SPACE HIERARCHY RESULTS FOR RANDOMIZED AND OTHER SEMANTIC MODELS

Jeff Kinne and Dieter van Melkebeek

Abstract. We prove space hierarchy and separation results for randomized and other semantic models of computation with advice where a machine is only required to behave appropriately when given the correct advice sequence. Previous works on hierarchy and separation theorems for such models focused on time as the resource. We obtain tighter results with space as the resource. Our main theorems deal with space-bounded randomized machines that always halt. Let $s(n)$ be any space-constructible monotone function that is $\Omega(\log n)$ and let $s'(n)$ be any function such that $s'(n) = \omega(s(n + as(n)))$ 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.

There exists a language computable by *zero-sided* error randomized machines in space $s'(n)$ with 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.

If, in addition, $s(n) = O(n)$ then the condition on s' above can be relaxed to $s'(n) = \omega(s(n + 1))$. This yields tight space hierarchies for typical space bounds $s(n)$ that are at most linear.

We also obtain results that apply to generic semantic models of computation.

Keywords. Space Hierarchy, Randomized Computations, Computations with Advice, Promise Classes, Semantic Models

Subject classification. 68Q15, 68Q10, 03D15, 68Q25

1. Introduction

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 nondeterministic machines is more complicated because a nondeterministic machine cannot easily complement another nondeterministic machine (unless NP=coNP). A variety of techniques can be used to overcome this difficulty, including translation arguments and delayed diagonalization (Cook 1973; Seiferas *et al.* 1978; Žàk 1983). In fact, 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.

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 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 (Shamir 1992), MIP = NEXP (Babai *et al.* 1991), BP.$\oplus$P = Σ_2.$\oplus$P (Toda 1991), and PCP($\log n$,1) = NP (Arora *et al.* 1998). A recent line of research (Barak 2002; Fortnow & Santhanam 2004; Fortnow *et al.* 2005; Goldreich *et al.* 2004; Van Melkebeek & Pervyshev 2007) has provided progress toward proving time hierarchy results for non-syntactic models, including two-sided 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, which the diagonalizing machine uses to avoid simulating a machine on an input for which that machine breaks the promise.

As opposed to the setting of time, fairly good space hierarchy theorems are known for certain non-syntactic models. In fact, 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 (Karpinski & Verbeek 1987), 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 deterministic space hierarchy theorem because randomized two-sided error computations that run in space $s(n)$ can be simulated deterministically in space $(s(n))^{1.5}$ (Saks & Zhou 1999). The same argument applies to non-syntactic 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, unambiguous machines, etc.

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 (Immerman 1988; Szelepcsényi 1988), such tight space hierarchies follow by straightforward diagonalization. For generic syntactic models, tight space hierarchies follow using the same techniques as in the time-bounded setting. Those techniques all require the existence of an efficient universal machine, which presupposes the model to be syntactic. For that reason they fail for non-syntactic models of computation such as bounded-error randomized machines.

In this paper we obtain space hierarchy results that are tight with respect to space by adapting to the space-bounded setting techniques that have been developed for proving hierarchy results for semantic models in the time-bounded setting.

1.1. Our Results. 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 (Barak 2002; Fortnow & Santhanam 2004; Fortnow *et al.* 2005; Goldreich *et al.* 2004; Van Melkebeek & Pervyshev 2007), 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).

1.1.1. Randomized Models. 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 1.1. *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 1.1 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, the condition on $s'(n)$ in the above can be relaxed to $s'(n) = \omega(s(n+1))$.

COROLLARY 1.2. *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 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.*

We point out that Corollary 1.2 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 Corollary 1.2 gives space hierarchies that are tight with respect to space for typical space bounds that are at most linear.

Our second main result gives a separation result with similar parameters as those of Theorem 1.1 but for the cases of one- and zero-sided error randomized machines. 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 1.3. *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 $s(n) = O(n)$.

COROLLARY 1.4. *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.*

1.1.2. Generic Semantic Models. 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 Van Melkebeek & Pervyshev (2007) that applies to a wide class of semantic models which the authors term *reasonable*. We refer to Section 4.4 for the precise definitions; but besides randomized two-, one-, and zero-sided error machines, the notion also encompasses bounded-error quantum machines

(Watrous 2003), unambiguous machines (Buntrock *et al.* 1991), Arthur-Merlin games and interactive proofs (Condon 1993), etc. When applied to the logarithmic space setting, the construction yields the following.

