Parallelization by Simulated Tunneling

Amos Waterland,[1,*] Jonathan Appavoo,[2] and Margo Seltzer[1]

[1] *Harvard University*
[2] *Boston University*

As highly parallel heterogeneous computers become commonplace, automatic parallelization of software is an increasingly critical unsolved problem. Continued progress on this problem will require large quantities of information about the runtime structure of sequential programs to be stored and reasoned about. Manually formalizing all this information through traditional approaches, which rely on semantic analysis at the language or instruction level, has historically proved challenging. We take a lower level approach, eschewing semantic analysis and instead modeling von Neumann computation as a dynamical system, i.e., a state space and an evolution rule, which gives a natural way to use probabilistic inference to automatically learn powerful representations of this information. This model enables a promising new approach to automatic parallelization, in which probability distributions empirically learned over the state space are used to guide speculative solvers. We describe a prototype virtual machine that uses this model of computation to automatically achieve linear speedups for an important class of deterministic, sequential Intel binary programs through statistical machine learning and a speculative, generalized form of memoization.

Introduction

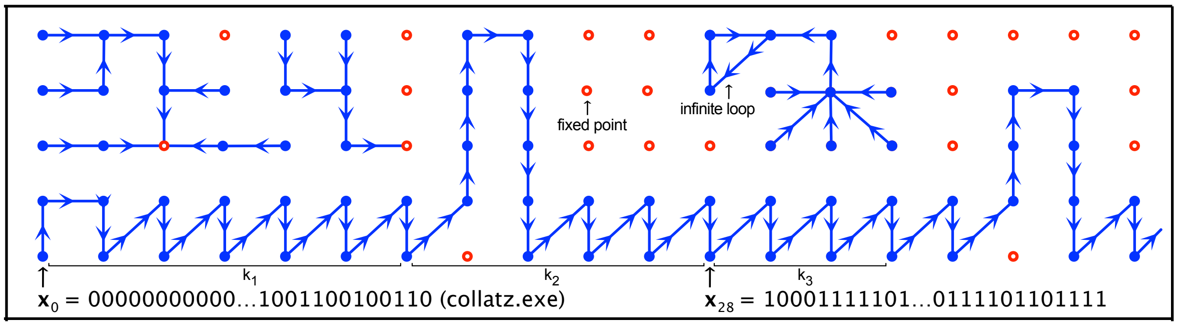
Major challenges still remain in making efficient use of highly parallel computing systems. Despite decades of sophisticated research in automatic parallelization, we still do not have computers that we can program in the sequential model best suited to the human mind, but whose performance automatically scales with the number of processors. In this paper, we take a radically different approach to automatic parallelization; rather than using the semantics of programming languages to identify opportunities for parallelism, we model computation at a lower semantic level in exchange for a powerful new set of conceptual and mathematical tools. In particular, this model allows us to automatically identify opportunities for parallelization via Fourier analysis and statistical machine learning.

Over the past century, physicists and mathematicians have distilled concepts developed across disparate fields whose objects of study are high-dimensional and time-dependent into a model called the *dynamical system.* This model has only two core mathematical *objects*—a state space and an evolution rule—but has a huge body of *concepts* and mathematical *techniques.* In this paper, we describe a concrete, practical way to model von Neumann computation as a dynamical system, and show how this gives a natural way to use probabilistic inference to organize and exploit large quantities of information about the runtime structure of sequential binary programs.

The dynamical systems model of computation is deeply related to familiar ways of thinking about computation; *e.g.* operational semantics and finite state automata, but allows us to bring to bear powerful new conceptual and mathematical tools. In particular, in this model the state of computation is just a *vector*, a trace of computation is just a *matrix*, and we have a *geometry* of computation complete with *distance* and *angle* between states of computation. This model gives a natural way to apply *inference*, in the form of Bayesian posterior maximization and deep artificial neural networks, through highly parallel execution of Markov Chain Monte Carlo, simulated annealing, and genetic algorithms.

In our model, the complete state of a computer is represented as the coordinates of a point in our *state space.* Computation is effected by repeatedly applying our *evolution rule*, which maps each point in the state space to its successor point by simulating the instruction encoded in the coordinates. A sequence of such transitions forms a path through state space called a *trajectory.* Some trajectories terminate in *fixed points* – points that are mapped to themselves by the evolution rule. The result of computation is obtained by waiting until the trajectory being solved halts at its fixed point, then reading out the answer from the coordinates. Programming is effected by preparing an *initial condition* – selecting a point in state space whose coordinates represent the initial values of the registers, machine code, and data.

