_v \times Y_v$ be the rectangle induced at node v by the protocol Π. The query and output actions listed in bold are the ones performed by our decision tree.

Algorithm Simulation protocol

1: Initialize $v :=$ root of Π; $R := [m]^n \times (\{0,1\}^m)^n$; $\rho = *^n$
2: **while** v is not a leaf **do**
3: *Precondition:* $R = X \times Y$ *is* ρ*-structured; for convenience define* $J := \text{fix}(\rho)$
4: Let v_ℓ, v_r be the children of v, and update $v \leftarrow v_\ell$ if $|R \cap R_{v_\ell}| \geq |R|/2$ and $v \leftarrow v_r$ otherwise
5: Execute Rectangle Partition on $(X \cap X_v) \times (Y \cap Y_v)$ and let $\mathcal{F} = \{(I_j, \alpha_j)\}_j$, $\{X^j\}_j$, $\{Y^{j,\beta}\}_{j,\beta}$ be the outputs
6: Apply Lemma 7 to $\mathcal{F}$, $\{X^j\}_j$, $\{Y^{j,\beta}\}_{j,\beta}$ to get some index j corresponding to $(I_j, \alpha_j) \in \mathcal{F}$
7: **Query** each variable z_i for every $i \in I_j$, and let $\beta \in \{0,1\}^{I_j}$ be the result
8: Update $X \leftarrow X^j$ and $Y \leftarrow Y^{j,\beta}$
9: Update $\rho \leftarrow \rho^{j,\beta}$ (recall that $\rho^{j,\beta} \in \{0,1,*\}^n$ is the restriction extending ρ by $\rho^{j,\beta}[I_j] = \beta$)
10: **Output** the same value as v does

Before we prove the correctness and efficiency of our algorithm, we note that we make no distinction between Alice speaking and Bob speaking in our procedure. Here we note that each R_v is a rectangle induced by the protocol Π, and so updating v only splits X or Y—corresponding to when Alice and Bob speak respectively—but not both, and so since $R \subseteq R_v$ we get that $|X \cap X_v| \geq |X|/2$ and $|Y \cap Y_v| \geq |Y|/2$.

Efficiency and correctness. To prove the efficiency and correctness of our algorithm, consider the start of the t-th iteration, where we are at a node v and maintaining $R^t = X^t \times Y^t$ and ρ^t.[6] Again for

[6] We understand that this notation is somewhat overloaded with X^j, $Y^{j,\beta}$, and $\rho^{j,\beta}$. Since the proof that the invariants hold is short and we only ever use t (or $t+1$) for the time stamps and j for the indices, hopefully this won't cause any confusion.

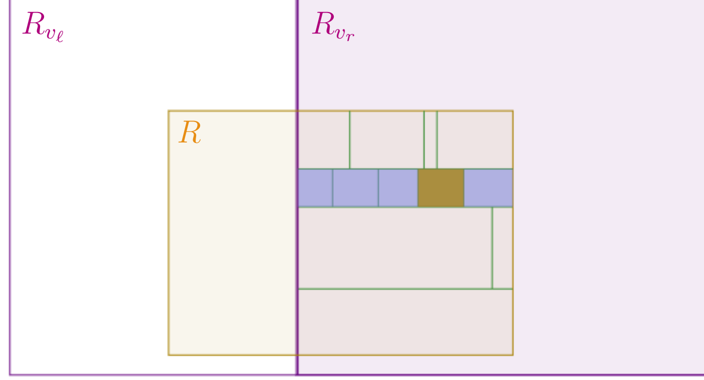

Figure 2.3: One iteration of Simulation Protocol. We perform Rectangle Partition (green lines) on the larger half of R after moving from v to its child (shaded in purple), use Lemma 7 to identify a part j (shaded in blue), and then query I_j and set R to $X^j \times Y^{j,\beta}$ for the result $z[I_j] = \beta$ (shaded in brown).

convenience we write $J^t := \text{fix}(\rho^t)$. Let (I^t, α^t) be the (possibly empty) assignment returned by Lemma 7 corresponding to index j^t, and let β^t be the result of querying $z[I^t]$.

We show that our precondition that R^t is ρ^t-structured holds for all $t \leq d \log m$, as well as the fact that ρ^t fixes at most $O(d)$ coordinates:

(i) $\text{IND}_m^{J^t}(X^t_{J^t}, Y^t_{J^t}) = \rho^t[J^t]$

(ii) X_{J^t} is fixed to a single value and $\boldsymbol{X}^t_{\overline{J^t}}$ has blockwise min-entropy at least $(1-\delta)\log m$

(iii) $|Y^t| \geq 2^{mn-t-|J^t|\log m}$.

(iv) $\mathbf{D}_\infty(\boldsymbol{X}^t_{\overline{J^t}}) \leq 2t - \delta|J^t|\log m$, which implies $|J^t| \leq (2/\delta)d$ by non-negativity of deficiency

All invariants hold at the start of the algorithm since $\rho^0 = *^n$ and $X^0 \times Y^0 = [m]^n \times (\{0,1\}^m)^n$. Inductively consider the $(t+1)$-th iteration assuming all invariant holds for the t-th iteration. After applying Rectangle Partition, invariant *(i)* follows by Lemma 5 and invariant *(ii)* follows by Lemmas 5 and 6. For invariant *(iii)* we first show that it is valid to apply Lemma 7 in the $(t+1)$-th iteration. First, because $|X^t \cap X_v| \geq |X^t|/2$ we know that the blockwise min-entropy of $(\boldsymbol{X}^t \cap \boldsymbol{X}_v)_{\overline{J^t}}$ is at most one less than the blockwise min-entropy of $\boldsymbol{X}^t_{\overline{J^t}}$, which is at least $(1-\delta)\log m$. Second, we have

$$|(Y^t \cap Y_v)| \geq |Y^t|/2 \geq 2^{mn-t-|J^t|\log m - 1} = 2^{mn-(t+1)-|J^t|\log m} \geq 2^{mn-(2/\delta+1)d\log m}$$

recalling that $t+1 \leq d\log m$. Thus we can apply Lemma 7 and we get

$$\begin{aligned} |Y^{t+1}| &= |Y^{j^t,\beta^t}| \\ &\geq 2^{mn-t-|J^t|\log m - 1 - |I^t|\log m} \\ &\geq 2^{mn-t-1-(|J^t|+I^t|)\log m} = 2^{mn-(t+1)-|J^{t+1}|\log m} \end{aligned}$$

For invariant *(iv)*, by Lemma 8 and induction we get that

$$\begin{aligned} \mathbf{D}_\infty(\boldsymbol{X}^{t+1}_{\overline{J^{t+1}}}) &= \mathbf{D}_\infty(\boldsymbol{X}^{j^t}_{\overline{J^{t+1}}}) \\ &\leq \mathbf{D}_\infty((\boldsymbol{X} \cap \boldsymbol{X}_v)_{\overline{J^t}}) - \delta|I^t|\log m + 1 \end{aligned}$$

$$\leq (2t - \delta|J^t|\log m + 1) - \delta|I^t|\log m + 1 = 2(t+1) - \delta|J^{t+1}|\log m$$

which completes the proof of our invariants. Thus our procedure is well-defined.

Leaves. Lastly we have to argue that if we reach a leaf v of Π while maintaining R and ρ with fixed coordinates J, then the solution $o \in \mathcal{O}$ output by Π is also valid solution to the values of z, of which the decision-tree knows that $z[J] = \rho[J]$. Suppose Π outputs $o \in \mathcal{O}$ at the leaf v, and assume for contradiction that there exists $\beta \in \{0,1\}^n$ consistent with ρ such that $\beta \notin f^{-1}(o)$. Since $\text{Ind}_m^J(x,y) = \rho[J] = \beta[J]$ for all $(x,y) \in R$, we focus on $\bar{J} = \text{free}(\rho)$. Since R is ρ-structured, $\mathbf{H}_\infty^\square(\boldsymbol{X}_{\bar{J}}) \geq (1-\delta)\log m$ and $|Y| > 2^{mn - d\log m - |J|\log m} > 2^{mn - 2n\log m}$. Thus we can again apply Full Range Lemma to $X \times Y$, we know that that there exists $(x,y) \in R$ such that $\text{Ind}_m^n(x,y) = \beta$, which is a contradiction as $R \subseteq R_v \subseteq (f \circ \text{Ind}_m^n)^{-1}(o)$.

2.2.4 Proving the Full Range Lemma

Our last task, and the main contribution of this thesis, is to prove Full Range Lemma. The key ingredient in the proof is a conversion from looking at x and y as pointers and strings to looking at them as *set systems*, at which point we can use tools from combinatorics. In particular we will use the following lemma, which enabled all of the recent innovations in the famous *sunflower lemma*, starting with the work of Alweiss et al. [ALWZ20] and subsequently improved by a flurry of work [FKNP19, Rao19, BCW21].

Blockwise Robust Sunflower Lemma. *There exists an absolute constant K such that the following holds: let $s \in \mathbb{N}$ and $\kappa > 0$, and let $\mathcal{F} : \{\gamma\}$ be a set system over any universe $\mathcal{U}$ such that 1) $|\gamma| \leq s$ for all $\gamma \in \mathcal{F}$; and 2) $\mathbf{H}_\infty^\square(\mathcal{F}) \geq \log(K\log(s/\kappa))$. Then*

$$\mathbf{Pr}_{\boldsymbol{y}\subseteq\mathcal{U}}(\forall\gamma \in \mathcal{F} : \gamma \not\subseteq y) \leq \kappa$$

where $\boldsymbol{y}$ is the uniform distribution over subsets of $\mathcal{U}$.

Blockwise Robust Sunflower Lemma has a straightforward but somewhat technical proof, and so we defer interested readers to e.g. [Rao19]. For now we use this result plus some simple math to prove Full Range Lemma, thus concluding the proof of Query-to-Communication Lifting Theorem.

Proof of Full Range Lemma. Assume for contradiction that for all x there exists a $\beta_x \in \{0,1\}^{\bar{J}}$ such that $|\{y \in Y : y[x] = \beta_x\}| = 0$, or in other words for all $(x,y) \in X \times Y$, $y[x] \neq \beta_x$. Our goal will be to show that $|\{y \in Y : \forall x, y[x] \neq \beta_x\}| < 2^{mn - 3n\log m}$, which is a contradiction as $|Y| > 2^{mn-2n\log m}$.

First, we claim that $|Y|$ is maximized when $\beta_x = 1^{\bar{J}}$ for all x. To do this we use the following small claim.

Claim 9. *Let $m, N \in \mathbb{N}$, and let $\mathcal{C} = C_1 \wedge \ldots \wedge C_m$ be a CNF on the variables $x_1 \ldots x_N$ such that no clause contains both the literals x_i and $\overline{x_i}$ for any i. Let $\mathcal{C}^{mon}$ be the result of replacing, for every i, every occurrence of x_i in $\mathcal{C}$ with $\overline{x_i}$.*[7] *Then*

$$|\{x \in \{0,1\}^N : \mathcal{C}(x) = 1\}| \leq |\{x \in \{0,1\}^N : \mathcal{C}^{mon}(x) = 1\}|$$

[7]Intuitively $\mathcal{C}^{mon}$ is the monotone version of $\mathcal{C}$, and note that it does not matter whether our monotone version has all variables occurring positively or negatively. This version will happen to be more suggestive later.

Proof. Let $\mathcal{C}^i$ be the result of replacing every occurrence of x_i in $\mathcal{C}$ with $\overline{x_i}$. It is enough to show that for any i, $\mathcal{C}^i(x)$ is satisfied by at least as many assignments $\beta \in \{0,1\}^N$ to x as $\mathcal{C}(x)$ is, as we can then apply the argument inductively for $i = 1 \dots N$. Let $\beta^{-i} \in \{0,1\}^{[N] \setminus \{i\}}$ be an assignment to every variable except x_i. We claim that for every β^{-i}, $\mathcal{C}^i(\beta^{-i}, x_i)$ is satisfied by at least as many assignments $\beta_i \in \{0,1\}$ to x_i as $\mathcal{C}(\beta^{-i}, x_i)$.

Since there are no clauses with both x_i and $\overline{x_i}$, each clause in $\mathcal{C}$ is of the form $x_i \vee A$, $\overline{x_i} \vee B$, or C, where A, B, and C don't depend on x_i; the corresponding clauses in $\mathcal{C}^i$ are $\overline{x_i} \vee A$, $\overline{x_i} \vee B$, and C. If $\mathcal{C}^i(\beta^{-i}, 1) = 1$, then $A(\beta^{-i}) = B(\beta^{-i}) = C(\beta^{-i}) = 1$ for all A, B, and C, and so $\mathcal{C}^i(\beta^{-i}, x_i)$ is always satisfied. If $\mathcal{C}^i(\beta^{-i}, 0) = 0$, then it must be that $C(\beta^{-i}) = 0$ for some C, and so $\mathcal{C}(\beta^{-i}, x_i)$ has no satisfying assignments. Finally assume neither of these cases hold, and so $\mathcal{C}^i(\beta^{-i}, 1) = 0$ and $\mathcal{C}^i(\beta^{-i}, 0) = 1$. Then it must be that either $A(\beta^{-i}) = 0$ for some A, in which case $\mathcal{C}(\beta^{-i}, 0) = 0$, or $B(\beta^{-i}) = 0$ for some B, in which case $\mathcal{C}(\beta^{-i}, 1) = 0$. Therefore $\mathcal{C}(\beta^{-i}, x_i)$ has at least one falsifying assignment, while $\mathcal{C}^i(\beta^{-i}, x_i)$ has exactly one. □

Consider the CNF over $y_1 \dots y_{mn}$ where clause C_x is the clause uniquely falsified by $y[x] = \beta_x$; then by Claim 9 we see that $|\{y \in (\{0,1\}^m)^n : \forall x, y[x] \neq \beta_x\}|$ is maximized when $\beta_x = 1^{\bar{J}}$. Thus because $Y \subseteq (\{0,1\}^m)^n$,

$$|\{y \in Y : \forall x, y[x] \neq \beta_x\}| \leq |\{y \in (\{0,1\}^m)^n : \forall x, y[x] \neq 1^{\bar{J}}\}|$$

Consider the space $[mn]$ where each element is indexed by $(i, \alpha) \in [n] \times [m]$. For each $x \in X$, let $S_x \subseteq [mn]$ be the set defined by including (i, α) iff $x[i] = \alpha$, and let $\mathcal{S}_X = \{S_x : x \in X\}$. By the fact that $m^{1-\delta} \geq O(n \log m)$ and $|\bar{J}| \leq n$,

$$\mathbf{H}_\infty^{\square}(\mathcal{S}_\mathbf{X}) \geq (1-\delta)\log m - O(1) \geq \log(O(n \log m)) > \log(K \log(|\bar{J}|/\kappa))$$

where $\kappa := 2^{-3n \log m}$ and K is the constant given by Blockwise Robust Sunflower Lemma. Thus we can apply Blockwise Robust Sunflower Lemma to $\mathcal{S}_X$ and get that $\mathbf{Pr}_{\mathbf{S_y} \subseteq [mn]}(\forall S_x \in \mathcal{S}_X, S_x \not\subseteq S_y) \leq \kappa$, and if we look at y as being the indicator vector for S_y then we get that $\mathbf{Pr}_{\boldsymbol{y} \sim \{0,1\}^{mn}}(\forall x \in X, y[x] \neq 1^{\bar{J}}) \leq \kappa$. Thus by counting we get

$$\begin{aligned} |Y| &= |\{y \in Y : \forall x, y[x] \neq \beta_x\}| \\ &\leq |\{y \in (\{0,1\}^m)^n : \forall x, y[x] \neq 1^{\bar{J}}\}| \\ &\leq \kappa \cdot 2^{mn} = 2^{mn - 3n \log m} \end{aligned}$$

which is a contradiction as $|Y| > 2^{mn - 2n \log m}$ by assumption. □

2.2.5 Afterword: near-linear size gadgets

What happens if δ is chosen to be subconstant? We cannot hope to get a tight lifting theorem, as our decision tree will be of depth $(2/\delta) \cdot d \notin O(d)$. Furthermore choosing $\delta = o(1/\log m)$ makes our blockwise min-entropy threshold $(1-\delta)\log m$ trivial, as $\log m$ is the maximum possible blockwise min-entropy for $\boldsymbol{X}$. However these are the only restrictions in the proof, and so as long as we choose $\delta = \Omega(1/\log m)$ the proof goes through:

Theorem 10. *Let f be a search problem over $\{0,1\}^n$, let $\delta \geq \frac{1}{\log n}$, and let m be such that $m^{1-\delta} \geq$*

$3Kn \log n$ for K given by Blockwise Robust Sunflower Lemma. Then for $g = \textsc{Ind}_m$,

$$\textsf{dec-tree-depth}(f) \cdot (\log m + 1) \;\geq\; \textsf{cc-tree-depth}(f \circ g) \;\geq\; \textsf{dec-tree-depth}(f) \cdot \Omega(\delta \log m)$$

Note that for $\delta = \Theta(1/\log m)$ this gives us a gadget of size $O(n \log n)$ while still guaranteeing that our decision tree for f has depth asymptotically as good as any communication protocol for $f \circ \textsc{Ind}_m$.

2.3 Dag-like lifting

In this section we show that we can perform our lifting theorem in the dag-like model, going from decision dags to communication dags. This was originally proven by Garg et al. [GGKS20] using an alternate proof of Full Range Lemma, and we follow their proof exactly; in fact the only difference is that the parameters in Full Range Lemma require them to define ρ-structured with $|Y| \geq 2^{mn-n^3}$, whereas our definition of ρ-structured is the stricter—in fact, even moreso than in Query-to-Communication Lifting Theorem—condition that $|Y| \geq 2^{mn-O(d \log m)}$, which will again allow us to show the same gadget size improvements.

Dag-like Lifting Theorem. *Let f be a search problem over $\{0,1\}^n$, and let $m = n^{1+\epsilon}$. Then*

$$\log \textsf{rect-dag}(f \circ \textsc{Ind}_m^n) \;=\; \textsf{dec-dag-width}(f) \cdot \Theta(\log m)$$

Again one direction is simple; given a conjunction dag D for f of width d we can construct a rectangle dag Π for $f \circ \textsc{Ind}_m^n$ of size $m^{O(d)}$ by simply replacing each edge in D with a short protocol that queries all variables fixed by the edge. Let Π be a communication protocol for $f \circ \textsc{Ind}_m^n$ of size m^d; our goal will be to construct a decision-dag D of width $O(d)$ for f. Again we fix δ such that $(1-\delta)(1+\epsilon) > 1$, and let $m = n^{1+\epsilon}$. In this proof we will also use the fact that $d = o(n)$; note that if $d = \Omega(n)$ then the theorem is trivial, as there always exists a decision-dag for f of width $O(n)$.

2.3.1 Rectangle partition and forgetting bits

We start by slightly changing the definition of ρ-structured, specifically by making the largeness condition on Y fixed with respect to d:

Definition 9 (Definition 8, dag-like version)**.** Let $\rho \in \{0,1,*\}^n$ be a partial assignment with $J := \text{fix}(\rho) \subseteq [n]$. A rectangle $R = X \times Y \subseteq [m]^n \times (\{0,1\}^m)^n$ is *ρ-structured* if the following conditions hold:

- the gadget is fixed according to ρ: $\textsc{Ind}_m^J(X_J, Y_J) = \{\rho[J]\}$
- X is fixed on fixed blocks and is free on free blocks: X_J is fixed to a single value α, and $\mathbf{H}_\infty^\square(\boldsymbol{X}_{\bar{J}}) \geq (1-\delta)\log m$
- Y is large: $|Y| \geq 2^{mn-(2/\delta+2)d \log m}$

If the second condition only holds for $(1-\delta)\log m - O(1)$, we say R is *ρ-almost structured*.

Our procedure is similar to before, maintaining a ρ-structured rectangle $R \subseteq R_v$ at every step, but now there's a slight twist: the protocol may have depth greater than d and can decide to "forget" some

bits at each stage, at which point we will have to make sure the assignment ρ we maintain also stays small.

This presents two problems. First off, it won't be enough to find a subrectangle of our current rectangle R, since R has some bits fixed that may be forgotten by the protocol. We circumvent this by applying the rectangle partition procedure to the *actual* rectangle R_v, which allows us to find the "important bits" as before, and then shift to a good rectangle $X^j \times Y^{j,\beta}$, leaving R behind.

The second challenge is that whenever we apply Rectangle Partition we need to ensure that every set I_j we find is of size $O(d)$. The Rectangle Lemma is the main technical lemma of [GGKS20], establishing extra properties of Rectangle Partition. We give a new proof of the Rectangle Lemma, showing that that by slightly modifying Rectangle Partition we can remove some "error sets" from X and Y and afterwards assume that all our rectangles $X^j \times Y^{j,\beta}$ are ρ-structured for some *small* restriction ρ, aka one that fixes $O(d)$ coordinates. Similar to Lemmas 5 and 6, here we don't require that $\boldsymbol{X}$ has high blockwise min-entropy or Y is large (although we will use those conditions at a different part of the proof); recall that in Rectangle Partition these conditions were only needed to a) find a "good" j and b) to ensure the deficiency of X drops, neither of which we will need.

Rectangle Lemma. *Let $R = X \times Y \subseteq \mathcal{X} \times \mathcal{Y}$ be a rectangle and let $d = o(n)$. Then there exists a procedure which outputs $\{X^j \times Y^{j,\beta}\}_{j,\beta}, X_{err}, Y_{err}$, where X_{err} and Y_{err} have density $2^{-2d \log m}$ in $\mathcal{X}$ and $\mathcal{Y}$ respectively, and for each j, β one of the following holds:*

- ***structured**: $X^j \times Y^{j,\beta}$ is $\rho^{j,\beta}$-structured for some $\rho^{j,\beta}$ of width at most $O(d)$*
- ***error**: $X^j \times Y^{j,\beta} \subseteq X_{err} \times \{0,1\}^{mn} \cup [m]^n \times Y_{err}$*

Finally, a query alignment property holds: for every $x \in [m]^n \setminus X_{err}$ there exists a subset $I_x \subseteq [n]$ with $|I_x| \leq O(d)$ such that every "structured" $X^j \times Y^{j,\beta}$ intersecting $\{x\} \times \{0,1\}^{mn}$ has $\mathrm{fix}(\rho^{j,\beta}) \subseteq I_x$.

We defer the proof of Rectangle Lemma to the end of the section; it will go in much the same way as the facts about Rectangle Partition in Query-to-Communication Lifting Theorem.

2.3.2 Simulation (without errors)

With Rectangle Lemma at hand, the simulation algorithm (Dag-like Simulation Protocol) and proof of correctness essentially follows [GGKS20]. We first present the proof under a simplifying assumption and then remove that assumption in Section 2.3.3.

($*$) *Assumption:* If we apply Rectangle Lemma for $k := 2d \log m$ to any rectangle R_v in Π, then each part in the produced partition $R^{j,\beta}$ satisfies the "structured case".

We will go through a similar iterative process as Simulation Protocol, for the moment assuming ($*$). As before, our goal will be to maintain a ρ-structured rectangle R where $|\mathrm{fix}(\rho)| \leq O(d)$.

Root and leaves. At the root of Π our rectangle will be $R = R_v = [m]^n \times (\{0,1\}^m)^n$, which is ρ-structured for $\rho = *^n$. The leaves case is analogous to Query-to-Communication Lifting Theorem as well; if we reach a leaf node v while maintaining a restriction ρ with $|\mathrm{fix}(\rho)| \leq O(d)$, applying Full Range Lemma to $R \subseteq R_v$ we are guaranteed that every extension to ρ is still possible, meaning that no assignment extending ρ can contradict the monochromatic label at v.

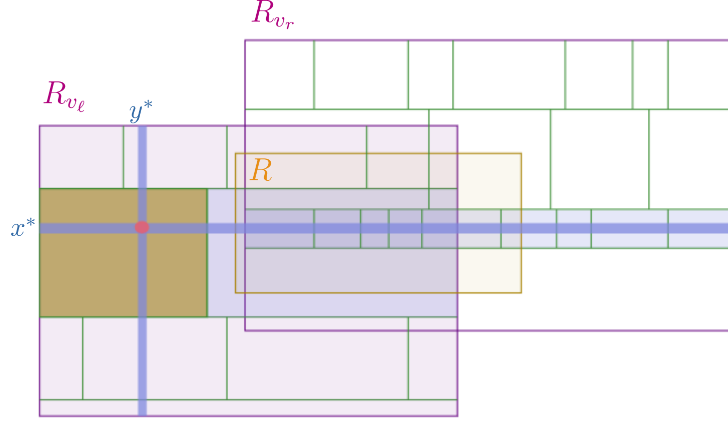

Figure 2.4: One iteration of Dag-like Simulation Protocol. We perform Rectangle Partition (green lines) on both R_{v_ℓ} and R_{v_r} separately, use Full Range Lemma find an $x^* \in R$ with full range, query all bits in the sets I_{j_ℓ} and I_{j_r} corresponding to $X^{j_\ell}, X^{j_r} \ni x^*$ (shaded in blue), find a y^* for which $\text{IND}_m^n(x^*, y^*)$ matches the result, and set R to $X^{j_c} \times Y^{j_c,\beta_c} \ni (x^*, y^*)$ (shaded in brown) for $c \in \{\ell, r\}$ (shaded in purple).

Internal nodes. Fix an internal node v of the protocol with children v_ℓ and v_r, and assume we are maintaining a ρ-structured rectangle $R \subseteq R_v \subseteq R_{v_\ell} \cup R_{v_r}$, where $J = \text{fix}(\rho)$ has size at most $O(d)$. Our challenge, as noted above, is to figure out which variables to forget in order to not cross the $O(d)$ threshold. To do this, we use R_{v_ℓ} and R_{v_r} as a guide not just for which new variables to query but also for which old variables to remember. We do so using Rectangle Lemma.

We apply Rectangle Lemma to R_{v_ℓ} and R_{v_r}, and by our assumption they will both only produce structured rectangles $R_\ell^{j,\beta}$ and $R_r^{j,\beta}$. Because R is ρ-structured, we can apply Full Range Lemma to R and get a row x^* which has the full range of the index gadget on blocks $\bar{J}$ available to it. Now because $R \subseteq R_\ell \cap R_r$ we know that $X \subseteq X_\ell \cup X_r$, and in particular that there exists some $c \in \{\ell, r\}$ such that $x^* \in X_c$. Furthermore, every row in R_ℓ and R_r belongs to a structured rectangle, and so there exists some j, associated with a set I_{x^*} by the query alignment property, such that $x^* \in X_c^j$. We query all variables in $I_{x^*} \smallsetminus J$, forget all variables in $J \smallsetminus I_{x^*}$, and set $R = X_c^j \times Y_c^{j,\beta}$, where β is the result of our query. This gives us a $\rho^{j,\beta}$-structured rectangle R where $\text{fix}(\rho) = I_x$, which has size $O(d)$ by construction.

2.3.3 Dealing with errors

We turn our attention now to the error sets; note that if x^* falls in the error sets of R_ℓ and R_r then we will run into trouble. Our solution will be to remove all the error sets from $\mathcal{X}$ and $\mathcal{Y}$ before the start of the protocol, which will ensure that all rectangles that R_v returns are structured.

We do this in a bottom-up fashion: for each v we remove from R_v all error sets appearing in *any descendant* of v, and then after applying Rectangle Lemma to the remaining rectangle we remove all error sets returned as well. By removing all error sets from the descendants of v, we ensure that any structured rectangle $R = R_v^{j,\beta}$ we transition into will only overlap with the structured parts of its children R_ℓ and R_r.

We also need to ensure that this preprocessing step does not throw away too much of our rectangles. Since the number of descendants of any node v is at most $|\Pi| = m^d$, we know that after having removed

all error sets below the current node v we've only lost a $m^d \times 2^{-2d \log m} \ll 1/2$ fraction of X_v and Y_v. At the root of Π, after processing R_v in total we've lost an $m^d \cdot 2^{-2d \log m} \ll 1/2$ fraction of $[m]^n$ and $(\{0,1\}^m)^n$ each, meaning we start with $|X_v| = m^n/2$ and $|Y_v| = 2^{mn}/2$. After this the rectangle associated with the root we will never encounter an error rectangle in our procedure. We note that this will be the only place where we use the fact that $|\Pi| = m^d$.

Algorithm Dag-like Simulation Protocol

1: **PREPROCESSING:** initialize $X^*_{err} = \emptyset$ and $Y^*_{err} = \emptyset$, and for all $v \in \Pi$ let $R_v := X_v \times Y_v$ be the rectangle corresponding to v
2: **for** $v \in \Pi$ starting from the leaves and going up to the root **do**
3: Update $X_v \leftarrow X_v \setminus X^*_{err}$ and $Y_v \leftarrow Y_v \setminus Y^*_{err}$
4: Apply Rectangle Lemma to $X_v \times Y_v$ and let $\{X^j_v\}_j$, $\{Y^{j,\beta}_v\}_{j,\beta}$, X_{err}, Y_{err}, $\{I_x\}_x$ be the outputs
5: Update $X^*_{err} \leftarrow X^*_{err} \cup X_{err}$ and $Y^*_{err} \leftarrow Y^*_{err} \cup Y_{err}$
6: Initialize $v :=$ root of Π; $R := R_v$; $\rho = *^n$
7: **while** v is not a leaf **do**
8: *Precondition:* $R = X \times Y$ *is* ρ*-structured, for convenience define* $J := \text{fix}(\rho)$*, and furthermore* $|J| \leq O(d)$
9: Apply Full Range Lemma to $X_{\bar{J}} \times Y$ to get $x^* \in X$
10: Let v_ℓ, v_r be the children of v, let j_ℓ, j_r be the indices such that $x^* \in X^{j_\ell}_{v_\ell}$ and $x^* \in X^{j_r}_{v_r}$, and let I_{j_ℓ} and I_{j_r} be the query alignment sets I_{x^*} for v_ℓ and v_r respectively
11: **Query** each variable z_i for every $i \in (I_{j_\ell} \cup I_{j_r}) \setminus J$, let $\beta_\ell \in \{0,1\}^{I_{j_\ell}}$ be the result concatenated with $\rho[J]$ and restricted to I_{j_ℓ}, and let $\beta_r \in \{0,1\}^{I_{j_r}}$ be defined analogously
12: Let $y^* \in Y$ be such that $\text{IND}^{I_{j_\ell}}_m(x^*, y^*) = \beta_\ell$ and $\text{IND}^{I_{j_r}}_m(x^*, y^*) = \beta_r$, and let $c \in \{\ell, r\}$ be such that $(x^*, y^*) \in X^{j_c}_{v_c} \times Y^{j_c,\beta_c}_{v_c}$
13: Update $X \times Y = X^{j_c}_{v_c} \times Y^{j_c,\beta_c}_{v_c}$ and $\rho \leftarrow \rho^{j_c,\beta_c}$
Output the same value as v does

We state the full algorithm formally in Dag-like Simulation Protocol. We briefly go over the invariants needed to run our algorithm. First, at the root node v we set $R = R_v$, and since $|X| \geq |\mathcal{X}|/2$ and $|Y| \geq |\mathcal{Y}|/2$, R is clearly ρ-structured for $\rho = *^n$ as before. In the main procedure, assuming the precondition of R being ρ-structured holds we meet all conditions for applying Full Range Lemma. Since x^* has full range we know that every $Y^{j_\ell,\beta_\ell}_{v_\ell}$ and $Y^{j_r,\beta_r}_{v_r}$ exists, and since we removed all error sets the rectangle $X^{j_c}_{v_c} \times Y^{j_c,\beta_c}_{v_c}$ we end up in must be in the "structured" case of Rectangle Lemma. Thus again end up in an R which is ρ^{j,β_c}-structured for some ρ^{j_c,β_c} which fixes at most $O(d)$ coordinates, and so we've met the preconditions for the next round.

Our argument at the leaves is identical to the proof of Query-to-Communication Lifting Theorem, but we restate it formally for completeness. Suppose Π outputs $o \in \mathcal{O}$ at the leaf v, and assume for contradiction that there exists $\beta \in \{0,1\}^n$ consistent with ρ such that $\beta \notin f^{-1}(o)$. Since $\text{IND}^J_m(x,y) = \rho[J] = \beta[J]$ for all $(x,y) \in R$, we focus on $\bar{J} = \text{free}(\rho)$. Since R is ρ-structured, $\boldsymbol{X}_{\bar{J}}$ has blockwise min-entropy $0.95 \log m$ and $|Y| > 2^{mn - d \log m - |J| \log m} > 2^{mn - 2n \log m}$. Thus applying Full Range Lemma to $X \times Y$, we know that that there exists $(x,y) \in R$ such that $\text{IND}^n_m(x,y) = \beta$, which is a contradiction as $R \subseteq R_v \subseteq (f \circ \text{IND}^n_m)^{-1}(o)$.

2.3.4 Proof of Rectangle Lemma

Our procedure for generating rectangles $X^j \times Y^{j,\beta}$ will be very similar to Rectangle Partition, but with a number of additions. First (and least important), we run Phase I until $X^{\geq j}$ is empty instead of stopping

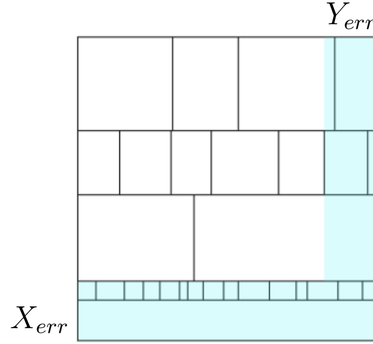

Figure 2.5: Error rectangles shaded in blue. X^j is added to X_{err} if I_j is too large (bottom), while $Y^{j,\beta}$ is added to Y_{err} if $Y^{j,\beta}$ is too small (right).

after partitioning half of X.[8] We then run Phase II exactly as before. Finally we will create the error sets X_{err} and Y_{err} as follows:

- X_{err} will contain all the X^j sets where the corresponding assignment (I_j, α_j) is too large, namely when $|I_j| > (2/\delta)d$
- Y_{err} will contain all the $Y^{j,\beta}$ sets which are too small, namely when $|Y^{j,\beta}| < 2^{mn-(2/\delta+2)d\log m}$.

Algorithm Small-set Rectangle Partition with Errors

1: Initialize $\mathcal{F} = \emptyset$, $j = 1$, and $X^{\geq 1} := X$
2: Initialize $X_{err}, Y_{err} = \emptyset$ and $J_x, J_y = \emptyset$
3: **PHASE I** (X^j)**:**
4: **while** $X^{\geq j} \neq \emptyset$ **do**
5: Let I_j be a maximal subset of $[n]$ such that $\boldsymbol{X}^{\geq j}$ violates $0.95\log m$-blockwise min-entropy on I_j, or let $I_j = \emptyset$ if no such subset exists
6: Let $\alpha_j \in [m]^{I_j}$ be an outcome such that $\mathbf{Pr}_{x\sim \boldsymbol{X}^{\geq j}}(x[I_j] = \alpha_j) > 2^{-0.95|I_j|\log m}$
7: Define $X^j := \{x \in X^{\geq j} : x[I_j] = \alpha_j\}$
8: Update $\mathcal{F} \leftarrow \mathcal{F} \cup \{(I_j, \alpha_j)\}$ and $X^{\geq j+1} := X^{\geq j} \setminus X^j$
9: Update $j \leftarrow j + 1$
10: **PHASE II** ($Y^{j,\beta}$)**:**
11: **for** $j, \beta \in \{0,1\}^{I_j}$ **do**
12: Define $Y^{j,\beta} := \{y \in Y : y[I_j, \alpha_j] = \beta\}$
13: **PHASE X-ERR** (X_{err})**:**
14: **while** $\exists j \notin J_x$ such that $|I_j| > 40d$ **do**
15: Update $X_{err} \leftarrow X_{err} \cup X^j$ and $J_x \leftarrow J_x \cup \{j\}$
16: **PHASE Y-ERR** (Y_{err})**:**
17: **while** $\exists (j, \beta) \notin J_y : j \notin J_x, \beta \in \{0,1\}^{I_j}$ such that $|Y^{j,\beta}| < 2^{mn-42d\log m}$ **do**
18: Update $Y_{err} \leftarrow Y_{err} \cup Y^{j,\beta}$ and $J_y \leftarrow J_y \cup \{(j, \beta)\}$
19: return $\mathcal{F}$, $\{X^j\}_{j\notin J_x}$, $\{Y^{j,\beta}\}_{(j,\beta)\notin J_y}$, X_{err}, Y_{err}

Our algorithm is presented in full in Rectangle Partition with Errors. We prove a series of short claims, most of which immediately follow in the same way as Lemmas 5, 6, and 7. The first puts these

[8] Our procedure doesn't require a drop in deficiency anymore, since it's enough to maintain the invariant that we've fixed at most $O(d)$ coordinates. However it is important to not leave out any of X, since you want to ensure that the x^* we get from Full Range Lemma falls in one of the X^js.

claims together to show that all rectangles corresponding to $j \notin J_x$, $(j,\beta) \notin J_y$ fulfill the "structured" case of Rectangle Lemma. In the second we handle the density of the error rectangles.

Lemma 11. *For all $j \notin J_x$ and all $\beta \in \{0,1\}^{I_j}$ such that $(j,\beta) \notin J_y$, $X^j \times Y^{j,\beta}$ is $\rho^{j,\beta}$-structured for some $\rho^{j,\beta}$ which fixes at most $O(d)$ coordinates.*

Proof. As usual, for all j and for all $\beta \in \{0,1\}^{I_j}$, define $\rho^{j,\beta} \in \{0,1,*\}^n$ to be the restriction where $\text{fix}(\rho^{j,\beta}) = I_j$ and $\rho^{j,\beta}[I_j] = \beta$. Then

- by Lemma 5, $X^j_{I_j}$ is fixed to α_j and $\text{IND}^{I_j}_m(X^j_{I_j}, Y^{j,\beta}_{I_j}) = \rho^{j,\beta}[I_j]$.
- by Lemma 6, $\mathbf{H}^{\square}_\infty(\boldsymbol{X}_{\overline{J \cup I_j}}) \geq (1-\delta)\log m$.
- since $(j,\beta) \notin J_y$, it must be that $|Y^{j,\beta}| \geq 2^{mn-(2/\delta+2)d\log m}$

and so $X^j \times Y^{j,\beta}$ is $\rho^{j,\beta}$-structured. Furthermore, since $j \notin J_x$ it must be the case that $|\text{fix}(\rho^{j,\beta})| = |I_j| \leq (2/\delta)d = O(d)$. ☐

Lemma 12. $|X_{err}| \leq m^n \cdot 2^{-2d\log m}$ *and* $|Y_{err}| \leq 2^{mn} \cdot 2^{-2d\log m}$

Proof. For X_{err} we have two cases: either X_{err} is empty, in which case the claim is trivial, or X_{err} is not empty and there is some minimal $j \in J_x$ such that X^j gets added to X_{err}, and by extension $|I_j| > (2/\delta)d$. By the fact that (I_j, α_j) violates $(1-\delta)\log m$ blockwise min-entropy in $X^{\geq j}$ we know that $|X^j| \geq |X^{\geq j}| \cdot 2^{-(1-\delta)|I_j|\log m}$, and because X^j is a set in $[m]^n$ fixed on coordinates $I_j \subseteq n$ we also know that $|X^j| \leq 2^{(n-|I_j|)\log m}$, which together gives us

$$|X_{err}| \leq |X^{\geq j}| < 2^{(n-|I_j|)\log m + (1-\delta)|I_j|\log m} < 2^{(n-\delta\cdot(2/\delta)d)\log m} < m^n \cdot 2^{-2d\log m}$$

For Y_{err}, as in the proof of Lemma 7 for all $k \in [(2/\delta)d]$ we get that $|\mathcal{F}(k)| \leq 2m^{(1-\delta)k}$,[9] and so for each there are at most $2^k \cdot 2m^{(1-\delta)k} < 2^{k\log m}$ tuples (I_j, α_j, β_j) such that $|\{y \in Y \setminus Y_{err} : y[I_j, \alpha_j] = \beta\}| < 2^{mn-(2/\delta+2)d\log m}$. Taking a union bound we get that

$$|Y_{err}| \leq \sum_{k=1}^{(2/\delta)d} 2^{k\log m} \cdot 2^{mn-(2/\delta+2)d\log m} \leq 2 \cdot 2^{mn-(2/\delta+2)d\log m + (2/\delta)d\log m} \ll 2^{mn} \cdot 2^{-2d\log m}$$

which completes the proof. ☐

The proof of Rectangle Lemma is now fairly immediate. The density of X_{err} and Y_{err} follows from Lemma 12. For any $X^j \times Y^{j,\beta}$, if $j \notin J_x$ and $(j,\beta) \notin J_y$, then by Lemma 11 this fulfills the structured case, while if $j \in J_x$ then $X^j \subseteq X_{err}$, while if $j \in J_y$ then $Y^{j,\beta} \subseteq Y_{err}$ by definition. The query alignment property holds by taking $I_x = I_j$ for all $x \notin X_{err}$, where $j \notin J_x$ is such that $x \in X^j$.

2.4 Graduated lifting

In this section we prove a variant on Query-to-Communication Lifting Theorem, which allows us to set the gadget size m in terms of the target decision tree depth d. The tree-like theorem was originally proven in [GKMP20] but it also follows immediately from our proof of Query-to-Communication Lifting

[9] Note that in the statement of the lemma we assume nothing about the blockwise min-entropy of X; however our union bound still holds because every rectangle X^j corresponds to an assignment which has probability *at least* $\exp(-(1-\delta)k\log m)$.

Theorem with significant improvements to the gadget size. The only technical detail is that for arbitrary search problems we cannot allow the gadget size to go below $\Omega(\log^{1+\epsilon} n)$,[10] although future improvements on the statement of Blockwise Robust Sunflower Lemma could change this restriction. In particular, the *Robust Sunflower Conjecture* states that the precondition in Blockwise Robust Sunflower Lemma can be improved to $\log O(\log 1/\epsilon)$; if this held then it would remove our restriction on d.

Theorem 13 (Graduated Lifting Theorem, large d). *1. Let f be a search problem over $\{0,1\}^n$ and let $m \geq \max(\textsf{dec-tree-depth}(f), \log n)^{1+\epsilon}$ for some $\epsilon > 0$. Then*

$$\textsf{cc-tree-depth}(f \circ \textrm{I}{\small\textrm{ND}}_m) \;\geq\; \textsf{dec-tree-depth}(f) \cdot \Omega(\log m)$$

2. Let f be a search problem over $\{0,1\}^n$ and let $m \geq \max(\textsf{dec-dag-width}(f), \log n)^{1+\epsilon}$ for some $\epsilon > 0$. Then

$$\log \textsf{rect-dag}(f \circ \textrm{I}{\small\textrm{ND}}_m^n) \;=\; \textsf{dec-dag-width}(f) \cdot \Theta(\log m)$$

Proof sketch. We focus on tree-like lifting, as the proof is analogous for dag lifting. We start with a given communication protocol Π of depth $d \cdot \delta \log m$ for the composed problem $f \circ \textrm{I}{\small\textrm{ND}}_m^n$, where δ is such that $(1-\delta)(1+\epsilon) > 1$, and we construct a decision-tree of depth $O(d)$ for f. We change the precondition on $|Y|$ in Full Range Lemma to read "$|Y| \geq 2^{mn-2d\log m}$", which is guaranteed by the preconditions of Lemma 7 and our ρ-almost structured invariant whenever it is applied. Accordingly, in the proof of Full Range Lemma we set $\kappa = 2^{-3d\log m}$. By our choice of m we get that $(1-\delta)\log m - O(1) \geq \log(K \log n + K \cdot 3d \log m) \geq \log K \log(n/\kappa)$. ☐

We also note that for many natural search problems, the restriction $m = \Omega(\log n)$ can be removed. For a search problem $\mathcal{S}$, a *certificate* for $(x, o) \in \mathcal{S}$ is a partial assignment $\rho \in \{0,1,*\}^n$ consistent with x such that for any y consistent with ρ we have $(y, o) \in \mathcal{S}$. The *certificate complexity* of $\mathcal{S}$ is the maximum over all inputs $x \in \{0,1\}^n$ of the minimum over all $o \in \mathcal{S}(x)$ of the least size of a certificate for (x, o). For example, if τ is an unsatisfiable k-CNF formula, then the *canonical search problem* $\mathcal{S}_\tau$, which we discuss extensively in the next chapter, has certificate complexity at most k, because every o is a clause of width at most k, and thus the certificate is just the assignment to each literal in o which falsifies it.