THEOREM 1.5 (**follows from Van Melkebeek & Pervyshev (2007)**). *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. 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 1.6. *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 1.7. *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 1.7 gives a tight separation result for slightly higher space bounds than Theorem 1.1 and Theorem 1.3, but the latter can handle more advice bits.

1.1.3. Promise Problems. 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 paper.

THEOREM 1.8 (**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 $s(n)$.*

1.2. Our Techniques. Recently, Van Melkebeek & Pervyshev (2007) showed how to adapt the technique of delayed diagonalization to obtain time hierarchies with one bit of advice for any reasonable semantic model of computation in which complementation can be performed with a linear-exponential overhead in space. For any constant a, they exhibit a language that is computable in polynomial time with one bit of advice but not in linear time with a bits of advice. Our results for generic models of computation (Theorem 1.5, Theorem 1.6, and Theorem 1.7) follow from a space-efficient implementation and a careful analysis of that approach.

Our stronger results for randomized machines follow a different type of argument, which roughly goes as follows. When N diagonalizes against machine M_i, it tries to achieve complementary behavior on inputs of length n_i by reducing the complement of M_i at length n_i to instances of some hard language L of length somewhat larger than n_i, say m_i. The hard language L is chosen so that it has a space-efficient recovery procedure, described momentarily. N

may not be able to compute L on those instances directly as it is unknown if L can be computed in small space. We instead use a delayed computation and copying scheme that forces M_i to aid N in the computation of L if M_i agrees with N on inputs larger than m_i. As a result, either M_i differs from N on some inputs larger than m_i, or else N can decide L at length m_i in small space and therefore diagonalize against M_i at length n_i.

The critical component of the copying scheme is the following task. Given a list of randomized machines with the guarantee that at least one of them satisfies the promise and correctly decides L at length m in small space, construct a single randomized machine that satisfies the promise and decides L at length m in small space. We call a procedure accomplishing this task a space-efficient *recovery procedure* for L.

The main technical contributions of this paper are the design of recovery procedures for adequate hard languages L. For Theorem 1.1 we use the computation tableau language, which is an encoding of bits of the computation tableaux of deterministic machines; we develop a recovery procedure based on the local checkability of computation tableaux. For Theorem 1.3 we use the configuration reachability language, which is an encoding of pairs of configurations that are connected in a nondeterministic machine's configuration graph; we develop a recovery procedure from the proof that NL=coNL (Immerman 1988; Szelepcsényi 1988).

1.2.1. Relation to Previous Work. Our high-level strategy is most akin to the one used in Van Melkebeek & Pervyshev (2007). In the time-bounded setting, Van Melkebeek & Pervyshev (2007) achieve a strong separation for two-sided error randomized machines using the above construction with satisfiability as the hard language L. The recovery procedure exploits the self-reducibility of satisfiability to obtain satisfying assignments for satisfiable formulae. As the partial assignment must be stored during the construction, this approach uses too much space to be useful in the setting of this paper.

Van Melkebeek & Pervyshev (2007) also derive a stronger separation for bounded error quantum machines in the time-bounded setting, with the hard language L being PSPACE-complete. A time-efficient recovery procedure for L follows from the existence of instance checkers (Blum & Kannan 1995) for L. The latter transformation of instance checkers into recovery procedures critically relies on large memory space. Instance checkers are only guaranteed to work when given a fixed oracle to test; their properties carry over to testing randomized procedures by treating randomized procedures as probability distributions over oracles. This works in the time-bounded setting because we

can ensure consistent answers to the oracle queries by storing the answers of the randomized procedure to all queries the first time they are asked. In the space-bounded setting we do not have the resources to store the answers to all queries, and it is not immediate that a small space instance checker for a language implies a small space recovery procedure. Using new ingredients, we develop a space-efficient recovery procedure for the computation tableau language from scratch. In fact, our argument shows that every language that admits a small space instance checker also admits a small space recovery procedure. However, this transformation inherently introduces two-sided error, and we use other techniques to develop recovery procedures for one- and zero-sided error machines.

For reasons of completeness, we point out that some of our results can also be obtained using a different high-level strategy than the delayed diagonalization with advice strategy of Van Melkebeek & Pervyshev (2007). Some of the results of Van Melkebeek & Pervyshev (2007) in the time-bounded setting can also be derived by adapting translation arguments to use advice (Barak 2002; Fortnow & Santhanam 2004; Fortnow *et al.* 2005; Goldreich *et al.* 2004). It is possible to derive our Theorem 1.1 and Theorem 1.3 following a space-bounded version of the latter strategy. However, the proofs still rely on the recovery procedure as a key technical ingredient and we feel that our proofs are simpler. Moreover, for the case of generic semantic models, our approach yields results that are strictly stronger.

