The Computational Complexity Column

BY

Michal Koucký

Computer Science Institute, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
koucky@iuuk.mff.cuni.cz
https://iuuk.mff.cuni.cz/~koucky/

Reusing Space: Techniques and Open Problems

Ian Mertz[*‡]

Abstract

In the world of space-bounded complexity, there is a strain of results showing that space can, somewhat paradoxically, be used for multiple purposes at once. Touchstone results include Barrington's Theorem and the recent line of work on catalytic computing. We refer to such techniques, in contrast to the usual notion of reclaiming space, as *reusing space*.

In this survey we will dip our toes into the world of reusing space. We do so in part by studying techniques, viewed through the lens of a few highlight results, but our main focus will be the wide variety of open problems in the field.

In addition to the broader and more challenging questions, we aim to provide a number of questions that are fairly simple to state, have clear practical and theoretical implications, and, most importantly, that a newcomer with little background experience can still sit down and play with for a while.

Contents

[*]Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, UK. Email: ian.mertz@warwick.ac.uk.

[†]The author received support from the Royal Society University Research Fellowship URF\R1\191059 and from the Centre for Discrete Mathematics and its Applications (DIMAP) at the University of Warwick.

The first subroutine I coded, learning the principles of object oriented programming in a CS 101 course, was the swap function. Every coder knows it by heart: three lines—their syntax is nearly universal across all languages—and only using the two input variables in question plus one additional:

```
temp = x
x = y
y = temp
```

Years later I was adding the same function as part of the preamble to a larger assignment, when the TA gave me a puzzle: can you swap two bits without the temp register? Yes, and with no increase in the length of the program:

```
x = x ⊕ y
y = x ⊕ y
x = x ⊕ y
```

While not as ubiquitous as the standard program, no shortage of experienced coders have encountered this problem before; indeed it is one of many gateways to the wide world of bit-tricks. But like all good hammers, the answer, an elegant improvement to one of the standard subroutines in all of computer science, begs one to go looking for nails, or, better still, more hammers of a similar ilk.

1 Introduction: TCS Wants YOU (To Reuse Space)

1.1 Who, me?

Why should we think about reusing space? Consider some clickbait-ized headlines for the successes of the field thus far:

- any function can essentially be computed using just a three-bit memory
- access to a hard drive can be more powerful than non-determinism, even when it's full
- our central approach to separating P from L contains a fatal flaw
- the max flow rate is not an upper bound on the size of messages that can be sent through a network

These are sensational highlights, and for those in space-bounded complexity or similar fields they may warrant scrutiny on their own merit. But in this work, rather than giving a status report on a distant field, I want to invite a broader audience to consider coming into the fold themselves, and so our focus will be slightly different.

The survey will be split into two parts. For the readers in search of hammers, we will give an overview of the major techniques in the field, which should be sufficiently general and straightforward so as to spark the reader's imagination. And for the readers who enjoy a good nail, we will present a broad swath of open problems in the field, from puzzles to be worked on over an idle lunch to the titanic central problems in space complexity.

Part I will give some background into the existing techniques on reusing space. This part will follow more of the typical pattern of a survey, but with all proofs kept at a fairly high level with minimal definitions or details; we eschew all formal preliminaries of basic objects—e.g. circuits, branching programs, etc.—in favor

of brief descriptions of relevant characteristics, as our goal is simply to give the reader a taste for past arguments. We also include a number of exercises to the reader[1], whose answers can be found in the appendix at the end.

In Part II we lay out a number of open problems in the world of reusing space. These sections will give a flavor of the problem itself, known results and techniques, possible hindrances, and consequences of solving them. Again the details will be left fairly light, focusing on breadth rather than depth in order to appeal to many types of problem solvers, as well as to avoid personal biases as much as possible, although in this I admit to have largely failed.

1.2 Reducing via Reusing (beyond Recycling)

Per the title, we focus on *reusing*, rather than simply *saving*, space, even though proving upper bounds through space-saving algorithms is our ultimate aim. Let us disambiguate these terms now.

What could reuse mean, beyond reclaiming space as it becomes available? The latter is something we understand quite well. When we study the space complexity of some function, we naturally focus on what the algorithm *needs* to remember during the computation, and so we design algorithms with the aim of remembering, at any point in time, as little information as possible.

In this survey we study a slightly different question: can we store *lots* of information but do so using *much less* space? By this we do not mean a simple question of compression, for which the same information theoretic bottlenecks still hold sway. We ask a more suggestive question: *can we use space for two things at the same time?*

We prime the reader to this possibility by mentioning two distinct uses of space as a resource writ large: *storage* and *work*. At any moment in a computation, there may be information concerning the global computation that must be written down for future use, while simultaneously there are local computations that must be performed with the aid of the tape as well.

The question of whether these need occupy separate places in memory—what one could call the *composition* question for space—may at first glance appear trivially true, but this is not the case. In Section 2 we will see how memory can be used for two such purposes at once by analyzing the proof of Barrington's seminal result [Bar89], as adapted from a follow-up work of Ben-Or and Cleve [BC92].

[1]Typically we write "this is left as an exercise to the reader" to denote either something trivial or something technical but lacking in key ideas; in any event, something to be glossed over. In this survey it means precisely the opposite: readers who want to get comfortable with the techniques presented are highly encouraged to try them out and check their work.

1.3 Our test module: catalytic computing

Our goal in this survey is to appeal to a broad audience within theoretical (and possibly even applied) computer science about the intriguing mysteries of reusing space. Thus as much as possible we will refrain from narrowing the focus to one model or another.

However, we cannot avoid introducing a model of space which has been intertwined with such questions for many years now, and which seems a natural first stop when studying any questions about reusing space: *catalytic computing*.

Consider the two uses alluded to in the previous section: storage space and work space. How can we focus on separating out these uses? The simplest way is to imagine a situation where the work space we seek to use stores information that is completely unrelated to any computation at hand; in fact, we go a step further and consider a work tape populated with *arbitrary* bits, and see whether or not such a tape can still be useful for computation.

To focus on this question, imagine a space-bounded machine in the usual sense, but now we also give it access to a second work space, which we call the *catalytic tape*. While our main work tape is initialized to be empty, our catalytic tape is initialized to be full; what information fills the tape is arbitrary and out of our control, and while we allow the machine to use the catalytic tape however it chooses, we stipulate that the machine should return this memory to its original state at the end of the computation.[2]

Seeing if clever use of the catalytic memory allows us to exceed the power of a typical bounded space machine is one clear way to test our ability to reuse space without simply reclaiming memory used for past computations. In Section 3 we will mention some key results in catalytic computing, and more importantly the techniques used in these results, which shows that such reuse is not only possible but quite powerful.

1.4 Many flavors of questions

Given these preliminary motivations, there are many types of problems we can ask. We loosely group these into four categories. In Section 4, we tackle the basic premise of our field by looking to apply the techniques of reusing space to answer questions in space-bounded complexity. In Section 5, we ask how the catalytic computing model compares with traditional complexity classes, both space-

[2]The term catalytic refers to a catalyst in chemistry, which is a chemical unrelated to, and ultimately preserved in quantity by, a given reaction, but whose presence is nevertheless necessary for the reaction to occur. We also note that there are multiple unrelated definitions of "catalytic computing" circulating in the CS literature, including one for network systems and another in quantum computing; all of these models, however, are named for the same physical phenomenon.

bounded and otherwise. Section 6 asks similar questions but comparing catalytic classes to one another in an attempt to flush out a parallel structural theory of space. Finally in Section 7 we go beyond space-bounded complexity classes and consider when the techniques we have seen in this work may apply to alternative computational models; in this section in particular I invite the reader to think about their own research and build novel connections to the framework of reusing space.

1.5 The purpose of this survey

Writing a survey article is a chance for the author to bring a new and (personally) exciting field to the reader's attention, and to isolate the central and furthest reaching successes therein. It can appeal to the utility of such results and proofs, ones the reader, a researcher with a full schedule and their own field and goals, may have never even heard of. It is, then, a way to plant the seed of interest while respecting the reader's time.

Why, then, do we focus not on the use to the reader, but rather to beg them to drop their own work and spend time on an alien set of questions? One answer is that the former duty has already been discharged in the 2016 edition of this column by Michal Koucký [Kou16]; while there have been exciting results since then, another review only seven years later is hardly warranted. We will spend Part I of our survey covering some of the basics that appeared in this excellent work, but it will be for the sake of definition and intuition; for all other purposes, I refer you to therein.

Another answer, which was alluded to in the preamble, is that the nascent field finds itself in a very fortunate position: many of the existing open questions in reusing space can be described, motivated, and attacked with very little background. Some revolve around basic arithmetic; some involve drawing small graphs. There are no shortage of questions that ask for a slight twist on some fundamental theorem from an introductory complexity course. Hence they may be appropriate for those looking for "toy" (but still consequential) problems to play with, such as early career researchers—I have even given some problems to undergraduates in the past—or those who enjoy doing puzzles at dinner.

Lastly, I hope that beyond the specific problems at hand, the reader will take with them, back to their own specific subfield, the question of how one might use space, or indeed any other resource, in more than one way at once, and to what end. The faith that motivates this survey is that the question of reusing space will both benefit and benefit from researchers from a wide variety of fields; I will present one approach to solving one type of question, but more than pushing this particular angle forward, what the field needs, and potentially offers, is a greater variety of approaches to, and understandings of, its central tenets.

Part I
WHAT WE KNOW (THE BASICS)

2 An introductory example: Barrington's Theorem

2.1 Statement and proof

In order to calibrate ourselves to the task of reusing space, let us see one simple but foundational example of what this actually looks like.

Our adversary will be the circuit, a classical model of computation wherein an input x is fed bit by bit into a network of AND, OR, and NOT gates. Let C be a circuit taking n inputs; to simplify matters, we remove every OR gate from C using de Morgan's laws, and assume every AND gate takes two inputs.

"Theorem": C can be computed using effectively three bits of memory.

The statement "C can be computed using three bits of memory" is, of course, assuredly false; the proof of our "theorem" will fall short of this, due to a variety of technical considerations, including uniformity, counters for runtime, etc. However, it is correct in a moral sense, one which can be converted into useful statements and algorithms, and, more important at the present, provides the basis for our view of reusing space throughout the rest of this survey.

The proof is self-contained and only relies on basic modular arithmetic, and thus we state the proof first and save the background and intuition for the rest of the section.

Proof sketch. Our argument will be by induction on the gates of the circuit C. Let (R_0, R_1, R_2) be our three bit memory, initialized to $(0, 0, 0)$, and fix the input x under consideration. Our inductive statement is as follows:

Lemma 1. *Let (R_0, R_1, R_2) be in some state $(\tau_0, \tau_1, \tau_2) \in \{0,1\}^3$, and let g be a gate in C which takes value v_g on input x. Then for $i \in \{0, 1, 2\}$, there is a program $P_g(i)$ which transforms the memory as follows:*

$$\begin{aligned} R_j &= \tau_j \oplus v_g \quad j = i \\ R_j &= \tau_j \qquad\quad j \neq i \end{aligned}$$

We see that this is sufficient to compute f. Our first recursive call will be to the output gate *out* of C, say $P_{out}(0)$, and here (τ_0, τ_1, τ_2) is the initial blank tape, i.e. $(0, 0, 0)$; thus at the end of the computation, R_0 will be in state $0 \oplus v_{out} = f(x)$.

So now we take up this task. From here out out we view everything through the lens of arithmetic modulo 2, meaning + denotes $\oplus$. We use notation R += v

to mean $R \leftarrow R + v$, and so our goal is to design, for every gate $g \in C$ and $i \in \{0, 1, 2\}$, a program $P_g(i)$ which computes $R_i \mathrel{+}= v_g$ while leaving all other memory untouched.

The base case is simple enough: if g is one of the inputs to the global function f, say x_j, then we can simply add the relevant input to whichever register we please and we are done:

$$R_i \mathrel{+}= x_j$$

Now let g be an internal gate in the circuit, i.e. either NOT or AND. NOT is simple enough: let $g = \neg h$, let v_h be the value of h, and by induction let $P_h(i)$ be a program computes $R_i \mathrel{+}= v_h$. Then simply running $P_h(i)$ and then adding 1 yields

$$R_i = \tau_i + v_h + 1 = \tau_i + \neg v_h = \tau_i + v_g$$

Thus our last case is to compute $g = g_1 \wedge g_2$, and without loss of generality we focus on $P_g(0)$, as $P_g(1)$ and $P_g(2)$ can be accomplished by relabeling.