Theorem 14 (Graduated Lifting Theorem, small d). *1. Let f be a search problem over $\{0,1\}^n$ with certificate complexity $2^{O(d\log d)}$, and let $m \geq (\textsf{dec-tree-depth}(f))^{1+\epsilon}$ for some $\epsilon > 0$. Then*

$$\textsf{cc-tree-depth}(f \circ \textrm{I}{\small\textrm{ND}}_m) \;\geq\; \textsf{dec-tree-depth}(f) \cdot \Omega(\log m)$$

2. Let f be a search problem over $\{0,1\}^n$ with certificate complexity $2^{O(d\log d)}$, and let $m \geq (\textsf{dec-dag-width}(f))^{1+\epsilon}$ for some $\epsilon > 0$. Then

$$\log \textsf{rect-dag}(f \circ \textrm{I}{\small\textrm{ND}}_m^n) \;=\; \textsf{dec-dag-width}(f) \cdot \Theta(\log m)$$

Proof sketch. We begin by apply all the changes to Full Range Lemma as stated in the previous proof. In order to remove the difficulty of having the set size n in the statement of Blockwise Robust Sunflower Lemma, we marginalize each x to subsets of size at most $2^{O(d\log m)}$.

[10] This necessarily holds whenever $d = \Omega(\log n)$, which can be considered the "natural" range of parameters as otherwise there is a variable that is never queried along *any* path of our decision tree, meaning f does not depend on all its variables.

Lemma 15 (d-Full Range Lemma)**.** *Let $m \geq d^{1+\epsilon}$ and let $J \subseteq [n]$. Let $X \times Y \subseteq [m]^{\bar{J}} \times (\{0,1\}^m)^n$ be such that $\boldsymbol{X}$ has blockwise min-entropy at least $(1-\delta)\log m - O(1)$ and $|Y| > 2^{mn - 2d\log m}$. Then there exists an $x^* \in X$ such that for every constant C, every $J' \subseteq \bar{J}$ with $|J'| \leq 2^{Cd\log m}$, and every $\beta \in \{0,1\}^{J'}$, there exists a $y_\beta \in Y$ such that $\textsc{Ind}_m^{J'}(x^*, y_\beta) = \beta$.*

Proof. Assume for contradiction that for all x there exists a set $I_x \subseteq \bar{J}$ of size at most $2^{Cd\log m}$ and an assignment $\beta_x \in \{0,1\}^{I_x}$ such that $|\{y \in Y : y[I_x, x[I_x]] = \beta_x\}| = 0$. As before, by Claim 9 we can assume that $\beta_x = 1^{I_x}$. For each $x \in X$, let $S_x \subseteq [mn]$ be the set defined by including (i, α) iff $x[i] = \alpha$ and $i \in I_x$, and let $\mathcal{S}_X = \{S_x : x \in X\}$.

As above we set $\kappa = 2^{-3d\log m}$. By our choice of m we get that $(1-\delta)\log m - O(1) \geq \log(K(C+3)d\log m) \geq \log K \log(2^{Cd\log m}/\kappa)$. Thus by Blockwise Robust Sunflower Lemma we get that $\mathbf{Pr}_{\mathbf{S_y} \subseteq [mn]}(\forall S_x \in \mathcal{S}_X, S_x \not\subseteq S_y) \leq \kappa$, and if we look at y as being the indicator vector for S_y then we get that $\mathbf{Pr}_{\boldsymbol{y} \sim \{0,1\}^{mn}}(\forall x \in X, y[I_x, x[I_x]] \neq 1^{I_x}) \leq \kappa$. Thus by counting we get

$$\begin{aligned}|Y| &= |\{y \in Y : \forall x, y[I_x, x[I_x]] \neq \beta_x\}| \\ &\leq |\{y \in (\{0,1\}^m)^n : \forall x, y[I_x, x[I_x]] \neq 1^{I_x}\}| \\ &\leq \kappa \cdot 2^{mn} = 2^{mn - 3d\log m}\end{aligned}$$

which is a contradiction as $|Y| > 2^{mn-2d\log m}$ by assumption. □

Marginalizing to sets of size $2^{O(d\log m)}$ causes no issue for us when we apply Full Range Lemma to show that there exists a good j in the rectangle partition, as we can assume that all sets have size at most $O(d)$ by either deficiency or error sets.

The only other place we apply Full Range Lemma—and thus the only other place where the gadget size matters—is at the leaves, where we need to certify that we output the correct answer. As usual assume for contradiction that there exists an assignment x consistent with ρ such that $f(x) \ni o$, where o is the label we assign to the leaf. Our challenge is that we can no longer assert that every joint assignment to *all* remaining free variables exists, and so it is possible that x itself is not in R. However, we note that we only need to focus on C bits of x, where C is the certificate complexity of f, because it is these C bits that fix the value of f. Because $C \leq 2^{O(d\log d)}$, applying Lemma 15 shows that the smallest partial assignment consistent with x which forces the output of f to not include o is indeed in R, and this gives us our contradiction as usual. □

2.5 A note on combinatorics and lifting

To reemphasize one more time, our central contribution to the lifting literature in this chapter is the use of Blockwise Robust Sunflower Lemma to prove Full Range Lemma. The utility of this straightforward application of combinatorics to lifting—an application which reduces a key piece of the analysis to one well-studied lemma and a negligible amount of extra work—should justify itself purely by its simplicity, utility, and future prospects. Nevertheless, we close this chapter by remarking that the addition of Blockwise Robust Sunflower Lemma is an important—or indeed, essential—step forward in lifting, beyond these obvious qualitative and quantitative contributions. In short, the sunflower lemma has always been deeply tied to lifting because their proofs follow the same path, based on building useful mathematical structures within chaotic systems.

In combinatorics, as well as in many other fields, there is a recurring and natural question of finding, inside any sufficiently large collection of objects, a particular well-behaved object of interest; the sunflower lemma itself is an obvious case in point, as are many famous statements, including the recent resolution of the Kahn-Kalai conjecture [PP22], as well as the best known techniques for proving monotone circuit lower bounds [Raz85, CKR20]. For such statements, the central recurring proof technique is that of "structure versus randomness" proofs. The key is to decompose our large collection into two parts: first, we isolate a component which is well-structured, i.e. well-behaved; and second, we treat the rest as being extremely random when we condition on having taken out the structured piece. We incorporate the structured piece into our final object in some way, and then we prove that we find ourselves in a win-win scenario: if the structured component is large, then we have our object of interest; whereas if the random component is large, then we find ourselves in the same place that we started, with an erratic but very large component, and with any drop in size over the original collection being compensated by having made progress towards our final object by utilizing the structured piece.

This is of course a very abstract description, but it may also strike the reader as familiar from the proofs in this chapter. In fact, the entire strategy of decomposing our almost-ρ-structured rectangle into ρ'-structured subrectangles comes directly from this framework: by fixing new coordinates I_j to some assignment α_j (structure), we can assume that the rectangle conditioned on this assignment regains blockwise min-entropy (random).[11] Despite this direct inspiration, however, former proofs of lifting have always involved various non-combinatorial arguments in the random case, unlike most other applications of the structure versus randomness argument. With the application of Blockwise Robust Sunflower Lemma, as well as with our other tweaks such as the new counting argument for $Y_=$, we have grounded the proof directly in such structural combinatorial ideas at every stage of the argument.

The next question is to explore this connection in greater detail. It may strike the reader as odd to claim that lifting is *inherently* related to combinatorics when this whole proof structure may just be one—possibly even a suboptimal one—among many; here it should be remarked that a converse is actually known, and that improvements to Query-to-Communication Lifting Theorem immediately imply better parameters for the sunflower lemma [LLZ18]. This seems to suggest that lifting is intimately tied in with the deeper phenomenon of hardness versus randomness, and that any simulation of communication protocols by way of fixing coordinates via decision tree queries gives us some way of iteratively isolating structure in the communication matrix which is made possible by such mathematical structures. The question is how tight this connection can be made, how this can be generalized to broader gadgets, and ultimately whether or not this same structural approach can come to bear on the KRW Conjecture in the non-monotone world.

Further reading

- *Deterministic Communication vs. Partition Number* [GPW18] / *Query-to-Communication Lifting for BPP* [GPW20]. These two works brought query-to-communication lifting back into the public eye and first developed most of the important tools used in the modern proofs.

[11] The debt to sunflowers is even more obvious when we consider the change from the original coordinate-by-coordinate proofs of [RM99, GPW18]—which explicitly alternated between a "thickness lemma", i.e. guaranteeing randomness, and a "projection lemma", i.e. fixing structure—to later blockwise proofs, a difference which also manifests itself in the change from the basic coordinate-wise recursive proof of the sunflower lemma by [ER60] to modern blockwise "spreadness" proofs a la [ALWZ20].

- *Monotone Circuit Lower Bounds from Resolution* [GGKS20]. The first results for dag-like query-to-communication lifting, and using a proof which has not been significantly changed since. This work also provided the impetus for us to study the Full Range Lemma.
- *Improved Bounds for the Sunflower Lemma* [ALWZ20] / *Coding for Sunflowers* [Rao19] / *A Note on Sunflowers* [BCW21]. The first progress on sunflowers in many decades, this sequence of works also feature very clean proofs. The second paper gives the cleanest overall exposition, while the third one gives a simplification of one key part of the former.
- *Raz-McKenzie Simulation With the Inner Product Gadget* [WYY17] / *Query-to-Communication Lifting Using Low-Discrepancy Gadgets* [CFK$^+$21]. Gives a query-to-communication lifting theorem in the style of the older proofs for the inner product gadget, or in general any gadget satisfying certain nice properties. As of now this is the only other gadget besides the index gadget for which we know query-to-communication lifting holds, although for other types of lifting the gadgets can vary considerably.

Chapter 3

Application: Cutting Planes Proofs are Hard to Find

Now that we have a base lifting theorem at our disposal, it is time to see it in action. In this chapter we will focus on one particular application of query-to-communication lifting in the world of *proof complexity*.

Proof complexity is the study of *propositional proof systems*, which are formal systems of reasoning which allow us to map *unsatisfiable formulas* F to *proofs of unsatisfiability* π, in such a way that the proof π can be efficiently checked for correctness. We measure efficiency of proofs primarily by their *size* (also referred to as their *length*), the definition of which varies from system to system, as well as a mix of related metrics such as *depth*, *degree*, *width*, etc. Two of the central problems in proof complexity, then, are for a given proof system $\mathcal{P}$ to understand: 1) which formulas F require large $\mathcal{P}$-proofs; and 2) how long does it take to find $\mathcal{P}$-proofs of F? We will focus on the second question in this chapter.

Propositional proof systems are by nature *non-deterministic*: a short refutation of a formula F in a particular proof system constitutes an easy-to-check certificate (an NP-witness) of F's unsatisfiability (which is a coNP-property). The question of efficiently finding such refutations is the foundational problem of *automated theorem proving* with applications to algorithm design, e.g., for combinatorial optimization [FKP19]. The following definition is due to Bonet et al. [BPR00].

Definition 10. A proof system $\mathcal{P}$ is *automatable* if there is an algorithm that on input an unsatisfiable CNF formula F outputs some $\mathcal{P}$-refutation of F in time polynomial in the length (or size) of the shortest $\mathcal{P}$-refutation of F.

Several basic propositional proof systems are automatable when restricted to proofs of bounded *width* or *degree*. For example, Resolution refutations of width w can be found in time $n^{O(w)}$ for n-variate formulas [BW01]. Efficient algorithms also exist for finding bounded-degree refutations in algebraic proof systems such as Nullstellensatz, Polynomial Calculus [CEI96], Sherali–Adams, and Sum-of-Squares (under technical assumptions) [O'D17, RW17].

Without restrictions on width or degree, many of these systems are known *not* to be automatable. For the most basic system, Resolution, a long line of work [Iwa97, ABMP01, AR08, MPW19] recently culminated in an optimal non-automatability result by Atserias and Müller [AM20]. They showed that Resolution is not automatable unless $\mathsf{P} = \mathsf{NP}$. This result is optimal in two senses: 1) their reduction shows that it is NP-hard to even approximate the minimum Resolution proof length up to a factor of 2^{n^ϵ} for some

$\epsilon > 0$, which is optimal up to the value of ϵ; 2) because UNSAT is coNP-complete, and any automating algorithm must be in P, we cannot choose a weaker assumption than $\mathsf{P} = \mathsf{NP}$. These results have since been extended to tree-like Resolution [dR21], regular and ordered Resolution [Bel20], Nullstellensatz, Polynomial Calculus, Sherali-Adams [dRGN$^+$21] ordered binary decision diagrams [IR22], and other systems. Furthermore, under stronger hardness assumptions non-automatability results are known for other systems such as k-Resolution [MPW19] and various Frege systems [KP98, BPR97, BDG$^+$04]. Our goal will be to extend the non-automatability results of [AM20, dR21] to the *Cutting Planes* system using lifting techniques we saw in the previous chapter.

Cutting Planes Non-Automatability Theorem. *Let $\mathcal{A}$ be an algorithm which, on input τ which is a unsatisfiable set of m linear equations over n variables which has a Cutting Planes refutation of size s, outputs a Cutting Planes refutation of $\mathcal{A}$. Then assuming $\mathsf{P} \neq \mathsf{NP}$ (assuming the Exponential Time Hypothesis), $\mathcal{A}$ requires time $N^{\omega(1)}$ ($2^{\Omega(N)}$, respectively), where $N = \max(n, m, s)$.*

Let $\mathcal{A}$ be an algorithm which, on input τ which is a unsatisfiable set of m linear equations over n variables which has a tree-like Cutting Planes refutation of size s, outputs a tree-like Cutting Planes refutation of $\mathcal{A}$. Then assuming the Exponential Time Hypothesis, $\mathcal{A}$ requires time $N^{\Omega(\log N/\log^2 \log N)}$, where $N = \max(n, m, s)$.

3.1 Proof complexity and lifting

3.1.1 Proof systems and query models

In order to prove Cutting Planes Non-Automatability Theorem via lifting, we need to connect the proof systems in question to our existing query-to-communication lifting paradigm. And even before that, we need to define the proof systems in question. For all the following definitions, Let $\tau = \{C_1, C_2, \ldots, C_m\}$ be an unsatisfiable CNF formula over $X = \{x_1 \ldots x_n\}$. For the rest of this chapter, we will use N to mean $\max(n, m, s)$, where s is the size of the smallest refutation of τ in whichever proof system we are considering; automatability is always defined with respect to N.

Resolution

The most well-studied proof system in proof complexity is the *Resolution* (Res) system. A Res refutation of τ is a sequence of clauses $\pi = \{D_1, D_2, \ldots, D_S\}$ such that $D_S = \emptyset$, and each line D_i is either some initial clause $C_j \in \tau$ or is derived from two previous lines using the *resolution rule*: from $(E \vee x)$, $(F \vee \overline{x})$ we derive $(E \vee F)$, where $x \in X$, E and F are clauses, and $E \vee F$ is their disjunction with repeated literals removed. We can view a Res proof π as a directed acyclic graph with a unique clause D_i at every vertex, with initial clauses $C_j \in \tau$ at the leaves, $\emptyset$ at the root, and having an edge from D_i to D_j if D_i was used to derive D_j. With this view, a *tree-like Resolution* (tree-Res) refutation is a Res refutation where all non-leaf vertices of the underlying graph have outdegree 1 (so the underlying graph of any tree-Res proof is tree-like).

Given a Res or tree-Res refutation $\pi = \{D_1, D_2, \ldots, D_S\}$, the size of π is the number of lines in π, in this case S. The *width* of a clause D_i is the number of literals in it, and the width of π is the maximum width of a clause in the proof. We also extend the notion of width to *block-width* as follows: let $\mathcal{X}_1 \ldots \mathcal{X}_k$ be a partition of the variables; then the block-width of any Res proof π refuting τ is the maximum

number of blocks any line $D \in \pi$ contains variables from, i.e. we replace each variable with its block label and then take the usual notion of width.

Using these, we define

$$\begin{aligned} \mathsf{res\text{-}dag}(\tau) &:= \text{ least } \textit{size} \text{ of a } \mathsf{Res} \text{ proof refuting } \tau, \\ \mathsf{res\text{-}tree}(\tau) &:= \text{ least } \textit{size} \text{ of a } \mathsf{tree\text{-}Res} \text{ proof refuting } \tau, \\ \mathsf{res\text{-}width}(\tau) &:= \text{ least } \textit{width} \text{ of a } \mathsf{Res} \text{ proof refuting } \tau, \\ \mathsf{res\text{-}block\text{-}width}(\tau) &:= \text{ least } \textit{block-width} \text{ of a } \mathsf{Res} \text{ proof refuting } \tau, \\ \mathsf{res\text{-}tree\text{-}depth}(\tau) &:= \text{ least } \textit{depth} \text{ of a } \mathsf{tree\text{-}Res} \text{ proof refuting } \tau. \end{aligned}$$

where block-width will be defined with respect to some variable partition. Note that for the tree-like models, i.e. tree-Res and decision trees, the relevant notion is depth, while for dag-like models, i.e. Res and decision-dags, the relevant notion is width, or rather, for what follows, block-width. The notion of block-width (with respect to a given variable partition as before) for decision-dags is immediate, and so we define

$$\mathbf{w}(S) := \text{least } \textit{block-width} \text{ of a decision-dag solving } S$$

As with formulas, our goal will be to connect res-dag and res-tree to the size of the associated query models, i.e. dec-dag and dec-tree (the latter term being the *size* of the smallest decision tree), respectively; thus we need to define some search problem $\mathcal{F}_\tau$ whose query complexity captures the complexity of resolution proofs of τ. In the *falsified clause search problem* $\mathcal{S}_\tau$, on an input $\alpha \in \{0,1\}^n$ to τ, our goal is to output some $i \in [m]$ such that the clause C_i is falsified by the assignment α.

Lemma 16. $\mathsf{dec\text{-}dag}(\mathcal{S}_\tau) = \mathsf{res\text{-}dag}(\tau)$ *and* $\mathsf{dec\text{-}tree}(\mathcal{S}_\tau) = \mathsf{res\text{-}tree}(\tau)$*. Furthermore* $\mathsf{dec\text{-}dag\text{-}width}(\mathcal{S}_\tau) = \mathsf{res\text{-}width}(\tau)$ *and* $\mathbf{w}(\mathcal{S}_\tau) = \mathsf{res\text{-}block\text{-}width}(\tau)$*.*

Before moving on to Cutting Planes we note that Res and tree-Res are both known to not be automatable, under widely believed assumptions. These breakthrough results, due to Atserias–Müller [AM20] for Res and de Rezende [dR21] for tree-Res, will be our starting point. These results are not unconditional—obviously they cannot be, as $\mathsf{P} = \mathsf{NP}$ would put automatability in P—but they work off three basic assumptions: 1) $\mathsf{P} \neq \mathsf{NP}$; 2) the *Exponential Time Hypothesis* (ETH), which states that SAT cannot be solved in time $2^{o(n)}$; 3) $\mathsf{W[P]} \neq \mathsf{FPT}$, an assumption from parameterized complexity (see e.g. [AR08, MPW19, dR21]).

Theorem 17.
1. Res *is not automatable in time* $N^{O(1)}$ *unless* $\mathsf{P} = \mathsf{NP}$*, and is not automatable in time* $2^{o(N)}$ *unless* ETH *is false.*
2. tree-Res *is not automatable in time* $N^{O(1)}$ *unless* $\mathsf{W[P]} = \mathsf{FPT}$*, and is not automatable in time* $N^{o(\log N)}$ *unless* ETH *is false.*

Clearly the results for Res are tight with respect to their respective assumptions; this is also essentially true for tree-Res.

Theorem 18. tree-Res *is automatable in time* $N^{O(\log N)}$*.*

Cutting Planes

Cutting Planes (CP) was first introduced algorithmically by Gomory and Chvátal [Gom63, Chv73] and reinterpreted as a proof system by Cook, Coullard, and Turán [CCT87]. This is not only an important system to study in proof complexity writ large, but particularly it is important to study the automatability of Cutting Planes, as it was originally defined in the context of solving integer linear programs and has been at the forefront of satisfiability solving for decades.

A CP refutation is a sequence of *linear threshold functions* (LTFs) $\ell_1, \ldots, \ell_m$, where ℓ_j has the form $\sum_i a_i x_i \geq b$ for $a_i, b \in \mathbb{Z}$. For a CP refutation of τ, we first convert every clause of τ to an LTF in the natural way: 1) we arithmetize each literal x_i as x_i and each literal $\overline{x_i}$ as $(1 - x_i)$; 2) we assert that the sum of the arithmetized literals is at least 1 (and reorganize all constant terms so they only appear on the right hand side). Each line in the refutation is either an LTF version of an input clause or is derived from a previous line; a $\mathsf{tree\text{-}CP}$ refutation is one where the corresponding DAG is a tree. In the original paper [CCT87] the derivation rules were: (1) deriving from $\ell_j, \ell_{j'}$ any non-negative integer linear combination of them; and (2) deriving from $\sum a_i x_i \geq b$ the line $\sum (a_i/c) x_i \geq \lceil b/c \rceil$ where $c \coloneqq \gcd(a_1, \ldots, a_n)$; this ceiling is where all our gains take place, as our goal is to refute integral solutions. Finally in order to refute τ we require that the last line is the contradictory line $0 \geq 1$. We define

$$\begin{aligned}\mathsf{cut\text{-}dag}(\tau) &\coloneqq \text{ least } \textit{size} \text{ of a } \mathsf{CP} \text{ proof refuting } \tau, \\ \mathsf{cut\text{-}tree}(\tau) &\coloneqq \text{ least } \textit{size} \text{ of a } \mathsf{tree\text{-}CP} \text{ proof refuting } \tau.\end{aligned}$$

As a side note, stronger rules have also been studied, e.g., [CKS90, BCC93], the most general being the *semantic rule*, which allows *any* sound inference: ℓ_i can be derived from $\ell_j, \ell_{j'}$ provided every boolean vector $x \in \{0,1\}^n$ that satisfies both ℓ_j and $\ell_{j'}$ also satisfies ℓ_i.[1]

To connect CP and $\mathsf{tree\text{-}CP}$ to query models, we need to use our abstract definition. We define an *LTF-dag* to be a $\mathcal{Q}$-dag where $\mathcal{Q}$ is the set of all n-bit LTFs over $\mathcal{Z}$ (and similarly for *LTF-trees*), and we define

$$\begin{aligned}\mathsf{ltf\text{-}dag}(S) &\coloneqq \text{ least } \textit{size} \text{ of an LTF-dag solving } S, \\ \mathsf{ltf\text{-}tree}(S) &\coloneqq \text{ least } \textit{size} \text{ of an LTF-tree solving } S.\end{aligned}$$

From the way we defined LTF-dags and trees, it should be no surprise that we have an exact equivalence with CP and $\mathsf{tree\text{-}CP}$ refutations.

Lemma 19. $\mathsf{ltf\text{-}dag}(\mathcal{S}_\tau) = \mathsf{cut\text{-}dag}(\tau)$ *and* $\mathsf{ltf\text{-}tree}(\mathcal{S}_\tau) = \mathsf{cut\text{-}tree}(\tau)$.

3.1.2 Communication protocols

Lemmas 16 and 19 have allowed us to connect both Resolution and Cutting Planes to query models; however, in order to utilize the results from Chapter 2 we need not just a query model but also a communication model. As observed in the previous chapter, there is often a trivial way to lower bound a query model, in this case our LTF models, by a communication model simply by splitting up the variables

[1] In this work, we adopt the best of all possible worlds: our lower bounds on CP refutation length will hold even against the semantic system and our upper bounds use the weakest possible rules (in fact, our upper bounds hold for Resolution, which is simulated by every variety of CP).

to make the search problem in question bipartite. We take a similar but more controlled approach, by generalizing the communication models seen in the previous chapter.

Real communication

We say that a function $g : \mathcal{I} \to \mathbb{R}$ is *monotone* iff $\mathcal{I}$ admits a total order $\preceq$ such that $g(\gamma) \leq g(\gamma')$ for every pair $\gamma \preceq \gamma'$. For example, every n-bit LTF is monotone as an n-partite function: the orderings are determined by the signs of the coefficients appearing in the linear form defining f. We also say that a function $g : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is monotone if both $\mathcal{X}$ and $\mathcal{Y}$ admit total orders $\preceq_X$, $\preceq_Y$, such that g is monotone with respect to both orders.[2]

For a bipartite search problem $S : \mathcal{X} \times \mathcal{Y} \times \mathcal{O}$, a *real communication protocol* is a $\mathcal{Q}$-tree where $\mathcal{Q}$ is the set of *(combinatorial) triangles*, which we define as any subset $T \subseteq \mathcal{X} \times \mathcal{Y}$ whose indicator function is monotone. Real protocols were originally introduced in [Kra98] as being communication protocols where each node is labeled with two functions $a_T : \mathcal{X} \to \mathbb{R}$ and $b_T : \mathcal{Y} \to \mathbb{R}$, with the interpretation that $(x, y) \in T$ iff $a_T(x) < b_T(y)$; clearly both definitions are equivalent as $b_T(y) - a_T(x)$ is monotone. For a bipartite search problem S we define

$$\begin{aligned} \textsf{real-cc-tree-depth}(S) &:= \text{ least } \textit{depth} \text{ of a real protocol solving } S, \\ \textsf{tri-dag}(S) &:= \text{ least } \textit{size} \text{ of a real dag-like protocol solving } S. \end{aligned}$$

Real protocols are an established method for proving length lower bounds on $\textsf{tree-CP}$:

Lemma 20 (Krajíček [Kra98])**.** *Let $S_\tau \subseteq \{0,1\}^{n_1} \times \{0,1\}^{n_2} \times \mathcal{O}$ be the search problem $\mathcal{S}_\tau$ for an unsatisfiable CNF formula τ together with an arbitrary bipartition of its $n_1 + n_2$ variables. Then*

$$\textsf{real-cc-tree-depth}(S_\tau) \;\leq\; O(\log \textsf{cut-tree}(\tau)).$$

Most known tree-like lifting theorems extend to real protocols with little difficulty, and this will be the case with our results from Chapter 2 as well. This will be enough to prove the case of $\textsf{tree-CP}$ with no changes to the structure of our lifting.

Multi-party communication

Our definition above naturally extends to real dag-like communication protocols, which are not only interesting for Cutting Planes but also are equivalent to real circuits [HC99, Pud97, HP18]. However, we will give ourselves one more edge in proving our lifting theorem for $\textsf{CP}$, exploiting the fact that we are using a multi-output gadget. Let $k \geq 1$ and consider a fixed k-partite input domain $\mathcal{I} := \mathcal{I}_1 \times \cdots \times \mathcal{I}_k$. We generalize the definition of monotonicity by saying that $g : \mathcal{I} \to \mathbb{R}$ is monotone iff each $\mathcal{I}_j$ admits a total order $\preceq_j$ such that $g(\gamma) \leq g(\gamma')$ for every pair γ, γ' such that $\gamma_j \preceq_j \gamma'_j$ for all j. We also say that a subset $A \subseteq \mathcal{I}_1 \times \cdots \times \mathcal{I}_k$ is a *(combinatorial) k-simplex* if its indicator function is monotone. Note that triangles are equivalent to 2-simplices.

For some partition of the variables $\mathcal{I}_1 \ldots \mathcal{I}_k$, a *simplex protocol* is a $\mathcal{Q}$-tree (or $\mathcal{Q}$-dag) where $\mathcal{Q}$ is again the set of monotone functions over $\mathcal{I}_1 \times \cdots \times \mathcal{I}_k$; we emphasize that any two $f, f' \in \mathcal{F}$ may not

[2] As a technical note we say that $x \preceq_X x$ for all $x \in \mathcal{X}$ and $y \preceq_Y y$ for all $y \in \mathcal{Y}$; this is an unnecessary condition in the non-bipartite case.

agree on the ordering of any part $\mathcal{I}_i$. For simplicity[3] we refer to dag-like simplex protocols as *simplex-dags*. We define

$$\textsf{sim-dag}(S) \;:=\; \text{least } \textit{size} \text{ of a simplex-dag solving } S.$$

The complexity measures introduced so far are related as follows:

$$\textsf{sim-dag}(S_k) \;\leq\; \textsf{ltf-dag}(S_n) \;\leq\; \textsf{dec-dag}(S_n) \;\leq\; n^{O(\textsf{dec-dag-width}(S_n))}.$$

where $S_n \subseteq \{0,1\}^n \times \mathcal{O}$ is any n-bit search problem, and $S_k \subseteq \{0,1\}^{Y_1} \times \cdots \times \{0,1\}^{Y_k} \times \mathcal{O}$ is a k-partite version of S_n obtained from an arbitrary partition $Y_1 \sqcup \cdots \sqcup Y_k = [n]$. The first inequality follows by noting that each LTF f, defined by $\sum_i a_i x_i \geq a_{n+1}$, is a monotone k-partite function when the i-th part $\{0,1\}^{I_i}$ is ordered according to the partial sum $\sum_{j \in Y_j} a_j x_j$ (breaking ties arbitrarily). The second inequality follows since every conjunction is an LTF. The last inequality is standard: the length of any width-w Resolution refutation can be made $n^{O(w)}$ by eliminating repeated clauses (and the same construction works for arbitrary search problems).

Our first motivation to consider multi-party models is that they have a flavor similar to block-width, in the sense that each party can be made responsible for one part of the input variables; the focus on block-width will become apparent when we consider our automatability proof later in this section. The second motivation is that they can be vastly weaker than two-party models, and hence one expects it to be easier to prove lower bounds for k-simplex-dags when k is large. For a toy example, consider the n-bit $\textsc{Xor}_n$ function. It is easy to compute for traditional two-party communication protocols regardless of how the n bits are split between the two players. By contrast, for n parties, each holding one input bit, $\textsc{Xor}_n$ is hard to compute.

3.1.3 Lifting

Proving the non-automatability of Resolution

Lastly we turn our attention to automatability lower bounds in particular, and see how lifting will allow us to move from Resolution to Cutting Planes. Thus we need to understand what our goal is when we want to prove non-automatability results. We start by stating the main lemmas of [AM20, dR21] and use them to prove Theorem 17.

Lemma 21. *1. There is a polynomial-time algorithm that on input an n-variate 3-CNF formula F outputs an unsatisfiable[4] CNF formula* $\mathrm{Ref}(F)$ *such that*

- *If F is* satisfiable*, then* $\mathrm{Ref}(F)$ *admits a $n^{O(1)}$-size $O(1)$-block-width* $\textsf{Res}$ *refutation.*
- *If F is* unsatisfiable*, then* $\mathrm{Ref}(F)$ *requires* $\textsf{Res}$ *refutations of size at leat $2^{\Omega(n)}$ and block-width at least $n^{\Omega(1)}$.*

2. There is a time $2^{O(\sqrt{n})}$ algorithm that on input an n-variate 3-CNF formula F outputs an unsatisfiable CNF formula $\mathrm{Ref}(F)$ *such that*

[3] No pun intended.

[4] Strictly speaking, $\mathrm{Ref}(F)$, as defined in [AM20], may sometimes be satisfiable, in which case its Resolution width/size complexity is understood as ∞. However this case is equivalent to our reformulation, as we can guarantee that $\mathrm{Ref}(F)$ is always unsatisfiable by considering instead the CNF formula $\mathrm{Ref}(F) \wedge T$ where T is some formula over disjoint variables known to require large width (e.g., Tseitin contradictions [Urq87]).

- *If F is* satisfiable*, then* $\mathrm{Ref}(F)$ *admits a* $\mathrm{poly}(n) \cdot 2^{O(\sqrt{n})}$*-size* tree-Res *refutation.*
- *If F is* unsatisfiable*, then* $\mathrm{Ref}(F)$ *requires* tree-Res *refutations of size at least* $2^{\Omega(n)}$.

Proof of Theorem 17. Our argument will proceed analogously in both cases, save for the specifics of runtime. Assume we have an algorithm $\mathcal{A}$ which, given an unsatisfiable CNF τ on n variables with m clauses, outputs a Res refutation in time $s(N)$ for some subexponential function s. We will use it to solve SAT in time $s(N)$, which gives us a contradiction of some assumption based on s.

Our reduction is simple: given a k-CNF F, we apply Lemma 21 to F to obtain $\mathrm{Ref}(F)$ such that $\mathrm{Ref}(F)$ has size $n^{O(1)}$ if F is satisfiable and $2^{\Omega(n)}$ otherwise. We now run $\mathcal{A}$ on $\mathrm{Ref}(F)$, and output "satisfiable" iff $\mathcal{A}$ halts in time s,[5] where $s = s(n^{O(1)})$ for Res and $s = s(2^{O(\sqrt{n})})$ for tree-Res. If F was satisfiable then $\mathrm{Ref}(F)$ does have a refutation of subexponential size, and thus $\mathcal{A}$ will halt by definition. Otherwise $\mathrm{Ref}(F)$ has no refutations of subexponential size, and so regardless of the guarantees on $\mathcal{A}$ there is no way for it to even output a refutation of $\mathrm{Ref}(F)$ in time subexponential time.

To quickly check the specific time conditions for s:

- $\mathcal{P} = \mathsf{Res}, s = N^{O(1)}$: since we can solve SAT in polynomial time, this shows $\mathsf{NP} \subseteq \mathsf{P}$
- $\mathcal{P} = \mathsf{Res}, s = 2^{o(N)}$: since we can solve SAT in subexponential time, this shows ETH is false
- $\mathcal{P} = \mathsf{tree\text{-}Res}, s = \mathrm{poly}(N)$: see [dR21] for definitions and proof, as this will not be useful for our results
- $\mathcal{P} = \mathsf{tree\text{-}Res}, s = o(\mathrm{quasipoly}(N))$: since we can solve SAT in time $N^{o(\log N)} = (\mathrm{poly}(n) \cdot 2^{O(\sqrt{n})})^{o(\sqrt{n})} = 2^{o(n)}$, this shows ETH is false

□

Our goal will be to take these tautologies and lift them to communication models which are closely related to Cutting Planes. Since lifting establishes tight upper and lower bounds on the communication complexity of the lifted function, both the upper bound in the satisfiable case and the lower bound in the unsatisfiable case are preserved–albeit with the $\Theta(\log m)$ factor—and from there we can derive Cutting Planes Non-Automatability Theorem in much the same way as Theorem 17.

Here we can take stock of the challenges facing us in obtaining Cutting Planes automatability lower bounds, even assuming Query-to-Communication Lifting Theorem and Dag-like Lifting Theorem carried over in the real communication realm. Applying Query-to-Communication Lifting Theorem in the case of tree-CP will not work, as our gadget size m will be something like $2^{O(\sqrt{n})}$, which will kill our upper bound as we from a refutation of size s for tree-Res we will get a refutation of size $m^{\mathsf{res\text{-}tree\text{-}depth}(\tau)} = (2^{O(\sqrt{n})})^{O(\sqrt{n})} = 2^n$ for tree-CP for the lifted formula, using the fact that $\mathsf{res\text{-}tree\text{-}depth}(\tau) = \log s$. Meanwhile applying Dag-like Lifting Theorem in the case of CP also fails, because $\mathrm{Ref}(F)$ will actually have width $n^{\Omega(1)}$ in both the upper and lower bound, meaning we lose the crucial gap needed for the reduction to solve SAT.

Tree-like lifting and gadget size

Our tree-like result is the more straightforward of the two. Before anything else, since we do not have a balancing theorem for decision trees, and by extension for tree-Res, we need the following extra condition of the tree-like tautology from Theorem 21.

[5] We do not even read the output of $\mathcal{A}$ here, although if $\mathcal{A}$ is defined to halt and output garbage after a certain amount of time, by the definition of a proof system we can check $\mathcal{A}$s output for validity in time s as well.

Lemma 22. *There is a time $2^{O(\sqrt{n})}$ algorithm that on input an n-variate 3-CNF formula F outputs an unsatisfiable CNF formula* $\mathrm{Ref}(F)$ *such that*

- *If F is* satisfiable*, then* $\mathrm{Ref}(F)$ *admits a $O(\sqrt{n})$-depth* tree-Res *refutation.*
- *If F is* unsatisfiable*, then* $\mathrm{Ref}(F)$ *requires* tree-Res *refutations of depth at least $\Omega(n)$.*

The above result does not directly show up in [dR21], and their result in particular is not amenable to lifting; however, as they discuss, their construction can be modified to give Lemma 22. Unfortunately this modification is only useful for the ETH result in Theorem 17.2, and thus Cutting Planes Non-Automatability Theorem does not extend the result assuming W[P] $\neq$ FPT.

Now that we are working with depth, we can confront the problem of the gadget size. It turns out that we we already created the tools to overcome this issue in Chapter 2: graduated lifting. While the depth in both cases is $\mathrm{poly}(n)$, the formula itself has size $2^{O(\sqrt{n})}$, and so the depth lower bound we are shooting for is only polylogarithmic in the size. Thus applying our graduated lifting gives us a $\log m = \log\log 2^{O(\sqrt{n})}$ loss in the exponent, rather than a $\log N$ loss which would kill our upper bound; while this does cause some loss in parameters over Theorem 17, it gets us quite close.

Dag-like lifting and block width

To move to dag-like lifting, we again consider the issue of width in Lemma 21. As remarked above, the issue is that the width is actually maximally high for both the upper and lower bound. However, we can get our gap by looking not at *width* but at *block-width*. Unlike for the tree-like case, this is immediate from inspecting the proof of [AM20].

Lemma 23. *There is a polynomial-time algorithm that on input an n-variate 3-CNF formula F outputs an unsatisfiable CNF formula* $\mathrm{Ref}(F)$ *such that*

- *If F is* satisfiable*, then* $\mathrm{Ref}(F)$ *admits a $O(1)$-block-width* Res *refutation.*
- *If F is* unsatisfiable*, then* $\mathrm{Ref}(F)$ *requires* Res *refutations of block-width at least $\Omega(n)$.*

We thus focus on proving a lifting theorem for block-width, which brings us back to our idea of using multi-party communication. The main idea is to switch from the index gadget outputting a *single* variable to a whole block of variables at once. If we consider our argument for Dag-like Lifting Theorem, our whole procedure was tuned towards charging the communication protocol for every variable it remembered (i.e. each variable in the width), and so it should be clear why reworking the proof with blocks is a good first step.

To do this, we will need to change the index gadget to be amenable to blockwise lifting. The *column-index* gadget $\mathrm{Ind}_{\ell\times m}\colon [m]\times\{0,1\}^{\ell\times m}\to\{0,1\}^{\ell}$ is defined by $\mathrm{Ind}_{\ell\times m}(x,y) :=$ "x-th column of y"; in other words $\mathrm{Ind}_{\ell\times m}$ is a version of Ind_m where each element of Bob's array that Alice can point to is now a whole vector of bits rather than a single bit. For the sake of making our composition easy, we will think of our one-party function f as having $\ell\cdot n$ inputs rather than n inputs, and so we can define $\mathcal{S}_f\circ\mathrm{Ind}_{\ell\times m}$ in the natural way as before; we refer to this as a *block-composed* search problem since we compose blocks of inputs to $\mathcal{S}_f$ rather than individual inputs.

We just defined block-composed search problems $S\circ\mathrm{Ind}^n_{\ell\times m}$, but how can we translate such objects back to CNF formulas? The standard recipe is as follows. Fix any search problem $S\subseteq\{0,1\}^n\times\mathcal{O}$ (not necessarily of a composed form). Recall our discussion of the *certificate complexity* of S from Section 2.4.

As a converse to the statement about the certificate complexity of $\mathcal{S}_\tau$, any total search problem S of certificate complexity k contains the search problem $\mathcal{S}_F$ associated with some unsatisfiable k-CNF formula F as a subproblem (S is at least as hard as $\mathcal{S}_\tau$). Namely, consider $\tau := \bigwedge_x \neg C_x$ where C_x is the conjunction that checks if the input is consistent with some fixed size-k certificate for x. Note that τ is unsatisfiable because S is total.

For unbounded-width CNF formulas (such as Atserias–Müller's $\mathrm{Ref}(F)$), we need to interpret the above recipe with care. Indeed, fix any unsatisfiable (unbounded-width) CNF formula F with $|F|$ many clauses and such that its $n\ell$ variables are partitioned into n blocks of ℓ variables each. Denote by b the maximum block-width of a clause of F. Then every clause D of F gives rise to a family of certificates for $S_F \circ \textsc{Ind}^n_{\ell\times m}$. Namely, a certificate in the family for D consists of at most $b \log m$ bits (reading b many pointer values associated with the blocks of D) together with $|D|$ many bits read from the pointed-to columns. Thus, altogether, we get at most $|F|m^b$ many certificates, at least one for each input to $S_F \circ \textsc{Ind}^n_{\ell\times m}$. We define $F \circ \textsc{Ind}^n_{\ell\times m}$ as the formula obtained by listing all these certificates (more precisely, the disjunctions that are the negations of the certificates).

The formula $\mathrm{Ref}(F)$ of Atserias and Müller is such that its clauses have block-width 3 [AM20, Appendix A]. Hence $\mathrm{Ref}(F) \circ \textsc{Ind}^n_{\ell\times m}$ has size $n^{O(1)}$ and moreover it is polynomial-time constructible.

Fact 24. *Given an n-variate 3-CNF F, we can construct $\mathrm{Ref}(F) \circ \textsc{Ind}_{\ell\times m}$ in polynomial time.*

Finally, we recall that our second motivation for considering multi-party communication was that it had the potential to simplify the lower bound. In our case, we shift from having a single player Bob to a set of players Bob^{ij}, each of which holds a single bit $y_{i,j}$ for $i \in [\ell]$ and $j \in [m]$. Recall that the difficult part in previous lifting proofs such as Dag-like Lifting Theorem was the proof of Full Range Lemma; in proving the dag-like case of Cutting Planes Non-Automatability Theorem using simplex-dags, having separate Bob players ends up giving an essentially trivial proof of Full Range Lemma, without even resorting to sunflowers.

3.1.4 A note on our proof technique

Here it bears mention that lifting for multi-party models is somewhat novel, particularly in the context of Cutting Planes. Non-automatability results for Cutting Planes have been elusive in part because of the limitations of existing techniques to prove lower bounds on refutation length; the only technique available for some twenty years has been *monotone feasible interpolation* [BPR97, Kra97, HP18], which translates lower bounds for (real) monotone circuits to lower bounds on Cutting Planes length. Historically, the downside with the technique was that it only seemed to apply to highly specialized formulas (e.g., clique-vs-coloring), although the technique was recently extended to handle a more general class of formulas, random $\Theta(\log n)$-CNFs [HP17, FPPR17].

The only other available lower-bound technique is two-party lifting, i.e. Dag-like Lifting Theorem. That technique is also powerful enough to prove lower bounds not only on CP length, but also on monotone circuit size. (Whether lifting should be classified under monotone interpolation is up for debate, since this depends on how broadly one defines monotone interpolation.) In contrast, our lifting theorem is *not* proved through monotone circuit lower bounds, but through the weaker model of computation that is simplex-dags. Considering a large number of communicating parties is what allows us to analyze multi-output gadgets; we do not know how to do this with only two parties.

3.2 Main proof 1: lifting for tree-like Cutting Planes

We first prove Cutting Planes Non-Automatability Theorem for $\textsf{tree-CP}$, as this will follow almost immediately from our previous results. Our main result is a lifting theorem for real communication protocols. We state our real lifting theorem in the most general way, subsuming tree-like all results from Chapter 2.

Theorem 25 (Real Graduated Lifting Theorem). *Let f be a search problem over $\{0,1\}^n$ and let m,δ be such that $\delta \geq \frac{1}{\log m}$, $m^{1-\delta} = \Omega(\textsf{dec-tree-depth}(f) \cdot \log m)$, and either $m^{1-\delta} = \Omega(\log n)$ or f has certificate complexity at most $2^{O(d \log d)}$. Then*

$$\textsf{real-cc-tree-depth}(f \circ \textrm{I}\textsc{nd}_m) \;\geq\; \textsf{dec-tree-depth}(f) \cdot \Omega(\delta \cdot \log m)$$

Proof. We need only go over one minor change to the proof of Query-to-Communication Lifting Theorem. In Simulation Protocol at node v the children of v partition our current rectangle R into two *triangles* T_ℓ, T_r, but our invariant will still be to maintain a rectangle. To do this, let $x_{1/2}$ and $y_{1/2}$ be the median elements of R under $\preceq_X$ and $\preceq_Y$, respectively, and let $c \in \{\ell, r\}$ be such that $(x_{1/2}, y_{1/2}) \in T_c$. Note that by monotonicity, either $R_{\leq} := \{x \times y \in R : x \preceq_X x_{1/2} \wedge y \preceq_Y y_{1/2}\}$ or $R_{\geq} := \{x \times y \in R : x_{1/2} \preceq_X x \wedge y_{1/2} \preceq_Y y\}$ is contained in T_c, and for $R' = X' \times Y' \in \{R_{\leq}, R_{\geq}\}$ satisfying $R' \subseteq T_c$ also satisfies $|X'| \geq |X|/2$ and $|Y'| \geq |Y|/2$. Note that this was already what we assumed in our invariants and when executing Rectangle Partition,[6] and so if we use R' in place of $R \cap R_v$ none of the rest of the proof needs to be changed. This also holds for the tree-like items of Theorems 13 and 14, as well as when $\delta = o(1)$ (see Section 2.2.5). ☐

We can now prove the tree-like case of Cutting Planes Non-Automatability Theorem. As discussed in the previous section, the graduated restriction is the key to avoiding a blowup in the gadget size. We come back to the strength of this result in Chapter 6.