1.3. Organization. Section 2 contains the elements of computational complexity theory we use in this paper. Section 3 contains the proofs of our separation results for randomized models (Theorem 1.1, Corollary 1.2, Theorem 1.3, and Corollary 1.4). Section 4 contains the proofs of our separation results for generic semantic models (Theorem 1.5, Theorem 1.6 and Theorem 1.7). Section 5 contains a proof of the hierarchy theorem for promise problems (Theorem 1.8).

2. Preliminaries

Here we introduce the machine models we use throughout the paper 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 Arora & Barak (2009) and Goldreich (2008).

2.1. Deterministic 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 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*

(i) on input i, $M(i)$ outputs a string y with $y \in S$,

(ii) for any $y \in S$, there exists an i such that $M(i)$ outputs y, and

(iii) $M(i)$ 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 all 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.

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 $a = \alpha_n$.

2.2. Randomized 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 Saks (1996).

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*

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

(ii) 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.

If we remove the requirement of bounded error in (i), 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 $x \in L$.

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*

(i) for every x, $\Pr_r[M(x;r) \notin \{0,1\}] \leq \frac{1}{2}$, and

(ii) 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.

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 take *advice* in much the same way that deterministic Turing machines take advice – as an additional input. We refer to Section 2.3 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, Motwani & Raghavan (1995, 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.*

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

(ii) 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 (i) and (ii), 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 ? if $M(x)$ outputs ? 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 (Nisan 1992; Saks & Zhou 1999), but the above suffices for our purposes.

2.3. Semantic Models. 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 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 (Watrous 2003), unambiguous machines (Buntrock *et al.* 1991), Arthur-Merlin games and interactive proofs (Condon 1993), etc. We refer to Van Melkebeek & Pervyshev (2007) 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 2.8 (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 Karp & Lipton (1982), 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 $n + a(n)$.

2.4. 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 2.9 (promise problem). *A promise problem is a pair of disjoint sets (Π_Y, Π_N) of strings.*

The set Π_Y in Definition 2.9 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 only has 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.

3. Randomized Machines with Bounded Error

In this section we give the constructions for Theorem 1.1, Corollary 1.2, Theorem 1.3, and Corollary 1.4. We first describe the high-level strategy used for these results. Most portions of the construction are the same for both, so we keep the exposition general. We aim to construct a randomized machine N and advice sequence α witnessing Theorem 1.1 and Theorem 1.3 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 define the latter as follows.

Definition 3.1 (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) reducing 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) performing a delayed computation of L at length m_i on padded inputs of length n_i^*, and (3) copying this behavior to smaller and smaller inputs down to input length m_i. 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.

We begin by assuming a hard language L as in (1) and develop an intuition for why advice and recovery procedures are needed to achieve (2) and (3) (Section 3.1). We then describe the hard language L and recovery procedure for L for the cases of two-sided error machines (Section 3.2) and one- and zero-sided error machines (Section 3.3). Finally, we complete the construction (Section 3.4) and give the analysis (Section 3.5) that yields the parameters stated in Theorem 1.1, Corollary 1.2, Theorem 1.3, and Corollary 1.4. An illustration of the completed construction with advice is given in Section 3. The

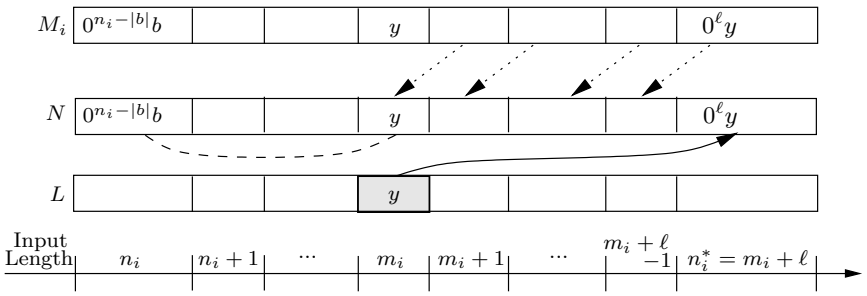

Figure 3.1: Illustration of the construction for Theorem 1.1 and Theorem 1.3. The solid arrow indicates that on input $0^\ell y$, N deterministically computes $L(y)$ for each y of length m_i. The dotted arrows indicate that for $\ell' \in [0, \ell - 1]$, on input $0^{\ell'} y$ with advice bit 1, N attempts 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 $N(y)$.

reader is encouraged to refer to Section 3 as we develop the construction.