Let g_1 and g_2 take values v_1 and v_2, and by induction let $P_1(1)$ and $P_2(2)$ be programs which send R_1 to $\tau_1 + v_1$ and R_2 to $\tau_i + v_2$, respectively. The following program sends R_0 to $\tau_0 + v_1 v_2$; the relevant memory states are listed inline, with brackets separating out the previous memory state and the new additions:

1.	$P_1(1)$	R_1	$= [\tau_1] + [v_1]$
2.	$R_0 \mathrel{+}= R_1 R_2$	R_0	$= [\tau_0] + [(\tau_1 + v_1)(\tau_2)]$
			$= \tau_0 + \tau_1\tau_2 + v_1\tau_2$
3.	$P_2(2)$	R_2	$= [\tau_2] + [v_2]$
4.	$R_0 \mathrel{+}= R_1 R_2$	R_0	$= [\tau_0 + \tau_1\tau_2 + v_1\tau_2] + [(\tau_1 + v_1)(\tau_2 + v_2)]$
			$= \tau_0 + \tau_1 v_2 + v_1 v_2$
5.	$P_1(1)$	R_1	$= [\tau_1 + v_1] + [v_1]$
			$= \tau_1 \quad \checkmark$
6.	$R_0 \mathrel{+}= R_1 R_2$	R_0	$= [\tau_0 + \tau_1 v_2 + v_1 v_2] + [(\tau_1)(\tau_2 + v_2)]$
			$= \tau_0 + \tau_1\tau_2 + v_1 v_2$
7.	$P_2(2)$	R_2	$= [\tau_2 + v_2] + [v_2]$
			$= \tau_2 \quad \checkmark$
8.	$R_0 \mathrel{+}= R_1 R_2$	R_0	$= [\tau_0 + \tau_1\tau_2 + v_1 v_2] + [(\tau_1)(\tau_2)]$
			$= \tau_0 + v_1 v_2 \quad \checkmark$

which completes the proof, as $R_0 = \tau_0 + v_g$ and $R_i = \tau_i$ for $i = 1, 2$. □

The statement and proof above are adaptations of the seminal works of Barrington [Bar89] and Ben-Or and Cleve [BC92]. We give a more exact statement at the end of this section, turning now to focus on a higher level understanding of their technique.

2.2 Back to basics

Let us now rewind and build back towards our "theorem" from the ground up. Our goal is to demystify the proof we have just seen, and in the process to begin thinking about how these principles can be be extended.

2.2.1 Circuits and space

In the world of Boolean functions, circuits are one of the most fundamental computation models, and are universal in the sense that any function has a corresponding circuit. However, we typically consider the class of functions f which are *efficiently* computable by circuits, both with respect to *total time*, measured by the number of gates, and *parallel time*, measured by the longest path from any input wire to the output of the circuit. We say the *size* of a circuit is the number of gates, while the *depth* of a circuit is the length of the longest input-output path.

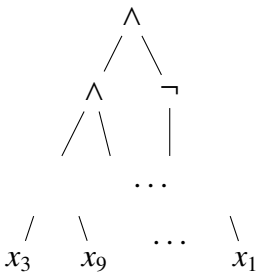

Fig.: circuit

More specifically, consider a family of functions $f = \{f_n\}_{n\in\mathbb{N}}$, each f_n taking in n input bits. Let f be computable by a family of circuits $C = \{C_n\}_{n\in\mathbb{N}}$, where each C_n has size s_n and depth d_n. Our goal will be to design a machine which computes f_n using the minimal possible space.

As should be clear from the initial statement, we will not be rigorous about which operations are allowed and such; we will assume that our machine has knowledge of C_n, and that at each step it makes use of all available information—by which we mean the circuit, the input, and the current state of our memory—to progress in the computation.

We begin with a sanity check: each f_n can be computed without reusing any space, and the space complexity corresponds to the size of C_n.

Claim 1. *f_n can be computed in space s_n.*

Proof sketch. We assign to each gate in C_n a spot on our work tape, and progress through the circuit in order writing down the value of each gate. If the gate uses any input bits directly we take them from the input tape, while if it takes previously computed gates as input we read their values from the work tape. The final gate will compute the output of C_n, and thus the value of f_n. □

This procedure is simplicity itself, and yet clearly wasteful. As soon as some internal gate of C_n becomes irrelevant to future computation, we could reclaim this space to use for later gates.

Alternatively, we can throw away information about a gate even if it is required for later use, as long as we are willing to pay to recompute it in the future, perhaps when more space is at our disposal.

Can this logic be exploited when we know nothing about C_n besides its size and depth? Indeed we can, if we consider how and when computations are reused. Thus we can reduce the space complexity from the total runtime of C_n to its parallel runtime, from size to depth.

Claim 2. *f_n can be computed in space d_n.*

Proof sketch. For each gate $g \in C_n$, we define the *level* of g to be the length of the longest finite path any input of the circuit takes to get to g. Since every input takes at most d_n steps to reach the output of the circuit, every gate is at level at most d_n, and the output gate is at level d_n exactly.

We will inductively prove that any gate at level k can be computed with space k, thus proving the claim by considering the output gate at level d_n. For any gate g at level 1, the inputs to g simply come from the input itself, and thus we can write down the value of g directly from considering the input tape.

Now for gate g at level k, let g_1 and g_2 be the inputs to g, each of which occur at some level strictly less than k. To compute g we first compute g_1, which by induction can be done in space at most $k - 1$. Now we erase the entire work tape save for the output of g_1, and we then compute g_2 in the free space on our tape. This again can be done in space at most $k - 1$, and combining this with the output of g_1 we saved earlier gives us space k to compute g itself. □

2.2.2 Constant space: who could expect it?

Here we see that a careful view of how the internal computation of a circuit works can point us towards lowered space. Can this be pushed further?

While it may seem bold to press on, we may take heart from observing that at any moment in time, only *two* bits of the work tape are relevant to the computation at hand, no matter where in the recursion we may find ourselves. Thus the true test of our resolve is to ask: can we compute f using only a *constant* amount of memory?

On the face of things, this task is manifestly impossible. It is certainly true that only two bits are needed at any moment in time; yet this is making a similar oversimplification as saying only the output gate of C_n is truly important for computing the value of f_n. Each internal computation is unimportant alone, but is vital for the next step, and as such it is not just which bits are relevant in *this* step, but also which bits will be relevant in *later* steps. For a concrete example, during the computation of g_2 in the proof of Claim 2, it may be correct to say the value of g_1

is irrelevant now, but to erase it would clearly be shooting ourselves in the foot in a moment's time.

Thus to begin, we must ask whether we can broaden our horizons in defining what it means to reuse space. So far we have limited ourselves to *reclaiming* space and using it as if it were fresh. A more daring idea is to reuse space *in situ*, or in other words to use space for more than one purpose *at the same time*.

Here we focus on two distinct uses of space during the computation: space as providing *inputs* to the current computation and space as *storage* for future computation. If a bit of memory could play both roles simultaneously—to again use our previous proof as a concrete reference, if we imagine that the space *storing* g_1 could be used in parallel for *computing* g_2—then our previous objection becomes weaker.

2.3 Proof redux

We are now ready to challenge our "theorem" once again. By building our ideas of how to reuse space from the ground up, we will once again reach the proof given at the beginning of this section.

Naturally we will once again attempt an inductive approach, computing the circuit gate by gate. The proof lies in two insights: first, in the way that it arrives at an *algebraic* definition for reusing space which can be turned into a suitable recursive statement; and second, in the way that definition's algebraic nature suggests the method by which the statement can be achieved, if one tries at every step to do what immediately brings them closer to the goal.

2.3.1 Idea 1: modular arithmetic

As suggested by the discussion from the previous section, we need to choose a recursive statement whereby g is successfully added in memory while not erasing, and in fact in a formal sense preserving, its current state.

The term "adding" in the previous statement should immediately call to mind one such potential definition: *XORing* the value of g to memory. To figure out our exact statement, let us put ourselves in a moment within the computation.

With respect to memory as storage, we will imagine our three-bit memory is in some state $(\tau_0, \tau_1, \tau_2) \in \{0,1\}^3$, the exact value of which will be dictated by the recursion thus far; rather than subject ourselves to working out an exact statement for the values τ_i, we consider them to be arbitrary and out of our control.

With respect to memory as computation, our goal will be to send one of these bits, say τ_0, to the value $\tau_0 + v_g \mod 2$, where again v_g is the value of g in question. This statement suggests that recursively we should assume we have the ability to

do the same below us, namely that we can send τ_i to $\tau_i + v_h \mod 2$ for any gate h which is an input to g.

Furthermore, in order to respect the stored values τ_i, we will design our program in such a way that after computing v_g into R_0, the values stored in R_1 and R_2, namely τ_1 and τ_2, are left unchanged, and whereby recomputing v_g allows us to recover τ_0, i.e. to restore our last piece of memory R_0.

This brings us squarely to our choice of recursion statement as given by Lemma 1. Given this concrete goal, and with the assurance that proving it is sufficient to proving the "theorem", we may be encouraged by how simply the base case, i.e. the input layer, as well as the case of $g = \neg h$, can be dismissed with.

This only leaves us with proving the recursive statement for $g = g_1 \wedge g_2$. The reader is invited to pause here and take stock by attempting to do so themselves; it is the certainly the only tricky part of the argument once Lemma 1 is formulated, but it can be accomplished with a little trial and error.

2.3.2 Idea 2: handling multiplication

It seems that the only place to start is adding $g_1 \wedge g_2$ to τ_0 in any way we can, and then seek to fix things up from there. The most natural way to do this is to execute $P_1(1)$ and $P_2(2)$, and add the AND of the resulting memory to τ_0. Viewing AND as multiplication modulo 2 and expanding the product, we get

$$\tau_0 + (\tau_1 + g_1)(\tau_2 + g_2) = \tau_0 + \tau_1\tau_2 + \tau_1 g_2 + g_1\tau_2 + g_1 g_2$$

which contains exactly the terms we want, namely $\tau_0 + g_1 g_2$, as well as three junk terms, namely $\tau_1\tau_2 + \tau_1 g_2 + g_1\tau_2$.

To remove these terms, let us start with $\tau_1 g_2$, which suggestively does not contain g_1. Executing $P_1(1)$ sends $\tau_1 + g_1$ to $\tau_1 + g_1 + g_1 = \tau_1$, and thus the AND of our two work bits is $\tau_1(\tau_2 + g_2) = \tau_1\tau_2 + \tau_1 g_2$. We add this to our target memory.

We can take care of $g_1\tau_2$ similarly, executing $P_2(2)$ to remove g_2 and $P_1(1)$ to get back g_1, and then adding $R_1R_2 = (\tau_1 + g_1)\tau_2$ to our target register. Thus our memory contains

$$(\tau_0 + g_1 g_2 + \tau_1\tau_2 + \tau_1 g_2 + g_1\tau_2) + (\tau_1\tau_2 + \tau_1 g_2) + (\tau_1\tau_2 + g_1\tau_2) = \tau_0 + g_1 g_2 + \tau_1\tau_2$$

at which point we can simply reset our "external memory" by executing $P_1(1)$ one last time, and then adding the AND of R_1 and R_2 one last time seals the deal.[3]

After all is said and done we have not only added $g_1 \wedge g_2$ to τ_0, but in fact have set R_1 and R_2 back to their original values, thus fulfilling the other requirement of our recursive call.

[3]Note that our original proof was more efficient in terms of recursive calls; our only instructions are recursive calls and R_0 += R_1R_2, meaning the order in which we add our terms is irrelevant.

2.4 Afterword: next steps

The techniques involved in this section are crucial to understanding many of the arguments at the forefront of reusing space. Thus we will use this opportunity to give the reader their first exercise; since the solution can be found earlier in this section, the solutions manual will only contain an alternate analysis of the multiplication program.

Exercise 1. *Watch an episode of Gilligan's Island*[4]*, then come back and attempt to reprove our "theorem".*

We also dismiss with the tedious use of "theorem" by stating the actual result in question. This statement, the true form of Barrington's Theorem [Bar89], follows by a more optimized variant of Lemma 1, while Ben-Or and Cleve [BC92] devised Lemma 1 essentially as stated to prove a more general theorem.

Space-bounded computation has a very well-studied syntactic model to call its own: branching programs. A branching program is an directed acyclic graph with a start (source) node and two potential end (sink) nodes, one for each output of the function, where each node is labeled with an input variable x_j and has one outgoing edge for each potential value of x_j, i.e. 0 or 1. Computation is done in the natural way, starting at the source and at each node following the edge whose label agrees with the value of the x_j labeling the node itself, until we reach an output node whose value is declared the output of the function.

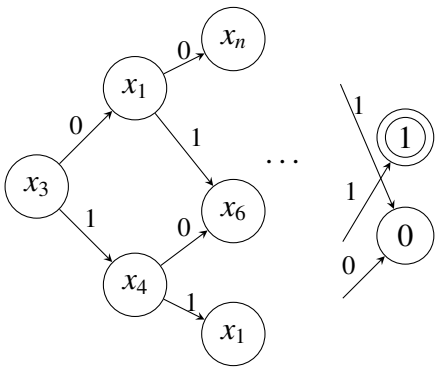

Fig.: branching program

While space is formally captured by the log of the size of the branching program, i.e. the number of bits needed to keep track of where in the graph we are at each moment in time, we have more fine-grained notions that are relevant. We say the program is *layered* if the nodes can be arranged into layers, starting with the source at layer 0, such that the edges coming out of any node at layer i go to nodes at layer $i + 1$. In this case we can speak of time and space as being the length and width of the program, i.e. the number of layers and the largest size of any layer, respectively.

[4]Any activity lasting a half hour or more and which does not involve mathematics will do; I suggest a cup of tea and a nice book. However, as this strategy was originally taught to me as "the Gilligan's Island method" (by a professor whose age was betrayed therein) I have preserved it as such.

Theorem 1 (Barrington's Theorem)**.** *Let C be a circuit of depth d. Then there exists a layered branching program of length 4^d and width 5 computing the same function as C.*

Lemma 1 immediately gives Theorem 1 with width $2^3 = 8$ instead of 5 (as well as some loss in the length). It can also be used to prove a more general arithmetic statement, which was the original motivation and result of [BC92]. This leads in to one of the key ideas in the upcoming section, and so we encourage readers to attempt this generalization for themselves.

