DOI:10.1145/3431281

Nachum Dershowitz

Viewpoint
Let's Be Honest

Seeking to rectify the two mutually exclusive ways of comparing computational power—encoding and simulation.

IMAGE BY ANDRIJ BORYS ASSOCIATES

We have a serious problem with how we have been teaching computability theory, a central component of the ACM/IEEE computer science curriculum.

Let me explain. For a fair number of years, I taught a computability course. Following the standard curriculum (such as described by Hopcroft and Ullman[14]), and in concert with my colleagues in the field, I made claims on countless occasions that one model of computation is more powerful than another or that two models have the same power of computation. In some cases the argument appealed to ordinary set inclusion, while at other times it involved a notion of simulation via encodings. Imagine my chagrin when I came to realize these two methods of comparison are in fact incompatible!

When two models work with the same entities, simple set inclusion of formal languages or sets of functions is employed naturally by everyone. We teach that finite-state automata recognize the same languages as defined by regular expressions but are strictly weaker than pushdown automata, and we bring palindromes or non-square words as proof positive.[13] Similarly, we assert that primitive recursion (or, equivalently, looping via bounded **for** loops only) is weaker than general recursion (with **while** loops, too) because of the two models, only the latter can compute the Ackermann function.[14]

When, on the other hand, the domains of the models under consideration differ, encodings are required before they can be compared. For example, Alan Turing, in an appendix to his profound landmark 1936 paper, showed that the lambda-computable functions and the functions that can be computed using his Turing machines are of equivalent computational power. "Standard" machine descriptions (lists of quintuples) were turned into decimal numbers, which in turn were expressed in the lambda calculus as Church numerals. Turing also proved that his machines and general recursion are equipotent.[24] To show that Turing machines can compute all general recursive functions, numbers are normally (and wastefully) represented on the machine tape as a sequence of tally marks in unary.[14]

Unfortunately, the preceding two methods of comparison, namely inclusion and simulation, can yield mutually exclusive outcomes. The simplest example is the counter machine (a.k.a. Minsky machine, abacus model). Each counter holds a natural number that can be incremented, decremented, and tested for zero (see the sidebar). With only two counters, the model is not even powerful enough to square its input[2,20] or recognize primes.[15] However, if we agree to represent a number n as the exponential 2^n (a simpler encoding than Gödel numbering of expressions), then, courtesy an ingenious proof by the late Marvin Minsky,[14,17] we find that two counters suffice to compute every computable function. This is why one encounters statements such as:

- It is well known that a finite-state

ACM Transactions on
Computing for Healthcare

ACM Transactions on Computing for Healthcare (HEALTH) is a multi-disciplinary journal for the publication of high-quality original research papers, survey papers, and challenge papers that have scientific and technological results pertaining to how computing is improving healthcare.

For further information and to submit your manuscript, visit health.acm.org

automaton equipped with two counters is Turing-complete.[9]

▸ [Minsky proved that a] two-counter machine is universal, and hence has an undecidable halting problem.[16]

Such claims of completeness or universality would be blatantly false were one to subscribe to the set-inclusion sense, whereas they are manifestly true in the simulation sense, which is indeed what Minsky proved. As Rich Schroeppel expressed it: "Any counter machine can be simulated by a 2CM, provided an obscure [sic!] coding is accepted for the input and output."[20] So, I take issue with a statement like this: "The surprising result about counter machines is that two counters are enough to simulate a Turing machine and therefore to accept every recursively enumerable language."[13] Two-counter machines do simulate Turing machines, but they do not "accept" all recursively enumerable languages in the usual "as-is," unencoded sense, primes being a prime example.

The point is it behooves teachers to be forthright and forthcoming and to address this inconsistency. We cannot carry on oblivious to the fact that by one of the methods of comparison that we use in our lectures 2-counter machines are strictly weaker than (the complete) 3-counter machines, while by a second method that we also endorse the two are to be deemed equivalent.

The Solution

All is not lost, thankfully. We can eat our proverbial cake and still have it, provided we invest extra effort.

To begin with, it would be an unmitigated disaster to abandon simulations, since the idea that all our traditional unrestrained models are of equivalent power, despite operating with different entities, stands at the core of computability theory, as enshrined in the Church-Turing thesis. Consequently, as painful as it may seem, we are obliged to give up the inclusion notion for paradigms that compute functions, such as general recursion and primitive recursion—though not for formal languages, as I will explain later.