3.1. The Need for Advice and Recovery Procedures. 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 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 3.2 (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 1.1, the recovery procedure may have two-sided error, while for Theorem 1.3, 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 Section 3.2 and Section 3.3 and complete the construction in Section 3.4.

3.2. 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 3.4 to complete the proof of Theorem 1.1. 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 3.3 (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 3.4. *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 3.4 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 $\frac{1}{9} + \frac{1}{9} + \frac{1}{9} = \frac{1}{3}$.

Input: $y = \langle x, t, j \rangle$ of length m; machines $P_1, P_2, ..., P_q$
Output: $\text{COMP}_D(y)$
(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 $P_d(\langle x, t', j' \rangle)$ lies in $[\frac{3}{8}, \frac{5}{8}]$ **then goto** (1) *P_d fails*
(4) **foreach** j' *Check base case – start configuration*
(5) $A \leftarrow$ majority of τ simulations of $P_d(\langle x, 0, j' \rangle)$
(6) **if** $A \neq j'^{th}$ 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) $A_{j'_c, t'-1} \leftarrow$ majority of τ simulations of $P_d(\langle x, t' - 1, j'_c \rangle)$
(12) $A_{j', t'} \leftarrow$ majority of τ simulations of $P_d(\langle x, t', j' \rangle)$
(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 $P_d(\langle x, t, j \rangle)$
(17) **return** 0 *No machines passed testing*

Figure 3.2: 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.

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 $[3/8, 5/8]$.

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 Section 3.2. 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 (13) use $O(s(m) + \log q)$ 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 $O(s(m) + \log q + \max_j(s_{P_j}(m)))$.

3.3. 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 3.4 to complete the proof of Theorem 1.3. 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 3.5 (CONFIG_M). *Given a randomized machine M, we define the configuration reachability language of M as follows.* $\mathit{CONFIG}_M = \{\langle x, c_1, c_2, t\rangle \mid$ *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 3.6. *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 3.6 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

```
Input: y = ⟨x, c1, c2, t⟩ of length m; machines P1, P2, ..., Pq
Output: CONFIG_M(y)
(1)  if c1 = c2 then Output "yes" and halt                    Trivial cases
(2)            else if t = 0 then Output "no" and halt
(3)  foreach d = 1..q          Try using Pd to compute CONFIG_M(y)
(4)    k0 ← 1          Number of configurations w/in distance 0 of c1
(5)    for ℓ = 1 to t                                  Compute kℓ given kℓ−1
(6)      kℓ ← 0
(7)      foreach configuration c          Is c w/in distance ℓ of c1?
(8)        k'ℓ−1 ← 0                    Re-experience all configurations-
(9)        foreach configuration c'                -within distance ℓ − 1
(10)         if Verify(⟨x, c1, c', ℓ − 1⟩, Pd) = "yes"
(11)           if M(x) transitions from c' to c in ≤ 1 time step
(12)                                          c is within distance ℓ of c1
(13)             if c = c2 then return "yes"
(14)             else kℓ ← kℓ + 1, and Try next c (line 7)
(15)           else
(16)             k'ℓ−1 ← k'ℓ−1 + 1
(17)       if k'ℓ−1 ≠ kℓ−1
(18)         Failed to experience all configs w/in distance ℓ − 1
(19)         if d < q then Try next d (line 3)                  Pd fails
(20)         else return "fail"                  All machines have failed
(21)   return "no"                  kt computed correctly and c2 not found
```

Figure 3.3: 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 Section 3.3.

from the fact that we are guaranteed that one of the machines in the list of machines correctly computes CONFIG_M at length m.

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 (Immerman 1988; Szelepcsényi 1988) 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 guesses have been made. For completeness, we explain how we make use of the nondeterministic algorithm of Immerman (1988) and Szelepcsényi (1988) 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 Section 3.3 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 Section 3.3. 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 Section 3.3. 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).

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) $c \leftarrow c_0$ *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** $P(\langle x, c'', c', j\rangle)$ outputs 1 on any of $O(s)$ trials
(9) $c \leftarrow c''$, 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 3.4: Pseudo-code for the verification subroutine used in the zero-sided error recovery procedure of Section 3.3. 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.