Proof (tree-like). Let $\mathcal{A}$ be any algorithm automating $\textsf{tree-CP}$ in time $s(N)$. Similar to proving Theorem 17, given a k-CNF F we can decide if it is satisfiable by the following procedure: 1) apply Lemma 21 to get an unsatisfiable formula $\mathrm{Ref}(F)$ in time $2^{O(\sqrt{n})}$; 2) lift $\mathrm{Ref}(F)$ using $\textrm{I}\textsc{nd}_m$ and convert it into an unsatisfiable formula τ in polynomial time (see previous section); 3) run $\mathcal{A}$ on τ and output "satisfiable" iff $\mathcal{A}$ halts in time $s(2^{O(\sqrt{n}\log n)})$. By Theorem 25 we have that

$$\textsf{cut-tree}(\tau) = \textsf{ltf-tree}(\mathrm{Ref}(F) \circ \textrm{I}\textsc{nd}_{\ell\times m}) \leq \textsf{res-tree}(\mathrm{Ref}(F) \circ \textrm{I}\textsc{nd}_{\ell\times m}) \leq 2^{O(\textsf{res-tree-depth}(\mathrm{Ref}(F)) \log \textsf{res-tree-depth}(\mathrm{Ref}(F)))}$$

which guarantees that $\textsf{cut-tree}(\tau) \leq 2^{O(\sqrt{n}\log n)}$ if F is satisfiable, and

$$2^{\Omega(\textsf{res-tree-depth}(\mathrm{Ref}(F)) \log \textsf{res-tree-depth}(\mathrm{Ref}(F)))} \leq \textsf{ltf-tree}(\mathrm{Ref}(F) \circ \textrm{I}\textsc{nd}_{\ell\times m}) = \textsf{cut-tree}(\tau)$$

which guarantees that $\textsf{cut-dag}(\tau) \geq 2^{\Omega(n)}$ if F is unsatisfiable. For $s = N^{o(\log N/\log^2 \log N)}$—where now $N == 2^{O(\sqrt{n}\log n)}$ in the SAT case—this allows us to solve SAT in time

$$(2^{O(\sqrt{n}\log n)})^{o(\sqrt{n}\log n/(0.5\log n + \log\log n)^2)} \leq 2^{o(n\log^2 n/(\log^2 n + \log n \log\log n))} = 2^{o(n)}$$

[6]Before we made this assumption even though it was actually the case that either $|X'| \geq |X|/2$ and $|Y'| = |Y|$ or vice-versa, just for the sake of ease of presentation.

thus violating ETH. □

3.3 Main proof 2: lifting for dag-like Cutting Planes

The purpose of this section is to prove our block-lifting theorem, which will give us the dag-like result of Cutting Planes Non-Automatability Theorem.

Theorem 26 (Real Blockwise Lifting Theorem)**.** *Let $S \subseteq (\{0,1\}^\ell)^n \times \mathcal{O}$ be any search problem. For $m := (n\ell)^5$ we have*

$$m^{\Omega(\mathbf{w}(S))} \;\leq\; \textsf{sim-dag}(S \circ \textsc{Ind}^n_{\ell\times m}) \;\leq\; \textsf{dec-dag}(S \circ \textsc{Ind}^n_{\ell\times m}) \;\leq\; m^{O(\mathbf{w}(\Pi))} \cdot |\Pi|,$$

where Π is any decision-dag solving S of size $|\Pi|$ and block-width $\mathbf{w}(\Pi)$.

Since this proof is lengthier, we start by proving the other half of Cutting Planes Non-Automatability Theorem assuming Theorem 26.

Proof (dag-like). Let $\mathcal{A}$ be any algorithm automating CP in time $s(N)$. Similar to proving Theorem 17, given a k-CNF F we can decide if it is satisfiable by the following procedure: 1) apply Lemma 21 to get an unsatisfiable formula $\mathrm{Ref}(F)$ in polynomial time; 2) lift $\mathrm{Ref}(F)$ using $\textsc{Ind}_{\ell\times m}$ and convert it into an unsatisfiable formula τ in polynomial time (see previous section); 3) run $\mathcal{A}$ on τ and output "satisfiable" iff $\mathcal{A}$ halts in time $s(n^{O(1)})$. By Theorem 26 we have that

$$\textsf{cut-dag}(\tau) = \textsf{ltf-dag}(\mathrm{Ref}(F) \circ \textsc{Ind}_{\ell\times m}) \leq \textsf{dec-dag}(\mathrm{Ref}(F) \circ \textsc{Ind}_{\ell\times m}) \leq m^{O(\mathbf{w}(\Pi))} \cdot |\Pi|$$

where Π has size $\mathrm{poly}(n)$ and block-width $O(1)$ by Lemma 21, which guarantees that $\textsf{cut-dag}(\tau) \leq n^{O(1)}$ if F is satisfiable, and

$$m^{\Omega(\mathbf{w}(\mathrm{Ref}(F)))} \leq \textsf{sim-dag}(\mathrm{Ref}(F) \circ \textsc{Ind}_{\ell\times m}) \leq \textsf{ltf-dag}(\mathrm{Ref}(F) \circ \textsc{Ind}_{\ell\times m}) = \textsf{cut-dag}(\tau)$$

which guarantees that $\textsf{cut-dag}(\tau) \geq 2^{\Omega(n)}$ if F is unsatisfiable. Our analysis for each $s(N)$ is the same as for Theorem 17. □

Upper bound. We now move on to proving Theorem 26. The last inequality is trivial as usual; we only sketch it here. Given a decision-dag Π for S, we construct a decision-dag Π' for $S \circ \textsc{Ind}^n_{\ell\times m}$. For every block-width-$b$ conjunction C in Π, there corresponds a family of exactly m^b many conjunctions in Π'. Namely, the family is constructed by replacing each positive literal x_{ij} (resp. negative literal $\bar{x}_{ij}$) of C with a sequence of $\log m + 1$ many literals that witness the j-th output bit of the i-th gadget being 1 (resp. 0). If C has children C', C'' that only touch blocks touched by C, then every conjunction in the family for C can be directly connected to the families of C', C''. However, if C', C'' touch some block i (there can be at most one) that is untouched by C, then the family for C is connected to the families of C', C'' via decision trees that query the pointer value of the i-th gadget. We have $|\Pi'| \leq m^{O(\mathbf{w}(\Pi))} \cdot |\Pi|$, as desired.

Lower bound. To prove the first inequality of Theorem 26, fix a simplex-dag Π solving $S \circ \textsc{Ind}^n_{\ell\times m}$ of size m^d. Our goal is to construct a decision-dag Π' solving S that has block-width $O(d)$. Our proof

follows closely the plan from Dag-like Lifting Theorem, but here our proof of Full Range Lemma is even simpler because of the multi-party nature of our lifting.

3.3.1 Rectangle partition: simplex edition

First we need to rework our definitions from Dag-like Lifting Theorem to the case of *boxes*, i.e. the higher-dimension analogue of rectangles. We define a structured box much in the same way as a structured rectangle, although we do not optimize the condition on the largeness of Y as we are not shooting for quasilinear size gadgets.

Definition 11 (Structured boxes)**.** Let $\rho \in \{0,1,*\}^n$ be a partial assignment with $J := \text{fix}(\rho) \subseteq [n]$. A box $R = X \times Y \prod_{ij} Y^{ij} \subseteq \mathcal{X} \times \prod_{ij} \mathcal{Y}^{ij}$ is *ρ-structured* if the following conditions hold:

1. the gadgets are fixed according to ρ: $\text{IND}^{J}_{\ell\times m}(X_J, \prod_{i\in J,j} Y^{ij}) = \{\rho[J]\}$
2. X has entropy on the free blocks: X_J is fixed to a single value α, and $\mathbf{H}^{\square}_{\infty}(\boldsymbol{X}_{\bar{J}}) \geq 0.9 \log m$
3. Y^{ij}s are large: $|Y^{ij}| \geq 2^{mn-m^{1/2}}$ for $i \in \bar{J}$, $j \in [\ell]$.

The following lemma is the simplex analogue of our Rectangle Lemma. We prove it in Section 3.3.3.

Simplex Lemma. *Let $T \subseteq \mathcal{X} \times \prod_{i,j} \mathcal{Y}^{ij}$ be a simplex and let $d = o(n)$. Then there exists a procedure which outputs $\{X^j \times Y^{j,\beta}\}_{j,\beta}, X_{err}, \{Y^{ij}_{err}\}_{i,j}$, where 1) $T \subseteq \sqcup_{j,\beta} X^j \times Y^{j,\beta}$; 2) X_{err} and each Y^{ij}_{err} have density $2^{-2d\log m}$ in $\mathcal{X}$ and $\mathcal{Y}^{ij}$ respectively; and 3) for each j, β one of the following holds:*

- ***structured**: $X^j \times Y^{j,\beta}$ is $\rho^{j,\beta}$-structured for some $\rho^{j,\beta}$ of width at most $O(d)$; moreover there exists an "inner box" $R^{\circ,j,\beta} \subseteq T \cap R^{j,\beta}$ which is also $\rho^{j,\beta}$-structured*
- ***error**: $R^{j,\beta} \subseteq X_{err} \times \prod_{i,j} \mathcal{Y}^{ij} \cup \bigcup_{i,j} (\mathcal{X} \times Y^{ij}_{err} \times \prod_{i'j' \neq ij} \mathcal{Y}^{ij})$*

Finally, a query alignment property holds: for every $x \in \mathcal{X} \setminus X_{err}$ there exists a subset $I_x \subseteq [n]$ with $|I_x| \leq O(d)$ such that every "structured" $X^j \times Y^{j,\beta}$ intersecting $\{x\} \times \{0,1\}^{mn}$ has $\text{fix}(\rho^{j,\beta}) \subseteq I_x$.

3.3.2 Simulation

As usual our simulation will rely on Full Range Lemma, which will also need to be extended to higher dimensions. We will rely on multi-party gadgets to give us an even simpler proof of Full Range Lemma, but this will prevent us from getting the gadget size improvements from the previous chapter; these will not be necessary for proving our main result.

Lemma 27. *Let $R \coloneqq X \times \prod_{ij} Y^{ij}$ be ρ-structured. Then there is an $x \in X$ so that $\text{IND}_{\ell\times m}(\{x\} \times \prod_{ij} Y^{ij}) = \rho$.*

Proof. Assume for simplicity that $\rho = *^n$. Thus our goal is to find an $x^* \in X$ such that $\text{IND}^{n}_{\ell\times m}(\{x\} \times \prod_{ij} Y^{ij}) = (\{0,1\}^\ell)^n$. The key observation is that since each of the $n\ell$ output bits is determined by a different Bob^{ij}, the output bits are independent: $\text{IND}^{n}_{\ell\times m}(\{x\} \times \prod_{ij} Y^{ij}) = \prod_{ij} \text{IND}_{1\times m}(\{x_i\} \times Y^{ij})$. Therefore it suffices to find an $x \in X$ such that for all $i \in [n]$, $j \in [\ell]$,

$$x \text{ is "good" for } Y^{ij}: \qquad \text{IND}_{1\times m}(\{x_i\} \times Y^{ij}) \;=\; \{0,1\}. \tag{3.1}$$

We claim that a uniform random choice $\boldsymbol{x} \in X$ satisfies all conditions (3.1) with positive probability. Indeed, for a fixed ij, how many "bad" values $x_i \in [m]$ are there that fail to satisfy (3.1)? Each bad value x_i implies that the x_i-th bit is fixed in Y^{ij}. But there can be at most $\mathbf{D}_\infty(Y^{ij}) \leq m^{1/2}$ fixed such bits. Using $\mathbf{H}_\infty(\boldsymbol{x}_i) \geq 0.9 \cdot \log m$ for $i \in [n]$ and recalling that $m = (n\ell)^5$ we have

$$\mathbf{Pr}[\, \boldsymbol{x}_i \text{ is "bad" for } Y^{ij} \,] \;\leq\; m^{1/2} \cdot 2^{-0.9 \log m} \;<\; 1/(n\ell).$$

A union bound over all the $n\ell$ many conditions (3.1) completes the proof. □

Given Simplex Lemma and Lemma 27, our proof will go in much the same way as Dag-like Lifting Theorem, using the inner boxes given by the structured case. As with Section 2.3.2 we can first state our high level idea by ignoring errors, i.e. by making the same assumption as (∗) for Simplex Lemma. We will trace down Π while maintaining a ρ-structured box R, where ρ is the restriction corresponding to our current path in the decision-dag D we have built.

At each node v we partition T_v using Simplex Lemma, obtaining a set of $\rho^{j,\beta}$-structured boxes which cover T_v and which also each have an inner $\rho^{j,\beta}$-structured box. Now our main subroutine for a given node v in Π—starting at the root with $R = \mathcal{X} \times \sqcup_{i,j} \mathcal{Y}^{ij}$—we we apply Simplex Lemma to partition the simplices T_ℓ and T_r associated with the left and right children of v, and apply Lemma 27 to find a row x^* in X which has the full range in free(ρ) available to it. We find a set of structured boxes $R_c^{j,\beta}$ for one of the nodes $c \in \{\ell, r\}$ such that $x^* \in R_c^{j,\beta}$, use query alignment to find the I_x which is fixed for every such box, and query those variables in our decision-dag. We then set ρ to be $\rho^{j,\beta}$ for the result β that we get from the query, namely by forgetting all variables in $\rho \smallsetminus \rho^{j,\beta}$, and then we set R to be the *inner* box of $R^{j,\beta}$ given by Simplex Lemma. Finally when we reach a leaf we output the corresponding label.

To handle the errors, we apply a bottom-up preprocessing step. Starting at the leaves, for every v we first remove all error sets $(X_{err} \times \sqcup_{i,j} Y^{ij}) \cup (\sqcup_{ij} X \times Y^{ij}_{err} \times \sqcup_{i'j' \neq ij} Y^{i'j'})$ coming from descendents of v from T_v, then we apply Simplex Lemma to find the same error sets at v and remove them from T_v as well. By the same union bound argument over m^d nodes,[7] this removes at most a quarter of the mass from the initial box at the root v, and for the rest of the procedure we can assume there are no error sets as we did above.

The analysis will be the same as for Dag-like Lifting Theorem. At the root we have a rectangle $R = T_v$ (since the root note is associated with a triangle that also happens to be a rectangle) which is ρ-structured for $\rho = *^n$, since we have at least half of the mass remaining after removing error sets. At each node we end up with a $\rho^{j,\beta}$-structured rectangle contained within our new T_v by the inner box property, and furthermore we have fixed at most $O(d)$ coordinates by the query alignment property. At the leaves we need only apply Lemma 27 to ensure that T_v has all possible joint assignments to the free coordinates available, which means that the fixed assignment given by the path in our decision-dag D is sufficient to know the output.

3.3.3 Proof of Simplex Lemma

Our first observation is that Rectangle Partition with Errors works equally well to partition boxes $B \subseteq \mathcal{X} \times \prod_{ij} \mathcal{Y}$. Indeed, each part output by Rectangle Scheme is obtained from $R := X \times Y$ by restricting the set X arbitrarily and, crucially, restricting Y only via bit-wise restrictions (Round 2 of

[7] The union bound is now over mn Bob dimensions Y^{ij}, but our density is small enough that this is still miniscule.

Algorithm Triangle Partition with Errors

1: Initialize $\mathcal{F} = \emptyset$, $j = 1$, and $\mathcal{R}_{alive} = \{R \cap T\}$
2: Initialize $X_{err}, Y_{err} = \emptyset$, $\mathcal{R}_{final} = \emptyset$, and $J_y = \emptyset$
3: Initialize $Y^{I,\alpha,\beta} = \{y \in Y : y[I,\alpha] = \beta\}$ for all $(I, \alpha, \beta) \in 2^{[n]} \times [m]^I \times \{0,1\}^I$
4: **PHASE Y-ERR (Y_{err}):**
5: **while** $\exists (I, \alpha, \beta) \in 2^{[n]} \times [m]^I \times \{0,1\}^I \setminus J_y, x \in [m]^n$ such that $|T \cap (x \times Y^{I,\alpha,\beta})| < 2^{mn-m^2}$ **do**
6: Update $Y_{err} \leftarrow Y_{err} \cup Y^{I,\alpha,\beta}$ and $J_y \leftarrow J_y \cup \{(I, \alpha, \beta)\}$
7: Update $Y \leftarrow Y \setminus Y_{err}$
8: **MAIN PHASE:**
9: **while** $\mathcal{R}_{alive} \neq \emptyset$ **do**
10: **for** $R = X \times Y \in \mathcal{R}_{alive}$ **do**
11: **while** $R \neq \mathit{emptyset}$ **do**
12: Let I_j be a maximal subset of $[n]$ such that X violates $0.9 \log m$-blockwise min-entropy on I_j, or let $I_j = \emptyset$ if no such subset exists
13: Let $\alpha_j \in [m]^{I_j}$ be an outcome such that $\mathbf{Pr}_{x \sim \boldsymbol{X}^{\geq j}}(x[I_j] = \alpha_j) > 2^{-0.95|I_j| \log m}$
14: Define $X^j := \{x \in X^{\geq j} : x[I_j] = \alpha_j\}$, and for all $\beta \in \{0,1\}^{I_j}$ define $Y^{j,\beta} = Y^{I_j,\alpha_j,\beta}$
15: If $|X^j \cap T| \geq |X^j|/2$ then update $\mathcal{F} \leftarrow \mathcal{F} \cup \{(I_j, \alpha_j)\}$, update $\mathcal{R}_{final} \leftarrow \mathcal{R}_{final} \cup \{R^{j,\beta}\}_\beta$, and update $j \leftarrow j - 1$
16: Else ($|X^j \cap T| < |X^j|/2$) then update $\mathcal{R}_{alive} \leftarrow \mathcal{R}_{alive} \cup X^{j,top} \times Y$, where $X^{j,top}$ is the half of X^j with larger intersection with T
17: Update $j \leftarrow j + 1$ and update $R \leftarrow R \setminus X^j \times Y$
18: **PHASE X-ERR (X_{err}):**
19: **while** $\exists j \notin J_x$ such that $|I_j| > 40d$ **do**
20: Update $X_{err} \leftarrow X_{err} \cup X^j$ and $J_x \leftarrow J_x \cup \{j\}$
21: return $\mathcal{F}$, $\mathcal{R}_{final}$, X_{err}, Y_{err}

Rectangle Partition with Errors fixes pointed-to bits in all possible ways). But such bit-wise restrictions when applied to a box $B := X \times \prod_{ij} Y^{ij}$ still result in a box. With this understanding, we may do all our normal rectangle partition procedures to a box, and so for the rest of this discussion we will focus on the two-dimensional case, as it generalizes by a similar argument.

To move to triangles (and by extension k-simplices), we will change our procedure slightly. Whenever we isolate an assignment (I_j, α_j) and consider some corresponding structured box $R^{j,\beta} = X^j \times Y^{j,\beta}$ (ignoring the error sets for the moment), we need to ensure that we can get an inner structured box. To do this we will remove any structured box $R^{j,\beta}$ such that $|T \cap X^{j,\beta}| < |X^{j,\beta}|/2$, similar to how we had to add an extra step in Theorem 25 where we make sure to go to the larger side of the rectangle. Since we still have to cover such $R^{j,\beta}$, we will actually recursively perform Rectangle Partition on all such bad $R^{j,\beta}$, and in order to make sure this process converges quickly we will toss out the half of $X^{j,\beta}$ that does not overlap with T.

The only issue with this process is that we could end up killing our inner structured box when we remove the set Y_{err} at the end of our procedure. Thus we will move the Y_{err} phase to the *beginning* of Rectangle Partition with Errors, before we even begin to partition T. This means we will have to loop over *every* potential choice of (I_j, α_j, β_j) since we do not know what the partition procedure will find. In fact, since we are dealing with a triangle T rather than a rectangle R, it is also necessary to loop over all x and make our loop condition $|T \cap (\{x\} \times (Y^{j,\beta} \setminus Y_{err})| < 2^{mn-m^2}$. In [GGKS20] this is called the *Column Cleanup* procedure.[8]

[8] Naturally ours will be the multi-dimensional variant, so $Y^{j,\beta}$ will be a product set as necessary; by our earlier observation this transformation is simple.

Our algorithm is given by Triangle Partition with Errors. Clearly every rectangle $R^{j,\beta}$ returned by Triangle Partition with Errors fulfills the structured case, while the inner box is given by taking the largest box $B = X^j \times Y$ satisfying $B \subseteq T$. Our analysis of the density of X_{err} is the same as Dag-like Lifting Theorem; for Y_{err} we union bound over all possible choices of (I, α, β, x), which gives us $2^n \cdot m^n \cdot 2^n \cdot m^n = 2^{2n(\log m+1)}$ possible sets, each of which has size at most $2^{mn-m^2} \gg 2^{mn-2d\log m-2n(\log m+1)}$.

3.3.4 Afterword: two-party dag-like real lifting with smaller gadgets

We note that the proof of Theorem 26 also works for our usual two-party setup once we revert back to using Full Range Lemma instead of Lemma 27. This gives us a dag-like variant of Theorem 25, or alternatively a real variant of Dag-like Lifting Theorem. Our only job is to make sure the gadget size goes through.

In proving Theorem 26 the gadget size appeared as usual during the equivalent of Full Range Lemma, which we do not need to say any more about now that we are going back to using Full Range Lemma itself. It also, however, appears in a new place, namely while performing the density-restoring partition phase, in this case Simplex Lemma. Our coarse union bound gave us a total of roughly $2^{2n\log m}$ sets, and so this presents no issue for getting a gadget of size $n^{1+\epsilon}$; our Y condition for ρ-structured rectangles can be changed to say $|Y| \geq 2^{mn-O(n\log m)}$ with no other changes.

However, this clearly presents an impediment for doing graduated lifting. Thus far we have managed to use the blockwise min-entropy violation of each (I_j, α_j) to upper bound the size of $\mathcal{F}$, and by extension Y_{err} (or alternatively $Y_=$ in the tree-like case). However, now we have to make our Y_{err} set using *all* possible settings of (I, α), not just the ones that cause blockwise min-entropy violations. Thus we have to use a union bound involving n, which is what prevents us from getting graduated lifting for real dag-like protocols.

We note that this union bound was used by all previous arguments even for Query-to-Communication Lifting Theorem, as the gadget size was nowhere near $n\log n$ and so finding a way around the naïve union bound was probably not considered a priority in previous works. This was also not an issue for tree-like graduated lifting as the argument was very different from Query-to-Communication Lifting Theorem; see Appendix A for more details.

Further reading

- *Automating Resolution is NP-Hard* [AM20]. The first paper to kick off the "modern era" of automatability lower bounds. Most recent results on automatability of other systems adapt their tautology.

- *Automating Tree-Like Resolution in Time $n^{o(\log n)}$ is ETH-Hard* [dR21]. The most relevant extension of [AM20] to our work, which circumvents the huge challenge posed by tree-Res's quasi-polynomial automatability.

- *Resolution is Not Automatizable Unless W[P] is Tractable* [AR08] / *On the Automatability of Polynomial Calculus* [GL10] / *Short Proofs Are Hard To Find* [MPW19]. While these works use an older method for automatability lower bounds, it involves a neat gadget setup which allow both works to turn statements involving *existential* statements, typically a difficult object to work with,

into tautologies involving *universal* statements as is more typical. See Appendix A for an overview of these proofs.

Part II

Space: Algorithms for reusing memory

Chapter 4

Upper Bounds for the Tree Evaluation Problem

In the second technical part of the thesis, we attack the z-f conjecture by showing that $\mathit{space}(\langle z, f\rangle) \ll |z| + \mathit{space}(f)$ for all f and sufficiently large z, where $\langle z, f\rangle(x_1 \ldots x_n) = \langle z, f(x_1 \ldots x_n)\rangle$.

We begin by looking at the approach of [CMW+12] and others to show $\mathsf{TreeEval}$ lower bounds against L. As discussed earlier, there is an algorithm called *pebbling* which was conjectured to be optimal. For $\mathsf{TreeEval}$ the pebbling algorithm can be understood in a simple recursive fashion. Let $\mathit{space}_k(h)$ be the space required to compute a height h instance of $\mathsf{TreeEval}$, fixing the value of k and recalling that we assume $d = 2$. Given any $\mathsf{TreeEval}_{k,2,h}$ instance, in order to compute the root we must compute both of its children, both of which are $\mathsf{TreeEval}_{k,2,h-1}$ instances. We first compute one of the two children, either one, using space $\mathit{space}_k(h-1)$. We then save the value we get, erase all our other memory, and compute the other child. Since we are storing $\log k$ bits for the first child, our space usage is $\max(\mathit{space}_k(h-1), \log k + \mathit{space}_k(h-1)) = \log k + \mathit{space}_k(h-1)$. Now we recurse: to compute the second child we compute both of its children, which we do by computing and saving one and then recursing on the other, and so forth. This strategy gives a clear $h \log k$ upper bound, but it also intuitively seems like this should be a bottleneck since computing one child of the root is useless without knowing the other.

Our angle of attack on this bottleneck will be to accept that this pebbling style recursive computation is necessary, but to object to the innocent suggestion that all the values we store need to be stored in different blocks of memory. The *catalytic computing* framework of [BCK+14], which came out of a fascinating line of work [Bar89, BC92] on branching programs and circuits, proposes a novel way to use space in a more efficient way when computing circuits with simple invertible operations. The idea is deceptively simple: assume that we have a small amount of clean work space but an exponentially larger amount of "catalytic space", which is free to use but is full of junk bits that have to be returned to their original configuration at the end of the computation. Since we have no assumptions on the bits in the catalytic space it would seem like it can't help us compute anything, but Buhrman et al. [BCK+14] show that if we are working with mathematical instructions that are invertible, this invertibility can help us in two ways: first, by letting us use the space in a way that can be easily reset at the end of the computation, and second, by cleverly cancelling out the "noise" that the bits in the catalytic space introduce into the computation by inverting the computation and then subtracting off the contribution of the noise.

While there has been a flurry of work [Pot17, BKLS18, CDKS18, GJST19, DGJ+20] following the

definition of catalytic computing in [BCK+14] (see e.g. [Kou16] for a survey of early results), the preliminary results of [Bar89, BC92] solved a slightly different type of problem. The catalytic computing model involves having a small clean work tape and exponentially more "catalytic space", but [Bar89] and [BC92] study what can be done by constantly reusing a small (even constant size) work tape. Since we are looking to rule out logspace algorithms for TreeEval, it is this latter approach which seems more immediately applicable.

We also make a note here that just having catalytic algorithms is not enough; the pebbling algorithm is optimal for a number of very natural space-bounded algorithms. In the *read-once* restriction, our algorithm only looks at each bit of the input at most once, while in the *thrifty* restriction the algorithm must read only bits corresponding to the actual evaluation of the tree may be read—to wit, if the children of a node v evaluate to x and y, the branching program must not read any values of the function at v other than the value at (x, y). The pebbling algorithm fulfills both of these conditions, but either one of them is enough to guarantee a lower bound of $\Omega(h \log k)$ [EMP18, CMW+12], and neither of these restrictions assume any other structure on the algorithm.

4.1 Preliminaries

In this chapter we will use inputs over the alphabet $[k]$ rather than $\{0, 1\}$. This is not only a trivial generalization to make (since we can always pick $k = 2$), but also for $\mathsf{TreeEval}_{k,2,h}$ we only ever care to read whole values in $[k]$ at once, as every input bit is either part of the value of a leaf or part of an entry in an internal node's table, both of which are values in $[k]$. Note that this will not affect the asymptotics in any of our statements.

4.1.1 Branching programs

Our first model is the standard syntactic notion of space-bounded computation (see [CMW+12]).

Definition 12 (Branching program [CMW+12])**.** Let $k, n := n(k), o := o(n, k) \in \mathbb{N}$, and let $x = \{x_1 \dots x_n\}$ be a set of variables over $[k]$. A *branching program*[1] is a directed acyclic graph G with the following properties:

- There is a single source node v and k^o sink nodes.
- Every non-sink node is labeled with an input variable x_i for $i \in [n]$ and has k outgoing edges, one for each value in $[k]$
- For every $j \in [k]^o$ there is one sink node labeled with j.

The *size* of the branching program is the number of non-sink nodes[2] in G.

Given an assignment to x in $[k]^n$, the execution of G on x is defined by the following process: 1) we initialize v to be the source of G; 2) while v is not a sink, read the value of the x_i labeling v, follow the edge labeled with this value, and update v to be the node we reach; 3) output the value labelling the

[1]Typically a branching program refers to the case when $k = 2$, and a k-wise branching program for general k; this distinction is unnecessary since we are treating the input and output as being values in $[k]$ rather than bits in $\{0, 1\}$.

[2]This is somewhat non-standard, but when talking about layered branching programs this simplifies things by defining the length as the number of times we read variables, which will in turn be connected to the number of instructions in our register program model. This choice does not affect the asymptotics of any results.

sink node v that we reach. The *output* of G, denoted $G(x)$, is the value this process outputs. We say that G computes the function $f : [k]^n \to [k]^o$ if $G(x) = f(x)$ for all $x \in [k]^n$.

Recall from our discussion of NC^1 in Chapter 1 that a *uniform* circuit is one which can be described succinctly; in this chapter we will similarly be focusing on uniform branching programs, although again we do not focus on the exact notion of uniformity in this work.[3] When we focus on uniform branching programs, the connection to space-bounded computation is immediate, and allows us to focus on branching programs for the rest of the thesis.

Observation 1. *Let $k \in \mathbb{N}$ and let $f_n : [k]^n \to [k]^{o(n)}$ be a family of functions. Then there exists a uniform family of branching programs $\{G_n\}$ such that G_n computes f_n and has size $k^{O(s(n))}$ iff f_n can be deterministically computed in space $O(s(n) \log k)$.*

Proof. Consider the natural bijection between nodes in a graph of size k^s and bitstrings of length $s \log k$ describing strings of length s over $[k]$. Then any space $s \log k$ algorithm can be mimicked by a branching program with k^s states by adding edges from state v corresponding to the transitions from the worktape state associated with v; this machine is uniform because the transitions can be locally determined from the Turing Machine. Conversely, any branching program with k^s states can be mimicked by a space $s \log k$ machine (possibly with some overhead for the uniformity of the program) by transitioning on state v according to the edges in the branching program from the node associated with v. □

The branching programs that appear in this work will all have restrictions that make them well-behaved. Most important and well-studied is that of *layered branching programs*. As the name suggests, these are branching programs where the nodes can be organized into successive layers, each one feeding only into the one immediately following it. With the added restriction that every node at the same layer queries the same input variable, this very much resembles a space-bounded Turing Machine model, where our runtime corresponds to the number of layers and we need only index into the current layer at any time stamp. In fact, one utility of this definition comes from paying individual attention to these two quantities, which we call *length* and *width*, rather than simply measuring size.

Definition 13. A branching program is *layered* if, for some $\ell \in \mathbb{N}$, there exists a function $\sigma : G \to [\ell + 1]$ such that for all $u \in G$, the outgoing edges of u go to nodes $v_1 \ldots v_k$ such that $\sigma(v_1) = \ldots = \sigma(v_k) = \sigma(u) + 1$; we call the set of nodes $\{v \in \sigma^{-1}(j)\}$ the jth *layer*. Furthermore for each $j \in [\ell]$ there exists an input variable x_{j_i} which is the variable labeling every $v \in \sigma^{-1}(j)$.[4] The *width* of G is $\max_{j \in [\ell]} |\sigma^{-1}(j)|$ and the *length* of G is ℓ. Note that the size of G is at most the product of the length and width of G.

In many cases we can place even further restrictions on layered branching programs and still capture an interesting class of algorithms. These will not be important for our $\mathsf{TreeEval}$ discussion, but will play a recurring role in Chapter 5, and so we defer them to later.

4.1.2 Register programs

Our second model comes from a line of work starting with [BC92], more recently fleshed out in [BCK+14] and used in many follow-up works on catalytic computation [CM20, CM21].

[3] We are aiming for logspace algorithms for $\mathsf{TreeEval}$ and our branching programs will be at least logspace uniform.

[4] By construction layer $\ell + 1$ will contain exactly the sink nodes of G. See the previous footnote for an explanation of this somewhat non-standard convention.

Definition 14 (Register program)**.** Let $k, n := n(k), o := o(n,k) \in \mathbb{N}$, let $x = \{x_1 \dots x_n\}$ be a set of variables over $[k]$. For some numbers $s \geq o, \ell, t \in \mathbb{N}$, a *register program* P over a ring $\mathcal{R}$ is defined by the following: 1) a set of s *registers* $R_1 \dots R_s$, each storing a value in $\mathcal{R}$, with o of these registers designated as output registers; and 2) an ordered list of t *instructions* where for every $j \in [t]$ the jth instruction has the form $R_\ell \leftarrow R_\ell + p_j(f_j(x_i), R_1, \dots, R_{\ell-1}, R_{\ell+1}, \dots, R_s)$ for some $i \in [n]$ (the *input index*), $\ell \in [s]$, function $f_j : [k] \to \mathcal{R}$, and polynomial $p_j \in \mathcal{R}^s \to \mathcal{R}$. The *size* of P is the number of registers s.

Given an assignment to x in $[k]^n$, the execution of P on x is defined by initializing every register to $0 \in \mathcal{R}$ and then executing each instruction in order. The *output* of P, denoted $P(x)$, is the tuple of values stored in the designated output registers at the end of the execution of P on x. We say that G computes the function $f : [k]^n \to [k]^o$ if $G(x) = f(x)$ for all $x \in [k]^n$.

We use register programs as a convenient means to describe branching programs, and by extension space-bounded machines. When converting a register program to a branching program (Observation 2, below), we will find that instructions that are independent of the input, or instructions that read the same input index as the previous instructions, do not affect the size of the branching program. The following definition of a register program's *time* is motivated by this.

Definition 15 (Time of a register program)**.** Let P be a register program with t instructions. For every $j \in [t]$ the jth instruction has one of the following two forms:

- an *input-dependent instruction* $R_\ell \leftarrow R_\ell + p_j(f_j(x_i), R_1, \dots, R_{\ell-1}, R_{\ell+1}, \dots, R_s)$ for some $i \in [n]$ (the *input index*), $\ell \in [s]$, function $f_j : [k] \to \mathcal{R}$, and polynomial p_j, or
- an *input-independent instruction* $R_\ell \leftarrow R_\ell + p_j(R_1, \dots, R_{\ell-1}, R_{\ell+1}, \dots, R_s)$ for some $\ell \in [s]$ and polynomial p_j.

The *time* of P is determined as follows. Let $i_1, \dots, i_{t'}$ be the input indices of the input-dependent instructions in P, in order, skipping all input-independent instructions. The *time* of P is the number of consecutive runs of the same index in that sequence, or equivalently, $1 + \sum_{j=1}^{t'-1} [i_j \neq i_{j+1}]$.[5]

There are two properties of our allowable instructions to take note of here. First, each register instruction can only directly depend on at most one input variable. We require this so that register programs can be transformed into branching programs; as defined, one direction of the connection between register programs and branching programs—and hence, in the case of uniformity, between register programs and space-bounded Turing Machines—is again immediate.

Observation 2. *Let $k \in \mathbb{N}$, let $\{f_n : [k]^n \to \mathcal{R}^{o(n)}\}_n$ be a family of functions, and let $\{P_n\}_n$ be a family of register programs such that P_n computes f_n with size $s(n,k,o(n))$ and time $t(n,k,o(n))$. Then*

1. *f_n can be computed by a family of layered branching programs $\{G_n\}_n$ of length $t(n)$ and width $|\mathcal{R}|^{s(n)}$; furthermore if $\{P_n\}_n$ is uniform then $\{G_n\}_n$ is also uniform*
2. *if P_n is uniform, then f_n can be deterministically computed in space $s(n) \cdot |\mathcal{R}| + \log t(n)$*

There is a converse to Observation 2 for permutation branching programs, but we will not use this fact anywhere in our results, as our upper bounds will always be in the form of register programs.

[5] Unlike many other models, register programs always run for the exact same amount of time regardless of the actual value of the input, i.e. we get no benefits from getting a "trivial" input; this will not be of any concern to us.

Second, all instructions are *reversible*; for any instruction $R_\ell \leftarrow R_\ell + p_j(f_j(x_i), R_1, \ldots, R_{\ell-1}, R_{\ell+1}, \ldots, R_s)$ in our program, we can always undo it using the instruction $R_\ell \leftarrow R_\ell + (-p_j(f_j(x_i), R_1, \ldots, R_{\ell-1}, R_{\ell+1}, \ldots, R_s))$. We could imagine a more general form of register program similar to space-bounded computation, for example allowing an instruction such as $R_1 \leftarrow 0$, but this reversibility condition will be highly useful throughout our constructions, especially for a restricted model of register programs called *clean* register programs.

Definition 16 (Clean register program)**.** Let $k, n := n(k), o := o(n,k), s := s(n,k,o) \in \mathbb{N}$, let $f : [k]^n \to [k]^o$, and let P be a register program over $\mathcal{R}$ with registers $R_1 \ldots R_s$ computing f; for ease of notation we assume $R_1 \ldots R_o$ are the designating output registers. We say that P *cleanly* computes f if for every value $(\tau_1 \ldots \tau_s) \in \mathcal{R}^s$ the following holds: on any input to x in $[k]^n$, instead of initializing each R_i to $0 \in \mathcal{R}$, we initialize R_i to τ_i for every i, and then execute P as before; then at the end of P's execution on x each non-output register R_i holds its initial value τ_i, while the tuple of output registers $(R_1 \ldots R_o)$ holds the value $(\tau_1 \ldots \tau_o) + f(x)$. We also say that P is a *clean* register program.

While this definition looks very stringent, the restriction to polynomial instructions is well-suited for clean register programs. For the output registers, our instructions are already built to add the outputs of functions to our target registers over the field. For the non-output registers, the reversibility of every instruction gives us an easy way to reset registers. Most importantly, although it is not yet clear how to deal with the initial τ_i values, the polynomial instructions will allow us to precisely control, and hence precisely cancel out, the contributions of the τ_is.

We make a last comment about the focus on register programs, or rather clean register programs, and the connection to composition. The clean restriction may seem unnecessary, because we only want to solve TreeEval in the classic space-bounded computation model where all memory is initialized to zero; in other words, we will not need our final register program for TreeEval to be clean. In short, we want all our *subroutines* to be clean in order to avoid composition lower bounds, as we can save earlier work while running our new computation over top of it. In particular, the power of the clean restriction will come in when we compose register programs with one another.

4.2 Main proof: recursive TreeEval register programs

Our main result will be a family of efficient register programs for TreeEval, and by extension an upper bound on the branching program size and Turing Machine space required to solve the Tree Evaluation Problem.

Tree Evaluation Algorithm. *For any k and h, $\mathsf{TreeEval}_{k,2,h}$ can be solved in space $O(h \log k / \log h) = o(h \log k)$.*

Our jumping off point will be a brilliant result of Ben-Or and Cleve [BC92] showing that TreeEval can be solved very efficiently in the special case when each internal node is the $+$ or $\times$ operation. They were interested in *arithmetic circuits*; for a field $\mathbb{F}$ define $\#\mathsf{NC}^1(\mathbb{F})$ to be the class of all polynomials which can be defined by a polynomial-size logarithmic-depth[6] circuit with fan-in two where all leaves are elements of $\mathbb{F}$ and each internal node computes either $+$ or $\times$.

[6]Note that by definition $h = O(\log n)$ for any $\mathsf{TreeEval}_{k,d,h}$ instance, even if k and d are 2.

Theorem 28 (Logspace TreeEval for $f_v \in \{+, \times\}$ [BC92]). *Let $\mathbb{F}$ be any field and let C be an $\#\mathsf{NC}^1(\mathbb{F})$ circuit. Then there exists a register program over $\mathbb{F}$ of size 3 and time* $\text{poly}(n)$ *which computes C.*

We will not prove Theorem 28 now, as it will be the warm-up for our main technical lemma; however it should be noted that the proof is extremely straightforward and can be shown with a short recursive subroutine and a paragraph of analysis. We bring up this simplicity to make a quick historical side note: [BC92] was directly inspired by the famous Barrington's Theorem [Bar89], which showed that any circuit in NC^1 can be simulated by a width-5 poly-length branching program. Barrington's Theorem was considered a bolt from the blue, both for the shocking result it proved and the somewhat bizarre and unlikely proof. However, in proving Theorem 28, [BC92] succeeded in reproving and generalizing Barrington's Theorem (up to a small loss in the width) even with their very simple proof. In particular, if we let $\mathbb{F} = \mathbb{F}_2$, then $\#\mathsf{NC}^1(\mathbb{F}) = \mathsf{NC}^1$, and combining Theorem 28 with Observation 2 gives us a width-8 poly-length branching program computing any $C \in \mathsf{NC}^1$.[7]

Unfortunately Theorem 28 is a very special case of TreeEval, and the purpose of using TreeEval for lower bounds in the first place is to plug in maximally difficult functions at each internal node. Thus our next goal will be to fit this more arbitrary case back into the framework of $+$ and $\times$. The natural candidate for such a transformation is *interpolation*; we can always look at a function f as being a polynomial over any field. To take just two examples of how to apply this to the function $f_v : [k]^2 \to [k]$ associated with the node v, we could get 1) a degree-2 polynomial over $\mathbb{F}_k$; or 2) a set of $\log k$ different degree-$(2 \log k)$ polynomial over $\mathbb{F}_2$, namely by representing each element in $[k]$ in binary.

However, this is not sufficient by itself.[8] Making this transformation does not immediately tell us the best way to compute TreeEval, and the variety of options of differing field size and degree begs the question of what parameters are relevant to the efficiency of our algorithm. This question of *encodings* is one of the key points of discussion. It should be clear how our choice of storing values comes into play, but to understand how degree comes into the mix, it is necessary to generalize [BC92] to work for products of higher degree; here we contribute to the literature on catalytic computing.

With all of these parameters in mind, our main result, an algorithm for TreeEval beating pebbling, is the result of a tradeoff between the *space* required for storing elements in $[k]$ based on our choice of encoding with the *time* required for computing products in a manner amenable to [BC92], i.e. reducing the number of elements in $[k]$ we need to store well below h, which will be determined by the degree of our polynomial representation of f_v given our chosen encoding. Our focus will be on a representation we call the *d-hot* encoding, which is a generalization of two very basic encodings.[9]

4.2.1 An encoded representation of TreeEval

The catalytic TreeEval approach begins with moving from elements in $[k]$ to a form that will be more amenable to our register programs. For this section we will not deal with clean register programs, and

[7] The construction in [Bar89] is actually stronger than stated, as it is a *permutation branching program*, a restricted type of branching program that we will see in Chapter 5. However we will also see that any clean branching program can be converted into a permutation branching program, and Theorem 28 is indeed clean.

[8] In fact, combining Theorem 28 with either of the polynomials in the previous paragraph will give an algorithm exactly matching the pebbling algorithm.

[9] For clarity, we emphasize that this d is *not* the same as the one in $\mathsf{TreeEval}_{k,d,h}$ as originally defined in Chapter 1. In this chapter we focus on the case of $\mathsf{TreeEval}_{k,2,h}$, i.e. where the fan-in of nodes in our TreeEval instance is 2; in this context, the parameter d has nothing to do with the TreeEval instance itself and will rather be the name of an unrelated parameter which we will optimize in the course of building our TreeEval algorithm. In Section 4.3 we briefly discuss the case of more general fan-in TreeEval instances, and use a different naming convention in order to avoid confusion.

so the reader is invited to think about the simple case where all registers are initialized to 0. This section will also lay the foundation for an inductive argument by which we can compute $\mathsf{TreeEval}_{k,2,h}$ for increasingly large values of h.

Encodings

Since we have chosen to work with register programs over $\mathbb{F}_2$, we will move from $[k]$ to bitstrings; however, we will not restrict ourselves to simply translating $x \in [k]$ into binary. An *encoding* for $[k]$ is a one-to-one mapping E from $[k]$ to $\{0,1\}^\kappa$ for some $\kappa \in \mathbb{N}$. We say that κ is the *size* of the encoding E.