We have designed and prototyped a system in which a collection of virtual machines share the same state space and evolution rule, but are given different initial conditions. These virtual machines, in operation on heterogeneous manycore processors or networked clusters, *cooperate* in parallel to accelerate each other's execution. Each virtual machine maintains a state vector representation of its simulated computer, and has a simulation loop that calls the evolution rule – which simulates one Intel instruction per invocation. Each virtual machine also maintains a compressed database of *initial* and *final points* of state space trajectories it has already solved, and when it has no other work to do, garbage collects and updates its database by speculatively solving trajectories given *predictions* for regions of state space that



fixed point
infinite loop
k1
k2
k3
x0 = 00000000000...1001100100110 (collatz.exe)
x28 = 10001111101...0111101101111

FIG. 1: *Schematic* representation of a tiny volume of the state space of computation. Each point is an n-dimensional bit vector whose coordinates represent the entire state of a computer. Unfilled circles are fixed points of the evolution rule – halting states. The initial condition of `collatz.exe`, an example studied in this paper, is shown as $\mathbf{x}_0$.

other virtual machines are likely to visit.

At regular intervals, each virtual machine broadcasts its current state vector to the other virtual machines, all of whom in response search their databases for a match. A match is found when the broadcast state vector is equivalent under a *symmetry transform* to the initial point of a previously solved trajectory. This trajectory was either speculatively solved given a prediction or lies on an unrelated trajectory whose initial condition is a different program that for a time executed a sequence of instructions identical to those of the current program. In both cases, the virtual machine that found a match applies the inverse symmetry transform and replies with the final point of the matched trajectory.

Upon receiving the reply, the virtual machine that sent the broadcast then jumps forward—in state space and simulation time—from its current point to the final point it received in reply. This instantaneous jump through state space—which skips over the many applications of the evolution rule that it would have had to do—is called *tunneling*. There is no free lunch, as some virtual machine somewhere had to solve each trajectory up to symmetries, but from the perspective of the virtual machine that sent the broadcast – it has been accelerated.

Tunneling can be seen as a speculative, generalized form of *memoization*. Traditional memoization speeds up programs by building a table of the inputs and outputs of *pure functions*. Once a memoized function has been executed for a particular input, it never has to be executed again when called with the same input; the result can simply be looked up in the table. Tunneling is a generalization of memoization in that it can jump forward from *any* program location – not just at a function boundary, and it can use results from one program to speed up a different program. It is speculative in that instead of just storing the results of computation that has already happened, it solves trajectories in the hope that they will be useful later to other virtual machines. These tunnels can be seen as warping the state space of computation so that useful and important trajectories are drastically compressed.

This system design is probabilistic, in that it uses Bayesian inference to calculate the predictive probability distributions used by speculative trajectory solvers, but it is not probabilistic or approximate computing in the sense that it might halt with the wrong result. When predictions are poor, the worst that can happen is that communication and simulation overhead is not recovered. We preserve the deterministic sequential programming model, our initial conditions are prepared by gcc, and the result of simulation is identical to that of running the input binary program on a uniprocessor Intel computer.

Our use of Bayesian inference gives a natural division of work on heterogeneous systems – in which speculative trajectory solvers run on full-featured cores, but feature extraction and predictive inference is done by massive arrays of simple processors in GPUs. Our vision is that on a single many-core computer a small collection of our virtual machines will accelerate unmodified sequential binary programs acceptably, but performance improves dramatically when a GPU is present, and improves further when a network is available through which the databases of remote virtual machines can be queried. By pursuing this vision at the runtime level, our Bayesian predictors are able to exploit our experimental evidence that there is significant *statistical structure*—not accessible at compile time—in the runtime behavior of real programs on real data that causes their dynamics to be confined to low dimensional manifolds in state space.

The contributions of this paper are: (1) showing how the dynamical systems model of computation can be operationalized, (2) showing how this model yields a new approach to automatic parallelization, and (3) evaluating a prototype implementation of the model and approach.