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 Section 3.3 and Section 3.3. 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 Section 3.3 uses $O(\log q)$ space. Line (8) of Section 3.3 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 $O(s(m) + \log q + \max_j(s_{P_j}(m)))$.

3.4. The Final Construction. We now complete the construction – which we began developing in Section 3.1 and is illustrated in Section 3 – used to prove Theorem 1.1 and Theorem 1.3. For Theorem 1.1, 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 3.2. For Theorem 1.3, 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 3.3.

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 Section 3.2 and Section 3.3). 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 $\ell = n_i^* - m_i$.

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

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 3.5: Pseudo-code for the diagonalizing machine N that witnesses Theorem 1.1 and Theorem 1.3. See Section 3.4 for a description of N in words.

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 3.5.

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 Theorem 1.1 and Theorem 1.3. Pseudo-code for the diagonalizing machine N/α described in this section is given in Section 3.4.

3.5. Analysis. We now explain how we come to the parameters given in the statements of Theorem 1.1,Theorem 1.3, Corollary 1.2 and Corollary 1.4.

3.5.1. Theorem 1.1 and Theorem 1.3. 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 Section 3.4 and the recovery procedures are given in Section 3.2, Section 3.3, and Section 3.3. The recovery procedure for each hard language (COMP_D in the case of Theorem 1.1 and CONFIG_M in the case of Theorem 1.3) 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 Section 3.4, $P_b(y')$ 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 Section 3.4 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 1.1, 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 1.3, 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 1.1 and st-connectivity for Theorem 1.3) 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 $\min(s(n), n)$.

3.5.2. Corollary 1.2 and Corollary 1.4. We now describe modifications to the construction that yield Corollary 1.2 and Corollary 1.4. 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 3.4 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 Corollary 1.2 and Corollary 1.4, 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 3.4, (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 3.4 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 $n \in [n_i, n_i^*]$.

3.5.3. Additional Remarks. We note that results corresponding to Theorem 1.1 and Corollary 1.2 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 1.3 and Corollary 1.4 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 1.3.

4. Separation Results for Generic Semantic Models

In this section, we prove our separation results for generic semantic models (Theorem 1.5, Theorem 1.6, and Theorem 1.7). The basic construction is the same for each, with only the analysis differing. We first review delayed diagonalization on syntactic models (Section 4.1), give the construction that adapts delayed diagonalization to semantic models with the use of advice (Section 4.2), analyze the construction for the particular case of each theorem (Section 4.3),

and finally distill the properties of a semantic model that are needed for our constructions to hold (Section 4.4).

4.1. Delayed Diagonalization on Syntactic Models. As the basic construction is an adaptation of delayed diagonalization (Žàk 1983) to handle advice, we first review delayed diagonalization on *syntactic* models. We wish to demonstrate a machine N using slightly more than $s(n)$ space which differs from all machines that use $s(n)$ space. 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, and (2) a scheme to copy this 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 space at length n_i^* to complement the behavior of M_i at length 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 $s(n)$ space, outputs a value if M_i does, and outright rejects if M_i uses more than $s(n)$ space. Suppose that M_i is a machine which uses at most $s(n)$ space 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 M_i differs from N on some input of length in $[n_i, n_i^*]$, or M_i uses more than $s(n)$ space. An illustration of delayed diagonalization is given in Section 4.1.

4.2. Delayed Diagonalization on Semantic Models. Consider the case of a *semantic* model of computation, defined in Section 2.3, where N must use not much more than $s(n)$ space, satisfy the promise on all inputs, and differ from each machine M_i which *behaves appropriately*, defined by the following generalization of Definition 3.1.

Definition 4.1 (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.*

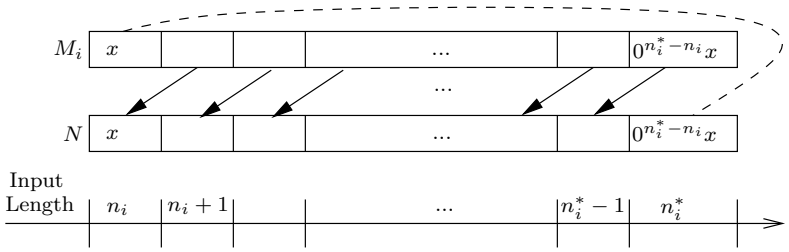

Figure 4.1: Illustration of delayed diagonalization on a *syntactic* model of computation. The solid arrows indicate that on inputs of the form $0^j x$, N simulates $M_i(0^{j+1}x)$. The dashed line indicates that on input $0^{n_i^*-n_i}x$, N outputs the complement of $M_i(x)$.