Exercise 2. *Let C be an arithmetic circuit of depth d, meaning a circuits whose inputs are from a ring $\mathcal{R}$ instead of $\{0,1\}$ and whose gates are $+$ and $\times$ over $\mathcal{R}$. Extend Theorem 1 to show that there exists a layered branching program of length 4^d and width $|\mathcal{R}|^3$ computing the same polynomial as C.*

3 A brief primer on catalytic computing

The proof in the previous section gave us a taste of how one could use full memory, represented in Lemma 1 by the arbitrary values τ_i in R_i, in a non-trivial way. We now broaden the scope of such ideas to discuss our central, although certainly not exclusive, model for testing the reuse of space: catalytic computation.

3.1 The basic definitions

We start with the definition of Buhrman et al. [BCK+14], who first introduced the concept of catalytic computation.

Definition 1. *A catalytic Turing Machine with space $s := s(n)$ and catalytic space $c := c(n)$ is a Turing Machine M with two read-write work tapes, which we call the work tape and the catalytic tape, which have lengths s and c respectively.*

In addition to the usual restrictions on space-bounded Turing Machines, M obeys the following additional property: for any $\tau \in \{0,1\}^c$, if we initialize the catalytic tape to τ, then on any input, M contains τ on the catalytic tape when it halts.

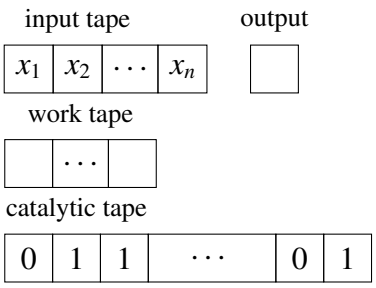

Fig.: catalytic Turing Machine

This definition gives rise to a natural complexity class, which is a variant of the ordinary class $\mathsf{SPACE}(s)$.

Definition 2. *The class* $\mathsf{CSPACE}(s, c)$ *is the set of all functions which can be computed by a catalytic Turing Machine which has space* s *and catalytic space* c*.*

$\mathsf{CSPACE}(s, c)$ sits between $\mathsf{SPACE}(s)$ and $\mathsf{SPACE}(s + c)$, with neither containment known to be strict in any interesting setting. Showing $\mathsf{CSPACE}(s, c)$ is strictly more powerful than $\mathsf{SPACE}(s)$ would be a validation of our ideas of reusing full space, even in the generic setting where we make no considerations of what the full memory actually contains.

The most well-studied variant of CSPACE is *catalytic logspace*, where s is logarithmic and c is polynomial. For most of this survey we frame our discussions around this class, but the reader should be aware that almost all results and problems we pose can be scaled up to pertain to other $\mathsf{CSPACE}(s, c)$ classes accordingly.

Definition 3. *The class* CL *is defined as* $\mathsf{CSPACE}(O(\log n), n^{O(1)})$.

Following our earlier discussion, CL sits somewhere between L and PSPACE. Since these two classes are not equal by the space hierarchy theorem, one of these two inclusions must be strict; current evidence, which we now turn to, suggests that in fact strictness holds for both containments.

3.2 Upper bounds

3.2.1 Compression: useful even when it fails

Let us disregard the techniques we saw in Section 2 and consider the definition of CSPACE at face value.

It is a natural knee-jerk reaction to think that $\mathsf{CSPACE}(s, c)$ must be approximately the same as $\mathsf{SPACE}(s)$ for any c. The only obvious approach to refuting such a statement would be applying some type of compression to the catalytic tape, and when one considers that such compression must succeed for any initialization of the catalytic tape—never mind the technical hurdles needed to implement such an approach—even this seems unlikely to help.

However, failure to compress such a string does not leave us with only free space s; it leaves us with free space s *plus* an incompressible string, hardly a trivial object to come by. Such a string may suggest many uses, but perhaps the first one that comes to mind is to use it as a source of entropy, i.e. randomness.

This insight can be pushed all the way through for simulating *randomized space*, due to a few peculiarities in how randomized space-bounded algorithms are defined. This proof [Lof] is unpublished as it was quickly subsumed by a very different argument, one which we turn to in the next subsection. We nevertheless reproduce it here because it is one of the few techniques which is known for catalytic space.

Theorem 2. BPL $\subseteq$ CL

Proof sketch. Let us recall the definition of BPL: these are the functions f for which there exists a logspace machine B taking in an input x and which can generate a polynomial amount of randomness r, such that for every x, $B(x)$ outputs $f(x)$ with probability at least $2/3$ over all choices of r. Furthermore, we have the crucial restriction that B can only read each bit of randomness once; any bits of r that it wishes to use in the future must be stored on the logspace work tape.

Let B be a BPL machine, fix an input x to B, and without loss of generality assume B reads a bit of its random tape at every time step during its execution. To start thinking about derandomizing B, we must ask what a random string r needs to look like in order to be useful to B.

Nisan [Nis93] provided the following test for a *collection* of strings R. Consider running B on each $r \in R$ up to a fixed time step i, and partition up R based on which configuration σ—meaning the contents of the work tape, location of all tape heads, etc—B ends up in at this step. Essentially, Nisan's condition is that no matter which i and σ we look at, the $i + 1$st bit should be fairly unbiased, i.e. roughly half 0 and half 1. This condition is checkable in logspace, and should it succeed then Nisan proves that a majority of strings in R will output the correct answer to $B(x)$.

We will let our catalytic tape be large enough that it can be broken into a collection R of candidate random strings. Using Nisan's criteria, we can use the normal work tape to check if the condition holds for each i and σ, and if it does then we simply run B on x using each $r \in R$ and take a majority vote. Note that in this case we never alter the catalytic tape, so we fulfill our additional requirement by default.

If this condition fails, then we can identify a timestep i and a configuration σ for which it fails. We let Γ be our set of strings from R which put B in configuration σ at step i, and by making $|R|$ sufficiently large we ensure that Γ has sufficiently many more strings with $b \in \{0, 1\}$ than $\neg b$ in the $i + 1$st location. Note that membership in Γ can be identified by running the BPL machine up to step i on our logspace worktape, and so by extension we can form a subtape T of the catalytic tape which exactly contains the $i + 1$st locations of strings in Γ, with this tape containing, without loss of generality, polynomially more zeroes than ones.

Because of the severe imbalance of T, we can compress it in place and free up a polynomial number of cells.[5] At this point we can simulate our BPL machine B in a brute force manner, by using the empty cells of T to try every possible random string and take a majority vote. Afterwards, we save the answer on the smaller

[5]We will not cover the details of this procedure. It is not difficult, but also not the style of argument we are concerned with in this survey.

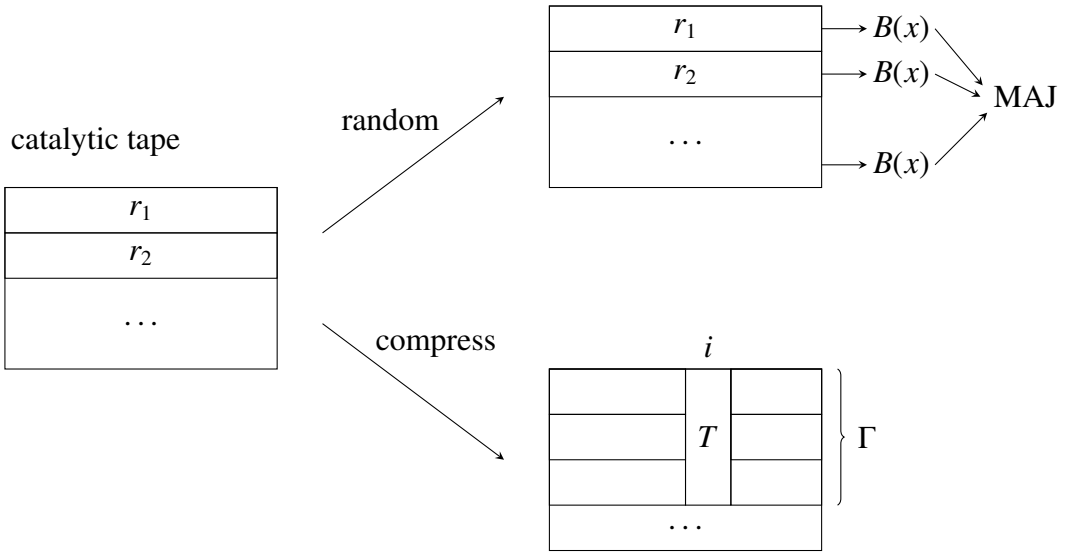

Fig.: compress-or-random argument

work tape, and before halting we run the inverse of our previous compression algorithm to return the catalytic tape to its original state. □

Throughout the rest of this survey, we refer to this proof as the *compress-or-random argument*. In thinking about the compress case, it seems like we have moved back from the goal of reusing space to that of simply saving space in the traditional sense. However, the real insight is coming from the random case, where the space that is being used for storage, i.e. the catalytic tape, is also being mobilized for a computational purpose.

3.2.2 Transparency and arithmetic

From the simple insight that incompressability gives us non-trivial power, we obtained a surprising use of catalytic machines. For this, we made essential use of the initial values stored in the catalytic tape. We now return to the ideas from Section 2 and take a different path to the power of full memory, one where we formally cancel out the contributions of the initialized catalytic tape without ever inspecting its values.

For this approach, we will define a toy model based on the statement in Lemma 1, in order to focus on the nature of our approach as both recursive and mathematical. Let $\mathcal{R}$ be a ring, which for our purposes is simply a set of values equipped with $+$ and $\times$ operations which act in an expected way, e.g. $x + 0 = x \cdot 1 = x$, $x \cdot 0 = 0$, are associative, etc.

Definition 4. *A register program P with space $s := s(n)$ and time $t := t(n)$ is comprised of a set of s blocks of memory, or what we will call registers, $R_1 \dots R_s$, each of which can hold a single value from $\mathcal{R}$, plus a list of t instructions, each of which updates a single register by adding to its current value some polynomial over all the other registers.*

$$R_i \leftarrow R_i + p(x_j, R_1 \dots R_{i-1}, R_{i+1} \dots R_s)$$

Often we will allow the program P to execute some other program P' in place of an instruction. In this case, we will usually keep two separate notions of time: the number of basic instructions and the number of recursive calls.

Clearly this definition captures the algorithms defined in Lemma 1; for example, for gate $g = g_1 \wedge g_2$, our program $P := P_g(0)$ used three registers over $\mathbb{F}_2$, four basic instructions, and four total recursive calls to the programs $P_1(1)$ and $P_2(2)$.

These programs had an essential characteristic which made them useful to recursion as well as the ultimate task of reusing space: they worked even when all the registers R_i were initialized to ring values τ_i, and at the end of the computation they left all but one register untouched. This was given explicit attention by Buhrman et al. [BCK+14] for its use in building recursive procedures.

Definition 5. *A transparent register program P is one in which all registers R_i are initialized to some value $\tau_i \in \mathcal{R}$. The result of the program is that for some register R_i,*

$$R_i = \tau_i + v$$

If our transparent register program further fulfills the property that

$$R_j = \tau_j \qquad \forall j \neq i$$

then we say it is a clean register program. In both cases we say that v is the value that P computes and R_i is the target register, and write

$$P: \quad R_i \leftarrow R_i + v$$

to indicate the function of P. We also occasionally say that P computes v into R_i.

We also assume that for every clean register program P there exists a clean register program P^{-1} such that

$$P^{-1}: \quad R_i \leftarrow R_i - v$$

for the same R_i and v as P.[6]

[6]This can be proven to exist by the definition of a register program—in essence, run the instructions in reverse order and flip all signs—and allows us to reset the one piece of memory left altered by a clean register program.

To put this definition to use in the catalytic setting, let us rephrase Lemma 1 in our new language.

Lemma 2. *Let P_1 and P_2 be clean register programs over $\mathbb{F}_2$ computing g_1 and g_2 into target registers R_1 and R_2 respectively. There exist clean register programs*

$$P_\neg : \quad R_0 \leftarrow R_0 + \neg g_1$$

$$P_\wedge : \quad R_0 \leftarrow R_0 + (g_1 \wedge g_2)$$

$P_\neg$ uses only the two registers R_1 and R_0, and makes one recursive call to P_1 plus two basic instructions. $P_\wedge$ uses only the three registers R_1, R_2, and R_0, and makes two recursive calls each to P_1 and P_2 plus four basic instructions.

Moving to the regime of CL allows us to consider a polynomial number of registers, rings of larger size, and so on. Thus we can ask what other functions can be computed by clean register programs using efficient time and space. For example, we could apply the principles of Lemma 1 at a larger scale to handle arbitrarily large products.

Exercise 3. *Let $P_1 \dots P_d$ be clean register programs over $\mathbb{F}_2$, where P_i computes g_i into register R_i for each i. Prove there exists a clean register program*

$$P_\wedge : \quad R_0 \leftarrow R_0 + \wedge_i g_i$$

where $P_\wedge$ uses only $O(d)$ registers and makes at most $2^{O(d)}$ recursive calls in total plus $2^{O(d)}$ basic instructions.

This exercise in fact has two solutions, both of which can be derived from a consideration of the program found in Lemma 1; one is both more straightforward and exponentially more efficient than the other, but we encourage readers to search for both, as the second solution has a separate practical utility.

Focusing on circuit models for which our recursively-structured register programs may be useful, the first stop above L would be AC^1, which contains unbounded fan-in ANDs, or perhaps VP, which contains unbounded fan-in + and fan-in two $\times$ over $\mathbb{Z}$. We will skip ahead to a much greater prize: unbounded fan-in majority.

Theorem 3. $\mathsf{CL} \supseteq \mathsf{TC}^1 (\supseteq \mathsf{NL})$

This statement is clear evidence of the power of catalytic computing; the catalytic tape captures the power of non-determinism, and likely much more.

