DETERMINISTIC SIMULATIONS AND HIERARCHY THEOREMS FOR RANDOMIZED ALGORITHMS

by

Jeffrey J. Kinne

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2010

ABSTRACT

In this dissertation, we present three research directions related to the question whether all randomized algorithms can be derandomized, i.e., simulated by deterministic algorithms with a small loss in efficiency.

Typically-Correct Derandomization A recent line of research has considered "typically-correct" deterministic simulations of randomized algorithms, which are allowed to err on few inputs. Such derandomizations may be easier to obtain and/or be more efficient than full derandomizations that do not make mistakes. We further the study of typically-correct derandomization in two ways.

First, we develop a generic approach for constructing typically-correct derandomizations based on seed-extending pseudorandom generators, which are pseudorandom generators that reveal their seed. We use our approach to obtain both conditional and unconditional typically-correct derandomization results in various algorithmic settings. For example, we present a typically-correct polynomial-time simulation for every language in BPP based on a hardness assumption that is weaker than the ones used in earlier work.

Second, we investigate whether typically-correct derandomization of BPP implies circuit lower bounds. We establish a positive answer for small error rates and in doing so provide a proof for the zero-error setting that is simpler and scales better than earlier arguments.

Monotone Computations Short of derandomizing all efficient randomized algorithms, we can ask to derandomize more restricted classes of randomized algorithms. Because a strong connection has been proved between circuit lower bounds and derandomization, and

there has been success proving worst-case circuit lower bounds for monotone circuits, randomized monotone computations are a natural candidate to consider. We show that, in fact, any derandomization of randomized monotone computations would derandomize all randomized algorithms, whether monotone or not. We prove similar results in the settings of pseudorandom generators and average-case hard functions – that a pseudorandom generator secure against monotone circuits is also secure with somewhat weaker parameters against general circuits, and that an average-case hard function for monotone circuits is also hard with somewhat weaker parameters for general circuits.

Hierarchy Theorems For any computational model, a fundamental question is whether machines with more resources are strictly more powerful than machines with fewer resources. Such results are known as hierarchy theorems. The standard techniques for proving hierarchy theorems fail when applied to bounded-error randomized machines and for other so-called "semantic" models of computation for which a machine must satisfy some promise to be valid. If all randomized algorithms can be efficiently derandomized in a uniform way, hierarchies for bounded-error randomized algorithms would follow from the deterministic hierarchy theorems. But can hierarchies be proved short of proving derandomization?

A recent line of work has made progress by proving *time* hierarchies for randomized and other semantic models that use one bit of advice. We adapt the techniques to prove results in the setting of *space*, proving space hierarchy results that are as tight as possible for typical space bounds between logarithmic and linear for randomized and other semantic models that use one bit of advice.

ACKNOWLEDGMENTS

I feel that a key part of being human is to seek the good of our community – family, friends, workplace, city, etc. My life has been full of others who share this view. In particular, I owe the completion of this dissertation to the many who provided support, encouragement and more. In acknowledging some of these people I give them thanks, but also I encourage all of us to echo the best of their attributes.

Let me begin with my advisor Dieter. When I think of the qualities I would most want in an ideal advisor, I think of: honesty, integrity, selflessness, openness, dedication, patience, and having the highest standards for research and teaching. Those who know Dieter will realize I have just described him! Whether in research or teaching, Dieter always places the best interests of the students first – even if that means extending class by "a few" minutes to finish the day's topic or going through "a few" revisions of a paper to achieve the desired quality ;). Dieter, thank you so much for everything. It has been an absolute pleasure working together.

Next, my wife Devon is the person most responsible for me seeking a PhD in the first place and for finishing the task. We have truly lived a dream together. From the time we began dating in high school, we mapped out what our life would look like. Our predictions were amazingly accurate, in large part to Devon's love and perseverance! She has always placed others' interests first (the children, myself, her students, the children's school, etc.) before her own. Our family has thrived due to her love, dedication, kindness, strong values, sincerity, graciousness, and humility. For example, on the day of my defense Devon dropped off Andrew and William at school, lugged Matthew around picking up and setting out refreshments, stayed for most of the defense while holding a sleeping Matthew, then appropriately discretely

left the room after Matthew woke up reaching for me with a smile and "da-da, da-da, da-da". I thank you eternally Devon, you truly are the better half. As for the kids, I am grateful for them for many reasons, not the least of which is their sense of wonder and inquisitiveness that is so contagious.

Thank you to the members of my committee – Eric Bach, Jin-Yi Cai, Shuchi Chawla, Steffen Lempp, and Lance Fortnow. Besides graciously helping me now at the end, you have been teachers to me in the classroom, have shared insights into the research process and academia, have written letters of support, and served on my preliminary defense committee. I would not be where I am today without you. I also very much appreciate the mentoring and encouragement from those who I have been a teaching assistant for – Dieter, John Strikwerda, and Rebecca Hasti.

I thank my co-authors Dieter and Ronen Shaltiel, who have exemplified the way research ought to be done – openly and with the true goal of bettering the community. These co-authors and a few others, notably Scott Diehl and Matt Anderson, have been invaluable as people to bounce ideas off of and discuss research.

Life as a graduate student would be far less enjoyable without the camaraderie of others to share successes and failures. Thanks go to Randy Smith and Margaret Richey for being good friends that helped us adjust to life in Madison and as graduate students. I also thank all of the theory students, which at various times included Denis Charles, Venkat Chakravarthy, Chris Kaiserlian, Rakesh Kumar, Yunpeng Li, Aparna Das, Giordano Fusco, Bess Berg, Eric Skaug, Mike Kowalczyk, Scott, Matt, Jurgen Van Gael, Anand Sinha, Vinay Choudhary, Tom Watson, Siddarth Barman, David Malec, William Umboh, Dalibor Zelený, Adeel Pervez, Baris Aydinlioglu, Balu Sivan, Tyson Williams, and Thea Hinkle. I will miss the lunches, reading groups, taking breaks to watch the construction out the window, trips to conferences together, practice presentations, etc. I will miss the conversations about all manner of topics – trying to solve homework problems, studying for the qual, sports, Indian versus American culture, religion, politics, child-raising strategies, "inventions", "points", "memorials" to past theory students, etc.

My life as a whole has been influenced by many many kind and generous people. Foremost are my parents Tim and Cathy Kinne, but this also includes the rest of my family, coworkers, and friends from Madison, Xavier, and St. X. A special thanks goes to the CS professors at Xavier – Michael Goldweber, Gary Lewandowski, and Liz Johnson – for encouraging my spark of interest in computer science and ultimately sending me off to graduate school! Thank you to all of you!

Finally, this research would not have been possible without the financial support of the University of Wisconsin-Madison, National Science Foundation grants CCR-0133693 and CCR-0728809, and a Cisco Systems Distinguished Graduate Fellowship.

TABLE OF CONTENTS

Page

ABSTRACT i

1 Introduction 1

1.1 The Power of Randomized Algorithms 5
1.2 Typically-Correct Derandomization 8
1.2.1 Applications of Our Approach 10
1.2.2 Typically-Correct Derandomization and Circuit Lower Bounds 11
1.3 Derandomization of Monotone Computations 13
1.3.1 Our Results 15
1.4 Space Hierarchy Theorems 17
1.4.1 Randomized Models with Advice 19
1.4.2 Generic Semantic Models with Advice 21
1.4.3 Promise Problems for Generic Semantic Models 22
1.5 Organization 23

2 Preliminaries 25

2.1 Deterministic Algorithms and Turing Machines 25
2.2 Randomized Algorithms and Turing Machines 28
2.2.1 Error Reduction 31
2.2.2 Deterministic Simulations 32
2.3 Nondeterministic and Unambiguous Machines 33
2.4 Other Randomized Models and Quantum Machines 34
2.5 Distance and Hardness 35
2.6 Circuits 36
2.6.1 Hardness Amplification 38
2.7 Semantic Models of Computation 39
2.8 Promise Problems 41

3 Hierarchy Theorems for Generic Semantic Models . . . 42

3.1 Promise Problems . . . 48
3.1.1 Proof of Theorem 1.12 . . . 49
3.2 Semantic Models with One Bit of Advice . . . 51
3.2.1 Delayed Diagonalization on Semantic Models with Advice . . . 53
3.2.2 Analysis . . . 57
3.2.3 Generic Semantic Models . . . 64

4 Hierarchy Theorems for Randomized Models . . . 67

4.1 Proof Outline . . . 69
4.2 The Need for Advice and Recovery Procedures . . . 70
4.3 Two-sided Error Recovery Procedure – Computation Tableau Language . . . 73
4.4 Zero-sided error Recovery Procedure – Configuration Reachability . . . 78
4.5 The Final Construction . . . 83
4.6 Analysis . . . 85
4.6.1 Theorems 1.7 and 1.8 . . . 85
4.6.2 Corollaries 4.1 and 4.2 . . . 87
4.6.3 Additional Remarks . . . 88

5 Typically-Correct Derandomization . . . 91

5.1 Typically-Correct Derandomization and the PRG Approach . . . 91
5.1.1 Notation and Concepts . . . 91
5.1.2 The Seed-Extending Pseudorandom Generator Approach . . . 93
5.1.3 Hardness-Based Constructions of Seed-Extending Generators . . . 94
5.1.4 Analysis of the Nisan-Wigderson Construction . . . 96
5.2 Conditional Results . . . 102
5.2.1 Bounded-Error Polynomial Time . . . 102
5.2.2 Extensions to Other Algorithmic Settings . . . 104
5.3 Unconditional Results . . . 108
5.3.1 Constant-Depth Circuits . . . 108
5.3.2 Constant-Depth Circuits with Few Symmetric Gates . . . 109
5.3.3 Multi-Party Communication Complexity . . . 111
5.4 Comparison with the Extractor-Based Approach . . . 115

6 Typically-Correct Derandomization and Circuit Lower Bounds . . . 120

6.1 Circuit Lower Bounds . . . 120
6.1.1 Results . . . 121
6.1.2 Proof for the Typically-Correct Setting . . . 123
6.1.3 Proofs for the Everywhere-Correct Setting . . . 127

6.1.4 Extensions 130
6.2 Relativization and Algebrization 130

7 Derandomizing Monotone Computations 133

7.1 Monotone Slice Functions 133
7.1.1 Proof Overviews 137
7.2 Average-Case Hardness 138
7.2.1 Monotone Hard Functions 142
7.3 Pseudorandom Generators 144
7.4 Derandomization 148

LIST OF REFERENCES 153

Chapter 1

Introduction

Computational complexity asks how efficiently problems can be solved on computers. For some problems we know of efficient solutions, and for others we know that there can be no efficient solution. For many important problems, we do not yet know whether there exist fast algorithms to solve the problem or not, and it is on these problems that we focus our attention. In particular, we focus on the class of problems that admit efficient "randomized algorithms", and we ask whether these algorithms can be "derandomized" – converted into deterministic algorithms without much loss in efficiency.

Dialog on Identity Testing There are some problems for which we know of fast algorithms that use random bits, but we do not yet know if the random bits are truly necessary to solve these problems efficiently. We begin by introducing one such problem, polynomial identity testing, and the concept of a randomized algorithm through a lesson that a teacher devises to engage a pupil in this fascinating topic.

Teacher: Today we study an incredibly exciting topic, algebraic identities!

Pupil: [*While texting*] uh huh.

Teacher: No cell phones today, the world of mathematics will keep us company.

Pupil: Alright, let's get this over with.

Teacher: Very well, let us start off easy for you. Please factor the polynomial $x^2 - 1$.

Pupil: Easy, $(x + 1) \cdot (x - 1)$.

Teacher: Prove it.

Pupil: We can use the properties of the real numbers to show that if we multiply this out...

Teacher: Yes, that is one way to do it. How would you prove
$(x-1)\cdot(x+1)\cdot(x^2-2)\cdot(x^2+2)\cdot(x^3-3)\cdot(x^3+3) = x^{12}-5x^6-x^{10}+9x^4-4x^8+36x^2-36$?

Pupil: Plug it into my calculator.

Teacher: Then you are trusting whoever programmed the calculator, not good enough.

Pupil: Do I really have to apply the properties of real numbers to show that multiplying out that big mess has that result? That seems like a lot of work.

Teacher: I agree with you.

Pupil: What? I thought your job was to make me do busy work.

Teacher: No, I want you to *think*. What if you move everything to the left-hand side.

Pupil: If you are not lying to me, then
$(x-1)\cdot(x+1)\cdot(x^2-2)\cdot(x^2+2)\cdot(x^3-3)\cdot(x^3+3)-(x^{12}-5x^6-x^{10}+9x^4-4x^8+36x^2-36) = 0.$

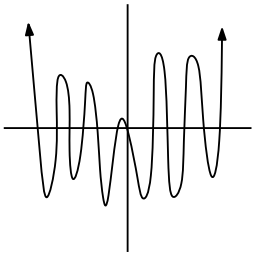
Univariate Polynomial Identity Testing

Teacher: Let us call the left hand side $p(x)$. If I am telling the truth, then p is identically 0, so $p(x) = 0$ for all x. And what does its graph look like?

Pupil: Straight horizontal line on the x axis.

Teacher: And what if I lied to you?

Pupil: [*Thinking, thinking...*] Well, it is some kind of degree 12 polynomial. It can have at most 12 roots, so it might look something like this.



Teacher: So to verify my claimed identity you need to determine whether p is the zero polynomial or some other degree at most 12 polynomial.

Pupil: [*Thinking, thinking...*] I can evaluate the polynomial on 13 different points. If you were telling the truth, $p(x) = 0$ for all the points I choose. If you made a mistake, at least one of those 13 points will evaluate to non-zero. That is so cool! And it is so much easier than multiplying out the polynomial!

Teacher: Indeed. Are you interested in continuing this discussion longer?

Pupil: You got me teacher, I am interested now.

Multi-variate Polynomial Identity Testing

Teacher: Okay, how will you verify this identity?
$(u^2 + v^2 + x^2 + y^2)^2 = (u^2 + v^2 - x^2 - y^2)^2 + (2ux + 2vy)^2 + (2uy - 2vx)^2$

Pupil: Easy, I look at the polynomial
$p(u, v, x, y) = (u^2 + v^2 + x^2 + y^2)^2 - (u^2 + v^2 - x^2 - y^2)^2 - (2ux + 2vy)^2 - (2uy - 2vx)^2$.
The total degree of any monomial in the expanded polynomial would be at most 4, so I just evaluate p on 5 different points and see if they all evaluate to 0. Right?

Teacher: Try the points $(0, 0, 0, 1)$, $(0, 0, 0, 2)$, $(0, 0, 0, 3)$, $(0, 0, 0, 4)$, and $(0, 0, 0, 5)$. And also try them on the polynomial $q(u, v, x, y) = p(u, v, x, y) + u \cdot v \cdot x \cdot y$.

Pupil: Both p and q evaluate to 0 on all 5 points. [*Thinking, thinking, ...*] That cannot be right. Since $q(u, v, x, y) - p(u, v, x, y) = u \cdot v \cdot x \cdot y$ is a non-zero polynomial, at least one of p or q should be non-zero too. What am I missing?

Teacher: Is it true that a *multi-variate* polynomial of total degree at most 4 has at most 4 roots?

Pupil: Oh, I see. That is only true for single variable polynomials.

Teacher: But all is not lost. Think up a few random points to plug into the polynomials, and see what you get.

Pupil: I will try $(0, 1, 2, 3)$, $(5, 2, 10, 100)$, and $(3, 3, 3, 3)$. I see that p evaluates to 0 on all of these points, but q evaluates to non-zero on the last two. So at least I know q is non-zero. But what about p? Can we do better than just picking points at random to try?

Teacher: You could just multiply everything out...

Pupil: But that would take sooo long.

Teacher: If you can figure out a significantly faster way to verify polynomial identities then make sure to let me know! In the meantime, let us see what we can say about the strategy of picking points at random to plug into the polynomial.

Pupil: [*Thinking, thinking...*] If the polynomial is not zero, then it evaluates to non-zero on at least one point. But can we say it evaluates to non-zero on *most* points?

A Randomized Algorithm

Teacher: Yes! Suppose the polynomial is non-zero with total degree at most d. If we pick each variable at random from a set S, then the probability we are unfortunate and land on one of the roots of the polynomial is at most $\frac{d}{|S|}$.

Pupil: Cool. [*Thinking, thinking...*] I guess we can prove that by induction on the number of variables, with single variable polynomials being the base case?

Teacher: [*Wipes away a tear.*] I am so proud of you, that's right. So what do we know?

Pupil: We have a fast randomized algorithm to test polynomial identities. It makes a mistake with only very small probability as long as we choose our variables at random from a very large set S. This is pretty cool! Back to my earlier question, can we get rid of the randomness? And are there other problems we can solve with neat randomized algorithms? What else can I learn about randomized algorithms today?

Teacher: I am glad you are so excited about this now. Now that I have you motivated, you can learn more on your own and we can discuss your findings next time we meet. To start

off with, here is a dissertation that explores randomized algorithms from a more generic perspective...

1.1 The Power of Randomized Algorithms

The teacher in the above dialog has drawn the pupil into the intriguing world of randomized algorithms, the main focus of this dissertation. Randomness has been a valuable algorithm design tool, and intriguingly, many important problems can be solved by randomized algorithms that are either much more efficient or much simpler to implement than the best-known deterministic algorithms. The example from the teacher and pupil dialog, polynomial identity testing, is an example of both – the best-known deterministic algorithms are much more complicated and much less efficient than the simple randomized algorithm described in the dialog. The important question is the following.

> Are randomized algorithms truly more powerful than deterministic algorithms, or can every randomized algorithm be *derandomized* – converted into a deterministic algorithm without much loss in efficiency?

Early canonical examples of problems solvable much more efficiently with randomness than without include primality testing and connectivity on undirected graphs: relatively simple randomized algorithms solve primality testing in polynomial time [Mil76, Rab80] and undirected connectivity in logarithmic memory space [AKL+79]. In two famous separate works, both of these problems have been derandomized – Agrawal et al. [AKS04] giving a deterministic polynomial-time algorithm for primality and Reingold [Rei08] giving a deterministic logarithmic-space algorithm for undirected connectivity. Whether polynomial identity testing can be efficiently derandomized remains an open and much-studied question (see [Sax09] for a survey of recent progress).

General-Purpose Derandomization The derandomization results for primality testing and undirected connectivity employ techniques that are very specific to the problems being solved. A natural question is whether there are more generic methods that can be applied

to any randomized algorithm. Let $M(x, r)$ be a randomized polynomial-time machine that takes input x, uses random bits r, and outputs a certain Boolean value $f(x)$ for most choices of r but may output $\neg f(x)$ for some choices of r. The trivial deterministic simulation of M outputs the majority vote over all random strings and takes time exponential in the number of random bits. A more efficient simulation follows if we can shrink the number of random bits needed by constructing a pseudorandom generator G secure against M, a function that takes a short "seed" s and outputs a longer "pseudorandom string" $G(s)$ with the property that algorithms with similar complexity to M cannot tell the difference between the uniform distribution on pseudorandom strings and the uniform distribution on all strings. If the pseudorandom generator has logarithmic seed length then there are polynomially many pseudorandom strings to consider, and the algorithm $\text{Majority}_y(M(x, G(y))$ that takes the majority vote over the pseudorandom strings is a polynomial-time deterministic simulation that outputs the correct value $f(x)$.

A long line of research (see [Mil01] for an introduction) has shown that a very reasonable complexity-theoretic hardness assumption can be used to construct such pseudorandom generators sufficient to derandomize all time-efficient randomized algorithms, and in particular sufficient to yield an efficient deterministic algorithm for polynomial identity testing. The hardness assumption states that there is a problem that can be solved in time $2^{\Theta(n)}$ but cannot be solved by a family of circuits that uses only $2^{o(n)}$ gates for inputs of length n. We call such a hardness condition a "circuit lower bound". Given the hardness assumption, all problems solvable by polynomial-time randomized algorithms that have error bounded by a constant less than one half on every input, called BPP problems, can be solved in polynomial time on deterministic machines. More concisely, given the hardness assumption, BPP = P.

In practice, this approach to derandomization may be too inefficient because the randomized algorithm needs to be executed once for each of a polynomial number of pseudorandom strings. Further, the circuit lower bound hardness assumption, though plausible and widely believed in the community, has been notoriously difficult to prove and recent work [KI04] has

shown that in fact any non-trivial derandomization of polynomial identity testing implies circuit lower bounds that will likely be difficult to prove.

Making Progress Towards Full Derandomization This dissertation contains three main research directions that are aimed at making progress towards full derandomization of efficient randomized algorithms. By full derandomization, we mean deterministic simulations that are correct on all inputs. One question is what can be accomplished short of proving full derandomization. Each of the three research directions contained in this thesis aims to answer important open questions that would follow easily from full derandomization but nonetheless have proved elusive in their own right. We hope that making progress on these intermediate goals can shed light on fundamental properties of randomized algorithms that could be a part of a solution to the ultimate question of the power of randomized algorithms.

- A recent line of research has considered the possibility of "typically-correct" derandomizations – deterministic simulations that are allowed to make a small number of mistakes. Whereas previous approaches were based on extractors, we develop a new approach to typically-correct derandomization based on pseudorandom generators. We use the new approach to prove unconditional results for a number of classes of algorithms and also prove a conditional result for all time-efficient randomized algorithms that is based on a weaker hardness assumption than previous work. We also initiate the study of whether typically-correct derandomization implies circuit lower bounds, showing that this is indeed true for small error rates.

  These results are introduced further in Section 1.2

- Our results on typically-correct derandomization use a paradigm that utilizes circuit lower bounds to construct pseudorandom generators for the purpose of derandomization. One area where strong worst-case circuit lower bounds are known is that of monotone functions. We study the possibility of using these circuit lower bounds to derandomize monotone computations. We show that derandomization of randomized

monotone computations would imply derandomization of general non-monotone randomized computations. We show similar results for pseudorandom generators and average-case hard functions – that a pseudorandom generator secure against monotone circuits is also pseudorandom against general circuits and that a function average-case hard for monotone circuits is also average-case hard for general circuits.

These results are introduced further in Section 1.3

- Finally, we consider hierarchy theorems for randomized computations. If indeed randomized algorithms can be derandomized in a uniform way, then good time and space hierarchy theorems for randomized algorithms would follow from the deterministic time and space hierarchies. Thus hierarchies for randomized algorithms are a necessary step towards proving general-purpose derandomization. A recent line of work has made progress by proving *time* hierarchy theorems for randomized and other models of computation that use one bit of advice. We use similar techniques to prove *space* hierarchy theorems for randomized and other models of computation that use one bit of advice.

  These results are introduced further in Section 1.4

1.2 Typically-Correct Derandomization

The ultimate goal in the study of derandomization is to obtain deterministic simulations that are always correct and efficient. An intermediate goal has been studied in which the deterministic simulation is allowed to err on some inputs. Impagliazzo and Wigderson were the first to consider derandomizations that succeed with high probability on any efficiently samplable distribution; related notions have subsequently been investigated in [Kab01, TV07, GSTS03, SU07]. Goldreich and Wigderson [GW02] introduced a weaker notion in which the deterministic simulation only needs to behave correctly on most inputs of any given length. We refer to such simulations as "typically-correct derandomizations". The hope is to construct typically-correct derandomizations that are more efficient than the best-known

everywhere-correct derandomizations, or to construct them under weaker assumptions than the hypotheses needed for everywhere-correct derandomization.

A number of works have continued the study initiated by [GW02] in the standard setting of time-bounded randomized algorithms and in other settings [MS05, Zim08, Sha09]. Each of these works takes an approach suggested by Goldreich and Wigderson [GW02] of obtaining typically-correct derandomizations by "extracting randomness from the input". An extractor E is a procedure that takes a source of imperfect randomness and produces a distribution close to uniform. To derandomize an algorithm $M(x, r)$ taking input x and randomness r, $r' = E(x)$ is extracted in a deterministic way such that $D(x) = M(x, E(x))$ behaves correctly on most inputs. The works diverge in the analysis and the conditions under which D indeed makes a small number of mistakes.

Our Approach In this dissertation we develop an alternative generic approach for constructing typically-correct derandomizations. The approach builds on "seed-extending pseudorandom generators" rather than "extractors". A seed-extending pseudorandom generator is a generator G which outputs the seed as part of the pseudorandom string, i.e., $G(s) = (s, E(s))$ for some function E. The well-known Nisan-Wigderson pseudorandom generator construction [NW94] can easily be made seed-extending. We show that whenever a seed-extending pseudorandom generator passes certain statistical tests defined by the randomized procedure $M(x, r)$, the deterministic procedure $D(x) = M(x, E(x))$ forms a typically-correct derandomization of M, where the error rate depends on the error probability of the original randomized algorithm and on the error of the pseudorandom generator.

Note that this approach differs from the standard use of pseudorandom generators in derandomization, where the pseudorandom generator G is run on every seed. As the latter induces a time overhead that is exponential in the seed length, one aims for pseudorandom generators that are computable in time exponential in the seed length. A polynomial-time simulation is achieved only in the case of logarithmic seed lengths. In contrast, we run G *only once*, namely with the input x of the randomized algorithm as the seed. We use the

pseudorandom generator to *select* one "coin toss sequence" $r = E(x)$ on which we run the randomized algorithm. As opposed to the traditional derandomization setting, our approach benefits from pseudorandom generators that are computable in time less than exponential in the seed length. With a pseudorandom generator computable in time polynomial in the output length, we obtain nontrivial polynomial-time typically-correct derandomizations even when the seed length is superlogarithmic, and indeed any subpolynomial seed length suffices.

1.2.1 Applications of Our Approach

One of the main advantages of our approach over previous approaches to typically-correct derandomization is that we can rely on weaker hardness assumptions. In some settings we derive *conditional* results that are based on weaker hardness assumptions than the earlier works, and in some settings we even obtain new *unconditional* results because suitable hard functions are known to exist.

For the setting of randomized bounded-error polynomial time algorithms we obtain the following typically-correct derandomization based on a function that is *mildly* hard on average for small circuits. In the following, we say that D computes L to within $\frac{1}{n^c}$ if D is correct, i.e., $D(x) = L(x)$, on all but a $\frac{1}{n^c}$ fraction of inputs at length n; H is $\frac{1}{n^c}$-hard for circuits of size n^d if no circuit of size n^d computes H to within $\frac{1}{n^c}$.

THEOREM 1.1 *Let L be a language that is computed by a randomized bounded-error polynomial-time machine M. For any positive constant c, there is a positive constant d (depending on c and the running time of M) such that the following holds. If there is a language H in P that is $\frac{1}{n^c}$-hard for circuits of size n^d, then there is a deterministic polynomial-time machine D that computes L to within $\frac{1}{n^c}$.*

[GW02] and [Sha09] also prove conditional typically-correct derandomization results for BPP, but both require hardness conditions stronger than that of Theorem 1.1.

We can similarly relax the hardness assumption in a host of other settings. For some settings, for example Arthur-Merlin protocols and space-bounded algorithms, we obtain conditional typically-correct derandomization results based on reasonable hardness conditions.

Unconditional Results In some settings, using our approach allows us to establish new *unconditional* typically-correct derandomizations, namely for models where functions that are *very* hard on average are not known but functions which are *mildly* hard on average are known unconditionally. One such model is that of constant-depth circuits that are allowed a small number of arbitrary symmetric gates, i.e., gates that compute functions which only depend on the Hamming weight of the input, such as parity and majority.

THEOREM 1.2 *Let L be a language and M a uniform randomized circuit of constant depth and polynomial size that uses $o(\log^2 n)$ symmetric gates such that M computes L with error at most ρ. Then there is a uniform deterministic circuit D of constant depth and polynomial size that uses exactly the same symmetric gates as M in addition to a polynomial number of parity gates such that D computes L to within $3\rho + \frac{1}{n^{\Omega(\log n)}}$.*

We also derive an unconditional typically-correct derandomization result for multi-player randomized communication protocols.

Comparison with the Extractor-Based Approach [Sha09] introduced a generic approach to typically-correct derandomization based on "extractors for recognizable distributions" and applied it to achieve unconditional results in a number of settings: streaming algorithms, decision trees, 2-party communication protocols, and constant-depth circuits. We demonstrate an interesting relationship between our approach and the extractor-based approach. Ours is a strict generalization in that each of the results of [Sha09] can be obtained using our approach while some of our results cannot be proved using the extractor-based approach.

1.2.2 Typically-Correct Derandomization and Circuit Lower Bounds

Hardness versus randomness tradeoffs have shown that strong enough circuit lower bounds imply pseudorandom generators. The converse is also known, so that obtaining pseudorandom generators strong enough to derandomize BPP is equivalent to proving circuit lower

bounds which seem beyond the scope of current techniques. We may hope to derandomize BPP algorithms in some other way that does not imply circuit lower bounds. But Kabanets and Impagliazzo [KI04] showed that *any* subexponential-time derandomization of BPP implies circuit lower bounds. The implied circuit lower bounds are not as strong as those needed to construct pseudorandom generators but still seem out of the reach of current techniques.

We initiate the study of whether subexponential-time *typically-correct* derandomizations imply such lower bounds. We provide an affirmative answer in the case of the error rates considered by Goldreich and Wigderson, namely algorithms that make at most 2^{n^ϵ} mistakes for all positive ϵ. We show that such a typically-correct derandomization of the BPP problem polynomial-identity testing implies either super-polynomial Boolean circuit lower bounds for nondeterministic exponential time (NEXP) or super-polynomial arithmetic circuit lower bounds for the permanent over the integers (Perm).

THEOREM 1.3 *If for every positive constant ϵ there exists a nondeterministic Turing machine which runs in time 2^{n^ϵ} and correctly decides ACZ, the language of all arithmetic circuits that compute the zero polynomial over the integers, on all but at most 2^{n^ϵ} of the inputs of length n for almost every n, then*

(i) NEXP does not have Boolean circuits of polynomial size, or

(ii) Perm does not have arithmetic circuits of polynomial size.

This result is a strengthening of [KI04] from the everywhere-correct setting to the typically-correct setting. In developing it, we also obtain a simpler proof for the everywhere-correct setting that scales better than the one in [KI04].

Relativization and Algebrization The fact that typically-correct derandomization of BPP with very low error rates implies circuit lower bounds indicates that any such derandomization of BPP must have ingredients that prove circuit lower bounds. It remains open whether typically-correct derandomization of BPP with higher error rates implies circuit lower bounds. However, we show that such weaker derandomization of BPP must contain

ingredients that do not algebrize – a notion developed by Aaronson and Wigderson [AW09] that includes both relativizing proof techniques as well as techniques based on arithmetization. Thus we know that a typically-correct derandomization of BPP with larger error rates cannot be proved with only relativizing techniques and arithmetization.

1.3 Derandomization of Monotone Computations

As discussed earlier, in many settings we know that the existence of hard functions implies pseudorandom generators suitable for derandomizing randomized algorithms. Strong enough hardness conditions imply efficient full derandomization, for example BPP = P. As discussed in Section 1.2, weaker hardness conditions imply efficient typically-correct derandomization.

A natural question then is, for which algorithmic settings do we have hard functions that yield derandomization? Monotone functions are one of the notable settings where hard functions are known. In this section we describe our work in considering whether these hardness results imply derandomization; and we ask the broader question of how derandomization of monotone computations relates to the derandomization of general non-monotone computations.

Monotone Boolean Functions Monotone circuits are one area where we know of very good *worst-case* lower bounds. A monotone Boolean function is one such that flipping an input bit from 0 to 1 can only change the output of the function from 0 to 1. Monotone functions can be computed by monotone circuits – circuits consisting of AND and OR gates but with no NOT gates. There are many examples of natural monotone languages based on graph properties – such as clique, connectivity, or perfect matchings – where adding edges can only make the property easier to satisfy. One hope in studying monotone Boolean functions is that the property of monotonicity can be used in proving interesting results. This hope has come to fruition in the area of circuit lower bounds. A long line of research has proved that various explicit monotone functions require monotone circuits of super-polynomial size (perfect matchings) or even exponential size (clique) (see [BS91] and [Kor03] for surveys).

Derandomizing Monotone Circuits An immediate question is whether these exponential *worst-case* lower bounds can be converted into *average-case* lower bounds of a sufficient quality for use in hardness-based pseudorandom generators to derandomize bounded-error randomized monotone circuits. The latter are monotone circuits C that take two inputs x and r such that for every x, $\Pr_R[C(x,R)=1] \geq \frac{2}{3}$ or $\Pr_R[C(x,R)=1] \leq \frac{1}{3}$. Including a uniformity condition – that there is a deterministic polynomial-time machine that on input 1^n outputs the circuit C – gives a natural monotone version of the complexity class BPP.

Consider the requirements needed to apply a hardness-based pseudorandom generator to derandomize monotone circuits. The proofs for hardness-based pseudorandom generators all argue the contrapositive: if the generator is not secure against small circuits, a reduction is given that uses those small circuits to approximately compute the presumed hard function. In the setting of monotone circuits, we would assume a small monotone circuit that distinguishes the output of the generator from uniform, and with this monotone distinguisher we should construct a small monotone circuit that approximately computes the presumed hard function. The reduction from the distinguisher to the circuit approximating the hard function should preserve monotonicity. Let us consider two different generator constructions that have been developed to derandomize time-bounded computations – the Shaltiel-Umans generator [SU05, Uma03] and the Nisan-Wigderson generator [NW94]. The Shaltiel-Umans generator uses elements such as list-decodable codes and finite field arithmetic that perform non-monotone operations such as parity, and it is unclear if these elements can be made monotone.

On the other hand, as observed in [Kar09], an examination of the reduction for proving the Nisan-Wigderson reveals that only a single negation is needed, and a monotone function hard for both monotone circuits and their negations could be used in this generator to derandomize monotone circuits. To derandomize a circuit of size n^k, the known proof of the generator requires a function that is $(\frac{1}{2} - \frac{1}{n^k})$-hard for small circuits. We ask, then, whether the known circuit lower bounds proofs for monotone circuits can be adapted to prove this level of average-case hardness. If so, this would add to the few classes of randomized algorithms for

which we have good unconditional derandomization. In fact [Kar09] poses such average-case lower bounds for monotone functions as an open problem.

A negative answer comes from work in learning theory. A series of results has culminated in the result of [OW09] that for any monotone function f, one of $\{0, 1, x_1, ..., x_n, \text{Majority}\}$ is within distance $\frac{1}{2} - \Omega(\log n/\sqrt{n})$ of f. In particular, no monotone function has hardness greater than this amount for circuits large enough to compute majority – linear-size general circuits or $O(n \log n)$ size monotone circuits. We observe that this barrier is close to tight by showing the existence of a monotone function that is $(\frac{1}{2} - \frac{1}{n^{1/2-\eta}})$-hard for circuits with $2^{n^{\Omega(1)}}$ gates, for any positive constant η.

1.3.1 Our Results

From the discussion above, we know that there can be no monotone function with high enough average-case hardness to be used in known hardness-based pseudorandom generators to derandomize monotone circuits. But the question remains open for general *non-monotone* functions, namely whether we can prove high average case hardness for some explicit non-monotone function for *monotone* circuits. We consider this goal and other questions related to derandomizing monotone circuits in this dissertation.

Hard on Average Functions First, we show that a function that is hard on average for monotone circuits is hard on average for general circuits with somewhat weaker parameters. We prove the contrapositive – that a general circuit approximating any function can be converted into a monotone circuit without too much loss in parameters. In the following, an anti-monotone circuit is the negation of a monotone circuit.

THEOREM 1.4 *Let f be any function. If there is a general circuit C with s gates that computes f to within $\frac{1}{2} - \epsilon$, then there is either a monotone or anti-monotone circuit with $2s + O(n \log^2 n)$ gates that computes f to within $\frac{1}{2} - \epsilon'$ for $\epsilon' = \max(\frac{\epsilon}{n+1}, \frac{c}{\sqrt{n \log(1/\epsilon)}})$ for $c > 0$ an absolute constant.*

We observe that Theorem 1.4 is tight to within a constant factor for the parity function. Parity can be computed exactly, so $\epsilon = \frac{1}{2}$, by a small general circuit. Applying Theorem 1.4 to this circuit gives a monotone circuit computing parity to within $\frac{1}{2} - \Omega(\frac{1}{\sqrt{n}})$. But it is well-known that no monotone function can compute parity to within more than $\frac{1}{2} - O(\frac{1}{\sqrt{n}})$, a fact which we prove for completeness. Thus Theorem 1.4 is tight at least for large values of ϵ.

Pseudorandom Generators Theorem 1.4 shows that one particular method of constructing a pseudorandom generator secure against monotone circuits – namely constructing a hard function for use in the Nisan-Wigderson generator – would also yield results for general non-monotone circuits. We show that in fact any method for constructing a pseudorandom generator secure against monotone circuits also implies a generator secure against general circuits with somewhat weaker parameters. A slightly weaker version of Theorem 1.5 was independently discovered by Karakostas [Kar09], namely with $\epsilon' = \frac{\epsilon}{2(n+1)}$.

THEOREM 1.5 *Let C be a circuit of size s that ϵ-distinguishes some distribution $\mathcal{D}$ from uniform. Then there is a monotone circuit C' of size $2s + O(n \log^2 n)$ that ϵ'-distinguishes $\mathcal{D}$ from uniform for $\epsilon' = \max(\frac{\epsilon}{2(n+1)}, \frac{c}{\sqrt{n \log(1/\epsilon)}})$ for $c > 0$ an absolute constant.*

In particular, if $\mathcal{D}$ is the output distribution of a pseudorandom generator G, then a distinguisher for G can be converted into a monotone circuit without too much loss in the distinguishing probability. We observe that Theorem 1.5 is nearly tight for pseudorandom generators with small stretch as follows. We prove that the generator which simply outputs its seed and the parity of the seed is $\epsilon = O(\frac{1}{\sqrt{n}})$ indistinguishable for monotone circuits. On the other hand this generator can be distinguished with $\epsilon = \frac{1}{2}$ by a small general circuit, and applying Theorem 1.5 to this circuit gives a monotone circuit distinguishing the generator with $\epsilon = \Omega(\frac{1}{\sqrt{n}})$.

Derandomization in General Constructing pseudorandom generators is one method to derandomize (monotone) randomized circuits. We show that any method of derandomizing monotone randomized circuits can also be used to derandomize general non-monotone randomized computations.

THEOREM 1.6 *Let L be any language computable by polynomial-time bounded-error randomized machines. There is a language L_{mon} computable by uniform monotone bounded-error polynomial-size randomized circuits such that L poly-time mapping reduces to L_{mon}. In particular, if $L_{mon} \in \mathrm{P}$ then $L \in \mathrm{P}$.*

1.4 Space Hierarchy Theorems

Hierarchy Theorems A hierarchy theorem states that the power of a machine increases with the amount of resources it can use. Time hierarchy theorems on deterministic Turing machines follow by direct diagonalization: a machine N diagonalizes against every machine M_i running in time t by choosing an input x_i, simulating $M_i(x_i)$ for t steps, and then doing the opposite. Deriving a time hierarchy theorem for computational models not known to be efficiently closed under complement, such as nondeterministic machines, is more complicated. A variety of techniques can be used to overcome this difficulty, including translation arguments and delayed diagonalization [Coo73, SFM78, Žàk83]. These techniques allow us to prove time hierarchy theorems for just about any *syntactic* model of computation. We call a model syntactic if there exists a computable enumeration of all machines in the model. For example, we can enumerate all nondeterministic Turing machines by representing their transition functions as strings and then iterating over all such strings to discover each nondeterministic Turing machine.

Hierarchy Theorems on Semantic Models Many models of computation of interest are not syntactic but *semantic*. A semantic model is defined by imposing a promise on a syntactic model. A machine belongs to the model if it is output by the enumeration of the underlying syntactic model and its execution satisfies the promise on every input.

Bounded-error randomized Turing machines, the machine model underlying the complexity class BPP, are an example of a non-syntactic semantic model. There does not exist a computable enumeration consisting of exactly all randomized Turing machines that satisfy the promise of bounded error on every input, but we can enumerate all randomized Turing machines and attempt to select among them those that have bounded error. In general promises make diagonalization problematic because the diagonalizing machine must satisfy the promise everywhere but has insufficient resources to determine whether a given machine from the enumeration against which it tries to diagonalize satisfies the promise on a given input.

Because of these difficulties good time hierarchies for semantic models are known only when the model has been shown equivalent to a syntactic model. These hierarchies result from equalities such as IP = PSPACE [Sha92], MIP = NEXP [BFL91], BP.$\oplus$P = Σ_2.$\oplus$P [Tod91], and PCP($\log n$,1) = NP [ALM+98]. Similarly, if BPP computations can be derandomized in a uniform way then a good time hierarchy for bounded-error randomized machines would follow as well. But can we prove a good time hierarchy short of proving derandomization?

A recent line of research [Bar02, FS04, GST04, FST05, MP07] has provided progress toward proving *time* hierarchy results for non-syntactic models, including bounded-error randomized machines. Each of these results applies to semantic models that take advice, where the diagonalizing machine is only guaranteed to satisfy the promise when it is given the correct advice. Many of the results require only one bit of advice. For some of these results, namely those of [MP07], at a high level the advice bit is used by the diagonalizing machine to avoid simulating a machine on an input for which that machine breaks the promise.

Our Results In this dissertation we consider hierarchy theorems for the amount of *memory space* rather than time used by randomized and other semantic models of computation. Our results adapt to the space-bounded setting techniques that had previously been developed in the time-bounded setting. Like the time hierarchy results in this line of research, our space

hierarchy results have a number of parameters: (1) the gap needed between the two space bounds, (2) the amount of advice that is needed for the diagonalizing machine N, (3) the amount of advice that can be given to the smaller space machines M_i, and (4) the range of space bounds for which the results hold.