Moving from $[k]$ to encodings E indeed allows us to move from k-wise branching programs to register programs over $\mathbb{F}_2$. In particular, we observe that we can simulate register programs over κ-bit strings by branching programs as in Observation 2, because we can always translate the inputs we read from elements of $[k]$ into E-encoded strings in $\{0,1\}^\kappa$, and at the end we can translate the encoded string representing the output back into $[k]$.

Observation 3. *Let $k, h, \kappa \in \mathbb{N}$, let I be a $\mathsf{TreeEval}_{k,2,h}$ instance for some h, and let E be an encoding of elements in $[k]$ of size κ. Assume there exists a register program of size s and time t computing the E-encoding of I. Then there exists a branching program of width $2^{\kappa \cdot s}$ and length t computing I.*

We note in passing that this gives rise to an algorithm for the already-trivial case of $h = 1$, which forms the base case of our induction.

Corollary 29. *Let $k, \kappa \in \mathbb{N}$, and fix an encoding E of size κ for values in $[k]$. Then $\mathsf{TreeEval}_{k,2,1}$ can be computed by a layered branching program of length 1 and width 2^κ.*

Polynomials

There are many different ways of choosing our encoding E, and one might immediately wonder why we do not immediately choose the simplest and most space-efficient encoding, namely writing each element of $[k]$ in binary using $\kappa = \lceil \log k \rceil$ bits. To answer this, first let us see how we will actually manipulate our κ-bit encoded strings using the polynomial instructions of our register program.

Definition 17 (Polynomial representation of a function)**.** Let $k \in \mathbb{N}$, fix an encoding E of size κ for values in $[k]$, and let $f : [k] \times [k] \to [k]$ be any function. A *polynomial representation of f* with respect to E is a tuple of κ polynomials $\overrightarrow{Q_f} = (Q_{f,1}, \ldots, Q_{f,s})$ over $\mathcal{R} = \mathbb{F}_2$ which together compute f in the following sense: for any $x_\ell, x_r \in [k]$, let $\overrightarrow{p_\ell}, \overrightarrow{p_r} \in \{0,1\}^\kappa$ be their encodings. Then $(Q_{f,i}(\overrightarrow{p_\ell}, \overrightarrow{p_r}))_{i \in [\kappa]}$ is the encoding of $f(x_\ell, x_r)$.

With our encodings we gave a base case for our inductive computation by giving a program to convert a leaf value into our chosen encoding. As an initial example of how to use our new polynomial, we give an example of how to compute an internal node using our polynomials.

Lemma 30. *Let $k, h \in \mathbb{N}$, fix an encoding E of size κ for values in $[k]$, and for some given $\mathsf{TreeEval}_{k,2,h}$ instance I and any node u, let $\overrightarrow{p_u} \in \mathbb{F}_2^s$ denote the encoding of the value at node u. Fix some node v with children v_ℓ and v_r, and let $\overrightarrow{R_\ell}$ and $\overrightarrow{R_r}$ each be a set of s registers such that $\overrightarrow{R_\ell}$ holds value $\overrightarrow{p_{v_\ell}}$ and $\overrightarrow{R_r}$ holds value $\overrightarrow{p_{v_r}}$. Then there is a register program P_v which computes $\overrightarrow{p_v}$ in space 3κ and time k^2.*

Proof. For each $i \in [\kappa]$ define the function $g_i : [k] \to \{0,1\}$ so that $g_i(x)$ is the ith coordinate of the E-encoding of x. Let $\overrightarrow{R_v}$ be a set of s registers where we will store $\overrightarrow{p_v}$.

Let $\overrightarrow{Q_v} = Q_1 \dots Q_s$ be the polynomial representation of $\overrightarrow{p_v}$ with respect to the encoding E and inputs $\overrightarrow{p_\ell}, \overrightarrow{p_r}$. We rewrite each Q_i as $\sum_{(y,z)\in[k]^2} m_{i,y,z}$, where $m_{i,y,z}$ only depends on the value $f_v(y,z)$ and no value $f_v(y',z')$ for $(y',z') \neq (y,z)$. We now compute every monomial in every Q_i in turn by cycling through every pair (y,z) and adding all monomials in $m_{i,y,z}$ to R_i for every i. By definition of $\overrightarrow{Q_v}$ this results in adding $\overrightarrow{p_v}$ to $\overrightarrow{R_v}$.

Clearly all of these instructions read the same piece of the input, namely the same entry in the table of f_v, and all other values are stored in the registers $\overrightarrow{R_\ell}$ and $\overrightarrow{R_r}$. Thus the time according to Definition 15 is k^2, and we used no space besides the 3κ registers in $\overrightarrow{R_v}, \overrightarrow{R_\ell}, \overrightarrow{R_r}$. □

Again we note in passing that this gives us a way to compute the already-trivial case of $h = 2$, which suggests a way to begin our inductive argument on h.

Corollary 31. *Let $k, \kappa \in \mathbb{N}$, and fix an encoding E for values in $[k]$ as κ-bit strings. Then* $\mathsf{TreeEval}_{k,2,2}$ *can be computed by a layered branching program of length $2 + k^2$ and width $2^{3\kappa}$.*

Three specific encodings

It is now clear that for any choice of encoding we can a) translate all inputs we read directly into their encoded values, b) compute the encoded values at any internal node, assuming we already have the encoded values of its children, and c) translate our final result back into $[k]$. Furthermore, we have only paid for the choice of encoding E by having to store 3κ bits for E of size κ, with no runtime or other dependence. Thus Lemma 30 still begs the question of why we would ever consider other encodings.

In this section we will nevertheless define three particular encodings which will be the focus of the next section. The first two are quite simple and natural, and the third is a particular way of generalizing both.[10]

Definition 18 (One-hot, binary, d-hot encodings)**.** Let $k \in \mathbb{N}$, and let $\text{digit}(b, x, i)$ denote the i-th digit of the base b representation of x. For any $x \in [k]$,

- The *one-hot encoding* of x is the vector $\vec{p} \in \{0,1\}^k$ where $p_x = 1$ and $p_{x'} = 0$ for all $x' \neq x$. (k bits)
- The *binary encoding* of x is just x written in base 2; that is, a vector $\vec{p} \in \{0,1\}^{\lceil \log k \rceil}$ where $p_i = \text{digit}(2, x, i)$. ($\lceil \log k \rceil$ bits)
- The *d-hot* encoding of x is parameterized by positive integers b, d where $b^d \geq k$. We write x as d digits in base b, and encode each digit using a one-hot encoding in $\{0,1\}^b$. In other words, the encoding is a vector $\vec{p} \in \{0,1\}^{d \cdot b}$ where for each $i \in [d]$, $p_{i,\text{digit}(b,x,i)} = 1$, and all other coordinates are 0. ($d \cdot b \geq d \cdot \lceil k^{1/d} \rceil$ bits)

[10][CM20] studied a different generalization which achieved weaker results.

	one-hot	binary	base 4	d-hot with $d = 2, b = 4$
0	0000000000000001	0000	00	0001 0001
1	0000000000000010	0001	01	0001 0010
5	0000000000100000	0101	11	0010 0010
15	1000000000000000	1111	33	1000 1000

Table 4.1: Example encodings with $k = 16$, written in reverse to match the usual convention for writing numbers. The encodings are described in Definition 18. The second-last column shows each number in base 4, for comparison with the d-hot encoding.

Figure 4.1 shows examples of each of the encodings in Definition 18. We can also explicitly define the polynomial representation of each of our encodings. For a Boolean statement F we let $[F]$ be the indicator function, i.e. 1 if F is true and 0 otherwise.

Observation 4. *Let $k \in \mathbb{N}$, and let $f : [k] \times [k] \to [k]$ be a function. For each encoding given by Definition 18, the polynomial representation of f given by Definition 17 is as follows:*

- *One-hot encoding: for $w \in [k]$,*

$$Q_{f,w}(\vec{p_\ell}, \vec{p_r}) = \sum_{(y,z)} [f(y,z) = w] p_{\ell,y} p_{r,z}$$

  (degree 2).

- *Binary encoding: for $i \in \lceil \log k \rceil$,*

$$Q_{f,i}(\vec{p_\ell}, \vec{p_r}) = \sum_{(w,y,z) \in [k]^3} [\text{digit}(2, w, i) = 1][f(y,z) = w] \cdot \prod_{i' \in [\lceil \log k \rceil]} (p_{\ell,i'} + \text{digit}(2, y, i') + 1)(p_{r,i'} + \text{digit}(2, z, i') + 1)$$

  (degree $2\lceil \log k \rceil$).

- *d-hot encoding: for $(i, a) \in [d] \times [b]$,*

$$Q_{f,i,a}(\vec{p_\ell}, \vec{p_r}) = \sum_{(w,y,z) \in [k]^3} [\text{digit}(b, w, i) = a][f(y,z) = w] \cdot \prod_{i' \in [d]} p_{\ell,i',\text{digit}(b,y,i')} p_{r,i',\text{digit}(b,z,i')}$$

  (degree $2d$).

For the rest of our discussion we will focus on the d-hot encoding, as it generalizes both other encodings.[11] However we close this section by contrasting the one-hot and binary encodings, representing the two extremes of the d-hot encoding. As discussed, the binary encoding is the most space-efficient choice possible, and thus appears to be optimal given the parameter dependence of Lemma 30. However,

[11] The binary encoding is not exactly the $\log k$-hot encoding, but they behave almost identically in all algorithms in our work, and the only difference in the encoding size is a negligible factor of 2.

the one-hot encoding, as space-inefficient as it may be, minimizes another parameter not yet highlighted: the *degree* of the corresponding polynomial representation. As we will see in the next section, while encoding size will directly correspond to the space of our register programs, polynomial degree will directly correspond to the time of those programs, and hence a tradeoff via the d-hot encoding becomes essential.

4.2.2 Catalytic Products

Let us return to our inductive argument. Fix some $k, h, \kappa \in \mathbb{N}$, some $\mathsf{TreeEval}_{k,2,h}$ instance I, and some encoding E of $[k]$ of size κ. In the previous section showed that we can read and immediately compute the E-encoding of any input value of I, meaning in particular that we can store the encoded value of any leaf of I in an κ-bit register in a single step; this gives us $h = 1$ at almost no cost by Corollary tep-cat:cor:height1. Furthermore, given the encodings of the values of the two children of node v, stored directly into registers R_ℓ and R_r, we can compute the encoded value of the function at node v directly into a third register R_v using the TreeEval polynomials; this gives us $h = 2$ by Corollary tep-cat:cor:height2 with only a modest additional cost over $h = 1$. What more is left to do?

Let us move on with our induction: when we consider $h = 3$ we are faced with a choice. The simple answer for computing the root node is to 1) compute the left child using 3κ binary registers; 2) erase the 2κ registers holding leaf values, and use those plus s more binary registers to compute the right child; 3) erase the leaf values once again, immediately compute the root function into one of the open chunks, and declare victory. This uses a mere 4κ registers and only requires us to read each relevant input—each leaf plus the appropriate entry in each internal node—exactly once! However, when we continue up the tree we immediately see that this is just the pebbling algorithm, and so we cannot expect to use less than $h \log k$ registers total.[12]

Nowhere yet have we really crucially used our encodings nor the TreeEval polynomials associated with them. In this section we put the pieces together and show how to do efficient space-bounded computation of polynomials using our notion of clean computation from Definition 16. The key insight is that using clean computation as a subroutine lets us save space: rather than allocating new registers for the subroutine's scratch work, we can re-use registers the parent computation is already using. Looking back at the $h = 3$ case, this means that in order to compute the root node, we 1) compute the left child using 3κ binary registers; 2) erase the 2κ registers holding leaf values, and use *the same* $3s$ *binary registers* to compute the right child into s of the free $2s$ registers (without erasing the left child); 3) erase the last open s registers, immediately compute the root function into the one open set , and declare victory. In fact for higher h we employ this recursively, where now we erase nothing and do all our computation cleanly, rotating the role of left child, right child, and parent node between each chunk of our 3κ registers.

Thus our main goal is to cleanly compute catalytic products using no additional space. As we will see, the time efficiency of our main lemma depends heavily on the degree of the polynomials we will compute, which finally gives us the motivation for our d-hot encoding; our main theorem will be a consequence of balancing d.

First, as a small technical aside, we define a type of nicely-behaved polynomial, which the TreeEval polynomial for all our encodings obeys by definition.

[12] Since we only read each value once and are "thrifty" with our function reads, this runs into both known k^h lower bounds from our earlier discussion.

Definition 19. Let $s, d \in \mathbb{N}$, and define $X_i := \{x_{i,1} \dots x_{i,d}\}$ for all $i \in [s]$. A polynomial $f(x_{1,1} \dots x_{s,d})$ is *set-multilinear* with respect to $\{X_i\}$ if every monomial in f is the product of exactly one variable from each set X_i. A set of polynomials $f_1 \dots f_t$ is set-multilinear with respect to $\{X_i\}$ if they are all set-multilinear with respect to $\{X_i\}$.

Note that all set-multilinear polynomials are homogeneous of degree d and have at most s^d monomials.

The following is our main subroutine, and also our main contribution to the study of catalytic computing. Here it is important for our recursion that we allow our subroutines to not just cleanly compute the full input from the level below, but rather any *subset* of the inputs from the level below. In previous works [BC92] every input was treated separately, and so this was essentially implicit; in our case, in order to control the number of recursive calls we need for every recursive call to access many, but again not all, inputs separately. This turns out to be a trivial fix, because computing any fixed subset of the outputs given access to programs computing any fixed set of inputs—the necessary structure for our subroutine to work inductively—will be essentially a trivial change from computing all outputs.

Catalytic Product Lemma. *Let $\kappa, d \in \mathbb{N}$, and let $f_1 \dots f_t$ be a set of set-multilinear polynomials with respect to $\{X_i = \{x_{i,j}\}_{j\in[\kappa]}\}_{i\in[d]}$. Suppose that for all subsets $S \subseteq [d] \times [\kappa]$, there exists a register program $P^{in}(S)$ which cleanly computes x_{ij} into register $R^{in}_{i,j}$ for all $(i,j) \in S$. Then for every $T \subseteq [t]$ there exists a register program $P^{out}(T)$ which cleanly computes $f_a(x_{1,1} \dots x_{d,\kappa})$ into R^{out}_a for all $a \in T$. $P^{out}(T)$ makes 2^d calls to programs $P^{in}(S)$, plus $2^d(|T| \cdot \kappa^d)$ basic instructions, and uses only the registers $R^{in}_{i,j}, R^{out}_i$.*

Proof. We proceed in stages, beginning from the case of a single degree-2 $\times$ function over $\mathbb{F}_2$, which we handle in a similar way as [BC92, BCK+14], and moving up to the fully general case of t degree-d polynomials over $\mathbb{F}_2^\kappa$. To do this we tackle the two given challenges, first individually and then together: *parallelizing* over many polynomials, and *scaling* up to higher degrees.

Warm-up: $\kappa = 1, d = 2, t = 1$**.** We start from the simplest possible case: computing a single product of two inputs. As promised at the beginning of this section, this case is exactly Theorem 28, and though it is simple to prove it contains all the centerpieces of our general proof. Let τ^{out}, τ^{in}_1, and τ^{in}_2 be the original values in $R^{out} (:= R^{out}_1)$, $R^{in}_1 (:= R^{in}_{1,1})$, and $R^{in}_2 (:= R^{in}_{2,1})$. Our program P^{out} uses four recursive calls plus four basic instructions:

1: $P^{in}(\{1\})$
2: $R^{out} \leftarrow R^{out} - R^{in}_1 R^{in}_2$ ▷ $R^{out} = \tau^{out} - \tau^{in}_1\tau^{in}_2 - x_1\tau^{in}_2$
3: $P^{in}(\{2\})$
4: $R^{out} \leftarrow R^{out} + R^{in}_1 R^{in}_2$ ▷ $R^{out} = \tau^{out} + \tau^{in}_1 x_2 + x_1x_2$
5: $P^{in}(\{1\})$
6: $R^{out} \leftarrow R^{out} - R^{in}_1 R^{in}_2$ ▷ $R^{out} = \tau^{out} - \tau^{in}_1\tau^{in}_2 + x_1x_2$
7: $P^{in}(\{2\})$
8: $R^{out} \leftarrow R^{out} + R^{in}_1 R^{in}_2$ ▷ $R^{out} = \tau^{out} + x_1x_2$

While correctness is given by the inline comments (the reader should verify the calculations), we motivate this program intuitively in a way that will generalize. Define $y_1 = x_1 + \tau_1$ and $y_2 = x_2 + \tau_2$. We can rewrite the product x_1x_2 over the y and τ variables using the substitution $x_i = y_i - \tau_i$ as

$$x_1x_2 = (y_1 - \tau_1)(y_2 - \tau_2) = y_1y_2 - y_1\tau_2 - \tau_1y_2 + \tau_1\tau_2$$

With this view, we can see each loop in our algorithm as being responsible for one distinct term in our new polynomial, since R_i^{in} either has the value τ_i or the value $x_i + \tau_i = y_i$.

More monomials: $d = 2, t = 1$. The next step is to consider when we have a polynomial f which is multilinear over two sets of variables $X_1 \sqcup X_2$. Looking closely at our algorithm, it has the nice property that if we insert some instruction of the form $R^{out} \leftarrow R^{out} + \delta$ anywhere in the program, the result will simply be that we get an extra δ in R^{out} at the end. This insight allows us to parallelize across all monomials in f in the natural way, by simply running the same program but replacing each step $R^{out} \leftarrow R^{out} + R_1^{in} R_2^{in}$ with steps $R^{out} \leftarrow R^{out} + R_{1,j_1}^{in} R_{2,j_2}^{in}$ for every monomial $x_{1,j_1} x_{2,j_2} \in f$. Each term $x_{1,j_1} x_{2,j_2}$ will be isolated by the instructions of the form $R^{out} \leftarrow R^{out} + R_{1,j_1}^{in} R_{2,j_2}^{in}$ while having no effect on all other calculations running in parallel. Since f is set-multilinear, we can replace $\{1\}$ with $\{(1,j)\}_{j \in [\kappa]}$ and $\{2\}$ with $\{(2,j)\}_{j \in [\kappa]}$ in our recursive calls, and each monomial will individually behave as in the previous lemma.

1: $P^{in}(\{(1,j)\}_{j \in [\kappa]})$
2: **for** $x_{1,j_1} x_{2,j_2} \in f$ **do**
3: $\quad R^{out} \leftarrow R^{out} - R_{1,j_1}^{in} R_{2,j_2}^{in}$ $\quad \triangleright\ R^{out} = \tau^{out} - \sum_{(i_1,i_2)} \tau_{i_1,1}^{in} \tau_{i_2,2}^{in} + x_{i_1,1} \tau_{i_2,2}^{in}$
4: $P^{in}(\{(2,j)\}_{j \in [\kappa]})$
5: **for** $x_{1,j_1} x_{2.j_2} \in f$ **do**
6: $\quad R^{out} \leftarrow R^{out} + R_{1,j_1}^{in} R_{2,j_2}^{in}$ $\quad \triangleright\ R^{out} = \tau^{out} + \sum_{(i_1,i_2)} \tau_{i_1,1}^{in} x_{i_2,2} + x_{i_1,1} x_{i_2,2}$
7: $P^{in}(\{(1,j)\}_{j \in [\kappa]})$
8: **for** $x_{1,j_1} x_{2,j_2} \in f$ **do**
9: $\quad R^{out} \leftarrow R^{out} - R_{1,j_1}^{in} R_{2,j_2}^{in}$ $\quad \triangleright\ R^{out} = \tau^{out} + \sum_{(i_1,i_2)} -\tau_{i_1,1}^{in} \tau_{i_2,2}^{in} + x_{i_1,1} x_{i_2,2}$
10: $P^{in}(\{(2,j)\}_{j \in [\kappa]})$
11: **for** $x_{1,j_1} x_{2,j_2} \in f$ **do**
12: $\quad R^{out} \leftarrow R^{out} + R_{1,j_1}^{in} R_{2,j_2}^{in}$ $\quad \triangleright\ R^{out} = \tau^{out} + \sum_{(i_1,i_2)} x_{i_1,1} x_{i_2,2} = \tau^{out} + f(x)$

As before we use no additional space, and need only four recursive calls. Each internal instruction is now split into one instruction for each monomial in f, which is at most κ^2.

Higher degree: $\kappa = 1, t = 1$. We put parallelism on hold for a moment and consider the other major roadblock, degree. Our goal now is to compute $f(x_1 \dots x_d) = \prod_{i \in [d]} x_i$. Again we consider the transformation of f using the definition $y_i = x_i + \tau_i$, which yields the new polynomial

$$f(y_1 - \tau_1, \dots, y_d - \tau_d) = \sum_{S \subseteq [d]} (-1)^{d-|S|} \cdot (\prod_{i \in S} y_i)(\prod_{i \notin S} \tau_i)$$

As before we will have a loop where each iteration takes care of one of the 2^d terms:

1: **for** $S \subseteq [d]$ **do**
2: $\quad P^{in}(S)$
3: $\quad R^{out} \leftarrow R^{out} + (-1)^{d-|S|} \prod_{i \in [d]} R_i^{in}$
4: $\quad P^{in}(S)$

The correctness of this algorithm is clear from our definition of $f(\vec{y}, \vec{\tau})$. We make two calls to each $P^{in}(S)$, which is twice as many recursive calls as we claimed. This can be fixed by removing the second

call to $P^{in}(S)$ and replacing the first call with $P^{in}(S \Delta S_{old})$, where S_{old} is the set of indices i such that R_i^{in} contains y_i at the start of the loop and $S \Delta S' = (S - S') \cup (S' - S)$ is the symmetric difference function; in other words we simply load in the values necessary to make $R_i^{in} = y_i$ iff $i \in S$ before executing our basic instruction. Thus we get a program that makes 2^d total recursive calls plus one basic instruction for each call.

General case. As in the case of $d = 2$, handling general κ is simply a matter of having an inner loop running over all monomials in the polynomial f. We can also address the parallelism in the output, namely when t is left unfixed, in the same way; clearly no instruction of the form $R_a^{out} \leftarrow R_a^{out} + \prod_{i \in [d]} R_{i,j_i}^{in}$ affects any register $R_{a'}^{out}$, neither directly nor indirectly.

Algorithm Main TreeEval Subroutine: Computing $P^{out}(T)$ using programs P^{in}

1: $S_{old} \leftarrow \emptyset$
2: **for** $S \subseteq [d]$ **do**
3: $\quad P^{in}(\{(i,j)\}_{i \in S \Delta S_{old}, j \in [\kappa]})$
4: $\quad$ **for** $a \in T, \prod_{i \in [d]} x_{i,j_i} \in f_a$ **do**
5: $\quad\quad R_a^{out} \leftarrow R_a^{out} + (-1)^{d-|S|} \prod_{i \in [d]} R_{i,j_i}^{in}$
6: $\quad\quad S_{old} \leftarrow S_{in}$

Our final algorithm is presented in Catalytic Product Procedure Again our efficiency follows immediately by the conditions of each loop. We require a total of (at most) 2^d recursive calls, and for each such call we have one basic instruction for each monomial appearing in any output polynomial f_a, for a total of $2^d(|T| \cdot \kappa^d)$ basic instructions. $\square$

From this we can recursively compute TreeEval, which will give our main result when applied to the d-hot encoding.

Theorem 32 (TreeEval algorithm). *Let $d \in \mathbb{N}$ and let $b \in \mathbb{N}$ be such that $b^d \geq k$. For any subset $T \subseteq [d] \times [b]$ there is a register program with $3db$ registers and length $O(2^{(2d+1)(h-1)} dbk^2)$ that cleanly computes bits T of the d-hot encoding of* $\mathsf{TreeEval}_{k,2,h}$.

Proof. We will prove by induction on the height h that the program has length at most $\frac{2^{(2d+1)h}-1}{2^{2d+1}-1} 2^{2d} dbk^2 = O(2^{(2d+1)(h-1)} dbk^2)$. Lemma 3 solves the base case $h = 1$ using space $db \leq 3db$ and at most $db \leq \frac{2^{(2d+1)0}-1}{2^{2d+1}-1} 2^{2d} dbk^2$ instructions. Now, assume for some height $h \geq 1$ that for every subset $S \subset [d] \times [b]$, bits S of the encoding of $\mathsf{TreeEval}_{k,2,h}$ can be cleanly computed. Given an instance of $\mathsf{TreeEval}_{k,2,h+1}$, Let v be the root and let ℓ and r be the children. Under the induction hypothesis, there exist programs P_ℓ and P_r which can cleanly compute the d-hot encoding of f_ℓ and f_r in space $3db$ and time $\frac{2^{(2d+1)h}-1}{2^{2d+1}-1} 2^{2d} dbk^2$. By Catalytic Product Lemma we can use P_ℓ and P_r to compute the d-hot encoding of f_v—and thus the output for the $\mathsf{TreeEval}_{k,2,h+1}$ instance—in space $3db$ and time at most $2^{2d}\left(2 \frac{2^{(2d+1)h}-1}{2^{2d+1}-1} 2^{2d} dbk^2 + dbk^2\right) = \frac{2^{(2d+1)(h+1)}-1}{2^{2d+1}-1} 2^{2d} dbk^2$ as desired. $\square$

The principle difference between Theorem 32 and previous algorithms that use the "catalytic" approach [CM20, Theorems 1–3] is the choice of encoding. Table 4.2 summarizes the trade-off between time and space for different encodings.

encoding	one-hot	binary	d-hot
encoding bits (Def. 18)	k	$\lceil \log k \rceil$	$db(\geq d\lceil k^{1/d} \rceil)$
total space	$3k$	$3\lceil \log k \rceil$	$3db$
degree (Def. 17)	2	$\lceil \log k \rceil$	d
time for leaf node (Lem. 3)	$3k$	$3\lceil \log k \rceil$	$3db$
time for recursive step (Catalytic Product Lemma)	$2(2t + k^3)$	$k^2(2t + k^2\lceil \log k \rceil^2)$	$2^{2d}(2t + dbk^2)$
total time	$\Theta(4^{h-1}k^3)$	$\Theta((2k^2)^{h-1}k^2 \log^2 k)$	$\Theta(2^{(2d+1)(h-1)}dbk^2)$

Table 4.2: Trade-offs in Theorem 32 if different encodings had been used. The number of registers depends on the encoding (Definition 18). The total number of instructions depends on the number of recursive calls in Catalytic Product Lemma, which in turn depends on the polynomial degree (Definition 17).

Theorem 32 immediately gives us the following two algorithms: one for the one-hot encoding ($d = 1, b = k$) and one for the binary encoding ($d = \log k, b = 2$); we refer readers interested in more precise time/length analysis to Table 4.2.

Theorem 33 (One-hot algorithm). *There exists a register program solving* $\mathsf{TreeEval}_{k,2,h}$ *using* $3k$ *registers over* $\{0,1\}$ *and* $O(2^{2h} \cdot \text{poly}\, k)$ *total instructions.*

Theorem 34 (Binary algorithm). *There exists a register program solving* $\mathsf{TreeEval}_{k,2,h}$ *using* $3\lceil \log k \rceil$ *registers over* $\{0,1\}$ *and* $O((2k^2)^h \cdot \text{poly}\, k)$ *total instructions.*

Lastly, getting Tree Evaluation Algorithm from Theorem 32 is simply a matter of balancing the parameters; we state it more precisely in a form similar to the other two encodings.

Theorem 35 (d-hot algorithm). *Let* $k, h \in \mathbb{N}$. *If* $k \geq h$, *there exists a layered branching program solving* $\mathsf{TreeEval}_{k,2,h}$ *with length at most* $k^{2h/\log h}\,\text{poly}(k)$ *and width at most* $k^{3h/\log h}$. *If* $k < h$, *then there exists a layered branching program solving* $\mathsf{TreeEval}_{k,2,h}$ *with length at most* $2^{2h}\,\text{poly}(k)$ *and width at most* 2^{3h}.

Proof. Set $d = \lceil \log k / \log h \rceil$ and $b = h$; note that $b^d \geq k$. If we apply Theorem 32 for $T = [d \cdot b]$ and for a set of registers initialized to 0, then we get a register program computing the d-hot encoding of $\mathsf{TreeEval}_{k,2,h}$ into some registers while returning all other registers to 0. The register program uses $3dh$ registers and has length $O(2^{(2d+1)(h-1)}dhk^2) \leq 2^{2dh}\,\text{poly}(k)$. When we convert this register program to a branching program, note that each reachable output state corresponds to a different possible value of the d-hot encoding of $\mathsf{TreeEval}_{k,2,h}$ in the output registers plus 0 in all other registers. Since there are only k such values—one for each value in $[k]$—we relabel the k output nodes with the value their output register value corresponds to. Thus this branching program clearly computes $\mathsf{TreeEval}_{k,2,h}$, and the length and width in both cases is as we claimed by plugging in the values of d and b we selected. □

4.2.3 Afterword: more general recursion

Our work in Catalytic Product Lemma is somewhat specialized to the task at hand, and so it is worth noting at this juncture the ways in which this can be used in more general ways. We hope that this lemma in particular can be of future use in catalytic computing.

First, and already implicit in the proof, Catalytic Product Lemma holds over any field $\mathbb{F}$. In our earlier proof, adding was equivalent to subtracting because we were working over $\mathbb{F}_2$; for a more general field, we need to define the inverse programs $(P^{in}(S))^{-1}$. Luckily, as noted in our introductory discussion

all instructions for a register program are reversible, and thus $(P^{in}(S))^{-1}$ can be obtained by running the instructions of $P^{in}(S)$ backwards and flipping the sign in each instruction. This adds no length or width to our program as compared to Catalytic Product Lemma.

Even more generally, it is not too hard to see that for any vector $B := (b_1 \dots b_t) \in \mathbb{F}^t$, we can create a program $P^{out}(A)$ which adds $b_a f_a$ over $\mathbb{F}$ to R_a^{out} for every $a \in [t]$.

1: **for** $S \subseteq [d]$ **do**
2: $\quad P^{in}(\{(i,j)\}_{i \in S, j \in [\kappa]})$
3: $\quad$ **for** $a \in T, \prod_{i \in [d]} x_{ij_i} \in f_a$ **do**
4: $\quad\quad R_a^{out} \leftarrow R_a^{out} + \prod_{i \in [d]} R_{ij_i}^{in}$

Again our efficiency follows immediately by the conditions of each loop. We require a total of (at most) 2^d recursive calls, and for each such call we have one basic instruction for each monomial appearing in any output polynomial f_a, for a total of $2^d(|T| \cdot \kappa^d)$ basic instructions.

4.3 A note on general fan-in TreeEval

As a closing remark for this chapter, we return to the general case of tree evaluation, where we have a full d-ary tree. We show that Tree Evaluation Algorithm generalizes to any d.

Theorem 36. *For any k, d, and h, $\mathsf{TreeEval}_{k,d,h}$ can be solved in space $O(dh \log k / \log h) = o(dh \log k)$.*

Proof. For clarity of notation we let Δ stand for the fan-in in our TreeEval instance—meaning we are given an instance of $\mathsf{TreeEval}_{k,\Delta,h}$—and d be from our d-hot encoding. Note that Catalytic Product Lemma is still relevant here; our polynomials will now be over not just $2d$ blocks of variables, but Δd blocks, as we will have Δ different incoming nodes with a d-hot encoded value in each. Our algorithm will need to be adapted to have $(\Delta+1)db$ registers, as we will need Δdb input registers and db output registers, but other than that our procedure will go uninhibited. We will also need to make $2^{\Delta d}$ recursive calls at each step.

For specific numbers, we will prove by induction on the height h that the program has length at most $\frac{2^{(\Delta d+1)h}-1}{2^{\Delta d+1}-1} 2^{\Delta d} dbk^2 = O(2^{(\Delta d+1)(h-1)} dbk^2)$, while using space $(\Delta+1)db$. Lemma 3 solves the base case $h = 1$ using space $db \leq (\Delta+1)db$ and at most $db \leq \frac{2^{(2d+1)0}-1}{2^{2d+1}-1} 2^{2d} dbk^2$ instructions. Now, assume for some height h that for every subset $S \subset [d] \times [b]$, bits S of the encoding of $\mathsf{TreeEval}_{k,\Delta,h}$ can be cleanly computed for some $h \geq 0$. Given an instance of $\mathsf{TreeEval}_{k,\Delta,h+1}$, Let v be the root and let $v_1 \dots v_\Delta$ be the children. Under the induction hypothesis, there exist programs P_{v_i} which can cleanly compute the d-hot encoding of f_{v_i} in space $(\Delta+1)db$ and time $\frac{2^{(2d+1)h}-1}{2^{2d+1}-1} 2^{2d} dbk^2$. By Catalytic Product Lemma we can use the P_{v_i} programs to compute the d-hot encoding of f_v—and thus the output for the $\mathsf{TreeEval}_{k,\Delta,h+1}$ instance—in space $(\Delta+1)db$ and time at most $\frac{2^{(\Delta d+1)(h+1)}-1}{2^{\Delta d+1}-1} 2^{\Delta d} dbk^2$ as desired.

Clearly the Δ factor does not change our balancing of parameters, and so setting $d = \lceil \log k / \log h \rceil$ and $b = h$ gives us space $(\Delta+1)h \log k / \log h$ and time $O(2^{(\Delta(\log k / \log h)+1)(h-1)} hk^2 \log h / \log k$. ☐

In terms of parameter dependence, the pebbling algorithm gives $O(dh \log k)$ while a logspace algorithm would require $O(h \log d + d \log k)$. As in the case of $d = 2$, the h factor on the $d \log k$ term is the focal point of the lower bound, and Theorem 36 makes the same improvement over pebbling on this factor. Thus the case of larger d seems to be largely irrelevant in the discussion of both upper and lower bounds.

Perhaps an even more direct way to see that larger d is relatively unimportant is to note that we can always turn *any* TreeEval algorithm which only works for some fixed d into one that works for any d, with the tradeoff being one-to-one between fan-in and height. In this light we can think of the statement of Theorem 36 as specifying where the hidden dependence in Tree Evaluation Algorithm on d lies, albeit with better parameters due to the white-box nature of the proof.

Theorem 37. 1. *Let $d \in \mathbb{N}$ and let A_d be an algorithm solving $\mathsf{TreeEval}_{k,d,h}$ for any $k, h \in \mathbb{N}$ in space $s(k, d, h)$. Then for every $d' < d$ there exists an algorithm $A_{d'}$ solving $\mathsf{TreeEval}_{k,d',h}$ for any $k, h \in \mathbb{N}$ in space $s(k, d', h/\log_{d'}(d)) + O(\log n)$.*

2. *Let $d \in \mathbb{N}$ and let A_d be an algorithm solving $\mathsf{TreeEval}_{k,d,h}$ for any $k, h \in \mathbb{N}$ in space $s(k, d, h)$. Then for every $d' > d$ there exists an algorithm $A_{d'}$ solving $\mathsf{TreeEval}_{k,d',h}$ for any $k, h \in \mathbb{N}$ in space $s(k^{d'/d}, d', h \cdot \log_d(d')) + O(\log n)$.*

Proof. In both cases we will describe a logspace reduction from $\mathsf{TreeEval}_{k',d',h'}$ to $\mathsf{TreeEval}_{k,d,h}$, where k' and h' are the corresponding values in each statement. This is enough to complete the proof by applying A_d. For simplicity we ignore divisibility concerns related to d, d', and h.

If $d'/d < 1$, then given a $\mathsf{TreeEval}_{k,d',h}$ instance we will obtain a $\mathsf{TreeEval}_{k,d,h/\log_{d'}(d)}$ instance by merging every $\log_{d'}(d)$ layers of fan-in d' into a single layer of fan-in $d'^{\log_{d'}(d)} = d$. This can clearly be done in logspace as every entry in the function at the new layer can be computed using $\log_{d'}(d)$ entries from the original instance.

If $d'/d > 1$, then given a $\mathsf{TreeEval}_{k,d',h}$ instance we will obtain a $\mathsf{TreeEval}_{k^{d'/d},d,h\cdot\log_d(d')}$ instance by doing the opposite: we will split every layer into $\log_d(d')$ layers of fan-in $d^{\log_d(d')} = d'$. However, we cannot do this naively using elements of $[k]$, as the functions will be restricted by the tree-like structure. Instead, let us view all inputs to the current layer—now split into $\log_d(d') - 1$ new layers plus the one original output layer—as being written in binary with $\log k = d^0 \log k$ bits. At internal layer i we will assume we have inputs written with $d^{i-1} \log k$ bits; we then concatenate those inputs and output the resulting string of $d^i \log k$ bits; this corresponds to an element in $[k^{d^i}]$. Finally at the output layer our function will be the original function, where we interpret the d incoming values in $[k^{d^{\log_d(d')-1}}] = [k^{d'/d}]$ as d' incoming values in $[k]$. □

The upshot of Theorem 37 is that we can always get from one d to another by paying appropriately in h and possibly k. The efficiency, and thus utility, of this simulation depends on how the runtime of A_d depends on k, d, and h; if the goal is to get TreeEval upper bounds against any parameter regime we can prove, it may be useful to choose one setting of d over another. We also note that our upper bound is further strengthened by working against a larger n; this follows because we do not lose any information in this transformation, and in fact we have a lot of redundant information, either in encoding a very well-behaved fan-in-d function as a generic fan-in-d function, or by encoding many levels of concatenation.

Further reading

- *Computing Algebraic Formulas Using a Constant Number of Registers* [BC92]. At only five pages, this is an elegant paper which served as the precursor both to our results and the core techniques of catalytic computing.

- *Computing with a Full Memory: Catalytic Space* [BCK+14]. The introductory paper in the field of catalytic computing is filled with techniques and ideas that underlie all our upper bounds. This paper also provides preliminary lower bounds against catalytic computing, oracle results, and other insights.

Chapter 5

Application: Catalytic/Amortized Algorithms for Every Function

Taking stock of our results from Chapter 4, it seems inevitable that we explore our polynomial technique further: combining it with Catalytic Product Lemma allowed us to compute *arbitrary* functions given by our TreeEval instance. However, the idea of clean computation seems built for iterated functions, and so it is not clear why this technique is useful outside of TreeEval-style composition. For this we need to move to a different type of space-bounded computation and explore what questions are open.

5.1 Catalytic and amortized computation

5.1.1 Catalytic computation

As we have already seen, the question of whether pebbling is optimal for TreeEval—a question we resolved in Chapter 4—partially hinged on whether or not the following space composition theorem holds: given input x and some string z written on the work tape, does computing $f(x)$ while still having z on the work tape when we finish require space $|z| + space(f)$? To take this question on directly, Buhrman et al. [BCK$^+$14] defined a variant of space-bounded computation where the work tape was partially filled by the string z.

Definition 20 (Catalytic space [BCK$^+$14])**.** Consider a space-bounded Turing Machine M which has three tapes: 1) a read-only input tape of length n; 2) a write-only output tape of length m; 3) a read-write work tape of length $s(n)$. A *catalytic Turing Machine* additionally has a read-write *catalytic tape* of length $c(n) \leq 2^{s(n)}$ which is initialized to an arbitrary state $\tau \in \{0,1\}^{c(n)}$. A function f is computable in $CSPACE(s, c)$ if there exists a catalytic Turing Machine M with work space s and catalytic space c such that, after executing M on x with the initial state of the catalytic tape being τ, the output tape contains $f(x)$ and the catalytic tape contains τ.

The term *catalytic* comes from chemistry, where a reaction which normally lacks the energy to occur can take place due to the presence of a reagent, which itself is used up but then ultimately refabricated by the end of the process. In our context, the catalytic tape represents the results of one copy of g that we save while computing a different copy.

5.1.2 Non-uniform catalytic computation

With this definition in mind, it is clear that Catalytic Product Lemma can be used to compute an arbitrary given function f given the right amount of catalytic space. However, there is another important change from TreeEval: f is no longer part of the input, and so a uniform Turing Machine does not know *what* polynomial to compute using Catalytic Product Lemma.

Here is where branching programs come in handy: while they seemed like an interloper in our previous chapter, a needless intermediary between register programs and space-bounded Turing Machines, they have the advantage of capturing *non-uniform* space-bounded computation, just as formulas can capture non-uniform parallel computation. Put simply, a non-uniform model is one where the construction can take the truth table of f, as well as any other information unrelated to the input, into account. Thus we start by moving to the branching program equivalent of catalytic computation.

Definition 21 (m-catalytic branching program [GKM15])**.** Let $k, n := n(k), o := o(n, k), m := m(n, k) \in \mathbb{N}$, and let $x = \{x_1 \dots x_n\}$ be a set of variables over $[k]$. An *m-catalytic branching program* is a directed acyclic graph G with the following properties:

- There are m source nodes and $m \cdot k^o$ sink nodes.
- Every non-sink node is labeled with an input variable x_i for $i \in [n]$ and has k outgoing edges, one for each value in $[k]$
- For every source node u and every $j \in [k]^m$ there is one sink node labeled with (u, j).

As usual, the *size* of the m-catalytic branching program is the number of non-sink nodes in G.

Given an assignment to x in $[k]^n$, the execution of G on x is defined by the following process: 1) set v to be an arbitrary source u of G; 2) while v is not a sink, read the value of the x_i labeling v, follow the edge labeled with this value, and update v to be the node we reach; 3) if v reaches a sink labeled (u, b), output b; otherwise, the program is deemed invalid. The *output* of G, denoted $G(x)$, is the value this process outputs. We say that G computes the function $f : [k]^n \to [k]^o$ if $G(x) = f(x)$ for all $x \in [k]^n$.

The essential condition, which matches up with the catalytic space restriction of resetting the catalytic tape at the end of the execution, is that v always reaches a sink labeled with the source it started at. In particular, it must be the case that from *any* source node u and any assignment to x, this process can only reach sink nodes of the form (u, b).

5.1.3 Amortized space-bounded computation

Potechin [Pot17] also gave a second view of m-catalytic branching programs. Clearly one alternative view of G is that if we restrict our attention to any fixed source node u, the set of reachable states of G on any execution forms a standard branching program, and furthermore if we do this for all source nodes u there is no overlap in the sinks of these residual branching programs. The trick is that outside the sinks, there may be a large amount of overlap between executions from different start nodes, as if these different programs are "sharing" memory configurations.

This suggests a second interesting view of m-catalytic branching programs: it gives a natural (non-uniform) view of *amortized* space complexity. In particular, if f can be computed with branching programs of size s, then for any m it can also be computed by m-catalytic branching programs of size

sm, but it is also possible that for large enough m it can also be computed by m-catalytic branching programs of size $s'm$, where $s' \ll s$.

To recap, given an m-catalytic branching program G computing f which has size s, there are two interpretations of interest. The first is that of a non-uniform catalytic machine with workspace $\log s/m$ and catalytic space $\log m$. The second is that of amortizing over non-uniform space-bounded machines, witnessing that f has amortized space complexity at most $\log s/m$ as long as we amortize over m machines. In both views, we want to simultaneously minimize s and m, or rather (log) s/m, and m, depending on the goal we want to achieve.

In fact, in defining m-catalytic branching programs, [GKM15] gave one such concrete minimization goal: given an *arbitrary function* f—a good start for us, considering we come at this from the perspective of Catalytic Product Lemma—minimize s/m for any choice of m. Clearly minimizing m in isolation is no problem, as every function has an ordinary branching program in the non-uniform world, while on the other hand a basic counting argument shows that s must be at least 2^n for almost every function. Thus, minimizing s/m irrespective of m is perhaps the most natural minimization question which the definition of m-catalytic branching programs poses.

Once we have achieved some s/m which we are happy with, we can then ask how small m can be made while still achieving the optimal or some near-optimal value for s/m. By the counting argument above, there is an inherent cap to how well we can do in this vein; if, for example, we could achieve $s/m = O(n)$, then we cannot hope for m to be any less than $2^n/O(n)$, as we know that $s \approx 2^n$ in the worst case. Achieving $m \approx 2^n$ and $s/m = O(n)$, for example, would show us that while linear space is necessary even in the non-uniform setting, almost all of this space can be made catalytic, which would be very surprising.

We can also ask for the program to have some additional structure. For example, we can require it to be a *read-k* branching program, meaning each variable is only seen at most k times during any execution. These objects are of huge interest in the study of branching programs.[1]

5.1.4 Known results

Along with defining the amortized view of m-catalytic branching programs, [Pot17] also fully answered this question: every function f has an m-catalytic branching program of size $s = O(mn)$, or in other words amortized size $O(n)$. The only catch is that the number of copies is doubly exponential; specifically, there exists a m-catalytic branching program of width $2m$ and length $4n$, where $m = 2^{2^n-1}$. In fact, it is a layered read-4 branching program, almost the most restricted type of program we could hope for.