TC^1 is defined as log depth circuits with unbounded fan-in majority gates, or, equivalently, log depth circuits with unbounded fan-in $g_{=\ell}$ gates for every ℓ, where $g_{=\ell}$ outputs 1 iff the number of 1-inputs is exactly ℓ. This characterization allows us to reduce Theorem 3 to a simple statement about register programs, plus a bit of care in the application.

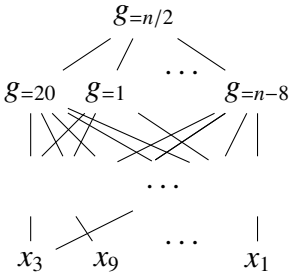

Fig.: TC^1

Lemma 3. *Let $m \in \mathbb{N}$ and let $p > m$ be a sufficiently large prime. For $i \in [m]$, let P_i be a clean register program over $\mathbb{F}_p$ computing*

$$P_i : \quad R_i \leftarrow R_i + v_i$$

Then for every $\ell \in [m]$ there exists a clean register program $P_{=\ell}$ which cleanly computes the indicator function

$$P_{=\ell} : \quad R_0 \leftarrow R_0 + \left[\sum_i v_i = \ell\right]$$

$P_{=\ell}$ uses $O(m)$ registers and makes eight total calls to each P_i plus $O(p)$ basic instructions.

We include an outline of proof below with a number of exercises for readers who want to get a feel for crafting register programs, although readers who want to see the full program for themselves will accordingly find it in the appendix.

Proof sketch. $P_{=\ell}$ works in two parts. First, we define program P_Σ which cleanly computes

$$P_\Sigma : \quad R_\Sigma \leftarrow R_\Sigma + \sum_i v_i$$

given the programs P_i. Second, given a program P_v which cleanly computes some value v into R_v, we define program P_k which cleanly computes

$$P_k : \quad R_0 \leftarrow R_0 + v^k$$

There are two straightforward tasks and one more difficult one. First, construction $P_{=\ell}$ given P_Σ and P_k is almost immediate from Fermat's Little Theorem; since $p > \sum_i v_i$:

$$\left[\sum_i v_i = \ell\right] \equiv 1 - \left(\ell - \sum_i v_i\right)^{p-1} \mod p$$

We first define our function P_v; as with Lemma 1 our notation is to list instructions on the left and their effect on memory—with brackets separating previous values and updates—on the right:

$$\begin{aligned}
&1.\quad P_\Sigma^{-1} && R_\Sigma &&= [\tau_\Sigma] - \left[\sum_i v_i\right] \\
&2.\quad R_\Sigma \mathrel{+}= \ell && R_\Sigma &&= \left[\tau_\Sigma - \sum_i v_i\right] + [\ell] \\
& && && = \tau_\Sigma + \left(\ell - \sum_i v_i\right)
\end{aligned}$$

and thus with P_v being our subroutine to P_k—using $v = \ell - \sum_i v_i$ and $R_v = R_\Sigma$—and choosing $k = p - 1$, the following program computes $P_{=\ell}$:

$$\begin{aligned}
&1.\quad P_{p-1}^{-1} && R_0 &&= [\tau_0] - \left[\left(\ell - \sum_i v_i\right)^{p-1}\right] \\
&2.\quad R_0 \mathrel{+}= 1 && R_0 &&= \left[\tau_0 - \left(\ell - \sum_i v_i\right)^{p-1}\right] + [1] \\
& && && = \tau_0 + \left(1 - \left(\ell - \sum_i v_i\right)^{p-1}\right) \quad \checkmark
\end{aligned}$$

which completes our program by the previous equation and the fact that our program is over $\mathbb{F}_p$.

Now we need only construct P_Σ and P_k, both of which we leave as exercises. P_Σ is almost immediate and should not be overthought.

Exercise 4. *Construct P_Σ making one call to each P_i and one call to each P_i^{-1}.*

P_k is a bit trickier, but it follows nicely from the following observation:

$$v^k = (\tau_v - (\tau_v - v))^k = \sum_{j=0}^{k} \binom{k}{j} (\tau_v)^j (-1)^{k-j} (\tau_v - v)^{k-j}$$

The utility of this equation is that we always have access to either τ_v (at the start of the program) or $\tau_v - v$ (after running P_v^{-1}). This is a variant of Lemma 1, but it takes a bit of clever thinking, plus using some external memory besides R_v and R_0.

Exercise 5. *Construct P_k making one call each to P_v and P_v^{-1}.*

When all is said and done there is a solution using $m + p + 2 = O(m)$ registers, eight calls to each program P_i or its inverse, and roughly $2p + 12 = O(p)$ basic instructions, although anything in this ballpark works fine. □

We refer to this and similar proofs as *register program arguments*. Such arguments are at the forefront of our knowledge with regards to catalytic computation, as we have not proven a stronger result using compress-or-random up to this point.

3.3 Lower bounds

3.3.1 The average catalytic tape

We began our discussion of the power of catalytic computation with the observation that even an incompressible tape can be useful. We now turn to the other side of the same coin: the average catalytic tape is incompressible. This has a surprisingly simple implication for the runtime of our machines.

Theorem 4. $\mathsf{CL} \subseteq \mathsf{ZPP}$

If Theorem 3 gives a strong indication that CL is strictly more powerful than L, Theorem 4 is an even stronger signal that CL is exponentially weaker than PSPACE.

Proof sketch. Recall the argument that $\mathsf{L} \subseteq \mathsf{P}$: there are at most $2^{O(\log n)} = \mathsf{poly}\, n$ possible machine configurations, and if any such configuration ever repeats then they must repeat ad infinitum, a contradiction.

Despite having an exponential number of configurations, the same argument applies to CL, albeit only in an average sense. Fix an input x, and consider the configuration graph of our CL machine; rather than a single line of polynomial length emanating from the all zeroes starting node, as in the case of L, there are as many lines as there are starting configurations τ of the catalytic tape, a number which we call m. By extension, the total number of configurations is at most $m \cdot 2^{O(\log n)}$.

One further observation is needed, and it comes from the catalytic restoration property: no two lines coming from different initial catalytic tapes may cross, for if they did then at most one of the two intersecting paths can correctly reset its catalytic tape. Thus the *average* length of a computation path is poly n as before.

Hence we can simulate our CL machine by a randomized P machine by choosing a random catalytic tape and running for a polynomial number of steps. The zero errorness, i.e. inclusion in ZPP rather than BPP, follows because the machine never errs; we declare failure only if the machine takes too long. □

Note that the above argument does not give us a guaranteed poly time bound on the runtime of a catalytic algorithm. Thus it is unknown whether $\mathsf{CL} \subseteq \mathsf{P}$, and Theorem 4 gives us little guidance in solving this problem, as coming up with random strings seems as hard as derandomizing polynomial time classes themselves.

3.3.2 Reversibility

As with many upper bounds against space-bounded classes, Theorem 4 prompts us to look at the structure and size of the configuration graph of a catalytic machine. There is a nice extension of the observation that such graphs, after fixing the input, are simply collections of lines, which is a *reversibility* property. Reversibility as a technique developed across many papers, most notably those by Bennett [Ben73, Ben89]; we refer the interested reader to the catalytic survey by Koucký [Kou16] for an historic overview.

For our purposes, this is not so much a direct lower bound as a tool used in results similar to Theorem 4, which we will discuss more in the next section. It also makes formal the intuition that a catalytic machine must undo all its work at the end of a computation. This version of the proof originally appeared in unpublished work by Dulek [Dul] and was later proven by Datta et al. [DGJ+20].

Theorem 5. *Let C be a deterministic catalytic machine. Then there is another deterministic catalytic machine C', computing the same function and using the same amount of work and catalytic space as C, such that at any point in the execution of C' on some input x, there is both a unique forward instruction and a unique backward instruction.*

We note that while the theorem of Lange et al. [LMT00]—from which the proof of Theorem 3.3.2 was adapted—allows us to make any Turing Machine reversible, we only know how maintain efficiency with respect to space; it is unknown, for example, how to make a $\mathsf{TIME}(t)$ machine reversible in anything less than time $\exp t$.

Proof sketch. In the proof of Theorem 4 we noted that after fixing an input x, no two configuration paths coming from different initial catalytic tape configurations can ever meet; hence we called such paths "lines". If the configuration graph was truly a collection of lines then we would be done; at every state there is at most one way forward and one way back.

This view is almost correct, but slightly off: there may be configurations unreachable from any start state but that nevertheless hang off the side of a path; namely, if we were given such an illegal configuration and were asked to take

a step forward, we may end up on a legal path. This is no issue when running forwards, but causes some concern when running in reverse.

The solution is to have our new machine C' take an Eulerian tour around the configuration graph of C. Hence as it travels along such forward lines, it may in fact stray into a branch of illegal configurations, but eventually it will work its way around and make it back to the original path. The key point is that it will never reach any state with the wrong answer—we start by altering C to make sure all accept/reject states have outdegree zero—nor any other path coming from a different initial catalytic tape by our earlier discussion.

The details of how to modify the transitions of C to allow this tour to happen, and how to use the right hand rule to avoid infinite loops, are mostly technical curios which we omit. Recall that for our deterministic machine C each transition only relies on a constant amount of information. Some other small tweaks to C may be in order. □

The upshot is that without loss of generality we can assume a catalytic machine runs forward until it discovers the output, then switches direction and runs the same algorithm in reverse until it returns to the beginning.

3.4 Variants of CSPACE

In defining a basic catalytic space-bounded computation class, we invariably have opened the door to a whole parallel complexity hierarchy, both of the basic CSPACE classes and of augmentations therein. We mention some of these variants and their highlights now.

3.4.1 Choices of s and c

In this survey we mostly focus on the case where $s = \log c$, but this is not the only possible consideration. Bisoyi et al. [BDS22] consider many different regimes, from the *high end* where $s = c^{\epsilon}$, to the *low end* where $s = \log\log c$ or even $s = O(1)$, which they call CR since it corresponds to regular languages with catalycity. Their investigation was preliminary, and so we do not discuss these results in more detail, but the low end regime will come back in some form during our discussion of branching programs in Section 6.

3.4.2 Randomized and non-deterministic computation

Two fundamental resources, both of which we have already seen in the classic space-bounded setting, are randomness and non-determinism. The randomized class CBPL was introduced by Datta et al. [DGJ+20] in relation to Theorem 5,

while CNL was defined by Buhrman et al. [BKLS18] in a follow-up to their original work.

In both cases there is an immediate question to be answered: when does the catalytic tape need to be reset? The answer given by both [DGJ+20] in the randomized case and [BKLS18] in the non-deterministic case is the safe one: the catalytic tape must always be reset, whether or not the correct answer is returned.

So far we do not have many results about the additional power of either CBPL or CNL. Structurally we have a few results in the non-deterministic world that mirror the traditional space-bounded setting, such as a (conditional) catalytic Immerman-Szelepcsényi Theorem [Imm88, Sze88], i.e. CNL = coCNL [BKLS18], and a conditional reduction of CNL to its unambiguous variant CUL [GJST19].

One other note about both models is that because we have no explicit runtime restrictions, our catalytic machines are allowed to use a superpolynomial amount of randomness or non-determinism as they so choose. It is then quite surprising that using both our upper bound techniques in tandem, namely average catalytic tapes and reversibility, gives us a way to upper bound both classes in ZPP just as with CL.

Exercise 6. *Use ideas from Theorems 4 and 5 to show that* CBPL *and* CNL *are contained in* ZPP.

Similar arguments can show various other results, such as 1) CBPL is contained in CZPL if we are allowed to read the randomness twice in the latter case; or 2) CL gains no power when we allow it to err during the resetting of the catalytic tape in $O(1)$ spots.

3.4.3 Non-uniform computation

One nice aspect of space complexity is that it is syntactically captured by the branching program model, which in particular gives a straightforward way to think about non-uniform computation. How does this model translate to the catalytic world?

Girard et al. [GKM15] define catalytic branching programs in the following way: rather than having a single start node and going to two different end nodes, we have m different start nodes, each with their own label τ, and $2m$ end nodes which are each labeled with both an output to the function and a start node τ. The catalytic property, naturally, states that on input x, each start node τ must reach exactly the end node labeled with $f(x)$ and τ.

If the program has size $s \cdot m$, we can think of this non-uniformly computing CSPACE$(\log s, \log m)$: we use $\log sm$ bits to remember the current node, but $\log m$ of which are set at the start and must be reset at the end.

Notice that we no longer need to address into the catalytic tape using the work tape, and so unlike with CSPACE it makes sense to talk about m as being much greater than 2^s. In fact, Potechin [Pot17] showed that for $m = 2^{2^n}$, $s = O(n)$ is sufficient for *any* function f, a non-uniform $\mathsf{CSPACE}(\log n, 2^n) = \mathsf{ALL}$ theorem.

Exercise 7. *Let $P_1 \dots P_n$ be clean register programs where P_i computes x_i into R_i for each i. Show that for any function $f(x_1 \dots x_n)$ there exists a clean register program P_f computing f into R_0, where P_f uses $2^{O(n)}$ registers and makes $O(1)$ recursive calls to each P_i.*

This is optimal in terms of "work space" s, while follow up works of Robere and Zuiddam [RZ21] and Cook and Mertz [CM22] have improved on the "catalytic space" m needed as well.