We consider (1) and (2) to be of the highest importance. We focus on space hierarchy theorems with an optimal separation in space – where any super-constant gap in space suffices. This is an improvement over corresponding time hierarchy results for semantic models [Bar02, FS04, GST04, FST05, MP07], which are not as tight with respect to time as the best time hierarchies for syntactic models. The ultimate goal for (2) is to remove the advice altogether and obtain uniform hierarchy results. As in the time-bounded setting, we do not achieve this goal but get the next best result – a single bit of advice for N suffices in each of our results. Given that we strive for space hierarchies that are tight with respect to space and require only one bit of advice for the diagonalizing machine, we aim to optimize parameters (3) and (4).

1.4.1 Randomized Models with Advice

Our strongest results apply to randomized models. For two-sided error machines, we can handle a large amount of advice and any typical space bound between logarithmic and linear.

THEOREM 1.7 *Let $s(n)$ be any space-constructible monotone function such that $s(n) = \Omega(\log n)$, and let $s'(n)$ be any function that is $\omega(s(n + as(n)))$ for all constants a. There exists a language computable by two-sided error randomized machines using $s'(n)$ space and one bit of advice that is not computable by two-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.*

For $s(n) = \log(n)$, Theorem 1.7 gives a two-sided error machine using only slightly larger than $\log n$ space that uses one bit of advice and differs from all two-sided error machines using $O(\log n)$ space and $O(\log n)$ bits of advice. Space-constructibility is a standard assumption in hierarchy theorems that is true of typical space bounds.

If $s(n)$ is a space-constructible monotone function that is at most linear, we show the condition on $s'(n)$ in the above can be relaxed to $s'(n) = \omega(s(n+1))$, giving a hierarchy that is as tight with respect to space as the space hierarchies for generic syntactic models. In fact, typical space bounds $s(n)$ that are $O(n)$ satisfy $s(n+1) = O(s(n))$, meaning the condition on $s'(n)$ can be relaxed further to $s'(n) = \omega(s(n))$. Thus we obtain space hierarchies that are tight with respect to space for typical space bounds that are at most linear.

Our second main result, Theorem 1.8, gives a separation result with similar parameters as those of Theorem 1.7 but for the cases of one- and zero-sided error randomized machines. A randomized machine computes a language with one-sided error if the machine has bounded-error for inputs in the language and is always correct for inputs not in the language. A zero-sided error machine may output "don't know" with probability less than half, must never output an incorrect answer, and must output the correct answer with probability at least half. We point out that the separation result for zero-sided error machines is new to the space-bounded setting as the techniques used to prove stronger separations in the time-bounded setting do not work for zero-sided error machines. In fact, we show a single result that captures space separations for one- and zero-sided error machines – that a zero-sided error machine suffices to diagonalize against one-sided error machines.

THEOREM 1.8 *Let $s(n)$ be any space-constructible monotone function such that $s(n) = \Omega(\log n)$, and let $s'(n)$ be any function that is $\omega(s(n + as(n)))$ for all constants a. There exists a language computable by zero-sided error randomized machines using $s'(n)$ space and one bit of advice that is not computable by one-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.*

As in the case of two-sided error, the condition on $s'(n)$ can be relaxed to $s'(n) = \omega(s(n+1))$ for space-constructible monotone space bounds $s(n) = O(n)$ and relaxed further to $s'(n) = \omega(s(n))$ for space bounds that satisfy $s(n+1) = O(s(n))$ as do typical space bounds that are at most linear.

1.4.2 Generic Semantic Models with Advice

The above results take advantage of specific properties of randomized machines that are not known to hold for arbitrary semantic models. Our next results involve a generic construction of [MP07] that applies to a wide class of semantic models which the authors term *reasonable* and that can be *safely complemented* with a limited overhead in space. The requirements for a reasonable model are very basic; we refer to Section 3.2.3 for the precise conditions but besides randomized two-, one-, and zero-sided error machines, the notion also encompasses bounded-error quantum machines, unambiguous machines, Arthur-Merlin games and interactive proofs, etc. For discussion and definitions of these models, see Chapter 2. A safe complementation is – loosely speaking – a machine that *always* satisfies the semantic conditions of the model, takes as its input a machine-input pair, and has the opposite behavior whenever the machine-input pair satisfies the semantic conditions; we refer to Definition 3.5 for the exact meaning. Most reasonable models, including all the above, can be safely complemented with a linear-exponential overhead in space.

THEOREM 1.9 (FOLLOWS FROM [MP07]) *Fix any reasonable semantic model of computation that can be safely complemented with a linear-exponential overhead in space. Let $s'(n)$ be any function with $s'(n) = \omega(\log n)$. There exists a language computable using $s'(n)$ space and one bit of advice that is not computable using $O(\log n)$ space and $O(1)$ bits of advice.*

The performance of the generic construction is poor on the last two parameters we mentioned earlier – it allows few advice bits on the smaller space side and is only tight for $s(n) = O(\log n)$. Either of these parameters can be improved for models that can be safely complemented with only a polynomial overhead in space – models for which the simple translation argument works. Examples of such models include randomized machines with bounded error and unambiguous machines. In fact, there is a trade-off between (a) the amount of advice that can be handled and (b) the range of space bounds for which the result is tight. By maximizing (a) we get the following.

THEOREM 1.10 *Fix any reasonable semantic model of computation that can be safely complemented with a polynomial overhead in space. Let d be a rational upper bound on the degree of the latter polynomial. Let $s'(n)$ be any function with $s'(n) = \omega(\log n)$. There exists a language computable using $s'(n)$ space and one bit of advice that is not computable using $O(\log n)$ space and $O(\log^{1/d} n)$ bits of advice.*

In fact, a tight separation in space can be maintained while allowing $O(\log^{1/d} n)$ advice bits for $s(n)$ any poly-logarithmic function, but the separation in space with this many advice bits is no longer tight for larger $s(n)$. By maximizing (b), we obtain a separation result that is tight for sufficiently smooth space bounds between logarithmic and polynomial. We state the result for polynomial space bounds.

THEOREM 1.11 *Fix any reasonable semantic model of computation that can be safely complemented with a polynomial overhead in space. Let d be a rational upper bound on the degree of the latter polynomial, let r be any positive constant, and let $s'(n)$ be any space bound that is $\omega(n^r)$. There exists a language computable in space $s'(n)$ with one bit of advice that is not computable in space $O(n^r)$ with $O(1)$ bits of advice.*

When applied to randomized machines, Theorem 1.11 gives a tight separation result for slightly higher space bounds than Theorems 1.7 and 1.8, but the latter can handle more advice bits.

1.4.3 Promise Problems for Generic Semantic Models

Our proofs use advice in a critical way to derive hierarchy theorems for *languages* computable by semantic models. We can obviate the need for advice by considering *promise problems* rather than languages. A promise problem only specifies the behavior of a machine on a subset of the inputs; the machine may behave arbitrarily on inputs outside of this set. For semantic models of computation, one can associate in a natural way a promise problem to each machine in the underlying enumeration. For example, for randomized machines with bounded error, the associated promise problem only specifies the behavior on inputs on which

the machine has bounded error. The ability to ignore problematic inputs allows traditional techniques to demonstrate good space and time hierarchy theorems for the promise problems computable by semantic models. This is a folklore result, but there does not appear to be a correct proof in the literature; we include one in this dissertation.

THEOREM 1.12 (FOLKLORE) *Fix any reasonable semantic model of computation that can be safely complemented with a computable overhead in space. Let $s(n)$ and $s'(n)$ be space bounds with $s(n) = \Omega(\log n)$ and $s'(n)$ space-constructible. If $s'(n) = \omega(s(n+1))$ then there is a promise problem computable within the model using space $s'(n)$ that is not computable as a promise problem within the model using space $s(n)$.*

1.5 Organization

For the remainder of this dissertation, we present our results in the chronological order of their discovery.

- Chapter 2 introduces the notation, terminology, and machine models used throughout the dissertation. A reader familiar with the basics of complexity theory may wish to skip this chapter and refer back to it as needed.

- Chapters 3 and 4 contain our results on hierarchy theorems for randomized and other semantic models. Chapter 3 contains our hierarchy results that apply to generic semantic models of computation, and Chapter 4 contains our stronger results that apply to bounded-error randomized machines. A preliminary version of the results of Chapters 3 and 4 was presented at the 25th International Symposium on Theoretical Aspects of Computer Science (Bordeaux, February 2008) [KM08]. The full version containing a more refined analysis of a number of the results is in press to be published in the journal Computational Complexity [KM10].

- Chapter 5 introduces the pseudorandom generator approach to typically-correct derandomization and applies this approach to a number of settings. Chapter 6 considers

whether typically-correct derandomization of BPP implies circuit lower bounds; we answer in the affirmative for very small error rates. A preliminary version of the results in Chapters 5 and 6 was presented at the 13th International Workshop on Randomization and Computation (Berkeley, August 2009) [KMS09]. A full version has been accepted to the special issue of the journal Computational Complexity for selected papers from the conference.

- Chapter 7 contains our results relating monotone and general computation in the settings of randomized algorithms, pseudorandom generators, and average-case hard functions. These results have not yet been published.

Chapter 2

Preliminaries

Here we introduce the notation, terminology, and machine models used throughout the dissertation and state relevant properties. A reader familiar with the basics of computational complexity may wish to skip this section and refer back to it as needed. For a more thorough treatment of these concepts and properties, see [AB09] and [Gol08].

2.1 Deterministic Algorithms and Turing Machines

As is standard, we use the multi-tape deterministic Turing machine as our base machine model. We use the notation $M(x) = 1$ to indicate that M halts and accepts x, $M(x) = 0$ to indicate that M halts and rejects x, and $M(x) =\uparrow$ to indicate that M on input x does not terminate. A language L, also known as a decision problem, is a subset of strings. When $x \in L$ we also write $L(x) = 1$, and when $x \notin L$ we say that $L(x) = 0$. Thus if $M(x) = L(x)$ then M halts and decides L correctly on input x.

The space usage of machine M on input x is defined as the number of work-tape cells that are touched during the computation; the space usage of M at input length n is defined as the maximum over all x of length n. For a space bound $s : \mathbb{N} \to \mathbb{N}$, we say M uses space at most s if M uses space at most $s(n)$ at input length n, for all $n \in \mathbb{N}$. Time usage of M is similarly defined based on the number of steps in M's execution.

We restrict ourselves to machines M that use the binary alphabet for their input and output tapes. However, M may have a number of work tapes and work-tape alphabet of its own choosing; for a fixed machine M its work-tape alphabet and number of work tapes

are of constant size. Allowing machines with arbitrary alphabet sizes has the following consequence. Suppose M uses space $s(n)$. Then for any constant $c > 0$, there exists a machine M' that uses at most $\max(c \cdot s(n), 1)$ space and behaves as M on every input. For $c < 1$, M' uses a larger alphabet size than M and compresses each block of roughly $1/c$ tape cells of M into one tape cell using its larger alphabet size. The ability to compress space usage by any constant factor implies machines that run in space $s(n)$ and $O(s(n))$ are equally powerful.

We can represent each Turing machine M as a binary string by encoding its number of work tapes, size of alphabet, transition function, etc. as binary strings. We use M to denote both the machine and the binary string that represents the machine. We can assume without loss of generality that a Turing machine M has a unique accepting *configuration* (internal state, tape contents, and tape head locations) by ensuring it clears its tape contents and resets its tape heads before entering a unique accepting state. We can similarly assume that M has a unique rejecting configuration. These transformations do not increase the space usage of the machine.

Conversely, we can assume that every string is a description of some Turing machine. This follows by taking a standard encoding of Turing machines and mapping any string that is not valid in that encoding to a default Turing machine, for example the Turing machine that immediately rejects on all inputs. We point out that this trivially makes deterministic Turing machines computably enumerable, as defined next.

DEFINITION 2.1 (COMPUTABLE ENUMERATION) *A set S is* computably enumerable *if there exists a Turing machine M such that*

1. *on input i, $M(i)$ outputs a string y with $y \in S$,*
2. *for any $y \in S$, there exists an i such that $M(i)$ outputs y, and*
3. *$M(i)$ halts for every input i.*

We note that in standard enumerations $(M_i)_{i=1,2,3,\ldots}$ of deterministic Turing machines, each machine M_i appears infinitely often as different encodings of the same machine. Each

of these encodings, though, has the same number of work tapes, the same tape alphabets, the same internal states, and the same behavior on any given input. Typical diagonalization arguments proceed by having a diagonalizing machine N iterate over each machine M_i in turn and ensure that N computes a language different than M_i. As M_i appears infinitely often within the enumeration, N has an infinite number of opportunities to successfully differentiate itself from M_i.

There exists a *space-efficient universal Turing machine* U to simulate other Turing machines. Namely, given input (M, x), $U(M, x) = M(x)$ and if $M(x)$ uses space s then U uses at most $a \cdot s$ space where a is a constant that only depends on the *control characteristics* of M – its number of tapes, work-tape alphabet size and number of states – but is the same for each of the infinitely many different occurrences M_i of the machine M in the enumeration of machines. We can equip the universal machine U with a space counter to keep it from using more space than we want. For any space-constructible function s (defined next), there exists a universal machine U_s such that $U_s(M, x) = M(x)$ if $M(x)$ uses at most s space, and $U_s(M, x)$ uses at most $a' \cdot s(|x|)$ space where a' is a constant depending only on s and the control characteristics of M. We implicitly use the universal machine throughout this paper whenever the diagonalizing machine needs to simulate another machine.

Definition 2.2 (space-constructible) *A space bound s is defined as* space-constructible *if there exists a Turing machine using $O(s(n))$ space which on input 1^n produces as output $s(n)$ many 1's.*

Most common space bounds we work with are space-constructible, including polynomials, exponentials, and logarithms.

Turing-Machine with Advice We can also equip Turing machines with *advice*. Turing machines with advice are a non-uniform model of computation in which the machine has access to an advice string that varies depending on the input length. This so-called advice is given as an additional input to the Turing machine. We use α and β to denote infinite sequences of advice strings.

Definition 2.3 (computation with advice) *A Turing machine M with advice sequence α decides on an input x by performing the computation $M(x;\alpha_{|x|})$, denoted $M(x)/\alpha_{|x|}$. M with advice sequence α, denoted M/α, computes a language L if for every x, $M(x)/\alpha_{|x|} = L(x)$. If $|\alpha_n| = a(n)$ for all n, we say that L can be computed with $a(n)$ bits of advice.*

When we are interested in the execution of M/α on inputs of length n, we write M/a where $a = \alpha_n$.

Oracle Turing-Machine A Turing Machine can also be given access to an oracle for some language L. The effect is that the machine has access to answers to the language L with only unit time cost per query. This is achieved by augmenting the machine with an oracle tape and a special "query oracle" state of its internal transition function; upon entering the query state, the machine in one time step places the answer to the query (1 if the query is a member of L, and 0 otherwise) at the current location of the first work-tape and clears the query tape. There are subtleties involved when considering space-bounded oracle Turing machines, but we will only require time-bounded oracle Turing machines so do not discuss those subtleties here.

We write P^L for the set of languages that can be solved in polynomial time with oracle access to L. For a complexity class $\mathcal{C}$, we write $\mathrm{P}^{\mathcal{C}}$ for the set of languages that can be solved in polynomial time given oracle access to some language $L \in \mathcal{C}$. In other words, $\mathrm{P}^{\mathcal{C}} = \cup_{L\in\mathcal{C}}\mathrm{P}^L$.

2.2 Randomized Algorithms and Turing Machines

A *randomized Turing machine* is a deterministic Turing machine that in addition is given a read-only one-way infinite tape of random bits in addition to the usual input, work, and output tapes. With the contents of the random bit tape fixed to some value, a randomized Turing machine behaves as a standard Turing machine. The behavior of a randomized Turing machine M on a given input x with the random bits r unfixed is a random variable $M(x;r)$ over the probability space of the random bit tape with the uniform distribution. In

particular, the contents of the output tape and whether the machine enters the accept or reject states are random variables.

We say that a randomized Turing machine M uses space $s(|x|)$ and time $t(|x|)$ if $M(x;r)$ uses at most $s(|x|)$ space and $t(|x|)$ time for every possible choice of randomness r.

In the case of space-bounded randomized Turing machines, it may be possible that a machine uses at most space s but nevertheless does not terminate for some values of the random bit tape. Allowing space-bounded randomized machines to execute indefinitely gives them significant power, namely the power of nondeterminism. We only consider space-bounded randomized machines which are *guaranteed to halt* for all possible contents of the random bit tape. One implication of this assumption is that a randomized machine M using space $s = \Omega(\log n)$ runs in 2^{as} time for a constant a that depends only on the control characteristics of M. This follows from the fact that the number of configurations of a space s machine is $O(n2^{O(s)})$, which is $2^{O(s)}$ for $s = \Omega(\log n)$, and none of these configurations can be repeated for a machine which always halts. For more on the basic properties of space-bounded randomized machines, see [Sak96].

Intuitively, a randomized machine computes a function f if for every input x, $M(x;r) = f(x)$ with high probability over r. In this paper we focus on decision problems f, or equivalently, languages L. We consider three different types of error behavior for a randomized machine computing a language: two-, one-, and zero-sided error.

Definition 2.4 (two-sided error) *A randomized machine M computes a language L with* two-sided error *if for every x, $\Pr_r[M(x;r) = L(x)] \geq \frac{2}{3}$.*

If $\Pr_r[M(x;r) = 1] < \frac{2}{3}$ and $\Pr_r[M(x;r) = 0] < \frac{2}{3}$ we say that M breaks the promise of two-sided error on input x; otherwise we say M satisfies the promise of two-sided error on input x. The complexity class BPL consists of the languages that can be computed by a logarithmic-space two-sided error Turing machine that always halts.

Definition 2.5 (one-sided error) *A randomized machine M computes a language L with* one-sided error *if*

1. *for every $x \in L$,* $\Pr_r[M(x;r) = 1] \geq \frac{1}{2}$, *and*
2. *for every $x \notin L$,* $\Pr_r[M(x;r) = 0] = 1$.

If $\Pr_r[M(x;r) = 1] < \frac{1}{2}$ and $\Pr_r[M(x;r) = 0] < 1$ we say that M breaks the promise of one-sided error on input x. The complexity class RL consists of the languages that can be computed by a logarithmic-space one-sided error Turing machine that always halts.

To define zero-sided error, we consider three possible outcomes of the computation: 1 meaning accept, 0 meaning reject, or ? meaning unsure.

Definition 2.6 (zero-sided error) *A randomized machine M computes L with* zero-sided error *if*

1. *for every x,* $\Pr_r[M(x;r) \notin \{0,1\}] \leq \frac{1}{2}$, *and*
2. *for every x,* $\Pr_r[M(x;r) = \neg L(x)] = 0$.

If $\Pr_r[M(x;r) \notin \{0,1\}] > \frac{1}{2}$ or ($\Pr_r[M(x;r) = 1] > 0$ and $\Pr_r[M(x;r) = 0] > 0$) we say that M breaks the promise of zero-sided error on input x. The complexity class ZPL consists of the languages that can be computed by a logarithmic-space zero-sided error Turing machine that always halts.

When speaking of a two-sided error (respectively one- or zero-sided error) randomized machine M, we say that $M(x) = 1$ if the acceptance condition of M on input x is met – namely that $\Pr_r[M(x;r) = 1] \geq \frac{2}{3}$ (respectively $\Pr_r[M(x;r) = 1] \geq \frac{1}{2}$ or ($\Pr_r[M(x;r) \notin \{0,1\}] \leq \frac{1}{2}$ and $\Pr_r[M(x;r) = 0] = 0$)). Similarly, we say that $M(x) = 0$ if the rejection condition of M on input x is met.

Properties As a randomized machine has at its base a deterministic Turing machine, many of the properties of deterministic Turing machines carry over. We can assume that there are unique accepting and rejecting configurations. We can encode randomized Turing machines

as binary strings such that every randomized Turing machine has infinitely many different encodings and every string represents some randomized Turing machine. This trivially gives a computable enumeration of randomized Turing machines where each machine appears infinitely often.

The space-efficient universal machine U also carries over from the class of deterministic Turing machines to the class of randomized Turing machines. In particular, this machine U allows for space-efficient simulations of randomized machines with two-, one-, or zero-sided error. However, U itself does not satisfy the promise of two-, one-, or zero-sided error on all inputs and therefore is not universal for two, one-, or zero-sided error machines. In fact, the existence of a space-efficient universal machine for two-, one-, or zero-sided error machines remains open, and if one exists then known diagonalization techniques immediately give tight space hierarchies for these models without advice.

Randomized Machines with Advice Randomized machines take *advice* in much the same way that deterministic Turing machines take advice – as an additional input. We refer to Section 2.7 for the precise meaning of a bounded-error machine with advice as a special case of semantic models with advice.

2.2.1 Error Reduction

Given a randomized machine deciding a language L, majority voting allows us to decrease the probability the machine errors. One way to view this is as an application of the Chernoff bound. We use the following instantiation (see, for example, [MR95, Theorem 4.2 and Theorem 4.3]).

THEOREM 2.7 (CHERNOFF BOUND) *Let X_i be independent identically distributed 0/1 random variables, and let $S_\tau = \sum_{i=1}^{\tau} X_i$. Let $\mu = \tau \cdot E[X_1]$ be the mean of S_τ.*

1. *For any $\Delta > 0$, $\Pr[S_\tau < \mu - \Delta] \leq e^{-\Delta^2/(2\mu)}$.*
2. *For $0 \leq \Delta \leq (2e-1)\mu$, $\Pr[S_\tau > \mu + \Delta] \leq e^{-\Delta^2/(4\mu)}$.*

Consider a randomized machine M on input x, and assume that $\Pr_r[M(x;r) = L(x)] = \frac{1}{2} + \gamma$ for some $\gamma > 0$. We run $M(x)$ some number τ times independently, that is, with fresh random bits for each execution. For $i = 1, 2, ..., \tau$, we let $X_i = 1$ if the i^{th} execution of $M(x)$ produces the correct result, and $X_i = 0$ otherwise. Theorem 2.7 tells us that the number of correct outputs in the τ trials does not stray far from the expected number. By applying both Theorems 1 and 2, the probability that the fraction of correct outputs lies outside of the range $[\frac{1}{2}, \frac{1}{2} + 2\gamma]$ is at most $e^{-\tau\gamma^2/4} + e^{-\tau\gamma^2/2} \leq 2e^{-\tau\gamma^2/4}$. In particular, the probability that the majority vote of τ independent trials of M is incorrect is exponentially small in τ.

For one- and zero-sided error machines, we can reduce the error somewhat more efficiently. For a one-sided error machine M, we take the OR of τ independent trials of $M(x)$. This preserves the one-sided error condition and if $\Pr_r[M(x;r) = 1] \geq \frac{1}{2}$ then the probability that the OR of τ independent trials is incorrect is at most $\frac{1}{2^\tau}$. The error of a zero-sided error machine M is similarly reduced to $\frac{1}{2^\tau}$ by taking τ independent trials and outputting 0 if $M(x)$ outputs 0 on any of the trials, 1 if $M(x)$ outputs 1 on any of the trials, and ? otherwise.

2.2.2 Deterministic Simulations

A space $s = \Omega(\log n)$ randomized Turing machine M_i that always halts can be simulated by a deterministic Turing machine D that runs in time $2^{as(n)}$ for some constant a that only depends on the control characteristics of M_i. D on input x accepts if $\Pr_r[M_i(x;r) = 1] \geq \frac{1}{2}$ and rejects otherwise.

We sketch this simulation. To achieve D, we first view M_i as defining a Markov chain whose states are the $t = 2^{O(s(n))}$ possible configurations of M_i and whose transition probabilities are governed by the transition function of M_i. As M_i on input x halts within t time steps, we determine if $\Pr_r[M_i(x;r) = 1]$ is at least $1/2$ by taking the t^{th} power of the Markov chain's transition matrix and examine the resulting probability for the state corresponding to the unique accepting configuration of M_i. The main task of D is to compute an entry in

the product of the t^{th} power of the $t \times t$ transition matrix of the Markov chain, which can be done in polynomial time in t, i.e., in time $2^{O(s(n))}$.

We point out that deterministic simulations of bounded-error randomized machines are known which use smaller space [Nis92, SZ99], but the above suffices for most of our purposes. When we require the more efficient simulations, we explicitly state this in the text.

2.3 Nondeterministic and Unambiguous Machines

If we remove the requirement of bounded error in part 1 of Definition 2.5 for one-sided error randomized machines, we are left with a syntactic model of computation, namely *nondeterministic Turing machines*, which is at least as powerful as the semantic model of one-sided error machines. When viewed as a nondeterministic machine, the random bits from the random bit tape are now viewed as "guess bits" from a nondeterministic tape. We say that a nondeterministic machine M computes a language L if for every x, $\Pr_r[M(x;r) = 1] > 0$ if and only if $x \in L$. The complexity class NP consists of the languages that can be computed by a polynomial-time nondeterministic Turing machine; NEXP consists of the languages solvable by exponential-time nondeterministic Turing machines; in general NTIME($t(n)$) denotes languages solvable by time t nondeterministic machines. The language of satisfiable Boolean formulas, denoted SAT, is one of many languages that are known to be complete for NP. The complexity class NL consists of the languages that can be computed by a log-space nondeterministic Turing machine, and reachability on directed graphs is a complete problem for NL.

Unambiguous machines are a semantic model of nondeterministic Turing machines where accepting paths are required to be unique. That is, a nondeterministic machine M computes a language L unambiguously if the following hold.

(i) For every $x \in L$, $M(x;r) = 1$ for exactly one choice of r.

(ii) For every $x \notin L$, $M(x;r) = 0$ for every choice of r.

We define the complexity class UP as the class of languages accepted by polynomial-time unambiguous machines, and similarly define UL for logarithmic-space unambiguous machines [BJLR91]. Unambiguous machines have been studied in the time-bounded setting due to the relation between the assumptions P $\neq$ UP and NP $\neq$ UP and the existence of various cryptographic primitives (see, e.g., [HT03]). In the log-space setting, certain restrictions of graph connectivity have been shown to be computable within UL (see, e.g., [BTV09, PTV10]), and an interesting question is the relation between UL and NL (see, e.g., [RA00]).

2.4 Other Randomized Models and Quantum Machines

The randomized computations discussed above are achieved by adding randomness to deterministic Turing machines. We can also add randomness to other types of computations. For a complexity class $\mathcal{C}$, we define the complexity class BP.$\mathcal{C}$ as all languages L such that for some constant k and $L' \in \mathcal{C}$ the following holds.

(i) For every $x \in L$, $\Pr_{R \leftarrow U_{n^k}}[(x, R) \in L'] \geq \frac{2}{3}$.

(ii) For every $x \notin L$, $\Pr_{R \leftarrow U_{n^k}}[(x, R) \in L'] \leq \frac{1}{3}$.

Arthur-Merlin Protocols The class of *Arthur-Merlin protocols*, denoted AM, was defined in [Bab85] and can be seen as an extension of NP proofs to include randomness. On input x, an efficient randomized procedure Arthur seeks to verify membership of x in a language L with the assistance of an all-powerful prover Merlin. Arthur and Merlin exchange a constant number of messages, viewed as Arthur sending questions or challenges that Merlin must answer to convince Arthur that $x \in L$, and at the end of the protocol Arthur decides to accept or reject. The protocol must satisfy the following properties.

(i) For every $x \in L$, with probability at least $\frac{2}{3}$ over Arthur's random bits, there are answers Merlin can give to Arthur's challenges that cause Arthur to accept.

(ii) For every $x \notin L$, Arthur rejects with probability at least $\frac{2}{3}$ regardless of Merlin's behavior.

The complexity class AM is defined by allowing Arthur to run in polynomial time. It can be shown that this definition of AM is equivalent to AM = BP.NP. Arthur-Merlin protocols have received attention in part due to the fact that AM contains certain problems, such as graph non-isomorphism, that are not known to be solvable with non-randomized proof systems, i.e., not known to be contained in NP.

[Con93] surveys the properties of log-space bounded Arthur-Merlin protocols and related interactive protocols which allow an arbitrary rather than constant number of messages.

Randomized Oracle Machines A randomized Turing machine can be equipped with an oracle in the same way that a deterministic Turing machine can be equipped with an oracle. For a language L, BPP^L denotes the set of languages that can be solved by a polynomial-time bounded-error randomized machine that has access to an oracle for L. It can be shown that for any language L, $\mathrm{BP}.L \subseteq \mathrm{BPP}^L$. In particular, $\mathrm{AM} = \mathrm{BP.NP} \subseteq \mathrm{BPP}^{\mathrm{SAT}}$.

Quantum Machines The complexity class BQP is a semantic model of computation defined as the class of problems solvable by polynomial-time quantum Turing machines that have error bounded by $\frac{1}{3}$ on every input. We do not state the precise definition of a quantum Turing machine here because it is not central to this dissertation. Quantum machines are defined to allow computers to take advantage of the effects of quantum mechanics, using "qubits" rather than random bits. Quantum algorithms have received attention in large part due to the fact that factoring and discrete-log can be solved in BQP [Sho97]. Watrous [Wat03] defined a space-bounded version of quantum Turing machines and investigated their properties.

2.5 Distance and Hardness

We say that a language L' is *within distance* δ of another language L if their characteristic functions are within Hamming distance δ, i.e., differ on at most δ fraction of inputs for each input length n. Distance between a language and a class of languages is similarly defined,

and we say a language L is δ-hard for a class of languages if no language in the class is within δ of L.

DEFINITION 2.8 (HARDNESS ON AVERAGE) *A language L is $\delta(\cdot)$-*hard *for a class of languages $\mathcal{C}$ if no language $L' \in \mathcal{C}$ is within Hamming distance $\delta(n)$ of L for almost all input lengths n.*

Notice that worst-case hardness corresponds to setting $\delta(n) = \frac{1}{2^n}$. We use the term "mild hardness" when $\delta(n) = \frac{1}{n^c}$ for some constant $c > 0$, and the term "very high hardness" when $\delta(n) = \frac{1}{2} - \frac{1}{2^{n^\epsilon}}$ for some constant $\epsilon > 0$.

For many of our results the relevant class of languages $\mathcal{C}$ are the languages computable by circuits or branching programs of a certain size. These are discussed in Section 2.6.

2.6 Circuits

Boolean Circuits and Branching Programs A *Boolean circuit* is a directed acyclic graph with each internal node labeled as an AND, OR, or NOT gate and with each root node labeled as either some input bit x_i or one of the constants 0 or 1. One of the leaf nodes is labeled as the output of the circuit, and this output is computed in the natural way.

A *branching program* is a directed acyclic graph where internal nodes and the root node are labeled with variables, edges correspond to either 0 or 1, leaves are labeled either "accept" or "reject", and the computation on a given input is performed as follows: at a given node labeled x_i, if $x_i = 0$ then proceed along the edge labeled 0 and otherwise proceed along the edge labeled 1, repeat this process until a leaf is reached.

We measure the *size* of a circuit or branching program by the string length of its standard description, and assume the description mechanism is such that the description of a circuit or branching program of size s can easily be padded into the description of an equivalent circuit or branching program of size s' for any $s' > s$. Note that up to a logarithmic factor, this measure corresponds to the number of connections in the circuit or branching program. For any function $s(n)$ we denote by SIZE($s(n)$) the class of languages L such that L at length

n can be decided by a Boolean circuit of size $s(n)$ for almost all input lengths n; we use the notation BP-SIZE($s(n)$) for the class of languages decidable by branching programs of size $s(n)$.

It can be shown that polynomial-size circuits correspond in power to non-uniform polynomial-time deterministic Turing machines: a polynomial-size circuit can simulate a polynomial-time machine that is allowed a polynomial amount of advice, and vice versa. Branching programs can be shown to correspond to non-uniform space-bounded computations: polynomial-size branching programs can simulate a log-space deterministic Turing machine that is given a polynomial amount of advice, and vice versa.

Arithmetic Circuits We also consider *arithmetic circuits*, which have internal nodes representing addition, subtraction, and multiplication, and leaves representing variables and the constants 0 and 1. We denote by ASIZE($a(n)$) the class of families $(p_n)_{n\in\mathbb{N}}$ of polynomials over $\mathbb{Z}$ where p_n has n variables and can be computed by an arithmetic circuit of size $a(n)$ for almost all $n \in \mathbb{N}$.

Oracle Circuits We let $\text{SIZE}^O(s(n))$ refer to the languages computable by Boolean circuits of size $s(n)$ that have access to oracle gates for the language O. For an oracle circuit, an oracle gate contributes its number of inputs to the size of the circuit.

Uniform Circuits and Branching Programs Both circuits and branching programs as stated so far are non-uniform models of computation – a different circuit or branching program is given for each input length n. We say a circuit or branching program of size s is polynomial-time (respectively log-space) *uniform* if the circuit or branching program can be computed by a polynomial-time (respectively log-space) machine, that on input $(1^n, i)$ outputs the i^{th} bit of the description in poly($s(n)$) time (respectively $O(\log s(n))$ space). Other more strict notions of uniformity could be considered, but we restrict our attention to these two basic notions to avoid delving into details which are orthogonal to the main ideas of the dissertation. For more information on circuits, issues of uniformity, etc., see [Vol99].

Constant-Depth Circuits The complexity class AC^0 consists of the set of languages that can be computed by non-uniform circuits of polynomial size and constant depth. This is a natural definition of languages that can be computed in constant time by parallel machines. We use the terminology "uniform AC^0" to refer to the languages solvable by uniform constant-depth circuits.

2.6.1 Hardness Amplification

For circuits and branching programs, hardness can be amplified using the XOR lemma. Several versions of the XOR lemma exist (see [GNW95] for an overview); we use the following instantiation for circuits.

LEMMA 2.9 (XOR LEMMA FOR CIRCUITS [IMP95]) *Let $H : \{0,1\}^n \to \{0,1\}$ be a language and define $H' : \{0,1\}^{k \cdot n} \to \{0,1\}$ by $H'(x_1, ..., x_k) = H(x_1) \oplus H(x_2) \oplus ... \oplus H(x_k)$. For any $\gamma > 0$, if H is δ-hard for size s circuits at input length n, then H' is δ'-hard for size s' circuits at input length $k \cdot n$, where $\delta' = \frac{1}{2} - (1-\delta)^k - \gamma$ and $s' = \Omega(\frac{\gamma^2}{\log(1/(\delta\gamma))}) \cdot s$.*

In some settings, the XOR Lemma may not be sufficient as a means for amplifying hardness. In particular, if the types of algorithms in question are not known to be efficiently closed under taking parities (often equivalent to being efficiently closed under complementation), then the amplified hard function would not have comparable complexity to H. A notable example is the setting of nondeterministic algorithms, which are not efficiently closed under taking parities unless NP=coNP. To obtain a hardness amplification lemma for NP algorithms, O'Donnell [O'D04] showed that a monotone combining function can be used in place of parity with some loss in parameters. Because the combining function is monotone, the technique can also be used in the setting of monotone functions, giving Theorem 2.10. A function is balanced if it outputs 0 and 1 with equal probability.

THEOREM 2.10 (FOLLOWS FROM [O'D04]) *Let H be a monotone function that is balanced and $\frac{1}{n^c}$-hard for circuits of size s, for some positive constant c. There is a polynomial p and*

poly-*time computable monotone function* C *such that* $H' : \{0,1\}^{n \cdot p(n)} \to \{0,1\}$ *defined as*

$$H'(x_1, x_2, ..., x_{p(n)}) = C(H(x_1), H(x_2), ..., H(x_{p(n)}))$$

is $(\frac{1}{2} - \frac{1}{(n \cdot p(n))^{1/2-\eta}})$*-hard for circuits of size* $\frac{s}{n^d}$ *on inputs of length* $n \cdot p(n)$*, where* d *is a constant that depends on* c*.*

2.7 Semantic Models of Computation

A *syntactic* model of computation is defined by a computable enumeration of machines $M_1, M_2, \ldots$, and a mapping that associates with each M_i and input x the output $M_i(x)$ (if any). Deterministic Turing machines and randomized Turing machines are examples of syntactic models, where for a randomized machine M on input x we can define $M(x) = 1$ if $\Pr_r[M(x; r) = 1] \geq \frac{1}{2}$, and $M(x) = 0$ otherwise.

A *semantic* model is obtained from a syntactic model by imposing a *promise* π, which is a Boolean predicate on pairs consisting of a machine M_i from the underlying enumeration and an input x. We say that M_i *satisfies the promise on input* x if $\pi(M_i, x) = 1$. A machine M_i is termed *valid*, or said to *fall within the semantic model*, if it satisfies the promise on all inputs. The models of randomized machines with two-, one- and zero-sided error are examples of semantic models. They can be obtained by imposing the promise of two-, one-, and zero-sided error on randomized Turing machines.

In fact, these models are examples of non-syntactic semantic models, i.e., there does not exist a computable enumeration that consists exactly of all machines within the model. To see that the class of bounded-error randomized Turing machines is not computably enumerable, we note that the complement of the halting problem reduces to the set of bounded-error randomized machines. Given a deterministic machine M and input x, the reduction maps (M, x) to a randomized Turing machine M' that behaves as follows. M' on input t simulates $M(x)$ for at most t steps; if $M(x)$ halts before this point then M' outputs 1 with probability 1/2 and 0 with probability 1/2, and if $M(x)$ does not halt within t steps then M' on input t outputs 1 with probability 1. Note that M' satisfies the promise of bounded error on

all inputs if and only if $M(x)$ does not halt. Thus, the complement of the halting problem reduces to the set of bounded-error randomized machines. Since the former is not computably enumerable, the latter cannot be either.

Other examples of non-syntactic semantic models include bounded-error quantum machines [Wat03], unambiguous machines [BJLR91], Arthur-Merlin games and interactive proofs [Con93], etc. We refer to [MP07] for a more formal treatment of syntactic versus semantic models.

We can equip a semantic model with advice and define advice within semantic models in much the same way we have for deterministic machines.

Definition 2.11 (semantic model with advice) *Given a semantic model, a machine M from the underlying enumeration with advice sequence α decides on input x by performing the computation $M(x;\alpha_{|x|})$, denoted $M(x)/\alpha_{|x|}$. M with advice sequence α, denoted M/α,* computes a language L within the model *if for every x, $M(x)/\alpha_{|x|}$ satisfies the underlying promise and $M(x)/\alpha_{|x|} = L(x)$.*

We do not require that M satisfy the promise when given an "incorrect" advice string. We note that this differs from the notion of advice introduced in [KL82], where the machine must satisfy the promise no matter which advice string is given. We point out that a hierarchy for a semantic model with advice under the stronger Karp-Lipton notion would imply the existence of a hierarchy without advice. Indeed, suppose we have a hierarchy with $a(n)$ bits of advice under the Karp-Lipton notion. Then there is a valid machine M' running in space $s'(n)$ and an advice sequence α'_0, α'_1, ... with $|\alpha'_n| = a(n)$ such that for all valid machines M running in space $s(n)$, and for all advice sequences α_0, α_1, ... with $|\alpha_n| = a(n)$, there is an input x such that $M'(x)/\alpha'_{|x|}$ and $M(x)/\alpha_{|x|}$ disagree. In particular, we have that M' and M disagree on $z = (x;\alpha'_{|x|})$. Thus M' is a valid machine using space $s'(n)$ on inputs of length $n + a(n)$ which differs from all valid machines that use space $s(n)$ on inputs of length $n + a(n)$.

2.8 Promise Problems

Promise problems are computational problems that are only specified for a subset of all possible input strings, namely those that satisfy a certain promise. We will only deal with promise decision problems, which can be defined formally as follows.

DEFINITION 2.12 (PROMISE PROBLEM) *A promise problem is a pair of disjoint sets* (Π_Y, Π_N) *of strings.*

The set Π_Y in Definition 2.12 represents the set of "yes" instances, i.e., the inputs for which the answer is specified to be positive. Similarly, Π_N denotes the set of "no" instances. The sets Π_Y and Π_N must be disjoint for consistency, but do not need to cover the space of all strings. If they do, we are in the special case of a language. Otherwise, we are working under the nontrivial promise that the input string lies in $\Pi_Y \cup \Pi_N$.

A machine solving a promise problem is like a program with a precondition – we do not care about its behavior on inputs outside of $\Pi_Y \cup \Pi_N$. In particular, for the time and space complexity of the machine we only consider inputs in $\Pi_Y \cup \Pi_N$. In the case of semantic models, the machine is only required to satisfy the promise π underlying the semantic model on inputs x that satisfy the promise $x \in \Pi_Y \cup \Pi_N$ of the promise problem.

Chapter 3

Hierarchy Theorems for Generic Semantic Models

In this chapter, we consider hierarchy theorems for generic semantic models of computation. We begin by reviewing one of the techniques – delayed diagonalization – that is known to give good time and space hierarchies for syntactic models such as nondeterministic machines, and we see that the technique encounters difficulties for *semantic* models such as bounded-error randomized machines. We then present two different ways to overcome these difficulties. In Section 3.1, we show that delayed diagonalization proves hierarchies for the promise problems computed by semantic models. In Section 3.2, we show how to adapt delayed diagonalization to semantic models that use one bit of advice. In Chapter 4 we demonstrate yet another way to adapt delayed diagonalization to bounded-error randomized machines that gives even stronger results for these models.

Direct and Delayed Diagonalization Recall that direct diagonalization suffices to prove good time and space hierarchy theorems for deterministic machines.

> **Direct Diagonalization**: A machine N diagonalizes against every machine M_i running in time t by choosing an input x_i, simulating $M_i(x_i)$ for t steps, and then doing the opposite.

This technique fails to prove a time hierarchy for nondeterministic machines because complementation cannot be performed time-efficiently within the model (unless NP=coNP).

