Catalytic Computing, Tree Evaluation, & Clean Computation

Ian Mertz
University of Toronto
`mertz@cs.toronto.edu`

November 30, 2020
Thesis Proposal, DCS, University of Toronto

Introduction. When proving lower bounds on space-bounded computation, often we end up in a scenario where we will force the machine in question to compute multiple separate instances of some computation, in the hopes that whichever instance it computes first will have to be remembered throughout the computation on the rest. An important example of such a program is the drive to get lower bounds on the *tree evaluation problem* ($\mathsf{TreeEval}$), defined by Cook et al. [CMW+12] as a potential language separating L from P. The setup is very general: we are given a full binary tree of height h, and for some number k every leaf is labeled with an element of $[k]$ and every internal node is labeled with an entire function $[k] \times [k] \to [k]$, with the solution to the problem being the value output at the root when the tree is evaluated in a bottom-up fashion in the natural way. Here we can hope to prove that the space required is $\omega(\log n) = \omega(h + \log k)$ by using a strategy (often called *pebbling*) as described before. More specifically, in order to compute both the left and right children of the root, we will have to eventually compute *one* of them, at which point we will be remembering $\log k$ bits. Then if we look at the other child, we have an instance of height $h-1$ to compute, and applying the same strategy all the way down gives us h layers in which to remember $\log k$ bits, for a total of $\Omega(h \log k)$ space. Follow-up results for read-once branching programs and other restricted models have confirmed this full lower bound in many natural settings, and Cook et al.[CBM+09] conjectured that $\mathsf{TreeEval}_{k,h} \in BPSIZE(k^{\Omega(h)})$.

However, there is a natural question that arises when taking this strategy: is the space required to both remember some old information and compute some new information approximately the total amount of space required to do each of them separately? At first glance this seems almost trivial; how could one hope to save any space computing a new function given some junk information in an arbitrary (possibly incompressible) state that has no relevance to the computation at hand? To formally study this problem, Buhrman et al. [BCK+14] introduced a model of computation called *catalytic computing*. We consider a variant of Turing machines where we have four tapes: first, a read-only input tape of length n; second, a write-only output tape of length m; third, a read-write work tape of length s; and fourth, a read-write "catalytic tape" of length $2^{O(s)}$ which is in an unknown inital state, and which the program is free to use as workspace provided it is returned to its original configuration at the end of the computation. Focusing on the setting where $s = O(\log n)$, which they call *catalytic logspace* (denoted CL), they show a surprising and counterintuitive result, which is that regardless of the setting of the catalytic tape, not only can we solve problems widely conjectured to not in L, but we can solve every problem in the class TC^1, which (likely strictly) contains NL!

This is not only a phenomenon in this new catalytic model, but rather has implications in the traditional space-bounded model that we seek to prove $\mathsf{TreeEval}$ lower bounds in. The main theorems in [BCK+14] used key subroutines appearing in a number of seminal results on the power of logspace, namely Barrington's Theorem showing that $\mathsf{NC}^1 \subseteq \mathsf{L}$ [Bar89] and its extension to show $\#\mathsf{NC}^1(R) \subseteq \mathsf{L}$ for small enough rings R [BoC92]. Recently we showed [CM20] how $\mathsf{TreeEval}_{k,h}$ can be solved by branching programs of size $k^{o(h)}$ by extending one key lemma from [BCK+14], disproving Cook's conjecture.[1] Furthermore we show a direct way to prove stronger upper bounds—even the optimal result $\mathsf{TreeEval} \in \mathsf{L}$—by further improving the parameters in this key lemma.

Clearly there is something more to be understood about the power of space-bounded computation, and in particular in the ability to "borrow" used space in a way that is useful for new computation without having to go through compression. Further upper bounds through techniques from catalytic computing could continue to show the surprising power of space-bounded computation, and on the flip side lower bounds for L will have to face these techniques sooner or later. Since we are seeking

[1] In this original paper we only achieve this upper bound for a limited range of parameters; in upcoming work we prove an even stronger upper bound for every setting of parameters.

to understand both the old model of space-bounded computation and the new model of catalytic computing, we turn now to a unifying model of computation through which we understand these key subroutines and lemmas.