In the spirit of connecting catalytic computation to broader questions in complexity, we mention that Potechin [Pot17] also made a nice connection of catalytic branching programs to a notion of amortized space-bounded complexity. For a catalytic branching program B with m start nodes of size $s \cdot m$ computing f, we can think of s as being the average size of a branching program needed to compute f, where the averaging is over the m "different branching programs" (one start node, two end nodes) for f embedded inside B. Thus the previous results also show that the amortized branching program size of any function is linear, and the question remains how much amortization is necessary to achieve this.

3.5 Afterword: eyes on the prize

We take a step back once again before moving to the open problems. We have now introduced catalytic computation, with catalytic Turing Machines and a set of complexity classes, plus some results.

Without any prompting from us, readers to whom this definition appeals can begin to form a network of open problems for themselves. We will leave one last exercise to these readers to start them on their way.

Exercise 8. *Pick your favorite function inside* TC^1—STConn, $\bigoplus$, Tribes,...—*and give a* CL *algorithm for it directly. You can use compress-or-random, register programs, or anything else you like, as long as you use the catalytic tape in an interesting way.*

However, our goal is ultimately to study reusing space, and so we turn back to the reader who is interested in techniques and applications to classical space-bounded classes rather than this new exotic definition. To reiterate: catalytic computation is a test bed for how to use memory both as storage and as computation simultaneously. Thus, for example, when we say

"problem x is computable in catalytic logspace"

we may take this to mean a variety of things aside from what is stated. Practically it could mean

"if we need to compute x as a subroutine many many times to compute problem y, the space to do so need not compound linearly"

or in terms of studying x itself it could mean

"while x may require a lot of space, it can be quite well-structured with regards to its space usage"

et cetera. Similarly, if we show that problem x is in, for example, CBPL or CNL, this means that a space-bounded algorithm with access to randomness or non-determinism can implement space reuse techniques to compute x as above.

Any statement about catalytic computation, even structural results, may be looked at in this way, and we encourage readers to interpret catalytic questions and results in whatever light is most useful.

Part II
WHAT WE DON'T KNOW (YET)

We now move to our second, and main, purpose in this work: a curated list of some of the open questions in the field. Below is the complete list of problems, presented in the order in which they will appear.

Most of the problems we state will rely on terminology introduced in the previous sections, but some will need some definitions later; in either case we endeavor to keep the statements themselves at a high level, and we then expound upon each of them in rough detail in the remainder of the survey. Readers who would prefer to see the problems in their proper context are free to skip to Section 4.

We make no attempt to segregate them by perceived difficulty or concreteness, but we provide this loose table of contents for readers who have a certain type of question in mind, or alternatively for those who want to get a broad sense of what questions are being asked before diving into specifics.

The list of problems

1. Give a simple, direct proof of $\mathsf{uSTConn} \in \mathsf{L}$.

2. Give a simple, direct proof of $\mathsf{uSTConn} \in \mathsf{CL}$.

3. Give a simple, direct proof of $\mathsf{STConn} \in \mathsf{CL}$.

4. Try to improve Savich's Theorem: prove $\mathsf{NSPACE}(s) \subseteq \mathsf{SPACE}(o(s^2))$.

5. Improve the deterministic space complexity of $\mathsf{BPSPACE}(s)$.

6. Decide the space complexity of $\mathsf{TreeEval}$.

7. Give a register program for computing any polynomial $p(x_1 \dots x_n)$ using $O(n)$ registers over a constant size ring $\mathcal{R}$ and $O(1)$ recursive calls to the input x.

8. Show that for any branching program B of sufficiently large width $w = \Omega(1)$ and length ℓ, there exists a branching program B' of width $w/2$ and length $O(\ell)$ computing the same function.

9. Show that for any branching program B of sufficiently large width w and length ℓ, there exists a branching program B' of width $w - 1$ and length $\mathsf{poly}(\ell)$ computing the same function.

10. Find any function whose optimal space algorithm can be made almost entirely catalytic, i.e. a function requiring—or even that we only know how to do in—$\mathsf{SPACE}(s)$ but which is computable in $\mathsf{CSPACE}(\ll s, \approx s)$.

11. Prove $\mathsf{CL} \subseteq \mathsf{P}$.

12. Show that $\mathsf{P} \not\subseteq \mathsf{L/poly}$ implies $\mathsf{CL} \subseteq \mathsf{P}$.

13. Show that $\mathsf{CL} \subseteq \mathsf{P}$ would give strong evidence $\mathsf{ZPP} \subseteq \mathsf{P}$.

14. Show that NC^2, or even any circuit of $\omega(\log n)$ depth, can be computed in CL.

15. Give a register program for computing x^k in the non-commutative setting using linear space and a constant number of recursive calls to x.

16. Show that $\mathsf{BPNC}^1 \subseteq \mathsf{CL}$.

17. Design a catalytic branching program with $2^{O(n)}$ start nodes and total size $2^{O(n)} \cdot O(n)$ for any function f.

18. What is the power of $\mathsf{CL}/poly$, and does it have a natural syntactic characterization?

19. Show the existence of an oracle D such that $\mathsf{CL}^D = \mathsf{EXP}^D$.

20. Extend the $\mathsf{BPL} \subseteq \mathsf{CL}$ simulation to show $\mathsf{CBPL} \subseteq \mathsf{CL}$.

21. Show that CL is equivalent even if we allow $\omega(1)$ many errors on the catalytic tape at the end, or alternatively if we allow $O(1)$ such errors in expectation over all inputs x and catalytic tapes τ.

22. Utilize non-determinism in conjunction with catalytic computing in a non-trivial way.

23. Prove $\mathsf{CNSPACE}(s, c) \subseteq \mathsf{CSPACE}(s^2, c^2)$.

24. Implement a catalytic algorithm such that it is actually useful.

25. What does quantum catalytic space look like?

26. Devise a register program using basic instructions inspired by unitary computation, and use it to show non-trivial results for e.g. BQP.

27. Devise a circuit that uses known results from space reuse and catalytic computing to efficiently solve some problem in a way that we do not know how to do directly.

28. Show $\mathsf{TC}^1 \subseteq \mathsf{VP}$.

29. Is the network coding conjecture true or false?

30. Prove or disprove the network coding conjecture when all nodes are restricted to sending linear transformations of their incoming messages.

31. Is there a meaningful notion of a catalytic data structure, or is there anything to be gained from a data structure stored in catalytic memory?

32. Show CL is contained in some subclass of P, perhaps NC, given a believable cryptographic assumption.

33. Show evidence against objects in cryptography based on techniques in reusing space.

34. Show the existence, conditional or otherwise, of a natural class of cryptographic objects by using clean computation.

35. Prove that the existence of one-way functions in CL, or even any one-way function computable by a poly-size poly-length register program, implies the existence of one-way functions in NC^0.

4 What can be done with reusing space?

We first turn our attention to questions in pure (i.e. non-catalytic) space-bounded complexity. Most of these problems will be well-known to the reader, and our only exhortation is to turn the tools seen in this survey upon them for a fresh look.

4.1 Connectivity

Reingold's brilliant result [Rei08] shows that $\mathsf{uSTConn} \in \mathsf{L}$, thus resolving the connection between logspace and symmetric logspace. The algorithm uses tools such as zig-zag product, which are pretty heavy hitting and incur large losses in both time and space. This is in contrast to e.g. the standard RL algorithm, which solves $\mathsf{uSTConn}$ efficiently with very high probability simply by taking a random walk on the graph.

Can reusing space be used to give a simpler, more efficient deterministic algorithm for $\mathsf{uSTConn}$? This would be a useful addition to our study of space-bounded complexity.

Problem 1. *Give a simple, direct proof of* $\mathsf{uSTConn} \in \mathsf{L}$.

Perhaps less ambitiously, consider the case of CL. Both compress-or-random and register programs are enough to show uSTConn $\in$ CL, but besides being fairly lossy in the constants, as of now neither technique can be nicely described in terms of uSTConn itself.

A simple, efficient, and clear algorithm for uSTConn $\in$ CL would be a useful way of illustrating the counterintuitive power of catalytic space to newcomers in the field, as well as potentially providing an angle on Problem 1.

Problem 2. *Give a simple, direct proof of* uSTConn $\in$ CL.

We also know that NL $\subseteq$ CL, meaning we can drop the undirected restriction and still have a CL algorithm for connectivity via register programs. If Problem 2 can be solved, we can hope it also is amenable to such a change.

Problem 3. *Give a simple, direct proof of* STConn $\in$ CL.

4.2 Savitch's Theorem

Taking a space reuse-style approach to STConn may also give us insights on one of the major unsolved questions in structural space complexity. Savitch's Theorem [Sav70], which states that NSPACE(s) $\subseteq$ SPACE($O(s^2)$), is one of the bedrocks of space complexity, but it is not known to be tight. After over fifty years, it may be time to seriously revisit this question, with reusing space being one of the new tools in our arsenal.

Problem 4. *Try to improve Savitch's Theorem: prove* NSPACE(s) $\subseteq$ SPACE($o(s^2)$).

If Problem 2 gives us a way to solve Problem 1, then perhaps Problem 3 will point the way to attacking Problem 4 in similar fashion.

4.3 Derandomizing space

Derandomizing BPL is a longstanding open problem with a flurry of recent work. There is a wide pool of techniques to draw from, including targeted and weighted PRGs, approximate matrix inversion, certified derandomization, and more; we refer readers to an excellent survey by Hoza [Hoz22] on the topic. Perhaps our techniques will be useful as well.

Problem 5. *Improve the deterministic space complexity of* BPSPACE(s).

We state Problem 5 less specifically than other problems in this survey because it is an active line of research, and so any target we lay out may be obsolete by the time the reader reaches this survey, and for reasons having nothing to do with reusing space. For example, Hoza [Hoz21] recently improved on the best known upper bound of SPACE($s^{3/2}$), due to Saks and Zhou [SZ99], for the first time in thirty years.

4.4 The Tree Evaluation Problem

Another key question in the study of space-bounded computation is how it compares to time-bounded computation. One central question is whether logspace is strictly contained in polynomial time or not.

Cook et al. [CMW+12] proposed that a function known as the Tree Evaluation Problem, or TreeEval for short, may be the key to separating L from P. The function is defined, for an alphabet size k and height h, by a height h rooted full binary tree, where leaves are given values in $[k]$ and internal nodes are labeled with functions from $[k] \times [k]$ to $[k]$. In other words, it is a sort of alternate circuit model where the values come from a broader alphabet than just $\{0, 1\}$, and the topology is fixed but the functions at each gate are given as input.

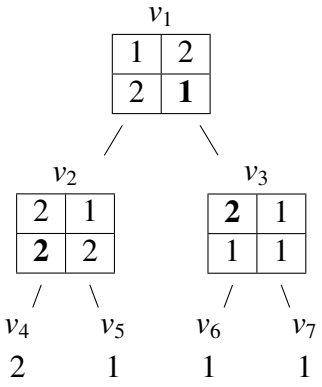

Fig.: TreeEval

Problem 6. *Decide the space complexity of* TreeEval.

The logic for thinking TreeEval $\notin$ L relies exactly on not being able to use space both for memory and for computation. Building on ideas we have seen previously, and in particular a generalization of Lemma 1 first to larger products (see Exercise 3) and then to arbitrary polynomials, Cook and Mertz [CM20, CM21] gave an algorithm for computing TreeEval more efficiently than Cook et al. conjectured. An optimal version of their key register program remains open, and would be sufficient to show TreeEval $\in$ L for all values of k and h.

Problem 7. *Give a register program for computing any polynomial $p(x_1 \dots x_n)$ using $O(n)$ registers over a constant size ring $\mathcal{R}$ and $O(1)$ recursive calls to the input x.*[7]

4.5 The power of formulas?

Barrington's Theorem allows us to characterize the class NC^1 of all polynomial size formulas as functions computable by branching programs of polynomial length and constant width. This is in contrast to L, whose branching programs can be polynomial in both length and width.

Problem 6 has been posed in the world of formulas as the *KRW conjecture* [KRW95], a depth-based composition theorem that states that no "depth reuse"

[7] We are assuming a batched version of the type of access to the input as in Lemma 1, i.e. a program P_x (as well as its inverse) whose effect is to add input x_j to register R_j for every j.

in the vein of our results should be possible. Resolving the KRW conjecture in the affirmative while also showing $\mathsf{TreeEval} \in \mathsf{L}$ would separate L from NC^1, an extremely fine-grained separation.

On the other hand, we have seen the surprising power of constant space, and so it is entirely possible that not just L but even NC^1 can implement our techniques. In fact, by the characterizations laid out above, a length-width tradeoff for branching programs would be sufficient.

Problem 8. *Show that for any branching program B of sufficiently large width $w = \Omega(1)$ and length ℓ, there exists a branching program B' of width $w/2$ and length $O(\ell)$ computing the same function.*

Problem 8 would show $\mathsf{NC}^1 = \mathsf{L}$. We note that a Savitch-style argument may work if we relax the length requirement to $\mathsf{poly}(\ell)$, which would only reprove that $\mathsf{L} \subseteq \mathsf{NC}^2$. We formulate an even weaker version of the question just to get the ball rolling.

Problem 9. *Show that for any branching program B of sufficiently large width w and length ℓ, there exists a branching program B' of width $w - 1$ and length $\mathsf{poly}(\ell)$ computing the same function.*

In Section 7 we return to the question of other models that may be able to implement our techniques.

5 Where does catalytic fit in to complexity theory?

Moving from catalytic techniques to catalytic computing itself, the most impactful questions remain those relating catalytic computing to more traditional complexity classes. While we already have a reasonably narrow range where classes such as CL can possibly sit, there are many important details to be resolved, as well as orthogonal questions about the use of catalytic computing.