Delayed diagonalization is one of the techniques that can be used to overcome this problem. We demonstrate the technique by sketching the proof in [Žàk83] of a time hierarchy

for nondeterministic machines. The reader may find it helpful to consult an illustration in Figure 3.1 while reading the following proof sketch. We wish to demonstrate a nondeterministic machine N using slightly more than $t(n)$ time which differs from all nondeterministic machines that use $t(n)$ time. For each machine M_i, N allocates an interval of input lengths $[n_i, n_i^*]$ on which to diagonalize against M_i. The construction consists of two main parts.

(1) A delayed complementation at length n_i^* of M_i's behavior at length n_i.

(2) A scheme to copy the complementary behavior down to smaller and smaller padded input lengths all the way to n_i.

For (1), we choose n_i^* large enough so that N has sufficient time at length n_i^* to complement the behavior of M_i at length n_i. By using brute-force search to perform the complementation, we can set $n_i^* = 2^{\Theta(n_i)}$. N performs a *delayed complementation* by ensuring that $N(0^{n_i^*-n_i}x) = \neg M_i(x)$ for x with $|x| = n_i$.

For (2), on inputs of the form $0^j x$ with $|x| = n_i$ and $0 \leq j < n_i^* - n_i$, N simulates $M_i(0^{j+1}x)$ while M_i uses at most $t(n)$ time, outputs a value if M_i does, and outright rejects if M_i uses more than $t(n)$ time. Suppose M_i is a machine which uses at most $t(n)$ time and computes the same language as N on all input lengths in $[n_i, n_i^*]$. This assumption and N's definition imply the following set of equalities for every input x of length n_i

$$\begin{aligned} M_i(x) &= N(x) = M_i(0x) = N(0x) = M_i(0^2x) = ... \\ &= M_i(0^{n_i^*-n_i}x) = N(0^{n_i^*-n_i}x) = \neg M_i(x). \end{aligned}$$

As $M_i(x)$ must take some definite value, we have reached a contradiction. Either N differs from M_i on some input of length in $[n_i, n_i^*]$, or M_i uses more than $t(n)$ time. We conclude that N indeed computes a language different than that computed by *any* time $t(n)$ nondeterministic machine.

The above proof takes advantage of only very basic properties of nondeterministic machines, in particular the following.

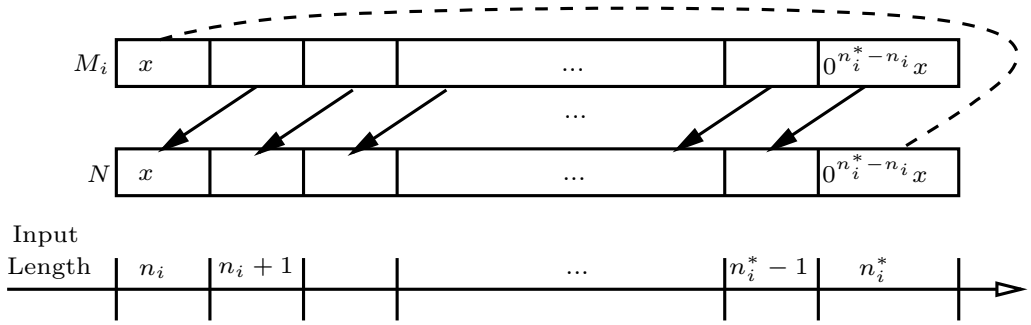


Figure 3.1 Illustration of delayed diagonalization for nondeterministic machines. The solid arrows indicate that on inputs of the form $0^j x$, N simulates $M_i(0^{j+1}x)$. The dashed line indicates that on input $0^{n_i^*-n_i}x$, N outputs the complement of $M_i(x)$.

(a) The list of all nondeterministic machines M_i is computably enumerable. We can enumerate all nondeterministic Turing machines by representing their transition functions as strings and then iterating over all such strings to discover each nondeterministic Turing machine.

(b) Nondeterministic computations can be complemented with some computable blowup in time (namely, an exponential blowup in time).

(c) A nondeterministic machine can efficiently simulate another.

Delayed diagonalization can be used to prove good time and space hierarchies for any model of computation that has these basic properties. In particular, the technique applies to just about any *syntactic* model of computation, a model such that the list of all machines is computably enumerable.

Semantic Models Now let us see what happens when we attempt to apply the above techniques of direct and delayed diagonalization to semantic models of computation. We begin by focusing on two-sided error randomized machines. At first glance, it might seem we can use direct diagonalization because a two-sided error computation can be complemented within the model. If given a two-sided error randomized machine M_i and input x, a diagonalizing machine N can easily complement $M_i(x)$ by simply simulating $M_i(x)$ and always

outputting the opposite value. If $\Pr[M_i(x) = 1] \geq \frac{2}{3}$, then $\Pr[N(x) = 0] \geq \frac{2}{3}$ and likewise if $\Pr[M_i(x) = 1] \leq \frac{1}{3}$. But any computable enumeration of randomized machines contains some machines that *do not* have bounded error. If $M_i(x)$ does not have bounded error, then by simulating $M_i(x)$ and outputting the opposite N also does not have bounded error. What we need for argument to work is a method to complement machines that have bounded error without falling into the above trap when we encounter a randomized machine that does not have bounded error. We call such a procedure a *safe complementation*, defined as follows.

Definition 3.1 (safe complementation) *Fix a semantic model of computation and let N and M be two machines in the computable enumeration of the underlying syntactic model. N on input y safely complements M on input x if $N(y)$ satisfies the promise (even if $M(x)$ does not), and if $M(x)$ satisfies the promise then $N(y) \neq M(x)$.*

A safe complementation in general incurs a blowup in space, even for models such as two-sided error machines which are closed under complementation, because N must avoid breaking the promise when working against a machine M_i which does break the promise. One way to achieve this is for $N(y)$ to deterministically simulate $M(x)$ and flip the result. For two-sided error randomized machines, the best known deterministic simulation incurs an exponential overhead in *time*.

Space Hierarchies using Direct Diagonalization However, the situation is better when we consider *space* as the resource. It is known [SZ99] that for any space-constructable bound $s(n)$, any randomized two-sided error computation running in space $s(n)$ can be simulated deterministically in space $(s(n))^{1.5}$, meaning there is also a safe complementation with this overhead in space. Using this safe complementation and direct diagonalization, we have that for any space constructible $s'(n) = \omega((s(n))^{1.5})$ there are languages computable by two-sided error randomized machines using space $s'(n)$ that are not computable by two-sided error randomized machines using space $s(n)$.

In fact, a stronger result is known. The following simple translation argument suffices to show that for any constant $c > 1$ there exists a language computable by two-sided error

randomized machines using $(s(n))^c$ space that is not computable by such machines using $s(n)$ space [KV87], for any space-constructible $s(n)$ that is $\Omega(\log n)$. Suppose by way of contradiction that every language computable by two-sided error machines in space $(s(n))^c$ is also computable by such machines in space $s(n)$. A padding argument then shows that in that model any language computable in $(s(n))^{c^2}$ space is computable in space $(s(n))^c$ and thus in space $s(n)$. We can iterate this padding argument any constant number of times and show that for any constant d, any language computable by two-sided error machines in space $(s(n))^d$ is also computable by such machines in $s(n)$ space. For $d > 1.5$ we reach a contradiction with the result stated at the end of the previous paragraph. The same argument applies to other non-syntactic semantic models where $s(n)$ space computations can be simulated deterministically in space $(s(n))^d$ for some constant d, including one- and zero-sided error randomized machines and unambiguous machines.

This simple method of proving hierarchy theorems for semantic models – use the best known deterministic simulation as a safe complementation together with direct diagonalization and a simple translation argument – falls short of our ultimate goals for two reasons. First, the technique only gives good results for models that are known to have efficient deterministic simulations. For many semantic models, for example Arthur-Merlin games, the best known deterministic simulations incur an exponential overhead in both time and space. Second, even for models such as bounded-error randomized algorithms where the results are fairly good, they still fall short of the best possible. Since we can always reduce the space usage by a constant factor by increasing the work-tape alphabet size, the tightest space hierarchy result one can hope for is to separate space $s'(n)$ from space $s(n)$ for any space-constructible function $s'(n) = \omega(s(n))$. For models like nondeterministic machines, which are known to be closed under complementation in the space-bounded setting [Imm88, Sze88], such tight space hierarchies follow by direct diagonalization. For generic *syntactic* models, very tight space hierarchies follow using techniques such as delayed diagonalization. This begs the question whether delayed diagonalization can be applied to prove tight hierarchy theorems for semantic models .

Delayed Diagonalization on Semantic Models The diagonalizing machine in a delayed diagonalization argument has two main tasks: (1) perform a delayed complementation, and (2) implement a copying scheme through simulations of the other machines M_i. When applied to semantic models, for (1) we seek a *safe* complementation. A safe complementation may incur a large overhead (exponential or more) in resources, but this is not a problem because delayed diagonalization is specifically designed for models where complementation cannot be achieved efficiently. For syntactic models, (2) is achieved by ensuring for certain values of j that $N(0^j x) = M_i(0^{j+1}x)$ by simply simulating the computation of M_i. It is these simulations which cause problems when operating in semantic models: if $M_i(0^{j+1}x)$ does not satisfy the promise underlying the semantic model, then N would likewise fail to satisfy the promise.

In the next two sections, we show two different methods to overcome this problem. Both methods begin with the following intuition. If $M_i(0^{j+1}x)$ does not satisfy the promise underlying the model, then N already computes differently on input $0^{j+1}x$ and we should abstain from the simulation $N(0^j x) = M_i(0^{j+1}x)$. We show how to achieve this intuition using either promise problems or computations that use one bit of advice.

Our Results In this chapter, we use techniques that apply to a very wide class of non-syntactic models, yielding *tight* space hierarchy results for promise problems and computations that use one bit of advice. In Section 3.2.3, after completing the proofs, we give a precise definition of the properties required of a computational model for the proofs in this chapter. These properties are very modest and are true of any "reasonable" semantic model of computation – two-, one-, or zero-sided error randomized machines, quantum machines, Arthur-Merlin games, interactive proofs, unambiguous machines, etc.

3.1 Promise Problems

In this section, we prove good space hierarchies for the promise problems computed by reasonable semantic models such as bounded-error randomized machines. For convenience, we have restated the theorem that is proved in this section.

THEOREM 1.12 (FOLKLORE) *Fix any reasonable semantic model of computation that can be safely complemented with a computable overhead in space. Let $s(n)$ and $s'(n)$ be space bounds with $s(n) = \Omega(\log n)$ and $s'(n)$ space-constructible. If $s'(n) = \omega(s(n+1))$ then there is a promise problem computable within the model using space $s'(n)$ that is not computable as a promise problem within the model using space $s(n)$.*

A promise problem is a pair of disjoint sets (Π_Y, Π_N) of strings, and we say that a machine from a semantic model solves the promise problem if for every $x \in \Pi_Y \cup \Pi_N$, the machine decides correctly and satisfies the promise underlying the semantic model. For inputs outside the promise of the problem, the machine can behave arbitrarily. See Section 2.8 for further discussion of promise problems.

The precise definition of a reasonable semantic model is deferred to Section 3.2.3, but the notion corresponds roughly to those semantic models where the underlying syntactic model has the modest properties required of delayed diagonalization discussed at the beginning of this chapter.

For concreteness, consider two-sided error randomized machines. A first attempt at proving the hierarchy is to use direct diagonalization. Namely, construct a diagonalizing machine that enumerates all randomized machines M_i, chooses a certain input x_i for machine M_i, and simulates $M_i(x_i)$ and does the opposite. But suppose $M_i(x_i)$ does not have two-sided error. Then any promise problem which M_i computes must have $x_i \notin \{\Pi_Y \cup \Pi_N\}$, and the same holds for our diagonalizing machine since it simulates and negates $M_i(x_i)$. As x_i has the same status with respect to both promise problems, we have not diagonalized against M_i after all.

Another complication arises when considering promise problems. In the context of two-sided error for a randomized machine M, the natural promise problem to associate with M is to set $\Pi_Y = \{x | \Pr[M(x) = 1] \geq 2/3\}$ and $\Pi_N = \{x | \Pr[M(x) = 1] \leq 1/3\}$. However, there are many other valid promise problems that M decides by ignoring certain inputs even though M has two-sided error on these. The diagonalizing machine N we construct must work against each M_i in such a way that the promise problem we associate with N differs from *every* promise problem which M_i solves.

To handle the latter problem, we will ensure there is an input y on which there is a safe complementation – $N(y)$ has two-sided error, and either $M(y)$ does not have two sided error or $N(y) \neq M(y)$ – in either case, the status of y with respect to the two promise problems is different. To achieve this goal, we use delayed diagonalization to initiate a delayed safe complementation. The rest of the argument amounts to carrying through the standard delayed diagonalization proof and verifying that the goal is achieved. For completeness, we sketch the complete argument.

3.1.1 Proof of Theorem 1.12

We first prove Theorem 1.12 for the particular case of two-sided error randomized machines. Let N be the machine we build to diagonalize against promise problems computable by two-sided error space $s(n)$ machines. For each randomized machine M_i, we allocate an interval of input lengths $[n_i, n_i^*]$ on which to diagonalize against M_i. The first part of the construction is a delayed complementation, which is achieved on input $0^{n_i^*}$. Let n_i^* be large enough so that N can deterministically compute the acceptance probability of $M_i(0^{n_i})$ using space $s(n_i^*)$. $N(0^{n_i^*})$ should do the opposite of $M_i(0^{n_i})$. This is ensured by placing $0^{n_i^*}$ within the promise of N and having $N(0^{n_i^*})$ output 1 with probability 1 if $\Pr[M_i(0^{n_i}) = 1] < \frac{1}{2}$, and output 0 with probability 1 otherwise. Notice that regardless of the status of $M_i(0^{n_i})$ in terms of a promise problem (either 0^{n_i} is in Π_Y, Π_N, or neither), $N(0^{n_i^*})$ is different.

The second part of the construction copies down the complementary behavior to smaller and smaller padded inputs. On input 0^{n_i+j} for $0 \leq j < n_i^* - n_i$, N simulates $M_i(0^{n_i+j+1})$

while it uses at most $s(n_i + j + 1)$ space, and we define N's promise to be the natural one on each of these inputs – the input is within the promise (either Π_Y or Π_N) when its probability of acceptance is either at least 2/3 or at most 1/3. On inputs other than those of the form 0^{n_i+j}, N rejects and halts immediately (these inputs are not used in the diagonalization).

Suppose there is a machine M_i using at most $s(n)$ space which computes the promise problem we associate with N on all inputs in the interval $[n_i, n_i^*]$. Because $0^{n_i^*}$ is in the promise of N, this is also true for M_i. $N(0^{n_i^*-1})$ by construction simulates $M_i(0^{n_i^*})$, and an input has been defined to be in the promise of N iff N has two-sided error on the input. So $0^{n_i^*-1}$ is in the promise of N, and therefore must also be in the promise of M_i. If we continue this argument through the entire interval, we conclude that each 0^{n_i+j} is contained within the promise of both N and M_i for $j = n_i^* - n_i, n_i^* - n_i - 1, ..., 0$. By the assumption that M_i computes the promise problem we associate with N, the fact that each input is in the promise of M_i and N, and the construction of N to simulate M_i, we have the following set of equalities

$$\begin{aligned} M_i(0^{n_i}) = &\quad N(0^{n_i}) = M_i(0^{n_i+1}) = N(0^{n_i+1}) = M_i(0^{n_i+2}) \\ = &\quad ... = M_i(0^{n_i^*-1}) = N(0^{n_i^*-1}) = M_i(0^{n_i^*}) = N(0^{n_i^*}). \end{aligned}$$

However, we have constructed $N(0^{n_i^*})$ so that it explicitly differs from $M_i(0^{n_i})$: if 0^{n_i} is in the promise of M_i, then N flips the output; otherwise 0^{n_i} is not in the promise of M_i even though $0^{n_i^*}$ is in the promise of N. In either case, $N(0^{n_i^*}) \neq M_i(0^{n_i})$ where $\neq$ means the promise problem is different on each. We have reached a contradiction, so there can be no promise problem defined on M_i that corresponds to the natural promise problem of N. Further, standard techniques guarantee that $s'(n)$ space is sufficient for N to carry out this construction against all randomized machines M_i for any $s'(n)$ with $s'(n) = \omega(s(n+1))$. Namely, equip N with a mechanism to ensure it never uses more than $s'(n)$ space, and use an enumeration of randomized machines where each machine appears infinitely often to ensure that for each machine M', at least once while working against M' the asymptotic behavior of s' and s has taken effect so that N successfully completes the construction against M'.

Generalization to Reasonable Semantic Models The above proof requires only a basic set of properties and holds for a wide range of semantic models. One requirement is that safe complementation can be achieved with space overhead σ for some computable function σ. The computability of σ and the fact that s' is a constructible bound that grows unboundedly allow us to construct a partition of the input lengths in intervals $[n_i, n_i^*]$ with the following properties: (1) the partition up to length n can be generated in space $O(\log n)$, and (2) if M_i runs in space $s'(n_i - 1)$ at length n_i (which is true for sufficiently large n if M_i runs in space $O(s(n_i))$ at length n_i and $s'(n) = \omega(s(n+1))$), then M_i can be safely complemented within space $s'(n_i^*)$ at length n_i. Note that $s'(n) = \omega(\log n)$, so the partitioning can be computed in space $O(s'(n))$. These properties suffice to carry through the above construction of a diagonalizing machine N that runs in space $O(s'(n))$, completing the proof of Theorem 1.12. A semantic model must also satisfy a few additional modest requirements for the above argument to carry through. These details are deferred to Section 3.2.3 at the end of this chapter.

By clocking the partitioning algorithm to run in time $O(n)$ rather than space $O(\log n)$, the above argument can be modified to yield the following time-bounded equivalent of Theorem 1.12.

THEOREM 3.2 (FOLKLORE) *Fix any reasonable semantic model of computation that has a safe complementation with a computable overhead in time. Let $t(n)$ and $t'(n)$ be time bounds with $t(n) = \Omega(n)$ and $t'(n)$ time-constructible. If $t'(n) = \omega(t(n+1) \cdot \log t(n+1))$ then there is a promise problem computable within the model using time $t'(n)$ that is not computable as a promise problem within the model using time $t(n)$.*

3.2 Semantic Models with One Bit of Advice

In the last section, we saw that delayed diagonalization can be applied to semantic models of computation by considering promise problems rather than languages. In this section, we show how to adapt delayed diagonalization to *languages* computed by semantic models *that*

use one bit of advice, proving the following results. Even with only one bit of advice, we show how to diagonalize against smaller space machines that use *many* bits of advice.

First, Theorem 1.9 is the most general, applying to any reasonable semantic model of computation. We define the notion in Section 3.2.3 after completing the construction and analysis of the proofs.

THEOREM 1.9 (FOLLOWS FROM [MP07]) *Fix any reasonable semantic model of computation that can be safely complemented with a linear-exponential overhead in space. Let $s'(n)$ be any function with $s'(n) = \omega(\log n)$. There exists a language computable using $s'(n)$ space and one bit of advice that is not computable using $O(\log n)$ space and $O(1)$ bits of advice.*

For Theorem 1.10, we show a stronger result for models with more efficient safe complementations – the smaller resource machines can be given more advice.

THEOREM 1.10 *Fix any reasonable semantic model of computation that can be safely complemented with a polynomial overhead in space. Let d be a rational upper bound on the degree of the latter polynomial. Let $s'(n)$ be any function with $s'(n) = \omega(\log n)$. There exists a language computable using $s'(n)$ space and one bit of advice that is not computable using $O(\log n)$ space and $O(\log^{1/d} n)$ bits of advice.*

Theorem 1.11 shows a different strengthening for models with more efficient complementations – the result holds for small as well as large space bounds. In fact, a tradeoff could be proved that interpolates between Theorems 1.10 and 1.11 for the amount of advice bits that can be given to the smaller resource machine and the largest space bound for which the result remains tight.

THEOREM 1.11 *Fix any reasonable semantic model of computation that can be safely complemented with a polynomial overhead in space. Let d be a rational upper bound on the degree of the latter polynomial, let r be any positive constant, and let $s'(n)$ be any space bound that is $\omega(n^r)$. There exists a language computable in space $s'(n)$ with one bit of advice that is not computable in space $O(n^r)$ with $O(1)$ bits of advice.*

As with the previous section, the results are very general and apply to a wide class of semantic models. The basic construction is the same for each, with only the analysis differing. We first describe the construction that adapts delayed diagonalization to semantic models with the use of advice (Section 3.2.1), analyze the construction for the particular case of each theorem (Section 3.2.2), and finally distill the properties of a semantic model that are needed for our constructions to hold (Section 3.2.3).

3.2.1 Delayed Diagonalization on Semantic Models with Advice

In this section we adapt delayed diagonalization to proving hierarchy theorems for the languages computed by semantic models. We begin by considering semantic models that do not use advice; advice arises naturally along the way as a method to overcome the difficulties of proving hierarchy theorems for semantic models. The goal is to construct a diagonalizing machine N that uses not much more than $s(n)$ space, satisfies the promise underlying the semantic model on all inputs, and differs from each machine M_i which *behaves appropriately*. The latter is defined as follows.

Definition 3.3 (appropriate behavior of machines in a semantic model) *Fix a semantic model of computation and a space bound $s(n)$. A machine M_i from the underlying syntactic model with advice sequence β* behaves appropriately *if M_i/β satisfies the promise of the model and uses at most $s(n)$ space on all inputs.*

We keep a few specific semantic models in mind during the development and analysis of the construction – Arthur-Merlin games for Theorem 1.9, and unambiguous machines for the stronger separations of Theorems 1.10 and 1.11. A reader unfamiliar with these semantic models may instead keep in mind bounded-error randomized machines. In fact, the ensuing construction and analysis apply to any semantic model of computation that satisfies some modest requirements. Rather than listing these requirements ahead of time, we determine the properties that are needed of a semantic model afterward, namely in Section 3.2.3.

The delayed diagonalization construction given at the beginning of this chapter fails for non-syntactic models: it may be the case that M_i breaks the promise on inputs of the form

$0^{j+1}x$, and N would also break the promise by copying the behavior of $M_i(0^{j+1}x)$ when given input 0^jx. In Section 3.1 we dealt with this problem by considering promise problems rather than languages and defining any inputs on which N does not satisfy the promise underlying the model as falling outside the promise problem. Now we are considering languages, and N must satisfy the promise underlying the model on all inputs. Likewise, N only must differ from machine M_i that satisfy the promise on all inputs. If M_i breaks the promise on some input, then N does not need to consider M_i and may simply abstain from working against M_i. We give N one bit of advice at each input length to indicate if performing the simulations at that length would cause N to break the promise. If the advice bit α_n is 1, then N/α performs the simulation. If the advice bit is 0, N/α abstains by immediately rejecting.

As N is allowed one bit of advice, M_i should also be allowed at least one advice bit. With M_i allowed one bit of advice, N now has two different machines at each input length that it is concerned with – $M_i/0$ and $M_i/1$. N should perform a given simulation if at least one of these behaves appropriately and copies N's behavior. This can be done by giving N two advice bits – one each to indicate whether each of $M_i/0$ and $M_i/1$ behaves appropriately and copies N's behavior on inputs of one larger length. In general, if M_i is allowed $a(n)$ bits of advice, N would require $2^{a(n+1)}$ advice bits to specify whether M_i with each advice string behaves appropriately and copies N's behavior on inputs of one larger length.

Consider the simulations of M_i at length n_i^* which N is responsible for copying to smaller padded inputs. Rather than giving N $2^{a(n_i^*)}$ advice bits to indicate for which advice strings M_i behaves appropriately, we instead wish to spread these advice bits over different input lengths so that N uses only one bit of advice. That is, for each of M_i's possible advice strings b at length n_i^*, we allocate a distinct slightly smaller input length from which N is responsible for simulating M_i/b at length n_i^*. For the input length responsible for advice string b, N's advice bit is set to indicate if M_i/b behaves appropriately at length n_i^*. If the advice bit is 1, N/α performs the simulation of M_i/b at length n_i^*. If the advice bit is 0, N abstains by immediately rejecting. Now N/α satisfies the promise on all inputs, and for each

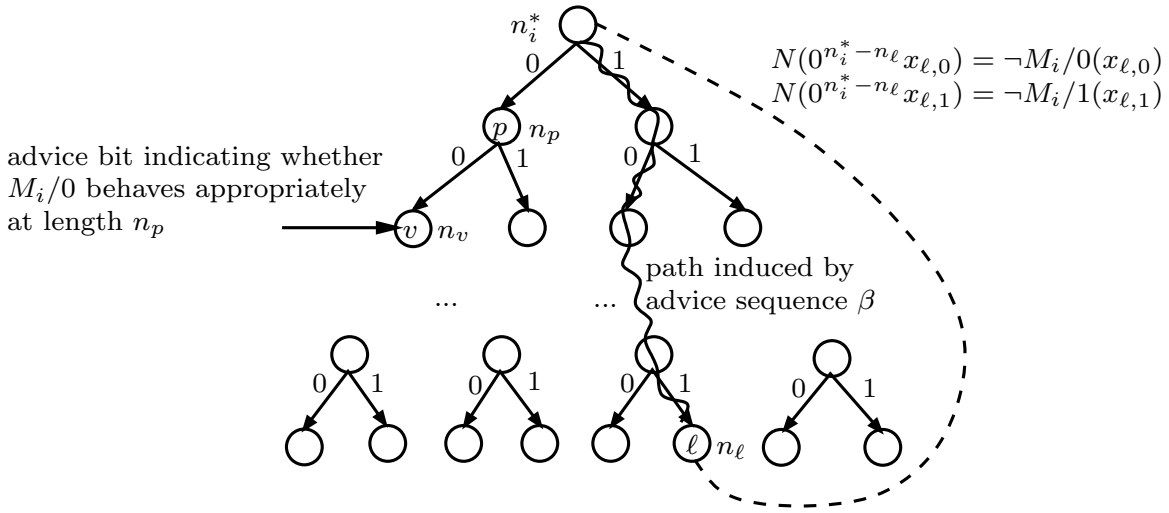


Figure 3.2 Illustration of N's execution for generic semantic models, shown for the case where M_i receives 1 bit of advice. Solid lines indicate that on the smaller input, N simulates M_i on padded inputs of the larger length, using the advice bit specified on the arrow. The dashed line indicates that on padded inputs of length n_i^*, N complements the behavior of M_i on inputs corresponding to the leaves of the tree of input lengths.

advice string that causes M_i to appropriately copy N's behavior at length n_i^*, N/α copies that behavior to a slightly smaller input length.

As with delayed diagonalization on syntactic models, we repeat the same process to copy the behavior at length n_i^* to smaller and smaller inputs. This is best visualized by a tree of input lengths with n_i^* being the root node. The tree node corresponding to n_i^* has one child input length for each possible advice string at length n_i^* as described above. Each of these input lengths is also considered a node of the tree of input lengths with as many children as different advice strings at that length. This is repeated until reaching a level of leaf nodes. The tree of input lengths is illustrated in Figure 3.2. We now give more details on the construction.

First consider an internal node corresponding to some input length n_p. This node must have a child node for all possible advice strings at length n_p. Each of these child nodes is responsible for simulating M_i on inputs of length n_p using a different advice string. Let n_v be a child node of node n_p that is responsible for simulating M_i with advice string b.

The advice string b can be efficiently computed from the input length n_v – we describe an encoding scheme with this property in the next section. N's advice bit at length n_v indicates whether M_i/b behaves appropriately at length n_p. If the advice bit is 1, then on inputs x of length n_v, N simulates $M_i(0^{n_p-n_v}x)/b$; otherwise, N abstains and rejects all inputs of length n_v.

Consider an input length n_ℓ that corresponds to a leaf node ℓ in the tree. It is the responsibility of the root node of the tree to complement the behavior of M_i on inputs of length n_ℓ for all possible advice strings for input length n_ℓ. The complementation is realized using inputs $x_{\ell,b}$ of length n_ℓ for each possible advice string b at length n_ℓ. The inputs are chosen in such a way that they are distinct for all leaf nodes ℓ and advice strings b and such that they remain distinct when they are padded with zeros to length n_i^*. In particular, we set $x_{\ell,b} = 10^{n_\ell-1-|b|}b$, and $N(0^{n_i^*-n_\ell}x_{\ell,b})$ complements $M_i(x_{\ell,b})/b$. Note that n_i^* must be large enough so that space $s(n_i^*)$ suffices for N to *safely complement* the behavior of M_i on all leaf nodes. Recall the definition of a safe complementation.

DEFINITION 3.1 (SAFE COMPLEMENTATION) *Fix a semantic model of computation and let N and M be two machines in the computable enumeration of the underlying syntactic model. N on input y safely complements M on input x if $N(y)$ satisfies the promise (even if $M(x)$ does not), and if $M(x)$ satisfies the promise then $N(y) \neq M(x)$.*

As discussed at the beginning of this chapter, one way to achieve this is for N at length n_i^* to deterministically simulate M_i at the leaf nodes and flip the result. For Arthur-Merlin games this can be accomplished with a linear-exponential overhead in space, for unambiguous machines a quadratic overhead is sufficient [Sav70], and for bounded-error randomized machines an overhead with exponent 3/2 is sufficient [SZ99].

On all input lengths in $[n_i, n_i^*]$ that are not used in the tree of input lengths, N acts trivially by rejecting all inputs of that length.

We claim that N/α constructed in this way satisfies the promise on all inputs and differs from M_i/β for all machines M_i and advice sequences β for which M_i/β behaves appropriately. N/α satisfies the promise on all inputs by setting the advice bits appropriately on all nodes

of the tree. Suppose there is an advice sequence β causing M_i to compute the same language as N while satisfying the promise on all inputs and using $s(n)$ space. The construction of the tree guarantees that there is a chain of inputs present in the tree for this advice sequence from the root node down to a leaf node. If we assume M_i/β computes the same language as N on all these inputs, then the complementary behavior initiated at the root node is copied down all the way to the leaf node, which is impossible. More precisely, let h be the height of the tree and $n_i^* = n_{i,h} > n_{i,h-1} > n_{i,h-2} > \ldots > n_{i,0} = n_\ell$ denote the path from the root of the tree to the leaf ℓ induced by β. By construction, we have for $b = \beta_{n_\ell}$ that

$$\begin{aligned}\neg M_i(x_{\ell,b})/b = N(0^{n_{i,h}-n_\ell}x_{\ell,b})/\alpha = M_i(0^{n_{i,h}-n_\ell}x_{\ell,b})/\beta_{n_{i,h}} = \\ N(0^{n_{i,h-1}-n_\ell}x_{\ell,b})/\alpha = M_i(0^{n_{i,h-1}-n_\ell}x_{\ell,b})/\beta_{n_{i,h-1}} = \ldots = \\ N(0^{n_{i,1}-n_\ell}x_{\ell,b})/\alpha = M_i(0^{n_{i,1}-n_\ell}x_{\ell,b})/\beta_{n_{i,1}} = N(x_{\ell,b})/\alpha = M_i(x_{\ell,b})/b,\end{aligned}$$

which is a contradiction. We conclude that N/α succeeds in differing from each machine M_i which satisfies the promise and uses at most $s(n)$ space on all inputs. It remains to show that N needs space not much more than $s(n)$ and determine the amount of advice the construction can handle.

3.2.2 Analysis

In this section, we give remaining details of the construction of the copying tree, ensuring N/α uses small space and determining the amount of advice bits that can be given M_i, proving Theorems 1.9, 1.10, and 1.11.

For clarity we focus on the case where $s(n) = \log n$ for now; we consider larger space bounds at the end of this section. Let $a(n)$ denote the amount of advice we allow M_i, and let $\sigma(n)$ be the smallest value such that $\log n$ space computations can be complemented within the model using $\sigma(n)$ space. To ensure that N/α requires not much more than $\log n$ space, we must balance two competing requirements – that n_i^* is large enough to be able to efficiently complement the behavior of the leaf nodes, and that each node in the tree is close enough to its parent node to be able to simulate it efficiently.

Each node in the tree corresponds to some input length in the interval $[n_i, n_i^*]$, where n_i^* corresponds to the root of the tree. We separate the tree into consecutive levels. We call the bottom-most level of leaf nodes "level 0", its parent nodes "level 1", and so on. Let h denote the number of non-leaf levels in the tree, so the root node at input length n_i^* is at level h.

To ensure the simulations take $O(\log n)$ space, we impose the restriction that a node n_v's parent n_p can correspond to an input length that is only polynomially larger: N incurs only a constant factor overhead in simulating M_i, and if M_i uses space at most $\log n$ and $n_p \leq n_v^c$ for some constant c, then the simulation requires $O(\log n_p) = O(\log(n_v^c)) = O(\log n_v)$ space. We ensure the input length of a node is separated from its parent's input length by at most a polynomial amount as follows. For each $j = 0, 1, ..., h-1$, we embed level j of the tree in the interval $[n_i^{c^j}, n_i^{c^{j+1}} - 1]$ for some constant c to be chosen later. Thus if a node has input length n_v, its parent has input length $n_p < (n_v)^{c^2}$.

Because each internal node must have as many children as possible advice strings at that length, each internal node in the tree would have a different degree. We simplify the construction and analysis by rounding up the amount of advice given to M_i to ensure that all nodes in the same level have the same degree. That is, all nodes in level j have degree $2^{a(n_i^{c^{j+1}})}$.

For completeness, we give the encoding scheme that identifies which input lengths in the tree correspond to a given node's children. Consider an input length n that is an internal node at level j in the tree, so $n = n_i^{c^j} + \Delta$ for some $\Delta < n_i^{c^{j+1}} - n_i^{c^j}$. We must specify which input lengths in level $j-1$ correspond to n's children for each advice string of length $a(n_i^{c^{j+1}})$. We use the most obvious encoding scheme, filling in the children for level j nodes from left to right within level $j-1$. That is, n's child corresponding to advice string b is at input length $n_i^{c^{j-1}} + 2^{a(n_i^{c^{j+1}})} \cdot \Delta + b$. This encoding scheme allows N to efficiently determine where any given input length falls within the tree, so N can efficiently determine which padded input and with which advice string it is to simulate M_i.

The above encoding scheme can only be realized if the interval $[n_i^{c^j}, n_i^{c^{j+1}} - 1]$ contains as many input lengths as there are nodes in level j of the tree, for each $j = 0, 1, 2, ..., h-1$.

The bottom-most level contains the largest number of nodes and has the smallest number of input lengths to work with, so the tree can be embedded into $[n_i, n_i^*]$ exactly when the bottom-most level fits within the interval $[n_i, n_i^c - 1]$. Because we have rounded up the degrees of the nodes, we get a simple expression for the number of leaf nodes in the tree: $2^{a(n_i^{c^h})} \prod_{j=2}^{h} 2^{a(n_i^{c^j})}$. By taking logarithms, there are enough input lengths in level 0 for these nodes exactly when

$$a(n_i^{c^h}) + \sum_{j=2}^{h} a(n_i^{c^j}) \leq \log(n_i^c - n_i). \tag{3.1}$$

Now consider the space usage of the construction. We have already guaranteed the simulations represented by the tree can be performed using $O(\log n)$ space. We must also ensure that the root node operates in $O(\log n_i^*)$ space. Because the root must complement all leaf nodes, the root node runs in $O(\log n_i^*)$ space if

$$\log n_i^* = \Omega(\sigma(n_i^c)). \tag{3.2}$$

If we can simultaneously satisfy both equation 3.1 and equation 3.2, we ensure the construction can be implemented correctly and in space $s'(n)$ for any $s'(n) = \omega(\log n)$. We now finish the analysis separately for two cases.

1. For some semantic models, such as Arthur-Merlin games, the most efficient safe complementation known within the model incurs a linear-exponential overhead in space. We handle such models using Theorem 1.9.

2. For some semantic models, such as unambiguous machines and bounded-error randomized machines, a safe complementation within the model is known with only a polynomial overhead in space. We handle these models using Theorem 1.10.

3.2.2.1 Complementation with Linear-Exponential Overhead (Theorem 1.9)

We first complete the analysis for the more general setting where there is a safe complementation within the model with a linear-exponential overhead in space, which is typically

achieved by using a deterministic simulation of the model and flipping the result. We now assume a semantic model where $\log n$ space computations can be complemented within the model in space $O(n^{d'})$ for some constant d'. In this case, equation 3.2 becomes

$$\log n_i^* = \log n_i^{c^h} = \Omega(n_i^{cd'}). \tag{3.3}$$

In other words, $n_i^* = 2^{\Omega(n_i^{cd'})}$, and we set $h = \lceil \log(\frac{n_i^{cd'}}{\log n_i}) / \log c \rceil = \Omega(\log n_i)$ to ensure equation 3.3. To fit the leaves of a tree that has depth $\Omega(\log n_i)$ within the interval $[n_i, n_i^c - 1]$, the degree at each node can be at most some constant. Let $a(n) = k$ for some constant k. Then equation 3.1 becomes

$$k + \sum_{j=2}^{h} k = h \cdot k \leq \log(n_i^c - n_i). \tag{3.4}$$

As the right-hand side grows faster with c than the left-hand side, we can pick c sufficiently large so that both equations 3.3 and 3.4 are satisfied. The construction works for any constant k, and we have shown that N/α uses $O(\log n)$ space where the constant only depends on s and the control characteristics of M_i and k.

We ensure that N/α has enough space to complete the construction by allocating the intervals of input lengths so that for each machine M_i and constant k, infinitely many of the intervals are allocated to N/α working against M_i with k bits of advice. We note that given an input x of length n, the computation of deciding which interval of input lengths $[n_i, n_i^*]$ that n lies within can be done space-efficiently. With $s'(n) = \omega(\log n)$ space available, N/α eventually has enough space to successfully complete the construction against M_i with k bits of advice. For intervals of input lengths where N/α does not have enough space to complete the construction, we set the advice bits to 0 over the entire interval, and N immediately rejects ensuring N/α does not go over its space quota. We point out that this use of N's advice bit obviates the need for $s'(n)$ to be space-constructible.

We have proved Theorem 1.9 for the case of semantic models such as Arthur-Merlin games. Section 3.2.3 contains a precise statement of the properties needed of a semantic model for our proof of Theorem 1.9 to apply.

3.2.2.2 Complementation with Polynomial Overhead (Theorem 1.10)

We now complete the analysis for semantic models where there is a safe complementation within the model with only a polynomial overhead in space. We assume now that M_i's behavior at length n while using space $\log n$ can be complemented within the model using $\sigma(n) = O(\log^d n)$ space. For example, $d = 2$ for unambiguous machines [Sav70] and $d = 3/2$ for bounded-error randomized machines [SZ99]. Thus equation 3.2 becomes $\log n_i^* = \Omega(\log^d(n_i^c))$, or equivalently, $n_i^* = 2^{\Omega(\log^d(n_i^c))}$. Now consider the first term of equation 3.1. Plugging in the above equality for n_i^* tells us that we must at least satisfy $a(2^{\gamma \log^d(n_i^c)}) < \log(n_i^c)$ for some constant $\gamma > 0$ if we are to satisfy equation 3.1. This imposes an upper bound on $a(n)$ of $O(\log^{1/d} n)$.

In fact, we can achieve $a(n) = \Theta(\log^{1/d} n)$ while still satisfying both equations 3.1 and 3.2, as follows. Let $a(n) = k \log^{1/d} n$ for some integer $k > 0$. Substituting into equation 3.1 yields

$$k \log^{1/d}(n_i^{c^h}) + k \sum_{j=2}^{h} \log^{1/d}(n_i^{c^j}) \leq \log(n_i^c - n_i). \tag{3.5}$$

For technical reasons, we aim to satisfy equation 3.2 by ensuring

$$c^3 \log n_i^* = c^3 \log(n_i^{c^h}) \geq \log^d(n_i^c), \tag{3.6}$$

which we satisfy by setting $h = \lceil (\log(c^{d-3} \log^{d-1} n_i)/\log c \rceil$.

Using the fact that $h \leq \frac{\log(c^{d-3} \log^{d-1}(n_i))}{\log c} + 1$, we bound the first term of the left-hand side of equation 3.5.

$$k \log^{1/d}(n_i^{c^h}) = k(c^h \log n_i)^{1/d} \leq k(c^{d-2} \log^d n_i)^{1/d} = kc^{(d-2)/d} \log n_i.$$

Assuming we pick c large enough such that $c^{1/d} - 1 \geq 1$, we now bound the second term.

$$\begin{aligned} k \textstyle\sum_{j=2}^{h} \log^{1/d}(n_i^{c^j}) &= k \frac{c^{2/d}(c^{(h-1)/d} - 1)}{c^{1/d} - 1} \log^{1/d} n_i \\ &\leq kc^{2/d}(c^{h-1})^{1/d} \log^{1/d} n_i \\ &\leq kc^{2/d}(c^{d-3} \log^{d-1} n_i)^{1/d} \log^{1/d} n_i \\ &= kc^{(d-1)/d} \log n_i. \end{aligned}$$

Adding up these two values satisfies equation 3.5 for large enough c.

We have shown that the space usage of N/α is $O(\log n)$ where the constant only depends on s and the control characteristics of M_i and k. As with Theorem 1.9, we allocate the intervals of input lengths so that for each machine M_i and constant k, N/α attempts the construction against M_i with k advice bits. With $s'(n) = \omega(\log n)$ space available, N/α eventually has enough space to complete the construction against M_i with k advice bits, completing the proof of Theorem 1.10. Among others, Theorem 1.10 applies to semantic models such as unambiguous machines and bounded-error randomized machines. Section 3.2.3 contains a precise statement of the properties required of a model for our proof of Theorem 1.10 to apply.