By "simulation" one usually means there is an (injective, 1-1) encoding c from the domain of the simulated model M into the domain of the simulating model M' such that every function f computed by the former is mirrored by a function f' computed by the simulator M' such that $f'(c(x_1),\ldots,c(x_n)) = c(f(x_1,\ldots,x_n))$ for all inputs $x_1,\ldots,x_n$ coming from the domain of M. The following are textbook quotations:

▸ To show two models are equivalent, we simply need to show that we can simulate one by the other ... Any two computational models that satisfy certain reasonable requirements can simulate one another and hence are equivalent in power.[22]

▸ Computability relative to a coding is the basic concept in comparing the power of computation models ... Thus, we can compare the power of computation models using the concept "incorporation relative to some suitable coding."[23]

But what should be deemed "reasonable" or "suitable" lies in the eyes of the beholder. If we do take this simulation route, and I believe we must, and if we are to have a mathematically satisfying theory of computation, then we are in dire need of a formal definition of allowable encodings c.

Hartley Rogers elucidated, "The coding is chosen so that it is itself given by an informal algorithm in the unrestricted sense."[19] This requirement, however valid, is at the same time too informal and potentially too generous. And it is circular, since our goal is to demarcate the limits of effective computation. As Richard Montague complained: "The natural procedure is to restrict consideration to those correspondences which are in some sense 'effective' ... But the notion of effectiveness remains to be analyzed, and would indeed seem to coincide with computability."[18] The only way around its informality would be to agree somehow on a uniform, formal notion of "effective" algorithm that crosses domains (such as Boker and Dershowitz[4]). Still, an "unrestricted" encoding could conceivably enlarge the set of functions that can be computed by the simulating model, as we saw with counter machines and an effective exponential encoding.

What other restrictions, then, should be imposed on encodings? Obviously, we need one and the same encoding c to work for all simulated functions f. Were one to examine lone

Counter Machines

Counter machines are one of the very simplest models of computation.

Think of a collection of bowls of marbles, alongside a heap containing an unlimited supply of more marbles.

An *n*-counter machine comes with *n* bowls.

A program consists of a list of instructions of the following five simple types:

(1) Place a marble taken from the pile into bowl X, where X is a particular bowl.

(2) Remove a marble from bowl X, and return it to the pile; do nothing if there is nothing in the bowl.

(3) Check if there are no marbles in bowl X; if so, continue with instruction *K*, where *K* is the number or label of one of the instructions in the program.

(4) Continue with instruction *K*, unconditionally.

(5) Halt.

The colors and sizes of the marbles do not matter; only the quantity does.

Initially, the bowls have some given number of marbles as input. When and if a program halts, the number of marbles in a designated bowl is the program's output.

For example, the following is a 4-counter program for multiplying the quantities initially in bowls A and B. Bowls C and D start out empty. The product of A and B will be in D at the end. Bowl C serves as a holding area.

***S*: If A is empty, continue at *H*.**
Remove a marble from A.
***L*: If B is empty, continue at *R*.**
Remove a marble from B.
Place a marble in C.
Place a marble in D.
Continue at *L*.
***R*: If C is empty, continue at *S*.**
Remove a marble from C.
Place a marble in B.
Continue at *R*.
***H*: Halt.**

It is a fact that three bowls suffice to compute any computable single-argument function over the natural numbers, but to compute them all with only two bowls is only possible with an encoding such as 2^i for *i*.

IMAGE BY THPSTOCK

functions, then it would be easy to come up with a bespoken "deviant encoding" that makes a single uncomputable function appear computable. For another thing, we must insist that the same encoding c be used both for the inputs x_i as well as for the output of f, or else everything can easily go belly-up (pace Butterfield et al.[8]). Ideally, the restrictions would ensure that (unlike for counter machines, or the lambda calculus, for that matter) no allowed encoding can expand the class of computed functions. Specifically, we must preclude the endowing of Turing's machines or the recursive functions with superpowers (what is termed "hypercomputation"). How can we guarantee this? Shapiro[21] has submitted that for an encoding of (Platonic) numbers to be "acceptable," the encoded successor function should also be computable by M'. In other words, M' must include a function s': $c(n) \mapsto c(n+1)$ simulating successor. I agree. But why successor?