Clean computation and logarithmic depth. Another notion of computation defined by [BCK$^+$14], building off the central lemma in [BoC92] and used as a way to describe the key operations needed to establish their results, is that of *clean computation.* In clean computation both our work space and the output tape are filled in with values at the start of the computation, and in addition to the catalytic restriction that the work tape be restored at the end of the computation, we say that the machine computes the function f if at the end the value of a function $f(x_1 \dots x_k)$ is *added* to the output tape.[2]

While this seems like an odd computation model to introduce when neither L nor CL have anything in the output registers,[3] in practice this definition is used *within* the computation to compute a subfunction g into a chunk of used space. From this we build up a recursive way of computing f; first, we recursively cleanly compute all subfunctions g_i needed to compute f into our space, whether it be the huge catalytic tape or the small logspace work tape, and from those we cleanly compute f. The important thing here is that as long as we have space to store the *outputs* of the subfunctions g_i, we don't have to worry about the actual space required to compute each of them, as we can borrow the output space from other g_is for computation.

Another motivating factor for this definition is that when we apply this recursively to some subfunction f and its subfunctions g_i, we quickly realize that the g_is cannot output their values into unused space, since there is only enough space set aside for f and other subfunctions it will be used alongside; hence why we add their values to used memory. However this brings up a complicating factor: rather than getting the input to f from the easy-to-read input tape, we need to extract the results of computing the g_is from the *used* space where they were cleanly computed into. In other words, the input to f is given by our ability to mask (and demask) each of the "catalytic input registers" by the result of g_i. Thus the lemmas in [Bar89, BoC92, BCK$^+$14] take on the following form: "Assume for all $i \in [m]$ we have a program P_i that cleanly computes g_i into R_i; then there exists a program P which cleanly computes $f(g_1 \dots g_m)$ into R."

The clearest (and most prominent) example of how this type recursive of computation is used is in simulating log-depth circuits. We can do a bottom-up simulation by defining a program P_g for each gate g in the circuit using recursive calls to the $P_{g'}$ programs for all g' feeding into g. It is not hard to see that the total runtime of P_f for the output gate f is t^d, where t is the number of recursive calls each P_g makes and d is the depth of the circuit. For example, to show that $\#\mathsf{NC}^1 \subseteq \mathsf{L}$ it is enough to show that given programs P_i and P_j computing g_i and g_j it is possible to cleanly compute $g_i + g_j$ and $g_i \times g_j$ using only a logarithmic number of registers and a constant number of recursive calls; in fact [BoC92] show that it can be done with *three* registers and four recursive calls.

Clean input-masked computation. Inspired by this framework, we propose a modification of clean computation called "clean input-masked computation". Unlike the four tapes of catalytic computing and three tapes of clean computation, we a model with only one read-write tape, partitioned into three parts: input, output, and work. This tape is completely catalytic; it starts in an initial configuration that is out of our control, and in the end we are required to cleanly compute

[2]This notion is independent of the field, and often we think of all the cells as being registers over some field rather than traditional bit registers. For this work it is fine to think of everything as being a bit register, and when we say addition we mean bitwise addition mod 2.

[3]In fact in these models the output registers are write-only, so this idea of adding the value of a function doesn't even make sense.

$f(x_1 \dots x_k)$ into the output tape, meaning all other space needs to be in its initial configuration. Furthermore the input tape $R_1 \dots R_k$ does not actually store the input; we are given access to a function P, which takes as input a set of input coordinates $S \subseteq [k]$ and adds x_i to $R_i \pmod 2$ for all $i \in S$. Thus we say f can be (s,t)-cleanly input-masked computed, where our measure of space will be the length of the tape and our measure of time will be the number of times we called P.