3.2.2.3 Larger Space Bounds (Theorem 1.11)

So far we have only considered the case with $s(n) = \log n$, where we have shown separation results that are tight with respect to space – that $s'(n)$ space suffices to differ from $s(n)$ space machines for any $s'(n) = \omega(s(n))$. Tightness with respect to space follows from satisfying: (1) each node of the copying tree is close enough to its parent so the simulations incur only a constant overhead in space, and (2) nodes are far enough apart so the height of the tree required to allow the root node to complement leaf nodes does not result in more leaf nodes than input lengths allocated in the bottom-most level of the copying tree. In the general setting where safe complementation requires a linear-exponential overhead in space, these cannot be simultaneously met for super-logarithmic space bounds – our construction still works but gives a result that is not tight with respect to space for $s(n) = \omega(\log n)$.

In the setting where safe complementation incurs only a polynomial overhead in space, we have more wiggle room and can derive a tight separation for space bounds up to any polynomial. In fact, an examination of the analysis for Theorem 1.10 shows the construction remains tight with respect to space for $s(n)$ any poly-logarithmic function. For larger space bounds the construction as given is not tight, but we can make some modifications to handle space bounds up to polynomial. The main idea is to place nodes of the copying tree closer

to their parent nodes to satisfy (1); this can be achieved for space bounds up to polynomial without breaking (2).

We now prove Theorem 1.11. Fix a semantic model where M_i's behavior while it uses $s(n) = \Omega(\log n)$ space can be safely complemented within the model using space $O(s(n)^d)$. Consider a space bound $s(n) = n^r$ for some constant $r > 0$. We would like to demonstrate a language computable within the model using $s'(n)$ space and one bit of advice that is not computable using $s(n)$ space and $O(1)$ bits of advice, for any $s'(n) = \omega(s(n))$. As alluded to above, we accomplish this by modifying the generic construction so that each level of the copying tree is embedded within a smaller interval of input lengths: we embed level j of the copying tree within input lengths $[c^j n_i, c^{j+1} n_i - 1]$ where c is a constant we may choose. This ensures that for each n_v, $n_p < c^2 \cdot n_v$ and performing the simulation of M_i on inputs of length n_p uses space $O(n_p^r) = O((c^2 \cdot n_v)^r) = O(c^{2r} n_v^r) = O(n_v^r) = O(s(n_v))$. Let h be the height of the copying tree. To ensure the root node has sufficient space to complement the leaf nodes, it must be that

$$(c^h n_i)^r = \Omega(((c \cdot n_i)^r)^d),$$

which we achieve by setting $h = \lceil \log(n_i^{d-1}) / \log c \rceil$. If M_i is allowed k advice bits the total number of leaf nodes is $2^{h \cdot k} = n_i^{k(d-1)/\log c}$, which must be smaller than $c \cdot n_i - n_i$ to ensure the leaf nodes fit within the range of input lengths we have allocated for them. We can choose c large enough to ensure this holds. As with Theorems 1.9 and 1.10, we allocate the intervals of input lengths so that for each machine M_i and constant k, N/α attempts the construction against M_i with k advice bits infinitely many times. With $s'(n) = \omega(n^r)$ space available, N/α eventually has enough space to complete the construction against M_i with k advice bits, ensuring N/α differs from M_i/β if M_i/β satisfies the promise and uses space at most $s(n) = n^r$ on all inputs. We have thus proved Theorem 1.11.

The main idea of the proof of Theorem 1.11 was to shrink the separation between each node and its parent until a node can space-efficiently simulate its parent. This can be achieved for any space bound that is polynomially bounded and sufficiently smooth (in the

sense that it does not have long intervals of slow growth followed by drastic jumps) by choosing the input lengths for the copying tree appropriately.

3.2.3 Generic Semantic Models

Consider the properties of the machine model used in the above analysis of Theorems 1.9, 1.10, and 1.11 and those required for the proof of Theorem 1.12 in Section 3.1. First, N can simulate any other machine M_i with only a constant factor overhead in space. This is needed to ensure that N needs only slightly more space than M_i. Second, N can efficiently perform certain deterministic tasks – e.g., for an input of length n, N performs arithmetic to determine which interval of inputs $[n_i, n_i^*]$ and which node within the copying tree n corresponds to. As these requirements are quite modest, any "reasonable" semantic model satisfies them. Here is a precise statement.

Definition 3.4 (reasonable semantic model) *Fix a semantic model of computation with $(M_i)_{i=1,2,3,\ldots}$ the computable enumeration of the underlying syntactic model. The semantic model is called* reasonable *if it satisfies the following conditions:*

1. *There exists a machine U in the underlying syntactic model such that for each $i \geq 1$, $x \in \{0,1\}^*$, and $s \geq s_{M_i}(x)$, U satisfies the promise on input $(M_i, x, 0^s)$ whenever M_i satisfies the promise on input x, and if so, $U(M_i, x, 0^s) = M_i(x)$. U must run in space $O(s + \log(|x| + |M_i|))$.*
2. *Let D be a deterministic transducer, i.e., a deterministic machine D that executes and either outputs an answer $a(x)$ or a query $q(x)$ to some machine M. For each such D and machine M_i, there must exist a machine $M_{i'}$ such that on each input x: if $D(x)$ outputs an answer $a(x)$, then $M_{i'}(x) = a(x)$ and satisfies the promise; and if $D(x)$ outputs a query $q(x)$ on which M_i satisfies the promise, then $M_{i'}(x) = M_i(q(x))$ and satisfies the promise. In addition, the space usage of $M_{i'}(x)$ must be $O(s_D(x))$ when $D(x)$ outputs an answer, and must be $O(s_D(x) + s_{M_i}(q(x)))$ when $D(x)$ outputs a query $q(x)$.*

If this holds, we say the model is efficiently closed under deterministic transducers.

The analysis of Theorems 1.9, 1.10, and 1.11 in Section 3.2.2 was broken up into two cases depending on the efficiency with which safe complementation is possible. We formalize the space overhead of a safe complementation in the model as follows.

Definition 3.5 (space overhead of safe complementation) *Fix a reasonable semantic model of computation with U the machine given by part (i) of Definition 3.4. Let σ be a function. We say the model can be safely complemented with space overhead σ if there is a machine S in the underlying enumeration of machines such that: S satisfies the promise on every input, $S(y) = \neg U(y)$ for every input $y \in \{0,1\}^*$ on which U satisfies the promise, and S runs within space $\sigma(s + \log(|x| + |M_i|))$ on input $y = (M_i, x, 0^s)$.*

Theorem 1.9 applies to any reasonable semantic models that can be safely complemented with $\sigma(m) = 2^{O(s(m))}$. As mentioned in the introduction, this includes a wide class of semantic models, and in particular includes models such as Arthur-Merlin games, for which the simple translation argument of [KV87] does not apply.

Theorems 1.10 and 1.11 apply to any reasonable semantic model that has a more efficient safe complementation, namely with $\sigma(m) = O(m^d)$ for some constant d. Note that due to the space-bounded derandomization of [SZ99], randomized two-sided, one-sided, and zero-sided error machines can be safely complemented with space overhead $\sigma(m) = O(m^{3/2})$. Unambiguous machines can be safely complemented with space overhead $\sigma(m) = O(m^2)$ due to Savitch's Theorem [Sav70]. We point out that it is unlikely that Arthur-Merlin games can similarly be safely complemented by a deterministic simulation with space overhead $m^{O(1)}$: a deterministic simulation of Arthur-Merlin games with polynomial overhead in space would imply that NC lies in DSPACE($\log^d n$) for some constant d [FL93].

The proof in Section 3.1 of a hierarchy for the promise problems computed by semantic models (Theorem 1.12) applies to an even broader range of semantic models, namely reasonable semantic models that can be safely complemented with space overhead σ for any computable σ and time overhead any computable function.

We point out that we have not assumed any efficiency requirements for the computable enumeration of machines $(M_i)_{i=1,2,3,\ldots}$ in Definition 3.4. Each of the particular machine models we have discussed has a very efficient enumeration – namely all binary strings – because under any encoding of machines into binary strings we can map unused strings to some default machine. However, being able to enumerate the machines efficiently is not a requirement of our results; if the enumeration $(M_i)_{i=1,2,3,\ldots}$ is space inefficient we can modify the locations of the intervals of inputs $[n_i, n_i^*]$ such that enumerating up to machine i can be done in $\log n_i$ space.

Chapter 4

Hierarchy Theorems for Randomized Models

In Chapter 3, we used techniques that were very general to prove hierarchy theorems for the promise problems computed by generic semantic models of computation and for languages computed by generic semantic models that use one bit of advice. In this chapter, we strengthen the latter for the particular case of two-, one-, and zero-sided error randomized machines using techniques tailored to these models. In particular, we prove the following results, restated here for convenience.

First, Theorem 1.7 shows that we can construct a two-sided error machine that takes one bit of advice and differs from two-sided error machines that take *many* bits of advice and use slightly less space.

THEOREM 1.7 *Let $s(n)$ be any space-constructible monotone function such that $s(n) = \Omega(\log n)$, and let $s'(n)$ be any function that is $\omega(s(n + as(n)))$ for all constants a. There exists a language computable by two-sided error randomized machines using $s'(n)$ space and one bit of advice that is not computable by two-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.*

For typical space bounds, the statement of Theorem 1.7 can be simplified. In particular for monotone space bounds we show the following.

COROLLARY 4.1 *Let $s(n)$ be any space-constructible monotone function such that $s(n) = \Omega(\log n)$ and $s(n) = O(n)$, and let $s'(n)$ be any function such that $s'(n) = \omega(s(n+1))$. There exists a language computable by two-sided error randomized machines using $s'(n)$ space and*

one bit of advice that is not computable by two-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.

For typical space bounds s that are $O(n)$, $s(n+1) = O(s(n))$ so that Corollary 4.1 gives a tight separation in space – any super-constant gap suffices.

We will also show similar results for one- and zero-sided error randomized machines. The following two results show that a *zero*-sided error machine with one bit of advice can diagonalize against *one*-sided error machines that use slightly less space and many bits of advice.

THEOREM 1.8 *Let $s(n)$ be any space-constructible monotone function such that $s(n) = \Omega(\log n)$, and let $s'(n)$ be any function that is $\omega(s(n + as(n)))$ for all constants a. There exists a language computable by zero-sided error randomized machines using $s'(n)$ space and one bit of advice that is not computable by one-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.*

COROLLARY 4.2 *Let $s(n)$ be any space-constructible monotone function such that $s(n) = \Omega(\log n)$ and $s(n) = O(n)$, and let $s'(n)$ be any function that is $\omega(s(n+1))$. There exists a language computable by zero-sided error randomized machines using $s'(n)$ space and one bit of advice that is not computable by one-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.*

We first describe the high-level strategy used for these results in Section 4.1. Most portions of the construction are the same for all the results, so we keep the exposition general. In Section 4.2 we introduce the notion of a recovery procedure – the key new technical ingredient of the proofs – and develop intuition for how these procedures arise naturally within our arguments. In Section 4.3 we develop an appropriate recovery procedure for use in the two-sided error setting of Theorem 1.7 and Corollary 4.1. In Section 4.4 we develop a recovery procedure for use in the one- and zero-sided error setting of Theorem 1.8 and Corollary 4.2. In Section 4.5 we give a complete and detailed description of the construction

using the recovery procedures developed in Section 4.3 and Section 4.4. Finally, in Section 4.6 we complete the analysis of the theorems and corollaries.

4.1 Proof Outline

We aim to construct a randomized machine N and advice sequence α witnessing Theorems 1.7 and 1.8 for some space bounds $s(n)$ and $s'(n)$. N/α should always satisfy the promise, run in space $s'(n)$, and differ from M_i/β for randomized machines M_i and advice sequences β for which M_i/β *behaves appropriately.* We defined this notion earlier for the case of generic semantic models (Definition 3.3). For convenience, we repeat the definition here for the particular case of bounded-error randomized machines.

Definition 4.3 (appropriate behavior of bounded-error machines) *In the context of two-sided (respectively one- or zero-sided) error randomized machines and given an underlying space bound $s(n)$, a randomized machine M_i with advice sequence β* behaves appropriately *if M_i/β satisfies the promise of two-sided (respectively one- or zero-sided) error and uses at most $s(n)$ space on all inputs.*

As with delayed diagonalization, for each M_i we allocate an interval of input lengths $[n_i, n_i^*]$ on which to diagonalize against M_i. That is, for each machine M_i and advice sequence β such that M_i/β behaves appropriately, there is an $n \in [n_i, n_i^*]$ such that N/α and M_i/β decide differently on at least one input of length n. The construction consists of three main parts.

(1) Reduce the complement of the computation of M_i on inputs of length n_i to instances of a hard language L of length m_i.

(2) Perform a delayed computation of L at length m_i on padded inputs of length n_i^*.

(3) Copy this behavior to smaller and smaller inputs down to input length m_i using suitable "recovery procedures" for the hard language.

These ensure that if M_i/β behaves appropriately, either N/α differs from M_i/β on some input of length larger than m_i, or N/α computes L at length m_i allowing N/α to differ from M_i/b for all possible advice strings b at length n_i.

Let us point out how this construction differs from the one given in Chapter 3 that applies to any "reasonable" semantic model of computation. In that construction (1) and (2) from above are replaced by a delayed complementation at length n_i^* of M_i's behavior on small input lengths, and (3) is replaced by copying this behavior through a tree of input lengths through simple simulations of M_i where the branching factor at each node of the copying tree corresponds to the number of possible advice strings M_i might take at that input length. The advice is used only to instruct N whether to perform a given simulation or not. That construction is very generic and only requires very basic properties of the semantic model, but the need to fit the copying tree within the input lengths $[n_i, n_i^*]$ places a limit on the branching factor at each node of the tree and thus of the amount of advice that can be given M_i. In contrast, the construction in this chapter takes advantage of specific properties of two-, one-, and zero-sided error machines to develop recovery procedures for hard languages, with the end result of being able to handle more advice.

An illustration of the completed construction of this chapter is given in Figure 4.1. The reader is encouraged to refer to Figure 4.1 as we develop the construction in subsequent sections.

4.2 The Need for Advice and Recovery Procedures

In this section, we begin by assuming a hard language L as in (1) above and develop an intuition for why advice and recovery procedures are needed to achieve (3). Let us first try to develop delayed diagonalization without advice to see where problems arise due to working in a semantic model and how advice and recovery procedures can be used to fix those.

On an input x of length n_i, N reduces the complement of $M_i(x)$ to an instance of L of length m_i. Because N must run in space not much more than $s(n)$ and we do not know how

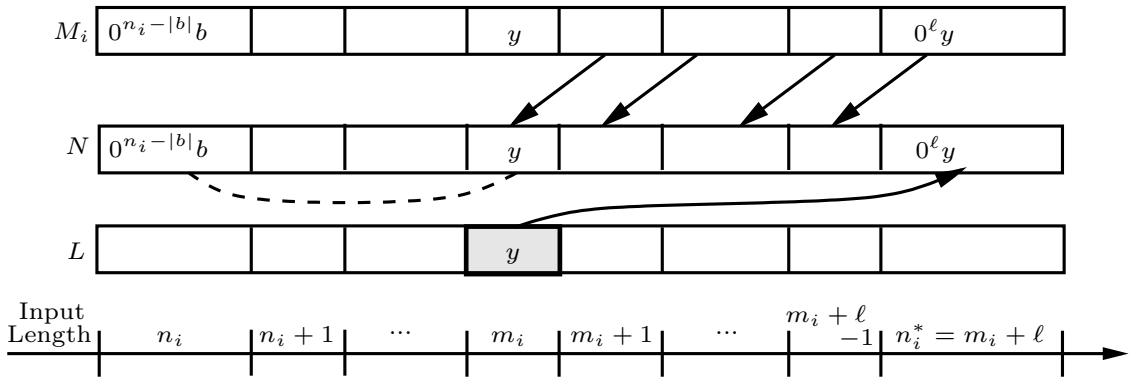


Figure 4.1 Illustration of the construction for Theorems 1.7 and 1.8. The solid arrow from y indicates that on input $0^\ell y$, N deterministically computes $L(y)$ for each y of length m_i. The diagonal arrows indicate that for $\ell' \in [0, \ell-1]$, on input $0^{\ell'}y$ with advice bit 1, N attempts to compute $L(y)$ by using the recovery procedure and making queries to M_i on padded inputs of one larger length. The dashed line indicates that on input $0^{n_i-|b|}b$ with advice bit 1, N complements $M_i(0^{n_i-|b|}b)/b$ by reducing to an instance y of L and simulating $N(y)$.

to compute the hard languages we use with small space, N cannot directly compute L at length m_i. However, L can be computed at length m_i within the space N is allowed to use on much larger inputs. Let n_i^* be large enough so that L at length m_i can be deterministically computed in space $s(n_i^*)$. We let N at length n_i^* perform a *delayed computation* of L at length m_i as follows: on inputs of the form $0^\ell y$ where $\ell = n_i^* - m_i$ and $|y| = m_i$, N uses the above deterministic computation of L on input y to ensure that $N(0^\ell y) = L(y)$.

Since N performs a delayed computation of L, M_i must as well – otherwise N already computes a language different than M_i. We would like to bring this delayed computation down to smaller padded inputs. The first attempt at this is the following: on input $0^{\ell'}y$, N simulates $M_i(0^{\ell'+1}y)$, for all $0 \leq \ell' < \ell$. If M_i behaves appropriately and performs the initial delayed computation, then $N(0^{\ell-1}y) = M_i(0^\ell y) = L(y)$, meaning that N satisfies the promise and performs the delayed computation of L at length m_i at an input length one smaller than before. However, M_i may not behave appropriately on inputs of the form $0^\ell y$; in particular M_i may fail to satisfy the promise, in which case N would also fail to satisfy the promise by performing the simulation. If M_i does not behave appropriately, N does not need

to consider M_i and could simply abstain from the simulation. If M_i behaves appropriately on inputs of the form $0^\ell y$, it still may fail to perform the delayed computation. In that case N has already diagonalized against M_i at input length $m_i + \ell$ and can therefore also abstain from the simulation on inputs of the form $0^{\ell-1}y$.

N has insufficient resources to determine on its own if M_i behaves appropriately and performs the initial delayed computation. Instead, we give N one bit of advice at input length $m_i + \ell - 1$ indicating whether M_i behaves appropriately and performs the initial delayed computation at length $n_i^* = m_i + \ell$. If the advice bit is 0, N acts trivially at this length by always rejecting inputs. If the advice bit is 1, N performs the simulation so $N(0^{\ell-1}y)/\alpha = M_i(0^\ell y) = L(y)$.

If we give N one bit of advice, we should give M_i at least one advice bit as well. Otherwise, the hierarchy result is not fair (and is trivial). Consider how allowing M_i advice affects the construction. If there exists an advice string b such that M_i/b behaves appropriately and $M_i(0^\ell y)/b = L(y)$ for all y with $|y| = m_i$, we set N's advice bit for input length $m_i + \ell - 1$ to be 1, meaning N should copy down the delayed computation from length $m_i + \ell$ to length $m_i + \ell - 1$. Note, though, that N does not know for which advice b the machine M_i/b appropriately performs the delayed computation at length $m_i + \ell$. N has at its disposal a list of machines, namely M_i with each possible advice string b, with the guarantee that at least one M_i/b behaves appropriately and $M_i(0^\ell y)/b = L(y)$ for all y with $|y| = m_i$. With this list of machines as its primary resource, N wishes to ensure that $N(0^{\ell-1}y)/\alpha = L(y)$ for all y with $|y| = m_i$ while satisfying the promise and using small space.

Aside from the padding involved, N can appropriately perform the above delayed computation when given a procedure that takes as input a string y of length m_i and list of randomized machines, and then appropriately recovers $L(y)$ as long as at least one of the input machines behaves appropriately and computes L at length m_i. We call the latter a recovery procedure for L at length m_i.

Definition 4.4 (recovery procedure) *A two-sided error (respectively one- or zero-sided error)* recovery procedure *for a language L at length m is a machine Rec which takes*

as input $z = (y, P_1, ..., P_q)$, where y is a string of length m and P_1, ..., P_q are randomized Turing machines, such that the following holds. If there exists $d \in \{1, 2, ..., q\}$ such that $P_d(y')$ satisfies the promise of two-sided error (respectively one- or zero-sided error) and $P_d(y') = L(y')$ on all inputs y' of length m then Rec on input z satisfies the promise of two-sided error (respectively one- or zero-sided error) and $Rec(z) = L(y)$.

Typically, the recovery procedure *Rec* at length m runs the machines P_j on various inputs of length m. The difficulty is that *Rec* does not know a priori which machine appropriately computes L at length m, and *Rec* must appropriately compute L no matter the behavior of the remaining machines that are given as input.

We point out that for Theorem 1.7, the recovery procedure may have two-sided error, while for Theorem 1.8, the recovery procedure must have zero-sided error even though it is only guaranteed a machine P_d that behaves appropriately with one-sided error. Recovery procedures are the main technical ingredients needed for our results on bounded-error randomized machines. We develop the recovery procedures in Sections 4.3 and 4.4 and complete the construction in Section 4.5.

4.3 Two-sided Error Recovery Procedure – Computation Tableau Language

In this section we define the hard language L and recovery procedure for L that are used in Section 4.5 to complete the proof of Theorem 1.7. When working against machine M_i over the interval of input lengths $[n_i, n_i^*]$, L must satisfy the following conditions. (1) If M_i behaves appropriately on inputs of length n_i, then the complement of its behavior can be space-efficiently reduced to L at some length $m_i \in [n_i, n_i^*]$. (2) There exists a space-efficient two-sided error recovery procedure for L at length m_i.

Recall from Section 2.2 that given M_i, there is a deterministic Turing machine D such that for each input x, $D(x) = 1$ if $\Pr_r[M_i(x; r) = 1] \geq \frac{1}{2}$ and $D(x) = 0$ otherwise, $D(x)$ uses $2^{as(|x|)}$ time for some constant a that only depends on the control characteristics of M_i, and D has a single bit in its configuration at time step $t = 2^{O(s(|x|))}$ that determines acceptance

or rejection. We use the computation tableau language for this deterministic machine D (hereafter written COMP_D) as the hard language L on the interval $[n_i, n_i^*]$.

DEFINITION 4.5 (COMP_D) *Given a deterministic machine D we define the computation tableau language for D as follows. $COMP_D = \{\langle x, t, j\rangle \mid$ the j^{th} bit in the machine's configuration after the t^{th} time step of executing $D(x)$, is equal to 1$\}$.*

We now present a space-efficient recovery procedure for COMP_D.

LEMMA 4.6 *Let $s = \Omega(\log n)$ be space-constructible and D a deterministic time $2^{O(s(m))}$ Turing machine. Then $COMP_D$ has a two-sided error recovery procedure at length m which uses space $O(s(m) + \log|z| + \max_j(s_{P_j}(m)))$ on input $z = (y, P_1, ..., P_q)$, where y is a string of length m, $P_1, ..., P_q$ are randomized Turing machines, and s_{P_j} denotes the space usage of machine P_j.*

We prove Lemma 4.6 in the rest of this section. Let $y = \langle x, t, j\rangle$ be an instance of COMP_D with $|y| = m$ that we wish to compute. Recall that we are guaranteed at least one machine P_d in the list of machines that computes COMP_D at length m with two-sided error. A natural way to determine $\text{COMP}_D(y)$ is to consider each machine P in the list P_1, ..., P_q one at a time and design a test with the following properties.

(i) If $\Pr_r[P(y'; r) = \text{COMP}_D(y')] \geq \frac{2}{3}$ for all y' of length m, then the test declares success with high probability (say with probability at least $\frac{8}{9}$).

(ii) If the test declares success with non-trivial probability (say greater than $\frac{1}{9q}$), then P gives the correct answer of $\text{COMP}_M(y)$ with high probability (say greater than $\frac{9}{16}$).

We call a randomized machine P "good" for a given y' if $P(y')$ is correct with probability at least $\frac{9}{16}$ and "bad" otherwise. Given a test with properties (i) and (ii), the recovery procedure iterates through each machine in the list in turn. We select the first machine P to pass testing, simulate $P(y)$ some number of times and output the majority answer, where the number of simulations of $P(y)$ is large enough to reduce the upper bound on P's error

probability from $\frac{7}{16}$ to $\frac{1}{9}$. By Theorem 2.7, a large enough constant number of simulations suffices. Before describing the tests that achieve (i) and (ii), we first verify that given such tests we in fact compute $\text{COMP}_D(y)$ with probability at least $\frac{2}{3}$. For the procedure to error on input y, at least one of the following bad events has to happen. (a) The machine P_d fails the test. (b) A machine P that is bad for y passes the test. (c) A machine P that is good for y is selected, but the majority vote of the simulations of $P(y)$ gives the incorrect answer. Error condition (a) occurs with probability at most $\frac{1}{9}$ by (i). By (ii), each individual machine P contributes at most probability $\frac{1}{9q}$ to error condition (b), and a union bound over all q machines shows that error condition (b) occurs with probability at most $\frac{1}{9}$. By (ii) and using a large enough constant number of simulations of $P(y)$ as described above, (c) occurs with probability at most $\frac{1}{9}$. A union bound over all three error conditions shows that given a testing procedure with properties (i) and (ii), we fail to compute $\text{COMP}_D(y)$ with probability at most $\frac{1}{9} + \frac{1}{9} + \frac{1}{9} = \frac{1}{3}$.

The technical heart of the recovery procedure is the testing procedure to select a good machine. This test is based on the local checkability of computation tableaux – the j^{th} bit of the configuration of $D(x)$ in time step $t > 0$ is determined by a constant number of bits from the configuration in time step $t - 1$, each of which can be determined within small space. For each bit position (t, j) of the tableau with $t > 0$, this gives a local consistency check – make sure that the value P claims for $\langle x, t, j\rangle$ is consistent with the values P claims for each of the bits of the tableau that this bit depends on. We implement this intuition as follows.

1. We test that for all positions in the tableau on input x, P's acceptance probability stays bounded away from $\frac{1}{2}$.

   More specifically, for each possible t' and j', we simulate $P(\langle x, t', j'\rangle)$ a number τ times (to be determined below) and fail the test if the fraction of accepting computation paths of $P(\langle x, t', j', \rangle)$ lies in the range $[3/8, 5/8]$.

```
Input: y = ⟨x, t, j⟩ of length m; machines P_1, P_2, ..., P_q
Output: COMP_D(y)
(1)  foreach  d = 1..q                                   Try using P_d to compute COMP_D(y)
(2)     foreach t' and j'                                          Bounded-error checks
(3)        if #accept runs of τ simulations of P_d(⟨x, t', j'⟩) lies in [3/8, 5/8] then goto (1)
           P_d fails
(4)     foreach j'                                  Check base case – start configuration
(5)        A ← majority of τ simulations of P_d(⟨x, 0, j'⟩)
(6)        if A ≠ j'^th bit of start configuration
(7)        then goto (1)                                                        P_d fails
(8)     foreach t' > 0 and j'                                    Local consistency checks
(9)        bit j' in time step t' depends on bits j'_1, j'_2, ..., j'_k in time step t' − 1
(10)       foreach c = 1, 2, ..., k
(11)          A_{j'_c,t'−1} ← majority of τ simulations of P_d(⟨x, t' − 1, j'_c⟩)
(12)       A_{j',t'} ← majority of τ simulations of P_d(⟨x, t', j'⟩)
(13)       if A_{j',t'}, A_{j'_1,t'−1}, A_{j'_2,t'−1}, ..., A_{j'_k,t'−1} violate transition function of D
(14)       then goto (1)                                                        P_d fails
(15)    P_d passed all tests
(16)    return majority of O(1) simulations of P_d(⟨x, t, j⟩)
(17) return 0                                                  No machines passed testing
```

Figure 4.2 Pseudo-code for the two-sided error recovery procedure for the computation tableau language. The list of machines is guaranteed to contain at least one computing COMP_D at length m with two-sided error in space $s(m)$. Lines 2, 4, and 8 loop over all t' and j' valid for D using $2^{O(s(m))}$ time and space, and indices t, j, t', and j' are padded so that all instances of COMP_D of interest are of length m. τ is set to a large enough function that is $O(s + \log q)$ as described in the text.

2. We explicitly check the initial configuration.

   Precisely, for each j', we simulate $P(\langle x, 0, j' \rangle)$ τ times and fail the test if the majority output is not consistent with the initial configuration of D on input x.

3. We run the consistency check for all positions in the tableau with $t' > 0$.

   That is, for each possible $t' > 0$ and j', we do the following. Let j'_1, ..., j'_k be the bits of the configuration in time step $t' - 1$ that bit j' in time step t' depends on. We simulate each of $P(\langle x, t', j' \rangle)$, $P(\langle x, t' - 1, j'_1 \rangle)$, ..., $P(\langle x, t' - 1, j'_k \rangle)$ τ times and fail the test if

the majority values of these simulations are not consistent with the transition function of D.

We argue that this series of tests satisfies (i) and (ii) from above. We first consider (i), so we assume a machine P that computes COMP_D with probability at least $\frac{2}{3}$ on all y' of length m. Then the Chernoff bound (Theorem 2.7) tells us that for τ independent executions of P on a given input y', the probability that at least $\frac{3}{8}$ of the trials gives an incorrect answer is exponentially small in τ. By taking a union bound over all $2^{O(s(m))}$ times that a value of the form $P(y')$ is needed in all tests, we can use τ a large enough linear function in s to ensure that the following occurs with probability at least $\frac{8}{9}$. P passes test 1, and tests 2 and 3 obtain the majority value for $P(y')$ each time this value is needed in these tests. As the majority value of $P(y')$ is correct for each y', P passes tests 2 and 3 in this case, and we have proved (i).

Now consider (ii). Given any randomized machine P, we can associate a computation tableau that P claims for the execution of $D(x)$ with it. Namely, for each t' and j', if $\Pr_r[P(\langle x, t', j' \rangle) = 1] \geq \frac{1}{2}$ then P claims the j'^{th} bit in D's configuration after the t'^{th} time step is equal to 1. Intuitively, if P passes test 1 with non-trivial probability, it must have error bounded away from half by some non-trivial amount; in this case with high probability the majority values of $P(y')$ are obtained for each query of $P(y')$ in tests 2 and 3, allowing these tests to correctly determine the correctness of the tableau claimed by P with high probability.

To make this precise, suppose P outputs its majority value with probability $\frac{1}{2} + \delta$ on some tableau bit, for some δ. By Theorem 2.7, the fraction of τ trials on which P outputs its majority value lies in the range $[\frac{1}{2}, \frac{1}{2} + 2\delta]$ with probability at least $1 - 2e^{\tau\delta^2/4}$. For $\delta = \frac{1}{16}$, we see that P fails test 1 with all but exponentially small probability in τ. By taking τ a large enough logarithmic function in q, if P passes test 1 with probability at least $\frac{1}{9q}$ overall, then for each tableau position P outputs its majority value with probability at least $\frac{1}{2} + \frac{1}{16}$. In this case, by taking τ a large enough function linear in s and logarithmic in q, a union bound ensures that with probability at least $1 - \frac{1}{9q}$ the testing procedure obtains the correct

majority output of P on all queries to P in tests 2 and 3 and correctly determines if P's majority outputs are correct on the tableau bits. Thus if P passes test 1 with probability at least $\frac{1}{9q}$ and tests 2 and 3 with probability at least $\frac{1}{9q}$, its majority values are correct on all tableau bits and it has error at most $\frac{1}{16}$, so we have shown (ii).

Consider the space usage of the recovery procedure, given in pseudo-code in Figure 4.3. The counter for line (1) uses $O(\log q)$ space. The counters for lines (2), (4), and (8) use $O(s(m))$ space because D is a time $2^{O(s(m))}$ machine. The counters of lines (3), (5), (11), and (12) use space $O(s(m)+\log q)$ because $\tau = O(s(m)+\log q)$ and the simulations of these lines use $\max_j(s_{P_j}(m))$ space. Lines (9) and (13) are space efficient because tableau bit $\langle x, t', j'\rangle$ depends on constantly many bits from the previous row, which can be determined and checked space-efficiently. Overall the space usage is $O(s(m) + \log q + \max_j(s_{P_j}(m)))$.

4.4 Zero-sided error Recovery Procedure – Configuration Reachability

In this section we define the hard language L and recovery procedure for L that are used in Section 4.5 to complete the proof of Theorem 1.8. When working against machine M_i over the interval of input lengths $[n_i, n_i^*]$, L must satisfy the following. (1) If M_i behaves appropriately on inputs of length n_i, then the complement of its behavior can be space-efficiently reduced to L at some length $m_i \in [n_i, n_i^*]$. (2) There exists a space-efficient zero-sided error recovery procedure for L at length m_i (even when the recovery procedure is only guaranteed a one-sided error machine P_d that behaves appropriately).

To determine whether $\Pr[M_i(x) = 1] < \frac{1}{2}$ for M_i a one-sided error machine that uses $s(n)$ space, we can ask whether the unique accepting configuration can be reached within $2^{as(|x|)}$ steps from the unique start configuration when M_i executes on input x, where a is a constant that only depends on the control characteristics of M_i. We use the configuration reachability language for M_i as the hard language L. As the recovery procedure works for any randomized machine M, we describe the recovery procedure for CONFIG_M, defined as follows.

DEFINITION 4.7 (CONFIG$_M$) *Given a randomized machine M, we define the configuration reachability language of M as follows. $CONFIG_M = \{\langle x, c_1, c_2, t\rangle \mid$ on input x, if M is in configuration c_1, then configuration c_2 is reachable within t time steps$\}$.*

We now present a space-efficient recovery procedure for CONFIG$_M$.

LEMMA 4.8 *Let $s = \Omega(\log n)$ be space-constructible and M a space $O(s(m))$ randomized machine that always halts. Then $CONFIG_M$ has a* zero-sided error *recovery procedure at length m, which works even when only guaranteed a machine P_d which appropriately computes $CONFIG_M$ with* one-sided error. *The procedure uses space $O(s(m) + \log|z| + \max_j(s_{P_j}(m)))$ on input $z = (y, P_1, ..., P_q)$, where y is a string of length m, $P_1, ..., P_q$ are randomized Turing machines, and s_{P_j} denotes the space usage of P_j.*

We prove Lemma 4.8 in the rest of this section. Let $y = \langle x, c_1, c_2, t\rangle$ be an instance of CONFIG$_M$ with $|y| = m$ that we wish to compute. As we need to compute CONFIG$_M$ with zero-sided error, we can only output a value of "yes" or "no" if we are sure this is correct. The outer loop of our recovery procedure is the following: cycle through each machine P in the list of machines P_1, ..., P_q, and execute a search procedure that attempts to use P to verify whether configuration c_2 is reachable from configuration c_1 in t steps. The search procedure may output "yes", "no", or "fail", and should have the following properties:

(i) If P computes CONFIG$_M$ at length m with one-sided error, the search procedure comes to a definite answer ("yes" or "no") with probability at least 1/2.

(ii) Whenever the search procedure comes to a definite answer, it is always correct, no matter P's behavior.

We cycle through all machines in the list, and if the search procedure ever outputs "yes" or "no", we halt and output that response. If the search procedure fails for all machines in the list, we output "fail". Given a search procedure with properties (i) and (ii), the correctness of the recovery procedure follows from the fact that we are guaranteed that one of the machines in the list of machines correctly computes CONFIG$_M$ at length m.

```
Input: y = ⟨x, c_1, c_2, t⟩ of length m; machines P_1, P_2, ..., P_q
Output: CONFIG_M(y)
(1)  if c_1 = c_2 then Output "yes" and halt                    Trivial cases
(2)            else if t = 0 then Output "no" and halt
(3)  foreach d = 1..q                          Try using P_d to compute CONFIG_M(y)
(4)     k_0 ← 1                          Number of configurations w/in distance 0 of c_1
(5)     for ℓ = 1 to t                                          Compute k_ℓ given k_{ℓ−1}
(6)        k_ℓ ← 0
(7)        foreach configuration c                          Is c w/in distance ℓ of c_1?
(8)           k'_{ℓ−1} ← 0                                  Re-experience all configurations-
(9)           foreach configuration c'                          -within distance ℓ − 1
(10)             if Verify(⟨x, c_1, c', ℓ − 1⟩, P_d) = "yes"
(11)                if M(x) transitions from c' to c in ≤ 1 time step
(12)                                                  c is within distance ℓ of c_1
(13)                   if c = c_2 then return "yes"
(14)                   else k_ℓ ← k_ℓ + 1, and Try next c (line 7)
(15)             else
(16)                k'_{ℓ−1} ← k'_{ℓ−1} + 1
(17)          if k'_{ℓ−1} ≠ k_{ℓ−1}
(18)             Failed to experience all configs w/in distance ℓ − 1
(19)             if d < q then Try next d (line 3)                          P_d fails
(20)             else return "fail"                                  All machines have failed
(21)     return "no"                                  k_t computed correctly and c_2 not found
```

Figure 4.3 Pseudo-code for the zero-sided error recovery procedure for the configuration reachability language. The list of machines is guaranteed to contain at least one computing CONFIG_M at length m with one-sided error in space $s(m)$. Configurations c_1, c_2, and c' and time values t and $\ell - 1$ are padded so that all instances of CONFIG_M of interest are of length m. The code for *Verify* used on line 10 is given in Figure 4.4.

The technical heart of the recovery procedure is a search procedure with properties (i) and (ii). Let P be a randomized machine under consideration, and $y = \langle x, c_1, c_2, t\rangle$ an input of length m we wish to compute. Briefly, the main idea is to mimic the proof that NL=coNL [Imm88, Sze88] to verify reachability and un-reachability, replacing nondeterministic guesses with simulations of an error-reduced version of P. If P computes CONFIG_M at length m with one-sided error, we can reduce P's error to a point that we have correct answers to all nondeterministic guesses with high probability, meaning property (i) is satisfied. Property (ii) follows from the fact that the algorithm can discover when incorrect nondeterministic

Verify
Input: $y = \langle x, c_0, c', t\rangle$ with $|y| = m$; machine P
Output: "yes" if by querying P it can be verified that y is in CONFIG$_M$, "fail" otherwise

```
(1)  if c_0 = c' then return "yes"                          Trivial cases
(2)  else if t = 0 then return "fail"
(3)  c ← c_0                                  Current configuration on path from c_0 to c'
(4)  for j = t − 1 down to 0                          Try to move w/in distance j of c'
(5)     foreach configuration c''
(6)        if M(x) transitions from c to c'' in ≤ 1 time step
(7)           if c'' = c' then return "yes"                    Have already reached c'
(8)           else if P(⟨x, c'', c', j⟩) outputs 1 on any of O(s) trials
(9)              c ← c'', try next j (line 4)                  Now c is one step closer
(10)    return "fail"                              Unable to move one step closer to c'
(11) return "fail"                                  After t steps, have not reached c'
```

Figure 4.4 Pseudo-code for the verification subroutine used in the zero-sided error recovery procedure of Figure 4.4. If configuration c' is within distance t of configuration c_0 and P appropriately computes CONFIG$_M$ at length m, then with high probability a path is verified and "yes" is returned. "Yes" is only returned when a path of length at most t has been verified. Configurations c_0, c', and c'', as well as time values t and j are padded so that all queries to CONFIG$_M$ of interest are of length m.

guesses have been made. For completeness, we explain how we make use of the nondeterministic algorithm of [Imm88] and [Sze88] in the current setting. The search procedure works as follows.

1. Let k_0 be the number of configurations reachable from c_1 within 0 steps, i.e., $k_0 = 1$.

2. For each value $\ell = 1, 2, ..., t$, compute the number k_ℓ of configurations reachable within ℓ steps of c_1, using only the fact that we have remembered the value $k_{\ell-1}$ that was computed in the previous iteration.

3. While computing k_t, experience all of the reachable configurations to see if c_2 is among them, for $t = 2^{O(s(m))}$ the maximum amount of time that M can take on inputs of length m.

Consider the portion of the second step where we must compute k_ℓ given that we have already computed $k_{\ell-1}$. We accomplish this in lines 6-20 of Figure 4.4 by cycling through all configurations c and for each one re-experiencing all configurations reachable from c_1 within $\ell - 1$ steps and verifying whether c can be reached in at most one step from at least one of them. To re-experience configurations reachable within distance $\ell - 1$, we try all possible configurations and query P to verify a nondeterministic path to each. The verification of a nondeterministic path is given in Figure 4.4. To check if c is reachable within one step of a given configuration, we use the transition function of M. If we fail to re-experience all $k_{\ell-1}$ configurations or if P gives information inconsistent with the transition function of M at any point we consider the search for reachability/un-reachability failed with machine P.

We now describe why this procedure satisfies properties (i) and (ii) from above. First consider (i), so we assume a randomized machine P that computes CONFIG_M at length m with one-sided error. By using a large enough number $O(s)$ of trials each time we simulate P, the error reduction for one-sided error algorithms (Section 2.2.1) along with a union bound over the total number of queries to P ensures that with probability at least $1/2$ we get correct answers each time we use line (8) of Figure 4.4. This implies that with probability at least $1/2$, $Verify$ functions as intended each time it is called (meaning $Verify(y', P)$ returns "yes" if $y' \in \text{CONFIG}_M$ and "fail" otherwise). Therefore for each configuration c and $\ell = 1, 2, ..., t$, the recovery procedure does re-experience all configurations reachable within $\ell - 1$ steps from c_1 when determining whether c is reachable within ℓ steps, and the consistency check of line (17) passes each time it is encountered while testing P. Thus with probability at least $1/2$ P comes to a definite answer, proving (i).

Now consider (ii), so we assume a definite answer either "yes" or "no" is reached while testing some machine P, and therefore the consistency check of line (17) must have passed each time it was encountered. This means that for each configuration c and $\ell = 1, 2, ..., t$, the recovery procedure did in fact re-experience all configurations reachable within at most $\ell - 1$ steps from c_1 when determining if c is reachable within ℓ steps. For $c = c_2$ and $\ell = t$, we

conclude that the recovery procedure determined correctly if c_2 is reachable from c_1 within at most t steps, proving (ii).