Mercifully, encoding turns out not to be a problem for the usual use cases. Indeed, no encoding whatsoever can break the Turing barrier. Specifically, one can prove that there is no (injective) encoding that allows one to simulate all the computable functions plus an incomputable one like halting.[3] It turns out, in fact, that simulating the successor function effectively is necessary and sufficient to guarantee that Turing machines cannot simulate (under a single-valued encoding) anything unexpected.[4] And this is all we need for the big picture to remain intact.

On the other hand, with just a bit of effort, one can devise a recursive function (a modification of Ackermann's function) that cannot be simulated by primitive recursion regardless of the encoding.[3] So it thankfully remains true that primitive recursion is strictly weaker than recursion—in the very strong sense that no (injective) encoding whatsoever would endow primitive recursion with full Turing power. Were it not likewise provable that 1-counter machines cannot simulate all recursive functions, statements like "Combining these simulations, we see that two-counter machines are as powerful as arbitrary Turing machines (one-counter machines are strictly less powerful)"[12] would be indefensible.

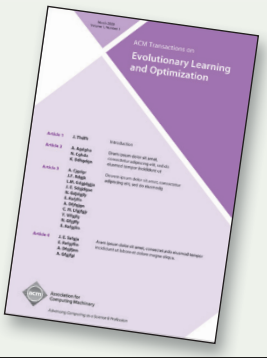
Turning to formal languages (sets of words over some alphabet), the situation is reversed. Encodings are bad; inclusion is good. Homomorphic mappings may preserve the relative power of most language models (with their purely local impact on the structure of strings), but more general injections or bijections do not. In fact, there is a nefarious bijection between the words of any (nonsingular) alphabet with the disconcerting property that all the regular *plus* all the context-free languages can be recognized by mere finite-state automata. The situation is actually infinitely more intolerable: one can at the same time also recognize countably many arbitrary undecidable languages with vanilla finite automata via such a mischievous bijection.[10]

In the case of languages, then, we are compelled to adhere to straightforward inclusion and ban (even computable) mappings of input strings when comparing the power of language models. Earlier, when dealing with (all) the computable functions, we did have the flexibility of simulating via mappings, but that was because the same mapping is also applied to the full range of possible function outputs.[5]

Association for Computing Machinery

ACM Transactions on Evolutionary Learning and Optimization (TELO)

ACM Transactions on Evolutionary Learning and Optimization (TELO) publishes high-quality, original papers in all areas of evolutionary computation and related areas such as population-based methods, Bayesian optimization, or swarm intelligence. We welcome papers that make solid contributions to theory, method and applications. Relevant domains include continuous, combinatorial or multi-objective optimization.



For further information and to submit your manuscript, visit telo.acm.org

Related Concerns

There is another ubiquitous use of encodings that similarly requires extra caution. Oftentimes, one wishes to compute a function on objects other than strings or numbers, such as graphs, logical formulae, or computer programs. For that purpose, one must somehow represent those objects in the input/output language of the computational model that is to manipulate them, typically strings or numerals. To quote: "A necessary preliminary to applying our work on computability ... is to code expressions by numbers .... There are many reasonable ways to code finite sequences, and it does not really matter which one we choose."[6]

To be "reasonable," however, one needs to be sure that the encoding does not do anything beyond faithfully representing the input.

For example, it is a trivial matter to concoct an encoding of Turing machines that turns an undecidable problem about machines into a readily computable one. Let w_0, w_1, ... be an enumeration of all binary strings (over the alphabet $\{0, 1\}$), and let M_0, M_1, ... be some enumeration of all Turing machines (over that input alphabet). The following are four typical decidability questions:

$T(i,j)$: machine M_i halts on input w_j.

$H(i)$: machine M_i halts on input w_0.

$U(i)$: machine M_i halts on all inputs w_0, w_1 ...

$D(i)$: machine M_i halts on input w_i.

Halting on a single particular input (like the empty word) is just the parity problem if one reorders a standard enumeration of machines so that the odd-numbered ones halt on that input while the even ones do not. The snag with such an encoding of Turing machines is that it also makes ordinary tasks incomputable. Specifically, once could not modify the code of a given machine to act in some related but different way, because one would need to ascertain the termination behavior of the modified machine. So whether problem H is decidable or not actually depends on exactly how machines are encoded.