To motivate this definition, we restate the key lemmas from [BoC92, BCK+14, CM20]; using the recursive analysis from the previous section it is a simple exercise to show their respective main theorems.

Lemma 1 ($\#\mathsf{NC}^1 \subseteq \mathsf{L}$)**.** *$f = x_1 + x_2$ and $f = x_1 \times x_2$ can both be $(3,4)$-cleanly input-masked computed.*

Lemma 2 ($\mathsf{TC}^1 \subseteq \mathsf{CL}$ (1))**.** *$f = \sum_{i \in [d]} x_i$ can be $(d+1, 2)$-cleanly input-masked computed.*

Lemma 3 ($\mathsf{TC}^1 \subseteq \mathsf{CL}$ (2))**.** *$f = x^d$ can be $(d+1, 2)$-cleanly input-masked computed over commutative rings.*

Lemma 4 ($\mathsf{TreeEval}_{k,h} \in BPSIZE(k^{o(h)})$)**.** *Let $x_1 \dots x_n$ be a set of inputs, let $m \leq 2^d$, and let $i_{ab} \in [n]$ for all $a \in [m]$, $b \in [d]$. Then for any polynomials $\{f_a\}$ the ensemble $[f_1(x_{i_{11}} \dots x_{i_{1d}}), \dots, f_m(x_{i_{m1}} \dots x_{i_{md}})]$ can be $(n+m, 2^d)$-cleanly input-masked computed.*

Thesis proposal. The field of clean computation is quite young, and there are a number of places where it may be possible to strengthen the existing clean input-masked computation lemmas with far-reaching consequences. I will propose two concrete lines of research on clean input-masked computation as well as their motivation.

1. The first is a direct way to obtain breakthrough upper bounds for the Tree Evaluation Problem. As stated earlier, in upcoming work we show that they can be computed with branching programs of size $k^{O(h/\log h)}$. Furthermore all evidence in the proof indicates that going beyond the frontier lies in clean input-masked computation.

   Theorem 5. *If Lemma 4 can be strengthened to be $(n+m, t(d))$-cleanly input-masked computed for any function $t(d)$, then $\mathsf{TreeEval}_{k,h}$ can be solved by branching programs of size $(\log t(k))^{O(h)} \cdot \mathrm{poly}(k)$. In particular if $t(d) = \mathrm{poly}(d)$ then it can be solved in size $O(\log k)^h \cdot \mathrm{poly}(k)$, and if $t(d) = O(1)$ then $\mathsf{TreeEval} \in \mathsf{L}$.*

   While it seems like asking to compute an exponential number of polynomials—each of which potentially having an exponential number of terms—using only a constant number of recursive calls seems like a big ask, the real question is how hard it is to compute individual terms versus many terms in the same variables. The following lemma in [CM20] was the stepping stone to proving Lemma 4.

   Lemma 6. *$f = \prod_{i \in [d]} x_i$ can be $(d+1, 2^d)$-cleanly input-masked computed.*

   The next insight in [CM20] is that this construction can be "parallelized" to work for any number of terms simultaneously at no extra cost; the space needed is exactly the space needed to store the input and output, while the time needed is exponential in the degree irrespective of the number of terms or polynomials. We also showed (not yet published) that $f = \prod_{i \in d} x_i$ can be $(d+1, O(1))$-cleanly input-masked computed, which improves Lemma 6 to its optimal space and time. The catch is that as of now this result doesn't parallelize to an arbitrary number of terms in the same way. Showing either an improvement on the construction in

Lemma 6 to reduce the time or on the new lemma to get parallelization would immediately improve the state of the art for $\mathsf{TreeEval}$. On the flip side, showing a lower bound would halt our forward progress; in upcoming work we show that our construction is essentially tight for the types of algorithms we proposed in [CM20], and so further progress crucially relies on computing parallel products.