5.1 Space versus catalytic space

At first glance, [BCK+14] settles the question of CSPACE in relation to SPACE once and for all. On one hand, for those that believe that $\mathsf{L} \neq \mathsf{NL}$, there are many natural classes sitting inside $\mathsf{CSPACE}(s, 2^s)$ and outside $\mathsf{SPACE}(s)$. Even more widely believed is that $\mathsf{ZPP} \neq \mathsf{PSPACE}$; in fact, it seems dubious that CL can even contain *any* class $\mathsf{SPACE}(s)$ where $s = \omega(\log n)$, as such classes are widely believed to be separate from P.

Yet if we put aside entire complexity classes for a moment, we can still look to individual problems and ask concrete questions about what catalytic space can

offer. Perhaps the most natural such question is to ask an instance-wise version of CL versus PSPACE.

Problem 10. *Find any function whose optimal space algorithm can be made almost entirely catalytic, i.e. a function requiring—or even that we only know how to do in—*$\mathsf{SPACE}(s)$ *but which is computable in* $\mathsf{CSPACE}(\ll s, \approx s)$.

This question, while interesting in its own right, is not a mere curio. Recently Doron and Tell [DT23] gave derandomization with almost no memory overhead conditioned on a few natural assumptions, but to get the optimal result requires a function computable by $\mathsf{CSPACE}(\epsilon s, s)$ but not by $\mathsf{SPACE}((1-\epsilon)s)$; an affirmative answer to our problem would also make this assumption hold unconditionally by padding. Note that here "almost entirely catalytic" means we cannot tolerate even a factor of 2 in the simulation; this is truly $\mathsf{CSPACE}(s, c)$ versus $\mathsf{SPACE}(s + c)$.

5.2 The catalytic holy grail: CL versus P

Problem 11. *Prove* $\mathsf{CL} \subseteq \mathsf{P}$.

Throughout this survey I have (hopefully without confusion) used hyperbole for stylistic flair, and calling any problem the "holy grail" of a nine-year-old field is as blatant as hyperbole can be. Nevertheless, resolving the relationship between CL and P has proved as fascinating as tenacious, and I can think of no better way to study the structure of catalytic machines than to attempt to resolve Problem 11.

For starters, even a conditional result—short of derandomizing ZPP, of course—could be very useful. It would be interesting to go outside the realm of derandomization in general, or at least to start from an earlier point, such as a novel use of the hardness-versus-randomness paradigm (see e.g. Pyne et al. [PRZ23] for a discussion of such techniques in the space-bounded setting).

Problem 12. *Show that* $\mathsf{P} \not\subseteq \mathsf{L/poly}$ *implies* $\mathsf{CL} \subseteq \mathsf{P}$.

On the flip side, there may be barrier results that make the proof difficult even for those who believe $\mathsf{ZPP} = \mathsf{P}$. Buhrman et al. [BCK+14] made initial progress on the relativization barrier, showing oracles A and B such that $\mathsf{CL}^A = \mathsf{PSPACE}^A$ and $\mathsf{NL}^B \not\subseteq \mathsf{CL}^B$. Perhaps a more direct barrier result is possible.

Problem 13. *Show that* $\mathsf{CL} \subseteq \mathsf{P}$ *would give strong evidence* $\mathsf{ZPP} \subseteq \mathsf{P}$.

5.3 The power of CL

For those more interested in the power of catalytic computing, the frontier of CL stands at Theorem 3. There is a long way to go even for those who believe $\mathsf{CL} \subseteq \mathsf{P}$.

5.3.1 Going up

The next clear challenge is going beyond logarithmic depth. Consider that when studying alternate problems such as Tree Evaluation, the register program technique has no obvious way of moving beyond objects of logarithmic depth, as recursively computing and uncomputing the inputs to a subroutine leads to runtime costs which are exponential in the recursion depth.

NC circuits contain only fan-in two AND and OR gates—Lemma 1 is meant to handle these circuits specifically—and so looking at $\log^2 n$ depth NC circuits allows us to focus on overcoming the depth barrier. Moving beyond logarithmic depth in any capacity would then allow us to consider bootstrapping or other such techniques.

Problem 14. *Show that* NC^2*, or even any circuit of* $\omega(\log n)$ *depth, can be computed in* CL.

There is one proposition to Problem 14 that skirts around the difficulty of higher depth: compress many layers into one, and then handle the resulting functions directly using register programs as before. For example, using the fact that NC^1 is contained in L, for which matrix powering is complete, an extension of Lemma 3 to the non-commutative realm could be sufficient to prove $\mathsf{NC}^2 \subseteq \mathsf{CL}$.

Problem 15. *Give a register program for computing* x^k *in the non-commutative setting using linear space and a constant number of recursive calls to* x*.*

To connect this to an earlier problem, all state-of-the-art approaches to Problem 6—i.e. register programs for arbitrary polynomials—work in the non-commutative setting as well, and thus optimal improvements therein may yield non-commutative powering as a special case.

5.3.2 Orthogonal improvements

There are other classes we could study in relation to CL, not linearly up from TC^1 but still interesting. One odd gap in our understanding is that of read-multiple randomness. It is well-known that while $\mathsf{NC}^1 \subseteq \mathsf{L}$, the same is not known for their randomized variants, because BPL has the restriction of only reading its random bits once. This is also the key to using Nisan's argument as it appeared in Theorem 2, and thus the following question is still open.

Problem 16. *Show that* $\mathsf{BPNC}^1 \subseteq \mathsf{CL}$.

If we treat the catalytic tape as a potential source of randomness à la Theorem 2, we can in fact read these "random" bits as many times as we need to compute the circuit. Thus Problem 16 boils down to understanding what properties of randomness are enough to fool circuits, and how to compress when these properties are not met.

5.4 Non-uniform catalytic computation

Moving to the non-uniform setting, we are free from many of the restrictions on CL that we previously faced. While many individual problems can be posed, it seems likely that non-uniform CL is universal even for the strictest setting of parameters, i.e. linear catalytic space and a log space work tape free of constant multipliers. This would obviate most other results that could be proposed.

Problem 17. *Design a catalytic branching program with $2^{O(n)}$ start nodes and total size $2^{O(n)} \cdot O(n)$ for any function f.*

As with Problem 15, this would follow directly from an optimal improvement to Problem 7.

However, we should note that calling this class "non-uniform CL" is somewhat of a misnomer. If we consider the class CL/poly , the equivalence of syntactic models and advice breaks down, as it is not at all obvious that this can capture such a class of exponential-sized branching programs; in fact considering $\mathsf{CL} \subseteq \mathsf{ZPP}$ and $\mathsf{ZPP/poly} = \mathsf{P/poly} \neq \mathsf{ALL}$, these are clearly not equivalent modulo Problem 17. It is worth understanding the actual non-uniform CL in its own right.

Problem 18. *What is the power of* CL/*poly, and does it have a natural syntactic characterization?*

5.5 Oracle results

As mentioned briefly before, Buhrman et al. [BCK+14] also give a few oracle results that complicate our potential attempts to resolve e.g. Problem 11. Oracle results in the world of space-bounded computation are notoriously finicky, and so they may not end up being of much use in the end, but for now it is useful to increase our understanding of catalytic computation.

One result we mentioned previously was an oracle A such that $\mathsf{CL}^A = \mathsf{PSPACE}^A$; the oracle in question is essentially an optimal compress-or-random object, which takes in a string w and either gives a compression of w if it has low entropy or the answer to a PSPACE-complete problem if it has high entropy. This kind of "password oracle" is similar to the one showing $\mathsf{ZPP}^B = \mathsf{EXP}^B$.[8] It seems plausible that the two approaches can be merged into one.

Problem 19. *Show the existence of an oracle D such that $\mathsf{CL}^D = \mathsf{EXP}^D$.*

[8] This is commonly attributed to Heller [Hel86], but according to Morgan Shirley the result as stated is actually unpublished and only appeared during a conference talk for a related paper. Very similar results and proofs do however exist in the literature, and it has been subsumed by Beigel et al. [BBF98]. My thanks to him for finding this information as well as the proof itself.

Note that this would make resolving Problem 11 much more difficult, as it would give an oracle with respect to which $\mathsf{CL}^D \neq \mathsf{P}^D$ by the relativized time hierarchy theorem. In particular, this would imply that any attempt to resolve Problem 11 must be non-relativizing.

6 Structural catalytic complexity

The second set of questions we have regarding catalytic computing is the structural complexity of catalytic computing itself, as a parallel to the hierarchy of traditional logspace-bounded classes. Since [BCK+14] introduced catalytic computation, this structural theory has seen the most exposition, and so there are many possible questions to study. We again refer the reader interested in catalytic computation in its own right to the survey of Koucký [Kou16].

6.1 Randomness

6.1.1 Derandomization

Problem 5 posed the question of derandomization for space-bounded complexity classes. We can also ask a catalytic version: if we cannot solve derandomize BPL into L, can we at least do the same with their catalytic variants?

Problem 20. *Extend the* $\mathsf{BPL} \subseteq \mathsf{CL}$ *simulation to show* $\mathsf{CBPL} \subseteq \mathsf{CL}$.

In the catalytic world, we have the compress-or-random argument from Theorem 2 in addition to all the tools coming from the study of BPL. This technique is worth regarding for Problem 20 for at least two reasons: 1) it is likely that variants of the argument can accommodate other derandomization techniques in tandem; and 2) considering CL has a catalytic tape itself, working against BPL with a catalytic tape may not be much harder than working against BPL.

Recall that $\mathsf{CBPL} \subseteq \mathsf{ZPP}$, meaning that we can remove the two-sided error from CBPL; in fact the argument only used randomness to pick an initial catalytic tape. [DGJ+20] Thus if CL can find a "good" catalytic tape, we are already done. However, there is an exponentially large configuration graph to consider if we pick the wrong tape. Furthermore our compress-or-random argument now has to work for random tapes that are useful against CBPL rather than CL, a harder condition to quantify.

6.1.2 Robustness

There is a related question, which is the robustness of the catalytic definition to small errors. Every bit of the catalytic tape forgotten is a bit that can be used

for free memory, and so we cannot expect CL to remain the same with too many errors, but as of now we only know how to recover from a constant number of errors on the catalytic tape[9], while intuitively even a logarithmic number should hardly be earth-shattering. An alternative result would be to show that CL can recover from only having a constant number of errors on average, rather than for every input and catalytic tape.

Problem 21. *Show that* CL *is equivalent even if we allow* $\omega(1)$ *many errors on the catalytic tape at the end, or alternatively if we allow* $O(1)$ *such errors in expectation over all inputs* x *and catalytic tapes* τ*.*

6.2 Non-determinism

Despite our structural results in the non-deterministic setting, we have no natural problems which have been placed in CNL but that are not known to be in CL. Right now non-deterministic catalytic computation seems to be a definition in search of an application.

Problem 22. *Utilize non-determinism in conjunction with catalytic computing in a non-trivial way.*

As a sign of our paucity of knowledge with regards to CNL, we do not yet have an equivalent of Savitch's Theorem in the catalytic world. This is one of the most fundamental questions remaining in importing the structural theory space complexity to the catalytic computing world.

Problem 23. *Prove* $\mathsf{CNSPACE}(s, c) \subseteq \mathsf{CSPACE}(s^2, c^2)$.

As in Problem 20, the tableau method of Savitch's original proof will result in an exponential number of configurations being considered, which seems to be the major obstacle. Even putting this aside, however, such a tableau of configurations of the original CNSPACE machine would have to be kept on the catalytic tape of the simulating CSPACE machine, to be written and read off without issue.

With that said, we note that $\mathsf{CNSPACE}(s, c)$ has the same $\mathsf{ZPTIME}(2^s)$ upper bound as $\mathsf{CSPACE}(s, c)$ [DGJ+20], and there has been no work to suggest it is significantly stronger.

[9]This result is originally due to Jain et al. and will appear in upcoming work. The proof can once again be seen by taking an Eulerian tour around the configuration graph, while keeping track of the (polynomially bounded, since there are only $O(1)$ errors possible on the catalytic tape) number of starting configurations we pass so we can find the correct one to go back to.

7 Reusing space beyond space

Our last set of problems is more speculative than the rest. In a word, we ask where techniques involving space reuse may find traction outside the study of space-bounded complexity itself.

There are as many proposals to do so as there are fields, and throughout this survey the reader has been encouraged to think to their own research, of which the author certainly knows very little. These are just the ones I personally see as having potential.

7.1 Practical implementations

When Buhrman et al. introduced catalytic computation, they began with a fun illustration of the potential of the model:

> Imagine the following scenario. You want to perform a computation that requires more memory than you currently have available on your computer. One way of dealing with this problem is by installing a new hard drive. As it turns out you have a hard drive but it is full with data, pictures, movies, files, etc. You don't need to access that data at the moment but you also don't want to erase it. Can you use the hard drive for your computation, possibly altering its contents temporarily, guaranteeing that when the computation is completed, the hard drive is back in its original state with all the data intact? [BCK+14]

Before considering any theoretical models beyond Turing machines, we should see if the algorithms we already have in the abstract can be put into concrete use.

Problem 24. *Implement a catalytic algorithm such that it is actually useful.*

James Cook [Coo21] has already taken a stab at streamlining the program in Theorem 3 for STConn and seeing how it does on an actual computer; his program involves concrete tricks from programming, such as replacing addition with rotation. If we take seriously the promise of using hard disk space as our catalytic memory, there are many optimizations that would need to happen in order to make such algorithms practical.

7.2 Quantum computation

As stated in Theorem 5, catalytic computation is built from reversible operations. This calls to mind another complexity setting where every operation is invertible: evolution by unitaries, i.e. quantum computation.