In terms of amortized size, i.e. s/m, the result of [Pot17] is clearly optimal up to constant factors, and so following [GKM15] the central open question they posed is to understand whether or not m can be improved while maintaining linear amortized size, and what the implications of this result may be. Taking up this challenge, Robere and Zuiddam [RZ21] showed that any function f can be computed by an m-catalytic branching program with the same parameters as [Pot17] even when $m = 2^{\binom{n}{\leq d}-1}$, where d is the degree of f as an $\mathbb{F}_2$ polynomial. Unfortunately this doesn't allow us to go beyond [Pot17] for most functions, but it provides a much sharper analysis for many functions that still appear quite difficult. The proof uses properties of $\mathbb{F}_2$ polynomials under permutations of the input variables.

[1] As an example, recall from our introductory discussion in Chapter 4 that one obstruction to proving TreeEval upper bounds was a result showing that no read-1 program could asymptotically beat pebbling.

Our goal, then, will be to attack this problem of reducing m by using our space-saving catalytic techniques from Tree Evaluation Algorithm.

5.2 Main proof: time/space tradeoffs for catalytic products

In this section, we show that we can indeed adapt our ideas from Tree Evaluation Algorithm in order to make huge improvements on both [Pot17] and [RZ21].

General Catalytic/Amortized Algorithm. *For any $\epsilon \geq 2/n$ and any function $f : \{0,1\}^n \to \{0,1\}$ there is an m-catalytic branching program with length $2^{1/\epsilon} \cdot 2\epsilon n$ and width $2m$ that computes f, where $m = 2^{n+\epsilon^{-1} \cdot 2^{\epsilon n}}$.*

Furthermore, if f is an $\mathbb{F}_2$ polynomial of degree at most d and $\epsilon \geq 2/d$, there is an m-catalytic branching program with length $2^{1/\epsilon} \cdot 2n$ and width $2m$ that computes f, where $m = 2^{n+\epsilon^{-1}\binom{n}{\leq \epsilon d}}$.

Improving or even reproving [Pot17] is not at all immediate from arithmetizing f and applying Catalytic Product Lemma; as before we need to find a way to use space to save in the polynomial degree in some way, or we will end up with a program of length 2^n and will have gained nothing. In fact it seems that our tools are completely backwards: we know how to save plenty of space, but only at the expense of time. So instead we will develop a new version of Catalytic Product Lemma which goes fully the other way and uses *space* to save in *time*. From there we can find a balance between this algorithm and our original Catalytic Product Lemma, ultimately giving General Catalytic/Amortized Algorithm as a consequence. As with Tree Evaluation Algorithm, this tradeoff framework also gives an immediate path forward, as any improvements to either version of Catalytic Product Lemma will shift the tradeoff and give better algorithms.

We present our algorithms in three steps:

- In Section 5.2.1 we show a simpler version of our algorithm which is sufficient to reproduce—with a negligible loss in parameters—Potechin's result [Pot17] that any function can be computed with a linear-amortized-size m-catalytic branching program. Our program has length $4n$ and width $2m$, where $m = 2^{2^n+n}$. This will only be a stepping stone to proving General Catalytic/Amortized Algorithm, introducing the reader to the ways we use, and depart from, the techniques in Chapter 4.
- In Section 5.2.2 we show how to trade off between m and amortized size, yielding for every $k \in [d]$ an m-catalytic branching program of length $2^k \cdot 4\lceil n/k \rceil$ and width $2m$, where $m = 2^{k \cdot 2^{\lceil n/k \rceil}+n}$. This proves the first part of General Catalytic/Amortized Algorithm.
- In Section 5.2.3 we show a simple modification of our first algorithm which reproduces—again with a negligible loss—the result of Robere and Zuiddam [RZ21] that m can be made as small as $2^{\binom{n}{\leq d}-1}$, where d is the degree of f as an $\mathbb{F}_2$ polynomial, with no cost to the length. Our program has length $4n$ and width $2m$, where $m = 2^{\binom{n}{\leq d}+n}$. We then show that the tradeoff algorithm gives us an m-catalytic branching program of length $2^k \cdot 2n$ and width $2m$, where $m = 2^{k \cdot \binom{n}{\leq \lceil d/k \rceil}+n}$.[2] This proves the second part of General Catalytic/Amortized Algorithm.

For the rest of this section, all our register programs will all operate over the field of two elements: $\mathcal{R}_n = \mathbb{F}_2$ for all n.

[2] While the program of [RZ21] matches or beats [Pot17] for all d, our improved version of [RZ21] is worse than our improved version of [Pot17] when $d = \Omega(n)$ (although still an improvement over the original results of both papers), and thus we state and prove both results separately rather than subsuming our improved version of [Pot17].

5.2.1 From space efficiency to time efficiency

In this section, we will prove the central result of [Pot17]:

Theorem 38. *For any function $f : \{0,1\}^n \to \{0,1\}$ there is an m-catalytic branching program with length $4n$ and width $2m$ that computes f, where $m = 2^{n+2^n}$.*

Proof. We will design a program that uses $n + 2^n$ work registers plus one output register R^{out}. First, we have n registers $R_1^{\text{in}}, \ldots, R_n^{\text{in}}$, corresponding to the n input bits. This correspondence is given by the following subroutine, which can be thought of as applying Lemma 3 at every "leaf", with the identity encoding from $\mathbb{F}_2$ to $\mathbb{F}_2$:

1: **procedure** ToggleInput
2:     **for** $i = 1, \ldots, n$ **do**
3:         $R_i^{\text{in}} \leftarrow R_i^{\text{in}} + x_i$

After ToggleInput runs, the registers have values $R_i^{\text{in}} = \tau_i^{\text{in}} + x_i$, where τ_i^{in} stands for the initial value of R_i^{in}. If we run it a second time, the registers are restored to their original values: $R_i^{\text{in}} = \tau_i^{\text{in}}$. Since we query all n variables once, ToggleInput requires time n to run once.

Now as before we view f as being a multilinear polynomial $p_f \in \mathbb{F}_2[x_1 \ldots x_n]$. If we apply Catalytic Product Lemma a single time, we will get an algorithm to compute p_f in time 2^n and using space $n+1$. This is the opposite of our goal, and so we take a different approach. Instead of having a loop iteration for each monomial $m \in p_f$, we will designate a *register* R_m to this monomial instead.

As in the proof of Tree Evaluation Algorithm, we switch to variables $\vec{y}$ and $\vec{\tau}$ by defining $q_f(\vec{y}, \vec{\tau}) = p_f(\overrightarrow{y - \tau})$; note that y_i is also the value in R_i^{in} after running ToggleInput. Every monomial $m \in q_f$ is of the form $\prod_{i \in S} \tau_i \prod_{i \in S'} y_i$ for some sets $S, S' \subseteq [n]$—in fact we know that $S \cap S' = \emptyset$ by construction but we will not need to use this fact. For every $S' \subseteq [n]$ we will have a register $R_{S'}$ which will be responsible for $\prod_{i \in S'} y_i$, which will be observed by the following subroutine:

1: **procedure** ToggleMonomials
2:     ToggleInput
3:     **for** $S' \subseteq [n]$ **do**
4:         $R_{S'} \leftarrow R_{S'} + \prod_{i \in S'} R_i^{\text{in}}$
5:     ToggleInput

After ToggleMonomials runs, we have $R_{S'} = \tau_{S'} + \prod_{i \in S'} y_i$ for each $S' \subseteq [n]$, where $\tau_{S'}$ stands for the register's initial value. The R^{in} registers have their initial values $R_i^{\text{in}} = \tau_i^{\text{in}}$. We run ToggleInput twice and have 2^n additional instructions, but since the additional instructions do not query any x variables they can be computed in the last x query of ToggleInput, for a total runtime of $2n$.

Our final algorithm for computing f is:

Algorithm Time-Efficient Catalytic Product Procedure

1: ToggleMonomials
2: $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S, S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} R_i^{\text{in}} \right) R_{S'}$
3: ToggleMonomials
4: $R^{\text{out}} \leftarrow R^{\text{out}} - \sum_{S, S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} R_i^{\text{in}} \right) R_{S'}$

The proof of the correctness of Time-Efficient Catalytic Product Procedure follows by the same

argument as that of Catalytic Product Procedure. The space of the program is $n + 2^n$ by construction, and lines 2 and 4 are input-independent, giving us a total runtime of $4n$ from the two calls to ToggleMonomials. □

5.2.2 Trading time and space

In this section, we will modify Time-Efficient Catalytic Product Procedure to make m dramatically smaller, in exchange for making the branching program longer. This proves the first part of General Catalytic/Amortized Algorithm.

Proof. Our goal is to balance Time-Efficient Catalytic Product Procedure with Catalytic Product Procedure by removing the registers R_S corresponding to large subsets S and using slow multiplication to build the polynomial q_f from the remaining small subsets. In particular, if we divide the input bits into k groups each of size $\lceil n/k \rceil$, and only store all subsets within each group, then any monomial $c_{S,S'} \prod_{i \in S} \tau_i^{\text{in}} \prod_{i \in S'} y_i$ can be computed by multiplying together one subset from each group, namely the restriction of S to the group. Instead of 2^n registers for all subsets, we use only $k \cdot 2^{\lceil n/k \rceil}$ registers corresponding to subsets in the k groups, and we can compute all the corresponding monomials into these registers in time $2n$ using the first half of Time-Efficient Catalytic Product Procedure. Then since we are only multiplying k monomials together, we can compute q_f using Catalytic Product Lemma in time $2^k \cdot 2 \cdot 2n$, since each recursive call will come from our $2n$ time execution of Time-Efficient Catalytic Product Procedure.

We first restate Catalytic Product Procedure as a register program using the notation from this chapter.

```
1: procedure ToggleSomeInputs(S')
2:     for i ∈ S' do
3:         R_i^in ← R_i^in + x_i
```

Algorithm Space-Efficient Catalytic Product Procedure

```
1: for S' ⊆ [n] do
2:     ToggleSomeInputs(S')
3:     R^out ← R^out + Σ_{S⊆[n]} c_{S,S'} · Π_{i∈[n]} R_i^in
4:     ToggleSomeInputs(S')
```

Now we continue on to balancing Time-Efficient Catalytic Product Procedure and Space-Efficient Catalytic Product Procedure. For $j \in [k]$ let $b_j = \lceil nj/k \rceil$, and divide the range $[n]$ into k groups: $G_1 = \{1, \ldots, b_1\}, G_2 = \{b_1 + 1, \ldots, b_2\}, \ldots, G_k = \{b_{k-1} + 1, \ldots, b_n = n\}$. For each group G_j, we have $2^{|G_j|}$ registers $R_{j,S}$ indexed by subsets $S \subseteq G_j$. As in all previous algorithms we also use n registers $R_1^{\text{in}}, \ldots, R_n^{\text{in}}$, corresponding to the n input bits, for a total of $n + \sum_{j=1}^{k} 2^{|G_j|}$ registers plus the output register R^{out}.

Now to use the $R_{j,S}$ registers, we need to account for $\tau_{j,S}$, which we do in the same way that we dealt with the τ_i^{in} values, namely by rewriting our polynomial q_f once again over a new set of variables. For every $S' \subseteq [n]$, define $S'_j := S' \cap G_j$, and for each $j \in [k]$ and $S \subseteq G_j$, let $z_{j,S} = \tau_{j,S} + \prod_{i \in S} y_i$, where $\tau_{j,S}$ is the initial value of register $R_{j,S}$. Now for every monomial in q_f, we split the term $\prod_{i \in S'} y_i$ in the monomial into k different products $\prod_{i \in S'_j} y_i$, each of which we can replace with $z_{j,S'_j} - \tau_{j,S'_j}$. This gives

us a new polynomial

$$r_f(\overrightarrow{\tau^{\text{in}}}, \vec{\tau}, \vec{z}) = \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{i=1}^{k} (z_{j,S'} - \tau_{j,S'}) \right).$$

As we did with q_f, for $S, S' \subseteq [n]$ and $T \subseteq [k]$, let $d_{S,S',T}$ be the coefficient of $(\prod_{i \in S} \tau_i^{\text{in}})(\prod_{j \in T} z_{j,S'_j})(\prod_{j \in [k] \setminus T} \tau_{j,S'_j})$ in r_f, so that

$$r_f(\overrightarrow{\tau^{\text{in}}}, \vec{\tau}, \vec{z}) = \sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} \sum_{T \subseteq [k]} d_{S,S',T} \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{j \in T} z_{j,S'_j} \right) \left(\prod_{j \in [k] \setminus T} \tau_{j,S'_j} \right)$$

which is equivalent to $f(x_1 \dots f_n)$ as long as $z_{j,S'_j} = \tau_{j,S'_j} + \prod_{i \in S'_j} (x_i + \tau_i^{\text{in}} + 1)$.

Following ToggleSomeInputs(S'), we define new versions of ToggleInput and ToggleMonomials from Section 5.2.1 which focus on some groups and not others. In fact we will only focus on a single group G_j rather than a subset of the groups, as we will order our subsets S' in such a way that we will only ever need to toggle one group at a time:

1: **procedure** ToggleInputForGroup(j)
2:     **for** $i \in G_j$ **do**
3:         $R_i^{\text{in}} \leftarrow R_i^{\text{in}} + x_i$

1: **procedure** ToggleMonomialsForGroup(j)
2:     ToggleInputForGroup(j)
3:     **for** $S \subseteq G_j$ **do**
4:         $R_{j,S} \leftarrow R_{j,S} + \prod_{i \in S} R_i^{\text{in}}$
5:     ToggleInputForGroup(j)

We are now ready to assemble our main algorithm, which completes the proof. As a small note, we will also order our subroutines in a way that allows us to shave a small factor off the length. A *Gray code* [Gra53] $T_0 = \emptyset, \dots, T_{2^k-1}$ is an ordering of all subsets of $[k]$ such that each consecutive pair of sets $T_\ell, T_{\ell+1 \bmod 2^k}$ differs by exactly one element $e_\ell \in [k]$. Performing our loops in this order, we will only need to toggle the group G_{e_ℓ} in each step.

Algorithm Balanced Catalytic Product Procedure

1: **for** $\ell = 0, \dots, 2^k - 1$ **do**
2:     $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} d_{S,S',T_\ell} \left(\prod_{i \in S} R_i^{\text{in}} \right) \left(\prod_{j=1}^{k} R_{j,S'_j} \right)$
3:     ToggleMonomialsForGroup(e_ℓ)

Each time Line 2 is reached, we have $R_{j,S} = \tau_{j,S} + \prod_{i \in S} y_j$ for $j \in T_\ell$, and $R_{j,S} = \tau_{j,S}$ for $j \in [k] \setminus T_\ell$. We also have $R_i^{\text{in}} = \tau_i^{\text{in}}$ for each $i \in [n]$. So the effect of the line is to add

$$\sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} d_{S,S',T_\ell} \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{j \in T_\ell} z_{j,S'_j} \right) \left(\prod_{j \in [k] \setminus T_\ell} \tau_{j,S'_j} \right)$$

to R^{out}. Summing this expression over all possible subsets $T_\ell \subseteq [k]$ gives $R^{\text{out}} = \tau^{\text{out}} + r_f(\cdots) = \tau^{\text{out}} + f(x_1, \dots, x_n)$, and so Balanced Catalytic Product Procedure cleanly computes f. ☐

It is not difficult to save k registers by removing $R_{1,\emptyset},\dots,R_{k,\emptyset}$, as we simply add each value $d_{S,\emptyset,T}(\prod_{i\in S} R_i^{\text{in}})$ to our polynomial without concerning ourselves with any x_i (and by extension any y_i or z_i) variables.

5.2.3 Less space from fewer monomials

Robere and Zuiddam [RZ21, Theorem 5.13] showed that if f is a polynomial of degree $d < n$, it is possible to improve on Potechin's theorem by decreasing $m = 2^{2^n-1}$ down to $m = 2^{\binom{n}{\leq d}-1}$. Here we show how to adapt Time-Efficient Catalytic Product Procedure to get a similar result, and then at the end of the section we build a tradeoff algorithm similar to Balanced Catalytic Product Procedure to improve it.

Theorem 39. *For any function $f : \{0,1\}^n \to \{0,1\}$ which is a degree-d polynomial, there is an m-catalytic branching program with length $4n+1$ and width $2m$ that computes f, where $m \leq 2^{n+\binom{n}{\leq d}}$.*

Again, while this is slightly worse than Robere and Zuiddam's original result, we include it to show the flexibility of our approach and as a stepping stone to our tradeoff result.

Proof. As before, let $p_f \in \mathbb{F}_2[x_1,\dots,x_n]$ be f as a polynomial. We make the following change to Time-Efficient Catalytic Product Procedure: *for every $S' \in [n]$ such that $c_{S,S'} = 0$ for all S, remove the register $R_{S'}$.*

Formally, as before define $\mathbb{F}_2$ polynomials

$$p_f(\vec{x}) = \sum_{\vec{y}\in\{0,1\}^n : f(\vec{y})=1} \prod_{i=1}^{n} (x_i + y_i + 1)$$

$$q_f(\overrightarrow{\tau^{\text{in}}}, \vec{y}) = p_f(\vec{y} - \overrightarrow{\tau^{\text{in}}}) = \sum_{S,S'\subseteq[n]} c_{S,S'} \left(\prod_{i\in S} \tau_i^{\text{in}}\right)\left(\prod_{i\in S'} y_i\right)$$

Let $\mathcal{M}_f \subseteq 2^{[n]}$ be the set of all S' such that there exists an S where $c_{S,S'} \neq 0$. We define the following subroutine:

1: **procedure** ToggleUsefulMonomials
2:     ToggleInput
3:     **for** $S \in \mathcal{M}_f$ **do**
4:         $R_S \leftarrow R_S + \prod_{i\in S} R_i^{\text{in}}$
5:     ToggleInput

The only difference from ToggleMonomials is that we ignore subsets S which are not in $\mathcal{M}_f$ (not "useful"). Our final algorithm is

Algorithm Low-Degree Time-Efficient Catalytic Product Procedure

1: ToggleUsefulMonomials
2: $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S\subseteq[n]} \sum_{S'\in\mathcal{M}_f} c_{S,S'} \left(\prod_{i\in S} R_i^{\text{in}}\right) R_{S'}$
3: ToggleUsefulMonomials
4: $R^{\text{out}} \leftarrow R^{\text{out}} - \sum_{S\subseteq[n]} \sum_{S'\in\mathcal{M}_f} c_{S,S'} \left(\prod_{i\in S} R_i^{\text{in}}\right) R_{S'}$

To conclude the proof of Theorem 39, we need to show $|\mathcal{M}_f| \leq \binom{n}{\leq d}$. Indeed, since p_f is a degree-d

polynomial, q_f also has degree d, which means $c_{S,S'} = 0$ whenever $|S| + |S'| > d$. So, $\mathcal{M}_f$ only contains sets S' with size at most d, of which there are $\binom{n}{\leq d}$. □

The original algorithms of [Pot17, RZ21] rely on the *symmetries* of f as an $\mathbb{F}_2$ polynomial, in essence having each start state represent a possible function g which can be obtained from f by negating input variables or taking $\oplus$ with f itself. [Pot17] takes this set of functions to be the space of all n-variable functions, while [RZ21] analyzes these rules and obtains a more exact characterization. While this characterization is phrased in terms of orbit closures, it can also be described in terms of polynomials as the span of the set of all monomials appearing in f as an $\mathbb{F}_2$ polynomial along with all *submonomials* of this set; this exactly coincides with our notion as $\prod_{i\in S} y_i$ generates all submonomials $\prod_{i\in S'\subseteq S} x_i$ for $y_i := x_i + \tau_i$, which leads to the quantitative results being essentially the same despite taking two completely different approaches.

Now we state our tradeoff algorithm, which goes much in the same way as the proof of the first part of General Catalytic/Amortized Algorithm but without breaking the variables into groups. This will prove the second part of General Catalytic/Amortized Algorithm and thus complete the proof of our main result.

Proof. For any $\Delta \in \mathbb{N}$, let $\mathcal{M}_f^{\Delta} \subseteq \binom{n}{\leq \Delta}$ be the set of all S'' of size at most Δ such that there exists an $S \subseteq [n]$ and $S' \supseteq S''$ where $c_{S,S'} \neq 0$. We will have k registers $R_{j,S''}$ for every $S'' \in \mathcal{M}_f^{\lceil d/k \rceil}$, as well as the usual registers $R_1^{\text{in}} \dots R_n^{\text{in}}, R^{\text{out}}$. Note that this gives us our target space, as $|\mathcal{M}_f^{\lceil d/k \rceil}| \leq \binom{n}{\leq \lceil d/k \rceil}$.

Following our proof of the first part of General Catalytic/Amortized Algorithm, let $z_{j,S} = \tau_{j,S} + \prod_{i\in S} y_i$, where $\tau_{j,S}$ is the initial value of register $R_{j,S}$, and for every monomial in q_f we split the term $\prod_{i\in S'} y_i$ in the monomial arbitrarily into k different products $\prod_{i\in S'_j} y_i$—each of which we can replace with $z_{j,S'_j} - \tau_{j,S'_j}$—where $S'_j \in \mathcal{M}_f^{\lceil d/k \rceil}$ and $\cup_{j\in[k]} S'_j = S'$. This is possible because each non-zero term in q_f has degree at most d, meaning that $|S'| \leq d$ and furthermore every subset of S' of size at most $\lceil d/k \rceil$ appears in $\mathcal{M}_f^{\lceil d/k \rceil}$ by construction.[3]

Fixing some particular partition $(S'_j)_{j\in[k]}$ for each S', this gives us a new polynomial

$$r_f(\overrightarrow{\tau^{\text{in}}}, \vec{\tau}, \vec{z}) = \sum_{S\subseteq[n]} \sum_{S'\subseteq[n]} \sum_{T\subseteq[k]} d_{S,S',T} \left(\prod_{i\in S} \tau_i^{\text{in}}\right) \left(\prod_{j\in T} z_{j,S'_j}\right) \left(\prod_{j\in[k]\setminus T} \tau_{j,S'_j}\right)$$

which is equivalent to $f(x_1 \dots f_n)$ as long as $z_{j,S'_j} = \tau_{j,S'_j} + \prod_{i\in S'_j}(x_i + \tau_i^{\text{in}} + 1)$. We define ToggleMonomialsForGroup as before, using ToggleInput instead of ToggleInputForGroup since the variables are no longer split into groups, and using a Gray code we get our final algorithm:

Algorithm Low-Degree Balanced Catalytic Product Procedure

1: **for** $\ell = 0, \dots, 2^k - 1$ **do**
2: $\quad R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S\subseteq[n]} \sum_{S'\subseteq[n]} d_{S,S',T_\ell} \left(\prod_{i\in S} R_i^{\text{in}}\right) \left(\prod_{j=1}^{k} R_{j,S'_j}\right)$
3: $\quad$ ToggleMonomialsForGroup(e_ℓ)

[3] Our use of j here is slightly different than in our previous proof; namely, j is not linked to a specific block of variables, and rather we arbitrarily partitioned S' into k sets and assigned them each a distinct j. This will result in us having to spend time n to load the monomials in, rather than time $\lceil n/k \rceil$ as in the previous proof, but this is necessary as we have no guarantee that there is a partition of the variables such that every monomial of degree at most d is split into k monomials of degree at most $\lceil d/k \rceil$. Note that this is where our algorithm performs worse than Balanced Catalytic Product Procedure when $d = \Omega(n)$.

The analysis of Low-Degree Balanced Catalytic Product Procedure is identical to that of Balanced Catalytic Product Procedure using Low-Degree Time-Efficient Catalytic Product Procedure in place of Time-Efficient Catalytic Product Procedure, except the runtime is $2^k \cdot 2n$ rather than $2^k \cdot 2\lceil n/k \rceil$ because we do not split the variables into groups. □

5.2.4 Afterword: a simplified view of Potechin's original argument

We came to our reproof and subsequent improvement of Theorem 38 from the confluence of two circumstances. First, as discussed in the introduction, our work on Catalytic Product Lemma immediately suggested that we may find other uses of the polynomial time-space tradeoff method, particularly in the realm of catalytic computing. Second, in reading and understanding the original proof of Theorem 38 given by [Pot17], we came across a somewhat simplified presentation of the original result with slightly better parameters. We present this proof below, although it must be stressed that this proof is more or less implicit in the original construction.

Theorem 40. *For any function $f : \{0,1\}^n \to \{0,1\}$ there is an m-catalytic branching program with length $4n$ and width $4m$ that computes f, where $m = 2^{2^n - 1}$.*

Proof. We will view our program as a clean register program with $2^n - 1$ registers whose values are already filled in, plus two additional registers initialized to 0: one for output and one for work. We will rename the registers as follows: our output register will be labeled R^{out}, our one free work register will be labeled $R_{00\ldots0}$, and our other registers will each be labeled R_α for some unique $\alpha \in \{0,1\}^n \setminus \{00\ldots0\}$. Furthermore we will only have the following two types of instruction: 1) swapping the labels of two registers; 2) adding a fixed constant value to a register, or (in one step) add one register value to another.

At every step we will view our memory (not including the output register) as the truth table of a function g, where the value of $g(\alpha)$ is given by the value in R_α. Thus at the start we are storing an arbitrary function g subject to the restriction that $g(00\ldots0) = 0$.

We now describe our program.

1. we read each value $x_1 \ldots x_n$, and for each x_i we do nothing if $x_i = 0$ and otherwise we swap each label R_α with $R_{\alpha^{\oplus i}}$, where $\alpha^{\oplus i}$ is α with the ith coordinate flipped. The net result is to go from our original function g to a new function $g^{\oplus x}$ where $g^{\oplus x}(x) = 0$, because the register $R_{00\ldots0}$ has been swapped into position R_x.

2. for every α we add $f(\alpha)$ to R_α modulo 2. The net result is to go from $g^{\oplus x}$ to $g^{\oplus x} \oplus f$ where $(g^{\oplus x} \oplus f)(x) = f(x)$.

3. we undo Step 1 by running it in reverse.[4] The net result is to go from $g^{\oplus x} \oplus f$ to $(g^{\oplus x} \oplus f)^{\oplus x}$ where $(g^{\oplus x} \oplus f)^{\oplus x}(00\ldots0) = f(x)$.

4. we add the value in $R_{00\ldots0}$ to R^{out}. We now have the answer $f(x)$ stored in R^{out}, since R^{out} was 0 beforehand and $R_{00\ldots0}$ is storing $f(x)$.

5. we undo Steps 1-3 by running them in reverse. The net result is to reset our function to the original g without touching our answer in register R^{out}.

The correctness of our program is given inline. The length of the program is $4n$ because we query each x_i four times, and the width of our program is $4m$ by construction. □

[4] The order does not actually matter but is helpful for visualizing what this step does.

5.3 Better results for restricted functions

Invariably there are many cases where General Catalytic/Amortized Algorithm can be dramatically improved. The obvious and immediate example is programs which already have read-$O(1)$ branching programs without any catalytic restriction; there is nothing to be done here, as we already have our results for $m = 1$. However, even when we move to functions known to have polynomial size (but not read-$O(1)$) branching programs, it is not immediately obvious how to proceed. We give an example of such a class, called TC^1, which can be made read-$O(1)$ using m-catalytic branching programs for singly exponential m. We also show how relaxing the read-$O(1)$ constraint to read-n^ϵ for any $\epsilon > 0$ allows us to capture a class not known to have, and conjectured to not have, polynomial size branching programs of any kind.

5.3.1 Register programs over larger fields

For the results in this section, we will need to switch from working over $\mathbb{F}_2$ to more general fields. Observation 2 produces an m-catalytic branching program of width $|\mathcal{R}_n|m$. The following lemma shows how to reduce the width to $3m$ when $\mathcal{R}_n$ is a finite field, at the cost of a factor of $|\mathcal{R}_n|$ increase in m.

Lemma 41. *Let K_n be a family of finite fields. Let $f_n : \{0,1\}^n \to \{0,1\}$ be a family of functions and let P_n be a family of register programs over the fields K_n with size $s(n) + 1$ and time $t(n)$ each cleanly computing f_n. Then f_n can be computed by a family of m-catalytic branching programs of width $3m$ and length $t(n)$, where $m = |K_n|^{s(n)+1}$.*

Proof. We will proceed as in Observation 2, except that instead of initializing R^{out} to 0, we will allow it to take on any starting value (hence the factor-of-$|K_n|$ increase in m). In order to detect whether R^{out} has changed at the end of the program, we store a minimal amount of information about the starting value of R^{out}.

Let $g : K_n \to \{0,1,2\}$ be a function with the following property: for any $x \in K$, $g(x+1) \neq g(x)$. One way to construct g is as follows. Let p be the characteristic of K, i.e. $\overbrace{1+1+\cdots+1}^{p} = 0$ in p. Then K_n can be viewed as a field extension of $\mathbb{F}_p$, and so K_n can be viewed as a vector space over $\mathbb{F}_p$. Let $\{e_1, e_2, \ldots, e_k\}$ be a basis for K_n over $\mathbb{F}_p$, where $e_1 = 1 \in K_n$. For any $x \in K_n$, let $x_1 \in \mathbb{F}_p$ be its first coordinate under this basis. Then set $g(x) = 2$ if $x_1 = p - 1$ and for $x_1 \in \{0, 1, \ldots, p-2\}$ set $g(x) = x_1 \bmod 2$.

Each node (except the sink nodes) will be labelled with a tuple $(a, R^{\text{out}}, R_1, \ldots, R_{s(n)}) \in \{0,1,2\} \times \mathcal{F}_p^{s(n)+1}$, representing an assignment of values to registers and one extra value $a \in \{0,1,2\}$.

Each intermediate (non-source non-sink) layer will have $3 \cdot p^{s(n)+1}$ nodes, representing all possible tuples. The source layer will have all tuples satisfying the constraint $a = g(R^{\text{out}})$, for a total of $m = p^{s(n)+1}$ nodes. In this way, the program "remembers" $g(\tau^{\text{out}})$ where τ^{out} is the initial value of R^{out}.

We proceed as in the proof of Observation 2: for each instruction of the register program querying an input variable x_i, we include a branching program layer which reads that same input, and modifies the values of the stored registers appropriately. These layers preserve the extra value a, so that all the non-sink nodes reached by a computation have the same value of a, which is equal to $g(\tau^{\text{out}})$.

The final (sink) layer is constructed the same way, with two changes. First, nodes where $a \notin \{g(R^{\text{out}}), g(R^{\text{out}} - 1\}$ are removed. Because the register program computes $f(x) \in \{0,1\}$, the final value of R^{out} is guaranteed to be either τ^{out} or $\tau^{\text{out}} + 1$, and so those removed nodes were not reachable

anyway. Second, we relabel each node $(a, R^{\text{out}}, R_1, \ldots, R_{s(n)})$ to $(a', R^{\text{out}}, R_1, \ldots, R_{s(n)})$ where $a' = 0$ if $a = g(R^{\text{out}}$ and $a' = 1$ if $a \neq g(R^{\text{out}})$. By the construction of g, a' is guaranteed to be 0 when $R^{\text{out}} = \tau^{\text{out}}$ and 1 when $R^{\text{out}} = \tau^{\text{out}} + 1$. □

5.3.2 Constant depth circuits

Recall our definition of *Boolean circuits* from Chapter 1: these are the dag-like generalization of formulas. The first result is fairly immediate from the main technical section of [BCK+14]. This does not directly use any of our work from Tree Evaluation Algorithm or General Catalytic/Amortized Algorithm directly, but the lemmas involved are very closely related to our algorithm in Catalytic Product Lemma.

Lemma 42. *Let f be a function computed by a circuit C which has depth d, size s, and consists only of MAJ gates. Then f can be computed by an m-catalytic branching program of length $4^d \cdot n$ and width $3m$, where $m = (2s)^{2s^3}$.*

Proof. Let p_s be an arbitrary prime in the range $(s, 2s]$. Section 3.3 of [BCK+14] gives a register program P simulating the computation of C, which reads leaves of C at most $4^d \cdot n$ times in total and uses $s \cdot 2p_s \cdot s/2 \leq 2s^3$ registers over $\mathbb{F}_{p_s}$. Our result follows by Lemma 41. □

Focusing on the case of TC^0, defined as the set of all circuits of depth $O(1)$ and size $\text{poly}(n)$ consisting only of MAJ gates, we get linear amortized size with m only singly exponential in n. While such circuits are known to have poly-size branching programs even for $m = 1$—following directly from the fact that $\mathsf{TC}^0 \subseteq \mathsf{L}$—no results for linear amortized size were previously known. For example, even applying the second part of General Catalytic/Amortized Algorithm to a single MAJ gate would result in m being almost maximally large, as MAJ has degree $n/2$ over $\mathbb{F}_2$.

Corollary 43. *Any function $f \in \mathsf{TC}^0$ can be computed by an m-catalytic branching program of amortized size $O(n)$, where $m = 2^{\text{poly}(n)}$.*

5.3.3 Arithmetic circuits

Our second result will take us out of the realm of Boolean functions and into the world of polynomials. Fix some field $\mathbb{F}$ for the rest of this section. An *arithmetic circuit* is similar to a Boolean circuit as defined before, but now the leaves will carry values in $\mathbb{F}$ and the gates will compute the functions $+$ and $\times$ over $\mathbb{F}$. Clearly such a circuit computes a polynomial over $\mathbb{F}$. Note that a branching program computing the output of an arbitrary such circuit would need to have $|\mathbb{F}| \cdot m$ output nodes, which would immediately rule out any linear amortized size. To get around this, we will assume that whenever all inputs comes from $\{0, 1\}$ (over $\mathbb{F}$), the output of our circuit will as well.[5]

We will focus on arithmetic circuits of logarithmic depth. Our techniques will be similar to the previous result, meaning that we will incur a cost that is exponential in the depth, which will unfortunately take us out of the regime of linear amortized size. Nevertheless, we will leverage the structure of VP along with the second part of General Catalytic/Amortized Algorithm to substantially mitigate this loss and still achieve a highly improved value of m.

[5]This corresponds to the *Boolean part* of an arithmetic circuit class, which gives a natural way of computing Boolean functions with arithmetic circuits, and by extension a natural way to fit classes of arithmetic circuits into the broader landscape of complexity theory.

Lemma 44. *Let $\mathbb{F}$ be any finite[6] field. Let f be a polynomial over $\mathbb{F}$ the output of which is always 0 or 1, and let C be an arithmetic circuit over $\mathbb{F}$ computing f which has depth d, size s, and consists of $+$ gates of unbounded fan-in and $\times$ gates of fan-in 2. Then for any $k \leq d$, f can be computed by an m-catalytic branching program of length $4^{\lceil d/k \rceil} \cdot n$ and width $3m$, where $m = |\mathbb{F}|^{\binom{s}{\leq 2^k} \cdot s}$.*

Proof. We will describe a register program which uses $\binom{s}{\leq 2^k} \cdot s$ registers which each contain an element of $\mathbb{F}$. Each register will be labeled with a unique gate g from the circuit C—in fact we will only need registers for some of the gates, but we will potentially overcount to keep things simpler—as well as a subset $S \subseteq [s]$ of size at most 2^k, and we write the corresponding register as $R_{g,S}$. We have $\binom{s}{\leq 2^k} \cdot s$ registers, so we get $m = |\mathbb{F}|^{\binom{s}{\leq 2^k} \cdot s}$.

Our goal will be cleanly compute, for $L = 1 \ldots \lfloor d/k \rfloor, d/k$, the value of every gate g at level Lk of the circuit into $R_{g,\emptyset}$, inductively using the fact that we can compute every gate appearing at level $(L-1)k$. To start this procedure off, we observe that for $L = 0$, we can compute all leaf nodes using n total input queries, namely by handling all leaves labeled x_i at the same time with one query to x_i. Now we proceed inductively using the following lemma:

Lemma 45. *Let $L \in [\lfloor d/k \rfloor]$, and let P_{L-1} be a register program which cleanly computes, for all g at level $(L-1)k$, the value at g into $R_{g,\emptyset}$. Then there exists a register program P_L making 4 calls to P_{L-1} which cleanly computes, for all g at level Lk, the value at g into $R_{g,\emptyset}$.*

Proof. Consider a single gate g at layer Lk, let $g_1 \ldots g_t$ be the gates at layer $(L-1)k$, and let

$$p_g = \sum_S c_S \prod_{i \in S} g_i$$

be the polynomial over $\mathbb{F}$ computed at gate g with inputs $g_1 \ldots g_t$. By induction every layer $\ell \in [(L-1)k..Lk]$ has degree at most $2^{\ell-(L-1)k}$ in $g_1 \ldots g_t$, and thus g has degree at most 2^k.

We now follow our proof of General Catalytic/Amortized Algorithm exactly, except for two changes. First, ToggleInput will be replaced with P_{L-1}. Second, some calls to P_{L-1} or ToggleUsefulMonomials$_g$ will be replaced with their *inverses* P_{L-1}^{-1} or ToggleUsefulMonomials$_g$. Note that any clean register program P has an inverse P^{-1} of the same length such that PP^{-1} has no effect; see for example [BCK+14]. In the previous sections this was not necessary because the field had characteristic 2, and so each procedure was its own inverse.

Let τ_i be the initial value in g_i, let x_i be the value computed into g_i, define $y_i = \tau_i + x_i$, and let

$$q_g(\vec{y}, \vec{\tau}) = \sum_{S,S'} c_{S,S'} \prod_{i \in S} \tau_i \prod_{i \in S'} y_i$$

be equal to $p(\vec{y} - \vec{\tau})$. Let $\mathcal{M}_g$ be the set of all non-empty monomials for which $c_{S,S'} \neq 0$ for some S; note that $|\mathcal{M}_g| \leq s^{2^k}$ by our degree upper bound. For each $S' \in \mathcal{M}_g$, register $R_{g,S'}$ will correspond to the sum of monomials with coefficients $c_{S,S'}$. Since $\emptyset \notin \mathcal{M}_g$, we can repurpose $R_{g,\emptyset}$ as our output register. Together, the following two procedures together compute q_g as before:

```
1: procedure ToggleUsefulMonomials_g
2:     P_{L-1}
3:     for S ∈ M_g do
```

[6] If $\mathbb{F}$ is not finite, the proof produces a branching program of infinite width, although later when we apply this lemma we will use reducibility results to get around this for $\mathbb{Z}$ and $\mathbb{Q}$.

4: $R_{g,S} \leftarrow R_{g,S} + \prod_{i \in S} R_{g_i,\emptyset}$
5: P_{L-1}^{-1}

1: **procedure** FINALCOMPUTE$_g$
2: TOGGLEUSEFULMONOMIALS$_g$
3: $R_{g,\emptyset} \leftarrow R_{g,\emptyset} + \sum_{S \subseteq [t]} c_{S,\emptyset} + \sum_{S' \in \mathcal{M}_g} c_{S,S'} \left(\prod_{i \in S} R_{g_i,\emptyset}\right) R_{g,S'}$
4: TOGGLEUSEFULMONOMIALS$_g^{-1}$
5: $R_{g,\emptyset} \leftarrow R_{g,\emptyset} - \sum_{S \subseteq [t]} c_{S,\emptyset} - \sum_{S' \in \mathcal{M}_g} c_{S,S'} \left(\prod_{i \in S} R_{g_i,\emptyset}\right) R_{g,S'}$

The analysis is identical to General Catalytic/Amortized Algorithm and so we leave it to the reader. To compute all g at level Lk, we replace every basic instruction in both TOGGLEUSEFULMONOMIALS$_g$ and FINALCOMPUTE$_g$ with the same instruction looped over all g at level Lk; this does not add any recursive calls to either program, and by the correctness of the original algorithm for one g this new algorithm correctly computes all g. Thus level Lk requires four calls to level $(L-1)k$ as claimed. □

If k divides d, then $P := P_{d/k}$ computes C correctly. Otherwise the same argument gives a program P making four calls to $P_{\lfloor d/k \rfloor}$ computing C correctly. This gives a recursion of total height $\lceil d/k \rceil$ where $h(0) = n$ and $h(L) = 4h(L-1)$, which gives us a program of length $4^{\lceil d/k \rceil} \cdot n$ as claimed. Thus our result follows by Lemma 41. □

One of the two most interesting classes of arithmetic circuits is VP , which corresponds to arithmetic circuits of depth $O(\log n)$ and size $\mathrm{poly}(n)$ consisting of unbounded fan-in $+$ gates and fan-in 2 $\times$ gates. As before let p_s be a prime in the range $(s, 2s]$. Using the fact that VP over $\mathbb{F}_{p_s}$, $\mathbb{Z}$, and $\mathbb{Q}$ are all logspace-reducible[7] to one another [AGM17][8], and fixing $k = c/\epsilon$ where $d = c \log_2 n$, we obtain the following quasilinear result for VP.

Corollary 46. *Let $\mathbb{F} \in \{\mathbb{F}_{p \in [\mathrm{poly}\, n]}, \mathbb{Z}, \mathbb{Q}\}$, and let $\epsilon > 0$. Any polynomial $f \in \mathsf{VP}$ can be computed by an m-catalytic branching program of amortized size $O(n^{1+\epsilon})$, where $m = 2^{\mathrm{poly}_\epsilon n}$.*

5.4 Length for permutation branching programs

In the previous section we took the length $4n$ branching programs of [Pot17, RZ21] as a starting point to analyze whether m could be significantly reduced while still maintaining a linear amortized size. In this section we investigate the opposite question: namely, is $4n$ optimal? Recall that if we do not restrict the amortized size of our program, then every function has a branching program of length n even for $m = 1$. We will not only consider branching programs of linear amortized size, but focus on the stricter model of *permutation branching programs.* Once again we will focus on the case when $\mathcal{R} = \mathbb{F}_2$ for this entire section.

[7] Logspace-reducibility does not make much sense for arithmetic classes, so we again emphasize that we are focusing on the *Boolean part* of these respective classes. It also may seem odd to talk about reducibility between syntactic classes, but notice that the range available to p_s depends on the input size, which may significantly change as a result of the reduction, a fact that is actually necessary to reduce $\mathbb{Z}$ and $\mathbb{Q}$ to $\mathbb{F}_{p_s}$. See [AGM17] for a more in-depth discussion of this result.

[8] This may strike the reader as odd, since $\mathbb{F}_{p_s}$ is finite and the other ones are not. The insight is that while e.g. $\mathbb{Z}$ may be infinite, the elements of $\mathbb{Z}$ that appear in our circuit are part of the input and thus are bounded in terms of n; similarly the circuit itself cannot make these elements grow too large to handle with a sufficiently large finite field.

5.4.1 Permutation branching programs

To start, let us define the central focus of this section, that of permutation branching programs. We start with an informal definition. Consider a (2-wise) layered branching program G which has length ℓ and width m. Then G is a permutation branching program if every layer has *exactly* m nodes, and at any layer j, if we fix the value of the variable x_{i_j} to $\alpha_{i_j} \in \{0,1\}$, every node in layer j goes to a unique node in layer $j+1$. Note that putting these two conditions together, we see that the edges going from j to $j+1$ labeled with $\alpha_{i_j} \in \{0,1\}$ correspond to a permutation of $[m]$, hence the name. We apply these conditions to the start and end layers as well, so unlike in a typical branching program we do not have one source and two sinks, but rather m sources and m sinks; the value computed by G will not directly correspond to the sink we reach from a source, but rather by the *permutation* that we get by composing all the layers together for the given input, guaranteed by the fact that every intermediate layer is a permutation itself.

Now we present the formal definition. Our notation will be non-standard and different from previous branching program models. We also add some technical quirks to this definition, which we later justify.

Definition 22. Let $n, m := m(n), \ell := \ell(m,n) \in \mathbb{N}$. A *permutation branching program* is a sequence $P = \pi_1 \dots \pi_\ell$, where each π_j is a pair $\langle i_j, \sigma_j \rangle$ where $i_j \in [0..n]$ and σ_j is a permutation of $[m]$. We refer to each π_j as an *instruction* of P. The *width* of P is m and the *length* of P is ℓ.

Let $o := o(k,n)$, and let $f : [k]^n \to [k]^o$ be a function. For any $\alpha \in \{0,1\}^n$ we define $P(\alpha)$ as follows: fix $\sigma = id$, and for every $j = 1 \dots \ell$, we set σ to $\sigma\sigma_j$[9] if $\pi_j = \langle 0, \sigma_j \rangle$ or $\pi_j = \langle i_j, \sigma_j \rangle$ where $\alpha_{i_j} = 1$, and leave σ unchanged otherwise (that is, if $\pi_j = \langle i_j, \sigma_j \rangle$ where $\alpha_{i_j} = 0$). Our output is the final value of σ. We say that P computes f if there exists a permutation $\sigma^* \neq \alpha$ such that $P(\alpha) = id$ if $f(\alpha) = 0$ and $P(\alpha) = \sigma^*$ if $f(\alpha) = 1$.

This model is very different from all our previous branching programs, and so some discussion is warranted. Let us start by seeing how it relates to our other models. Clearly we can extract a branching program from a permutation branching program by choosing any source node u and removing all nodes not reachable from u.