Consider the space usage of the recovery procedure, given in pseudo-code in Figures 4.4 and 4.4. Many of the lines of these figures consist of dealing with the configurations of M – checking whether two configurations are the same or adjacent, storing copies of the configurations, and iterating over all configurations. These tasks use $O(s(m))$ space because M is a space $O(s(m))$ machine. Line (2) of Figure 4.4 uses $O(\log q)$ space. Line (8) of Figure 4.4 uses $\max_j(s_{P_j}(m)) + O(s(m))$ space, with the first term from simulating a machine P and the second term from constructing s and keeping a counter to simulate P $O(s)$ times. Overall the space usage is $O(s(m) + \log q + \max_j(s_{P_j}(m)))$.

4.5 The Final Construction

We now complete the construction – which we began developing in Section 4.2 and is illustrated in Figure 4.1 – used to prove Theorems 1.7 and 1.8. For Theorem 1.7, we use COMP_D as the hard language L and make use of the two-sided error recovery procedure for COMP_D given in Section 4.3. For Theorem 1.8, we use CONFIG_M as the hard language L and make use of the zero-sided error recovery procedure for CONFIG_M (that works even when only guaranteed a machine P_d that behaves appropriately with one-sided error) given in Section 4.4.

We allocate an interval of input lengths $[n_i, n_i^*]$ on which to diagonalize against M_i, which is allowed $a(n) = \min(s(n), n)$ bits of advice at input length n. On an input x of length n_i, N reduces the complement of $M_i(x)$ to an instance of L of length m_i using some reduction function f (described along with L in Sections 4.3 and 4.4). The languages L are paddable so we can assume the reduction function f produces instances of L of the same length m_i for all x of length n_i. n_i^* is chosen large enough so that L at length m_i can be deterministically computed in space $s(n_i^*)$. For the hard languages we use, $n_i^* = 2^{c \cdot m_i}$ for a suitable absolute constant c suffices. N at length n_i^* performs the delayed computation: $N(0^\ell y) = L(y)$ where $|y| = m_i$ and $\ell = n_i^* - m_i$.

Diagonalizing machine N
Input: $(x, \alpha_{|x|})$, let n denote $|x|$
(1) if $\alpha_n = 0$ then return 0
(2) $i \leftarrow 0$, $n_0 \leftarrow 0$, $n_0^* \leftarrow 0$
(3) **while** $n > n_i^*$
(4) $\quad i \leftarrow i+1$, $n_i \leftarrow n_{i-1}^* + 1$,
(5) $\quad m_i \leftarrow |f(M_i/b, y)|$ for $|y| = n_i$ and $|b| = a(n_i)$, $n_i^* \leftarrow 2^{c \cdot m_i}$
(6) switch
(7) $\quad$ case $n = n_i^*$ and $x = 0^{n_i^* - m_i} y$ for some y
(8) $\quad\quad$ deterministically compute and return $L(y)$
(9) $\quad$ case $n \in [m_i, n_i^* - 1]$ and $x = 0^{n-m_i} y$ for some y
(10) $\quad\quad$ return $Rec(y, \{P_b | b \in \{0,1\}^{a(n+1)}\})$
(11) $\quad$ case $n = n_i$ and $x = 0^{n-a(n)} b$ for some b
(12) $\quad\quad$ $y = f(M_i/b, x)$
(13) $\quad\quad$ return $N(y)/\alpha$
(14) $\quad$ else
(15) $\quad\quad$ return 0

Figure 4.5 Pseudo-code for the diagonalizing machine N that witnesses Theorems 1.7 and 1.8. See Section 4.5 for a description of N in words.

For input length $n = m_i + \ell - 1$, N's one bit of advice α_n is set to indicate if there exists an advice string causing M_i to appropriately perform the delayed computation of L from input length m_i to input length $n+1$. If $\alpha_n = 1$, N/α uses the space-efficient recovery procedure for L to perform the delayed computation of L on padded inputs of length n as follows. On input $0^{n-m_i}y$, N removes the padding and executes the recovery procedure at length m_i on input $z = \langle y, \{P_b\}\rangle$, where b ranges over all possible advice strings for M_i at length $n+1$ and $P_b(y')$ acts in the following way. $P_b(y')$ simulates $M_i(0^{n+1-m_i}y')/b$ as long as the latter uses at most $s(n+1)$ space, outputting a result if one is reached and arbitrarily rejecting otherwise. Note that if M_i/b appropriately performs the delayed computation of L to length $n+1$ then the space restriction has no effect and P_b falls within the model and computes L at length m_i using space $O(s(n+1))$. The reason we break off the computation of $M_i(0^{n+1-m_i}y')/b$ when it uses more than $s(n+1)$ space is to make sure the recovery procedure runs in space $O(s(n+1))$. We will get back to this in the analysis of Section 4.6.

By the correctness of the recovery procedure, if $\alpha_n = 1$, then N/α performs the delayed computation with bounded error on padded inputs of length n. If the advice bit is 0, N/α acts trivially at input length n by rejecting immediately.

We repeat the same process on smaller and smaller padded inputs. We reach the conclusion that either (a) there is a largest input length $n \in [m_i + 1, n_i^*]$ where for no advice string b, M_i/b appropriately performs the delayed computation of L at length n; or (b) N/α correctly computes L on inputs of length m_i. If (a) is the case, N/α performs the delayed computation at length n whereas for each b either M_i/b does not behave appropriately at length n or it does but does not perform the delayed computation at length n. In either case, N/α has diagonalized against M_i/b for each possible b at length n. N's remaining advice bits for input lengths $[n_i, n-1]$ are set to 0 to indicate that nothing more needs to be done, and N/α immediately rejects inputs in this range. If (b) is the case N/α diagonalizes against M_i/b for all advice strings b at length n_i by acting as follows. On input $x_b = 0^{n_i - |b|}b$, N reduces the complement of the computation $M_i(x_b)/b$ to an instance y of L of length m_i and then simulates $N(y)/\alpha$, so $N(x_b)/\alpha = N(y)/\alpha = L(y) = \neg M_i(x_b)/b$.

We have now completed the construction used for Theorems 1.7 and 1.8. Pseudo-code for the diagonalizing machine N/α described in this section is given in Figure 4.5.

4.6 Analysis

We now explain how we come to the parameters given in the statements of Theorems 1.7 and 1.8 and Corollaries 4.1 and 4.2.

4.6.1 Theorems 1.7 and 1.8

We first consider the space usage of our constructions when the diagonalizing machine N/α is working against space $s(n)$ randomized machines. The base construction is given in Figure 4.5 and the recovery procedures are given in Figures 4.3, 4.4, and 4.4. The recovery procedure for each hard language (COMP_D in the case of Theorem 1.7 and CONFIG_M in the case of Theorem 1.8) uses space $O(s(m) + \log q + \max_j(s_{P_j}(m)))$ when trying to solve

instances of the hard language of length m. In line (10) of Figure 4.5, $P_b(y')$ simulates $M_i(0^{n+1-m_i}y')/b$ as long as the latter uses $s(n+1)$ space, and b ranges over all possible advice strings that M_i could have at length $n+1$. By choosing $a(n) \leq s(n)$ for each length n, we thus ensure that the recovery procedure in line (10) uses $O(s(m_i)+s(n+1)+s(n+1))$ space, which is $O(s(n+1))$ because s is monotone and $m_i \leq n+1$ for these n. We point out that we need the space-constructibility of s to clock the space usage of the simulations of M_i/b.

Using the facts that $s(n) = \Omega(\log n)$ and the hard languages can be decided in $O(n)$ space, n_i^* is chosen large enough so line (8) of Figure 4.5 uses at most $s(n)$ space, which is at most $s(n+1)$ by the monotonicity of s. Consider line (12). The reductions to the hard languages are very space-efficient. For COMP_D we can use a fixed deterministic machine D that takes the particular machine M_i as an extra parameter; the reduction also employs some padding involving the space bound s to ensure all instances map to the same input length m_i. As s is space-constructible, the padding can be achieved in $O(s(n_i))$ space. The reduction for CONFIG_M can similarly be realized in $O(s(n_i))$ space. For line (13) N calls itself on y. Together with the space usage of line (12) and the monotonicity of s, N's space usage at length n_i is big-O of its space usage at length m_i.

The remaining tasks of N, such as computing the interval $[n_i, n_i^*]$ that a given input length n lies within, can be achieved with $O(s(n+1))$ space. We point out that storing the value of n_i^* in line (5) may take more space. However, all that is needed here is determining whether n is larger than n_i^*, and this can be done with $O(\log n)$ space without storing n_i^*.

We have shown that N's space usage is $O(s(n+1))$ for input lengths $n \in [m_i, n_i^*]$. For input length n_i, N's space usage is big-O of its space usage at length m_i, namely $O(s(m_i+1))$. For the case of Theorem 1.7, we reduce to COMP_D, and the size m_i of the instance of COMP_D we reduce to is $n_i + O(s(n_i))$. For the case of Theorem 1.8, we reduce to CONFIG_M, and m_i is also of size $n_i + O(s(n_i))$. In both cases, the space usage of N on inputs of length n_i is $O(s(n_i + O(s(n_i))))$. By the monotonicity of s, the space usage of N on all input lengths n is $O(s(n + O(s(n))))$. We point out that we chose COMP_D and CONFIG_M as hard languages

over other natural candidates (such as the circuit value problem for Theorem 1.7 and st-connectivity for Theorem 1.8) because COMP_D and CONFIG_M reduce the blowup in input size incurred by the reductions while still allowing for space-efficient recovery procedures.

The constants in both big-O terms of $O(s(n_i + O(s(n_i))))$ – N's space usage at input length n_i – come from a variety of sources throughout the construction including reducing to the hard languages as well as simulating and clocking the space usage of M_i/b. It can be verified that for each of these the constant factor incurred only depends on s and the control characteristics of M_i. In particular, the constant factor is the same for all infinitely many appearances of machines equivalent to M_i that appear in the computable enumeration of randomized Turing machines. If $s'(n) = \omega(s(n + as(n)))$ for all constants a, N operating in space $s'(n)$ eventually encounters M_i on an interval $[n_i, n_i^*]$ where N has enough space to successfully diagonalize against M_i. If N does not yet have enough space, its advice bits are set to 0 on the entire interval. Note that this use of advice obviates the need for $s'(n)$ to be space constructible.

Now consider the amount of advice $a(n)$ that the smaller space machines can be given at length n. As discussed above, $a(n)$ is chosen to be at most $s(n)$ to ensure the recovery procedure operating at length n uses at most $s(n + 1)$ space, for $n \in [m_i, n_i^* - 1]$. Also, to complement M_i for each advice string it can receive at length n_i, we need at least one input at length n_i for each of these advice strings. Thus, the amount of advice that can be allowed is $\min(s(n), n)$.

4.6.2 Corollaries 4.1 and 4.2

We now describe modifications to the construction that yield Corollaries 4.1 and 4.2. Recall from above that when the diagonalizing machine N works against machine M_i over the interval of input lengths $[n_i, n_i^*]$, the space usage of N for $n \in [m_i, n_i^*]$ is $O(s(n + 1))$, which is already efficient enough for the corollaries.

For input length n_i, N's space usage is $O(s(m_i + 1))$ for $m_i = n_i + O(s(n_i))$ where the constants in both big-O terms depend only on s and the control characteristics of M_i. Since

we now have a monotone space bound $s(n) = O(n)$ we can assume that $m_i = a \cdot n_i$ and that N's space usage at input length n_i is at most $a' \cdot s(m_i)$ for constants a and a' depending only on s and the control characteristics of M_i.

If the space bound $s(n)$ satisfies $s(a \cdot n) = O(s(n))$ for all constants a then the construction as given in Section 4.5 already suffices to prove the corollaries. If s is a space bound where $s(a \cdot n)$ can be much larger than $s(n)$, the basic idea is to examine a number of candidate input lengths n_i' until finding one where $s(a \cdot n_i')$ is not much larger than $s(n_i')$. Specifically, if n_i is the first potential input length for working against machine M_i, we consider input lengths n_i' of the form $n_i' = a^k n_i$ for $k = 0, 1, 2, ...$, and select the first one where $s(an_i') \leq ds(n_i')$ for some fixed constant d. Such an n_i' must exist with $d = a^3$ for some $k \leq \frac{\log n_i}{\log a}$ for sufficiently large n_i; otherwise we would have that $s(n_i^2) > n_i^3 s(n_i)$, which contradicts the fact that $s(n) = O(n)$.

To prove Corollaries 4.1 and 4.2, we modify the construction as follows. When working against machine M_i, let a be a constant depending only on s and the control characteristics of M_i so that the behavior of M_i at length n reduces to an instance of the hard language of length $a \cdot n$. The diagonalizing machine N (1) allocates an interval of input lengths $[n_i, n_i^*]$ with $n_i^* = 2^{c \cdot a \cdot n_i^2}$ for the absolute constant c mentioned in Section 4.5, (2) chooses the first input length $n_i' \in [n_i, n_i^2]$ such that $s(an_i') \leq a^3 s(n_i')$, and (3) carries out the construction as described in Section 4.5 with $[n_i', n_i^*]$ the interval of input lengths. We have guaranteed that the space usage of N on input length n_i' is now $O(s(n_i'))$ where the constant in the big-O depends only on s and the control characteristics of M_i. The only extra space usage incurred is determining the appropriate $n_i' \in [n_i, n_i'^2]$, which can be done in space $O(s(n))$ for all input lengths $n \in [n_i, n_i^*]$.

4.6.3 Additional Remarks

We note that results corresponding to Theorem 1.7 and Corollary 4.1 also hold for space-bounded quantum machines: COMP_D can be used as the hard language (a space $s(n)$ quantum machine can be simulated deterministically using $2^{O(s(n))}$ time), and the space-efficient

recovery procedure for COMP_D follows through for quantum machines. A key component of the latter is error reduction – requiring taking the majority of $2^{O(s(n))}$ simulations of a space $O(s(n))$ machine while using $O(s(n))$ space – which can be done on space-bounded quantum machines.

Finally, recall that Theorem 1.8 and Corollary 4.2 give separations between zero- and one-sided error machines. These trivially imply separation results for zero-sided error machines (i.e., where N/α is a zero-sided error machine differing from space s zero-sided error machines M_i/β) with the same parameters. Conversely, we point out that in our setting a separation result for zero-sided error machines immediately implies a separation between zero- and one-sided error machines, although with a slight loss in parameters. Indeed, suppose that for appropriate choices of s' and s there is a zero-sided error machine N using space $s'(n)$ and one bit of advice that computes a language different than any zero-sided error machine using $s(n)$ space and $\min(s(n), n)$ bits of advice, but that all languages decided by zero-sided error machines using $s'(n)$ space and one bit of advice can be decided by one-sided error machines using $s(n)$ space and $a(n)$ bits of advice, for some function $a(n)$. In particular, both the language decided by N/α and its complement can be decided by one-sided error machines using $s(n)$ space and $a(n)$ bits of advice. Consider the following algorithm for computing the same language as that of N/α: (1) execute the one-sided error algorithm for deciding N/α which uses $s(n)$ space and $a(n)$ bits of advice, and output "yes" if this algorithm outputs "yes", (2) execute the one-sided error algorithm for deciding the complement of N/α which uses $s(n)$ space and $a(n)$ bits of advice, and output "no" if this algorithm outputs "yes", (3) otherwise output "fail". Given the correct advice strings for the algorithms in (1) and (2), this is a zero-sided error algorithm for deciding N/α; it uses $s(n)$ space and $2a(n)$ bits of advice. This contradicts the assumed hardness of N/α against zero-sided error machines provided $2a(n) \leq \min(s(n), n)$, and we conclude that there is a language computable by zero-sided error algorithms using $s'(n)$ space and one bit of advice that is not computable by one-sided error algorithms using $s(n)$ space and $\frac{1}{2}\min(s(n), n)$ bits of advice. Note that the notion of advice we use – a zero-sided error algorithm is only required to maintain zero-sided

error when given the correct advice string – is critical for this argument to hold. Also note that the maximum amount of advice that can be handled with this argument is a factor of two smaller than that given by Theorem 1.8.

Chapter 5

Typically-Correct Derandomization

In this chapter we introduce a new approach to typically-correct derandomization based on seed-extending pseudorandom generators. We develop the approach in Section 5.1, apply the approach to achieve conditional derandomizations in Section 5.2 and unconditional results in Section 5.3, and finally compare our approach to an earlier approach of Shaltiel in Section 5.4.

5.1 Typically-Correct Derandomization and the PRG Approach

In this section we state and prove the key lemma showing that seed-extending pseudorandom generators yield typically-correct derandomization, and introduce and analyze the seed-extending pseudorandom generator construction used for most of our results. We begin by discussing the notation and concepts used throughout this chapter.

5.1.1 Notation and Concepts

We view a randomized algorithm as defined by a deterministic machine $M(x, r)$ where x denotes the input and r the string of "coin tosses". We typically restrict our attention to one input length n, in which case M becomes a function $M : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ where m represents the number of random bits that M uses on inputs of length n. We say that $M : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ *computes* a function $L : \{0,1\}^n \to \{0,1\}$ with error ρ if for every $x \in \{0,1\}^n$, $\Pr_{R \leftarrow U_m}[M(x,R) \neq L(x)] \leq \rho$, where U_m denotes the uniform distribution over $\{0,1\}^m$ and $R \leftarrow U_m$ denotes that R is a random variable with distribution U_m. We say

that the randomized machine M *computes* a language L with error $\rho(\cdot)$, if for every input length n, the function M computes the function L with error $\rho(n)$.

Given a randomized machine M for L, our goal is to construct a deterministic machine D of complexity comparable to M that is typically correct for L. By the latter we mean that D and L agree on most inputs of any given length, or equivalently, that the relative Hamming distance between D and L at any given length is small.

DEFINITION 5.1 (TYPICALLY-CORRECT BEHAVIOR) *Let $L : \{0,1\}^n \to \{0,1\}$ be a function. We say that a function $D : \{0,1\}^n \to \{0,1\}$ is* within distance δ *of L if $\Pr_{X \leftarrow U_n}[D(X) \neq L(X)] \leq \delta$. We say that a machine D* computes *a language L* to within $\delta(\cdot)$ *if for every input length n, the function D is within distance $\delta(n)$ of the function L. For two classes of languages $\mathcal{C}_1$ and $\mathcal{C}_2$, we say that $\mathcal{C}_1$ is within $\delta(\cdot)$ of $\mathcal{C}_2$ if for every language $L_1 \in \mathcal{C}_1$ there is a language $L_2 \in \mathcal{C}_2$ that is within $\delta(\cdot)$ of L_1.*

In general, a function $G : \{0,1\}^n \to \{0,1\}^\ell$ is ϵ-pseudorandom for a test $T : \{0,1\}^\ell \to \{0,1\}$ if $|\Pr_{R \leftarrow U_\ell}[T(R) = 1] - \Pr_{S \leftarrow U_n}[T(G(S)) = 1]| \leq \epsilon$. In this chapter we are dealing with tests $T(x, r)$ that receive two inputs, namely x of length n and r of length m, and with corresponding pseudorandom functions G of the form $G(x) = (x, E(x))$, where x is of length n and $E(x)$ of length m. We call such functions "seed-extending". [1]

DEFINITION 5.2 (SEED-EXTENDING FUNCTION) *A function $G : \{0,1\}^n \to \{0,1\}^{n+m}$ is* seed-extending *if it is of the form $G(x) = (x, E(x))$ for some function $E : \{0,1\}^n \to \{0,1\}^m$. We refer to the function E as the extending part of G.*

Note that a seed-extending function G with extending part E is ϵ-pseudorandom for a test $T : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ if

$$\left| \Pr_{X \leftarrow U_n, R \leftarrow U_m}[T(X, R) = 1] - \Pr_{X \leftarrow U_n}[T(X, E(X)) = 1] \right| \leq \epsilon. \tag{5.1}$$

[1] Borrowing from the similar notion of "strong extractors" in the extractor literature, such pseudorandom generators have been termed "strong" in earlier papers. In coding-theoretic terms, they could also be called "systematic". However, we find the term "seed-extending" more informative.

A seed-extending $\epsilon(\cdot)$-pseudorandom generator for a family of tests $\mathcal{T}$ is a deterministic algorithm G such that for every input length n, G is a seed-extending $\epsilon(n)$-pseudorandom function for the tests in $\mathcal{T}$ corresponding to input length n.

5.1.2 The Seed-Extending Pseudorandom Generator Approach

Recall that a seed-extending pseudorandom generator G is a pseudorandom generator that outputs its seed, i.e., $G(x) = (x, E(x))$ for some function E. Our key observation is that good seed-extending pseudorandom generators G for certain simple tests based on the machine M yield good typically-correct derandomizations of the form $D(x) = M(x, E(x))$. The following lemma states the quantitative relationship.

LEMMA 5.3 (MAIN LEMMA) *Let $M : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ and $L : \{0,1\}^n \to \{0,1\}$ be functions such that*

$$\Pr_{X \leftarrow U_n, R \leftarrow U_m}[M(X,R) \neq L(X)] \leq \rho. \tag{5.2}$$

Let $G : \{0,1\}^n \to \{0,1\}^{n+m}$ be a seed-extending function with extending part E, and let $D(x) = M(G(x)) = M(x, E(x))$.

1. *If G is ϵ-pseudorandom for tests of the form $T(x,r) = M(x,r) \oplus L(x)$, then D is within distance $\rho + \epsilon$ of L.*
2. *If G is ϵ-pseudorandom for tests of the form $T_{r'}(x,r) = M(x,r) \oplus M(x,r')$ where $r' \in \{0,1\}^m$ is an arbitrary string, then D is within distance $3\rho + \epsilon$ of L.*

Note that if M computes L with error ρ then condition (5.2) of the lemma is met. The two parts of the lemma differ in the complexity of the tests and in the error bound. The complexity of the tests plays a critical role for the existence of pseudorandom generators. In the first item the tests use the language L as an oracle, which may result in too high a complexity. In the second item we reduce the complexity of the tests at the cost of introducing non-uniformity and increasing the error bound. The increase in the error bound is often not an issue as we can easily reduce ρ by slightly amplifying the success probability of the original machine M before applying the lemma.

Proof of Lemma 5.3. For the first item, notice that a test of the form $T(x,r) = M(x,r) \oplus L(x)$ passes iff $M(x,r) \neq L(x)$. If G is ϵ-pseudorandom for T then $|\Pr_{X \leftarrow U_n}[M(X,E(X)) \neq L(X)] - \Pr_{X \leftarrow U_n, R \leftarrow U_m}[M(X,R) \neq L(X)]| \leq \epsilon$. By assumption the latter probability is at most ρ, so $\Pr_{X \leftarrow U_n}[M(X,E(X)) \neq L(X)] \leq \rho + \epsilon$.

For the second item, pick a string r' that minimizes $\Pr_{X \leftarrow U_n}[M(X,r') \neq L(X)]$. An averaging argument shows that the latter probability is at most ρ. By the pseudorandomness of G, we have

$$|\Pr_{X \leftarrow U_n}[M(X,E(X)) \neq M(X,r')] - \Pr_{X \leftarrow U_n, R \leftarrow U_m}[M(X,R) \neq M(X,r')]| \leq \epsilon. \tag{5.3}$$

As $\Pr_{X \leftarrow U_n, R \leftarrow U_m}[M(X,R) \neq L(X)] \leq \rho$ and $\Pr_{X \leftarrow U_n}[M(X,r') \neq L(X)] \leq \rho$, the second term of (5.3) is at most 2ρ, so $\Pr_{X \leftarrow U_n}[M(X,E(X)) \neq M(X,r')] \leq 2\rho + \epsilon$. Using again that $\Pr_{X \leftarrow U_n}[M(X,r') \neq L(X)] \leq \rho$, we conclude that $\Pr_{X \leftarrow U_n}[M(X,E(X)) \neq L(X)] \leq 3\rho + \epsilon$. □

5.1.3 Hardness-Based Constructions of Seed-Extending Generators

Some of the constructions of pseudorandom generators in the literature are seed-extending or can be easily modified to become seed-extending. The generators that we consider are hardness-based, i.e., they are procedures G with access to an oracle for a language H such that the function G_H they compute is pseudorandom for a given class of tests as long as the language H is hard for a related class of algorithms.

Nisan and Wigderson [NW94] described a hardness-based pseudorandom generator construction that can be applied in a wide variety of algorithmic settings. We use a seed-extending variant of the Nisan-Wigderson construction for all of our results in Sections 5.2 and 5.3. We state the properties that we need for the algorithmic setting of circuits in the following lemma. For completeness, in Section 5.1.4 we review the Nisan-Wigderson construction and in particular verify that it can be made *seed-extending* in the way stated next and analyze the behavior of the generator for algorithmic settings other than circuits.

LEMMA 5.4 (SEED-EXTENDING NW-GENERATOR FOR CIRCUITS [NW94]) *Let n and m be positive integers and $H : \{0,1\}^{\lfloor\sqrt{n}/2\rfloor} \to \{0,1\}$ a function. There is a seed-extending function $\mathrm{NW}_{H;n,m} : \{0,1\}^n \to \{0,1\}^{n+m}$ with the following properties.*

1. *If H is $(\frac{1}{2} - \frac{\epsilon}{m})$-hard at input length $\lfloor\sqrt{n}/2\rfloor$ for circuits of size $s + m \cdot 2^{O(\log m/\log n)}$ and depth $d+1$ then $\mathrm{NW}_{H;n,m}$ is ϵ-pseudorandom for tests $T : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ computable by circuits of size s and depth d.*
2. *For each $1 \leq j \leq m$, the j^{th} bit in the extending portion of $\mathrm{NW}_{H;n,m}(x)$ is equal to $H(y_j)$ for some y_j of length $\lfloor\sqrt{n}/2\rfloor$; there is a Turing machine that outputs y_j on input (x, n, m, j) and that runs in $O(\log(m+n))$ space as long as $m(\cdot)$ is constructible in that amount of space.*

Some of our typically-correct derandomization results are unconditional because languages of the required hardness to use for H have been proven to exist. Others are conditioned on reasonable but unproven hypotheses regarding the existence of languages H that are hard on average. For the conditional results, we can assume a mildly hard function and use the XOR Lemma (Lemma 2.9) to amplify the hardness to the level required in Lemma 5.4.

We point out that the construction in Lemma 5.4 is almost optimal in the following sense. The existence of a seed-extending ϵ-pseudorandom generator for circuits of size s implies the existence of a language H that is $(\frac{1}{2} - \epsilon)$-hard at length n for circuits of size $s - O(1)$, namely for H the function that outputs the first bit in the extending portion of G.

Remark Our applications do not benefit from seed-extending pseudorandom generator constructions that recover in a blackbox fashion and are based on *worst-case* rather than average-case hardness. By definition, whenever such a pseudorandom generator $G = G_H$ based on $H : \{0,1\}^\ell \to \{0,1\}$ fails a test (5.1), there exists a small oracle circuit C, say of size s, such that $C^T = H$. This property implies that G_H has to query H in at least $(\frac{1}{2} - \epsilon)2^\ell/s$ positions, as can be argued directly and also follows from [Vio05]. The latter

condition rules out the combination of mild hardness levels (say $s = n^{O(1)}$ and $\ell = n^{\Omega(1)}$) and a polynomial running time for G, which we need for our applications.

5.1.4 Analysis of the Nisan-Wigderson Construction

Our typically-correct derandomization results use the Nisan-Wigderson generator construction [NW94]. Lemma 5.4 states that given a sufficiently hard function, the construction gives a *seed-extending* pseudorandom generator. In this section we review this well-known construction to verify that the original analysis carries through when the generator outputs its seed. A reader familiar with the Nisan-Wigderson construction may wish to skip this section and refer back to it as needed.

Definition of NW-Generator When taking a seed of length n and outputting m bits, the generator makes use of the following combinatorial object.

Definition 5.5 (combinatorial design) *A (k, ℓ) design of size m over $[n]$ is a sequence $S_1, S_2, \ldots, S_m$ of subsets of $[n]$ such that (a) $|S_i \cap S_j| \leq k$ for $1 \leq i < j \leq m$, and (b) $|S_i| = \ell$ for $1 \leq i \leq m$.*

The following construction suffices for our results. It has been (re)derived and used in several contexts, including in [NW94]. We provide a proof for completeness.

Lemma 5.6 *For any positive integers n, k, ℓ, m, and n such that $\ell \leq \sqrt{n}/2$ and $k \geq \frac{\log m}{\log \ell}$ there is a (k, ℓ) design of size m over $[n]$. Further, there is a Turing machine that on input (k, ℓ, m, n, i) outputs the i^{th} set and uses $O(\log(m + n))$ space.*

Proof. For q a positive integer, let $\mathrm{GF}(2^q)$ denote the finite field of size 2^q. The main idea is to view the elements of $[n]$ as points in $\mathrm{GF}(2^q) \times \mathrm{GF}(2^q)$, let the sets S_i correspond to the graphs of polynomials of degree at most k over $\mathrm{GF}(2^q)$, and use the fact that two distinct such polynomials can intersect in at most k points.

Now we provide the details. Let q be the integer such that $\sqrt{n}/2 < 2^q \leq \sqrt{n}$. We identify the elements of $\mathrm{GF}(2^q)$ with the bit strings of length q. Since under the given conditions

$m \leq 2^{(k+1)q}$, we can view $i \in [m]$ as defining a sequence of $k+1$ strings of length q (by padding with 0's as needed), and thus as a sequence of $k+1$ elements over GF(2^q). We interpret this sequence as the successive coefficients of a polynomial p_i of degree at most k over GF(2^q). We take the first ℓ points y_1, ..., y_ℓ in GF(2^q), say in lexicographic order, and define S_i as

$$S_i = \{(y_1, p_i(y_1)), ..., (y_\ell, p_i(y_\ell))\}.$$

Note that GF(2^q) contains at least ℓ elements as $\ell \leq \sqrt{n}/2 < 2^q$, and that $|S_i| = \ell$. The intersection size $|S_i \cap S_j|$ equals the number of y's on which p_i and p_j agree. For distinct i and j, that number is upper bounded by the maximum degree k.

Finally, consider the complexity of generating the set S_i. We must (a) perform arithmetic of $O(\log(m+n))$ bit numbers to determine q, keep counters, etc., (b) determine an irreducible polynomial of degree q over GF(2), and (c) using the irreducible polynomial perform arithmetic over GF(2^q). (b) can be performed in $O(q) = O(\log n)$ space by exhaustive search, and both (a) and (b) can be performed in $O(\log(m+n))$ space as well. □

Given such a design, we define the NW-generator as follows based on a presumed hard Boolean function H. Our definition differs from the original one [NW94] only in that the generator additionally outputs its seed.

Definition 5.7 (seed-extending NW generator [NW94]) *Let n and m be integers, and $S_1, S_2, \ldots, S_m$ the (k, ℓ)-design of size m over $[n]$ with $\ell = \lfloor \sqrt{n}/2 \rfloor$ and $k = \lceil \frac{\log m}{\log \ell} \rceil$ provided by Lemma 5.6. Given a function $H : \{0,1\}^\ell \to \{0,1\}$ the Nisan-Wigderson generator $\mathrm{NW}_{H;n,m} : \{0,1\}^n \to \{0,1\}^{n+m}$ is defined as*

$$\mathrm{NW}_{H;n,m}(x) = (x, H(x|_{S_1}), ..., H(x|_{S_m})),$$

where $x|_{S_i}$ denotes the substring of x of length ℓ formed by taking the bits of x indexed by S_i.

The NW-construction has the property that if the function H is hard on average for a certain class of algorithms, then $\mathrm{NW}_{H;n,m}$ is pseudorandom for related tests. Lemma 5.4 formalizes this property in the case of circuits. We include a proof sketch for reasons of

completeness, where we focus on verifying that the argument given in [NW94] goes through with our modification of the generator. The proof sketch also gives us an opportunity to point out how the argument translates to other types of algorithms we consider; we provide these observations following the proof sketch.

Proof sketch of Lemma 5.4. The argument goes by contradiction: we assume a test T computable by a circuit of size s and depth d that ϵ-distinguishes the output of $\mathrm{NW}_{H;n,m}$ from uniform in the sense that

$$|\Pr_{X\leftarrow U_n, R\leftarrow U_m}[T(X,R)=1] - \Pr_{X\leftarrow U_n}[T(\mathrm{NW}_{H;n,m}(X))=1]| \geq \epsilon.$$

We use T to construct a circuit that is not much larger and that computes H well on average, contradicting the assumed hardness of H. There are two parts to the argument, namely the construction of a predictor $\tilde{T}$, and the construction of a circuit that uses $\tilde{T}$ to compute H well on average.

Construction of a predictor A circuit $\tilde{T}$ is an ϵ'-predictor for $\mathrm{NW}_{H;n,m}$ if there is an index j such that when given the first $j-1$ bits of a sample from $\mathrm{NW}_{H;n,m}$, $\tilde{T}$ predicts the jth bit with success at least $\frac{1}{2}+\epsilon'$. The transformation from an ϵ-distinguisher to an ϵ'-predictor with $\epsilon' = \frac{\epsilon}{m}$ is a standard step in hardness-based pseudorandom generators. The key observation for our purposes is that the first n bits of $\mathrm{NW}_{H;n,m}$ are uniform at random and so cannot be predicted with any advantage. Thus the bit j has to fall within the extending part of $\mathrm{NW}_{H;n,m}$, which means that the original analysis carries through without any change in the parameters. Let us go through the analysis in some detail.

We consider the behavior of T on hybrid distributions D_i that output their first $n+i$ bits according to $\mathrm{NW}_{H;n,m}$ and output their remaining $m-i$ bits uniformly, for $i=0,...,m$. Notice that $D_0 \equiv U_{n+m}$ and $D_m \equiv \mathrm{NW}_{H;n,m}$ so that we have by assumption $|\Pr_{Z\leftarrow D_0}[T(Z)=$

$1] - \Pr_{Z \leftarrow D_m}[T(Z) = 1]| \geq \epsilon$. Using this fact we have that

$$\begin{aligned} \epsilon &\leq |\Pr_{Z \leftarrow D_0}[T(Z) = 1] - \Pr_{Z \leftarrow D_m}[T(Z) = 1]| \\ &= |\sum_{i=1}^{m} \Pr_{Z \leftarrow D_i}[T(Z) = 1] - \Pr_{Z \leftarrow D_{i-1}}[T(Z) = 1]| \\ &\leq \sum_{i=1}^{m} |\Pr_{Z \leftarrow D_i}[T(Z) = 1] - \Pr_{Z \leftarrow D_{i-1}}[T(Z) = 1]|, \end{aligned}$$

so there must exist an index i for which $|\Pr_{Z \leftarrow D_i}[T(Z) = 1] - \Pr_{Z \leftarrow D_{i-1}}[T(Z) = 1]| \geq \frac{\epsilon}{m}$. From this point, an averaging argument shows that there is a way to fix the last $m - i + 1$ bits so that either T or $\neg T$ with these bits fixed indeed predicts the $(n+i)^{th}$ bit of $\mathrm{NW}_{H;n,m}$ with success $\frac{1}{2} + \frac{\epsilon}{m}$ when given the first $n + i - 1$ bits. We let $\tilde{T}$ be this circuit, so we have that

$$\Pr_{X \leftarrow U_n}[\tilde{T}(X, H(X|_{S_1}), ..., H(X|_{S_{i-1}})) = H(X|_{S_i})] \geq \frac{1}{2} + \frac{\epsilon}{m}.$$

Using $\tilde{T}$ to compute H In this part of the argument, we use $\tilde{T}$ to construct a circuit not much larger than the circuit for T that computes H well on average. An averaging argument shows that there is a way to fix the bits in X that are outside of S_i to preserve the prediction probability of $\tilde{T}$. Let $\tilde{Y}$ denote a string of length n that has these positions of X fixed to these values and with $X|_{S_i} = Y$. Then we have that

$$\Pr_{Y \leftarrow U_\ell}[\tilde{T}(\tilde{Y}, H(\tilde{Y}|_{S_1}), ..., H(\tilde{Y}|_{S_{i-1}})) = H(Y)] \geq \frac{1}{2} + \frac{\epsilon}{m}. \tag{5.4}$$

Consider $H(\tilde{Y}|_{S_j})$ for some $1 \leq j \leq i - 1$. Notice that $\tilde{Y}$ has all bits fixed except those indexed by S_i, so for each $1 \leq j \leq i - 1$, the function $H(\tilde{Y}|_{S_j})$ is a function that depends on only $|S_j \cap S_i|$ many bits – which by construction is most $k = O(\log m / \log n)$. We plug in either a DNF or CNF into $\tilde{T}$ for each of these functions, and we are left with a circuit that computes H on inputs of length $\ell = \lfloor \sqrt{n}/2 \rfloor$ with success at least $\frac{1}{2} + \frac{\epsilon}{m}$.

Parameters Consider the size and depth of the circuit that we have created. $\tilde{T}$ has the same size and depth as T, and to this we have added at most m circuits for the functions

$H(\tilde{Y}|_{S_j})$, each of which is a CNF or DNF of size $2^{O(k)} = 2^{O(\log m/\log n)}$. Choosing either a CNF or DNF for each to ensure the depth increases only by one, this yields the parameters stated in Item 1 of Lemma 5.4. The efficiency of constructing the generator, Item 2, follows by the efficiency of the designs of Lemma 5.6. □

Remark The argument in the proof of Lemma 5.4 can be adapted for (non-uniform) models of computation other than circuits. We point out the modifications and observations about the above proof we need for the models we consider.

- Relativized circuits.

  The above argument carries through when both the circuits underlying the hardness hypothesis and the circuits underlying the tests can have gates that compute some fixed oracle O. Such oracle gates contribute their number of inputs to the size of the circuit. In particular, if H has the stated hardness for circuits that have oracle gates for an oracle O, then $\mathrm{NW}_{H;n,m}$ is ϵ-pseudorandom for tests T with the stated parameters that have access to O oracle gates.

- Circuits with a limited number of special gates.

  If the tests T of Item 1 of Lemma 5.4 are allowed a certain number of special gates (e.g., gates for arbitrary symmetric functions), then $\mathrm{NW}_{H;n,m}$ is ϵ-pseudorandom for T provided H has the stated hardness for circuits that have access to the same exact number and type of special gates as the tests T. This follows from the argument above because the circuit that approximates H consists of a single copy of the test circuit T or its negation, with some of its input bits fixed and others computed by small regular circuits without special gates.

- Branching programs.

  The correctness argument carries over as such for branching programs instead of circuits. The size parameter in Item 1 becomes slightly different. Each of the functions $H(\tilde{Y}|_{S_j})$ can be computed by a branching program of size $2^{O(k)}$. Incorporating those

into the branching program for $\tilde{T}$ means replacing some edges of the branching program for $\tilde{T}$ with a branching program of size $2^{O(k)}$, resulting in an overall blowup in size of $2^{O(k)}$ for $k = O(\log m / \log n)$. Thus, if H is $(\frac{1}{2} - \frac{\epsilon}{m})$-hard at input length $\lfloor \sqrt{n}/2 \rfloor$ for branching programs of size $s \cdot 2^{O(\log m / \log n)}$ then $\mathrm{NW}_{H;n,m}$ is ϵ-pseudorandom for tests $T : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ computable by branching programs of size s.

- Communication protocols.

  In the proof of Theorem 5.11 in Section 5.3.3 we use a hardness-based pseudorandom generator $G_{H;n,\ell,m}$ that can be seen as a degenerate form of the Nisan-Wigderson construction with the sets S_i pairwise disjoint. The above proof carries through for this generator as well. Namely, let T be a randomized communication protocol taking k-tuples of n bit inputs and using m bits of randomness and q bits of communication that ϵ-distinguishes the output of the generator. Then the approximation to H given in (5.4) is within $\frac{1}{2} + \frac{\epsilon}{m}$ of H on k-tuples of ℓ-bit strings. The approximation can be computed by running the protocol T or its negation with certain input bits fixed and others set to the outcome of $H(\tilde{Y}|_{S_j})$ for some $j < i$. As the S_j are chosen disjointly for the generator $G_{H;n,\ell,m}$, $H(\tilde{Y}|_{S_j})$ is a function with *all* input bits fixed and therefore does not require any additional communication between the players. Altogether, the approximation given in (5.4) can be computed by a non-uniform protocol that uses q bits of communication.

  We conclude that if H is $(\frac{1}{2} - \frac{\epsilon}{m})$-hard for non-uniform protocols operating on k-tuples of ℓ-bit inputs that use q bits of communication then $G_{H;n,\ell,m}$ is ϵ-pseudorandom for non-uniform randomized communication protocols that operate on k-tuples of n-bit inputs, use m random bits, and q bits of communication.

5.2 Conditional Results

In this section we obtain a number of typically-correct derandomization results that are conditioned on unproven but reasonable hardness hypotheses. These results are summarized in Figure 5.1.

5.2.1 Bounded-Error Polynomial Time

The first setting we consider is that of BPP. We use a modest hardness assumption to show that any language in BPP has a polynomial-time deterministic algorithm that errs on a polynomially small fraction of the inputs. The result is restated here for convenience.

THEOREM 1.1 (TYPICALLY-CORRECT DERANDOMIZATION OF BPP) *Let L be a language that is computed by a randomized bounded-error polynomial-time machine M. For any positive constant c, there is a positive constant d (depending on c and the running time of M) such that the following holds. If there is a language H in P that is $\frac{1}{n^c}$-hard for circuits of size n^d, then there is a deterministic polynomial-time machine D that computes L to within $\frac{1}{n^c}$.*

Before proving Theorem 1.1, let us compare it to previous conditional derandomization results for BPP. We first consider everywhere-correct results. Plugging our assumption into the hardness versus randomness tradeoffs of [NW94] gives the incomparable result that BPP is in deterministic subexponential time, i.e., in time 2^{n^ϵ} for every positive constant ϵ. We remark that to obtain this result one can relax the assumption and allow the language H to be in deterministic linear-exponential time, i.e., E=DTIME($2^{O(n)}$).