There are many potential ways of approaching the potential connection between quantum and catalytic computation. One is to observe that the one-clean qubit model, known as DQC1 [KL98], is a catalytic-looking quantum model that has been widely studied for many years. Another is to think about how efficient space simulation of circuits may carry over when intermediate circuit computations no longer have the locality we exploited to get down to constant space in Section 2. A third is more optimistic: what other reversible tools for space reuse can come from the wide toolbox of unitaries?

Before jumping the gun, the precise definition of quantum catalytic computing has to be nailed down.

Problem 25. *What does quantum catalytic space look like?*

Even more so than with CBPL and CNL, care needs to be taken to get the right definition. Probably the most standard model in quantum computation is that of quantum circuits, capturing such classes as BQP. However, a circuit has a fixed notion of time, i.e. depth, and as we saw in Theorem 4, while we can reason about the average time a catalytic algorithm could take, there is no subexponential guarantee on the worst-case runtime of algorithms such as the compress-or-random argument in Theorem 2. Thus the quantum Turing Machine model of Watrous [Wat99], while somewhat non-standard, may be more appropriate.

Instead of focusing on catalytic computing per se, we could also look to existing models of quantum computation and ask what techniques such as register programs could buy us.

Problem 26. *Devise a register program using basic instructions inspired by unitary computation, and use it to show non-trivial results for e.g.* BQP.

7.3 Circuits

Recall that our main examples of space reuse, such as our "theorem" in Section 2, were about space-efficient computation of circuits. In the hierarchy of syntactic models, it is known that circuits can efficiently simulate branching programs, while there are separations known in the other direction; hence why results such as Theorem 3 are so interesting.

However, if we consider the power of circuits to simulate branching programs, this means that in some ways a circuit class of sufficient power should be able to implement the space reuse techniques we have seen. It is unclear what that actually looks like when we convert our branching programs over, or if it is even necessary or interesting to do so.

Problem 27. *Devise a circuit that uses techniques from space reuse and catalytic computing to solve some problem in low size or depth in a way that we do not know how to do directly.*

With all the circuit classes sitting between L and P, there is also the question of whether or not implementing catalytic-style techniques can help clean up this landscape.

For example, a longstanding conjecture of Immerman and Landau [IL95] states that the determinant of integral matrices is complete for TC^1. It is known that the determinant is complete for the class of log-depth arithmetic circuits whose + gates have unbounded fan-in and whose $\times$ gates have fan-in two, a class known as VP. Given that Theorem 3 states that TC^1 can be simulated by register programs, and such programs are arithmetic in nature, perhaps VP can implement our theorem as well.

Problem 28. *Show* $\mathsf{TC}^1 \subseteq \mathsf{VP}$.

7.4 Network coding

Our first example of saving space in this survey was the XOR trick for swapping values. Between this and our register program technique, it makes sense to look at areas where bit tricks and XOR tricks come in handy.

One such area is the network coding conjecture. In essence, the question is whether or not, for a given *undirected* graph with capacities on each edge and a set of source-sink pairs, we can send more message bits from each source to its corresponding sink than the network flow should allow if we think of the messages as being generic commodities instead. While this seems implausible, we note that this is in fact possible in the *directed* graph setting, as alluded to in our bulleted list in Section 1, and the counterexample uses an XOR trick.

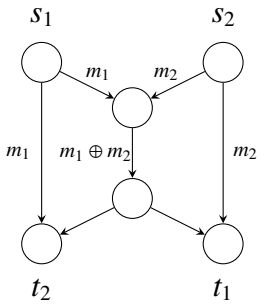

Fig.: network coding (directed counterexample)

Problem 29. *Is the network coding conjecture true or false?*

Resolving Problem 29 in the affirmative, i.e. showing that no tricks are possible for sending more bits than the network flow, would lead to circuit lower bounds for a number of fundamental problems such as sorting [FHLS20] and multiplication [AFKL19].

There are many potential approaches to Problem 29 beyond XOR tricks, but for our purposes it is sufficient to focus on this case for starters. Such "linear" strategies are known to not be optimal in terms of how many bits can be sent [DFZ05], but to the best of our knowledge we still have not resolved the network coding conjecture even in this case.

Problem 30. *Prove or disprove the network coding conjecture when all nodes are restricted to sending linear transformations of their incoming messages.*

7.5 Data structures

One setting where space usage is a key metric to study is that of data structures. Unlike in the usual complexity setting, the balance of time and space is separated into two phases: first, a preprocessing phase where some amount of space is consumed in order to prepare useful information about the input; and second, we receive a list of queries about the input that we want to answer quickly, using both the input itself as well as whatever we prepared in the first phase.

This two-phase process somewhat muddies the waters when it comes to thinking about space reuse, which is all about recomputing things many times, usually at the cost of time, in order to avoid some bottleneck in the information we have to store. It seems to be an orthogonal style of question.

Nevertheless, with space as one of the main players in the world of data structures, it seems that at least a basic investigation may be warranted.

Problem 31. *Is there a meaningful notion of a catalytic data structure, or is there anything to be gained from a data structure stored in catalytic memory?*

7.6 Cryptography

As with quantum computing, there are many ways to approach the possible link between reusing space and cryptography.

An obvious consequence of Theorem 4, as well as other explicit results [BKLS18, GJST19], is that cryptography implies lower bounds against catalytic computing, as an appropriate pseudorandom generator would show CL and many of its variants are contained in P. Such results could also resolve many of our questions about randomized computation without using any of our space reuse techniques; from the perspective of this survey this would be somewhat disappointing, but in the end results are results.

Problem 32. *Show* CL *is contained in some subclass of* P*, perhaps* NC*, given a believable cryptographic assumption.*

There is also the contrapositive side to Problem 32, of whether reusing space provides a potential barrier against cryptography just as it was a barrier against composition for space. For example, the notion of proofs of space, a variant of the proofs of work paradigm used in cryptocurrency, was complicated by Pietrzak [Pie19], who showed that a stronger object known as catalytic proofs of space would be necessary to constitute an actual "proof of work".

Problem 33. *Show evidence against objects in cryptography based on techniques in reusing space.*

To again err on the side of optimism however, we prefer to ask not what cryptography can do for catalytic computing, but rather what catalytic computing can do for cryptography.

While compress-or-random seems like the more relevant argument for randomized computation, we would like to draw attention to the register program argument at this juncture. Consider our proof of e.g. Lemma 1 again, and now let our initial values τ_i be chosen i.i.d. from $\{0, 1\}$. Then a clear consequence is that at every individual point in the execution of our algorithm, our memory is distributed i.i.d. as well, i.e. (τ_0, τ_1, τ_2) is distributed uniformly across $\{0, 1\}^3$.

In other words, our memory at every point in time reveals no secrets about the computation of f. This calls to mind potential applications such as homomorphic encryption, leakage resilience, etc. There are definitions and problems to be codified, and more steps may be needed for this approach to work; for example, any *two* points in time together may tell quite a bit about the computation of f.

Problem 34. *Show the existence, conditional or otherwise, of a natural class of cryptographic objects by using clean computation.*

A more concrete problem would be to extend the results of Applebaum et al. [AIK06] to show that cryptography in CL implies cryptography in NC^0. The warm-up for their result is based on Barrington's Theorem, i.e. our "theorem" from Section 2, and so it seems natural to believe that register programs in general could be turned into one-way functions with appropriate tinkering.

Problem 35. *Prove that the existence of one-way functions in* CL*, or even any one-way function computable by a poly-size poly-length register program, implies the existence of one-way functions in* NC^0.

8 Conclusion: to a broader theory

This brings an end to our list of open questions. We will conclude this survey, just as we began, with an entreaty, and my gratitude, to the reader who has made it to the end.

The two most important questions in the field of reusing space are also the most general: to develop new techniques and to develop new applications. New techniques may involve using more cutting-edge tools in complexity, group theory, etc., or they may simply be new ways of approaching the model beyond compression or arithmetic-style reuse. New applications could be with questions that have to do with space, or it could be that the techniques we have described in this work have a home in a very different model.

This survey was an attempt to solve a third, no less important problem: to develop new interest. I hope to have provided the reader with enough of a peek at this burgeoning field to garner interest, and to have provided at least a few problems which can be played with without too much further context. With any luck these techniques and problems will be only a prelude to the exploration of a wider world of reusing space.

Acknowledgments

I feel very lucky to have the chance to write such a lengthy and indulgent survey, and so my greatest thanks to Michal Koucký for giving it the green light to appear in the Bulletin of the EATCS. My thanks to everyone who gave me comments since its initial publication: James Cook, Igor C. Oliveira, and Ninad Rajgopal. As they say, all remaining errors are mine alone.

References

[AFKL19] Peyman Afshani, Casper Benjamin Freksen, Lior Kamma, and Kasper Green Larsen. Lower bounds for multiplication via network coding. In *International Colloquium on Automata, Languages, and Programming, ICALP*, volume 132 of *LIPIcs*, pages 10:1–10:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in nc^0. *SIAM J. Comput.*, 36(4):845–888, 2006.

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1. *J. Comput. Syst. Sci.*, 38(1):150–164, 1989.

[BBF98] Richard Beigel, Harry Buhrman, and Lance Fortnow. NP might not be as easy as detecting unique solutions. In *ACM Symposium on the Theory of Computing (STOC)*, pages 203–208. ACM, 1998.

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

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

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

[Ben73] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[Ben89] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, 1989.

[BKLS18] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic space: Non-determinism and hierarchy. *Theory Comput. Syst.*, 62(1):116–135, 2018.

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

[CM21] James Cook and Ian Mertz. Encodings and the tree evaluation problem. *Electron. Colloquium Comput. Complex.*, page 54, 2021. URL: https://eccc.weizmann.ac.il/report/2021/054.

[CM22] James Cook and Ian Mertz. Trading time and space in catalytic branching programs. In *37th Computational Complexity Conference, CCC 2022*, volume 234 of *LIPIcs*, pages 8:1–8:21, 2022.

[CMW+12] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, 2012.

[Coo21] James Cook. How to borrow memory, 2021. URL: https://www.falsifian.org/blog/2021/06/04/catalytic/.

[DFZ05] R. Dougherty, C. Freiling, and K. Zeger. Insufficiency of linear coding in network information flow. *IEEE Transactions on Information Theory*, 51(8):2745–2759, 2005.

[DGJ+20] Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Randomized and symmetric catalytic computation. In *CSR*, volume 12159 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2020.

[DT23] Dean Doron and Roei Tell. Derandomization with minimal memory footprint. In Amnon Ta-Shma, editor, *Computational Complexity Conference, CCC 2023*, volume 264 of *LIPIcs*, pages 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[Dul] Yfke Dulek. Catalytic space: on reversibility and multiple-access randomness.

[FHLS20] Alireza Farhadi, Mohammad Taghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. *Commun. ACM*, 63(10):97–105, 2020.

[GJST19] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. Unambiguous catalytic computation. In *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2019*, volume 150 of *LIPIcs*, pages 16:1–16:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

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

[Hel86] Hans Heller. On relativized exponential and probabilistic complexity classes. *Inf. Control.*, 71(3):231–243, 1986.

[Hoz21] William M. Hoza. Better pseudodistributions and derandomization for space-bounded computation. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2021*, volume 207 of *LIPIcs*, pages 28:1–28:23, 2021.

[Hoz22] William M. Hoza. Recent progress on derandomizing space-bounded computation. *Bull. EATCS*, 138, 2022.

[IL95] Neil Immerman and Susan Landau. The complexity of iterated multiplication. *Inf. Comput.*, 116(1):103–116, 1995.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[KL98] E. Knill and R. Laflamme. Power of one bit of quantum information. *Phys. Rev. Lett.*, 81:5672–5675, 1998.

[Kou16] Michal Koucký. Catalytic computation. *Bull. EATCS*, 118, 2016.

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

[LMT00] Klaus-Jörn Lange, Pierre McKenzie, and Alain Tapp. Reversible space equals deterministic space. *J. Comput. Syst. Sci.*, 60(2):354–367, 2000.

[Lof] Bruno Loff. Private correspondence.

[Nis93] Noam Nisan. On read-once vs. multiple access to randomness in logspace. *Theor. Comput. Sci.*, 107(1):135–144, 1993.

[Pie19] Krzysztof Pietrzak. Proofs of catalytic space. In Avrim Blum, editor, *Innovations in Theoretical Computer Science Conference, ITCS*, volume 124 of *LIPIcs*, pages 59:1–59:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Pot17] Aaron Potechin. A note on amortized branching program complexity. In *Computational Complexity Conference*, volume 79 of *LIPIcs*, pages 4:1–4:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[PRZ23] Edward Pyne, Ran Raz, and Wei Zhan. Certified hardness vs. randomness for log-space. *CoRR*, 2023.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, 2008.

[RZ21] Robert Robere and Jeroen Zuiddam. Amortized circuit complexity, formal complexity measures, and catalytic algorithms. In *FOCS*, pages 759–769. IEEE, 2021.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

[SZ99] Michael E. Saks and Shiyu Zhou. BP $_{\mathrm{h}}$space(s) subseteq dspace(s$^{3/2}$). *J. Comput. Syst. Sci.*, 58(2):376–403, 1999.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[Wat99] John Watrous. Space-bounded quantum complexity. *J. Comput. Syst. Sci.*, 59(2):281–326, 1999.