Furthermore, if we focus on the case when σ^* is a set of disjoint swaps, this begins to look like a restriction of our m-catalytic branching program model, where now each source of G is labeled (u,b) and on input x we should reach the sink labeled $(u, b \oplus f(x))$, with the restriction that G is layered and with fixed width. An astute reader may notice here that these conditions are in fact already satisfied when we convert a clean register program to an m-catalytic branching program; in fact, permutation branching programs were the original context for clean register programs [Bar89, BC92].

Observation 5. *Let P be a clean register program over $\mathcal{R}$ with size s and time t such that for every input x, $P(x) \in \{0,1\}$. Then there exists a permutation branching program G of width $|\mathcal{R}|^s$ and length t such that $G(x) = P(x)$ for every x.*

This gives us some initial justification for restricting our attention to permutation branching programs; all our algorithms in General Catalytic/Amortized Algorithm can be converted to permutation branching programs. In fact, even the original results of [Pot17, RZ21] are amenable to this change:

Theorem 47. *Every function f can be computed by a read-4 permutation branching program of width 2^{2^n} (or $2^{\binom{n}{\leq d}}$, where d is the $\mathbb{F}_2$-degree of p_f).*

[9] We write $\sigma_1\sigma_2$ as a shorthand for $\sigma_2 \circ \sigma_1$. As a branching program model, these permutations are applied left to right instead of the more typical right-to-left form appearing in group theory.

Given the strength of these results, only a factor of 4 away from the best conceivable amortized size, there is no reason *a priori* to believe that permutation branching programs are too weak to consider even better upper bounds. Indeed we will show that improvements are possible.

This leads to our other reason for focusing on permutation branching programs: lower bounds against general branching programs are notoriously difficult. Besides the basic counting argument, the best known branching program lower bounds for an explicit function are not even quadratic, using techniques known to go no further [Neč66]. Considering the amortized branching program size needed to compute any function f is always at most the basic branching program size, and considering the upper bound of $4n$ given by [Pot17], proving lower bounds for concrete functions seems exceedingly difficult. Furthermore, even if we were to seek refuge in focusing on non-constructive lower bounds, the basic counting argument fails to prove *any* non-trivial lower bounds in the case of $m \geq 2^n/n$.

Now we discuss the definition of permutation branching programs specifically as presented in Definition 22, which has some non-standard elements; in the process we will establish some basic tools for our results. We make four points. First, the restriction that $f(0 \dots 0) = 0$ will be a convenience; we can always compute $\neg f$ instead if this condition does not hold, or change our definition such that $P(\alpha) = id$ if $f(\alpha) = 1$ and vice versa.[10]

Second, in a layer $\langle i, \sigma_j \rangle$ reading variable x_i, we only fix a permutation in the case that x_i is set to 1. This is without loss of generality, as adding a layer of the form $\langle 0, \sigma'_j \rangle$ before an instruction can be thought of as choosing a permutation in the case that x_i is set to 0 (while the permutation for $x_i = 1$ can be adjusted accordingly).

Before going on to our third observation, we state and prove four simple lemmas which will allow us to conveniently restructure our programs P.

Lemma 48. *Let P be a permutation branching program computing f and let j be such that $i_j = i_{j+1}$. Then the program P' resulting from replacing π_j, π_{j+1} with $\pi'_j = \langle i_j, \sigma_j \sigma_{j+1} \rangle$ is also a valid program for computing f.*

Proof. In both P and P', the permutations σ_j and σ_{j+1} are both applied when $i_j = 1$ and neither are applied when $i_j = 0$. □

Lemma 49. *Let P be a permutation branching program computing f and let j be such that $\sigma_j \sigma_{j+1} = \sigma_{j+1} \sigma_j$. Then the program P' resulting from switching the order of π_j and π_{j+1} is also a valid program for computing f.*

Proof. Consider any assignment α to x. In the case that either α_{i_j} or $\alpha_{i_{j+1}}$ is set to 0, these programs compute identical permutations as either σ_j or σ_{j+1} will not be applied. If both are set to 1, then

$$P'(\alpha) = \Sigma_1 \sigma_{j+1} \sigma_j \Sigma_2 = \Sigma_1 \sigma_j \sigma_{j+1} \Sigma_2 = P(\alpha)$$

where Σ_1, Σ_2 are the permutations corresponding to the rest of the instructions on input α. □

Lemma 50. *Let $P = \pi_1 \dots \pi_s$ be a permutation branching program computing f, let $\pi_j = \langle i_j, \sigma_j \rangle$ for all j, and let $j^* \in [\ell]$ be such that $i_{j^*} = 0$. Then there exists a permutation branching program $P' = \pi'_1 \dots \pi'_{j^*-1} \pi'_{j^*+1} \dots \pi'_\ell \pi_{j^*}$ computing f, where $\pi'_j = \langle i_j, \sigma'_j \rangle$ for some permutation σ'_j.*

[10]We also note that if P computes $\neg f$, we can compute f by appending the instruction $\langle 0, (\sigma^*)^{-1} \rangle$ to P. We avoid taking this route because a later observation will allow us to remove these fixed layers, but only when $f(\alpha) = 0$, which would cause our logic to become circular.

Proof. For $j < j^*$ define $\sigma'_j = \sigma_j$, and for $j > j^*$ define $\sigma'_j = \sigma_{j^*}\sigma_j\sigma_{j^*}^{-1}$. Clearly because $\sigma_{j^*}^{-1}$ and σ_{j^*} cancel out for every adjacent pair of permutations σ'_j, $P'(\alpha)$ contains exactly the same permutations as $P(\alpha)$ in the same order regardless of the assignment α.[11] □

Our last observation is that the layers of the form $\langle 0, \sigma_j \rangle$ are only a convenience and are not necessary to our definition. Let P be our program for f and let $\sigma_1 \dots \sigma_k$ be the permutations corresponding to instructions π_j where $i_j = 0$. By our restriction that $f(0\dots0) = 0$ we get $\sigma_1 \dots \sigma_k = P(0\dots0) = id$, and by Lemma 50 we can move the instructions π_j with $i_j = 0$ to the end of the program, in order, at which point we can simply remove them all using Lemma 48 as they compose to the identity for any input α.

We can also generalize Lemma 50 for *restrictions* of the function f, meaning when we fix the values of some variables and consider the function on the remaining variables. This is simply the observation that fixing variable x_i turns all instructions of the form $\langle i, \sigma_j \rangle$ into fixed layers $\langle 0, \sigma_j \rangle$.

Corollary 51. *Let $\rho \in \{0, 1, *\}^n$ and let f_ρ be the function f with x_i fixed to $\rho(i)$ wherever $\rho(i) \neq *$. Let $P = \pi_1 \dots \pi_\ell$ be a permutation branching program computing f, let $\pi_j = \langle i_j, \sigma_j \rangle$ for all j, and let $j^* \in [s]$ be such that $i_{j^*} = 0$ or $\rho(i_{j^*}) \neq *$. Then there exists a permutation branching program $P' = \pi'_1 \dots \pi'_{j^*-1}\pi'_{j^*+1} \dots \pi'_\ell \pi_{j^*}$ computing f_ρ, where $\pi'_j = \langle i'_j, \sigma'_j \rangle$ for some permutation σ'_j and $i'_j = i_j$ iff $\rho(i_j) = *$ and 0 otherwise.*

Proof. Let program P'' be the result of replacing i_j with 0 in each instruction $\pi_j \in P$ such that $\rho(i_j) \neq *$. Clearly this program computes f_ρ, and so applying Lemma 50 to P'' completes the proof. □

Assuming that $f_\rho(0\dots0) = 0$, by our previous observation this allows us to remove all layers that read variables fixed by ρ. We also note that the other three lemmas hold for f_ρ with no changes.

Lastly, another option for defining permutation branching programs is to simply require that $P(\alpha)$ is not id whenever $f(\alpha) = 1$, and not restrict ourselves to a specific $\sigma^* \neq id$. For example, this stronger model was used in [Bar89] to capture NC^1. Our lower bounds will actually hold against this stronger model, while our upper bounds will hold in the weaker model presented in Definition 22, and so our choice is somewhat arbitrary. We chose the definition as presented because it is much closer to the definition of m-catalytic branching programs which we have used in the rest of this chapter, and in fact all our upper bounds come directly from these prior results.

5.4.2 Upper bounds

For our main upper bound, we modify our algorithm recreating the result of [Pot17] (and analogously [RZ21]) to have length $4n - 4$. In particular, our program will read all but two variables four times, while the last two variables will be read twice.

Theorem 52. *For every function f, there is a read-4 permutation branching program of width 2^{2^n-1} and length $4n - 4$ computing f.*

Proof. First, we make an easy change to Theorem 38 which allows us to achieve $4n - 3$. Observe that in ToggleInput the order in which we add the inputs is irrelevant, and so consider ToggleMonomials where we reverse the order of toggling on Line 5. Then notice that the last query on Line 2 and the first query on Line 5 are both made to x_n, and so we can merge these two layers along with our entire for loop

[11] This argument actually allows us to move π_{j^*} to any spot in the program we want, but we are content with just moving them to the end, for reasons which will become immediately clear.

(which reads no variables) into a single layer querying x_n. Moving to Time-Efficient Catalytic Product Procedure, this means we only query x_n twice, and furthermore the last query on Line 1 and the first query on Line 3 are both made to x_1, and so again by merging these two queries along with Line 2 we only query x_1 three times.

Now we will change our program so that x_1 is only read twice. Consider two new functions obtained by fixing the value of x_1, namely $f^0 = f(0, x_2 \dots x_n)$ and $f^1 = f(1, x_2 \dots x_n)$. Recall that we used the following polynomial to compute f, where $y_i = \tau_i^{\text{in}} + x_i$:

$$q_f = \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{i \in S'} y_i \right)$$

If we choose $b \in \{0, 1\}$ and fix $\tau_1^{\text{in}} = 0$ and $y_1 = x_1 = b$, we get the following, which can be used to compute (since it is equal to) f^b:

$$q_{f^b} = \sum_{S,S' \subseteq [2..n]} (c_{S,S'} + b \cdot c_{S,S' \cup \{1\}}) (\prod_{i \in S} \tau_i^{\text{in}}) (\prod_{i \in S'} y_i)$$

We will use Time-Efficient Catalytic Product Procedure to compute q_{f^b}, where $b = x_1$, by removing all reference to x_1 from TOGGLEINPUT and TOGGLEMONOMIALS, and querying x_1 whenever we execute Lines 2 or 4 to determine whether to compute q_{f^0} or q_{f^1} in place of q_f. More specifically, TOGGLEINPUT will now only loop over $i = 2 \dots n$, while TOGGLEMONOMIALS will now only loop over $S \subseteq [2..n]$. Finally in Time-Efficient Catalytic Product Procedure we change Lines 2 and 4 to

$$R^{\text{out}} \leftarrow R^{\text{out}} \pm \sum_{S,S' \subseteq [2..n]} (c_{S,S'} + x_1 \cdot c_{S,S' \cup \{1\}}) (\prod_{i \in S} R_i^{\text{in}}) R_{S'}$$

where Line 2 uses + and Line 4 uses −. Note that to execute these lines correctly, we will query x_1 and perform the corresponding instruction; thus we no longer ignore these two lines in calculating our program length.

By our earlier definition of q_{f^b}, this exactly computes q_{f^b} for $x_1 = b$ as claimed. As above we will reverse the order of the queries in TOGGLEINPUT the second time it is called in TOGGLEMONOMIALS, which allows us to read x_n only once per execution for a total of two reads. x_1 will be queried in Lines 2 and 4, and all other variables will be queried four times. □

Note 5.4.1. This strategy also allows us to save an exponential number of registers, as we only need a register R_S for each $S \subseteq [2..n]$. While it may be tempting to extend this trick to more variables, say by fixing the values of both x_1 and x_2, the fact that Lines 2 and 4 depend on the value(s) of the fixed variable(s) means that we will have to store at least one of these values in a non-catalytic register, which will add to our width and take us out of the realm of permutation programs. If we go back to m-catalytic branching programs, this gives us another way to save over [Pot17, RZ21], but with worse parameters; for any $k \in [n]$ by fixing k values we can get a program of length $2(k + 1) + 4(n - k - 1)$ and amortized size $2^k \cdot O(n)$ as before, but for $m = 2^{2^{n-k}-1}$ instead of $2^{2^{n/k}-1}$.

There are two known cases in which we can achieve better than read-4 for AND: $n = 2, 3$. The $n = 2$ case is unsurprising, as our argument allows for two variables to be read twice; it has appeared in many previous works (see c.f. [Bar89]). The case of $n = 3$ is more surprising, and suggests that read-3 may be

achievable in general. Note that because of the small values of n involved, neither result gives a program smaller than length $4n-4$.

Lemma 53. *There is a read-2 permutation branching program of width* 3 *computing* $AND(x_1, x_2)$.

Proof. Choose any two permutations σ_1 and σ_2 such that $\sigma_1\sigma_2\sigma_1^{-1}\sigma_2^{-1} \neq id$; for example we can choose $\sigma_1 = (12)$ and $\sigma_2 = (23)$. Then consider the following program:

$$\langle 1, \sigma_1\rangle, \langle 2, \sigma_2\rangle, \langle 1, \sigma_1^{-1}\rangle, \langle 2, \sigma_2^{-1}\rangle$$

By definition of σ_1 and σ_2, $P(1,1) \neq id$, and if either variable is set to 0 then the only permutations left are σ_j and σ_j^{-1} for some $j \in \{1,2\}$, and the composition of these permutations is id. □

Lemma 54. *There is a read-3 permutation branching program of width* 3 *computing* $AND(x_1, x_2, x_3)$.

Proof. We state the program and leave the reader to check correctness.[12] Our permutations σ_j are given in cycle notation.

$$\langle 1,(23)\rangle, \langle 2,(12)\rangle, \langle 3,(123)\rangle,$$
$$\langle 1,(12)\rangle, \langle 2,(13)\rangle, \langle 3,(23)\rangle,$$
$$\langle 1,(132)\rangle, \langle 2,(132)\rangle, \langle 3,(13)\rangle$$

□

Note 5.4.2. Recall that we could consider a stronger definition of permutation branching programs where we only require that $P(\alpha) \neq id$ whenever $f(\alpha) = 1$. In this case, it is not hard to show that for any n, if we can compute $\text{AND}(x_1 \dots x_n)$ in length ℓ, we can also compute any function $f(x_1 \dots x_n)$ in length ℓ by "tensoring" the permutations in P with themselves for each $\alpha \in f^{-1}(1)$.

5.4.3 Lower bounds

In this section we show that if one tries to get a program of length less than $3n$, one cannot beat Theorem 52.

Theorem 55. *Any permutation branching program computing* $AND(x_1, \dots, x_n)$ *which reads some variable at most twice must have length at least* $4n-4$.

Proof. Let $P = \pi_1\pi_2 \dots \pi_s$ be any program computing $\text{AND}(x_1, \dots, x_n)$. We will write σ_i^j to refer to the permutation in the jth instruction in P that reads variable x_i; in other words, the instructions corresponding to x_i will be $\langle i, \sigma_i^1\rangle \dots \langle i, \sigma_i^k\rangle$ for some k. Since we are focusing on AND, which is symmetric, we will assume without loss of generality that x_i is read for the first time before any $x_{i'}$ is read for $i' > i$. As usual let $\sigma^* \neq id$ refer to the permutation resulting from P when all variables are set to 1.

Claim 56. *Any program* P *computing AND of more than one variable must read every variable at least twice*

[12] It should be noted that we found this program through an automated search, and it would be interesting to see what nice properties of the program—of which there are many candidates—could be useful in searching for read-3 programs for higher n.

Proof. Assume that some variable x_i is read only once in P. Then setting $x_{i'} = 0$ for all $i' \neq i$, we get $\sigma_i^1 = P(0 \dots 0, 1, 0, \dots, 0) = id$. Therefore P acts identically whether x_i is 0 or 1, which is a contradiction because AND depends on x_1. □

Now consider when some variable x_i is read exactly twice. Let $j_1 < j_2$ be the locations of the two instructions containing i, i.e. $\pi_{j_1} = \langle i, \sigma_{j_1} \rangle$ and $\pi_{j_2} = \langle i, \sigma_{j_2} \rangle$, and let $\Pi_1 = \pi_{j_1+1} \dots \pi_{j_2-1}$. The following is our main claim.

Claim 57. *Every variable besides x_i is read at least once in Π_1, and there is at most one such variable $x_{i'}$ which is not read at least twice in Π_1.*

Proof. First, assume for contradiction that there exists $i' \neq i$ such that $x_{i'}$ does not appear in Π_1. Then if we fix $x_{i''} = 1$ for all $i'' \neq i, i'$, we can apply Lemma 51 to move all instructions querying variables other than x_i and $x_{i'}$ to the end of the program, and then apply Lemma 48 to get an equivalent program of the following form which computes AND$(x_i, x_{i'})$:

$$\langle i, {\sigma'}_i^1 \rangle, \langle i, {\sigma'}_i^2 \rangle, \langle i', \sigma' \rangle, \langle 0, \sigma_* \rangle$$

where σ' and σ_* are some permutations (σ_* comes from all the instructions π_{j^*} produced by Lemma 51). This contradicts Claim 56 as i' is only read once.

Next, assume for contradiction that there exist $i' \neq i'' \neq i$ such that i' and i'' appear only once each in Π_1. If we fix $x_{i'''} = 0$ for all $i''' \neq i, i', i''$, applying Lemmas 51 and 48, without loss of generality the following program computes AND$(x_i, x_{i'})$:

$$\langle i, {\sigma'}_i^1 \rangle, \langle i', \sigma_{i'} \rangle, \langle i'', \sigma_{i''} \rangle, \langle i, {\sigma'}_i^2 \rangle, \Sigma$$

where $\sigma_{i'}$ and $\sigma_{i''}$ are some permutations and Σ is a set of instructions reading only the variables $x_{i'}$ and $x_{i''}$.

Define $\Sigma_{i'}$ to be the result of fixing $x_{i''} = 0$ in Σ, and define $\Sigma_{i''}$ to be the result of fixing $x_{i'} = 0$ in Σ. Note that if only one remaining variable is set to 1 then the program must output 0, so ${\sigma'}_i^2 = ({\sigma'}_i^1)^{-1}$, $\Sigma_{i'} = (\sigma_{i'})^{-1}$, and $\Sigma_{i''} = (\sigma_{i''})^{-1}$. Thus if we set $x_{i''} = 0$ our resulting program is

$$\langle i, {\sigma'}_i^1 \rangle, \langle i', \sigma_{i'} \rangle, \langle i, ({\sigma'}_i^1)^{-1} \rangle, \langle i', (\sigma_{i'}^1)^{-1} \rangle$$

and so setting $x_i = x_{i'} = 1$ we get that ${\sigma'}_i^1 \sigma_{i'} ({\sigma'}_i^1)^{-1} (\sigma_{i'})^{-1} = id$. Therefore by Lemma 49 we can swap the order of these two instructions and get an equivalent program for AND$(x_i, x_{i'}, x_{i''})$ of the form

$$\langle i', \sigma_{i'} \rangle, \langle i, {\sigma'}_i^1 \rangle, \langle i'', \sigma_{i''} \rangle, \langle i, ({\sigma'}_i^1)^{-1} \rangle, \Sigma$$

and similarly by fixing $x_{i'} = 0$ we have ${\sigma'}_i^1 \sigma_{i''} ({\sigma'}_i^1)^{-1} (\sigma_{i''})^{-1} = id$, which by Lemma 49 leaves us with the program

$$\langle i', \sigma_{i'} \rangle, \langle i'', \sigma_{i''} \rangle, \langle i, {\sigma'}_i^1 \rangle, \langle i, ({\sigma'}_i^1)^{-1} \rangle, \Sigma$$

and applying Lemma 48 on our two layers reading i gives us a program which never reads x_i, which is a contradiction. □

Define $\Pi_2 = \pi_{j_2+1} \dots \pi_s, \pi_1 \dots \pi_{j_1-1}$; to prove Claim 57 holds for Π_2 as well we will need one more observation. This is similar to our tools at the start of the section, but specifically for AND.[13]

Claim 58. *Let P be a permutation branching program computing AND whose first instruction is π_1. Then the program P' resulting from removing π_1 from the beginning of P and adding it to the end of P is also a valid program for computing f.*

Proof. Let $\pi_1 = \langle i_1, \sigma_1 \rangle$ for some i_1, and recall that $\sigma^* = P(1 \dots 1) \neq id$. Note that P' is equivalent to the program $\langle i_1, \sigma_1^{-1} \rangle P \langle i_1, \sigma_1 \rangle$. Thus $P'(1, 1, \dots, 1) = \sigma_1^{-1} \sigma^* \sigma_1 \neq id$, and for all $\alpha \neq 1, 1, \dots 1$, $P'(\alpha)$ will either be $\sigma_1^{-1}(id)\sigma_1 = id$ or id. □

Now we can prove the same statement for Π_2.

Claim 59. *Every variable besides x_i is read at least once in Π_2, and there is at most one such variable $x_{i'}$ which is not read at least twice in Π_2.*

Proof. Applying Lemma 58 repeatedly to P, we can get an equivalent program $P' = \pi_2 \Pi_2 \pi_1 \Pi_1$, and apply Claim 57 to P'. □

By a simple analysis of Claims 57 and 59, one of two cases must occur for the variables besides x_i: either 1) one variable $x_{i'}$ is read at least twice and all other variables are read at least four times or 2) two variables $x_{i'}, x_{i''}$ are read at least three times and all other variables are read at least four times. This is because a read-2 variable can only be read at most once in each of Π_1 and Π_2, while a read-3 variable will be read at most once in either Π_1 or Π_2. In either of these cases, our branching program must have length at least $4(n-2) + 2 \cdot 2 = 4(n-3) + 2 \cdot 1 + 3 \cdot 2 = 4n - 4$. □

Note 5.4.3. Besides the fact that our lower bound in Theorem 55 quantitatively matches up with our upper bound in Theorem 52 in the case of reading any variable twice, qualitatively both cases in the analysis at the end of our lower bound proof match with a possible construction given by our upper bound. After fixing a read-2 variable to condition f on, we get two halves to our top level program, and in each of them we will merge two reads of a variable in order to save a further layer. The choice of which variable to merge the reads of is arbitrary, so consider our choices for the first and second half. If we choose the same variable for both halves, it will be read twice and all other variables will be read four times. If we choose different variables in each half, both will be read three times and the rest will be read four times.

Further reading

- *A Note on Amortized Branching Program Complexity* [Pot17]. This was the first paper to study upper bounds for general m-catalytic branching programs, as well as drawing the connection between them and amortized analysis. Their proof is different from the one we presented, but no more difficult to understand.

[13] If we used our stronger notion of permutation branching program from the upper bounds section, it would apply to any f more generally, but this is unnecessary for our proof.

- *Amortized Circuit Complexity, Formal Complexity Measures, and Catalytic Algorithms* [RZ21]. Besides giving the first improvement to the results of [Pot17], this paper also sought to standardize the study of formal complexity measures like the ones in this thesis, and showed that linear upper bounds hold for the amortized complexity of more than just branching programs.

Chapter 6

Conclusion

This brings the present work to a close. Throughout the proofs we have tried to bring attention to various bottlenecks, snags, and angles for future attack, and here we will state these and other open problems in concrete terms, as well as a discussion about the possible future of composition as a general approach within theoretical computer science.

6.1 Goals for the potential contrarian

We begin in an unorthodox way, by putting the call out to those who are skeptical of one or both of the central goals of this thesis, namely proving composition lower bounds for depth and proving composition upper bounds for space. There is indeed reason to be on the fence, but we hope that the evidence here has tipped the reader's mental scales somewhat.

Nevertheless, before ending on a positive note and recapitulating future directions which may interest the believers, here we will give two potential research projects for the skeptics: one for the NC^1 optimists and one for the L pessimists. In the spirit of positivity, however, we will attempt do so without directly undermining either of our central goals; both approaches will rely on composition, but neither will come to bear on questions of the complexity of $\mathsf{TreeEval}$ itself.

Towards $\mathsf{NC}^1 = \mathsf{L}$

As we remarked on when introducing composition-based lower bounds in Chapter 1, it is widely conjectured that $\mathsf{NC}^1 \neq \mathsf{L}$, and in suggesting opposite answers to the conjectures that $\mathsf{TreeEval} \notin \mathsf{NC}^1$ and $\mathsf{TreeEval} \notin \mathsf{L}$, we may have contributed to that perception. Of course, since many people believe both of these conjectures, our suggestion that NC^1 and L are different comes from an argument that others would typically not make.

However, there may be reason to believe that NC^1 actually *can* solved composed function, whether or not this would be in a manner powerful enough to compute $\mathsf{TreeEval}$. Ironically, this comes from the very origins of all our upper bounds in Part II, a classic result known as Barrington's Theorem [Bar89]. Put simply, we *do* have a characterization of NC^1 in terms of branching programs.

Theorem 60 (Barrington's Theorem [Bar89])**.** NC^1 *is exactly equal to the class of problems solvable by permutation branching programs of polynomial length and width 5.*

The non-trivial direction[1] of Barrington's Theorem is showing that such branching programs capture NC^1, and so the proof itself speaks to the weakness of NC^1 rather than its strength. Nevertheless, now that we have this characterization the question becomes whether or not we can use tricks such as our time-space tradeoff to capture L. In other words, whatever one's beliefs about $\mathsf{TreeEval}$'s inclusion or non-inclusion in L, it seems natural for us to test out our newfound compression strategies in the absolute limit: register programs which store only a constant amount of information.

Towards $\mathsf{L} \neq \mathsf{P}$

While we are a long way off from showing $\mathsf{TreeEval} \in \mathsf{L}$, there seems no reason to be convinced that constructions given by our techniques, i.e. Catalytic Product Lemma, are optimal. Still, it is entirely possible that $\mathsf{TreeEval} \in \mathsf{L}$ but L is still far from capturing P (see below for questions about the hardness of $\mathsf{TreeEval}$ itself). Instead of $\mathsf{TreeEval}$, the real challenge for L to overcome is the *circuit evaluation problem*, the canonical P-complete problem.

Related to this problem, but even more immediate from our work, is the question of whether the *dag evaluation problem*, which we define as $\mathsf{TreeEval}$ with no tree-like restriction, can be computed in L. Here all our strategies run up against the wall of height; whether in our work or in the broader discussion of catalytic logspace, we know of no techniques that are equipped to handle superlogarithmic height, on which all recursive-based techniques exponentially depend on. We have faith that some progress can be made on this front, and below we discuss the possibility for CL to capture classes such as NC^2; however, the threat posed by height—or perhaps, for those interested in this section, the possibility—is not to be underestimated.

6.2 Open problems

We now leave nay-saying behind and go through some of the central questions posed directly and indirectly by our results.

6.2.1 Main theorems

Our first and most obvious set of open problems is to improve the key theorems we have seen in this work.

Stronger lifting

The key bottleneck in improving the gadget size in Query-to-Communication Lifting Theorem with current techniques is the power of Full Range Lemma, which is determined by the strength of Blockwise Robust Sunflower Lemma. However, attacking the parameters of Blockwise Robust Sunflower Lemma directly may not be the best strategy: improving the blockwise min-entropy requirement to $\log \log 1/\epsilon$ would give us a linear-size gadget, but it would also imply the famous Sunflower Conjecture of Erdos and Rado [ER60], one of the most famous open problems in combinatorics; furthermore it would be an optimal improvement, which would put us very far from constant-size gadgets.

[1] Neither direction is strictly trivial, but showing that such branching programs can be simulated in NC^1 is a straightforward application of divide-and-conquer of the variety that would appear in an undergraduate course.

A better strategy may be to use the structure of our rectangles to get more specialized proofs. For example, we could try to prove alternate forms of Blockwise Robust Sunflower Lemma which get better results for more restricted set systems such as those coming from the rectangle partition. Alternatively, we can reach a different conclusion than Blockwise Robust Sunflower Lemma, either by having a different distribution over the y set, or possibly even a non-probabilistic statement which can give us our contradiction.

Resolving the non-automatability of tree-like Cutting Planes

Our results in Cutting Planes Non-Automatability Theorem all but close the door for the automatability of CP, and like [AM20] we do so with nearly-optimal parameters and hardness assumptions, i.e. superpolynomial hardness assuming $\mathsf{P} \neq \mathsf{NP}$ and exponential hardness assuming ETH. The situation for $\mathsf{tree\text{-}CP}$, on the other hand, is much murkier.

As discussed before, [dR21] is optimal for $\mathsf{tree\text{-}Res}$, as the system is indeed quasi-polynomially automatable, and this also implies that starting from the assumption that $\mathsf{P} \neq \mathsf{NP}$ is unlikely to yield similar results, as a polytime reduction from no instances of SAT to tautologies requiring quasipolynomial-size $\mathsf{tree\text{-}Res}$ proofs would also put NP in quasipolynomial time. Because no quasipolynomial algorithm for automating $\mathsf{tree\text{-}CP}$ is known, we cannot assume that our lifted formula is tight in either the parameters nor the hardness assumption we base our results on.

If it turns out that $\mathsf{tree\text{-}CP}$ does indeed have an $N^{O(\log N)}$-time automating algorithm (where again here $N := \max(n, m, \mathsf{cut\text{-}tree}(\tau)))$, then there is still some tightening up to do; Cutting Planes Non-Automatability Theorem only shows that assuming ETH, we cannot hope for a runtime of $N^{o(\log N/\log^2 \log N)}$. Clearly a tight result would follow from lifting using a constant-size gadget, giving another motivation for the program discussed in Chapter 2 (although one could reasonably feel that this automatability question is one of the weaker motivations for such an important result).

On the other hand, it could turn out that $\mathsf{tree\text{-}CP}$ is hard to even sub-exponentially automate. Here a constant-size lifting result would be a start, but this would still only give us a quasipolynomial lower bound. It would be good to understand the state of upper bounds on the automatability of $\mathsf{tree\text{-}CP}$ to know whether or not to pursue a lower bound past quasipolynomial, and if we suspect that it is indeed exponentially hard to automate then we cannot rely on lifting from $\mathsf{tree\text{-}Res}$ to take us any further, and so other techniques will be necessary.

Better $\mathsf{TreeEval}$ space upper bounds

For the case of space-bounded computation, our mandate is simple: improve the state of Tree Evaluation Algorithm. The most natural way to do so would be to improve Catalytic Product Lemma via a more efficient algorithm than Catalytic Product Procedure, as any reduction in the runtime of computing a degree-d polynomial below exponential would immediately give better $\mathsf{TreeEval}$ algorithms through a new optimal tradeoff in our choice of d.

While our algorithm easily handled the challenge of parallelizing the computation of all of our degree-d monomials, this is actually an obstruction to a very natural approach to such an improvement. A very astute and clever reader may have noticed that we can actually do a single degree-d product much more efficiently than exponential time, namely by repeatedly applying the algorithm from the $d = 2$ case to iteratively reduce the degree by half, resulting in only d recursive calls. The issue is that we know of no way to parallelize this algorithm across s^d monomials without requiring the space for all of them. It is

possible that a modification of this idea could parallelize, which after balancing would give an algorithm for $\mathsf{TreeEval}_{k,2,h}$ which only uses space $O(h \log\log k)$.

Alternatively, if one believes that $\mathsf{TreeEval}$ should ultimately be outside of L, it seems natural to start by lower bounding our technique. Proving that Catalytic Product Lemma is essentially optimal would shed a lot of light on the limitations of existing catalytic techniques, which could either suggest lower bounds against classes such as CL or motivate us to start looking for different approaches beyond polynomials.

Improved catalytic branching program width

Again the clearest path towards improving General Catalytic/Amortized Algorithm is in improving Catalytic Product Procedure, which corresponds to Time-Efficient Catalytic Product Procedure in our case. While the space-time tradeoff is focused on computing monomials rather than working with encodings, getting Time-Efficient Catalytic Product Procedure to be more efficient allows us to balance the parameters and ultimately could take us to $m = 2^{2^{o(n)}}$. Also as with Tree Evaluation Algorithm, an optimal improvement, i.e. a constant number of recursive calls with only a linear size blowup, would give a near-optimal result of linear amortized size for $m = 2^{O(n)}$.

Besides improving the catalytic techniques we developed for tree evaluation, another possible way to improve m for amortized branching programs would be to leverage the non-uniformity of the model in a more clever way. At the moment we only rely on the $\mathbb{F}_2^n$ polynomial representation of our target function, and our branching program acts highly “uniformly” with this representation in hand. All known catalytic techniques were developed for the uniform setting, but it may be interesting to consider non-uniform catalytic tools. Aside from amortized analysis, it is possible that these techniques could be used to show upper bounds for ordinary branching program where traditional space algorithms seem elusive, for example in simulating restricted circuit classes such as TC^1.

As before, lower bounds on Catalytic Product Lemma would rule out optimal tradeoffs and thus pose an interesting obstruction to getting smaller m. A more simple and concrete open problem would be to simply prove that some function requires $m = 2^n$ in order to obtain linear amortized size. A simple counting argument shows that most functions require roughly exponential-size m-catalytic branching programs for any choice of m, and by extension most functions require $m = 2^n/\Omega(n \log n)$. Shaving this factor off the denominator already necessitates going out of the realm of these easy arguments, and may give us some nice structural lower bound techniques.

6.2.2 Other improvements

There are a number of other sub-results that we discussed throughout this thesis that also could be improved upon, as well as generalizations and extensions of our main results to other useful settings.

From lifting to KRW

Equally as important as extending our main theorems, we must not lose sight of the fact that we are setting our sights on the KRW Conjecture, where g is not restricted and there is no monotonicity condition when we move to formula depth. Of these two generalizations, the latter will require us to more fully understand the path from query-to-communication lifting to formula composition.

A recent line of work [dRMN+20] has begun the assault on non-monotone KRW by way of traditional (i.e. non-lifting) composition techniques in the *semi-monotone* case. Perhaps there is a way to fit this setting into a lifting theorem which could then be proven using the techniques developed in Query-to-Communication Lifting Theorem.

Randomized lifting, inner product lifting, etc.

Our new proof of Query-to-Communication Lifting Theorem has thus far generalized to dag-like and graduated lifting, but there are many more lifting theorems worthy of attention. The next clear frontier for this counting style of lifting is the *BPP lifting* of [GPW20]. Most of the framework in Query-to-Communication Lifting Theorem is also as it appears in [GPW20], and so the trouble can be boiled down to the fact that Lemma 7 incurs a *multiplicative* loss in $|Y^{j,\beta}|$ (of $2^{-d \log m}$) rather than an *additive* loss (of $n^{-O(1)}$). Note that this also has to work for a *random* choice of $x \sim X$, rather than just one x^* given by Full Range Lemma, but this is not an issue as we can iteratively find good x^*s and then remove them so we won't rediscover them.

Another frontier, much more open, is that of lifting with the *inner product* gadget, or similarly low-discrepancy gadgets. If we apply the arguments from Query-to-Communication Lifting Theorem as they are, making some minor changes such as flipping the role of Alice and Bob based on who speaks, we will run into trouble when applying Full Range Lemma because Y should no longer be thought of as a large subset of $(\{0,1\}^m)^n$, but rather a small subset of $(\{0,1\}^m)^n$ corresponding to the truth tables of a large number of tuples of linear functions over $\mathbb{F}_2^{\log m}$. Thus we need a version of Blockwise Robust Sunflower Lemma that works for a different distribution of the random y sets; perhaps going back to the proof of Blockwise Robust Sunflower Lemma itself, which finds its roots in Janson's Inequality [Jan90], will give us some insight.

Non-automatability for more systems

There are many proof systems for which automatability results are not known, the two most important ones being *Sherali-Adams* and *Sum-of-Squares*.[2] Most of these systems are of "intermediate" strength, meaning that we have strong lower bounds for certain tautologies but not for very many examples. Here the challenge is to generalize our lower bound techniques in said systems such that we can prove a superpolynomial lower bound against whatever tautology is produced by the reduction, or possibly to find new ways of doing the reduction besides the refutation tautologies of Atserias and Müller or even those of earlier works.

As an example of such an alternative style of proving non-automatability, for strong systems such as Extended Frege there are results known even for the weak version of automatability, but only modulo cryptographic assumptions such as the hardness of factoring. There is an inherent limitation on using Atserias-Müller style arguments for the time being, namely that they would require us to prove at least superpolynomial lower bounds on concrete tautologies coming from the reduction, and so far *no* superpolynomial lower bounds are known. If we ever want to base non-automatability on a more standard assumption such as $\mathsf{P} \neq \mathsf{NP}$ or ETH, this is a barrier we will eventually have to cross.

[2] Both systems were claimed to have been solved in an earlier work [dRGN+21], but these proofs were later retracted.

Catalytic branching programs for more restricted functions

Rather than focusing exclusively on generic functions, it may be useful to approach the question of linear amortized size from the bottom up, by extending our results to other restricted classes of functions. There is a wide range of well-behaved classes at and around the ones studied here, and going slowly up the complexity landscape may bring us incremental updates to the catalytic computing program.

For a concrete example, it is possible that our result for VP could be extended to VNP, with the idea that VNP can be viewed as an exponential number of parallel copies of VP, and exponential parallelism is something we are willing to suffer. This result may be delicate though; such a result would presumably need to use non-uniformity in a stronger way lest we accidentally prove that uniform VNP is contained in CL—and by extension ZPP [BCK$^+$14].

Optimal length for permutation branching programs

Closing the gap between $3n$ and $4n-4$ on the upper and lower bounds for the optimal length of permutation branching programs seems within reach. Given the qualitative and quantitative alignment of Theorems 52 and 55, it seems natural to conjecture that the answer should be one or the other of these values, and not something in between. At the very least, this view may be the most useful to base our work on in the near-term.

A cursory machine search gave no read-3 permutation branching programs for AND on four variables, and if we could formally verify this then it would immediately lead to fully closing the gap at $4n-4$. On the other hand, it may be that some properties of our program from Lemma 54 can be scaled up, or even that a program for lower n such as $n=3$ implies programs for larger n with similar parameters in a black-box way.

6.2.3 Deeper structural results

Finally we outline a number of questions about the relationships between computation models, as well as the underlying structure of our proofs. These are questions with less immediate direction than our other problems, and yet they cut deeper to the core of the power and limitations of composition for depth and space; these can be thought of as a personal research statement for the years to come.

Lifting and combinatorics

It seems that focusing on combinatorial techniques to understand circuit lower bounds and the power of lifting seems not just convenient but also integral. As previously mentioned, a converse to our result, showing that better lifting theorems would imply progress on the sunflower conjecture, was previous shown in [LLZ18], and so our results together suggest that the connection between lifting theorems and sunflowers is deeper than we understand. Furthermore there are a number of monotone circuit lower bounds, including the current benchmark of $2^{n^{1/2-\epsilon}}$ [CKR20], which both use the sunflower lemma directly as well as broader combinatorial techniques used in its proof. Thus if we want to achieve truly exponential monotone lower bounds, or to move beyond the monotone setting—at least in the non-uniform case—we feel that the focus on combinatorics is essential.

Catalytic techniques for uniform log-depth circuit classes

In this vein of understanding the landscape of log-depth poly-size circuit classes, our interest in catalytic computing originally came from studying the structure of complexity classes in the gap between L and P, particularly the wide variety of uniform poly-size log-depth circuit classes. There are arithmetic classes such as VP and $\#\mathsf{AC}^1$, Boolean classes such as AC^1 and TC^1, and hybrid classes such as $\mathsf{AC}^1[p]$ and $\mathsf{CC}^1[p]$. While we know many containment relationships between these classes, We believe that mathematical properties, particularly those of groups and polynomials, should allow us to make more such connections and may even yield surprising equivalences.

In previous work we [AGM17] defined a number of new arithmetic circuit classes whose power seem to sit between VP and $\#\mathsf{AC}^1$, and in doing so we discovered many hidden connections between existing classes in this gap. Perhaps most surprising, we showed that over some fields, poly-size log-depth circuits corresponding to polynomials of (formal) degree $n^{\log n}$ can be transformed into poly-size log-depth circuits corresponding to polynomials of (formal) degree $n^{\log \log n}$! This would normally be impossible, and so the fact that the field size is constant—where there always exists a (potentially exponential size) circuit with degree $n^{O(1)}$—is essential, but this is still a huge step towards showing the power of uniform VP, most notably laid out in the Immerman-Landau conjecture [IL95].

All these classes sit within the class TC^1, which we know by [BCK$^+$14] is contained in CL, and contain the class NC^1, which besides being the subject of interest in Part I was also the class where much of the early work on cleverly reusing space [Bar89, BC92]—work which would eventually inspire catalytic computing—was applied. It would be good to understand which classes in this range can utilize catalytic techniques and to what extent, which we feel is essential to showing many of the equivalences conjectured by our work, as well as understanding the fine-grained landscape between L and P.

The complexity of $\mathsf{TreeEval}$

Despite being a very general problem which seems easy to reduce to—very closely related to one very canonical P-complete problem called the *Circuit Value Problem*—$\mathsf{TreeEval}$ is not known to be complete for any standard complexity class. It is not hard to see that it lies in AC^1, which puts it very close to NL in terms of "standard" complexity classes (i.e. depth-restricted circuit classes and space-restricted Turing Machine classes), but previous work has conjectured $\mathsf{TreeEval}$ to not only be outside L but also NL. Therefore it may be easier to look at a type of complexity class which lies close AC^1 but is not known to be comparable to NL; this has an added benefit, which is that the bold among us can work towards proving $\mathsf{TreeEval} \in \mathsf{L}$ without worrying that it will prove $\mathsf{L} = \mathsf{NL}$.

The most natural candidate is $\mathsf{LogDCFL}$, which is equivalent to the class of *multiplexor circuits*. Multiplexor is another name for the type of internal nodes we have in $\mathsf{TreeEval}$, where the truth table of a function is part of the input; in fact, while we chose to refer to $\mathsf{TreeEval}$ as the Tree Evaluation Problem through this work, it also is known as the *Iterated Multiplexor Problem* for this reason. However, there are some slight mismatches in parameters, not to mention the tree-like restriction, and so a different class in the same vein may be more appropriate for $\mathsf{TreeEval}$ completeness.

Relating CL and P

There is a large gap in our understanding of where CL lies; we know that it contains TC^1 and can be computed in zero-error randomized polynomial time (ZPP). The proof that $\mathsf{CL} \supseteq \mathsf{TC}^1$ goes by showing

that CL can compute many useful composed functions—such as $f(x) \cdot g(x)$ or $f^k(x)$—while only having to recompute the subfunctions a constant number of times. However, this also means that we are inherently "paying" for the compositional structure of circuits exponentially in the depth, which prevents us from moving past circuits of depth $O(\log n)$. Because computing subfunctions with low amounts of recursion has been more fruitful than finding ways of circumventing recursion altogether, it may be that more powerful subfunctions, such as st-connectivity, can also be done efficiently, which could potentially circumvent the barrier posed by depth. In fact, a statement similar to Time-Efficient Catalytic Product Procedure for undirected s-t connectivity may be enough to show $\mathsf{CL} \supseteq \mathsf{NC}^2$, which would give additional hope of bootstrapping up to NC hierarchy more generally.

In the other direction, resolving $\mathsf{CL} \subseteq \mathsf{P}$ is perhaps the largest open problem in catalytic computing, and there are many potential approaches that are worth exploring.

Non-uniform space models

In terms of connecting uniform and non-uniform models of space, $\mathsf{L}/\operatorname{poly}$ is equivalent to the class of problems solvable by $\operatorname{poly} n$-size branching programs. However, this gets trickier for *catalytic logspace* (CL), as the corresponding object for $\mathsf{CL}/\operatorname{poly}$ would be m-catalytic branching programs of amortized size $\operatorname{poly}(n)$ for $m = 2^{\operatorname{poly}(n)}$, which has exponential size and thus cannot be written down in polynomial advice. It would be very interesting to understand the connection between such m-catalytic branching programs and $\mathsf{CL}/\operatorname{poly}$, as this would immediately give lower bounds on m for random functions.

6.3 Epilogue: Composition as computation

To the reader that has made it to these final pages, I hope that you have gained an impression of the important technical questions raised by composition. I want to close by returning to the idea of what composition could mean as a way of understanding computation itself. Without overstating the case, following our initial discussions in Chapter 1 one can posit a view of computation, as we understand it, as *inherently* compositional. I wish to use this view to pose a few broader research discussions.

We began this thesis by taking the abstract question of solving two tasks and rigorously defining the terms of engagement: how are they combined, and what do we want to optimize? Rather than choosing these answers based on our whims—not to mention our capabilities—let us turn these questions outward, as the authors of our main conjectures did, and use them as new angles to attack the tenacious problems of our field. Can we find a new algorithm by focusing on using resources for which we know composition does not hold? Can we find a new lower bound by treating our hard function as compositional in an inobvious way?