We next compare Theorem 1.1 to previous conditional results on typically-correct derandomization of BPP [GW02, Sha09]. The assumption that we use is weaker than the assumptions that are used by previous work. More specifically, [GW02] needs H to be $\frac{1}{n^c}$-hard for circuits of size n^d with a SAT oracle, and [Sha09] requires that H be $(\frac{1}{2} - \frac{1}{2^{n^{\Omega(1)}}})$-hard for circuits of size n^d.

Theorem	Setting	Hardness Assumption	Conclusion
Thm 1.1	BPP=BP.P	P $\frac{1}{n^c}$-hard for SIZE(n^d)	BPP within $\frac{1}{n^c}$ of P
Thm 5.8	BP.$\oplus$P	$\oplus$P $\frac{1}{n^c}$-hard for SIZE$^{\oplus\mathrm{SAT}}(n^d)$	BP.$\oplus$P within $\frac{1}{n^c}$ of $\oplus$P
Thm 5.9	AM=BP.NP	NP $\cap$ coNP $\frac{1}{n^c}$-hard for SIZE$^{\mathrm{SAT}}(n^d)$	AM within $\frac{1}{n^c}$ of NP
Thm 5.10	BP.L	L $\frac{1}{n^c}$-hard for BP-SIZE(n^d)	BP.L within $\frac{1}{n^c}$ of L

Figure 5.1 Our conditional typically-correct derandomization results.

Thus, the approaches of [GW02] and [Sha09] do not yield any typically-correct derandomization when starting from the modest assumption that we use. Under their respective stronger assumptions, the other approaches do yield typically-correct algorithms that are closer to L. We remark that we can match the distance in [Sha09] if we are allowed to assume the same hardness hypothesis.

Proof of Theorem 1.1. Let M be a polynomial-time randomized bounded-error machine computing a language L, and let $c > 0$ be a constant. We obtain the typically-correct deterministic machine D by using Item 2 of Lemma 5.3 with the Nisan-Wigderson construction as the generator. More specifically, we set

$$D(x) = M'(\mathrm{NW}_{H';n,n^b}(x))$$

where M' is an error-reduced version of M that uses n^b random bits for a constant b depending on the running time of M and where H' is the result of applying a certain amount of hardness amplification to H. We now analyze how to set the parameters of the various ingredients and establish the stated properties.

1. Error Reduction.

   To keep the error term 3ρ from invoking Item 2 of Lemma 5.3 less than $\frac{1}{2n^c}$, we let M' take the majority vote of $O(\log n)$ independent trials of M so that M' has error at most $\frac{1}{6n^c}$.

2. Nisan-Wigderson construction.

   Setting $\rho = \frac{1}{6n^c}$ in Item 2 of Lemma 5.3, D computes L to within distance $\frac{1}{n^c}$ if

$\mathrm{NW}_{H';n,n^b}$ is $\frac{1}{2n^c}$-pseudorandom against tests $T_{r'}$ of the form $T_{r'}(x,r) = M'(x,r) \oplus M'(x,r')$ for r' an arbitrary string. Using the standard reduction from Turing machines with advice to circuits, the tests $T_{r'}$ are circuits of size $O(n^{2b})$ for some constant b depending on the running time of M. By Lemma 5.4, $\mathrm{NW}_{H';n,n^b}$ is $\frac{1}{2n^c}$-pseudorandom against the tests $T_{r'}$ if H' is $(\frac{1}{2} - \frac{1}{2n^{c+b}})$-hard for circuits of size $O(n^{2b})$ on inputs of length $\lfloor\sqrt{n}/2\rfloor$. Thus a sufficient hardness condition for H' is to be $(\frac{1}{2} - \frac{1}{n^a})$-hard for circuits of size n^a on inputs of length n, for $a = 2\max(c+b, 2b) + 1$.

3. XOR Lemma.

   Let $H : \{0,1\}^n \to \{0,1\}$ be $\frac{1}{n^c}$-hard for circuits of size n^d and define $H' : \{0,1\}^{k \cdot n} \to \{0,1\}$ by $H'(x_1, ..., x_k) = H(x_1) \oplus H(x_1) \oplus ... \oplus H(x_k)$. By the XOR Lemma (Lemma 2.9), H' is $(\frac{1}{2} - \frac{1}{n^a})$-hard for circuits of size n^a if we can choose k and γ such that (i) $(1 - \frac{1}{n^c})^k + \gamma \leq \frac{1}{(nk)^a}$ and (ii) $n^d \cdot (\frac{\gamma^2}{\log(n^c/\gamma)}) \geq (nk)^a$. To satisfy (i), we choose $\gamma = \frac{1}{2(nk)^a}$ and set $k = n^{c+1}$ to ensure that for sufficiently large n, $(1 - \frac{1}{n^c})^k \leq e^{-k/n^c} = e^{-n} \leq \frac{1}{2(nk)^a}$. With these choices, (ii) simplifies to $n^d \geq 8n^{3(c+2)a} \log(2n^{c+(c+2)a})$ which can be satisfied by choosing $d = 3(c+2)a + 1$.

This establishes the correctness of D, i.e., D computes L to within $\frac{1}{n^c}$ provided H is $\frac{1}{n^c}$-hard for circuits of size n^d. Now consider the complexity of D. By Item 2 of Lemma 5.4, $\mathrm{NW}_{H';n,n^b}$ is computable in time polynomial in n provided H' is, which in turn is computable in time polynomial in n provided H is. □

5.2.2 Extensions to Other Algorithmic Settings

[KM02] observed that the Nisan-Wigderson generator can be used to give hardness versus randomness tradeoff results in a number of different algorithmic settings. This approach also works within our typically-correct derandomization framework. In this section we discuss the last three applications listed in Figure 5.1.

5.2.2.1 BP.⊕P Algorithms

Our conditional results for BP.⊕P algorithms and Arthur-Merlin protocols rely on the fact that all the ingredients in the proof of Theorem 1.1 relativize: error reduction using majority voting, the Nisan-Wigderson construction relativizes (see the remark after the proof of Lemma 5.4), the XOR Lemma, and our main lemma. Thus, we have the following as a corollary to the proof of Theorem 1.1.

THEOREM 5.8 (RELATIVIZED VERSION OF THEOREM 1.1) *Let O be any language, and let L be a language that is computed by a randomized bounded-error polynomial-time machine M that has oracle access to O. For any positive constant c, there is a positive constant d (depending on c and the running time of M) such that the following holds. If H is a language that is $\frac{1}{n^c}$-hard for circuits of size n^d that have access to O oracle gates, then there is a polynomial-time machine D that uses oracle access to both H and O that computes L to within $\frac{1}{n^c}$.*

Theorem 5.8 immediately yields a typically-correct derandomization result for the class BP.⊕P, a class that is of interest as a key step in the result that any language within the polynomial hierarchy can be solved with an oracle for counting [Tod91]. Recall that a language L in BP.⊕P is defined by a deterministic procedure M that on input (x, R, z) with $|x| = n$ runs in time n^k for some constant k and such that

(i) for every $x \in L$, $\Pr_{R \leftarrow U_{n^k}}[|\{z \in \{0,1\}^{n^k} \text{ s.t. } M(x,R,z) = 1\}| \equiv 1 \pmod 2] \geq \frac{2}{3}$, and

(ii) for every $x \notin L$, $\Pr_{R \leftarrow U_{n^k}}[|\{z \in \{0,1\}^{n^k} \text{ s.t. } M(x,R,z) = 1\}| \equiv 1 \pmod 2] \leq \frac{1}{3}$.

⊕SAT is a natural ⊕P-complete language consisting of Boolean formulae that have an odd number of satisfying assignments. Applying Theorem 5.8 with the oracle O set to ⊕SAT, requiring the hard function H to lie within ⊕P, and using the facts that BP.⊕P= $\text{BPP}^{\oplus\text{SAT}}$ and $\text{P}^{\oplus\text{SAT}} = \oplus\text{P}$, we obtain the typically-correct derandomization result for BP.⊕P algorithms listed in Figure 5.1.

5.2.2.2 Arthur-Merlin Protocols

The complexity class AM consists of *Arthur-Merlin* protocols that have a randomized polynomial-time verifier, as defined in Section 2.4. As discussed there, AM can be viewed as BP.NP, and if we remove the randomness we would be left with an NP predicate. Thus, derandomizing AM means obtaining simulations of AM on nondeterministic machines. Using the fact that $\mathrm{AM} = \mathrm{BP.NP} \subseteq \mathrm{BPP}^{\mathrm{NP}}$, an immediate application of Theorem 5.8 with the oracle O set to SAT yields a conditional typically-correct derandomization of AM into $\mathrm{P}^{\mathrm{SAT}}$ under the assumption of a language $H \in \mathrm{NP}$ that is mildly hard on average for polynomial-size circuits that have access to SAT oracle gates. By looking more closely at the proof of Theorem 5.8 and strengthening the assumption on the complexity of the hard function H, namely to $\mathrm{NP} \cap \mathrm{coNP}$, we obtain conditional typically-correct derandomization of AM into NP.

THEOREM 5.9 (TYPICALLY-CORRECT DERANDOMIZATION OF AM) *Let L be a language computable by a polynomial-time Arthur-Merlin protocol. For every constant $c > 0$ there is a constant d such that if* $\mathrm{NP} \cap \mathrm{coNP}$ *contains a language H that is $\frac{1}{n^c}$-hard for circuits of size n^d that have access to* SAT *oracle gates, then there is a nondeterministic polynomial-time machine D that computes L to within $\frac{1}{n^c}$.*

Proof. We follow the proofs of Theorems 1.1 and 5.8. We define D as the language of all inputs x for which $\exists z \in \{0,1\}^{n^b} V'(\mathrm{NW}_{H';n,n^b}(x))$, where V' is an error-reduced version of the verification predicate V that uses n^b random bits for a constant b depending on the running time of V and where H' is the result of applying some amount of hardness amplification to the assumed hard function H. We need to verify both the *correctness* and the *complexity* of D. Correctness follows by Theorem 5.8 and the fact that $\mathrm{AM} \subseteq \mathrm{BPP}^{\mathrm{SAT}}$, as discussed above.

As for the complexity of D, we first point out that error-reduction can be performed within AM using parallel repetition, so that an AM protocol with verification procedure V'

and reduced error can be given. Now,

$$D(x) = 1 \Leftrightarrow \exists z \in \{0,1\}^{n^b} V'(x, H'(y_1), ..., H'(y_{n^b}), z), \tag{5.5}$$

where each y_1, ..., y_{n^b} is some efficiently computable substring of x of length $\lfloor \sqrt{n}/2 \rfloor$. By Item 2 of Lemma 5.4 and the fact that $H \in \mathrm{NP} \cap \mathrm{coNP}$, $V'(x, H'(y_1), ..., H'(y_{n^b}), z)$ defines a predicate on (x, z) that is decidable in $\mathrm{P}^{\mathrm{NP} \cap \mathrm{coNP}} = \mathrm{NP} \cap \mathrm{coNP}$, which turns the right-hand side of (5.5) into an NP-predicate on x. Thus, D is in NP. □

Remark In the context of everywhere-correct derandomization it is known that hardness for nondeterministic circuits (rather than circuits with access to a satisfiability oracle) is sufficient to derandomize Arthur-Merlin protocols [MV05, SU05]. In fact, [SU06] shows that the assumption that EXP contains a language that cannot be computed by small nondeterministic circuits implies that EXP contains a language that cannot be computed by small circuits that make non-adaptive calls to a satisfiability oracle. In the context of typically-correct derandomization we need hard languages that can be computed in P^{NP} or $\mathrm{NP} \cap \mathrm{coNP}$ and we do not know whether we can replace hardness for circuits with oracle access to satisfiability by hardness for nondeterministic circuits.

5.2.2.3 Space-Bounded Setting

We obtain the final result listed in Figure 5.1 by observing that the proof of Theorem 1.1 follows through in the setting of derandomizing BP.L algorithms – randomized algorithms that run in logarithmic space and are allowed two-way access to their random bits [Nis93].[2]

THEOREM 5.10 (TYPICALLY-CORRECT DERANDOMIZATION OF BP.L) *Let L be a language that is computed by a randomized bounded-error log-space machine M that has two-way access to its random bits. For any positive constant c, there is a positive constant d (depending on c and the space usage of M) such that the following holds. If there is a language H*

[2] Recall that BP.L algorithms are potentially much more powerful than randomized space-bounded algorithms that are given *one-way* access to their randomness – referred to as BPL algorithms. While it is known that BPL is contained in $\mathrm{DSPACE}(\log^{1.5} n)$ [SZ99], all that is known for BP.L is that BP.L $\subseteq$ BPP $\subseteq$ PSPACE.

computable in logarithmic space that is $\frac{1}{n^c}$-hard for branching programs of size n^d, then there is a deterministic log-space machine D that computes L to within $\frac{1}{n^c}$.

Proof. We follow the same outline as the proof of Theorem 1.1. That is, we define D by $D(x) = M'(\mathrm{NW}_{H';n,n^b}(x))$ where M' is an error-reduced version of M that uses n^b random bits for a constant b depending on the running time of M and where H' is the result of applying the XOR lemma to H. We need to verify the *correctness* and the *complexity* of D.

Correctness follows as in the proof of Theorem 1.1 with two modifications. First, we make use of the remark after the proof of Lemma 5.4 to apply the Nisan-Wigderson construction to branching programs. Second, we use a version of the XOR lemma for branching programs, which reads the same as Lemma 2.9 except that we replace "circuits" by "branching programs", and set $\delta' = \frac{1}{2} - (1-\delta)^k - \gamma$ and $s' = \Omega(\frac{\gamma^4}{\log^2(1/(\delta\gamma))}) \cdot s$.

As for the complexity of D, we first observe that $\mathrm{NW}_{H';n,n^b}$ is computable in logarithmic space by Item 2 of Lemma 5.4 and the assumption that H is computable in logarithmic space. As M' is also computable in logarithmic space and D is the composition of M' and $\mathrm{NW}_{H';n,n^b}$, D is computable in logarithmic space. □

5.3 Unconditional Results

In this section we obtain *unconditional* typically-correct derandomization results in a number of algorithmic settings.

5.3.1 Constant-Depth Circuits

Nisan [Nis91] used the NW-construction together with the fact that the parity function is $(\frac{1}{2} - \frac{1}{2^{n^{\Omega(1)}}})$-hard for constant-depth circuits [Hås87] to obtain everywhere-correct derandomization of uniform randomized constant-depth circuits ($\mathrm{BP.AC}^0$) by uniform quasipolynomial-size constant-depth circuits. The transformation works for various notions of uniformity, including log-space and polynomial-time uniformity.

[Sha09] obtained a more efficient derandomization of uniform $\mathrm{BP.AC}^0$ in the typically-correct setting, replacing "quasipolynomial-size" by "polynomial-size". The approach of

[Sha09] relies on certain extractors that have exponentially small error. We elaborate on the extractor-based approach of [Sha09] in Section 5.4 and point out that it can only handle randomized algorithms that use a sublinear number of random bits. In order to handle algorithms that use a polynomial number of random bits, [Sha09] first uses Nisan's generator to reduce the randomness of a uniform BP.AC^0 circuit to sublinear and then uses the exponentially strong lower bounds for constant-depth circuits computing parity once more to construct the extractor that is needed.

By using a single application of Nisan's generator along with Lemma 5.3, our approach gives a simpler proof of the typically-correct derandomization results for uniform BP.AC^0 of [Sha09]. As before, the result holds for either log-space or polynomial-time uniformity and shows that for any constant c, uniform BP.AC^0 is within distance $\frac{1}{n^c}$ of uniform AC^0, the class of uniform polynomial-size constant-depth circuits. The error can be further reduced by allowing the deterministic algorithm parity gates: uniform BP.AC^0 is within distance $\frac{1}{2^{n^{\Omega(1)}}}$ of uniform $\mathrm{AC}^0[\oplus]$.

5.3.2 Constant-Depth Circuits with Few Symmetric Gates

In contrast to the approach of [Sha09], our techniques also yield results in settings where the best-known lower bounds only yield moderate hardness on average. One such model is that of constant-depth circuits that are allowed a small number of arbitrary symmetric gates, i.e., gates that compute functions which only depend on the Hamming weight of the input, such as parity and majority. In this setting Viola [Vio06] constructed a simple function computable by uniform constant-depth circuits that have access to parity gates that is $(\frac{1}{2} - \frac{1}{s})$-hard for circuits of size s that use $\log s$ symmetric gates, for a function $s = n^{\Omega(\log n)}$. As the approach of [Sha09] requires a hard function with exponentially strong hardness to build a seedless extractor with exponentially small error, that approach cannot make use of this hardness result to achieve derandomization of randomized circuits with few symmetric gates. Our approach *can* exploit these weaker hardness results and gives the following for both log-space and polynomial-time uniformity.

THEOREM 1.2 (TYPICALLY-CORRECT DERANDOMIZATION OF AC^0 WITH FEW SYMMETRIC GATES) *Let L be a language and M a uniform randomized circuit of constant depth and polynomial size that uses $o(\log^2 n)$ symmetric gates such that M computes L with error at most ρ. Then there is a uniform deterministic circuit D of constant depth and polynomial size that uses exactly the same symmetric gates as M in addition to a polynomial number of parity gates such that D computes L to within $3\rho + \frac{1}{n^{\Omega(\log n)}}$.*

We point out that the error term 3ρ can be removed using standard error reduction provided M uses even fewer symmetric gates. For example, suppose M computes a language L using $o(\log n)$ symmetric gates and let M' be the randomized algorithm that takes the majority vote of $O(\log n)$ independent trials of M to reduce ρ to $\frac{1}{4n^c}$ for some constant c. Then M' uses $o(\log^2 n)$ symmetric gates and by Theorem 1.2 there is a uniform deterministic polynomial-size constant-depth circuit that uses $o(\log^2 n)$ symmetric gates in addition to a polynomial number of parity gates and computes L to within $\frac{1}{n^c}$.

Proof of Theorem 1.2. Let M be a uniform circuit of depth d and size n^b that uses $o(\log^2 n)$ symmetric gates and computes a language L with error at most ρ on every input, for some constants d and b. We obtain the typically-correct deterministic algorithm D by using Item 2 of Lemma 5.3 with the Nisan-Wigderson construction as the generator, i.e., we set

$$D(x) = M(\mathrm{NW}_{H;n,n^b}(x))$$

for some H. We first explain how to set the parameters and choose the hard language H so as to verify the *correctness* of D – that D computes L to within distance $3\rho + \frac{1}{n^{\Omega(\log n)}}$.

1. Nisan-Wigderson construction.

   By Item 2 of Lemma 5.3 D computes L to within distance $3\rho + \epsilon$ if $\mathrm{NW}_{H;n,n^b}$ is ϵ-pseudorandom against tests $T_{r'}$ of the form $T_{r'}(x, r) = M(x, r) \oplus M(x, r')$, which are circuits of size $O(n^b)$ and depth $d + 1$ that use $o(\log^2 n)$ symmetric gates. In a remark following the proof of Lemma 5.4, we point out that the NW generator is secure with the same parameters given in Item 1 of Lemma 5.4 for circuits T that have access to

a certain number of symmetric gates if the hard function H is hard with the same parameters stated in the lemma with respect to circuits that have access to the exact same symmetric gates. In particular, $\mathrm{NW}_{H;n,n^b}$ is ϵ-pseudorandom against the tests $T_{r'}$ if H is $(\frac{1}{2} - \frac{\epsilon}{n^b})$-hard on inputs of length $\lfloor\sqrt{n}/2\rfloor$ for circuits of size $O(n^b)$ and depth $d+2$ that use $o(\log^2 n)$ symmetric gates.

2. Hard language H.

   [Vio06] exhibits a function H that is computable by log-space uniform linear-size constant-depth circuits that have access to parity gates such that H is $(\frac{1}{2} - \frac{1}{s})$-hard on inputs of length n for circuits of size s and depth $d+2$ that use at most $\log s$ symmetric gates, for $s = n^{\alpha \log n}$ where α is a constant depending on d. Then H has the required hardness provided $\frac{\epsilon}{n^b} \geq \frac{1}{\lfloor\sqrt{n}/2\rfloor^{\alpha \log(\lfloor\sqrt{n}/2\rfloor)}}$. We can choose ϵ of the form $\frac{1}{n^{\Omega(\log n)}}$ to satisfy this inequality.

This guarantees the correctness of D. Now consider the complexity of D. By Item 2 of Lemma 5.4 and the complexity of computing H stated above, $\mathrm{NW}_{H;n,n^b}$ is computable by a log-space uniform constant-depth polynomial-size circuit that has access to parity gates. Thus D is computable by a circuit as described in the statement of Theorem 1.2 and maintains the uniformity of M (either log-space or polynomial-time). □

5.3.3 Multi-Party Communication Complexity

Let us first recall the multi-party communication model. We use the number on the forehead model [BNS92], where the input consists of k strings x_1, ..., x_k each of length n such that the j^{th} player sees each string except x_j. For a randomized protocol all players also have read-only access to a publicly shared random string r. The players communicate by taking turns writing messages on a shared blackboard until one of the players stops the protocol and outputs an answer. A randomized protocol M using m bits of public randomness computes a language L with error ρ if for every $(x_1, ..., x_k) \in L$, $\Pr_{R \leftarrow U_m}[M(x_1, ..., x_k; r) \neq L(x)] \leq \rho$. A protocol is polynomial-time uniform if whenever a player sends a message, that message

can be computed in polynomial time as a function of the player's view. We similarly define the notion of log-space uniformity.

[Sha09] proves a typically-correct derandomization result for uniform two-party communication protocols. The proof of [Sha09] is tailored to the two-party case and does not extend to the general case of k-party communication. Using our approach we can handle $k > 2$. We show that every uniform randomized k-party communication protocol has a corresponding uniform deterministic k-party communication protocol that is typically correct and has a communication cost that is larger by a factor roughly equal to the amount of randomness of the original randomized protocol. The following statement holds for both log-space and poly-time uniformity.

THEOREM 5.11 (TYPICALLY-CORRECT DERANDOMIZATION OF COMMUNICATION PROTOCOLS) *Let L be a language over k-tuples of n-bit strings and let M be a uniform randomized communication protocol that computes L with error at most ρ using k players, q bits of communication, and m bits of public randomness, with k, q, m, and $\log(1/\epsilon)$ functions computable within the uniformity bounds. There is a positive constant α such that for $q' = \alpha \cdot 4^k \cdot m \cdot (q + \log(m/\epsilon))$ there is a uniform deterministic communication protocol D using k players and q' bits of communication that computes L to within $3\rho + \epsilon$ if $q' \leq n$.*

For $k = 2$, Theorem 5.11 yields a weaker result than that of [Sha09] – which gives a deterministic protocol with communication complexity $O(q + m)$ rather than $O(q \cdot m + m \log m)$ – although we can also obtain the stronger result of [Sha09] using the pseudorandom generator approach, as explained in Section 5.4.

We point out that the error term 3ρ can be removed by using error reduction. For example, by using randomness-efficient error reduction [CW89, IZ89], for any constant c the randomized protocol M can be replaced with a protocol M' that has error at most $\frac{1}{n^c}$ using $m + O(\log n)$ random bits and $O(q \cdot \log n)$ bits of communication.

Proof of Theorem 5.11. Let M be a uniform randomized communication protocol that computes a language L with error at most ρ on every input and uses k players, q bits of communication and m bits of public randomness. We obtain the typically-correct deterministic protocol D by using Item 2 of Lemma 5.3 with the following seed-extending hardness-based pseudorandom generator $G_{H;n,\ell,m}$. The generator simply partitions its inputs into ℓ disjoint blocks and applies a hard function H on each block in order to generate the m pseudorandom bits. More precisely, for any $\ell \leq \lfloor n/m \rfloor$ we define $G_{H;n,\ell,m}$ as

$$G_{H;n,\ell,m}(x_1, \ldots, x_k) = (x_1, \ldots, x_k; H(x_1|_{S_1}, \ldots, x_k|_{S_1}), \ldots, H(x_1|_{S_m}, \ldots, x_k|_{S_m})),$$

where $S_1, ..., S_m$ are disjoint subsets of $[n]$ each of size ℓ and $x|_{S_i}$ is the substring of x of length ℓ formed by taking the bits of x indexed by S_i. We point out that $G_{H;n,\ell,m}$ is only well-defined when $\ell \cdot m \leq n$. G has the property that if H is $(\frac{1}{2} - \frac{\epsilon}{m})$-hard for non-uniform communication protocols operating on k-tuples of ℓ-bit inputs that use q bits of communication, then G is ϵ-pseudorandom against non-uniform randomized communication protocols that operate on k-tuples of n-bit inputs, use m bits of randomness, and use q bits of communication. This pseudorandomness guarantee can be argued directly; it also follows from the remark after the proof of Lemma 5.4, where we observe that G can be seen as a degenerate case of the Nisan-Wigderson construction.

We next set the parameters and the language H so as to ensure that the function

$$D(x_1, ..., x_k) = M(G_{H;n,\ell,m}(x_1, ..., x_k))$$

is within $3\rho + \epsilon$ from L (as long as $q' \leq n$).

1. Pseudorandom generator $G_{H;n,\ell,m}$.

   By Lemma 5.3, D computes L to within $3\rho + \epsilon$ if $G_{H;n,\ell,m}$ is a seed-extending ϵ-pseudorandom generator secure against tests $T_{r'}$ of the form $T_{r'}(x, r) = M(x_1, ..., x_k; r) \oplus M(x_1, ..., x_k; r')$, which are communication protocols that use at most $2q$ bits of communication.

2. Hard language H.

   By the pseudorandomness property stated above, $G_{H;n,\ell,m}$ is ϵ-pseudorandom for tests $T_{r'}$ if H is $(\frac{1}{2} - \frac{\epsilon}{m})$-hard on k-tuples of ℓ-bit inputs for protocols that use $2q$ bits of communication. [BNS92] demonstrate a function, the generalized inner product, which for some positive constant β and any $\epsilon' > 0$ is $(\frac{1}{2} - \epsilon')$-hard for non-uniform k-party communication protocols on k-tuples of ℓ-bit inputs that use at most $\beta \cdot (\frac{\ell}{4^k} - \log(1/\epsilon'))$ bits of communication. Letting H be this function, H has the hardness needed if $2q \leq \beta \cdot (\frac{\ell}{4^k} - \log(m/\epsilon))$. We choose $\ell = \lceil 4^k \cdot (\frac{2q}{\beta} + \log(m/\epsilon)) \rceil$ so that if $\ell \cdot m \leq n$ then $G_{H;n,\ell,m}$ is well-defined and H has the required hardness.

We conclude that for $\ell = \lceil 4^k \cdot (\frac{2q}{\beta} + \log(m/\epsilon)) \rceil$, if $\ell \cdot m \leq n$ then $G_{H;n,\ell,m}$ is an ϵ-pseudorandom generator against the tests $T_{r'}$ and thus D computes L to within $3\rho + \epsilon$.

We next exhibit a protocol of the prescribed form to evaluate the function

$$D(x_1, \ldots, x_k) = M(x_1, \ldots, x_k; H(x_1|_{S_1}, \ldots, x_k|_{S_1}), \ldots, H(x_1|_{S_m}, \ldots, x_k|_{S_m})).$$

Phase 0: All players calculate the value ℓ given above and terminate the protocol if $\ell \cdot m > n$.

Phase 1: Player 1 writes $x_2|_{S_1}, ..., x_2|_{S_m}$ on the public blackboard.

Phase 2: Player 2 evaluates each of $H(x_1|_{S_1}, ..., x_k|_{S_1}), ..., H(x_1|_{S_m}, ..., x_k|_{S_m})$ and writes the results on the public blackboard.

Phase 3: All players execute the protocol for M on input $(x_1, ..., x_k; r)$ using the bits written on the blackboard from Phase 2 as the random bits r.

Phase 1 requires $\ell \cdot m$ bits of communication and guarantees that player 2 has all inputs needed to evaluate H in Phase 2, Phase 2 requires m bits of communication, and Phase 3 requires q bits of communication. Altogether we can evaluate D using $\ell \cdot m + m + q$ bits of communication. Taking α a sufficiently large constant such that $q' = \alpha \cdot 4^k \cdot m \cdot (q + \log(m/\epsilon)) \geq \ell \cdot m + m + q$, the protocol requires at most q' bits of communication. Noting that $q' > \ell \cdot m$ we also have that $G_{H;n,\ell,m}$ is well-defined and D computes L to within $3\rho + \epsilon$ if $q' \leq n$.

We finally remark on the uniformity of the construction. Each player must determine the block size ℓ, execute the protocol M, and player 2 must compute H. The latter can be performed in logarithmic space for H the generalized inner product problem, and the remainder can be done within the uniformity bounds of M assuming each of the quantities k, q, m, and $\log(1/\epsilon)$ are constructible within the uniformity bounds. □

5.4 Comparison with the Extractor-Based Approach

We have seen several settings in which seed-extending pseudorandom generators allow us to prove typically-correct derandomization results that do not follow from an earlier extractor-based approach of [Sha09]. We now show that the approach of [Sha09] is essentially equivalent to having seed-extending pseudorandom generators with *exponentially small error*. This reaffirms our claim that our approach is more general since we additionally obtain meaningful results from pseudorandom generators with larger error. The comparison also leads to the question how much randomness both approaches can handle.

Overview of the Extractor-Based Approach We start with a high-level overview of the approach of [Sha09] that uses a notion of extractors for recognizable distributions, which we now explain. For any function $f : \{0,1\}^n \to \{0,1\}$, [Sha09] defines the distribution *recognized* by f as $U_n|f = 1$, i.e., the uniform distribution over $f^{-1}(1) = \{x \in \{0,1\}^n \mid f(x) = 1\}$. A function $E : \{0,1\}^n \to \{0,1\}^m$ is a (k,ϵ)-extractor for distributions recognizable by some collection of functions $f : \{0,1\}^n \to \{0,1\}$, if for every such function f with $|f^{-1}(1)| \geq 2^k$, the distribution $E(U_n|f = 1)$ has statistical distance at most ϵ from the uniform distribution on m bit strings, i.e.,

$$\sum_{r \in \{0,1\}^m} |\frac{1}{2^m} - \Pr_{X \leftarrow U_n}[E(X) = r | f(X) = 1]| \leq \epsilon.$$

[Sha09] shows the following general approach towards typically-correct derandomization. Let $M : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ be a randomized algorithm that computes some language

L with error ρ at length n. Let $\Delta = 100m$ and let $E : \{0,1\}^n \to \{0,1\}^m$ be an $(n - \Delta, 2^{-\Delta})$-extractor for distributions recognizable by functions of the form $f_{r_1,r_2}(x) = M(x, r_1) \oplus M(x, r_2)$ where $r_1, r_2 \in \{0,1\}^m$ are arbitrary strings. Then $D(x) = M(x, E(x))$ is within $3\rho + 2^{-10m}$ of L at length n.

Comparison The above approach requires extractors with error that is exponentially small in m, and breaks down completely when the error is larger. We now observe that an extractor with exponentially small error yields a seed-extending pseudorandom generator with exponentially small error.

THEOREM 5.12 *Let $T : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ be a function. Let $\Delta = m + \log(1/\epsilon)$ and let $E : \{0,1\}^n \to \{0,1\}^m$ be an $(n - \Delta, 2^{-\Delta})$-extractor for distributions recognizable by functions of the form $f_r(x) = T(x, r)$ where $r \in \{0,1\}^m$ is an arbitrary string. Then, $G(x) = (x, E(x))$ is ϵ-pseudorandom for T.*

As a consequence the extractors used in [Sha09] can be viewed as seed-extending pseudorandom generators with exponentially small error. More precisely, given a randomized algorithm $M : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ the extractor-based approach sets $\Delta = 100m$ and requires an $(n - \Delta, 2^{-\Delta})$-extractor for distributions that are recognizable by functions of the form $f_{r_1,r_2}(x) = M(x, r_1) \oplus M(x, r_2)$. The pseudorandom generator approach of this chapter requires a seed-extending generator $G(x) = (x, E(x))$ that fools tests of the form $T_{r_2}(x, r_1) = M(x, r_1) \oplus M(x, r_2) = f_{r_1,r_2}(x)$. By Theorem 5.12, an extractor E that can be used to obtain typically-correct derandomization following the extractor-based approach gives rise to a seed-extending ϵ-pseudorandom generator $G(x) = (x, E(x))$ with $\epsilon = 2^{m-\Delta} = 2^{-99m} < 2^{-10m}$ which can be used to obtain typically-correct derandomization following the approach of this chapter.

We remark that in some algorithmic settings, e.g., 2-party communication protocols, [Sha09] obtains typically-correct derandomizations that are more efficient than the ones that follow from applying our methodology directly based on the NW-construction and known

hardness results. Nevertheless, by Theorem 5.12 the extractors used in [Sha09] define seed-extending pseudorandom generators that yield typically-correct derandomizations matching the efficiency of the extractor-based approach.

We now prove Theorem 5.12. The analysis below uses the same approach as the analysis of [Sha09] showing that extractors yield typically-correct derandomization.

Proof of Theorem 5.12. Consider a probability space with two independent random variables $X \leftarrow U_n$ and $R \leftarrow U_m$. By conditioning on R we have that

$$\begin{aligned}
&|\Pr[T(X,R)=1] - \Pr[T(X,E(X))=1]| \\
&= \Big|\sum_{r\in\{0,1\}^m} \Pr[T(X,r)=1 \wedge R=r] - \Pr[T(X,r)=1 \wedge E(X)=r]\Big| \\
&= \Big|\sum_{r\in\{0,1\}^m} \Pr[T(X,r)=1]\cdot(\Pr[R=r \mid T(X,r)=1] - \Pr[E(X)=r \mid T(X,r)=1])\Big| \\
&\leq \sum_{r\in\{0,1\}^m} \Pr[T(X,r)=1]\cdot|\Pr[R=r \mid T(X,r)=1] - \Pr[E(X)=r \mid T(X,r)=1]| \qquad (5.6)
\end{aligned}$$

We next argue that the contribution of each individual $r \in \{0,1\}^m$ to the right-hand side of (5.6) is at most $2^{-\Delta}$. This yields an upper bound of $2^m 2^{-\Delta} = \epsilon$ on the left-hand side of (5.6), which by definition means that $G(x) = (x, E(x))$ is ϵ-pseudorandom for T.

We consider two cases. If $\Pr[T(X,r)=1] < 2^{-\Delta}$ then the contribution of r to the right-hand side of (5.6) is less than $2^{-\Delta}$ because of the first factor. Otherwise, the set $f_r^{-1}(1)$ has size at least $2^{n-\Delta}$ and by the given extractor property of E, $|\frac{1}{2^m} - \Pr[E(X)=r \mid f_r(X)=1]| \leq 2^{-\Delta}$. Since $\Pr[R=r \mid T(X,r)=1] - \Pr[E(X)=r \mid T(X,r)=1] = \frac{1}{2^m} - \Pr[E(X)=r \mid f_r(X)=1]$, the second factor on the right-hand side of (5.6) is at most $2^{-\Delta}$, and so is the entire term corresponding to r. □

Conversely, we observe that seed-extending pseudorandom generators with error that is exponentially small in m yield extractors for recognizable distributions.

THEOREM 5.13 *Let $f : \{0,1\}^n \to \{0,1\}$ be a function and let $E : \{0,1\}^n \to \{0,1\}^m$ be a function such that $G(x) = (x, E(x))$ is ϵ-pseudorandom for tests $T(x,r)$ of the form*

$T_z(x,r) = f(x) \wedge (r = z)$ where $z \in \{0,1\}^m$ is an arbitrary string. If $\epsilon \leq 2^{-(m+2\Delta)}$ then E is an $(n-\Delta, 2^{-\Delta})$-extractor for the distribution recognized by f.

Proof of Theorem 5.13. Consider the test $T_z(x,r) = f(x) \wedge (r = z)$ for any $z \in \{0,1\}^m$. By the given pseudorandomness property we have that for $X \leftarrow U_n$ and $R \leftarrow U_m$,

$$\begin{aligned} &|\Pr[T_z(X,R) = 1] - \Pr[T_z(X,E(X)) = 1]| \\ &= |\Pr[f(X) = 1] \cdot \Pr[R = z] - \Pr[f(X) = 1] \cdot \Pr[E(X) = z \,|\, f(X) = 1]| \\ &= \Pr[f(X) = 1] \cdot |\Pr[R = z] - \Pr[E(X) = z \,|\, f(X) = 1]| \leq \epsilon. \end{aligned}$$

Letting P denote the distribution recognized by f and setting $k = \log(|f^{-1}(1)|)$, we can rewrite the above inequality as $2^{k-n} \cdot |2^{-m} - \Pr[E(P) = z]| \leq \epsilon$, which implies that

$$\sum_{z \in \{0,1\}^m} |2^{-m} - \Pr[E(P) = z]| \leq 2^m \epsilon / 2^{k-n}. \tag{5.7}$$

We want to show that the right-hand side of (5.7) is at most $2^{-\Delta}$ for $k \geq n - \Delta$. This is the case since $\epsilon \leq 2^{-(m+2\Delta)}$. □

Together, Theorems 5.12 and 5.13 essentially say that in many algorithmic settings, $(n - cm, 2^{-cm})$-extractors for a sufficiently large constant $c > 1$ give seed-extending pseudorandom generators with error $\epsilon = 2^{-c'm}$ for a constant $c' > 1$ and vice versa. As a consequence the approach of [Sha09] is essentially equivalent to the special case of seed-extending pseudorandom generators with error that is exponentially small. This means that the results we obtain using seed-extending pseudorandom generators with larger than exponentially small error, such as the conditional result of Theorem 1.1 and the unconditional result of Theorem 1.2, do not follow from the [Sha09] approach.

Handling algorithms that toss a super-linear number of coins Another advantage of the approach of this chapter is that we can directly handle randomized algorithms that toss a super-linear number of coins. This is because we can use *stretching* seed-extending pseudorandom generators, in which the length of the extending part $E(x)$ is super-linear.

In contrast an extractor $E(x)$ cannot have an output length that is super-linear as it is impossible to extract more random bits than are present in the input distribution. Indeed, this is why [Sha09] handles randomized algorithms that toss a super-linear number of coins by first applying a pseudorandom generator to reduce the number of coins to sub-linear and only then running an extractor.

In some algorithmic settings both the approach of this chapter and [Sha09] can only handle sub-linear randomness. For example, consider the setting of communication protocols from Section 5.3.3. We cannot hope for unconditional stretching seed-extending pseudorandom generators that fool tests $M(x_1, \ldots, x_k; r)$ defined by randomized k-party communication protocols. This is because in such a protocol we only place limitations on communication complexity and allow the computation of an arbitrary function of r for free. Therefore, such a protocol can implement any statistical test at no cost and distinguish a uniformly chosen string r from one that is generated deterministically from fewer random bits. Even if we restrict our attention to polynomial-time uniform protocols, we are still allowing each party in the protocol to apply an arbitrary polynomial-time computable function to the public random coin sequence r. Thus, the existence of a pseudorandom generator for such protocols presumes the existence of pseudorandom generators for polynomial time, which we do not know to exist unconditionally.

More generally, what differentiates randomized communication protocols from say randomized algorithms corresponding to $\mathrm{BP.AC}^0$ is the way that they are charged for performing computations on the random coin sequence r. Communication protocols can compute any function of r for free, whereas algorithms for $\mathrm{BP.AC}^0$ are restricted to functions in AC^0. It remains open whether one can obtain typically-correct derandomizations of communication protocols that toss a super-linear number of coins.[3]

[3] [New91] shows that every randomized k-party communication protocol can be simulated by another randomized k-party protocol which tosses only $O(\log n)$ coins. However, the transformation does not preserve uniformity.

Chapter 6

Typically-Correct Derandomization and Circuit Lower Bounds

In Chapter 5 we developed an approach to typically-correct derandomization based on seed-extending pseudorandom generators. In this chapter we consider what will be needed to prove typically-correct derandomization of BPP. In Section 6.1 we show that typically-correct derandomization of BPP with very small error rates implies either super-polynomial Boolean circuit lower bounds for NEXP or super-polynomial arithmetic lower bounds for the permanent. This is a generalization of a result of [KI04]; we also develop a new proof for the everywhere-correct setting that scales better with different parameters of the result. In Section 6.2 we show that any typically-correct derandomization of BPP even with very large error rates will require non-algebrizing, non-relativizing proof techniques.

6.1 Circuit Lower Bounds

It is well-known that the existence of pseudorandom generators for polynomial-size circuits (which yields everywhere-correct derandomization of BPP) implies that EXP does not have polynomial-size circuits; this is the easy direction of the hardness versus randomness tradeoffs. Impagliazzo et al. [IKW02] showed that everywhere-correct derandomization of promise-BPP into NSUBEXP implies that NEXP does not have polynomial-size circuits. Building on [IKW02], Kabanets and Impagliazzo [KI04] showed that everywhere-correct derandomization of BPP into NSUBEXP implies that NEXP does not have Boolean circuits of polynomial size or that the permanent over $\mathbb{Z}$ does not have arithmetic circuits of polynomial

size. We present a simpler proof of the latter result and show how to extend it to the setting of typically-correct derandomization. In addition, our proof scales better than the one in [KI04], yields the same lower bound for a smaller class, and does not rely on the result from [IKW02] that NEXP having polynomial-size circuits implies that NEXP coincides with EXP.

6.1.1 Results

In the following, ACZ denotes the language of all arithmetic circuits that compute the zero polynomial over $\mathbb{Z}$. Perm denotes the permanent of matrices over $\mathbb{Z}$, and 0-1-Perm its restriction to matrices with all entries in $\{0, 1\}$. SIZE($s(n)$) denotes Boolean circuits of size $s(n)$, and ASIZE($a(n)$) denotes arithmetic circuits of size $a(n)$. See Section 2.6 for further details on the definitions for circuit size.