Consider an assertion such as the following:

▸ One of Turing's key insights was the Halting Problem H (which takes an integer n and outputs $H(n) = 1$ if and only if $n = \langle P \rangle$ is an encoding of a valid [self-contained] program P and P terminates) is "undecidable."[7]

It would be an unmitigated disaster to abandon simulations, since the idea that all our traditional unrestrained models are of equivalent power, despite operating with different entities, stands at the core of computability theory, as enshrined in the Church-Turing thesis.

For your usual, straightforward encodings of machine descriptions, the problem is indeed undecidable, but for any number of alternative encodings it becomes decidable. The same goes for the "universal halting" problem U.

On the other hand, the more basic halting problem, $T(i,j)$, which asks about the behavior of the ith machine on the jth possible input, is undecidable regardless of how machines are encoded or how inputs are enumerated. Nevertheless, one must be careful how pairs $\langle i,j \rangle$ are encoded for models of computation—such as run-of-the-mill Turing machines—that allow only a single input representing both i and j. Similarly, the "diagonal" language D, consisting of the indices of those machines that halt when the input string has the same index in its enumeration as does the machine in its encoding, is not computable for any and all machine encodings and string enumerations. So it is true that "no encoding [of all Turing machines as numbers] can represent a TM M such that $L(M) = L_d$ [the diagonal language]," as claimed in

Hopcroft et al.,[13] but the same immunity to encoding does not hold true for the collection of machines that accept nothing (L_e).[13] Regrettably, no textbook I have seen clarifies which encodings of machines are valid and for what purpose and why. Nothing in the following remark, for example, precludes a representation from incorporating a finite amount of uncomputable information about the represented machine, such as whether it always terminates or halts on a specific input:

▸ The details of the representation scheme of Turing machines as strings are immaterial [as long as]: (1) We can represent every Turing machine as a string. (2) Given the string representation of a Turing machine M and an input x, we can simulate M's execution on the input x.[1]

The standard part of a malicious string encoding would allow one to simulate execution as usual, while tacked-on extras can allow an algorithm to decide otherwise undecidable questions about them. Overzealous encoding is not, however, a problem in programming languages that pass unadulterated programs as arguments, sans encoding.

As a final comment, when it comes to complexity comparisons, everyone realizes that representation is an issue to be taken into account, but the requirements remain vague: "The intractability of a problem turns out to be essentially independent of the particular encoding scheme ... used for determining time complexity ... It would be difficult to imagine a 'reasonable' encoding scheme for a problem that differs more than polynomially from the standard ones ... What we mean here by 'reasonable' cannot be formalized ..."[11]

It would seem to me that standard string and image compression schemes are perfectly reasonable encodings, despite reducing size exponentially in many cases. In any event, a formal, principled definition of "reasonableness" is still sorely lacking for the theory of complexity. (But see Boker and Dershowitz[5] for one proposal.)

Takeaway

To recapitulate the main points of the problem raised here:

▸ Every single course in automata or computability utilizes set inclusion as the means of comparing the computational power of different formalisms for language definition.

▸ Virtually every such course claims equivalence of a wide variety of models of computation in support of the Church-Turing thesis, an equivalence that is based on mutual simulations.

▸ These two notions are logically incompatible as we have witnessed.

▸ No textbook nor any instructor I have encountered recognizes, let alone addresses, this fundamental inconsistency.

At a bare minimum, then, we must make the following changes in the manner this subject is traditionally taught:

▸ One should use set inclusion only as a means to compare classes of formal languages, such as in the demonstration that context-free grammars are a strictly more inclusive formalism than are regular expressions.

▸ We should never use set inclusion to compare the power of primitive recursion with general recursion, or **for**-loop programs with **while**-loop ones, or one-counter machines with two counters, without mentioning that it has in fact been demonstrated that the one can also not *simulate* all of the other.

▸ Instructors ought to emphasize that one must always be careful with encodings, as they easily alter computational power, while pointing out it has been proved this is not an issue for the usual use case of Turing-level computability.

▸ One should definitely avoid using halting-on-empty-tape, or empty-language acceptance, or similar problems as fundamental examples of undecidability, as their decidability is encoding-dependent. Instead, we need to explicate the subtle role of input encodings when reducing the standard two-input halting problem to those other problems.