As an initial proposition of how to apply this lens, let us go through the other major complexity measures in our field and decide which ones are or are not subject to composition theorems, and possibly even focusing on multiple different "styles" of composition *a la* sequential vs parallel. This can be a powerful method for separating complexity classes, as our conjectures about $\mathsf{TreeEval}$ hardness were meant to do: if composition lower bounds hold for complexity measure A but not complexity measure B, then the corresponding classes can be separated by a function built by the appropriate composition of both measures. We also gain from focusing on just one measure. If we fail to refute composition for s, this gives us a class of lower bounds which we can exploit going forward. If we fail to prove composition

for s, then we can shift to analyzing not just functions that previously seemed impossible, but specifically those for which the composition with respect to s is well understood.

This is hardly a radical suggestion, nor a view to adopt to the exclusion of all others; in fact it only amounts to a different verbal description of the methods we employ every day. However, sometimes our description can orient our thinking, and indeed studying composition has drastically changed the way I approach my own work. I hope that we can integrate it alongside other philosophical views underpenning our field, such as metacomplexity, hardness versus randomness, etc. to reach new heights in our understanding—quantitative and qualitative alike—of the theory of computation.

Bibliography

[ABMP01] Michael Alekhnovich, Samuel R. Buss, Shlomo Moran, and Toniann Pitassi. Minimum propositional proof length is np-hard to linearly approximate. *J. Symb. Log.*, 66(1):171–191, 2001.

[AD08] Albert Atserias and Víctor Dalmau. A combinatorial characterization of resolution width. *J. Comput. Syst. Sci.*, 74(3):323–334, 2008.

[AGHP92] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple construction of almost k-wise independent random variables. *Random Struct. Algorithms*, 3(3):289–304, 1992.

[AGM17] Eric Allender, Anna Gál, and Ian Mertz. Dual VP classes. *Comput. Complex.*, 26(3):583–625, 2017.

[ALN16] Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. *ACM Trans. Comput. Log.*, 17(3):19:1–19:30, 2016.

[ALWZ20] Ryan Alweiss, Shachar Lovett, Kewen Wu, and Jiapeng Zhang. Improved bounds for the sunflower lemma. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 624–630. ACM, 2020.

[AM20] Albert Atserias and Moritz Müller. Automating resolution is np-hard. *J. ACM*, 67(5):31:1–31:17, 2020.

[AR08] Michael Alekhnovich and Alexander A. Razborov. Resolution is not automatizable unless W[P] is tractable. *SIAM J. Comput.*, 38(4):1347–1363, 2008.

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

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

[BCC93] Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Math. Program.*, 58:295–324, 1993.

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

[BCW21] Tolson Bell, Suchakree Chueluecha, and Lutz Warnke. Note on sunflowers. *Discret. Math.*, 344(7):112367, 2021.

[BDG+04] Maria Luisa Bonet, Carlos Domingo, Ricard Gavaldà, Alexis Maciel, and Toniann Pitassi. Non-automatizability of bounded-depth frege proofs. *Comput. Complex.*, 13(1-2):47–68, 2004.

[Bel20] Zoë Bell. Automating regular or ordered resolution is np-hard. *Electron. Colloquium Comput. Complex.*, page 105, 2020.

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

[BPR97] Maria Luisa Bonet, Toniann Pitassi, and Ran Raz. No feasible interpolation for tc0-frege proofs. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science, FOCS '97*, pages 254–263. IEEE Computer Society, 1997.

[BPR00] Maria Luisa Bonet, Toniann Pitassi, and Ran Raz. On interpolation and automatization for frege systems. *SIAM J. Comput.*, 29(6):1939–1967, 2000.

[BW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2):149–169, 2001.

[CCK+20] Parinya Chalermsook, Marek Cygan, Guy Kortsarz, Bundit Laekhanukit, Pasin Manurangsi, Danupon Nanongkai, and Luca Trevisan. From gap-exponential time hypothesis to fixed parameter tractable inapproximability: Clique, dominating set, and more. *SIAM J. Comput.*, 49(4):772–810, 2020.

[CCT87] William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discret. Appl. Math.*, 18(1):25–38, 1987.

[CDKS18] Diptarka Chakraborty, Debarati Das, Michal Koucký, and Nitin Saurabh. Space-optimal quasi-gray codes with logarithmic read complexity. In *ESA*, volume 112 of *LIPIcs*, pages 12:1–12:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[CEI96] Matthew Clegg, Jeff Edmonds, and Russell Impagliazzo. Using the groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, STOC 1996*, pages 174–183. ACM, 1996.

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

[Chv73] Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discret. Math.*, 4(4):305–337, 1973.

[CKLM19] Arkadev Chattopadhyay, Michal Koucký, Bruno Loff, and Sagnik Mukhopadhyay. Simulation theorems via pseudo-random properties. *Comput. Complex.*, 28(4):617–659, 2019.

[CKR20] Bruno Pasqualotto Cavalar, Mrinal Kumar, and Benjamin Rossman. Monotone circuit lower bounds from robust sunflowers. In *Proceedings of LATIN 2020: Theoretical Informatics - 14th Latin American Symposium*, volume 12118 of *Lecture Notes in Computer Science*, pages 311–322. Springer, 2020.

[CKS90] William J. Cook, Ravi Kannan, and Alexander Schrijver. Chvátal closures for mixed integer programming problems. *Math. Program.*, 47:155–174, 1990.

[CL19] Yijia Chen and Bingkai Lin. The constant inapproximability of the parameterized dominating set problem. *SIAM J. Comput.*, 48(2):513–533, 2019.

[CLRS16] Siu On Chan, James R. Lee, Prasad Raghavendra, and David Steurer. Approximate constraint satisfaction requires large LP relaxations. *J. ACM*, 63(4):34:1–34:22, 2016.

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

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

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

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

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

[dR21] Susanna F. de Rezende. Automating tree-like resolution in time $n^{o(\log n)}$ is eth-hard. In *Proceedings of the XI Latin and American Algorithms, Graphs and Optimization Symposium, LAGOS 2021*, volume 195 of *Procedia Computer Science*, pages 152–162. Elsevier, 2021.

[dRGN+21] Susanna F. de Rezende, Mika Göös, Jakob Nordström, Toniann Pitassi, Robert Robere, and Dmitry Sokolov. Automating algebraic proof systems is np-hard. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 209–222. ACM, 2021.

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

[dRNV16] Susanna F. de Rezende, Jakob Nordström, and Marc Vinyals. How limited interaction hinders real communication (and what it means for proof and circuit complexity). In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 295–304. IEEE Computer Society, 2016.

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

[ER60] Paul Erdős and Richard Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, 35(1):85–90, 1960.

[FKNP19] Keith Frankston, Jeff Kahn, Bhargav Narayanan, and Jinyoung Park. Thresholds versus fractional expectation-thresholds. *CoRR*, abs/1910.13433, 2019.

[FKP19] Noah Fleming, Pravesh Kothari, and Toniann Pitassi. Semialgebraic proofs and efficient algorithm design. *Found. Trends Theor. Comput. Sci.*, 14(1-2):1–221, 2019.

[FPPR17] Noah Fleming, Denis Pankratov, Toniann Pitassi, and Robert Robere. Random Θ(log n)-cnfs are hard for cutting planes. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017*, pages 109–120. IEEE Computer Society, 2017.

[FRS88] Lance Fortnow, John Rompel, and Michael Sipser. On the power of multi-power interactive protocols. In *Computational Complexity Conference*, pages 156–161. IEEE Computer Society, 1988.

[GGKS20] Ankit Garg, Mika Göös, Pritish Kamath, and Dmitry Sokolov. Monotone circuit lower bounds from resolution. *Theory Comput.*, 16:1–30, 2020.

[GHM+22] Uma Girish, Justin Holmgren, Kunal Mittal, Ran Raz, and Wei Zhan. Parallel repetition for all 3-player games over binary alphabet. *CoRR*, 2022.

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

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

[GKMP20] Mika Göös, Sajin Koroth, Ian Mertz, and Toniann Pitassi. Automating cutting planes is np-hard. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 68–77. ACM, 2020.

[GL10] Nicola Galesi and Massimo Lauria. On the automatizability of polynomial calculus. *Theory Comput. Syst.*, 47(2):491–506, 2010.

[GLM+16] Mika Göös, Shachar Lovett, Raghu Meka, Thomas Watson, and David Zuckerman. Rectangles are nonnegative juntas. *SIAM J. Comput.*, 45(5):1835–1869, 2016.

[Gom63] Ralph E Gomory. An algorithm for integer solutions to linear programs. *Recent advances in mathematical programming*, 64(14):260–302, 1963.

[Göö15] Mika Göös. Lower bounds for clique vs. independent set. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 1066–1076. IEEE Computer Society, 2015.

[GP18] Mika Göös and Toniann Pitassi. Communication lower bounds via critical block sensitivity. *SIAM J. Comput.*, 47(5):1778–1806, 2018.

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

[GPW20] Mika Göös, Toniann Pitassi, and Thomas Watson. Query-to-communication lifting for BPP. *SIAM J. Comput.*, 49(4), 2020.

[Gra53] Frank Gray. Pulse code communication. https://patents.google.com/patent/US2632058A/en, 1953. US Patent 2632058A.

[HC99] Armin Haken and Stephen A. Cook. An exponential lower bound for the size of monotone real circuits. *J. Comput. Syst. Sci.*, 58(2):326–335, 1999.

[HN12] Trinh Huynh and Jakob Nordström. On the virtue of succinct proofs: amplifying communication complexity hardness to time-space trade-offs in proof complexity. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012*, pages 233–248. ACM, 2012.

[HP17] Pavel Hrubes and Pavel Pudlák. Random formulas, monotone circuits, and interpolation. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017*, pages 121–131. IEEE Computer Society, 2017.

[HP18] Pavel Hrubes and Pavel Pudlák. A note on monotone real circuits. *Inf. Process. Lett.*, 131:15–19, 2018.

[HR00] Danny Harnik and Ran Raz. Higher lower bounds on monotone size. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, STOC 2000*, pages 378–387. ACM, 2000.

[IL95] Neil Immerman and Susan Landau. The complexity of iterated multiplication. *Inf. Comput.*, 116(1):103–116, 1995.

[IR22] Dmitry Itsykson and Artur Riazanov. Automating OBDD proofs is np-hard. *Electron. Colloquium Comput. Complex.*, page 46, 2022.

[Iwa97] Kazuo Iwama. Complexity of finding short resolution proofs. In *Mathematical Foundations of Computer Science 1997, 22nd International Symposium, MFCS'97*, volume 1295 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 1997.

[Jan90] Svante Janson. Poisson approximation for large deviations. *Random Struct. Algorithms*, 1(2):221–230, 1990.

[Juk12] Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012.

[KMR17] Pravesh K. Kothari, Raghu Meka, and Prasad Raghavendra. Approximating rectangles by juntas and weakly-exponential lower bounds for LP relaxations of csps. In *STOC*, pages 590–603. ACM, 2017.

[Kou16] Michal Koucký. Catalytic computation. *Bull. EATCS*, 118, 2016.

[KP98] Jan Krajícek and Pavel Pudlák. Some consequences of cryptographical conjectures for $s^1{}_2$ and EF. *Inf. Comput.*, 140(1):82–94, 1998.

[Kra97] Jan Krajícek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symb. Log.*, 62(2):457–486, 1997.

[Kra98] Jan Krajícek. Interpolation by a game. *Math. Log. Q.*, 44:450–458, 1998.

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

[Kus97] Eyal Kushilevitz. Communication complexity. *Adv. Comput.*, 44:331–360, 1997.

[KW90] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. *SIAM J. Discret. Math.*, 3(2):255–265, 1990.

[LLZ18] Xin Li, Shachar Lovett, and Jiapeng Zhang. Sunflowers and quasi-sunflowers from randomness extractors. In *APPROX-RANDOM*, volume 116 of *LIPIcs*, pages 51:1–51:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[LMM+22] Shachar Lovett, Raghu Meka, Ian Mertz, Toniann Pitassi, and Jiapeng Zhang. Lifting with sunflowers. In *ITCS*, volume 215 of *LIPIcs*, pages 104:1–104:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[LMV] James R. Lee, Raghu Meka, and Thomas Vidick. Private correspondence.

[LRS15] James R. Lee, Prasad Raghavendra, and David Steurer. Lower bounds on the size of semidefinite programming relaxations. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 567–576. ACM, 2015.

[MPW19] Ian Mertz, Toniann Pitassi, and Yuanhao Wei. Short proofs are hard to find. In *ICALP*, volume 132 of *LIPIcs*, pages 84:1–84:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Neč66] E.I. Nečiporuk. A boolean function. *Dokl. Akad. Nauk SSSR*, 169(4), 1966.

[NN93] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993.

[O'D17] Ryan O'Donnell. SOS is not obviously automatizable, even approximately. In *ITCS*, volume 67 of *LIPIcs*, pages 59:1–59:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

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

[PP22] Jinyoung Park and Huy Tuan Pham. A proof of the kahn-kalai conjecture. *CoRR*, abs/2203.17207, 2022.

[PR17] Toniann Pitassi and Robert Robere. Strongly exponential lower bounds for monotone computation. In *STOC*, pages 1246–1255. ACM, 2017.

[PR18] Toniann Pitassi and Robert Robere. Lifting nullstellensatz to monotone span programs over any field. In *STOC*, pages 1207–1219. ACM, 2018.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997.

[Pud00] Pavel Pudlák. Proofs as games. *Am. Math. Mon.*, 107(6):541–550, 2000.

[Pud10] Pavel Pudlák. On extracting computations from propositional proofs (a survey). In *FSTTCS*, volume 8 of *LIPIcs*, pages 30–41. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

[Rao19] Anup Rao. Coding for sunflowers. *CoRR*, abs/1909.04774, 2019.

[Raz85] Alexander A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Doklady Akademii Nauk SSSR*, pages 281:798–801, 1985. English translation in Soviet Math. Doklady 31 (1985), 354–357.

[Raz95a] Ran Raz. A parallel repetition theorem. In *STOC*, pages 447–456. ACM, 1995.

[Raz95b] Alexander A. Razborov. Unprovability of lower bounds on circuit size in certain fragments of bounded arithmetic. *Izvestiya: Mathematics*, 59(1):205–227, feb 1995.

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

[Rob18] Robert Robere. Unified lower bounds for monotone computation. *Ph.D Thesis*, 2018.

[Ros14] Benjamin Rossman. The monotone complexity of k-clique on random graphs. *SIAM J. Comput.*, 43(1):256–279, 2014.

[RPRC16] Robert Robere, Toniann Pitassi, Benjamin Rossman, and Stephen A. Cook. Exponential lower bounds for monotone span programs. In *FOCS*, pages 406–415. IEEE Computer Society, 2016.

[RW17] Prasad Raghavendra and Benjamin Weitz. On the bit complexity of sum-of-squares proofs. In *ICALP*, volume 80 of *LIPIcs*, pages 80:1–80:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

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

[She11] Alexander A. Sherstov. The pattern matrix method. *SIAM J. Comput.*, 40(6):1969–2000, 2011.

[Sok17] Dmitry Sokolov. Dag-like communication and its applications. In *CSR*, volume 10304 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2017.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.

[Ver96] Oleg Verbitsky. Towards the parallel repetition conjecture. *Theor. Comput. Sci.*, 157(2):277–282, 1996.

[WYY17] Xiaodi Wu, Penghui Yao, and Henry S. Yuen. Raz-mckenzie simulation with the inner product gadget. *Electron. Colloquium Comput. Complex.*, page 10, 2017.

[Yao79] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing (preliminary report). In *STOC*, pages 209–213. ACM, 1979.

Appendix A

By the Wayside

After six years of work, it is inevitable that a good body of earlier proofs should be subsumed. In fact, nearly all of the results that appear in this thesis are not just the finished product after a proof has been refined; they are actually the second incarnation of previous published work which are now qualitatively and quantitatively obsolete. In this appendix we recap all of these proofs, chapter by chapter, and note how the work from the body of the thesis compares.

We include these for two reasons. First, while they fail to surpass the results and proofs presented in the thesis, there is no *provable* reason that they will not circumvent future barriers and ultimately provide better results. These are proofs that in many ways follow drastically different paths than the ones taken, and thus it is quite possible that they will be useful, if not to these results then to others. In addition to comparing these results to the ones in the thesis, we will point out the tangible ways in which they may provide such improvements.

The second reason is a social one: I believe that there is value in showing younger researchers the volume of work which falls by the wayside over the course of six years. These results are not particularly impressive—if I may boldly claim that the proofs appearing in this thesis are any more noteworthy—yet following Pareto's Law[1] they represent the sum total of almost three-quarters of my time in graduate school. Their style, preserved with fewer edits than the rest of this thesis, is similarly uneven in many parts, with an emphasis on readability coming in more and more as the proofs get more recent. I am glad to have spent so many years writing mediocre proofs; in the end they taught me just as much.

A.1 Full range without Blockwise Robust Sunflower Lemma

The improved sunflower and robust sunflower lemmas of [ALWZ20, Rao19, BCW21] are powerful new tools that came about in the past few years, and our sunflower approach in Chapter 2 was made possible by the key subroutine Blockwise Robust Sunflower Lemma. Our project started well before these results, however, and the parameter regime was such that we were not even considering it could lead to subquadratic size gadgets.

This is not the only issue, however. Blockwise Robust Sunflower Lemma was a statement with the exact form we needed, stating that high blockwise min-entropy implies that the probability of a random subset of the universe missing every set in our set system is low. The original robust sunflower lemma, due

[1] More commonly known as the 80/20 rule.

to Rossman [Ros14], was of a similar but more sunflower-like form: we need only assume the set system is large enough, but in exchange the same statement only holds after conditioning on, and removing, a *core* of elements.

Definition 23 (Robust sunflowers)**.** Let $p, \kappa \in (0,1)$. A set system $\mathcal{S}$ over universe $\mathcal{U}$ is called a (p,κ)*-robust sunflower* if there exists a set $C \subseteq \mathcal{U}$, called the *core* of $\mathcal{S}$, such that $S \supseteq C$ for all $S \in \mathcal{S}$ and

$$\mathbf{Pr}_{\boldsymbol{y} \subseteq_p \mathcal{U} \setminus C}(\forall \gamma \in \mathcal{S} : \gamma \setminus C \not\subseteq y) \leq \kappa$$

where $\subseteq_p$ means every element of $\mathcal{U}$ is added independently with probability p. If $p = 1/2$ we simply say that $\mathcal{S}$ is a κ-robust sunflower.

Original Robust Sunflower Lemma. *There exists an absolute constant K such that the following holds: Let $s \in \mathbb{N}$ and $\kappa > 0$. Let $\mathcal{F}$ be a set system over $\mathcal{U}$ such that a) $|\gamma| \leq s$ for all $\gamma \in \mathcal{F}$; and b) $|\mathcal{F}| \geq (K \log(s/\kappa))^s$. Then $\mathcal{F}$ contains a κ-robust sunflower.*

Blockwise Robust Sunflower Lemma can be thought of as a version of Robust Sunflower Lemma where the κ-robust sunflower we get is promised to have an empty core, which is essential to the proof of Full Range Lemma. In this section we circumvent this issue and reprove Full Range Lemma—or rather, the d-wise marginal version, i.e. Lemma 15—using only Robust Sunflower Lemma. The parameters will be worse than Lemma 15, and the proof will be slightly more intricate.

Lemma 61. *Let $J \subseteq [n]$, let $d = o(n)$, let $m = d^\Delta$ for some $\Delta \geq 3+\epsilon$ for some $\epsilon > 0$, and let $\delta = \frac{\Delta-3}{\Delta} - \epsilon'$ for some $\epsilon' > 0$. Let $X \subseteq [m]^J$ be such that $\boldsymbol{X}$ has blockwise min-entropy $(1-\delta)\log m - O(1)$, and let $\mathcal{F} = \{\gamma_j\}_j$ be a block-respecting set system over $[m]^J$ such that 1) for all $x \in X$ there exists a $\gamma_j \in \mathcal{F}$ consistent with x, and 2) $|\gamma_j| \leq O(d)$ for all j. Then for any constant $c > 0$,*

$$\mathbf{Pr}_{\boldsymbol{y} \subseteq [mn]}(\forall j : \gamma_j \not\subseteq y) < 2^{-cd\log m}$$

A.1.1 Proof sketch

To give our proof in one line, we iteratively replace a sunflower with its core in our set system until we hit a sunflower with an empty core, i.e. one for which Robust Sunflower Lemma and Blockwise Robust Sunflower Lemma give the same conclusion, while keeping track of the small amount of error this produces. This technique originated with Razborov's method of approximations [Raz85] and since has seen much use, including the previous record for monotone circuit lower bounds [HR00] and the proof of Blockwise Robust Sunflower Lemma itself.

We now go into more technical detail. Our goal will be to show that $\mathcal{F}$ contains an $2^{-cd\log m}$-robust sunflower with an empty core. We do not directly have a sie lower bound of $\mathcal{F}$, which is necessary to apply Robust Sunflower Lemma, and instead use the blockwise min-entropy of X to ensure $\mathcal{F}$ is large. More specifically, because the blockwise min-entropy of $\boldsymbol{X}$ is at least $(1-\delta)\log m - O(1)$, for any non-empty set $\gamma_j \in \mathcal{F}$ the set of all $x \in X$ consistent with γ_j can only cover a $2^{-(1-\delta)|\gamma_j|\log m + O(|\gamma|)}$ fraction of X. Since each $x \in X$ must be consistent with some γ_j, there must be a huge number of sets γ_j in $\mathcal{F}$, and so by Robust Sunflower Lemma $\mathcal{F}$ contains some κ-robust sunflower $\mathcal{F}_{\bar{S}}$ even for very small $\kappa \ll 2^{-cd\log m}$.

If $|S| = 0$ then we are done, but unfortunately using Robust Sunflower Lemma we have no control over $|S|$. Instead, we employ an iterative strategy where we drive down the size of the smallest core S for

which $\mathcal{F}_{\bar{S}}$ is an κ-robust sunflower. For simplicity assume there is some $s \leq O(d)$ such that every set in $\mathcal{F}$ has size s, and so in the worst case we can assume that every core S for which $\mathcal{F}_{\bar{S}}$ is an κ-robust sunflower has size $s-1$. We want to show now that there exist enough such cores S that the collection of these cores *itself* is an κ-robust sunflower, and so it must have a core S' of size at most $s-2$. If this is true then it turns out $\mathcal{F}_{\bar{S}'}$ is an κ'-robust sunflower for κ' only slightly larger than κ. From this we've made progress; by increasing κ slightly we've found a core of a smaller size.

Using this idea, at a high level we will perform an iterative procedure, where we repeat the following three steps until we find a sunflower with an empty core in $\mathcal{F}$: 1) repeatedly pluck κ-robust sunflowers from $\mathcal{F}$; 2) when we have enough sunflowers, pluck an robust sunflower from their cores; 3) increase κ enough so that the core of this new sunflower is the core of an κ-robust sunflower in $\mathcal{F}$ as well. In our actual calculations we will need to keep track of the sets of cores of each size, as well as to focus only on the sets in $\mathcal{F}$ of a certain size. This will allow us to know when we should pluck a sunflower from the cores, and will give us a measure of progress towards finding an empty core, which will allow us to choose our κ small enough to get $2^{-cd \log m}$ at the end.

The last remaining piece is showing that we can actually pluck enough sunflowers from $\mathcal{F}$ to repeat this procedure enough to get an empty core, without running out of sets in $\mathcal{F}$. Unfortunately when we find a core S and pluck the sunflower $\mathcal{F}_{\bar{S}}$, we have no control over how many sets are actually in $\mathcal{F}_{\bar{S}}$, and so it seems hopeless to control how many rounds we can run for. However, note that the $(1-\delta)\log m - O(1)$ lower bound on the blockwise min-entropy of X holds for *any* S over $[m]^J$, which applies to a) the original sets $\gamma_j \in \mathcal{F}$, and b) the cores S that we pluck. Thus instead of arguing that each $\mathcal{F}_{\bar{S}}$ we find is small, we instead argue that the fraction of X covered by sets remaining in $\mathcal{F}$ is large, using the blockwise min-entropy of X for all (non-empty) cores S we've found so far. Then, again using the blockwise min-entropy of X on $\mathcal{F}$, we know that $\mathcal{F}$ must still have many sets to cover the remaining fraction of X, as we did when showing that $\mathcal{F}$ was originally big enough to apply Robust Sunflower Lemma.

A.1.2 Full proof

It is sufficient to show that $\mathcal{F}$ contains an κ-robust sunflower with an empty core for some $\kappa \leq 2^{-cd \log m}$. Again we assume $\emptyset \notin \mathcal{F}$ as the lemma is trivial otherwise, and so for all $s \in [O(d)]$ let $\mathcal{F}(s)$ be the set of all sets in $\mathcal{F}$ of size exactly s, and let $X(s)$ be the set of all $x \in X$ consistent with a set in $\mathcal{F}(s)$. Since every x is consistent with some $\gamma \in \mathcal{F} = \cup_s \mathcal{F}(s)$, we know that $X = \cup_s X(s)$. Therefore by averaging there must exist some $s \in [O(d)]$ such that $|X(s)| \geq \frac{1}{O(d)}|X|$, and so we fix an arbitrary such s.

We define an iterative procedure to find an robust sunflower with an empty core in $\mathcal{F}(s)$. Recall that $m = d^\Delta \geq d^{3+\epsilon}$ and $\delta = \frac{\Delta-3}{\Delta} - \epsilon'$, and define $\delta' := 1 - \frac{2}{\Delta} - \frac{\epsilon'}{2}$.[2]

[2] Again if the reader is more interested in the general idea than the best lifting theorem, an easy setting of parameters is $m := d^7$, $\delta := 0.05$, and $\delta' := 0.4$; at this range we can ignore ϵ and ϵ'.

Algorithm Finding a sunflower with an empty core: pluck and replace

1: Initialization: set $\mathcal{S}^k \leftarrow \emptyset$ for all $k = 0 \dots s-1$, $t \leftarrow 0$, $\kappa_0 \leftarrow 2^{-cd \log m - s^2 \log m}$, $\mathcal{F}^0 := \mathcal{F}(s)$
2: **while** $\mathcal{S}^0 = \emptyset$ **do**
3: ***Abort*** *if the following invariants ever do not hold: 1)* $|\mathcal{S}^k| \leq 2^{(1-\delta')k \log m}$*; 2)* $|\mathcal{F}^t| \geq 2^{(1-\delta')s \log m}$*; 3) for every* k *and every* $S \in \mathcal{S}^k$*,* $|S| = k$ *and* $\mathcal{F}(s)_{\bar{S}}$ *is an* κ_t*-robust sunflower; 4)* $\kappa_t < 2^{-cd \log n}$
4: Let $\mathcal{F}^t_{\bar{S}}$ be an κ_t-robust sunflower in $\mathcal{F}^t$; if none exists, **abort**
5: Increment t and set $\kappa_t \leftarrow \kappa_{t-1}$
6: Set $\mathcal{S}^{|S|} \leftarrow \mathcal{S}^{|S|} \cup \{S\}$ and set $\mathcal{F}^t \leftarrow \mathcal{F}^{t-1} - \mathcal{F}^{t-1}_{\bar{S}}$
7: **while** there exists k such that $|\mathcal{S}^k| = 2^{(1-\delta')k \log m}$ **do**
8: If $k = 0$, **exit** and return κ_t
9: Let $\mathcal{S}^k_{\bar{S}}$ be an κ_0-robust sunflower in $\mathcal{S}^k$; if none exists, **abort**
10: Increment t and set $\kappa_t \leftarrow \kappa_{t-1} + \kappa_0$
11: Set $\mathcal{S}^{|S|} \leftarrow \mathcal{S}^{|S|} \cup \{S\}$, set $\mathcal{S}^{k'} \leftarrow \mathcal{S}^{k'} - \mathcal{S}^{k'}_{\bar{S}}$ for all $k' > |S|$, and set $\mathcal{F}^t \leftarrow \mathcal{F}^{t-1} - \mathcal{F}^{t-1}_{\bar{S}}$

If this process exits without aborting, clearly by invariants (c) and (d), $\mathcal{F}(s)$ is a $2^{-cd \log n}$-robust sunflower with an empty core as desired (note that when the procedure exits, $|\mathcal{S}^0| = 2^{(1-\delta') \cdot 0 \log m} = 1$). Thus we prove that the process never aborts.

First we show that by Robust Sunflower Lemma, in steps 4 and 9 we always find an robust sunflower. Note that by our choice of m,[3]

$$\begin{aligned} m^{1-\delta'} &= d^{(2/\Delta + \epsilon'/2)\Delta} > d^{2+\Omega(1)} \\ &\gg (O(d^2 \log m \cdot \log s \cdot \log\log s))^{1+o(1)} \\ &\geq (\log s) \cdot (2 \log\log s \cdot (cd \log m + s^2 \log m))^{1+o(1)} \end{aligned}$$

For step 4, by invariant (b) and the fact that $1/\kappa_t \leq 1/\kappa_0 = 2^{d \log m + s^2 \log m}$ we have

$$\begin{aligned} |\mathcal{F}^t| &\geq 2^{(1-\delta')s \log m} = (m^{1-\delta'})^s \\ &\geq ((\log s) \cdot (2 \log\log s \cdot (cd \log m + s^2 \log m))^{1+o(1)})^s \\ &= (\log s)^s \cdot (2 \log\log s \cdot \log 1/\kappa_0)^{(1+o(1))s} \\ &\geq (\log s)^s \cdot (2 \log\log s \cdot \log 1/\kappa_t)^{(1+o(1))s} \end{aligned}$$

and for step 9 by the inner loop condition and the fact that $k \leq s$, the same calculation shows

$$\begin{aligned} |\mathcal{S}^k| &= 2^{(1-\delta')k \log m} = (m^{1-\delta'})^k \\ &\geq ((\log k) \cdot (2 \log\log k \cdot (cd \log m + s^2 \log m))^{1+o(1)})^k \\ &\geq (\log k)^k \cdot (2 \log\log k \cdot \log 1/\kappa_0)^{(1+o(1))k} \end{aligned}$$

We now prove that the invariants hold. For invariant (a), clearly after exiting the inner loop $|\mathcal{S}^k| < 2^{(1-\delta')k \log m}$ for all k. Before the inner loop runs we add at most one element to at most one set $\mathcal{S}^k$, and thus for that set $|\mathcal{S}^k| < 2^{(1-\delta')k \log m} + 1$, or in other words $|\mathcal{S}^k| \leq 2^{(1-\delta')k \log m}$. At the start of each iteration of the inner loop at most one set $\mathcal{S}^k$ has size $2^{(1-\delta')k \log m}$, and since we remove at least

[3] Using the easier parameters listed above, $m^{1-\delta'} = (d^7)^{0.4} \gg d^{2.5}$.

one element from it and add at most one element to at most one other set we maintain that invariant.

For invariant (b), assume for contradiction that $|\mathcal{F}^t| < 2^{(1-\delta')s\log m}$. Recall that $\boldsymbol{X}$ has blockwise min-entropy at least $(1-\delta)\log m$, meaning that *every* set S over $[m]^N$ covers at most $2^{-(1-\delta)|S|\log m+O(|S|)}\cdot|X|$ elements in X, and by extension in $X(s)$. In particular this applies to every set $\gamma_j \in \mathcal{F}^t$ as well as every set $S \in \mathcal{S}^k$. Lastly by assumption $|\mathcal{F}^t| < 2^{(1-\delta')s\log m}$, and likewise by invariant (a) we know that $|\mathcal{S}^k| < 2^{(1-\delta')k\log m}$ for every k. Therefore since[4]

$$m^{\delta-\delta'} = d^{(\frac{\Delta-3}{\Delta}-\epsilon'-1+\frac{2}{\Delta}+\epsilon'/2)\Delta} = d^{\Delta-3-\Delta+2-\Delta\epsilon'/2} \ll O(\frac{1}{s})$$

it follows that

$$\begin{aligned}
|X(s)| &\leq |\mathcal{F}^t|\cdot(2^{-(1-\delta)s\log m+O(s)}\cdot|X|) + \sum_{k=1}^{s-1}|\mathcal{S}^k|\cdot(2^{-(1-\delta)k\log m+O(k)}\cdot|X|)\\
&< 2^{(1-\delta')s\log m}\cdot 2^{-(1-\delta)s\log m+O(s)}\cdot|X| +\\
&\quad \sum_{k=1}^{s-1} 2^{(1-\delta')k\log m}\cdot 2^{-(1-\delta)k\log m+O(k)}\cdot|X|\\
&= (\sum_{k=1}^{s} 2^{(\delta-\delta')k\log m+O(k)})\cdot|X|\\
&\leq O(m^{\delta-\delta'})\cdot|X| \ll \frac{1}{s}|X|
\end{aligned}$$

which is a contradiction of our choice of s.

For invariant (c), we first note the following simple observation about sunflowers.

Fact 62. *Let $\mathcal{F}$ and $\mathcal{H}$ be any two set systems such that $\mathcal{H} \subseteq \mathcal{F}$, let $\kappa, \kappa' > 0$ be such that $\kappa \leq \kappa'$, and let S be any set. Then if $\mathcal{H}_{\bar{S}}$ is an κ-robust sunflower, $\mathcal{F}_{\bar{S}}$ is also an κ' robust sunflower.*

Consider $S \in \mathcal{S}^k$. Clearly $|S| = k$ by construction, and so we show that $\mathcal{F}(s)_{\bar{S}}$ is an κ_t-robust sunflower. We consider only the value of t when S was added to $\mathcal{S}^k$, as κ_t only grows, and we do this by induction on t. First observe that for any t, if S was added to $\mathcal{S}^k$ in step 4 then the claim follows immediately since $\mathcal{F}^t \subseteq \mathcal{F}(s)$. This establishes the base case since at $t = 0$ we are at the start of the procedure, and so we consider $t > 0$. We show this with induction on k in reverse order from $s-1$ to 0. If $k = s-1$, since there is no $\mathcal{S}^{k'}$ for $k' > s-1$ it must have been added in step 4, and so again the claim follows immediately. Thus we consider $k < s-1$ and assume S was added in step 9.

Let $k' > k$ be such that $\mathcal{S}^{k'}_{\bar{S}}$ was the sunflower discovered in step 9 which made us add S to $\mathcal{S}^k$. We claim that $\mathcal{F}(s)_{\bar{S}}$ is an $(\kappa_{t-1}+\kappa_0)$-robust sunflower, which completes the claim since $\kappa_t = \kappa_{t-1}+\kappa_0$. Consider the probability that a random set $y \subseteq [mN] - S$ doesn't contain any set in $\mathcal{F}(s)_{\bar{S}}$. For this to happen, for every set $S' \in \mathcal{S}^{k'}_{\bar{S}}$ either y contains no sets in $\mathcal{F}(s)_{\bar{S}'}$ or it does not contain S' itself. If there is some S' such that $S' \subseteq y$, then by the inductive hypothesis on t and k we know that $\mathcal{F}^{t'}_{\bar{S}'}$ is an $\kappa_{t'}$-robust sunflower, where $t' \leq t-1$ was the value of t when S' was added to $\mathcal{S}^k$. Since $\kappa_{t'} \leq \kappa_{t-1}$ and $\mathcal{F}^{t'} \subseteq \mathcal{F}(s)$, by extension y avoids every set in $\mathcal{F}(s)_{S'}$ with probability at most κ_{t-1}. In the other case where no such S' exists, then because $\mathcal{S}^{k'}_{\bar{S}}$ is an κ_0-robust sunflower y avoids every set $S' \in \mathcal{S}^{k'}_{\bar{S}}$ with probability at most κ_0. Taking a union bound over these two events gives us our claim.

[4]Using the easier parameters listed above, $m^{\delta-\delta'} = (d^7)^{-0.35} \ll d^{-2}$.

Finally for invariant (d), we claim that $t \leq 2^{s^2 \log m} - 1$ when the process ends. Putting this fact together with $\kappa_0 := 2^{-cd \log m - s^2 \log m}$ and $\kappa_t \leq \kappa_{t-1} + 2^{-cd \log m - s^2 \log m}$ for all t gives us

$$\kappa_t \leq \kappa_0 + t \cdot \kappa_0 \leq 2^{s^2 \log m} \cdot 2^{-cd \log m - s^2 \log m} = 2^{-cd \log m}$$

We associate each tuple $\mathcal{S} := (\mathcal{S}^k)_{k=1\ldots s-1}$ with the string $\tau(\mathcal{S}) = |\mathcal{S}^1|\#|\mathcal{S}^2|\#\ldots\#|\mathcal{S}^{s-1}|$. We claim that for every t there is a unique string τ_t corresponding to $\tau(\mathcal{S})$ at the time t was incremented. This is simply because in every round of the outer loop we increase the size of at least one set $\mathcal{S}^k$, and in every round of the inner loop that we cause some $\mathcal{S}^k$ to shrink in some round of the inner loop, we also cause some set $\mathcal{S}^{k'}$ to grow where $k' < k$. By invariant (a) and the inner loop condition, $|\mathcal{S}^k| \leq 2^{(1-\delta')k \log m}$ for every k whenever we updated t, and so as long as $|\mathcal{S}^0| = 0$—in other words for all t except the very last one—we have

$$t \leq |\tau(\mathcal{S})| = \prod_{k=1}^{s-1} 2^{(1-\delta')k \log m} = (2^{(1-\delta') \log m})^{\sum_{k=1}^{s-1} k} < 2^{s^2 \log m} - 2$$

and so at the end of the procedure $t \leq 2^{s^2 \log m} - 1$.

A.2 Graduated lifting through megacoordinates

In the previous section we saw another proof in which Full Range Lemma was amenable to setting $m = d^{1+\epsilon}$. In either case, since Full Range Lemma was the only place in Query-to-Communication Lifting Theorem where m was relevant, this gave us an immediate path to graduated lifting. As we noted, however, there used to be another reliance on m in Lemma 7, namely when we union bound $|Y_=|$; our new argument on $Y_=$ exploited the blockwise min-entropy violations of each assignment corresponding to an X^j, which gave us $2^{O(d \log m)}$.

What would happen if we did not have this more clever dodge, such as in our real dag-like lifting? Is there *still* a way to do graduated lifting when the union bound gives us $2^{O(n \log m)}$ sets to work over? In our original paper [GKMP20] we did not seek to avoid this union bound, as even the existing Fourier-based techniques for Full Range Lemma were not amenable to moving from n to d. Thus, yet a third proof of graduated lifting is required.

Our approach was to bring down the universe size. Let $N = |\operatorname{free}(\rho)|$ be the set of unfixed coordinates, and we run Rectangle Partition as before, knowing that all our assignments have size $O(d)$. We group $[N]$ into $\operatorname{poly}(d)$ "megacoordinates" of size $N/\operatorname{poly}(d)$ such that most (I, α) assignments in the rectangle partition each only point to one value per megacoordinate. We use the (I, α)'s to replace the xs with shorter x' vectors which only point to one value per megacoordinate, and repeat the argument in Rectangle Lemma but only using sets of megacoordinates $I \subseteq [\operatorname{poly}(d)]$. Since $m = \operatorname{poly}(d)$, $m^{-\Omega(|I|)}$ is enough to cancel out $(\operatorname{poly}(d))^{|I|}$, and so the union bound goes through, giving an x' that makes $\textsc{Ind}^I_{m \cdot N/\operatorname{poly}(d)}(x', \mathbf{y})$ close to uniform. Using the way we constructed the x's out of the rectangle partition, this will give us an assignment (I, α) which is equally close to uniform from the partition.

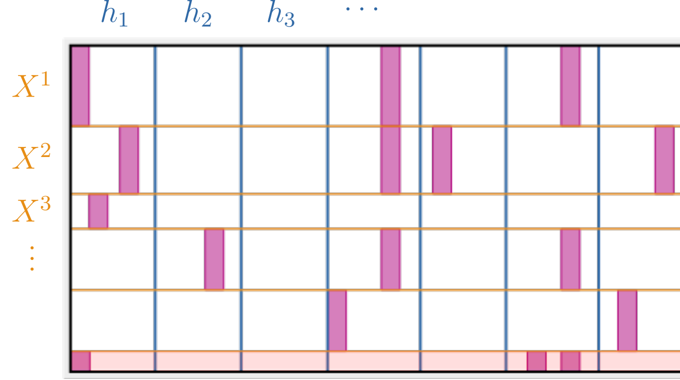

Figure A.1: After partitioning X into $\{X^j\}$ (purple regions are the coordinates of I_j, the restriction α_j to I_j not pictured), we randomly block up the coordinate space $[N]$ into poly(d) megacoordinates (labeled h_i here). With high probability only a small fraction of X will be lost due to collisions.

A.2.1 Reproof of Lemma 7

We index the free coordinates free(ρ) by $[N]$, where $N = |\operatorname{free}(\rho)|$, and for convenience we assume that $N \geq d^5$.[5] We group the coordinates in $[N]$ into d^3 mega-coordinates. Let $\mathbf{h}$ be a random variable which is uniform over all functions h mapping $[N] \to [d^3]$ where $|h^{-1}(i^h)| = \frac{N}{d^3}$ for all $i^h \in [d^3]$. Consider the subset of $\mathcal{F}$ consisting only of pairs (I_j, α_j) such that all coordinates in I_j are mapped to different mega-coordinates by h, or formally

$$\mathcal{F}^h = \{(I_j, \alpha_j) \in \mathcal{F} : \forall i \neq i' \in I_j, h(i) \neq h(i')\}$$

Let $X_h \subseteq X$ be the union of all X^j sets of the rectangle partition such that $(I_j, \alpha_j) \in \mathcal{F}_h$.

Claim 63. *With high probability over $h \sim \mathbf{h}$, we have $|X_h| \geq 0.99|X|$.*

Proof. We show that for a uniform choice of x from X, with high probability the unique part $X^{j(x)}$ which contains x survives into X_h. See Figure A.1 for an illustration. Formally, $\mathbf{Pr}_{h\sim\mathbf{h}}[\mathbf{Pr}_{x\sim\mathbf{x}}(X^{j(x)} \not\subseteq X_h)] < 0.01$. First we consider the case of a fixed x. We will switch the calculation by treating h as a fixed partition from $\mathbf{h}$ and treating $I_{j(x)}$ as a random set of size at most $10d$. To see that these are equivalent, we can treat $h \sim \mathbf{h}$ as simply being a uniformly random permutation on $[N]$ with a fixed partition into d^3 equal sized megacoordinates, and so we can view $I_{j(x)}$ as a random set over $h([N])$.

Recalling that $N \geq d^5$, a straightforward calculation shows that

$$\begin{aligned}
\mathbf{Pr}_{I_{j(x)}}(\forall i \neq i' \in I_{j(x)} : h(i) \neq h(i')) &= \prod_{i=0}^{10d} 1 - \frac{i \cdot (N/d^3 - 1)}{N - i} \\
&\geq (1 - \frac{10d \cdot N/d^3}{N/2})^{10d} \\
&\geq (1 - \frac{20}{d^2})^{10d} \\
&\geq e^{-200/d} \geq 0.99
\end{aligned}$$

[5] Indeed, real lifting theorems already exist for large enough gadgets; moreover, the statement is only easier to prove in the case of large d.