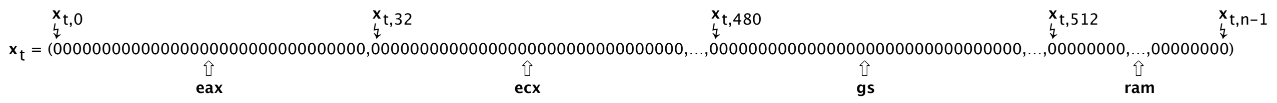

x_{t,0}
x_{t,32}
x_{t,480}
x_{t,512}
x_{t,n−1}
x_t = (00000000000000000000000000000000,00000000000000000000000000000000,…,00000000000000000000000000000000,…,00000000,…,00000000)
eax
ecx
gs
ram

FIG. 2: State vector organization.

Model of Computation

A number of physicists have previously observed that computers can be abstractly modeled as dynamical systems [1–4, 6, 8]. *Definition* – a discrete time, finite dynamical system is a mathematical structure $(\mathcal{X}, f)$, where $\mathcal{X}$ is a finite set called the *state space*, and f is a map from $\mathcal{X}$ to $\mathcal{X}$ called the *evolution rule.*

This model might seem quite abstract. However, we have shown that we can *operationalize* it by building a carefully designed virtual machine. The internals of this virtual machine are explicitly organized as a dynamical system, but the external interface is just an Intel architecture functional simulator that runs programs compiled by gcc. These two design decisions combine to make it possible to get real, practical engineering utility out of the model's mathematical results.

When selecting our state space $\mathcal{X}$, we weighted heavily the following considerations. Above all, we want to avoid confinement to combinatorics, and to have a meaningful *geometry* of computation [7] imposed by a *distance* and *angle* between computer states. We want to meaningfully *subtract* two states to obtain a difference vector. In sum, we want our state space to have the structure of an abelian group, and for it to be easy to *embed* this group in $\mathbb{R}^d$, so that we may bring to bear the mathematical structure of a vector space with an inner product and metric. In particular, an embedding in $\mathbb{R}^d$ allows us to bring to bear much of modern machine learning. These considerations drove us to what is perhaps the obvious choice. Our *state space* $\mathcal{X}$ is just the set of n-dimensional bit vectors, constructed as the n-fold product of $\mathbb{Z}_2$: $\mathcal{X} = \mathbb{Z}_2 \times \mathbb{Z}_2 \times \ldots \times \mathbb{Z}_2 = \mathbb{Z}_2^n$.

Our virtual machine simulates the Intel instruction set at the user mode privilege level, so we have simply that $n = 16 \times 32 + m$, which corresponds to the sixteen user mode visible 32-bit registers and m bits of memory. We do not model a disk or any I/O devices, and all input data is specified at launch time. Since the smallest possible program consists of one `hlt` instruction, we have that n is bounded below by $16 \times 32 + 8 = 520$. The space $\mathbb{Z}_2^{520}$ then serves as the *fundamental subspace* for all programs. When preparing its initial condition, the virtual machine chooses $n \geq 520$ based on the `.text` segment of the input binary program and an estimated heap and stack size. At simulation time step t, the current coordinates in state space are represented by the *state vector* $\mathbf{x}_t \in \mathcal{X}$. Figure 2 shows how we organize the state vector so that the leftmost coordinates represent the sixteen 32-bit user-visible registers. The first 32 coordinates correspond to `eax`, the next 32 coordinates to `ecx`, and so forth in the usual Intel order. After the coordinates of `gs`, the 16th register, we have the 8 coordinates corresponding to the lowest byte of memory with physical address `0x0`, and so forth. Our *evolution rule* f maps the current state vector $\mathbf{x}_t$ to its successor $\mathbf{x}_{t+1}$ as a time-invariant map having the same domain and codomain: $f : \mathcal{X} \to \mathcal{X}$, *i.e.* $f : \mathbb{Z}_2^n \to \mathbb{Z}_2^n$.

Note that f is a *single* rule, but it is *parametrized* by its argument. That is, what distinguishes dynamical systems capable of computation from other dynamical systems is that their evolution rule can be "programmed" to simulate other rules by suitably arranging its argument. Our virtual machine's internal representation of f is as a pointer to a function whose signature is `evolution(uint8_t *y, uint8_t *x, int n)`. This function maps the state vector $\mathbf{x}$ to its successor state vector $\mathbf{y}$ by simulating a single instruction and then returning. Internally it is just a straightforward Intel architecture simulator. The main loop of the virtual machine repeatedly calls the evolution function until it reaches a fixed point, which causes it to halt. Figure 1 gives a 2-dimensional schematic representation of this model of computation – the true geometry is of course n-dimensional. Each point in the figure corresponds to an n-dimensional state vector. The space is partitioned into trajectories. Some trajectories do not have a fixed point and correspond to infinite loops. Trajectories cannot branch, but they can merge, since the system is forward but not backward deterministic.