We keep a few specific semantic models in mind during the development and analysis of the construction – Arthur-Merlin games for Theorem 1.5, and unambiguous machines for the stronger separations of Theorem 1.6 and Theorem 1.7. 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 figure out what properties are needed of a semantic model afterward, namely in Section 4.4.

The delayed diagonalization construction given in Section 4.1 fails for non-syntactic models: it may be the case that M_i breaks the promise on inputs of the form $0^j x$, and N would also break the promise by performing the simulations described above. However, 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 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. The construction of Section 3 avoided this problem by considering a particular behavior that M_i might have – computing a hard language – and using this behavior to handle M_i with many advice strings at once. This entailed a recovery procedure for the hard language, which we do not know how to achieve for generic semantic models. In this section, we use a different approach that does apply to generic semantic models, which can be thought of as a copying scheme that allows N to spread the $2^{a(n+1)}$ advice bits needed to appropriately simulate M_i at a given length over many input lengths.

Consider the simulations of M_i at length n_i^* which N is responsible for copying to smaller padded inputs. We would like to give N one advice bit for each of M_i's possible advice strings at length n_i^*, indicating for each whether M_i with that advice string behaves appropriately. We spread these advice bits across multiple input lengths. 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 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 Section 4.2. 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

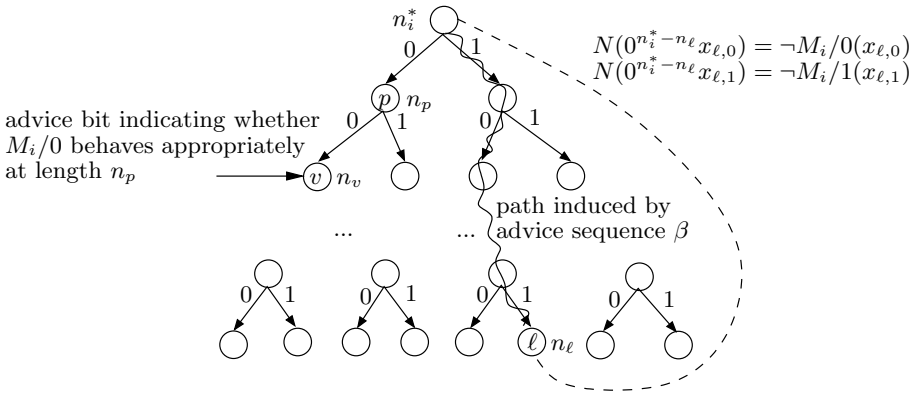

Figure 4.2: Illustration of N's execution for generic semantic models, shown for the case where M_i receives 1 bit of advice. Solid lines indicate that on the smaller input, N simulates M_i on padded inputs of the larger length, using the advice bit specified on the arrow. The dashed line indicates that 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.

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.

Definition 4.2 (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 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 (Savitch 1970), and for bounded-error randomized machines an overhead with exponent 3/2 is sufficient (Saks & Zhou 1999).

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.

4.3. 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 Theorem 1.5, Theorem 1.6, and Theorem 1.7.

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

$$a(n_i^{c^h}) + \sum_{j=2}^{h} a(n_i^{c^j}) \leq \log(n_i^c - n_i). \tag{4.3}$$

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

$$\log n_i^* = \Omega(\sigma(n_i^c)). \tag{4.4}$$

If we can simultaneously satisfy both (4.3) and (4.4), 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 1.5.

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 1.6.

4.3.1. Complementation with Linear-Exponential Overhead (Theorem 1.5). 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, (4.4) becomes

$$\log n_i^* = \log n_i^{c^h} = \Omega(n_i^{cd'}). \tag{4.5}$$

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 (4.5). 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 (4.3) becomes

$$k + \sum_{j=2}^{h} k = h \cdot k \leq \log(n_i^c - n_i). \tag{4.6}$$

As the right-hand side grows faster with c than the left-hand side, we can pick c sufficiently large so that both (4.5) and (4.6) 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 1.5 for the case of semantic models such as Arthur-Merlin games. Section 4.4 contains a precise statement of the properties needed of a semantic model for our proof of Theorem 1.5 to apply.