Exercise solutions

Solution 1. *As stated, we will only seek to reprove the correctness of the program for $g_1 \wedge g_2$. Define $y_1 = \tau_1 + g_1$ and $y_2 = \tau_2 + g_2$; in other words, before running P_1 the register R_1 contains τ_1, while afterwards it contains y_1, and likewise for P_2. Then by simple arithmetic*

$$g_1 g_2 \equiv y_1 y_2 + y_1 \tau_2 + \tau_1 y_2 + \tau_1 \tau_2 \mod 2$$

Thus we can add $g_1 g_2$ to R_0 by adding each of the four monomials in this expansion to R_0, one by one, each obtained by running a combination of P_1 and P_2.

In fact, by ordering them as

$$g_1 g_2 \equiv \tau_1 \tau_2 + \tau_1 y_2 + y_1 y_2 + y_1 \tau_2 \mod 2$$

this can be done using only four recursive calls as presented in the original proof of Lemma 1, rather than however many were in our subsequent exposition.

Solution 2. *We will use the same recursive statement as Lemma 1, with two changes. First, we will need a program to handle gates of the form $g = g_1 + g_2$. This is clearly accomplished by the following program $P_g(i)$:*

1. $P_1(i)$ $\quad R_i = [\tau_i] + [v_1]$
2. $P_2(i)$ $\quad R_i = [\tau_i + v_1] + [v_2] = \tau_i + v_1 + v_2$ ✓

Before moving on to $\times$ gates, we need to handle the fact that executing $P_g(i)$ twice no longer resets memory, as we are no longer over $\mathbb{F}_2$. To handle this, we will extend our recursion to state that in addition to $P_g(i)$ we also have a program $P_g^{-1}(i)$ whose function is to subtract v_g from R_i. This is clearly true at the leaves since reading inputs is atomic, and swapping $P_1(i)$ and $P_2(i)$ for their inverses in our + program above gives $P_g^{-1}(i)$.

Now we can move on to $g = g_1 g_2$. Our program from Lemma 1 works almost directly for $\times$ gates, with the only catch being that we convert + into − in some places in order to get the cancellations to work. Concretely our program $P_g(0)$ is as follows:

1. $P_1(1)$ $\quad R_1 = [\tau_1] + [v_1]$
2. $R_0 \mathrel{+}= -R_1 R_2$ $\quad R_0 = [\tau_0] + [-(\tau_1 + v_1)(\tau_2)]$
   $= \tau_0 - \tau_1\tau_2 - v_1\tau_2$
3. $P_2(2)$ $\quad R_2 = [\tau_2] + [v_2]$
4. $R_0 \mathrel{+}= R_1 R_2$ $\quad R_0 = [\tau_0 - \tau_1\tau_2 - v_1\tau_2] + [(\tau_1 + v_1)(\tau_2 + v_2)]$
   $= \tau_0 + \tau_1 v_2 + v_1 v_2$
5. $P_1^{-1}(1)$ $\quad R_1 = [\tau_1 + v_1] + [-v_1]$
   $= \tau_1$ ✓
6. $R_0 \mathrel{+}= -R_1 R_2$ $\quad R_0 = [\tau_0 + \tau_1 v_2 + v_1 v_2] + [-(\tau_1)(\tau_2 + v_2)]$
   $= \tau_0 - \tau_1\tau_2 + v_1 v_2$
7. $P_2^{-1}(2)$ $\quad R_2 = [\tau_2 + v_2] + [-v_2]$
   $= \tau_2$ ✓
8. $R_0 \mathrel{+}= R_1 R_2$ $\quad R_0 = [\tau_0 - \tau_1\tau_2 + v_1 v_2] + [(\tau_1)(\tau_2)]$
   $= \tau_0 + v_1 v_2$ ✓

and it is easy to show that running the same program with all signs flipped gives $P_g^{-1}(0)$ as well. Once again all other programs $P_g^{-1}(i)$ can be obtained by relabeling.

Solution 3. *As mentioned, there are in fact two solutions, both of which has its own advantage. The first is to execute Lemma 1 on a tree of fan-in two $\wedge$ gates of height $\log d$; this can be done in place using very few registers and $4^{\log d} = d^2$ recursive calls.*

The second solution is much less efficient in terms of recursion:

$$\begin{array}{l} \textit{for all } S \subseteq [d]: \\ \quad 1.\quad P_i \qquad \forall i \in S \\ \quad 2.\quad R_0 \mathrel{+}= \prod_{i=1}^{d} R_i \\ \quad 3.\quad P_i^{-1} \qquad \forall i \in S \end{array}$$

This follows by a generalization of the alternate analysis given in the previous exercise, namely

$$\prod_{i=1}^{d} g_i \equiv \sum_{S \subseteq [d]} \left(\prod_{i \in S} \tau_i + x_i \right) \left(\prod_{i \notin S} \tau_i \right)$$

by inclusion-exclusion.

In terms of utility, the first is clearly much more efficient with respect to time. However, it turns out that the second is more useful for some generalized applications, in particular when we want to compute not just one product but an arbitrary number of products in parallel. The first algorithm must be scaled linearly in terms of time or space, while it turns out the latter can be modified to work with no increase in either.

Solution 4. *The following program computes P_Σ:*

$$\begin{array}{lll} 1.\quad P_i \qquad \forall i & & R_i = [\tau_i] + [v_i] \qquad \forall i \\ 2.\quad R_\Sigma \mathrel{+}= \sum_i R_i & & R_0 = [\tau_0] + \left[\sum_i (\tau_i + v_i) \right] \\ & & \quad = \tau_0 + \sum_i \tau_i + \sum_i v_i \\ 3.\quad P_i^{-1} \qquad \forall i & & R_i = [\tau_i + v_i] + [-v_i] \qquad \forall i \\ & & \quad = \tau_i \quad \forall i \quad \checkmark \\ 4.\quad R_\Sigma \mathrel{+}= -\sum_i R_i & & R_0 = \left[\tau_0 + \sum_i \tau_i + \sum_i v_i \right] + \left[-\sum_i \tau_i \right] \\ & & \quad = \tau_0 + \sum_i v_i \quad \checkmark \end{array}$$

We note that this also gives a program P_Σ^{-1} by running the program in reverse order and flipping all signs.

Solution 5. *The following program computes P_k given P_v, and uses k registers $R'_1 \dots R'_k$ in addition to R_0 and R_v:*

$$\begin{array}{lll} 1.\quad P_v^{-1} & & R_v = [\tau_v] + [-v] \\ 2.\quad R'_j \mathrel{+}= (R_v)^j \qquad \forall j = 0 \dots k & & R'_j = [\tau'_j] + [(\tau_v - v)^j] \qquad \forall j = 0 \dots k \end{array}$$

3.	P_v	R_v	$= [\tau_v - v] + [v]$
			$= \tau_v$
4.	$R_0 \mathrel{+}= \sum_{j=0}^{k} \binom{k}{j} (R_v)^j (-1)^{k-j} R'_{k-j}$	R_0	$= [\tau_0] + \left[\sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau'_{k-j} + (\tau_v - v)^{k-j}) \right]$
			$= \tau_0 + \sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau'_{k-j})$
			$\quad + \sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau_v - v)^{k-j}$
5.	P_v^{-1}	R_v	$= [\tau_v] + [-v]$
6.	$R'_j \mathrel{+}= -(R_v)^j \qquad \forall j = 0 \ldots k$	R'_j	$= [\tau'_j + (\tau_v - v)^j] + [-(\tau_v - v)^j]$
			$= \tau'_j \qquad \forall j = 0 \ldots k \quad \checkmark$
7.	P_v	R_v	$= [\tau_v - v] + [v]$
			$= \tau_v \quad \checkmark$
8.	$R_0 \mathrel{+}= -\sum_{j=0}^{k} \binom{k}{j} (R_v)^j (-1)^{k-j} R'_{k-j}$	R_0	$= [\tau_0 + \sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau'_{k-j})$
			$\quad + \sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau_v - v)^{k-j}]$
			$\quad + \left[- \sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau'_{k-j}) \right]$
			$= \tau_0 + \sum_{j=0}^{k} \binom{k}{j} \tau_v^j (-1)^{k-j} (\tau_v - v)^{k-j}$
			$= \tau_0 + v^k \quad \checkmark$

As before we get an inverse program by running in reverse and flipping all signs.

Solution 6. *The same averaging argument as Theorem 4 shows that a random catalytic tape will only have a polynomial size component in the configuration graph, even when this component is no longer a line and can branch based on randomness or non-uniformity. We choose a random catalytic tape using the randomness of* ZPP*, copy it to remember where we started, and then take an Eulerian tour of the given component in the same way as Theorem 5.*

To simulate CNL *it is enough to look for any accept state we see before returing to our starting configuration, while for* CBPL *we keep a list of how many accept and reject states we encounter and take a majority vote. Note that in either case we will never get the wrong answer, although as before we may have to declare "I don't know" if we pick a bad starting catalytic tape and the procedure takes too long.*

Solution 7. *As in our alternate proof of Lemma 1 (see the solution to Exercise 1, given above), define $y_i = \tau_i + x_i$ for each i. Let $p_f(x_1 \ldots x_n)$ be the $\mathbb{F}_2$ polynomial computing f given by interpolation, and let $q_f(\tau_1 \ldots \tau_n, y_1 \ldots y_n)$ be defined by substitution into p_f, namely*

$$q_f(\tau_1, \ldots, \tau_n, y_1, \ldots, y_n) = p_f(y_1 - \tau_1, \ldots, y_n - \tau_n)$$

Given this substitution, consider all monomials in q_f, i.e.

$$q_f = \sum_{S,T \subseteq [n]} c_{S,T} (\prod_{i \in S} \tau_i)(\prod_{i \in T} y_i)$$

for some constants $c_{S,T} \in \mathbb{F}_2$. Note that by construction we need only consider disjoint S and T, since p_f is over $\mathbb{F}_2$ and thus is multilinear. For convenience we relabel these products τ^S and y^T respectively.

We will use registers R_i for each i as well as registers $R_{S,\tau}$ and $R_{T,y}$ for each $S, T \subseteq [n]$. Our idea is to store each τ^S into the corresponding $R_{S,\tau}$ and similarly for y, so that

$$R_{S,\tau} \cdot R_{T,y} = (\tau_{S,\tau} + \tau^S)(\tau_{T,y} + y^T) = \tau_{S,\tau}\tau_{T,y} + \tau_{S,\tau}y^T + \tau^S\tau_{T,y} + \tau^S y^T$$

and as in Lemma 1 we use four rounds to cancel out the various junk terms to get exactly $\tau^S y^T$. Thus the following program computes P_f:

1.	$R_{S,\tau}$ += $\prod_{i\in S} R_i$ $\quad \forall S \subseteq [n]$	$R_{S,\tau} = [\tau_{S,\tau}] + [\tau^S] \quad \forall S \subseteq [n]$
2.	R_0 += $\sum_{S,T\subseteq[n]} c_{S,T} R_{S,\tau} R_{T,y}$	$R_0 = [\tau_0] + \left[\sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau} + \tau^S)(\tau_{T,y})\right]$ $= \tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau}\tau_{T,y} + \tau^S\tau_{T,y})$
3.	$P_i \quad \forall i \in [n]$	$R_i = [\tau_i] + [v_i] \quad \forall i \in [n]$
4.	$R_{T,y}$ += $\prod_{i\in T} R_i$ $\quad \forall T \subseteq [n]$	$R_{T,y} = [\tau_{T,y}] + [y^T] \quad \forall T \subseteq [n]$
5.	R_0 += $\sum_{S,T\subseteq[n]} c_{S,T} R_{S,\tau} R_{T,y}$	$R_0 = [\tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau}\tau_{T,y} + \tau^S\tau_{T,y})]$ $+ \left[\sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau} + \tau^S)(\tau_{T,y} + y^T)\right]$ $= \tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau}y^T + \tau^S y^T)$
6.	$R_{S,\tau}$ += $\prod_{i\in S} R_i$ $\quad \forall S \subseteq [n]$	$R_{S,\tau} = [\tau_{S,\tau} + \tau^S] + [\tau^S] \quad \forall S \subseteq [n]$ $= \tau_{S,\tau} \quad \forall S \subseteq [n]$ ✓
7.	R_0 += + $\sum_{S,T\subseteq[n]} c_{S,T} R_{S,\tau} R_{T,y}$	$R_0 = [\tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau}y^T + \tau^S y^T)]$ $+ \left[\sum_{S,T\subseteq[n]} c_{S,T}\tau_{S,\tau}(\tau_{T,y} + y^T)\right]$ $= \tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau}\tau_{T,y} + \tau^S y^T)$
8.	$P_i \quad \forall i \in [n]$	$R_i = [\tau_i + v_i] + [v_i] \quad \forall i \in [n]$ $= \tau_i \quad \forall i \in [n]$ ✓
9.	$R_{T,y}$ += + $\prod_{i\in T} R_i$ $\quad \forall T \subseteq [n]$	$R_{T,y} = [\tau_{T,y} + y^T] + [y^T] \quad \forall T \subseteq [n]$ $= \tau_{T,y} \quad \forall T \subseteq [n]$ ✓
10.	R_0 += + $\sum_{S,T\subseteq[n]} c_{S,T} R_{S,\tau} R_{T,y}$	$R_0 = [\tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}(\tau_{S,\tau}\tau_{T,y} + \tau^S y^T)]$ $+ \left[\sum_{S,T\subseteq[n]} c_{S,T}\tau_{S,\tau}\tau_{T,y}\right]$ $= \tau_0 + \sum_{S,T\subseteq[n]} c_{S,T}\tau^S y^T = \tau_0 + q_f$ ✓

Solution 8. *The world is your oyster.*