▸ We should be cautious to never say or imply that two-counter machines recognize all recursively enumerable languages (they do not), nor that they *compute* (as opposed to simulate) all Turing-computable functions.

▸ One should not choose the lambda calculus as a primary exemplar of a fully empowered computational model (since it simulates *more* than it computes). C

References

1. Barak, B. *Introduction to Theoretical Computer Science.* 2020. Online text (version of Dec. 25, 2020); https://bit.ly/3bZnLVi
2. Barzdins, J.I. Ob odnom klasse machin turinga (machiny minskogo) [On one class of Turing machines (Minsky machines)]. *Algebra i Logika* [*Algebra and Logic*] *1*, 6 (1963), 42–51. In Russian.
3. Boker, U. and Dershowitz, N. Comparing computational power. *Logic Journal of the IGPL 14*, 5 (2006), 633–648; https://bit.ly/3cEzd99
4. Boker, U. and Dershowitz, N. The Church-Turing thesis over arbitrary domains. In A. Avron, N. Dershowitz, and A. Rabinovich, Eds., *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of Lecture Notes in Computer Science, Springer, Berlin, 2008, 199–229; https://bit.ly/3eUun99
5. Boker, U. and Dershowitz, N. Honest computability and complexity. In E. Omodeo and A. Policriti, Eds., *Martin Davis on Computability, Computational Logic & Mathematical Foundations*, volume 10 of Outstanding Contributions to Logic Series, Springer, Cham, Switzerland, 2017, 153–175; https://bit.ly/3cH8589
6. Boolos, G.S., Burgess, J.P., and Jeffrey, R.C. *Computability and Logic.* Cambridge University Press, Cambridge, U.K., 4th edition, 2002.
7. Braverman, M. Computing with real numbers, from Archimedes to Turing and beyond. *Commun. ACM 56*, 9 (Sept. 2013), 74–83.
8. Butterfield, A., Ngondi, G.E., and Kerr, A., Eds. Machine simulation entry. *A Dictionary of Computer Science.* Oxford University Press, Oxford, U.K., 7th edition, 2016.
9. D'Souza, D. and Shankar, P. *Modern Applications of Automata Theory.* World Scientific, River Edge, NJ, 2011.
10. Endrullis, J., Grabmayer, C., and Hendriks, D. Regularity preserving but not reflecting encodings. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Kyoto, Japan, July 2015. IEEE Computer Society), 535–546; https://bit.ly/3cDSR3M
11. Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, NY, 1979.
12. Harel, D., Kozen, D., and Tiuryn, J. *Dynamic Logic.* MIT Press, Cambridge, MA, 2000.
13. Hopcroft, J.E., Motwani, R., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education, Boston, MA, 3rd edition, 2007.
14. Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.
15. Ibarra, O.S. and Trân, N.Q. A note on simple programs with two variables. *Theor. Comput. Sci. 112*, 2 (1993), 391–397.
16. Lipton, R.J. and Regan, K.W. Minsky the theorist. (Jan. 27, 2016); https://bit.ly/2P45zRn
17. Minsky, M.L. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, NJ, 1967.
18. Montague, R. Towards a general theory of computability. *Synthese 12*, 4 (1960), 429–438.
19. Rogers, Jr., H. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, NY, 1966.
20. Schroeppel, R. A two counter machine cannot calculate 2^N. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, 1972; ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-257.pdf
21. Shapiro, S. Acceptable notation. *Notre Dame Journal of Formal Logic 23*, 1 (1982), 14–20.
22. Sipser, M. *Introduction to the Theory of Computation.* Thomson, Boston, MA, 2nd edition, 2006.
23. Sommerhalder, R. and van Westrhenen, S.C. *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility.* Addison-Wesley, Workingham, England, 1988.
24. Turing, A.M. Computability and λ-definability. *The Journal of Symbolic Logic 2*, 4 (1937), 153–163; https://bit.ly/3eOJEbF

Nachum Dershowitz (nachum@tau.ac.il) is a Professor (Emeritus) in the School of Computer Science, Tel Aviv University, Ramat Aviv, Israel.

The author thanks Jörg Endrullis for his perceptive comments and Centrum Wiskunde and Informatica (CWI) for its hospitality.

Copyright held by author.