Everywhere-Correct Derandomization Our approach yields the following parameterized version of the main result of [KI04], namely circuit lower bounds that follow from everywhere-correct derandomization of the specific BPP-language ACZ. We use $(\mathrm{N} \cap \mathrm{coN})\mathrm{TIME}(\cdot)$ as a shorthand for $\mathrm{NTIME}(\cdot) \cap \mathrm{coNTIME}(\cdot)$.

THEOREM 6.1 *Let $\gamma(n)$ denote the maximum circuit complexity of Boolean functions on n inputs. There exists a constant $c > 0$ such that the following holds for any functions $a(\cdot)$, $s(\cdot)$, and $t(\cdot)$ such that $a(\cdot)$ and $s(\cdot)$ are constructible, $a(\cdot)$ and $t(\cdot)$ are monotone, and $n \leq s(n) < \gamma(n)$.*

If $\mathrm{ACZ} \in \mathrm{NTIME}(t(n))$ *then*

(i) $(\mathrm{N} \cap \mathrm{coN})\mathrm{TIME}\left(t((s(n))^c \cdot a((s(n))^c))\right) \not\subseteq \mathrm{SIZE}(s(n))$, *or*

(ii) $\mathrm{Perm} \notin \mathrm{ASIZE}(a(n))$.

In particular, we obtain the following instantiation for the exponential time bounds considered for part (i) in [KI04].

COROLLARY 6.2 *Let $a(\cdot)$, $s(\cdot)$, and $t(\cdot)$ be functions such that $a(\cdot)$ and $s(\cdot)$ are constructible, $a(\cdot)$ and $t(\cdot)$ are monotone, and $s(n) \geq n$. The following holds as long as for every constant c and sufficiently large n,*

$$t\left((s(n))^c \cdot a((s(n)^c))\right) \leq 2^n. \tag{6.1}$$

If $\mathrm{ACZ} \in \mathrm{NTIME}(t(n))$ *then*

(i) $(\mathrm{N} \cap \mathrm{coN})\mathrm{TIME}(2^n) \not\subseteq \mathrm{SIZE}(s(n))$, *or*

(ii) $\mathrm{Perm} \notin \mathrm{ASIZE}(a(n))$.

Let us compare Theorem 6.1 and Corollary 6.2 to the corresponding results in [KI04]. First, we point out that part (i) states a lower bound for $(\mathrm{N} \cap \mathrm{coN})\mathrm{TIME}(\cdot)$ rather than for $\mathrm{NTIME}(\cdot)$, where we use $(\mathrm{N} \cap \mathrm{coN})\mathrm{TIME}(\cdot)$ as a shorthand for $\mathrm{NTIME}(\cdot) \cap \mathrm{coNTIME}(\cdot)$. Theorem 6.1 and Corollary 6.2 give such a lower bound for the entire range of the parameters; [KI04] only manages to do so in the case where all the parameters are polynomially bounded. More importantly, due to the use of the implication that EXP having polynomial-size circuits implies that EXP coincides with MA [BFNW93], the arguments in [KI04] can only give lower bounds for time bounds on the left-hand side of (i) that are exponential. This is true even when all of $a(n)$, $s(n)$, and $t(n)$ are polynomial, in which case our Theorem 6.1 only needs the time bound in the left-hand side of (i) to be superpolynomial. Finally, due to its dependence on the result from [IKW02] that NEXP having polynomial-size circuits implies that NEXP coincides with EXP, the proof in [KI04] only works when $s(n)$ is polynomially bounded; our proof gives nontrivial results for $s(n)$ ranging between linear and linear-exponential.[1]

Typically-Correct Derandomization We initiate the study of whether typically-correct derandomization of BPP implies circuit lower bounds. We show that it does in the case of typically-correct derandomizations that run in NSUBEXP and are of the quality considered by Goldreich and Wigderson [GW02].

[1] Scott Aaronson and we independently came up with an earlier argument that does not rely on [IKW02] but does use [BFNW93]. The result does not scale as well as Theorem 6.1 and can only handle time bounds on the left-hand side of (i) that are exponential. See [AM10] for more details.

THEOREM 1.3 *If for every positive constant ϵ there exists a nondeterministic Turing machine which runs in time 2^{n^ϵ} and correctly decides ACZ on all but at most 2^{n^ϵ} of the inputs of length n for almost every n, then*

(i) NEXP does not have Boolean circuits of polynomial size, or

(ii) Perm does not have arithmetic circuits of polynomial size.

Note that Theorem 1.3 strengthens the main result of [KI04], which establishes the theorem in the special case where the nondeterministic machines decide ACZ correctly on all inputs. We can parameterize Theorem 1.3 in the same way as Theorem 6.1. However, we only obtain nontrivial results for polynomially bounded $a(n)$ and $s(n)$, in which case $t(n)$ can be subexponential. For that reason, we only state the latter special case. The error rate considered in Theorem 1.3 is the largest one for which our argument gives nontrivial lower bounds.

We first prove Theorem 1.3 and then analyze how the argument parameterizes to Theorem 6.1 and Corollary 6.2 in the case of zero error rate. We end with some extensions and variations of both theorems.

6.1.2 Proof for the Typically-Correct Setting

The proof of Theorem 1.3 has two main ingredients. The first ingredient is an unconditional circuit lower bound for $\mathrm{P}^{\text{0-1-Perm[1]}}$, the class of languages that can be decided in polynomial time with one query to an oracle for 0-1-Perm.

CLAIM 6.3 *For every constant d, $\mathrm{P}^{\text{0-1-Perm[1]}} \not\subseteq \mathrm{SIZE}(n^d)$.*

The second ingredient gives a conditional simulation of that class in nondeterministic subexponential time with subpolynomial advice.

CLAIM 6.4 *If the hypothesis of Theorem 1.3 holds and* Perm *has arithmetic circuits of polynomial size, then*

$$\mathrm{P}^{\text{0-1-Perm[1]}} \subseteq \cap_{\epsilon>0}\mathrm{NTIME}(2^{n^\epsilon})/n^\epsilon.$$

By combining both claims we obtain that if the hypothesis of Theorem 1.3 holds and Perm has arithmetic circuits of polynomial size, then for every constant d, $\mathrm{NTIME}(2^n)/n \not\subseteq \mathrm{SIZE}(n^d)$. The latter implies that for every constant d, $\mathrm{NTIME}(2^n) \not\subseteq \mathrm{SIZE}(n^d)$. Otherwise, any language in $\mathrm{NTIME}(2^n)/n$ can be decided on inputs of length n by a circuit of size $(2n)^d$, namely a circuit simulating an $\mathrm{NTIME}(2^m)$-computation on an input of length $m = n + n$ with its second input hardwired to an advice string of length n. Since NEXP contains a language that is hard for $\mathrm{NTIME}(2^n)$ under linear-time reductions, the statement that $\mathrm{NTIME}(2^n) \not\subseteq \mathrm{SIZE}(n^d)$ for every constant d implies that NEXP does not have circuits of polynomial size. This finishes the proof of Theorem 1.3 modulo the proofs of both claims.

Proof of Claim 6.3. The claim follows because the polynomial-time hierarchy PH does not have circuits of fixed polynomial size [Kan82], PH is contained in $\mathrm{P}^{\#\mathrm{P}[1]}$ [Tod91], and 0-1-Perm is complete for #P under reductions that make a single query [Zan91]. □

In the rest of the proof we establish Claim 6.4.

Proof of Claim 6.4. It is enough to consider $\mathrm{P}^{\text{0-1-Perm}[1]}$-computations that run in time n. Consider such a computation, and let M denote the query it makes to its 0-1-Perm-oracle on a given input of length n. The dimension m of M cannot exceed $\sqrt{n}$ as the computation does not have enough time to generate larger square matrices. By the paddability of 0-1-Perm, we can assume without loss of generality that M has dimension $m = \sqrt{n}$ independent of the input of length n, and maintain a running time of $O(n)$.

It suffices to design, for every $\epsilon > 0$, a nondeterministic machine N_ϵ running in time 2^{n^ϵ} and an advice sequence $a(\cdot, \epsilon)$ where $a(n, \epsilon)$ has length at most n^ϵ such that the following holds: On input an m-by-m 0-1-matrix M, N_ϵ with advice $a(n, \epsilon)$ outputs $\mathrm{Perm}(M)$ on every accepting computation path, and has at least one such computation path. Our machine N_ϵ acts as follows.

1. Guess a polynomial-sized candidate arithmetic circuit C for Perm on matrices of dimension m.

2. Verify the correctness of C. Halt and reject if the test fails.

3. Use the circuit C to determine the permanent of M in deterministic polynomial time.

The circuit in step 1 exists by virtue of the hypothesis that Perm has polynomial-size arithmetic circuits. Say the circuit C we guess is of size $s \leq m^b$ and purportedly computes the permanent of m-by-m matrices over $\mathbb{Z}$. The constant b is chosen large enough so that such a circuit exists. The crux of the procedure is the second step, which is a nondeterministic test that has an accepting computation path on input C iff C does what it is purported to do. Once that test is passed, we evaluate C modulo $m!+1$ on the given 0-1-matrix M. Evaluating C this way ensures that the intermediate results remain small so the computations can be done in polynomial time; since the permanent of M is a non-negative integer no larger than $m!$, the outcome of the computation gives the correct value of the permanent of M.

The test in the second step is based on the following well-known translation to ACZ exploiting the downward self-reducibility of the permanent. For completeness we include a proof.

LEMMA 6.5 *There exists a polynomial-time algorithm that takes an arithmetic circuit C and an integer m, and produces an arithmetic circuit $\tilde{C}$ such that C computes the permanent of m-by-m matrices over $\mathbb{Z}$ iff $\tilde{C} \in$ ACZ.*

Proof. We use the following notation. Let M be an m-by-m matrix M, $0 \leq k \leq m$, and $1 \leq i, j \leq k$. We denote by $M^{(k)}$ the matrix obtained by taking the m-by-m identity matrix and replacing the top left k-by-k submatrix by the corresponding submatrix of M. By $M^{(k-1)}_{-i,-j}$ we denote the same for $k-1$ but starting from the matrix M with the i-th row and j-th column deleted.

We have that C correctly computes the permanent of m-by-m matrices over $\mathbb{Z}$ iff for each $1 \leq k \leq m$, the polynomial

$$\tilde{C}_k = C(X^{(k)}) - \sum_{j=1}^{k} C(X^{(k-1)}_{-k,-j}) \cdot x_{kj}$$

is identically zero, as well as the polynomial $\tilde{C}_0 = C(X^{(0)}) - 1$, where X denotes an m-by-m matrix of variables $(x_{ij})_{i,j=1}^{m}$. By introducing one more variable x_0, those conditions can be expressed equivalently as whether the following polynomial is identically zero: $\tilde{C} = \sum_{k=0}^{m} \tilde{C}_k \cdot x_0^k$. The straightforward implementation of $\tilde{C}$ given C yields an arithmetic circuit that consists of $O(m^2)$ copies of C and some simple additional circuitry. That arithmetic circuit is in ACZ iff C correctly computes the permanent on m-by-m matrices over $\mathbb{Z}$. □

We use Lemma 6.5 to transform the circuit C into the circuit $\tilde{C}$, and show how to test that $\tilde{C}$ is in ACZ. We will exploit the fact that ACZ is in coNP and that it is highly paddable to transform the almost-correct nondeterministic subexponential-time tests given by the hypothesis of Theorem 1.3 into perfect nondeterministic subexponential-time tests for ACZ with small advice. Let N'_ϵ denote the nondeterministic Turing machine from the hypothesis of Theorem 1.3 corresponding to ϵ. We will use $N'_{\epsilon'}$ for some ϵ' related to ϵ.

Note that the false positives $\tilde{C}$ of $N'_{\epsilon'}$ can be detected nondeterministically by guessing an accepting computation path of $N'_{\epsilon'}$ on input $\tilde{C}$, guessing an input x and a modulus μ, evaluating $\tilde{C}$ on input x modulo μ, and verifying that the result is nonzero. Since the modulus μ never needs to be larger than $2^{\tilde{s}}$, where $\tilde{s}$ denotes the size of the circuit $\tilde{C}$, the overhead of the test beyond running $N'_{\epsilon'}$ is only polynomial. Now, suppose that we are given the exact number $\mathrm{fp}(\tilde{s}, \epsilon')$ of false positives of $N'_{\epsilon'}$ at length $\tilde{s}$. Then the following nondeterministic test for membership to ACZ is sound for instances $\tilde{C}$ of length $\tilde{s}$, i.e., if the test accepts $\tilde{C}$ then $\tilde{C}$ is in ACZ for sure.

(a) Guess a list of $\mathrm{fp}(\tilde{s}, \epsilon')$ distinct instances of length $\tilde{s}$ and nondeterministically test that they are all false positives of $N'_{\epsilon'}$. If there is a test that fails, halt and reject.

(b) Accept iff $\tilde{C}$ is not on that list.

Note that this test runs in time $\mathrm{fp}(\tilde{s}, \epsilon') \cdot 2^{\tilde{s}^{\epsilon'}} \cdot \mathrm{poly}(\tilde{s})$, which is $2^{O(\tilde{s}^{\epsilon'})}$. Note also that we can make sure that the size s of C as well as the size $\tilde{s}$ of $\tilde{C}$ only depend on m in an easily computable way, say $\tilde{s} = m^c$ for some constant c. This follows from the paddability of circuit

descriptions as mentioned in Section 2.6. As a result, the information $\mathrm{fp}(\tilde{s}, \epsilon')$ really takes on the form of an advice.

The above test is sound but not necessarily complete – it may still have false negatives. In order to remedy that problem, we exploit a further paddability property of circuit descriptions, namely that we can obtain many different circuits equivalent to a given circuit by adding a little bit of circuitry that isn't used in the evaluation of the output gate. Consider the equivalents of $\tilde{C} \in \mathrm{ACZ}$ of length ℓ that we can obtain using this type of padding. If the number of distinct pads exceeds the total number of errors $N'_{\epsilon'}$ makes at length ℓ, we can nondeterministically guess a pad that is accepted by $N'_{\epsilon'}$ and therefore also by the above test when provided with $\mathrm{fp}(\ell, \epsilon')$ as advice.

How large does ℓ need to be for this approach to work? There exists a positive constant α such that the number of padded versions of $\tilde{C}$ of length $\ell = \tilde{s} + \Delta$ is at least $2^{\alpha\Delta}$. We need $2^{\alpha\Delta} > 2^{\ell^\epsilon}$. The latter condition is satisfied for every $0 < \epsilon < 1$ and sufficiently large $\tilde{s}$ when we set $\Delta = \tilde{s}$, i.e., $\ell = 2\tilde{s}$.

The resulting nondeterministic test for $\tilde{C}$ runs in time

$$2^{O(\ell^{\epsilon'})} = 2^{O(\tilde{s}^{\epsilon'})} = 2^{O(m^{c\epsilon'})}, \tag{6.2}$$

and works correctly when provided $\mathrm{fp}(\ell, \epsilon')$ as advice. The bit length of the advice is bounded by the logarithm of (6.2). Plugging in this test as the second step in the three-step approach mentioned at the beginning of the proof, we obtain a machine N_ϵ with the properties we need for any constant ϵ with $\epsilon > c\epsilon'$ by setting $a(n, \epsilon) = \mathrm{fp}(2m^c, \epsilon')$. □

6.1.3 Proofs for the Everywhere-Correct Setting

We establish Theorem 6.1 by analyzing how the proof of Theorem 1.3 parameterizes in the case of zero error rate.

Proof of Theorem 6.1. The two ingredients in the proof of Theorem 1.3 translate as follows given the parameters of Theorem 6.1.

Claim 6.6 *There exists a constant c such that for every time constructible function $s(\cdot)$ satisfying $n \leq s(n) < \gamma(n)$,* $\mathrm{DTIME}^{\text{0-1-Perm[1]}}((s(n)^c)) \not\subseteq \mathrm{SIZE}(s(n))$.

Claim 6.7 *There exists a constant d such that the following holds for any functions $a(\cdot)$ and $t(\cdot)$ with $a(\cdot)$ constructible and $t(\cdot)$ monotone. If* $\mathrm{ACZ} \in \mathrm{NTIME}(t(n))$ *and* $\mathrm{Perm} \in \mathrm{ASIZE}(a(n))$*, then*

$$\mathrm{DTIME}^{\text{0-1-Perm[1]}}(n) \subseteq \mathrm{NTIME}(t(n \cdot \log^d n \cdot a(\sqrt{n}))).$$

Given those two claims, we obtain the following by padding Claim 6.7 to length $(s(n))^c$, exploiting the closure under complementation of deterministic computations, and combining it with Claim 6.6: If $\mathrm{ACZ} \in \mathrm{NTIME}(t(n))$ and $\mathrm{Perm} \in \mathrm{ASIZE}(a(n))$, then

$$(\mathrm{N} \cap \mathrm{coN})\mathrm{TIME}\left(t((s(n))^c \cdot \log^d((s(n))^c) \cdot a((s(n))^{c/2}))\right) \not\subseteq \mathrm{SIZE}(s(n)).$$

Theorem 6.1 follows by simplifying the last expression using the monotonicity of $a(\cdot)$ and $t(\cdot)$ and the fact that $s(n) \geq n$. All that remains are the proofs of the claims.

Proof of Claim 6.6. The argument of [Kan82] gives that $\Sigma_4\mathrm{TIME}(s(n)\log^a(s(n))) \not\subseteq \mathrm{SIZE}(s(n))$ for some constant a. [Tod91] shows that there exists a constant b and a problem $A \in \#\mathrm{P}$ such that for any constructible function $t(\cdot)$ with $t(n) \geq n$, $\Sigma_4\mathrm{TIME}(t(n)) \subseteq \mathrm{DTIME}^{A[1]}((t(n))^b)$. The claim follows by combining the above as before with the completeness of 0-1-Perm for #P under reductions that make a single query [Zan91]

□

Proof of Claim 6.7. We follow the proof of Claim 6.4 and set $m = \sqrt{n}$.

The crux is the 3-step construction of a nondeterministic machine N that takes an m-by-m 0-1-matrix M and outputs $\mathrm{Perm}(M)$ on every accepting computation path, and has at least one such computation path. In the first step N guess an arithmetic circuit of size $a(m)$. By the constructibility of $a(\cdot)$, this step takes time $O(a(m))$. In the second step, we run the nondeterministic algorithm for ACZ from the hypothesis on the circuit $\tilde{C}$ given by Lemma 6.5. A careful reading of the proof of the lemma reveals that this step takes time

$t(m^2 \cdot \log^d m \cdot a(m))$ for some constant d. The third step takes time $O(m^2 \cdot \log^d m \cdot a(m))$. As we can assume without loss of generality that $t(n) \geq n$ and since $t(\cdot)$ is monotone, the three steps combined take time $O(t(m^2 \cdot \log^d m \cdot a(m)))$. The total running time of the nondeterministic simulation of the given $\mathrm{DTIME}^{\text{0-1-Perm[1]}}(n)$-computation is of the same order. □

This finishes the proof of Theorem 6.1. □

The proof of Corollary 6.2 immediately follows from Theorem 6.1.

Proof of Corollary 6.2. Note that condition (6.1) gives an upper bound of 2^n on the time bound on the left-hand side of (ii) in the statement of Theorem 6.1. Also, we can assume without loss of generality that $t(n) \geq n$ for almost all n; otherwise, the hypothesis of Corollary 6.2 fails as a nondeterministic machine deciding ACZ needs to be able to look at its entire input. Thus, condition (6.1) implies that $s(n)$ is upper bounded by $2^{n/c}$, which is less than $\gamma(n)$ for $c > 1$ and n sufficiently large. Corollary 6.2 then follows from Theorem 6.1 verbatim. □

We already discussed how Theorem 6.1 and Corollary 6.2 compare to the corresponding results in [KI04]. In order to compare our argument with the one from [KI04], let us see how both obtain a contradiction from the hypotheses that ACZ is in NP, NEXP has polynomial-size circuits, and Perm has polynomial-size arithmetic circuits. Both proofs use the first and the third hypothesis to collapse $\mathrm{P}^{\#\mathrm{P}}$ into NP. [KI04] then uses the result from [IKW02] that NEXP having polynomial-size circuits implies that NEXP coincides with EXP, and the result from [BFNW93] that EXP having polynomial-size circuits implies that EXP coincides with MA, to conclude that NEXP is in $\mathrm{P}^{\#\mathrm{P}}$. This in turn collapses NEXP all the way down to NP, which contradicts the time hierarchy for nondeterministic machines. Our proof does not attempt to collapse NEXP into NP. Instead we use the fact that NEXP having polynomial-size circuits implies that NP has circuits of size n^c for some fixed constant c. Since we know unconditionally that $\mathrm{P}^{\#\mathrm{P}}$ does not have the latter property, we obtain a contradiction as we already derived that $\mathrm{P}^{\#\mathrm{P}}$ is in NP.

6.1.4 Extensions

We observe a few variations of Theorems 6.1 and 1.3. First, the theorems also hold when we simultaneously replace ACZ by AFZ (the restriction of ACZ to arithmetic formulas), and "arithmetic circuits" by "arithmetic formulas".

Second, we can play with the underlying i.o. and a.e. quantifiers. In fact, we can strengthen both theorems by either relaxing the hypothesis to hold only i.o. rather than a.e. or by improving one of the lower bound conclusions (i) or (ii) to hold a.e. rather than i.o. This follows because on the one hand the lower bounds in Claims 6.3 and 6.6 hold a.e. rather than just i.o. as stated. On the other hand, if one of the hypotheses of Claims 6.4 and 6.7 holds only i.o., the concluding simulation can be made to work i.o. when provided with a pointer to a nearby input length where the hypotheses hold. The latter can be handled with a logarithmic amount of advice, which the rest of the argument can handle.

As an example, in the case of Theorem 1.3 it suffices for the nondeterministic machines N_ϵ to correctly decide ACZ on all but at most 2^{n^ϵ} of the inputs of length n for *infinitely many* n. Related to the latter variation, we point out that by [IW01] EXP differs from BPP iff all of BPP has deterministic typically-correct derandomizations that run in subexponential time and err on no more than a polynomial fraction of the inputs of length n for infinitely many n. Thus, extending this i.o.-version of Theorem 1.3 to the setting with polynomial error rates would show that EXP$\neq$BPP implies circuit lower bounds.

6.2 Relativization and Algebrization

In Section 6.1, we showed that typically-correct derandomizations of BPP with the parameters considered by Goldreich and Wigderson [GW02] imply circuit lower bounds (Theorem 1.3). In particular, this implies that any proof of such typically-correct derandomization must contain ingredients that prove circuit lower bounds. Although we do not know if typically-correct derandomizations of BPP with the weaker parameters of say Theorem 1.1

imply circuit lower bounds, we do show in this section that such derandomizations would require non-relativizing, and indeed non-algebrizing ingredients.

Algebrization Let us recap the notion of algebrization [AW09], which generalizes the concept of relativization. A complexity class inclusion $\mathcal{C}_1 \subseteq \mathcal{C}_2$ is said to *algebrize* if for every oracle A and every low-degree extension $\tilde{A}$ of A, $\mathcal{C}_1^A \subseteq \mathcal{C}_2^{\tilde{A}}$. A complexity class separation $\mathcal{C}_1 \nsubseteq \mathcal{C}_2$ is said to *algebrize* if for every oracle A and low-degree extension $\tilde{A}$ of A, $\mathcal{C}_1^{\tilde{A}} \nsubseteq \mathcal{C}_2^A$. An inclusion or separation is said to *relativize* if the above holds with $\tilde{A}$ replaced by A.

Notice that any statement which relativizes also algebrizes. The converse does not hold. As an example, the inclusion PSPACE $\subseteq$ IP [Sha92] does not relativize but does algebrize. In fact, [AW09] observe that all known non-relativizing proofs that are based on arithmetization algebrize. At the same time [AW09] argues that several open questions in complexity theory require non-algebrizing techniques to be settled.

Typically-Correct Derandomization and Algebrization We show that the same is true of the question whether typically-correct derandomizations of BPP exist. On the one hand, a negative answer cannot algebrize, even for zero error. This is because ruling out typically-correct derandomization of BPP in particular implies BPP $\nsubseteq$ P, but for any PSPACE-complete language A and its multi-linear extension $\tilde{A}$, $\mathrm{BPP}^{\tilde{A}} \subseteq \mathrm{PSPACE}^{\tilde{A}} \subseteq \mathrm{P}^A$. On the other hand, we show that a positive answer cannot algebrize either, even for very large error rates and even if we only want simulations in nondeterministic subexponential time.

THEOREM 6.8 *There exists an oracle B and a multi-quadratic extension $\tilde{B}$ of B such that there is a language in* $\mathrm{BPTIME}^B(O(n))$ *that is* $(\frac{1}{2} - \frac{1}{2^{n/3}})$*-hard for* $\mathrm{NTIME}^{\tilde{B}}(2^n)$.

Proof. The construction can be broken up into two main parts.

1. Construct B and a multi-quadratic extension $\tilde{B}$ of B such that any language computable in $\mathrm{NTIME}^{\tilde{B}}(2^n)$ can be computed in $\mathrm{BPTIME}^B(c \cdot n)$ for some constant c. The proof follows very closely the construction due to [Wil85] of an oracle B such that

$\mathrm{NTIME}^B(2^n) \subseteq \mathrm{SIZE}^B(c \cdot n)$. [AW09] show that the proof of [Wil85] can be carried out in the more general algebrization setting, showing that there exists an oracle B and a multi-quadratic extension $\tilde{B}$ of B such that $\mathrm{NTIME}^{\tilde{B}}(2^n) \subseteq \mathrm{SIZE}^B(c \cdot n)$. In fact, both in the original result of [Wil85] and the generalization in [AW09], the non-uniformity can be replaced by randomness. That is, we can replace SIZE by BPTIME, which is what we need to complete the first part of the proof.

2. Given B and $\tilde{B}$ construct a hard language L. We derive the hard language L using a relativizing hierarchy theorem of [GW00] for deterministic machines, which shows that for any constant c there is a language $L \in \mathrm{DTIME}^B(2^{O(n)})$ that is $(\frac{1}{2} - \frac{1}{2^{n/3}})$-hard for $\mathrm{DTIME}^B(2^{c \cdot n})$. By the first part $\mathrm{NTIME}^{\tilde{B}}(2^n) \subseteq \mathrm{BPTIME}^B(c \cdot n) \subseteq \mathrm{DTIME}^B(2^{c \cdot n})$, so the language L has the required hardness. Moreover, L is computable in $\mathrm{DTIME}^B(2^{O(n)}) \subseteq \mathrm{NTIME}^{\tilde{B}}(2^{O(n)}) \subseteq \mathrm{BPTIME}(O(n))$, where the latter inclusion follows from $\mathrm{NTIME}^{\tilde{B}}(2^n) \subseteq \mathrm{BPTIME}^B(O(n))$ by padding.

□

We point out that weaker hierarchy theorems for deterministic time could have been used in place of the one from [GW00] in order to conclude that a positive answer cannot algebrize. We stated the result using the [GW00] hierarchy theorem because it holds for almost every input length and achieves hardness very close to $\frac{1}{2}$.

Chapter 7

Derandomizing Monotone Computations

In this chapter we prove our results concerning the derandomization of monotone computations and the relation with derandomizing general computations. In Section 7.1 we introduce the key concept (monotone slice function) and its properties used in the main results of this chapter. In Section 7.2 we show that functions that are average-case hard for monotone circuits are hard with somewhat weaker parameters for general circuits. In Section 7.3 we show that pseudorandom generators that are secure against monotone circuits are secure with somewhat weaker parameters against general circuits. In Section 7.4 we show that derandomizing randomized monotone computations into P would derandomize all of BPP into P.

7.1 Monotone Slice Functions

In this section we introduce the key concept used in many of the proofs in this chapter – slice functions – and discuss the properties we will need. We also define terminology and notation used throughout the chapter.

First, recall that a *monotone* Boolean function is defined by the property that flipping any input bit from 0 to 1 can only change the output value from 0 to 1. Equivalently, a monotone function can be computed by a monotone circuit – a Boolean circuit consisting only of AND and OR gates, i.e., with no NOT gates. An *anti-monotone* function is the negation or complement of a monotone function; anti-monotone functions share many of the key combinatorial properties of monotone functions. For a binary string x, we use the

notation $\overline{x}$ to denote the string resulting from negating each bit of x. Anti-monotone circuits can equivalently be viewed as either negations of monotone circuits or monotone circuits that are given $\overline{x}$ as input rather than x. When we speak of anti-monotone circuits we refer to the former by default.

Slices of the Boolean Cube We use the terminology "k-th slice" of the Boolean n-cube to refer to the set of n-bit strings that have Hamming weight exactly k. We use the notation $|x|$ to refer to the Hamming weight of a string x, so $|x|$ is equal to the number of ones in x. The "middle slice" refers to the $\lfloor n/2 \rfloor$-th slice. The k-th slice contains $\binom{n}{k}$ strings. The middle slice contains $\binom{n}{\lfloor n/2 \rfloor}$ strings, which can be shown to be $\Theta(\frac{1}{\sqrt{n}}2^n)$ by Stirling's formula.

Slice Functions A *monotone slice function for the k-th slice* is a monotone function that can take arbitrary values for inputs on the k-th slice, evaluates to 1 above the k-th slice, and evaluates to 0 below the k-th slice. An *anti-monotone slice function* for the k-th slice takes arbitrary values for inputs on the k-th slice, evaluates to 0 above the k-th slice, and evaluates to 1 below the k-th slice. When applies to both monotone and anti-monotone slice functions, we say simply "slice function".

Two key properties of slice functions play a prominent role in the proofs of this chapter.

(i) The monotone and general circuit complexity of slice functions are polynomially related.

(ii) The truth table of any Boolean function f on n bits can be embedded within the middle slice of another function f' on $m > n$ bits, for an appropriate choice of $m = n + O(\log n)$.

We first prove these properties and then discuss how they are used to prove some of our results.

Monotone Complexity of Slice Functions Here we discuss property (i) from above, that the monotone and general circuit complexity of slice functions is polynomially related.

Berkowitz [Ber82] was the first to observe this, as follows. Let f be a monotone slice function for the k-th slice. Note that for x with Hamming weight exactly k, $\neg x_i = 1$ if and only if $x \setminus x_i$ has weight at least k and $\neg x_i = 0$ if and only if $x \setminus x_i$ has weight less than k. Then given a circuit for computing f, we first push all negations to the inputs (this at most doubles the size of the circuit) and then replace any instance of $\neg x_i$ by a threshold circuit over $n-1$ bits. As thresholds can be computed by $O(n \log n)$ size monotone circuits [AKS83], the resulting monotone circuit is of size $2s + O(n^2 \log n)$.

The construction can also be used to produce an anti-monotone circuit that agrees with f on the k-th slice – produce a monotone circuit computing the monotone slice function that is the complement of f on the k-th slice and then negate this circuit. Similarly, if f is an anti-monotone slice function, the process can be used to produce either a monotone or anti-monotone circuit agreeing with f on the k-th slice.

[Val86] gives a slightly more efficient construction that computes the threshold circuits for each x_i simultaneously with $O(n \log^2 n)$ many gates, which implies that if f has general circuits with s gates then f has monotone circuits with $2s + O(n \log^2 n)$ gates. Further, the construction is poly-time uniform: there is a poly-time machine that on input $(1^n, k, C)$, where C is a general circuit with s gates computing f at length n, outputs a monotone circuit with $2s + O(n \log^2 n)$ gates that computes f at length n.

THEOREM 7.1 ([VAL86]) *Let f be any slice function and let C be a circuit with at most s gates for computing f. There is a monotone circuit C_{mon} and an anti-monotone circuit $C_{anti-mon}$ such that both agree with f on the slice in question, compute slice functions, are of size $2s + O(n \log^2 n)$, and are uniformly constructable given C.*

Embedding Functions Within Slices Here we discuss property (ii) from above, that any function f on n bits can be embedded within a slice of a function f' on $m > n$ bits with m not too much larger than n. First, we describe a very easy method and then present a method with better parameters. One method is to let f' take $m = 2n$ bits as input and embed the truth table of f within the middle slice of f' as follows. For each n-bit string x, set

$f'(x, \overline{x}) = f(x)$. For each $2n$-bit string x' not of the form $(x, \overline{x})$, set $f'(x') = 1$ if $|x'| \geq n$ and set $f'(x') = 0$ otherwise. f' is a monotone slice function, so its monotone circuit complexity is polynomially related to its general circuit complexity by Theorem 7.1. However, only a very small fraction of the middle slice of f' is used in the embedding, namely $\frac{2^n}{\binom{2n}{n}} = \Theta(\frac{\sqrt{n}}{2^n})$.

For our application in Theorem 1.6, we need an embedding that uses a larger fraction of the input space of f'. If we let f' take m-bit inputs, then it is possible to embed the truth table of f into the middle slice of f' provided $\binom{m}{\lfloor m/2 \rfloor} \geq 2^n$. Because the binomial coefficient $\binom{m}{\lfloor m/2 \rfloor}$ grows by less than a factor of 2 for each increment of m, m can also be chosen so that $\binom{m}{\lfloor m/2 \rfloor} \leq 2 \cdot 2^n$, so the embedding occupies a constant fraction of the slice.

Such an embedding follows by associating an n-bit number x with a version of its "k-binomial representation" for an appropriate k, which we now develop. Given any non-negative integers a and $k \leq a$, the identity

$$\binom{a}{k} = \binom{a-1}{k} + \binom{a-2}{k-1} + ... + \binom{a-(k-1)}{2} + \binom{a-k}{1} + 1$$

can be verified by considering the $\binom{a}{k}$ strings of length a with Hamming weight k. Exactly $\binom{a-1}{k}$ of these strings begin with a 0, the first term in the identity. The remaining strings begin with a 1 followed by a string of length $a-1$ with Hamming weight $k-1$. Of these, $\binom{a-2}{k-1}$ begin with a 0, the second term in the identity. The remaining strings begin with 11 followed by a string of length $a-2$ with Hamming weight $k-2$. We can continue in this way until we are left with the number of strings beginning with $1^{k-1}0$ that have Hamming weight k – the second to last term in the identity – and finally the number of strings beginning with 1^k that have Hamming weight k – the last term in the identity.

We use the identity to prove the following claim. To embed f within the k-th slice of the m-cube of a function f', we apply Claim 7.2 and associate x with the m-bit string x' that has ones precisely in positions $a_k + 1$, $a_{k-1} + 1$, ... $a_1 + 1$. We set set f' to be the slice function that has $f'(x') = f(x)$. After proving the claim, we summarize the relevant properties of this embedding in Lemma 7.3.

CLAIM 7.2 *For any integer* $0 \leq x < \binom{m}{k}$, x *has a unique representation as*

$$x = \binom{a_k}{k} + \binom{a_{k-1}}{k-1} + ... + \binom{a_1}{1}$$

where $m > a_k > a_{k-1} > ... > a_1 \geq 0$ *and with the definition that* $\binom{a_i}{i} = 0$ *if* $a_i \leq i$.

Proof. We prove Claim 7.2 by induction, as follows. If $x < \binom{m-1}{k}$ then also $a_k < m-1$, and we can use induction to obtain the representation. If $x \geq \binom{m-1}{k}$, the identity tells us that $a_k = m-1$ because otherwise the terms could not sum to x. Thus $x = \binom{m-1}{k} + y$ for $y = x - \binom{m-1}{k}$. Because $x < \binom{m}{k}$, $y < \binom{m}{k} - \binom{m-1}{k} = \binom{m-1}{k-1}$, and we complete the representation for x by using induction on y with $m' = m-1$ and $k' = k-1$. When we reach $k = 1$, we have by assumption that $x < \binom{m}{1} = m$, and a_k is chosen to be precisely x. The cases for $n = 1$ and $k = 0$ can be easily verified. □

Consider the efficiency of computing this representation and its inverse. Given x, we take the largest a_k such that $\binom{a_k}{k} \leq x$ and recurse. The determination of x given the representation consists of arithmetic. Each of these processes can be carried out in polynomial time.

LEMMA 7.3 *For any positive integers* n, m, k *with* $m > n$, $m > k$ *such that* $\binom{m}{k} \geq 2^n$ *there is a one-to-one mapping* ϕ *from* $\{0,1\}^n$ *into the set of* m*-bit strings with Hamming weight exactly* k*; the mapping is computable and invertible in* $\mathrm{poly}(m)$ *time.*

For any function f*, we define a function* f' *with* $f'(x') = 1$ *for all* x' *with* $|x'| > k$, $f'(x') = f(x)$ *for* x' *with* $|x'| = k$ *and* $x' = \phi(x)$*, and* $f'(x) = 0$ *for all other* x'*. Then* f' *is a slice function for the* k*-th slice, and for any* s*,* f *has circuits of size* $s + \mathrm{poly}(m)$ *if and only if* f' *has circuits of* $s + \mathrm{poly}(m)$.

This is true in particular for the smallest m such that setting $k = \lfloor m/2 \rfloor$ and $\binom{m}{k} \geq 2^n \geq \binom{m}{k}$.

7.1.1 Proof Overviews

Here we give a quick overview how the properties (i) and (ii) stated above are used to prove some of our main results.

Theorem 1.4 states that a general circuit C that approximates a function f can be converted into a monotone approximating circuit C_{mon} with some loss in parameters. The basic idea is to find a slice k on which C computes f well and let C_{mon} be a monotone circuit that computes the monotone slice function that agrees with C on the k-th slice. Property (i) shows that the size of the monotone circuit C_{mon} is not much larger than C.

Theorem 1.5 states that a circuit C that distinguishes some distribution (e.g., the output of a pseudorandom generator) from uniform can be made monotone with some loss in parameters. The main idea is similar to that of Theorem 1.4 but for the setting of a distinguisher rather than computing a function, and again property (i) is key.

Theorem 1.6 states that for any BPP language L there is a language L_{mon} computed by a uniform polynomial-time monotone bounded-error randomized circuit such that L poly-time many-one reduces to L_{mon}. The main idea is to use property (ii) to convert the BPP machine into a monotone circuit and then use property (i) to show the resulting monotone computation has polynomial-size monotone circuits.

7.2 Average-Case Hardness

In this section we prove our results concerning average-case hardness. The main results of this section show the following. (1) Functions that are hard on average for monotone circuits are hard on average for general circuits with somewhat weaker parameters. (2) There exist monotone functions with average-case hardness approaching a barrier implied by results from learning theory.

Reduction to Monotone Circuits Our main result of this section, Theorem 1.4, shows that if we can prove average-case hard functions for monotone circuits with strong enough parameters then we would have average-case hard functions for general circuits with some loss in parameters. We prove this by showing that a circuit which approximates a given function can be made monotone without too much loss in accuracy.

THEOREM 1.4 *Let f be any function. If there is a general circuit C with s gates that computes f to within $\frac{1}{2} - \epsilon$, then there is either a monotone or anti-monotone circuit with $2s + O(n \log^2 n)$ gates that computes f to within $\frac{1}{2} - \epsilon'$ for $\epsilon' = \max(\frac{\epsilon}{n+1}, \frac{c}{\sqrt{n \log(1/\epsilon)}})$ for $c > 0$ an absolute constant.*

Proof. The main idea is that there must be some slice on which C computes f well and contains a large fraction of all inputs. Once this is proven, we show that either the monotone or anti-monotone circuit that agrees with C on the slice in question must compute f on at least $\frac{1}{2} + \epsilon'$ fraction of inputs. The choice between the monotone or anti-monotone circuit is made to ensure the circuit computes f with probability at least $\frac{1}{2}$ on inputs outside of the slice of interest.

We begin by considering for each slice i, the value A_i that the i-th slice contributes to the advantage C has in computing f,

$$A_i = \sum_{x \text{ s.t. } |x|=i} 1_{C(x)=f(x)} - 1_{C(x)\neq f(x)}.$$

We have by assumption that

$$\sum_{i=0}^{n} A_i \geq 2^n(2\epsilon).$$

By an averaging argument, there exists an index i such that $A_i \geq 2^n \frac{2\epsilon}{n+1}$. Theorem 7.1 gives us both a monotone circuit C_{mon} and an anti-monotone circuit $C_{anti-mon}$ of size $2s + O(n \log^2 n)$ that agree with C on the i-th slice. C_{mon} and $C_{anti-mon}$ thus have advantage at least $2^n \frac{2\epsilon}{n+1}$ in computing f on the i-th slice. Because C_{mon} and $C_{anti-mon}$ are complements outside of the i-th slice, exactly one of them agrees with f on at least $\frac{1}{2}$ of all inputs outside of the i-th slice. Altogether, we have that either C_{mon} or $C_{anti-mon}$ has total advantage at least $2^n \frac{2\epsilon}{n+1}$ in computing f; equivalently at least one of the circuits computes f to within $\frac{1}{2} - \frac{\epsilon}{n+1}$.

The alternate value for ϵ' comes by only considering $\Theta(\sqrt{n \log(n/\epsilon)})$ slices around the middle which together contain $1 - \frac{\epsilon}{2}$ fraction of all strings. The Chernoff Bound of Theorem 2.7 tells us that if we pick an n-bit string at random, the probability that the Hamming weight

deviates from $\lfloor \frac{n}{2} \rfloor$ by at least j is at most $\frac{\epsilon}{2}$ if we set j such that $2e^{-(n/2-(j+1))^2/(2n)} \leq \frac{\epsilon}{2}$, so $j = \Theta(\sqrt{n \log(1/\epsilon)})$. Thus we remove from consideration at most $\frac{\epsilon}{2}2^n$ strings by restricting to the $\Theta(\sqrt{n \log(1/\epsilon)})$ many slices closest to the middle, and therefore C must compute f correctly on at least $2^n(\frac{1}{2} + \frac{\epsilon}{2})$ of these. We can now carry out an argument similar to the above – where instead of $n+1$ many slices we consider $\Theta(\sqrt{n \log(1/\epsilon)})$ many and have a circuit that is correct on at least $2^n(\frac{1}{2} + \frac{\epsilon}{2})$ of the strings rather than $2^n(\frac{1}{2} + \epsilon)$ – to obtain the alternate value of ϵ'. □

Tightness of Theorem 1.4 We observe that Theorem 1.4 is within a constant factor of being tight for large ϵ, as follows. It is well-known that no monotone function can compute the parity function to within more than $\frac{1}{2} - O(\frac{1}{\sqrt{n}})$, stated in Lemma 7.4. On the other hand, parity is easily computable by a small general circuit. Applying Theorem 1.4 to this circuit, with $\epsilon = \frac{1}{2}$, gives a monotone circuit computing parity to within $\frac{1}{2} - \frac{c}{\sqrt{n}}$ for some constant c, within a constant factor of the best possible. For completeness we provide a proof of Lemma 7.4.