New Approach to Parallelization

At an intuitive geometric level, our parallelization approach is to try to divide a program's state space trajectory into segments that can be solved in parallel. At a more precise algebraic level, note that in *every* dynamical system the evolution rule has by definition the same domain and codomain. The dynamics—which simulate von Neumann computation in our case—are produced by repeated *compositions* of the evolution rule – our computer state at time step k is $\mathbf{x}_k = f(f(\cdots f(\mathbf{x}_0)\cdots))$. A standard result in dynamical systems theory is that evolution rules under composition have the algebraic structure of a *composition monoid.* That is, the *iterates* of f form a structure $(\{f^{(0)}, f^{(1)}, \ldots, f^{(k)}\}, \circ)$ that is closed, associative, and has an identity. So the computer state at time step k can also be written as

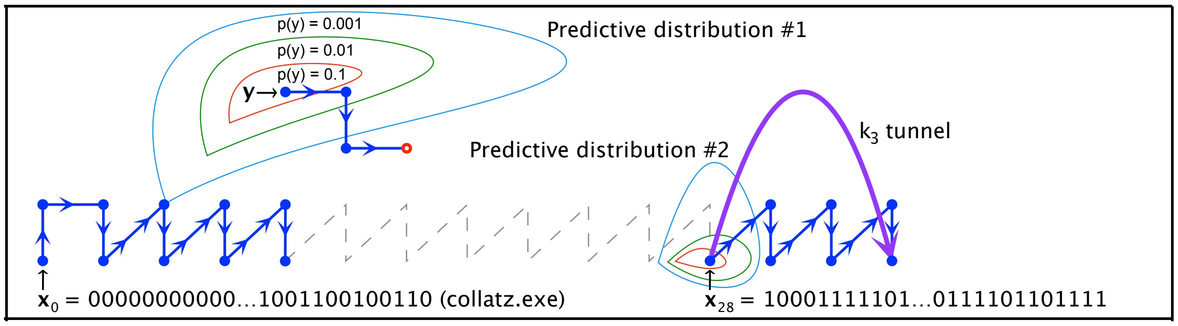

p(y) = 0.001
p(y) = 0.01
p(y) = 0.1
y→
Predictive distribution #1
Predictive distribution #2
k₃ tunnel
x₀ = 00000000000...1001100100110 (collatz.exe)
x₂₈ = 10001111101...0111101101111

FIG. 3: The *same* region of state space as in Figure 1, but instead of an oracle view it represents the system's *beliefs*.

$\mathbf{x}_k = f \circ f \circ \cdots \circ f(\mathbf{x}_0) = f^{(k)}(\mathbf{x}_0)$, and in particular as $\mathbf{x}_k = f^{(k_1)} f^{(k_2)} \cdots f^{(k_p)}(\mathbf{x}_0)$, where $\sum k_i = k$. The key idea behind our approach is to automatically parallelize sequential computation by solving the $f^{(k_i)}$ in parallel.

In the schematic of Figure 1, this corresponds to solving segments k_1 and k_3 in parallel. But in order to solve segment k_3 the system would need to know $\mathbf{x}_{28}$. Of course, if we are given only $\mathbf{x}_0$ *a priori*, the only way to *know* $\mathbf{x}_{28}$ is to solve it directly as $\mathbf{x}_{28} = f^{(28)}(\mathbf{x}_0)$, so we don't seem to have saved ourselves anything. However, our virtual machine simulates deterministic, sequential binary programs. It does not simulate an interrupt controller, and is a *program* virtual machine rather than a *system* virtual machine – it runs one program at a time with all input data encoded in $\mathbf{x}_0$. The real time clock is not simulated and programs are not allowed to issue instructions that read from it. So the notation $\mathbf{x}_{28}$ must be understood as: the point in state space reached after 28 compositions of f on $\mathbf{x}_0$, and in turn $\mathbf{x}_0$ must be understood as: the point in state space that represents the initial condition of some program. So the *same* point can be labelled $\mathbf{x}_i$ and $\mathbf{x}_j$ where $i \neq j$, and the time index must be understood as *relative* to some initial condition. Now, the usual way to prepare an initial condition is by loading a program into memory and suitably initializing the instruction and stack pointers, but *any* point in state space can serve as an initial condition, since we are working with a dynamical system. This fact is key to our system design. We try to *predict* that $\mathbf{x}_{28}$ is likely to lie on the trajectory, and then *speculatively* solve segment k_3. If a trajectory solver ever—today or years from now—reaches $\mathbf{x}_{28}$, it can *tunnel* instantaneously to the final point of k_3. So we have done a reduction of the hard problem of parallelizing sequential von Neumann computation to another—possibly equally hard—problem of predictive *inference* in a dynamical system.