2. The second potential improvement is new containment results for CL. As discussed earlier it is difficult to use clean input-masked computation as a subroutine when the depth of the circuit class you want to simulate is superlogarithmic, since you are necessarily paying a t^d price in the runtime. On the other hand, the field has had great success in driving down the time needed to cleanly input-masked compute individual operations, with the best example being the fact that the majority function (and indeed an ensemble of $\mathrm{poly}(n)$ majority functions) can be $(\mathrm{poly}(n), O(1))$-cleanly input-masked computed, leading to the current record of $\mathsf{TC}^1 \subseteq \mathsf{CL}$. Thus one potential way to circumvent the log-depth barrier is to "compress" high-depth circuits into log-depth by replacing $\omega(1)$ layers at a time with a single gate computing a hard function, and then working on new upper bounds for cleanly masked-input computing that function.

   Theorem 7. *If an ensemble of* $\mathrm{poly}(n)$ *undirected s-t connectivity instances can be* $(\mathrm{poly}(n), O(1))$*-cleanly input-masked computed, then* $\mathsf{NC}^2 \subseteq \mathsf{CL}$.

   One interesting observation about this theorem is that it can be seen as a strengthening of Lemma 3 to work for non-commutative rings, as powering the adjacency matrix of a graph can be directly used to solve undirected s-t connectivity. However it is unclear whether or not such a strengthening is possible.

Hedging my bets. While we've made progress on clean input-masked computation in previous and upcoming work, these two challenges still seem quite formidable. As such I also propose two additional lines of research, both of which are based on past and ongoing research.

1. The first is to study the relationship between CL and P. Currently we know that CL and related non-deterministic variants are in ZPP, which means that under the plausible hypothesis that $\mathsf{ZPP} = \mathsf{P}$ we have $\mathsf{CL} \subseteq \mathsf{P}$; furthermore this containment is known under other well-believed derandomization hypotheses as well. Regardless, since actually proving $\mathsf{ZPP} = \mathsf{P}$ seems to be a far-off dream for the field, it would be useful to have a direct and unconditional proof that $\mathsf{CL} \subseteq \mathsf{P}$, and we have a number of potential approaches for doing so. Alternatively it could be that resolving this question is similar to resolving $\mathsf{ZPP} = \mathsf{P}$ in terms of difficulty, and that this can be stated formally as a barrier to proving $\mathsf{CL} \subseteq \mathsf{P}$.
2. The second is unrelated to catalytic computing, and lies with *lifting theorems*. Query-to-communication lifting has attracted a lot of attention in the past few years for its ability to prove strong lower bounds for difficult models such as communication complexity and proof complexity. In recent work [LMM$^+$20] we show that the standard arguments in most lifting theorems can be greatly simplified using robust sunflowers, a tool from combinatorics. Furthermore these arguments allow us to improve the size of the index gadget used as the inner lifting function; if strengthened further this could have implications in monotone circuit complexity and extended formulations. We propose two research problems: 1) further improving the construction to reduce the gadget size; and 2) extending these simplified arguments to other gadgets (namely the inner product) and other lifting theorems (namely randomized lifting).

References

[Bar89] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[BCK$^+$14] Harry Buhrman, Richard Cleve, Michal Kouckỳ, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Proccedings of the 46th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2014*, pages 857–866. ACM, 2014.

[BoC92] Michael Ben-or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, February 1992.

[CBM$^+$09] Stephen Cook, Mark Braverman, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Branching programs: Avoiding barriers. Talk at Barriers Workshop at Princeton, August 2009. URL: https://www.cs.toronto.edu/~sacook/barriers.ps.

[CM20] James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 752–760. ACM, 2020.

[CMW$^+$12] Stephen 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, January 2012.

[LMM$^+$20] Shachar Lovett, Raghu Meka, Ian Mertz, Toniann Pitassi, and Jiapeng Zhang. Lifting with sunflowers. *Electron. Colloquium Comput. Complex.*, page 111, 2020. URL: https://eccc.weizmann.ac.il/report/2020/111.