4.3.2. Complementation with Polynomial Overhead (Theorem 1.6). 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 (Savitch 1970) and $d = 3/2$ for bounded-error randomized machines (Saks & Zhou 1999). Thus (4.4) 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 (4.3). 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 (4.3). 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 (4.3) and (4.4), as follows. Let $a(n) = k \log^{1/d} n$ for some integer $k > 0$. Substituting into (4.3) yields

$$k \log^{1/d}(n_i^{c^h}) + k \sum_{j=2}^{h} \log^{1/d}(n_i^{c^j}) \leq \log(n_i^c - n_i). \tag{4.7}$$

For technical reasons, we aim to satisfy (4.4) by ensuring

$$c^3 \log n_i^* = c^3 \log(n_i^{c^h}) \geq \log^d(n_i^c), \tag{4.8}$$

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 inequality (4.7).

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

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

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

Adding up these two values satisfies inequality (4.7) 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 1.5, 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 1.6. Among others, Theorem 1.6 applies to semantic models such as unambiguous machines and bounded-error randomized machines. Section 4.4 contains a precise statement of the properties required of a model for our proof of Theorem 1.6 to apply.

4.3.3. Larger Space Bounds (Theorem 1.7). 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 1.6 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 1.7. 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 Theorem 1.5 and Theorem 1.6, 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 1.7.

The main idea of the proof of Theorem 1.7 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.

4.4. Generic Semantic Models. Consider the properties of the machine model used in the above analysis of Theorem 1.5, Theorem 1.6, and Theorem 1.7. 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 4.9 (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:*

(i) 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 $O(s + \log(|x| + |M_i|))$.

(ii) 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 Theorem 1.5, Theorem 1.6, and Theorem 1.7 in Section 4.3 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 4.10 (space overhead of safe complementation). *Fix a reasonable semantic model of computation with U the machine given by part (i) of Definition 4.9. 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 1.5 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 Karpinski & Verbeek (1987) does not apply.

Theorem 1.6 and Theorem 1.7 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 Saks & Zhou (1999), 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 (Savitch 1970). 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 (Fortnow & Lund 1993).

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 4.9. 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.

5. Promise Problems

For a wide class of semantic models with at least one bit of advice we have shown that a little bit more space allows them to compute strictly more languages. Such tight hierarchies remain open if we do not allow any advice at all. However, we can establish tight hierarchies for generic semantic models without advice if we consider promise (decision) problems (defined in Section 2.4) instead of languages. In fact, plain delayed diagonalization suffices to do so. The proofs exploit the freedom which a machine solving a promise problem has to violate the promise underlying the semantic model on inputs that do not satisfy the promise underlying the problem. In the previous sections we introduced advice exactly to prevent the machine that witnesses the hierarchy from violating the promise underlying the model. This is why the transition from decision problems to promise problems obviates the need for advice.

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. We remedy both this problem and the above by using delayed diagonalization. We first prove Theorem 1.8

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 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 n_i is in Π_Y, Π_N, or neither), $N(0^{n_i^*})$ does something 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'.

The above proof requires only a basic set of properties and holds for any reasonable semantic model in which safe complementation can be achieved with a computable overhead in space, i.e., a model that has a safe complementation with overhead σ for some computable σ in Definition 4.10. 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, 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 1.8.

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 1.8.

THEOREM 5.1 (**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 $t(n)$.*

Acknowledgements

A preliminary version of this work appeared under the title "Space Hierarchy Results for Randomized Models" in the 25^{th} *annual International Symposium on Theoretical Aspects of Computer Science*, held in Bordeaux, France 2008.

We thank Scott Diehl for many useful discussions, in particular pertaining to the proof of Theorem 1.3. We also thank the anonymous reviewers of both

the conference and journal versions of this paper for their time and many useful suggestions.

Portions of this work were completed while both authors were supported by NSF awards CCR-0133693 and CCR-0728809 and while the first author was supported by a Cisco Systems Distinguished Graduate Fellowship.

References

Sanjeev Arora & Boaz Barak (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press. To appear, preliminary version available from http://www.cs.princeton.edu/theory/complexity/.

Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan & Mario Szegedy (1998). Proof Verificiation and the Hardness of Approximation Problems. *Journal of the ACM* **45**(3), 501–555.

László Babai, Lance Fortnow & Carsten Lund (1991). Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity* **1**(1), 3–40.

Boaz Barak (2002). A Probabilistic-Time Hierarchy Theorem for Slightly Non-Uniform Algorithms. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, José D. P. Rolim & Salil P. Vadhan, editors, volume 2483 of *Lecture Notes in Computer Science*. Springer-Verlag.