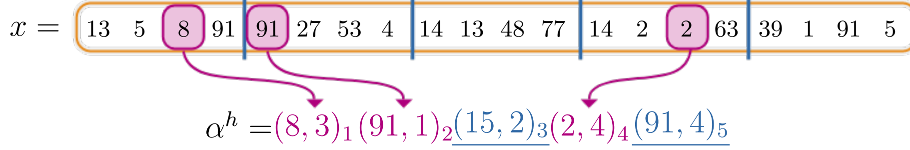

Figure A.2: Example of sampling α^h for $d^3 = 5$ megacoordinates of size $mN/d^3 = 4$. Here (I_j, α_j) for $I_j = \{3, 5, 15\}$ and $\alpha_j = \{(8)_3, (91)_5, (2)_{15}\}$ is sampled. $(8)_3$ goes to $(8, 3)$ in the first coordinate, $(91)_5$ goes to $(91, 1)$ in the second coordinate, and $(2)_{15}$ goes to $(2, 4)$ in the fourth coordinate. For the third and fifth coordinate a pair in $[m] \times [5]$ is chosen uniformly, choosing $(15, 2)$ for the third and $(91, 4)$ for the fifth.

and so the same holds for $\mathbf{Pr}_{h\sim\mathbf{h}}(\forall i \neq i' \in I_{j(x)} : h(i) \neq h(i'))$ by our previous argument. Therefore

$$\begin{aligned}
\mathbf{Pr}_{h\sim\mathbf{h}}[\mathbf{Pr}_{x\sim\mathbf{x}}(X^{j(x)} \not\subseteq X_h)] &= \mathbf{Pr}_{h\sim\mathbf{h}}[\mathbf{Pr}_{x\sim\mathbf{x}}(\exists i \neq i' \in I_{j(x)} : h(i) = h(i'))] \\
&= \mathbf{Pr}_{x\sim\mathbf{x}}[\mathbf{Pr}_{h\sim\mathbf{h}}(\exists i \neq i' \in I_{j(x)} : h(i) = h(i'))] \\
&\leq \sum_{x\in X} \mathbf{Pr}_{x'\sim\mathbf{x}}(x' = x)\mathbf{Pr}_{h\sim\mathbf{h}}(\exists i \neq i' \in I_{j(x)} : h(i) = h(i')) \\
&< \sum_{x\in X} \mathbf{Pr}_{x'\sim\mathbf{x}}(x' = x) \cdot 0.01 \\
&= 0.01 \sum_{x\in X} \mathbf{Pr}_{x'\sim\mathbf{x}}(x' = x) = 0.01
\end{aligned}$$

which completes our claim. □

Henceforth, fix any h satisfying $|X_h| \geq 0.99|X|$. We shift to viewing each $y \in Y$ as a matrix $y^h \in Y^h$ with $m \cdot N/d^3$ rows and d^3 columns in the canonical way, where each entry $((\alpha, i), i^h)$ in y^h corresponds to the entry (α, i') in the original matrix y, where i' is the ith element of the megacoordinate i^h. Following our usual conventions let $\mathbf{x}_h$ be the uniform random variable for selecting x from X_h and let and $\mathbf{y}^h$ be the uniform random variable for selecting y from Y and viewing it as y^h as described above.

Recall that X satisfied $0.9 \log m$-blockwise min-entropy, and so for any $I \subseteq [N]$, $H_\infty(X_I) \geq 0.9 \cdot |I| \log m$. Thus for all assignments α_I,

$$\begin{aligned}
\mathbf{Pr}_{x\sim\mathbf{x}_h}(x_I = \alpha_I) &\leq \frac{|X|}{|X_h|}\mathbf{Pr}_{x\sim\mathbf{x}}(x_I = \alpha_I) \\
&\leq \frac{1}{0.99} \cdot 2^{-0.9|I|\log m} \leq 2^{-0.89|I|\log m}
\end{aligned}$$

and so X_h satisfies $0.89 \log m$-blockwise min-entropy.

Now we define the random variable $\boldsymbol{\alpha}^h$ on $([m] \times [\frac{N}{d^3}])^{d^3}$ to be a random restriction on x that picks one location in each mega-coordinate and assigns it a restriction α. Note that this can also be viewed as choosing a location in each column of y^h. The restriction will be sampled according to $\mathcal{F}^h$, by first sampling $x \sim \mathbf{x}_h$ and taking all assignments in the corresponding pair $(I_j, \alpha_j)_{h(I_j)}$ where $j = j(x)$, and then choosing a random assignment $(i, \alpha^i)_{i^h}$ for all mega-coordinates i^h left unassigned by α_j.

Formally we define $\boldsymbol{\alpha}^h$ by the following procedure:

- sample $x \sim \mathbf{x}_h$ and let $j = j(x)$
- for each $i^h \in h(I_j)$ let i be the coordinate in I_j mapping to i^h

and set $\alpha^h \leftarrow \alpha^h \cup ((\alpha_j)_i, i)_{i^h}$

- for each $i^h \notin h(I_j)$ choose i uniformly from $h^{-1}(i^h)$, choose α^i uniformly from $[m]$, and set $\alpha^h \leftarrow \alpha^h \cup (\alpha^i, i)_{i^h}$
- return α^h

Note that extending α_j uniformly to α^h does not change the min-entropy. Thus because X_h has blockwise min-entropy at least $0.89 \log m$, $\boldsymbol{\alpha}^h$ has blockwise min-entropy at least $0.89 \log m$ as well, and the coordinates of every α^h are exactly $[d^3]$.

To proceed we now state a key lemma which is a generalized version of the Uniform Marginals Lemma of [GPW20]. For completeness, we prove it in Section A.2.2.

Definition 24 (Multiplicative uniformity)**.** We say a random variable $\boldsymbol{x} \in S$ is ϵ-multiplicatively uniform if $\mathbf{Pr}[\boldsymbol{x} = x] = (1 \pm \epsilon) \cdot \frac{1}{|S|}$ for all outcomes $x \in S$.

Large Index Lemma. *Let $\boldsymbol{x} \subseteq [\ell]^k$ and $\boldsymbol{y} \in (\{0,1\}^\ell)^k$ be random variables such that $\boldsymbol{x}$ has blockwise min-entropy $\geq 50 \log k$ and $\mathbf{D}_\infty(\boldsymbol{y}) \leq k$. Then there exists $x \in \operatorname{supp}(\boldsymbol{x})$ such that $\textsc{Ind}_\ell^k(x, \boldsymbol{y})$ is $o(1)$-multiplicatively uniform.*

We apply Large Index Lemma with $\boldsymbol{x} \coloneqq \boldsymbol{\alpha}^h$, $\boldsymbol{y} \coloneqq \boldsymbol{Y}^h$, $\ell \coloneqq mN/d^3$, $k \coloneqq d^3$. Note that $\mathbf{D}_\infty(\boldsymbol{y}) \leq O(d) \leq k$ and that $\boldsymbol{x}$ has blockwise min-entropy $\geq 0.89 \log m \geq 0.89 \log d^{999} \geq 50 \log d^3 = 50 \log k$. We conclude that there is an $\alpha^h \in \operatorname{supp} \boldsymbol{\alpha}^h$ such that $\textsc{Ind}_{mN/d^3}(\alpha^h, \boldsymbol{y}^h)$ is $o(1)$-multiplicatively uniform. Fix such an α^h and let (I_j, α_j) be any pair from which α^h can be sampled in our previous procedure.

We can now undo our grouping into mega-coordinates: Because $\textsc{Ind}_{mN/d^3}(\alpha^h, \mathbf{y}^h)$ is $o(1)$-multiplicatively uniform, by marginalizing to I_j we have that for all $x \in X^j$, $\textsc{Ind}_m^{I_j}(x, \mathbf{y}) = \textsc{Ind}_{mN/d^3}^{I_j}(\alpha_j, \mathbf{y}^h)$ is also $o(1)$-multiplicatively-close to uniform.

$$\mathbf{Pr}_{y \sim \mathbf{y}}(y \in Y^{j,\beta}) \geq (1 \pm o(1)) 2^{-|I_j|} \geq \frac{1}{2} \cdot 2^{-|I_j|}$$

A.2.2 Proof of Large Index Lemma

We state two key lemmas before proving Large Index Lemma. For convenience we shorten the base of the expectation when the variable in the inner expression is clear. The first lemma is a standard application of Fourier analysis which appears in different forms in many papers; we state the version needed to prove Large Index Lemma and prove it at the end of this subsection, following the proof of [LMV].

Lemma 64. *Let Λ and Γ be random variables on $X \coloneqq [\ell]^k$ and $Y \coloneqq (\{\pm 1\}^\ell)^k$ respectively. Assume that Λ has blockwise min-entropy $\beta > 1/2$ and Γ has deficiency s. Then for every $I \subseteq [k]$,*

$$|\mathbb{E}_{\Lambda,\Gamma}[\chi_I(y_x)]| \leq (2^{-\beta/2-1}(k+s))^{|I|}$$

where $\chi_I(y_x) = \prod_{i \in I} y_i(x_i)$

The second lemma appeared in a different form in [GPW20] as Lemma 9. We omit the proof and defer interested readers to [GPW20].

Lemma 65. *Let $x \in [\ell]^k$ and $Y \subseteq \{\pm 1\}^{\ell \times k}$ be such that*

$$|\mathbb{E}_{\mathbf{y}}[\chi_I(y_x)]| \leq 2^{-10|I| \log k}$$

for all $I \subseteq [k]$. Then $\mathbf{y}_x$ is $1/k^3$-multiplicatively-close to uniform.

Proof of Large Index Lemma. We map all y from elements of $\{0,1\}^{\ell \times k}$ to $(\{\pm 1\}^{\ell})^k$ in the natural way. Applying Lemma 64 we get that for all $I \subseteq [k]$

$$|\mathbb{E}_{\Lambda,\mathbf{y}}[\chi_I(y_x)]| \leq (2^{-25\log k - 1}(k+k))^{|I|} \leq 2^{-20|I|\log k}$$

where the second inequality is by assumption. By Markov's inequality then, for any $I \subseteq [k]$

$$\mathbf{Pr}_{x\sim\Lambda}(|\mathbb{E}_{\mathbf{y}}[\chi_I(y_x)]| > 2^{-10|I|\log k}) \leq 2^{-10|I|\log k}$$

We say x is *good* if $|\mathbb{E}_{\mathbf{y}}[\chi_I(y_x)]| \leq 2^{-10|I|\log k}$ for all $I \subseteq [k]$. Taking a union bound over all such I we get

$$\begin{aligned}\mathbf{Pr}_{x\sim\Lambda}(x \text{ is not good}) &\leq \sum_{I\subseteq[k]} \mathbf{Pr}_{x\sim\Lambda}(|\mathbb{E}_{\mathbf{y}}[\chi_I(y_x)]| > 2^{-10|I|\log k}) \\ &\leq \sum_{I\subseteq[k]} 2^{-10|I|\log k} \\ &\leq \sum_{t=1}^{k} \binom{k}{t} 2^{-10t\log k} \\ &\leq \sum_{t=1}^{k} 2^{-9t\log k} \leq 2/k^9\end{aligned}$$

Hence most x are good, and by Lemma 65 for any good x we have that $\textsc{Ind}_k(x,\mathbf{y})$ is $1/k^3$-multiplicatively-close to uniform. ☐

Proof of Lemma 64. Because marginalizing Γ to any $S \subseteq \ell \times k$ cannot increase the deficiency of Γ_S in Y_S, it is enough to show that

$$|\mathbb{E}_{\Lambda,\Gamma}[\chi(y_x)]| \leq (2^{-\beta/2-1}(k+s))^k$$

Let $\Lambda(x) = \mathbf{Pr}(\Lambda = x)$. Because Λ has blockwise min-entropy β, it has Renyi entropy at least $\beta \cdot k$, meaning $\sum_x \Lambda(x)^2 \leq 2^{-\beta\cdot k}$. By Cauchy-Schwarz

$$\begin{aligned}|\mathbb{E}_{\Lambda,\Gamma}[\chi(y_x)]| &= \sum_x \Lambda(x)|\mathbb{E}_{\Gamma}[\chi(y_x)]| \\ &\leq (\sum_x \Lambda(x)^2)^{1/2}(\sum_x |\mathbb{E}_{\Gamma}[\chi(y_x)]|^2)^{1/2} \\ &\leq 2^{-(\beta/2)k} \cdot (\sum_x |\mathbb{E}_{\Gamma}[\chi(y_x)]|^2)^{1/2} \\ &= 2^{-(\beta/2)k} \cdot (\sum_x |\mathbb{E}_{\Gamma}[\chi(y_x)]|^2)^{1/2}\end{aligned}$$

We thus turn our attention to proving a bound on $\sum_x |\mathbb{E}_{\Gamma}[\chi(y_x)]|^2$. Let $\chi_{\geq i}(y_x) = \chi_{\{i\ldots k\}}(y_x)$. Again by Cauchy-Schwarz

$$\begin{aligned}\sum_x |\mathbb{E}_{\Gamma}[\chi(y_x)]|^2 &= \sum_x |\prod_i \mathbb{E}_{\Gamma}[\chi_{\geq i}(y_x)]|^2 \\ &\leq \sum_x \prod_i \mathbb{E}_{\Gamma}[\chi_{\geq i}(y_x)]^2\end{aligned}$$

$$= \sum_{x_2 \dots x_k} \prod_{i \geq 2} \mathbb{E}_\Gamma[\chi_{\geq i}(y_x)]^2 \cdot \sum_{x_1} \mathbb{E}_\Gamma[\chi_{\geq 1}(y_x)]^2$$

Since $\mathbf{H}_\infty(\Gamma) \geq \ell k - s$, for a fixed $x_2 \dots x_k$

$$\mathbf{H}(\chi_{\geq 1}(y_x)) = \mathbf{H}(\Gamma_1 \mid \Gamma(x_2) \dots \Gamma(x_k)) \geq \ell - (k+s)$$

By Pinsker's inequality $\mathbb{E}_\Gamma[\chi_{\geq 1}(y_x)]^2 \leq (1 - \mathbf{H}(\chi_{\geq 1}(y_x)))/2$, and so by sub-additivity of the expectation

$$\sum_{x_1} \mathbb{E}_\Gamma[\chi_{\geq 1}(y_x)]^2 \leq (\ell - (\ell - (k+s))/2 = (k+s)/2$$

Plugging this back into our previous expression we get

$$\sum_{x_2 \dots x_k} \prod_{i \geq 2} \mathbb{E}_\Gamma[\chi_{\geq i}(y_x)]^2 \cdot \sum_{x_1} \mathbb{E}_\Gamma[\chi^2_{\geq 1}(y_x)]^2 \leq \frac{k+s}{2} \sum_{x_2 \dots x_k} \prod_{i \geq 2} \mathbb{E}_\Gamma[\chi_{\geq i}(y_x)]^2$$

Finally we repeat for all $i = 2 \dots k$, and in the end we get

$$\begin{aligned}
\sum_x |\mathbb{E}_\Gamma[\chi(y_x)]|^2 &\leq \sum_x \prod_i \mathbb{E}_\Gamma[\chi_{\geq i}(y_x)]^2 \\
&\leq \frac{k+s}{2} \sum_{x_2 \dots x_k} \prod_{i \geq 2} \mathbb{E}_\Gamma[\chi_{\geq i}(y_x)]^2 \\
&\dots \\
&\leq (\frac{k+s}{2})^k \prod_{i > k} \mathbb{E}_\Gamma[\chi_{\geq i}(y_x)]^2 = (\frac{k+s}{2})^k
\end{aligned}$$

Putting this bound on $\sum_x |\mathbb{E}_\Gamma[\chi(y_x)]|^2$ together with the earlier proof completes the lemma. □

A.3 Quasipolynomial non-automatability of many systems

Before the results of Atserias and Müller [AM20], the automatability of Resolution and other systems was wide open. However, it was not the case that nothing was known. Alekhnovich and Razborov [AR08] gave the first major result for weaker systems, showing that under an assumption from parameterized complexity—namely $\mathsf{W[P]} \neq \mathsf{FPT}$, the same assumption as in de Rezende's result on non-polynomial automatability of tree-like Resolution [dR21]—Resolution and tree-like Resolution are not polynomially automatable. This result was later extended to two other systems, namely *Nullstellensatz* and *Polynomial Calculus*, by Galesi and Lauria [GL10] using the same reduction.

In a later work, we [MPW19] adapted the reduction to give the first quasipolynomial automatability lower bounds on all the above systems, as well as *k-Resolution*, under the stronger ETH assumption. This was also the result that we lifted from in our original work on the non-automatability of tree-like Cutting Planes [GKMP20]. These results have since been entirely subsumed by the extensions of [AM20] of the past two years [dR21, dRGN+21], but the technique is still quite interesting as it cleverly encodes a universal statement into an existential statement, which may have uses in the future. In this section we will go through this reduction and sketch the proofs of non-automatability from them.

A.3.1 Basic setup

We will not introduce the non-Resolution systems mentioned in the introduction, as we direct interested readers to [MPW19] if they are interested in the (fairly standard) non-automatability proofs themselves. In this section we focus on the reduction itself, as this contains the interesting ideas for future work.

First, we need a hard problem to encode. The starting point of our departure from [AR08] is our use of the (gap version of the) *hitting set* problem. This will be more amenable to superpolynomial hardness than the more general minimum monotone circuit satisfying assignment problem used in previous works.

Definition 25. Let $\mathcal{S} = \{S_1, \dots, S_n\}$ be a collection of non-empty sets S_j over $[n]$. A *hitting set* $H \subseteq [n]$ is a set of elements such that $H \cap S_j \neq \emptyset$ for all $j \in [n]$. Let $\gamma(\mathcal{S})$ be the size of the smallest hitting set for $\mathcal{S}$. The *gap hitting set problem* is the task of distinguishing, on input $(\mathcal{S}, k, hk)$, the following two cases: (1) $\gamma(\mathcal{S}) \leq k$; (2) $\gamma(\mathcal{S}) > hk$.

In terms of the hardness of gap hitting set, our first goal was to deduce the following hardness statement from an earlier construction by Chen and Lin [CL19].[6].

Lemma 66 (ETH-Hardness of Hitting Set)**.** *Assuming* ETH*, for sufficiently large* n *and* $k = O(\log^{1/7-\epsilon} \log n)$ *no algorithm can solve the gap hitting set problem* $(\mathcal{S}, k, k^2)$ *in time* $n^{o(k)}$.

Our main lemma will be similar to that of Cutting Planes Non-Automatability Theorem and other works discussed: we will build a tautology whose proof size depends on the value of the given hitting set instance, thus allowing us to solve gap hitting set given an automating algorithm.

Lemma 67. *Let* $\mathcal{Q} \in \{\mathsf{Res}, \mathsf{tree\text{-}Res}, \mathsf{Nullsatz}, \mathsf{PC}, \mathsf{PCR}\}$*. For sufficiently large* n *and* $k = O(\log^{1/3} n)$*, let* $(\mathcal{S}, k, k^2)$ *be an instance of the gap hitting set problem over* $[n]$*. Then there exists an unsatisfiable CNF* $\tau_{\mathcal{S}}$ *which can be computed in time* $n^{O(1)}$ *such that the following two properties hold*

1. *if* $\gamma(\mathcal{S}) \leq k$ *then* $S_{\mathcal{Q}}(\tau_{\mathcal{S}}) \leq n^{O(1)}$*;*
2. *if* $\gamma(\mathcal{S}) > k^2$ *then* $S_{\mathcal{Q}}(\tau_{\mathcal{S}}) \geq n^{\Omega(k)}$*.*

Furthermore for $\mathcal{Q} = \mathsf{Res}[r]$*, the same holds except* $S_{\mathcal{Q}}(\tau_{\mathcal{S}}) \geq n^{\Omega(k/\exp(r^2))}$ *in the latter case.*

Hereafter, fix $k = O(\log^{1/7-\epsilon} \log n)$. Note that the non-automatability that we get is only a function of k for which we can prove hardness of gap hitting set, i.e. the value of k appearing in Lemma 66.

A.3.2 Constructing a hard tautology

We now move into the construction of our tautology. Define $m := n^{1/k}$, and observe that $k \log m = \log n$ and $k^2 < \log m$ for large enough n.

Basic tautology

Consider a set $A \subseteq \{0,1\}^m$ of m-bit strings such that $|A| = m$. We say that A is (m,k)*-universal* if for every subset $J \subseteq [m]$ of up to k distinct positions in $[m]$, the projection $A|_J$ (restricting the strings in A to these positions) contains all possible $2^{|J|}$ binary strings of length $|J|$. Observe that we can take

[6] The actual lemma we prove is slightly more technical but actually slightly stronger than the one we state here. Also a similar result holds for the *Gap Exponential Time Hypothesis*, which we similarly deduced from recent work of Chalermsook et al [CCK+20]. See the full version for more details.

the dual of the set A in the following sense: if $A = \{a_1, \ldots, a_m\}$, and let $B \subseteq \{0,1\}^m$ be the set of all strings b_j for $j \in [m]$ such that the ith bit of b_j is the jth bit of a_i. Another way to think about B is taking the strings of A to be the columns of an $m \times m$ matrix and letting B be the columns of that matrix's transpose. We say A is (m,k)-*dual-universal* if B is (m,k)-universal. Equivalently A is (m,k)-dual-universal if for every ordered subset $I \subseteq A$ of up to k distinct strings in A and for every string $s \in \{0,1\}^{|I|}$, there exists some position $j \in [m]$ such that s is the string formed by concatenating the jth bit of all strings in I in order.

The existence of efficiently constructible $(m, \log m/4)$-universal sets is known. It is also known that there exist efficiently constructible sets that are both $(m, \log m/4)$-universal *and* $(m, \log m/4)$-dual-universal. For a concrete example, [AR08] uses the *Paley graph* G_m on m vertices [7] We will fix an arbitrary A that is efficiently computable and is both $(m, \log m/4)$-universal and $(m, \log m/4)$-dual-universal.

In what follows we will abuse notation and x_i, y_j will denote a tuple of Boolean variables (rather than a single Boolean variable). The tuple size of x_i, y_j will be clear from context, but generally x_i will be a $O(\log m)$-tuple and y_j will be a $O(\log n)$-tuple. Additionally $\vec{x} = x_1, \ldots, x_n$, $\vec{y} = y_1, \ldots, y_m$ will denote vectors of the tuples x_i and y_j. α_i and β_j will denote a 0/1 assignment to the tuples x_i and y_j respectively, and $\vec{\alpha}, \vec{\beta}$ will each denote a 0/1 assignment to the vector of tuples $\vec{x}, \vec{y}$ respectively. We also define the *characteristic vector* of a set $S \subseteq [n]$ to be the binary vector $s \in \{0,1\}^n$ such that $s_i = 0$ for all $i \notin S$ and $s_i = 1$ for all $i \in S$.

The formula $\psi_{\mathcal{S}}$ will have variables $\vec{x}$ and $\vec{y}$ that will respectively encode n-by-m matrices M and N. The variables of $\vec{x}$ will define M such that each of the n rows of M is some vector in A, and the variables $\vec{y}$ will define N such that each of the m columns of N is the characteristic vector for some set S from the hitting set instance $\mathcal{S}$. In particular, x_i will indicate a vector in A to serve as the ith row of M, while y_j will indicate a set in $\mathcal{S}$ whose characteristic vector will serve as the jth column of N, with each x_i and y_j being chosen separately. For the remainder of the section, we restrict our attention to matrices M and N defined this way. We say that M and N *intersect* if $M[i,j] = N[i,j] = 1$ for some pair (i,j). $\psi_{\mathcal{S}}$ will be defined so that it is falsified whenever M and N intersect and satisfied otherwise.

Notice that when some column of M is the characteristic vector of a hitting set, $\psi_{\mathcal{S}}$ is falsified because there is no way to pick the corresponding column in N so that the two columns do not intersect. Conversely, if none of the columns in M represent a hitting set, then there is always a way to pick N so that $\psi_{\mathcal{S}}$ is satisfied (for each column we simply pick the set that was not hit). Therefore proving that $\psi_{\mathcal{S}}$ is unsatisfiable boils down to proving that for any choice of M, some column of M represents a hitting set.

Claim 68. *$\psi_{\mathcal{S}}$ is unsatisfiable when $\gamma(\mathcal{S}) \leq \frac{\log m}{4}$.*

Proof sketch. Let H be any hitting set of size at most $\frac{\log m}{4}$, which we interpret of as a set of row indices into M. By the $(m, (\log m)/4)$-dual-universality of A, any set I of at most $(\log m)/4$ strings from A has a location such that all the strings in I contain a 1 at that location.[8] Since rows of M are strings in A, taking $I = H$ there must exist a column j^* such that $M[i, j^*] = 1$ for every $i \in H$. Because H is a

[7] Many examples of universal sets (including the Paley graph construction) are discussed in [Juk12], as well as [NN93, AGHP92]. Alternate constructions use properties such as k-wise independent sample spaces and linear codes, and counting arguments for different parameter regimes exist. Notably the Paley construction fulfills our four essential properties of being small (of size m), polytime constructible, $(m, \log m/4)$-universal, and $(m, \log m/4)$-dual-universal.

[8] We do not require that the rows of M are *distinct* rows of A, but because we are only looking for a location with a 1 for every row this does not pose an issue. In fact we only ever use the universal and dual universal properties to search for a location with either all 0 or all 1, where repetition doesn't break the universal properties we need.

hitting set and the jth column of N is the indicator vector of a set $S \in \mathcal{S}$, there must be some $i^* \in H$ such that $N[i^*, j^*] = 1$, and so M and N intersect at (i^*, j^*). □

Next, we define the formula more formally. The variables of $\psi_\mathcal{S}$ are $\vec{x} = \{x_i \mid i \in [n]\}$ where x_i is a tuple of $\log m$ boolean variables, and $\vec{y} = \{y_j \mid j \in [m]\}$ where y_j is a tuple of $\log n$ boolean variables. Given an assignment $\vec{\alpha} = \{\alpha_i \mid i \in [n]\}$ to the $\vec{x}$-variables, $\vec{\alpha}$ encodes an n-by-m matrix $M_{\vec{\alpha}}$ where the i-th row of $M_{\vec{\alpha}}$ equals $a_{\alpha_i} \in A$ (interpreting α_i as an index in $[m]$). Similarly given an assignment $\vec{\beta} = \{\beta_j \mid j \in [m]\}$ to the $\vec{y}$-variables, $\vec{\beta}$ encodes an n-by-m matrix $N_{\vec{\beta}}$, where column j is the characteristic vector of the set $S_{\beta_j} \in \mathcal{S}$ (interpreting β_j as an index in $[n]$). We will sometimes write $M_{\vec{\alpha}}[i,j]$ as $M_{\alpha_i}[i,j]$ to stress that the ith row of $M_{\vec{\alpha}}$ is determined by α_i. Similarly, we will sometimes write $N_{\vec{\beta}}[i,j]$ as $N_{\beta_j}[i,j]$.

Lastly, we formally define the clauses in $\psi_\mathcal{S}$ so that it is falsified whenever $M_{\vec{\alpha}}$ and $N_{\vec{\beta}}$ intersect and satisfied otherwise.

Definition 26. For every $i \in [n]$ and $j \in [m]$, and for every pair of values $\alpha_i \in \{0,1\}^{\log m}$, $\beta_j \in \{0,1\}^{\log n}$ such that $M_{\alpha_i}[i,j] = 1$ and $N_{\beta_j}[i,j] = 1$, we have the clause $\overline{x_i^{\alpha_i} \wedge y_j^{\beta_j}}$ where $x_i^{\alpha_i} = \wedge_{t \in [n]} (x_i)_t^{(\alpha_i)_t}$ is the conjunction of all variables in x_i, each of which occurs positively when the corresponding bit of α_i is 1 and negatively when the corresponding bit of α_i is 0 (we define $y_j^{\beta_j}$ in the same way). This axiom is falsified iff x_i is assigned value α_i and y_j is assigned value β_j.

This formula has the property we want because if $M_{\vec{\alpha}}$ and $N_{\vec{\beta}}$ intersect at some location i, j, then the axiom $\overline{x_i^{\alpha_i} \wedge y_j^{\beta_j}}$ exists in $\psi_\mathcal{S}$ and would be falsified. Conversely, if $\psi_\mathcal{S}$ is falsified, then some axiom $\overline{x_i^{\alpha_i} \wedge y_j^{\beta_j}}$ is falsified, which means $M_{\vec{\alpha}}[i,j] = N_{\vec{\beta}}[i,j] = 1$.

It is easy to check that the number of variables in $\psi_\mathcal{S}$ is $n \log m + m \log n$. The number of clauses is at most $n^2 m^2$, since for each $i \in [n]$ and $j \in [m]$, each of the mn possible assignments to (x_i, y_j) adds at most one clause to $\psi_\mathcal{S}$.

Redundantly encoding $\psi_\mathcal{S}$

In order to prove our result we will need a way of proving both upper and lower bounds on $S_\mathcal{Q}(\psi_\mathcal{S})$, but it turns out that the lower bounds are difficult to prove if we use $\psi_\mathcal{S}$ as is. Thus, we will employ a standard trick in proof complexity, which is to redundantly encode the variables in the formula; more specifically we follow [AR08] and redundantly code blocks of variables, namely each row and column, using error-correcting codes. It is interesting to note that for our formulas, we are unable to prove even width lower bounds without the redundant encoding. In contrast, most proof complexity applications use this trick solely for the purpose of reducing size lower bounds to width lower bounds.

Definition 27. For $q, r, s \in \mathbb{N}$, a (q,r,s)-*code* is a total function f from $\{0,1\}^q$ to $\{0,1\}^r$ with the property that for any $\rho \in \{0,1,*\}^q$ such that ρ fixes at most s values to $\{0,1\}$, $f|_\rho$ is surjective on $\{0,1\}^r$. Efficiently computable constructions using linear codes are known for any $r, q = 6r, s = 2r$ (see e.g. [AR08]). We say that f is r-surjective.

Let $f_x : \{0,1\}^{6 \log m} \to [m]$ be a $(6 \log m, \log m, 2 \log m)$-code and let $f_y : \{0,1\}^{6 \log n} \to [n]$ be a $(6 \log n, \log n, 2 \log n)$-code. We will have a vector $x_i \in \{0,1\}^{6 \log m}$ for each $i \in [n]$ and a vector $y_j \in \{0,1\}^{6 \log n}$ for each $j \in [m]$. Given an assignment $\vec{\alpha}$ to all of the $\vec{x}$-variables, we will associate with $\vec{\alpha}$ an n-by-m matrix $M_{\vec{\alpha}}$, where the ith row of $M_{\vec{\alpha}}$ will be the vector $a_{f_x(\alpha_i)} \in A$. Similarly given an

assignment $\vec{\beta}$ to all of the $\vec{y}$-variables, we will associate with $\vec{\beta}$ an n-by-m matrix $N_{\vec{\beta}}$, where column j is the characteristic vector corresponding to the set $S_{f_y(\beta_j)} \in \mathcal{S}$ In other words, $N_{\vec{\beta}}[i,j]$ is 1 if and only if set $S_{f_y(\beta_j)}$ contains element i.

We now define our unsatisfiable CNF $\tau_{\mathcal{S}}$ in the same way as $\psi_{\mathcal{S}}$ using these redundant encodings. Note that it is unsatisfiable for exactly the same reason as stated before.

Definition 28. The clauses of $\tau_{\mathcal{S}}$ are defined as follows. For every $i \in [n], j \in [m]$ and for every pair of assignments (α_i, β_j) to (x_i, y_j) such that $M_{\alpha_i}[i,j] = 1$ and $N_{\beta_j}[i,j] = 1$, we have the clause $\overline{x_i^{\alpha_i} \wedge y_j^{\beta_j}}$.

In the redundant encoding we have $n \cdot 6 \log m$ x-variables and $m \cdot 6 \log n$ y-variables, for a total of $O(n \log m)$ variables when $m = n^{1/k} \ll n$. The number of clauses in $\tau_{\mathcal{S}}$ is at most $n^7 m^7$, since for each $i \in [n]$ and $j \in [m]$, each of the $m^6 n^6$ possible assignments to (x_i, y_j) adds at most one clause to $\tau_{\mathcal{S}}$. We also note that $\tau_{\mathcal{S}}$ can indeed be constructed in polynomial time, which is important to using Lemma 67 to get non-automatability results.

It may be instructive to note that both the upper and lower bounds we will shoot for on the size of proofs of $\tau_{\mathcal{S}}$ are exactly $n^{\Theta(\gamma(\mathcal{S})/k)} = m^{\Theta(\gamma(\mathcal{S}))}$, which is polynomial in the number of distinct assignments to $\alpha_1 \dots \alpha_{\gamma(\mathcal{S})}$, assuming without loss of generality that the minimum hitting set of $\mathcal{S}$ is the first $\gamma(\mathcal{S})$ elements $\{1 \dots \gamma(\mathcal{S})\} \subseteq [n]$.

A.3.3 Upper bound

In order to prove Lemma 67, we need to start by showing that an upper bound exists in the case that there exists a small hitting set. It suffices to prove the upper bound for tree-like Resolution, as all other systems discussed can *polynomially simulate* tree-Res.

The proof is just a formalization of the argument given in the proof of Claim 68. Recall from Lemma 16 that res-tree$(\tau_{\mathcal{S}})$ = dec-tree$(\tau_{\mathcal{S}})$, and so it suffices to give a decision tree solving the search problem for $\tau_{\mathcal{S}}$; that is, a decision tree (over the underlying variables of $\tau_{\mathcal{S}}$), where every leaf l is labelled with a clause of $\tau_{\mathcal{S}}$ that is falsified by the partial assignment that labels the path to l.

We will first show that if $\gamma(\mathcal{S}) \leq k$, then there is a height $2 \log n$ decision tree (and therefore size n^2) for the unencoded formula $\psi_{\mathcal{S}}$. Since $\gamma(\mathcal{S}) \leq k$, assume without loss of generality that $H = \{1, \dots, k\}$ is a valid hitting set for $\mathcal{S}$. The decision tree for $\psi_{\mathcal{S}}$ consists of two phases. First, the decision tree will branch on all of the Boolean variables in $x_1, \dots, x_k$. This will result in a full binary tree, call it T, of depth $k \log m$. In the second phase, at each leaf vertex of T we will query all of the variables of some y_j variable, where the choice of y_j will be a function of the path taken in T.

Consider some path in T leading to leaf $l_{\vec{\alpha}}$, corresponding to the assignment $\vec{\alpha} = \alpha_1, \dots \alpha_k$ for $x_1, \dots, x_k$. The assignment $\vec{\alpha}$ corresponds to an ordered set of strings $I \subseteq A$, where $|I| \leq k$. Since $k \in O(\log^{1/3} n)$ and $m = n^{1/k}$, $k \leq \frac{\log m}{4}$ for large n. By the $(m, \log m/4)$-dual-universal property of A there is some $j \in [m]$ such that I restricted to position j is all 1's, and thus $M_{\vec{\alpha}}[i,j] = 1$ for all $i \in [k]$. In the second phase, at this leaf vertex $l_{\vec{\alpha}}$ of T we will then query all of the Boolean variables in y_j. Let β_j be one partial assignment to these variables and consider the path labelled by $\vec{\alpha}\beta_j$ leading to the leaf vertex $l_{\vec{\alpha}\beta_j}$. Since $\{1, \dots, k\}$ is a hitting set for $\mathcal{S}$ we are guaranteed that $N_{\vec{\beta}_j}[i,j] = 1$ for at least one $i \in [k]$, and since $M_{\vec{\alpha}}[i,j] = 1$ for all $i \in [k]$, one of the clauses in $\tau_{\mathcal{S}}$ must be violated by the partial assignment $\vec{\alpha}, \beta_j$, so we label $l_{\vec{\alpha}\beta_j}$ with any such clause. The resulting decision tree thus solves the search problem associated with $\psi_{\mathcal{S}}$ and has height $k \log m + \log n = 2 \log n$.

The decision tree for the redundant version $\tau_{\mathcal{S}}$ is essentially the same but instead we query the redundant encodings of the variables. First, we query $x_1, \ldots, x_k$, resulting in a full binary tree of height $k \cdot 6 \log m$, and then, we query a particular y_j (depending on the path taken in T), which is $6 \log n$ variables, and thus the height is $k \cdot 6 \log m + 6 \log n = 12 \log n$.

A.3.4 Lower bound

The other side of Lemma 67, and naturally the much more difficult side, is the lower bound. To give a flavor of how we leverage our construction, as well as the standard techniques for proving lower bounds against Resolution, we prove this lower bound for Res. Note that this also gives a lower bound for Res.

We begin by proving a *wide clause lemma* for $\tau_{\mathcal{S}}$, which alone is enough to prove lower bounds for tree-Res (using the size-width relationship for tree-Res due to Ben-Sasson and Wigderson [BW01]); for general Res, we apply a standard application of random restrictions to reduce to width.

Our notion of "wide" will be a bit richer than the usual definition. For a clause D, let $I_0(D)$ be the set of all $i \in [n]$ for which there are at least $\log m$ literals in D that correspond to variables from x_i. Likewise let $J_0(D)$ be the set of all $j \in [m]$ for which there are at least $\log n$ literals in D that correspond to variables from y_j.

Wide Clause Lemma. *For sufficiently large n, if $\gamma(\mathcal{S}) > k^2$ and f_x (f_y) is $\log m$-surjective ($\log n$-surjective, respectively), then for any* Res *refutation π refuting $\tau_{\mathcal{S}}$ there exists a clause $D \in \pi$ such that $|I_0(D)| \geq k^2$ or $|J_0(D)| \geq k$.*

Proof. We follow the *prover-delayer game* of [Pud00, AD08] in the style of [ALN16]. The width-w game on an unsatisfiable formula τ is played between a Delayer, who is asserting that she has a satisfying assignment for τ, and a Prover, who is trying to force the Delayer into a contradiction by asking her values of the underlying variables. However, the Prover has limited memory and can only remember the values of up to w of the variables at a time.

Both players know τ and the contents of the Prover's memory, which is initially empty. At the start of each round there are at most $w - 1$ values in memory. The Prover asks the Delayer the value of some variable whose value is not currently in memory. The Delayer responds with an answer (either 0 or 1), and upon receiving the answer, the Prover adds this assignment to his memory (increasing the number of stored values by 1). He can then erase (forget) any existing values from memory, possibly decreasing the number of stored values. The Prover declares victory if at some point, the partial assignment written in his memory falsifies one of the clauses of τ. The Delayer has a winning strategy for the width-w game on τ if no matter how the Prover plays the game, he cannot win. It was shown [Pud00, AD08] that the Delayer has a winning strategy for the width-w game if and only if the Res width of τ is at least $w - 1$.

For our formula $\tau_{\mathcal{S}}$, the game proceeds as above, but now let D be the set of literals in the Prover's memory, and we demand instead of only holding w variables total in memory that $|I_0(D)| \leq k^2$ and $|J_0(D)| \leq k$. By the transformation from [Pud00], the Prover has a winning strategy for this game if there is a Res refutation such that $|I_0(D)| \leq k^2 - 1$ and $|J_0(D)| \leq k - 1$ for every clause D. Therefore the Delayer has a winning strategy for this game if and only if the lemma holds. The Delayer's winning strategy is as follows.

- If the Prover asks about a variable in x_i:
  - If $i \notin I_0(D)$ and after adding this bit there are still less than $\log m$ variables from x_i in memory, the Delayer can answer with either 0 or 1 arbitrarily.

  - If $i \notin I_0(D)$ but after adding this bit to memory there are now $\log m$ variables from x_i in memory, the Delayer uses the fact that $|J_0(D)| \leq k \leq \log m/4$ and the $(m, \log m/4)$-universal property of A to find a string $a_0 \in A$ such that $a_0|_{J_0(D)}$ is the all-zeros string, and uses the surjective property of f_x to find an assignment α_i consistent with the assignment to the x_i variables in memory such that $f_x(\alpha_i) = a_0$. The Delayer will remember the assignment α_i for x_i from now on, and note that $I_0(D)$ now contains i.
  - Finally if $i \in I_0(D)$ then the Delayer is maintaining an assignment α_i for x_i, so she answers according to α_i.

- If the Prover asks about a variable in y_j:
  - If $j \notin J_0(D)$ and after adding this bit there are still less than $\log n$ variables from y_j in memory, the Delayer can answer with either 0 or 1 arbitrarily.
  - If $j \notin J_0(D)$ but there are now $\log n$ variables from y_j in memory, the Delayer uses the fact that $|I_0(D)| \leq k^2 < \gamma(\mathcal{S})$ and finds a set S_0 that doesn't contain any element $i \in I_0(D)$, and uses the surjective property of f_y to find an assignment β_j consistent with the assignment to the y_j variables in memory such that $f_y(\beta_j) = S_0$. The Delayer will remember the assignment β_j for x_j, and note that $J_0(D)$ now contains j.
  - Finally if $j \in J_0(D)$ then the Delayer is already maintaining an assignment β_j for y_j, so she answers according to β_j.

- Whenever the Prover erases a variable from x_i from his memory, if $i \in I_0$ and now there are less than $\log m$ variables from x_i in memory, the Delayer forgets α_i. (note that i is no longer in I_0) Similarly, whenever the Prover erases a variable from y_j from his memory, if $j \in J_0$ and now there are less than $\log n$ variables from y_j in memory, the Delayer removes β_j from J_0. (note that j is no longer in J_0)

Assume for contradiction the game ends with the Prover winning. Consider when the game ends, and say the Prover claims the axiom $\overline{x_i^{\alpha_i} \wedge y_j^{\beta_j}}$ was falsified, and thus that $M_{\vec{\alpha}}[i,j] = N_{\vec{\beta}}[i,j] = 1$. First, consider the case when either $i \notin I_0$ or $j \notin J_0$. In either case there are is at least one variable in the axiom that is not in memory, which means that it has not been falsified, which is a contradiction. So assume that $i \in I_0$ and $j \in J_0$, and consider the last time that i was added to I_0 and the last time that j was added to J_0. Assume that i was added after j. Since j was in J_0 at the time we defined α_i, $M_{\alpha_i}[i,j] = 0$ by our choice of α_i, which is a contradiction. Finally assume that j was added after i. Then since i was in I_0 at the time we defined β_j, $f_y(\beta_j)$ does not contain i, and so $N_{\beta_j}[i,j] = 0$, which is also a contradiction. □

Before proceeding on to the lower bound, we need to change Wide Clause Lemma slightly, in order to be able to apply a restriction argument to turn width lower bounds into size lower bounds for $\tau_{\mathcal{S}}$. We use the notation $f|_\rho$ to denote the *restriction* of the function f over $x_1 \dots x_s$ by $\rho \in \{0,1,*\}^s$, which is the function f over the variables x_i for all $i \in \rho^{-1}(*)$ obtained by setting all other variables x_j to $\rho(j)$. Likewise we use the notation $\tau|_\rho$ to denote the restriction of the tautology τ by ρ.

Definition 29. Let $\rho_{x_i} \in \{0,1,*\}^{x_i}$ and let $\rho_{y_j} \in \{0,1,*\}^{y_j}$. Furthermore, let $\mathcal{R}$ be the set of all $\vec{\rho} = \{\rho_{x_1} \dots \rho_{x_n}, \rho_{y_1} \dots \rho_{y_m}\}$, such that for all $i \in [n]$ and $j \in [m]$, $|\rho_{x_i}^{-1}(*)| = 5 \log m$ and $|\rho_{y_j}^{-1}(*)| = 5 \log n$.

Let f_x^i be the function f_x on the variables $\rho_{x_i}^{-1}(*)$ after restricting all other inputs to ρ_{x_i}, and likewise for f_y^j.

Lemma 69 (Wide Clause Lemma under restrictions)**.** *For sufficiently large n and $\rho \in \mathcal{R}$, if $\gamma(\mathcal{S}) > k^2$ then for any* Res *refutation π refuting $\tau_{\mathcal{S}}|_{\vec{\rho}}$ there exists a clause $D \in \pi$ such that $|I_0(D)| \geq k^2$ or $|J_0(D)| \geq k$.*

We omit the proof of Lemma 69, as it is essentially identical to Wide Clause Lemma. The only difference is that in each row i the Delayer chooses α_i based on f_x^i instead of f_x, and likewise for the columns. Note that f_x was $2 \log m$ surjective before the restriction, and since only $\log m$ variables are fixed in every row f_x^i is still $\log m$ surjective (and similarly for f_y^j).

We can now complete our lower bound. Let π be a Res refutation of $\tau_{\mathcal{S}}$ and assume for contradiction that $|\pi| < n^{k/16}$. First, consider a clause $D \in \pi$ such that $|I_0(D)| \geq k^2$. For each $i \in I_0(D)$, the chance that a randomly chosen $\vec{\rho} \in \mathcal{R}$ doesn't set one of the x_i literals in D to 1 is less than $(1 - (\frac{1}{6} \cdot \frac{1}{2}))^{\log m}$. Thus the probability that no $i \in I_0(D)$ sets D to 1 is at most $(\frac{11}{12})^{k^2 \log m} = (\frac{11}{12})^{k \log n} < \frac{1}{n^{k/8}}$. By a union bound the probability that some clause D in π satisfying $|I_0(D)| \geq k^2$ is not set to 1 is less than $\frac{n^{k/16}}{n^{k/8}} = \frac{1}{n^{k/16}}$, using the fact that $|\pi| < n^{k/16}$.

Similarly the probability that some clause $D \in \pi$ satisfying $|J_0(D)| \geq k$ is not set to 1 is at most $\frac{1}{n^{k/16}}$. Thus with probability at least $1 - \frac{2}{n^{k/16}}$, all clauses D satisfying $|I_0(D)| \geq k^2$ or $|J_0(D)| \geq k$ are set to 1 by a random restriction, and thus there exists a restriction $\vec{\rho} = \{\rho_{x_1} \ldots \rho_{x_n}, \rho_{y_1} \ldots \rho_{y_m}\}$ setting all such clauses to 1. However, this contradicts Lemma 69, as $\tau_{\mathcal{S}}|_{\vec{\rho}}$ must still have at least one such clause. Thus $S_{\mathcal{Q}}(\tau_{\mathcal{S}}) \geq n^{c_l k}$ for $c_l = \frac{1}{16}$.