We share the view of Jaynes [5] that a powerful—and in a sense, the *only consistent*—way to do inference is Bayesian probability. So our predictions are samples from a posterior predictive distribution calculated from a likelihood that encapsulates our transition model and a prior that summarizes our domain knowledge. Our basic predictive building blocks are conditional distributions that give $p(\mathbf{x}_{t+k}|\mathbf{x}_t)$ for any current state $\mathbf{x}_t$ and any offset k into the future.

We show a schematic representation of this view in Figure 3, which is the same tiny volume of state space as in Figure 1, but instead of an oracle view it shows a schematic representation of the system's *belief*—under two different predictive probability distributions—after 8 time steps. At time step 4, predictive distribution #1 issued a prediction for the point denoted $\mathbf{y}$, and predictive distribution #2 issued a prediction for the point denoted $\mathbf{x}_{28}$. Referring to Figure 1 we see that the first prediction will be wasted, but the second will be used to accelerate computation by going through the k_3 tunnel.

Prototype Implementation

In this section we focus on a base case – programs with loops for which each iteration is independent of the others. These programs have been heavily studied in the compiler literature, where they are called "DOALL loops," and encompass a surprising amount of computation that humans are interested in. We

```
void main() {
    unsigned int i, j;
    for (i = 1; ; i++) {
        for (j = i; j > 1; ) {
            if (j % 2 == 0)
                j = j / 2;
            else
                j = 3 * j + 1;
        }
    }
}
```

Listing 1: Source code for collatz.c

study in particular the `collatz.c` kernel given in Listing 1 because it is a short `C` program with complex runtime dynamics. This program implements a search for a counterexample to the famous *Collatz Conjecture* – the hypothesis that the inner loop always terminates. For some integers the inner loop is chaotic and is known to take millions of iterations, but thus far no one has found an integer for which it does not eventually terminate.

Most successful applications of Bayesian machine learning rely heavily on a good *inductive bias* – a set of assumptions about the problem domain, and good *feature extraction* – transformations from the raw state space into a dimensionally-reduced feature space in which the inference problem is easier. Our virtual machine is organized as an *ensemble* – so it is easy for us to plug in predictors built from different inductive biases and feature extractors. This results in the same points in state space being assigned *different* probabilities by different predictors, as depicted in Figure 3. In addition, since calculating normalized probabilities is often intractable due to the partition function, our virtual machine only requires that predictors produce an energy functional $E : \mathbb{R}^d \to \mathbb{R}$. That is, $E(\mathbf{y})$ maps a feature vector to a real number, where small numbers are "good". When the virtual machine needs a prediction in order to select the initial condition for a speculative trajectory solver, it begins with an effective temperature T and an initial prediction $\mathbf{y}'$, then proposes a move $\mathbf{y}''$, which it accepts with probability proportional to the ratio $e^{-E(\mathbf{y}'')/T}/e^{-E(\mathbf{y}')/T}$. It repeats this process to do in effect simulated annealing into a basin in the energy landscape – then selects a point from the bottom of the basin as its prediction. In summary, a predictor just takes in a *training set* of feature vectors and produces an energy functional $E(\mathbf{y})$ – the virtual machine takes care of the rest. In the following, we develop an inductive bias and feature extractor optimized for the base case class of programs.