Manuel Blum & Sampath Kannan (1995). Designing Programs that Check Their Work. *Journal of the ACM* **42**(1), 269–291.

Gerhard Buntrock, Birgit Jenner, Klaus-Jorn Lange & Peter Rossmanith (1991). Unambiguity and Fewness for Logarithmic Space. In *Proceedings of the 8th International Conference on Fundamentals of Computation Theory 1991,* Gosen, Germany, Lothar Budach, editor, volume 529 of *Lecture Notes in Computer Science*, 168–179. Springer-Verlag.

Anne Condon (1993). The complexity of space bounded interactive proof systems. In *Complexity Theory: Current Research*, Steven Homer, Uwe Schöning & Klaus Ambos-Spies, editors, 147–190. Cambridge University Press.

Stephen Cook (1973). A Hierarchy Theorem for Nondeterministic Time Complexity. *Journal of Computer and System Sciences* **7**, 343–353.

Lance Fortnow & Carsten Lund (1993). Interactive proof systems and alternating time-space complexity. *Theoretical Computer Science* **113**(1), 55–73.

Lance Fortnow & Rahul Santhanam (2004). Hierarchy theorems for probabilistic polynomial time. In *Proceedings of the 45nd Annual IEEE Symposium on Foundations of Computer Science,* Rome, Italy, 316–324.

Lance Fortnow, Rahul Santhanam & Luca Trevisan (2005). Hierarchies for semantic classes. In *Proceedings of the Thirty-seventh Annual ACM Symposium on the Theory of Computing,* Baltimore, Maryland, USA, 348–355.

Oded Goldreich (2008). *Complexity Theory: A Conceptual Perspective.* Cambridge University Press.

Oded Goldreich, Madhu Sudan & Luca Trevisan (2004). From Logarithmic Advice to Singe-bit Advice. Technical Report TR-04-093, Electronic Colloquium on Computational Complexity.

Neil Immerman (1988). Nondeterministic Space is Closed Under Complementation. *SIAM Journal on Computing* **17**(5), 935–938.

Richard Karp & Richard Lipton (1982). Turing machines that take advice. *L'Enseignement Mathématique* **28**(2), 191–209.

Marek Karpinski & Rutger Verbeek (1987). Randomness, provability, and the separation of Monte Carlo time and space. In *Computation Theory and Logic*, Egon Börger, editor, volume 270 of *Lecture Notes in Computer Science*, 189–207. Springer-Verlag.

Dieter van Melkebeek & Konstantin Pervyshev (2007). A Generic Time Hierarchy for Semantic Models With One Bit of Advice. *Computational Complexity* **16**, 139–179.

Rajeev Motwani & Prabhakar Raghavan (1995). *Randomized Algorithms.* Cambridge University Press.

Noam Nisan (1992). RL $\subseteq$ SC. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Theory of Computing,* Victoria, British Columbia, Canada, 619–623.

Michael Saks (1996). Randomization and Derandomization in Space-Bounded Computation. In *Proceedings of the 11th IEEE Conference on Computational Complexity,* Washington DC, 128–149.

Michael Saks & Shiyu Zhou (1999). $BP_HSPACE(S) \subseteq DSPACE(S^{3/2})$. *Journal of Computer and System Sciences* **58**, 376–403.

W. Savitch (1970). Relationship between nondeterministic and deterministic tape classes. *Journal of Computer and System Sciences* **4**, 177–192.

Joel Seiferas, Michael Fischer & Albert Meyer (1978). Separating Nondeterministic Time Complexity Classes. *Journal of the ACM* **25**, 146–167.

Adi Shamir (1992). IP = PSPACE. *Journal of the ACM* **39**(4), 869–877.

Róbert Szelepcsényi (1988). The method of forced enumeration for nondeterministic automata. *Acta Informatica* **26**(3), 279–284.

Seinosuke Toda (1991). PP is as hard as the Polynomial-time Hierarchy. *SIAM Journal on Computing* **20**(5), 865–877.

John Watrous (2003). On the complexity of simulating space-bounded quantum computations. *Computational Complexity* **12**, 48–84.

Stanislav Žàk (1983). A Turing Machine Time Hierarchy. *Theoretical Computer Science* **26**, 327–333.

Manuscript received December 12, 2008

Jeff Kinne
Department of Computer Sciences
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706-1685
USA
jkinne@cs.wisc.edu

Dieter van Melkebeek
Department of Computer Sciences
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706-1685
USA
dieter@cs.wisc.edu