LEMMA 7.4 *The parity function is $\delta = \frac{1}{2} - \frac{c}{\sqrt{n}}$ hard for both monotone and anti-monotone circuits of any size, for c an absolute constant.*

Proof. Let f be a monotone Boolean function. The idea of the proof is to progressively modify f so that it outputs 0 on strings with Hamming weight at most $\lfloor n/2 \rfloor - 1$ and outputs 1 on strings with Hamming weight at least $\lceil n/2 \rceil + 1$ without decreasing agreement with parity in the process. The result is a function f that has error exactly $\frac{1}{2}$ on strings outside of the middle one or two slices (middle one for even n, middle two for odd n). The middle one or two slices occupy a fraction $\Theta(\frac{1}{\sqrt{n}})$ of the inputs, so even if f were correct on all of these the total agreement with parity is $\frac{1}{2} + O(\frac{1}{\sqrt{n}})$. An iterative application of Claim 7.5 and a similar claim for levels above $\lceil n/2 \rceil + 1$ accomplishes the goal of setting f to 0 "on the bottom half" and to 1 "on the top half" of the n-cube. All that remains is to prove Claim 7.5.

CLAIM 7.5 *Let $0 \leq j < \lfloor n/2 \rfloor - 1$, and let f be a monotone Boolean function on n bits that is 0 on all x with $|x| < j$. There is a monotone function f' that is 0 on all x with $|x| \leq j$, agrees with f for all x with $|x| > j+1$, agrees with f for all x with $|x| = j+1$ if j is even, and is no farther from parity than f.*

First suppose j is even. Then set f' to agree with f on all x with $|x| \neq j$ and set $f'(x) = 0$ for all x with $|x| = j$. This brings f' closer to parity, and f' is monotone if f is.

Suppose j is odd. Claim 7.6 shows that by setting all bits in slices j and $j+1$ to 0, we cannot go farther away from parity. This satisfies Claim 7.5, so all that remains is to prove Claim 7.6.

CLAIM 7.6 *Let $0 \leq j < \lfloor n/2 \rfloor - 1$, and let f be a monotone Boolean function on n bits. f takes the value 1 on at least as many inputs with Hamming weight $j+1$ as inputs with Hamming weight j.*

Claim 7.6 states that when looking at the "bottom half" of a monotone function, the number of 1's on each slice is non-decreasing. Claim 7.6 can be proved as a corollary to various results in combinatorics. We will the fact that the Boolean n-cube can be partitioned into disjoint symmetric chains. To state this result, we view an n-bit Boolean string as the set S of the positions in the string equal to 1. Then $S \subseteq [n]$ and if $|x| = k$ then $|S| = k$. A "chain" in the Boolean cube is a sequence $S_1, S_2, ..., S_\ell$ such that for each $1 \leq i < \ell$, $S_i \subset S_{i+1}$. In other words, the input $x_{S_{i+1}}$ associated with S_{i+1} is obtained from the input x_{S_i} associated with S_i by flipping one or more 0's to 1's. A chain is symmetric if $|S_1| + |S_\ell| = n$, which implies that $|S_{i+1}| = |S_i| + 1$ for each i.

The proof that the Boolean n-cube can be partitioned into symmetric chains is by induction. To go from n bits to $n+1$ bits, replace each chain $S_1, S_2, ..., S_\ell$ from the symmetric chain decomposition of the n-cube with the two chains $S_1, S_2, ..., S_\ell, S_\ell \cup \{n+1\}$ and $S_1 \cup \{n+1\}, S_2 \cup \{n+1\}, ..., S_{\ell-1} \cup \{n+1\}$, where the second chain is only added if $\ell \geq 2$.

Now let us see how to use this result to prove Claim 7.6. Because $j < \lfloor n/2 \rfloor - 1$, there are at least as many strings on the $(j+1)$-st slice as the j-th slice. The symmetric chain

decomposition of the n-cube gives a matching between the elements of the j-th slice and a subset of size $\binom{n}{j}$ of elements of the $(j+1)$-st slice. For each element x_S in the j-th slice, there is a corresponding element $x_{S'}$ in the $(j+1)$-st slice with $S \subset S'$ so that by monotonicity of f, if $f(x_S) = 1$ then $f(x_{S'}) = 1$ as well. Claim 7.6 follows because the symmetric chains are disjoint, meaning each input on the j-th slice taking the value 1 is matched with a different input on the $(j+1)$-st slice taking the the value 1. □

7.2.1 Monotone Hard Functions

As discussed in Section 1.3, results from learning theory tell us that no monotone function can be more than $(\frac{1}{2} - \Omega(\frac{\log n}{\sqrt{n}}))$-hard for linear-size general circuits or $O(n \log n)$ size monotone circuits. The known circuit lower bounds proofs for monotone circuits give hardness that is little better than worst-case hardness (they give hardness $2^{n^\alpha - n}$ for some constant $0 < \alpha < 1$). A natural question then is how close in hardness a monotone function can come to the $\frac{1}{2} - \Omega(\frac{\log n}{\sqrt{n}})$ barrier. In this sub-section, we show that there do exist monotone functions whose hardness approaches this barrier.

[Weg84] observed that there exist monotone languages that are *worst-case hard* for general circuits with $\Theta(2^n/n^{3/2})$ gates; this follows from the fact that for small enough $s = O(2^n/n^{3/2})$ the number of monotone slice functions for the middle slice is larger than the number of circuits with s gates. [ACR97] used more refined probabilistic techniques to prove a result which implies the existence of a mildly average-case hard monotone function. They prove an asymptotic characterization of how inapproximable a function can be on any subset of its inputs; in particular there exist functions that are hard to approximate on their middle slice. Thus there exist monotone slice functions which are hard to approximate on the middle slice; the particular parameters are stated in Lemma 7.7.

LEMMA 7.7 (FOLLOWS FROM [ACR97]) *There exist constants $c_1, c_2 > 0$ and a balanced monotone function f such that for sufficiently large n, no circuit with at most $s = \frac{c_1 2^n}{n^{3/2}}$ gates computes f to within $1 - \delta(n)$ at length n for $\delta(n) = \frac{c_2}{\sqrt{n}}$.*

[BT96] have shown that for any monotone function f and any $\delta(n) > 0$, there is a Boolean circuit with $2^{O((1/\delta(n))\cdot\sqrt{n}\log(\sqrt{n}\cdot\delta(n)))}$ gates that computes f to within $\delta(n)$. Thus the hard function of Lemma 7.7 has hardness within a constant factor of the best possible for circuits with $s = 2^{\Theta(n)}$ gates.

The results proved in [ACR97] are more general and the techniques more involved than needed for Lemma 7.7. For completeness, we provide a simple proof of Lemma 7.7.

Proof. We show that there exists a monotone slice function for the middle slice which has the stated hardness. This can be shown directly using a probabilistic argument by comparing the number of monotone slice functions for the middle slice with the number of functions within a certain distance on the middle slice of size s circuits.

Alternatively, we can take a general n-bit function h with high average-case hardness and create a monotone function f on m bits by letting it be a monotone slice function resulting from embedding the truth table of h within the m-cube. Let h be a function that is $\frac{1}{4}$-hard for circuits of size $\frac{c2^n}{n}$ for a positive constant c. Such an h can be proved by a probabilistic argument [Pip76]. We let f be defined by choosing the smallest m such that $\binom{m}{\lfloor m/2 \rfloor} \geq 2^n$ and using the embedding of Lemma 7.3 of h into the middle slice of f. Because $\binom{m}{\lfloor m/2 \rfloor}$ grows by at most a factor of two with m and is $\Theta(\frac{2^m}{\sqrt{m}})$, we have that the embedding uses at least $1/2$ of the middle slice and $m \leq n + \log n$ for sufficiently large n. We can ensure f is balanced by setting bits appropriately in the middle slice and neighboring slices that are not used in the embedding. Given a circuit of size s that computes f on at least a fraction $\frac{3}{4}$ of the strings used in the embedding of h, we would get a circuit of size $s + \text{poly}(n)$ that computes h to within $\frac{1}{4}$. For a suitable constant c_1 depending on c, we get a contradiction to the hardness of h if $s \leq \frac{c_1 2^m}{m^{3/2}}$ as follows. We can upper-bound s by

$$O(\frac{2^m}{m^{3/2}}) \leq O(\frac{2^n\sqrt{m}}{m^{3/2}}) \leq O(\frac{2^n}{n})$$

where the constant in the final big-O decreases towards 0 as c_1 decreases towards 0. The first inequality follows because m was chosen so that $\binom{m}{\lfloor m/2 \rfloor}$ – which is $\Theta(\frac{2^m}{\sqrt{m}})$ – is $\leq 2 \cdot 2^n$. Then the circuit for computing h to within $\frac{1}{4}$ has size $O(\frac{2^n}{n}) + \text{poly}(n) = O(\frac{2^n}{n})$ where the

constant in the big-O decreases with c_1, and we get a contradiction to the hardness of h when this is less than c. Finally, the hardness δ for f as a function of m is at least a fraction $\frac{1}{4}$ of the strings used in the embedding, which is $\Omega(\frac{1}{\sqrt{m}})$ because the embedding uses at least 1/2 of the middle slice. □

Given a mildly hard function, the XOR lemma can often be used to produce a function that is more inapproximable, but applying the XOR lemma to the hard function of Lemma 7.7 would produce a function with amplified hardness *that is no longer monotone*. We can instead use a hardness amplification procedure that preserves monotonicity. O'Donnell [O'D04] developed a hardness amplification procedure tailored for use in the NP setting that has the property we need – given a mildly hard monotone function, the procedure produces a function with increased hardness that remains monotone. We have stated this result as Theorem 2.10. By applying Theorem 2.10 to the hard function of Lemma 7.7, we obtain the following.

THEOREM 7.8 *For every constant $\eta > 0$ there exists a constant $c(\eta) > 0$ and a monotone function f such that for sufficiently large n, f at length n is $\delta = \frac{1}{2} - \frac{1}{n^{1/2-\eta}}$ hard for circuits with $s = 2^{n^{c(\eta)}}$ gates.*

Note that the hardness $\frac{1}{2} - \frac{1}{n^{1/2-\eta}}$ comes close to the barrier of $\frac{1}{2} - \Omega(\frac{\log n}{\sqrt{n}})$ discussed earlier. We also point out that the hard function of Theorem 7.8 is computable in $\mathrm{E}^{\Sigma_2^p}$, exponential time with an oracle to the second level of the polynomial hierarchy, using the same techniques of [Kan82] that show $\mathrm{E}^{\Sigma_2^p}$ contains a language with maximal circuit complexity (as observed for example in [MVW99]).

7.3 Pseudorandom Generators

In the last section we showed that any function that is average-case hard for monotone circuits is average-case hard for general circuits with somewhat weaker parameters. These results are motivated by the possibility of using an average-case hard function to build a pseudorandom generator suitable for derandomizing randomized monotone circuits. In this

section, we show that any method for constructing a pseudorandom generator secure against randomized monotone circuits also must give a pseudorandom generator secure against general circuits with somewhat weaker parameters.

Reduction to Monotone Adversaries The following theorem states that a circuit that distinguishes a distribution from uniform can be converted into a monotone distinguisher with somewhat weaker parameters. Stated in the contrapositive, if G is an ϵ'-pseudorandom generator against size s monotone circuits, then G is an ϵ-pseudorandom generator against general circuits of size $\frac{s}{2} - O(n \log^2 n)$.

We point out that a slightly weaker version of Theorem 1.5 was independently discovered by Karakostas [Kar09], namely with $\epsilon' = \frac{\epsilon}{2(n+1)}$.

THEOREM 1.5 *Let C be a circuit of size s that ϵ-distinguishes some distribution $\mathcal{D}$ from uniform. Then there is a monotone circuit C' of size $2s + O(n \log^2 n)$ that ϵ'-distinguishes $\mathcal{D}$ from uniform for $\epsilon' = \max(\frac{\epsilon}{2(n+1)}, \frac{c}{\sqrt{n \log(1/\epsilon)}})$ for $c > 0$ an absolute constant.*

Proof. The proof is essentially identical to that of Theorem 1.4 except in the setting of distinguishers rather than computing a Boolean function. The main idea is to find a slice i on which C ϵ'-distinguishes and let C' compute the monotone slice function agreeing with C on that slice. A simple calculation then shows that either C' or the threshold function outputting 1 iff $|x| > i$ distinguishes with probability ϵ' over all inputs.

Let C be an ϵ-distinguisher of size s for $\mathcal{D}$. By definition, either $\Pr_{X \leftarrow U_n}[C(X) = 1] - \Pr_{Y \leftarrow \mathcal{D}}[C(Y) = 1] \geq \epsilon$ or $\Pr_{Y \leftarrow \mathcal{D}}[C(Y) = 1] - \Pr_{X \leftarrow U_n}[C(X) = 1] \geq \epsilon$. Without loss of generality, we assume the former. By breaking these probabilities into disjoint events, we have that

$$\sum_{i=0}^{n} (\Pr_{X \leftarrow U_n}[C(X) = 1 \text{ and } |X| = i] - \Pr_{Y \leftarrow \mathcal{D}}[C(Y) = 1 \text{ and } |Y| = i]) \geq \epsilon.$$

By an averaging argument, there exists an index i such that $\Pr_{X \leftarrow U_n}[C(X) = 1 \text{ and } |X| = i] - \Pr_{Y \leftarrow \mathcal{D}}[C(Y) = 1 \text{ and } |Y| = i] \geq \frac{\epsilon}{n+1}$. By Theorem 7.1, there is a monotone circuit C_{mon}

that agrees with C on the i-th slice and uses at most $2s + O(n \log^2 n)$ gates. The overall distinguishing probability of C_{mon} can be expressed as

$$\begin{aligned}&(\Pr_{X \leftarrow U_n}[C_{mon} = 1 \text{ and } |X| = i] - \Pr_{Y \leftarrow \mathcal{D}}[C_{mon} = 1 \text{ and } |Y| = i])\\ &+(\Pr_{X \leftarrow U_n}[C_{mon} = 1 \text{ and } |X| > i] - \Pr_{Y \leftarrow \mathcal{D}}[C_{mon} = 1 \text{ and } |Y| > i])\\ &+(\Pr_{X \leftarrow U_n}[C_{mon} = 1 \text{ and } |X| < i] - \Pr_{Y \leftarrow \mathcal{D}}[C_{mon} = 1 \text{ and } |Y| < i]).\end{aligned}$$

The last term is 0 because C_{mon} outputs 0 on strings of weight less than i. The middle term is $\Pr_{X \leftarrow U_n}[|X| > i] - \Pr_{Y \leftarrow \mathcal{D}}[|Y| > i]$ because C_{mon} outputs 1 on strings of weight greater than i. If the absolute value of this term is greater than $\frac{\epsilon}{2(n+1)}$, then the threshold function that outputs 1 iff $|X| > i$ – computable by $O(n \log n)$ size monotone circuits [AKS83] – is an $\frac{\epsilon}{2(n+1)}$-distinguisher. Otherwise, the distinguishing probability of C_{mon} is at least

$$(\Pr_{X \leftarrow U_n}[C_{mon}(X) = 1 \text{ and } |X| = i] - \Pr_{Y \leftarrow \mathcal{D}}[C_{mon}(Y) = 1 \text{ and } |Y| = i]) - \frac{\epsilon}{2(n+1)} \geq \frac{\epsilon}{2(n+1)}.$$

The alternate value for ϵ' comes by only considering $\Theta(\sqrt{n \log(n/\epsilon)})$ layers around the middle, which together contain a fraction $1 - \frac{\epsilon}{2}$ of all strings. These layers collectively distinguish with $\frac{\epsilon}{2}$ advantage, so one of them must distinguish with $\Omega(\frac{\epsilon}{\sqrt{\log(n/\epsilon)}})$ advantage. The analysis for this case is the same as for this case of Theorem 1.4. □

Remark: In the setting of general circuits, it is known that the existence of explicit pseudorandom generators is equivalent to the existence of explicit functions that are hard on average. A natural question is whether this remains true in the setting of monotone circuits; if so then Theorem 1.5 for the case of pseudorandom distributions would follow as a corollary to Theorem 1.4. A simple argument shows that the language L defined as the set of strings output by a pseudorandom generator secure against certain adversaries must be *worst-case* hard for those same adversaries. The argument carries through for monotone circuits, but worst-case hardness is not enough to apply Theorem 1.4. For general circuits and pseudorandom generators computable in exponential time in the seed length, [NW94] observe that L must be *average-case* hard by appealing to the known worst-case to average-case reductions for languages computable in exponential time. These reductions do not seem to preserve

monotonicity so do not prove a connection between pseudorandom generators secure against monotone circuits and average-case hard functions for monotone circuits.

Tightness of Theorem 1.5 One question is whether the parameters in Theorem 1.5 can be tightened further. We have mentioned in Lemma 7.4 that the parity function is $\frac{1}{2} - \Omega(\frac{1}{\sqrt{n}})$ hard for monotone circuits. A standard argument shows that a hard function yields a pseudorandom generator with 1 bit stretch by outputting the seed along with the value of the hard function on the seed. In the following theorem, we show that this argument carries through for monotone circuits with the parity function, denoted $\oplus$, as the hard function. Theorem 7.9 shows that Theorem 1.5 is tight to within a constant factor: $G^{\oplus}$ is easily distinguishable with $\epsilon = \frac{1}{2}$ by a small general circuit, and applying Theorem 1.5 to this circuit produces a monotone circuit that $\frac{\gamma}{\sqrt{n}}$-distinguishes $G^{\oplus}$ from uniform for some constant γ – a monotone distinguisher within a constant factor of optimal.

THEOREM 7.9 *Define a generator $G^{\oplus}$ as follows: $G^{\oplus}(x) = (x, \oplus(x))$. Then $G^{\oplus} : \{0,1\}^n \to \{0,1\}^{n+1}$ is a $\epsilon = \frac{c}{n^{1/2}}$ pseudorandom generator secure against monotone and anti-monotone circuits of any size, for c an absolute constant.*

Proof. We follow the standard proof from the general setting and keep track of monotonicity to verify the final circuit is monotone or anti-monotone. We assume a monotone or anti-monotone circuit C that ϵ-distinguishes the output of $G^{\oplus}$ from uniform. We would like to use C to compute parity on some n-bit string x. If C were a perfect distinguisher then for any x, $C(x, \oplus(x)) = 1$ and $C(x, \overline{\oplus(x)}) = 0$. C is not a perfect distinguisher, but we treat it as if it were and analyze the probability that we are correct. Namely, we choose a random bit b and query the value $C(x, b)$. If $C(x, b) = 1$ we assume $\oplus(x) = b$; if $C(x, b) = 0$ we assume $\oplus(x) = \overline{b}$. A random bit b is equal to $\oplus(x)$ with probability $\frac{1}{2}$ and is equal to $\overline{\oplus x}$ with probability $\frac{1}{2}$, so the probability we output the correct value for $\oplus(x)$ is

$$\frac{1}{2}(\Pr[C(x, \oplus(x)) = 1] + \Pr[C(x, \overline{\oplus(x)}) = 0]). \tag{7.1}$$

We use the fact that C is an ϵ-distinguisher to lower bound (7.1). We have that

$$|\Pr_{X\in U_n}[C(X,\oplus(X))=1]-\Pr_{X\in U_n,\beta\in U_1}[C(X,\beta)=1]|\geq\epsilon.$$

By expressing the second term as a sum depending on whether β is $\oplus(X)$ or $\overline{\oplus(X)}$, we have

$$\begin{gathered}\tfrac{1}{2}|\Pr_{X\in U_n}[C(X,\oplus(X))=1]-\Pr_{X\in U_n}[C(X,\overline{\oplus(X)})=1]|\geq\epsilon,\text{ and therefore}\\ \tfrac{1}{2}|\Pr_{X\in U_n}[C(X,\oplus(x))=1]+\Pr_{X\in U_n}[C(X,\overline{\oplus(X)})=0]-1|\geq\epsilon.\end{gathered}$$

If the sign on the absolute value is positive, we have that (7.1) is at least $\frac{1}{2}+\epsilon$. Otherwise we have that (7.1) is at most $\frac{1}{2}-\epsilon$; in that case the negation of our strategy is correct with probability at least $\frac{1}{2}+\epsilon$.

Let us verify that this strategy produces a monotone or anti-monotone circuit. First, there is a value for b that preserves the probability of success, and we can fix this value into the circuit. If b is fixed to 1, then our strategy outputs $C(x,1)$; if b is set to 0, our strategy outputs $\overline{C(x,0)}$. Due to the sign on the absolute value, we may need to place an additional negation at the top of the final circuit. We have that if C is an ϵ-distinguisher for $G^{\oplus}$ then one of $C(x,1),\overline{C(x,1)},\overline{C(x,0)},C(x,0)$ computes parity to within $\frac{1}{2}-\epsilon$. If C is monotone or anti-monotone, then so are each of these circuits, and for $\epsilon\geq c\frac{1}{\sqrt{n}}$ for the constant c of Lemma 7.4 we have a contradiction. □

Remark: A standard modification to the construction is to apply the hard function – parity – on disjoint subsets of the seed to produce more output bits. If we define $G_k^{\oplus}(x^1,...,x^k)=(x^1,\oplus(x^1),x^2,\oplus(x^2),...,x^k,\oplus(x^k))$, with $|x^i|=\lfloor n/k\rfloor$ for all i, then the proof can be modified to show that $G_k^{\oplus}:\{0,1\}^n\to\{0,1\}^{n+k}$ is a $\epsilon=\Theta(\frac{1}{(\lfloor n/k\rfloor)^{1/2}})$-pseudorandom generator secure against monotone circuits of any size.

7.4 Derandomization

In the last two sections, we showed that average-case hard functions for monotone circuits are also average-case hard for general circuits with somewhat weaker parameters, and pseudorandom generators secure against monotone circuits are also secure against general

circuits with somewhat weaker parameters. Constructing pseudorandom generators is one particular method to obtain derandomization of randomized monotone circuits, and average-case hard functions are one ingredient that can be used to build pseudorandom generators. In this section, we show that *any* method of derandomizing monotone randomized circuits can also be used to derandomize general non-monotone circuits.

Monotone Randomized Computations One natural definition for the class of monotone randomized computations is the set of BPP languages that are also monotone. But it is easy to give a reduction from any BPP language L to this class by simply embedding the truth table of f within the middle slice of a monotone function using Lemma 7.3.

We instead consider another natural definition of monotone randomized computations, namely the set of languages that can be solved by uniform bounded-error monotone randomized circuits. The uniformity requirement is that on input 1^n, the circuit can be output in $\text{poly}(n)$ time. The resulting circuit should be monotone in both the input and random bits and should have bounded error on every input. In Theorem 1.6, we show an efficient reduction from any BPP language L to languages solvable by this weaker model of randomized monotone computations. Thus if these computations can be solved in P, then all of BPP is in P.

We point out that there exist monotone languages in BPP that are not computable by uniform bounded-error monotone randomized circuits. This follows from two facts. First, randomness can be removed from bounded-error monotone randomized by reducing the error to be less than 2^{-n} (which only uses majority and thus preserves monotonicity) and then fixing a random string that is correct for all inputs; thus bounded-error randomized monotone circuits can be simulated efficiently by non-uniform deterministic monotone circuits. Second, [Raz] and [Tar87] demonstrate monotone languages in P, and thus also BPP, that require non-uniform monotone circuits of super-polynomial size (exponential size for the result of [Tar87]).

THEOREM 1.6 *Let L be any language computable by polynomial-time bounded-error randomized machines. There is a language L_{mon} computable by uniform monotone bounded-error polynomial-size randomized circuits such that L poly-time mapping reduces to L_{mon}. In particular, if $L_{mon} \in \mathrm{P}$ then $L \in \mathrm{P}$.*

Proof. Let M be a bounded-error randomized machine running in time n^k computing a BPP language L, for some constant k. The basic idea is to take the function computed by the deterministic machine underlying M and embed this within a monotone slice function. Viewing this monotone slice function as a randomized monotone circuit, we must ensure the following.

(i) The circuit has bounded error on all inputs.

(ii) L many-one reduces to the language computed by the circuit.

Let $f : \{0,1\}^n \times \{0,1\}^{n^k} \to \{0,1\}$ be the function computed by M given an n-bit input x and random string r of length n^k. To produce a randomized monotone circuit, we separately embed both the input and the random string into the middle slice of larger Boolean cubes. To embed the input we can use the simple embedding associating x with the $2n$-bit string $(x, \overline{x})$. We must take more care with the embedding of the random bits because the circuit must have error bounded away from one half on each input. We achieve this by using the embedding of Lemma 7.3 so that the strings involved in the embedding occupy a constant fraction of the middle slice and thus a $1/\operatorname{poly}$ fraction of all random strings.

Now we carry out the above outline. Let m be the smallest even integer such that $\binom{m}{\lfloor m/2 \rfloor} \geq 2^{n^k}$. Because $\binom{m}{\lfloor m/2 \rfloor}$ grows by less than a factor of two for each increment of m, we also have that $\binom{m}{\lfloor m/2 \rfloor} \leq 4 \cdot 2^{n^k}$. We define a randomized monotone circuit in terms of the function f_{mon} that it computes. The function takes an input x' of $2n$ bits and a random string r' of m bits and behaves as follows.

1. *Slice function of x'*

   If $|x'| > n$, set $f_{mon}(x', r') = 1$. If $|x'| < n$, set $f_{mon}(x', r') = 0$.

2. *Slice function of r' for x' on middle slice*

   If $|x'| = n$ and $|r'| > m/2$, set $f_{mon}(x', r') = 1$.

   If $|x'| = n$ and $|r'| < m/2$, set $f_{mon}(x', r') = 0$.

3. *Embed f within middle slice of f_{mon}*

   If $x' = (x, \overline{x})$ for some x of length n and $|r'| = m/2$, do the following. If r' is among the 2^{n^k} strings matched with $\{0,1\}^{n^k}$ by the embedding of Lemma 7.3, let r be the associated value and set $f_{mon}(x', r') = f(x, r)$. For r' that do not have a match within $\{0,1\}^{n^k}$ (because r' is not among the 2^{n^k} "smallest" strings in the middle slice of the m-cube), set $f_{mon}(x', r')$ to 0 on half of these and 1 on half.

4. *Other x' on the middle slice*

   If $|x'| = n$, x' is not of the form $(x, \overline{x})$, and $|r'| = m/2$, set $f_{mon}(x', r') = 0$.

For x' of the form $(x, \overline{x})$, this construction ensures $\Pr_{r'}[f_{mon}(x', r') = 1] = \frac{1}{2} \cdot (1 - \rho) + \rho \cdot \Pr_r[f(x, r) = 1]$, where ρ is the fraction of strings used by the embedding of n^k-bit random strings into the middle slice of the m-cube. As stated above, m was chosen so that $\rho = \Theta(\frac{1}{\sqrt{m}})$ and $m = n + O(\log n)$. Thus the majority value of $f_{mon}(x', \cdot)$ agrees with the majority value of $f(x, \cdot)$, and the error is bounded away from one half by $1/\operatorname{poly}$.

For x' with $|x'| = n$ that is not of the form $(x, \overline{x})$, the last step ensures error bounded away from one half as well – for such x', $\Pr_{r'}[f_{mon}(x', r') = 0] \geq \frac{1}{2} + \frac{1}{\operatorname{poly}}$. For x' with $|x'| \neq n$, $f_{mon}(x', \cdot)$ is either the constant 0 or constant 1 by the first step.

Let us see that f_{mon} can be computed by a uniform polynomial-size circuit. Let C be a uniform polynomial-size circuit for f_{mon}; we wish to remove the negations from this circuit without increasing the size too much. As in the proof of Theorem 7.1, we first push the negations to the inputs, at most doubling the circuit size. Because f_{mon} is a monotone slice function of x', as noted in the proof of Theorem 7.1, we can replace the negations of those variables by a monotone circuit of size $O(n \log^2 n)$. For x' on the non-trivial slice of f_{mon}, f_{mon} is a monotone slice function of r', so we can replace the negations of those variables by a

monotone circuit of size $O(m \log^2 m)$. We conclude that f_{mon} has a uniform polynomial-size circuit.

To satisfy (i) and (ii), it only remains to lower the error from $\frac{1}{2} - 1/\operatorname{poly}$ to $\frac{1}{3}$. We can reduce the error to $\frac{1}{3}$ by using standard error reduction consisting of taking multiple trials and majority voting. This can be implemented by a uniform monotone circuit of polynomial size [AKS83]. The result is a uniform polynomial-size monotone circuit C_{mon} that has bounded error on every input and such that $M(x) = C_{mon}((x, \overline{x}))$, completing the proof. □

LIST OF REFERENCES

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[ACR97] Alexander E. Andreev, Andrea E. F. Clementi, and José D. P. Rolim. Optimal bounds for the approximation of boolean functions and some applications. *Theoretical Computer Science*, 180(1-2):243–268, 1997.

[AKL+79] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, Laszlo Lovasz, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 218–223, 1979.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1983.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxen. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verificiation and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[AM10] Scott Aaronson and Dieter van Melkebeek. A note on circuit lower bounds from derandomization. 2010. In preparation.

[AW09] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM Transactions on Computation Theory*, 1(1):1–54, 2009.

[Bab85] Laszlo Babai. Trading group theory for randomness. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 421–429, 1985.

[Bar02] Boaz Barak. A probabilistic-time hierarchy theorem for slightly non-uniform algorithms. In José D. P. Rolim and Salil P. Vadhan, editors, *Proceedings of the International Workshop on Randomization and Computation (RANDOM)*, volume 2483 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Ber82] S.J. Berkowitz. On some relationships between monotone and non-monotone circuit complexity. Technical report, University of Toronto, 1982.

[BFL91] Lásló Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1(1):3–40, 1991.

[BFNW93] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3:307–318, 1993.

[BJLR91] Gerhard Buntrock, Birgit Jenner, Klaus-Jorn Lange, and Peter Rossmanith. Unambiguity and fewness for logarithmic space. In Lothar Budach, editor, *Proceedings of the International Conference on Fundamentals of Computation Theory*, volume 529 of *Lecture Notes in Computer Science*, pages 168–179. Springer-Verlag, 1991.

[BNS92] László Babai, Noam Nisan, and Márió Szegedy. Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs. *Journal of Computer and System Sciences*, 45(2):204–232, 1992.

[BS91] Ravi B. Boppana and Michael Sipser. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*, chapter The Complexity of Finite Functions, pages 757–804. MIT Press, 1991.

[BT96] Nader H. Bshouty and Christino Tamon. On the Fourier spectrum of monotone functions. *Journal of the ACM*, 43(4):747–770, 1996.

[BTV09] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. *ACM Transactions on Computation Theory*, 1(1), 2009.

[Con93] Anne Condon. The complexity of space bounded interactive proof systems. In Steven Homer, Uwe Schöning, and Klaus Ambos-Spies, editors, *Complexity Theory: Current Research*, pages 147–190. Cambridge University Press, 1993.

[Coo73] Stephen Cook. A hierarchy theorem for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7:343–353, 1973.

[CW89] Aviad Cohen and Avi Wigderson. Dispersers, deterministic amplification, and weak random sources (extended abstract). In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 14–19, 1989.

[Eng97] Konrad Engel. *Sperner Theory*, volume 65 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1997.

[FL93] Lance Fortnow and Carsten Lund. Interactive proof systems and alternating time-space complexity. *Theoretical Computer Science*, 113(1):55–73, 1993.

[FS04] Lance Fortnow and Rahul Santhanam. Hierarchy theorems for probabilistic polynomial time. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 316–324, 2004.

[FST05] Lance Fortnow, Rahul Santhanam, and Luca Trevisan. Hierarchies for semantic classes. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 348–355, 2005.

[GNW95] Oded Goldreich, Noam Nisan, and Avi Wigderson. On Yao's XOR-lemma. *Electronic Colloquium on Computational Complexity (ECCC)*, 2(50), 1995.

[Gol08] Oded Goldreich. *Complexity Theory: A Conceptual Perspective*. Cambridge University Press, 2008.

[GST04] Oded Goldreich, Madhu Sudan, and Luca Trevisan. From logarithmic advice to singe-bit advice. Technical Report TR-04-093, Electronic Colloquium on Computational Complexity (ECCC), 2004.

[GSTS03] Dan Gutfreund, Ronen Shaltiel, and Amnon Ta-Shma. Uniform hardness versus randomness tradeoffs for Arthur-Merlin games. *Computational Complexity*, 12(3–4):85–130, 2003.

[GW00] Oded Goldreich and Avi Wigderson. On pseudorandomness with respect to deterministic observers. In Carleton Scientific, editor, *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 77–84, 2000.

[GW02] Oded Goldreich and Avi Wigderson. Derandomization that is rarely wrong from short advice that is typically good. In *Proceedings of the International Workshop on Randomization and Computation (RANDOM)*, pages 209–223, 2002.

[Hås87] Johan Håstad. *Computational Limitations of Small-Depth Circuits*. MIT Press, Cambridge, MA, USA, 1987.

[HT03] Christopher M. Homan and Mayur Thakur. One-way permutations and self-witnessing languages. *Journal of Computer and System Sciences*, 67(3):608–622, 2003.

[IKW02] Russell Impagliazzo, Valentine Kabanets, and Avi Wigderson. In search of an easy witness: exponential time vs. probabilistic polynomial time. *Journal of Computer and System Sciences*, 65(4):672–694, 2002.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[Imp95] Russell Impagliazzo. Hard-core distributions for somewhat hard problems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 538–545, 1995.

[IW01] Russell Impagliazzo and Avi Wigderson. Randomness vs time: Derandomization under a uniform assumption. *Journal of Computer and System Sciences*, 63(4):672–688, 2001.

[IZ89] Russell Impagliazzo and David Zuckerman. How to recycle random bits. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 248–253, 1989.

[Kab01] Valentine Kabanets. Easiness assumptions and hardness tests: Trading time for zero error. *Journal of Computer and System Sciences*, 63(2):236–252, 2001.

[Kan82] Ravi Kannan. Circuit-size lower bounds and nonreducibility to sparse sets. *Information and Control*, 55(1):40–56, 1982.

[Kar09] George Karakostas. General pseudo-random generators from weaker models of computation. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC)*, pages 1094–1103, 2009.

[KI04] Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1/2):1–46, 2004.

[KL82] Richard Karp and Richard Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28(2):191–209, 1982.

[KM02] Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM Journal on Computing*, 31(5):1501–1526, 2002.

[KM08] Jeff Kinne and Dieter van Melkebeek. Space hierarchy results for randomized models. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 433–444, 2008.

[KM10] Jeff Kinne and Dieter van Melkebeek. Space hierarchy results for randomized and other semantic models. *Computational Complexity*, 2010. In press.

[KMS09] Jeff Kinne, Dieter van Melkebeek, and Ronen Shaltiel. Pseudorandom generators and typically-correct derandomization. In *Proceedings of the International Workshop on Randomization and Computation (RANDOM)*, pages 574–587, 2009.

[Kor03] A. D. Korshunov. Monotone boolean functions. *Russian Math. Surveys*, 58(5):929–1001, 2003.

[KV87] Marek Karpinski and Rutger Verbeek. Randomness, provability, and the separation of Monte Carlo time and space. In Egon Börger, editor, *Computation Theory and Logic*, volume 270 of *Lecture Notes in Computer Science*, pages 189–207. Springer-Verlag, 1987.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, (13):300–317, 1976.

[Mil01] Peter Bro Miltersen. Derandomizing complexity classes. In *Handbook of Randomized Computing*, pages 843–941. Kluwer Academic Publishers, 2001.

[MP07] Dieter van Melkebeek and Konstantin Pervyshev. A generic time hierarchy for semantic models with one bit of advice. *Computational Complexity*, 16:139–179, 2007.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MS05] Dieter van Melkebeek and Rahul Santhanam. Holographic proofs and derandmization. *SIAM Journal on Computing*, 35(1):59–90, 2005.

[MV05] Peter Bro Miltersen and N. V. Vinodchandran. Derandomizing Arthur-Merlin games using hitting sets. *Computational Complexity*, 14(3):256–279, 2005.

[MVW99] Peter Bro Miltersen, N. V. Vinodchandran, and Osamu Watanabe. Superpolynomial versus half-exponential circuit size in the exponential hierarchy. In *COCOON*, pages 210–220, 1999.

[New91] Ilan Newman. Private vs. common random bits in communication complexity. *Information Processing Letters*, 39(2):67–71, 1991.

[Nis91] Noam Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, 11(1):63–70, 1991.

[Nis92] Noam Nisan. RL $\subseteq$ SC. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Theory of Computing,* Victoria, British Columbia, Canada, pages 619–623, 1992.

[Nis93] Noam Nisan. On read-once vs. multiple access to randomness in logspace. *Theoretical Computer Science*, 107(1):135–144, 1993.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49(2):149–167, 1994.

[O'D04] Ryan O'Donnell. Hardness amplification within NP. *Journal of Computer and System Sciences*, 69(1):68–94, 2004.

[OW09] Ryan O'Donnell and Karl Wimmer. KKL, Kruskal-Katona, and monotone nets. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2009.

[Pip76] Nicholas Pippenger. Information theory and the complexity of boolean functions. *Theory of Computing Systems*, 10(1):129–167, 1976.

[PTV10] Aduri Pavan, Raghunath Tewari, and N. V. Vinodchandran. On the power of unambiguity in logspace. *CoRR*, abs/1001.2034, 2010.

[RA00] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118–1131, 2000.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, (12):128–138, 1980.

[Raz] Alexandor Razborov. A lower bound on the monotone network complexity of the logical permanent. *Matematicheskie Zametki*, 37(6):887–900 (in Russian). English transliation in *Mathematical Notes of the Academy of Sciences of the USSER* 37:6, 485–493.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4), 2008.

[Sak96] Michael Saks. Randomization and derandomization in space-bounded computation. In *Proceedings of the IEEE Conference on Computational Complexity*, pages 128–149, 1996.

[Sav70] W. Savitch. Relationship between nondeterministic and deterministic tape classes. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[Sax09] Nitin Saxena. Progress in polynomial identity testing. *Bulletin of the EATCS*, (99):49–79, 2009.

[SFM78] Joel Seiferas, Michael Fischer, and Albert Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25:146–167, 1978.

[Sha92] Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992.

[Sha09] Ronen Shaltiel. Weak derandomization of weak algorithms: explicit versions of Yao's lemma. In *Proceedings of the IEEE Conference on Computational Complexity*, 2009.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[SU05] Ronen Shaltiel and Christopher Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *Journal of the ACM*, 52(2):172–216, 2005.

[SU06] Ronen Shaltiel and Christopher Umans. Pseudorandomness for approximate counting and sampling. *Computational Complexity*, 15(4):298–341, 2006.

[SU07] Ronen Shaltiel and Christopher Umans. Low-end uniform hardness vs. randomness tradeoffs for AM. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 430–439, 2007.

[SZ99] Michael E. Saks and Shiyu Zhou. $\mathrm{BP_HSPACE(S)} \subseteq \mathrm{DSPACE}(\mathrm{S}^{3/2})$. *Journal of Computer and System Sciences*, 58(2):376–403, 1999.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[Tar87] Eva Tardos. The gap between monotone and non-monotone circuit complexity is exponential. *Combinatorica*, 7(4):141–142, 1987.

[Tod91] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[TV07] Luca Trevisan and Salil P. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. *Computational Complexity*, 16(4):331–364, 2007.

[Uma03] Christopher Umans. Pseudo-random generators for all hardnesses. *Journal of Computer and System Sciences*, 67(2):419–440, 2003.

[Val86] Leslie G. Valiant. Negation is powerless for boolean slice functions. *SIAM Journal on Computing*, 15(2):531–535, 1986.

[Vio05] Emanuele Viola. The complexity of constructing pseudorandom generators from hard functions. *Computational Complexity*, 13(3-4):147–188, 2005.

[Vio06] Emanuele Viola. Pseudorandom bits for constant-depth circuits with few arbitrary symmetric gates. *SIAM Journal on Computing*, 36(5):1387–1403, 2006.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity*. Springer-Verlag, 1999.

[Wat03] John Watrous. On the complexity of simulating space-bounded quantum computations. *Computational Complexity*, 12:48–84, 2003.

[Weg84] Ingo Wegener. On the complexity of slice functions. In *Mathematical Foundations of Computer Science*, pages 553–561, 1984.

[Wil85] Christopher B. Wilson. Relativized circuit complexity. *Journal of Computer and System Sciences*, 31(2):169–181, 1985.

[Žàk83] Stanislav Žàk. A Turing machine time hierarchy. *Theoretical Computer Science*, 26:327–333, 1983.

[Zan91] Viktoria Zanko. #P-completeness via many-one reductions. *International Journal of Foundations of Computer Science*, 2(1):77–82, 1991.

[Zim08] Marius Zimand. Exposure-resilient extractors and the derandomization of probabilistic sublinear time. *Computational Complexity*, 17(2):220–253, 2008.