Most programs in the base case class have an integer *induction variable* controlling their outer loop – `i` in Listing 1. So our feature extraction function takes a state vector $\mathbf{x}_t$ and maps each 32-bit register and word of memory to an integer – resulting in a *feature vector* $\mathbf{y}_t \in \mathbb{R}^{n/32}$. When we study the geometry of the dynamics in feature space, we see that under the L_2-norm the system at irregular intervals always returns *near* elements of a small set of accumulation points. This is caused by the fact that a good optimizing compiler like gcc will in effect try to minimize entropy – only keep information in the registers and stack that it thinks the program really needs. In particular, since each iteration of the outer loop is independent, the machine code generated by gcc results in the *entire* state of the computer being one of a small set of accumulation states between iterations of the outer loop – modulo a change to the feature that contains the induction variable. So our predictor's inductive bias is that there is one configuration of the computer that occurs repeatedly – it just has to find it modulo changes to the induction variable feature. In its training phase, our predictor marginalizes out each component of the feature space, selects one denoted m that most reduces the empirical entropy, and selects the mode μ of the resulting dimensionally-reduced space. Its energy functional is then $E(\mathbf{y}) = ||\mu - \mathbf{y}_{[-m]}||_2$. Intuitively, the closer a proposed feature vector is to the mode μ—ignoring the contents of feature m—the lower its energy so the higher the probability it will be accepted as a prediction.

Our prototype virtual machine is a parallel MPI program in operation on a Blue Gene supercomputer. One of its predictors is the one we have just described. In Figure 4 we show the scaling results it achieves on `collatz.exe`. The Intel architecture simulator encoded in our evolution rule is not yet tuned, so the figure shows the raw number of calls to

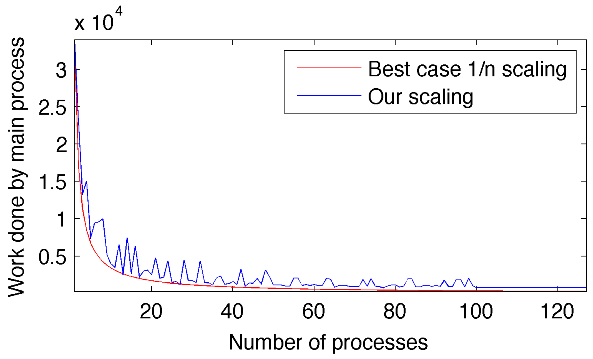


FIG. 4: Scaling results for Collatz kernel.

the evolution rule the main virtual machine had to do – the more times it tunneled the fewer calls to the evolution rule it had to do.

One could imagine refactoring the code in Listing 1 to an outer loop that repeatedly calls a function and then "speculatively memoizing" the function. However, the function often returns fairly quickly – so memoization proper is not worth the overhead. However, our virtual machine is in fact able to profitably parallelize this refactored code, because it "memoizes" trajectories – each of which contain several hundred successive invocations of the function.

Conclusion

Scientific progress is sometimes made by turning hard problems into different hard problems for which we have better mathematical tools. We have turned the problem of parallelizing sequential von Neumann computation into one of predictive inference in a dynamical system. Bayesian predictive inference has seen rapid growth in recent decades due to increased high-performance computing resources, but turning inference on von Neumann computation *itself* into a high-performance computing job has received little attention. In this paper we described a vision for doing precisely this, and reported our current experimental progress toward validating this vision.

* Electronic address: apw@seas.harvard.edu

[1] Henry G. Baker, *Thermodynamics and garbage collection*, SIGPLAN Not. **29** (1994), no. 4, 58–63.

[2] Roger W. Brockett, *Dynamical systems that sort lists, diagonalize matrices and solve linear programming problems*, Proc. 27th IEEE Conf. Dec. and Control (Austin, TX), Dec. 1988, pp. 799–803.

[3] Marco Giunti, *Computation, dynamics, and cognition*, Oxford University Press, 1997.

[4] John J. Hopfield, *Hopfield network*, Scholarpedia **2** (2007), no. 5.

[5] E.T. Jaynes, *Probability theory: The logic of science*, Cambridge University Press, 2003.

[6] Todd Mytkowicz, Amer Diwan, and Elizabeth Bradley, *Computer systems are dynamical systems*, Chaos: An Interdisciplinary Journal of Nonlinear Science **19** (2009), no. 3, 033124.

[7] Gerald Jay Sussman, Personal communication: How the Shape of a Computational Process is Controlled by the Program, 2010.

[8] Tommaso Toffoli, *Action, or the fungibility of computation*, pp. 349–392, Perseus Books, Cambridge, MA, USA, 1999.