A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory

Adam Yedidia
Scott Aaronson
Massachusetts Institute of Technology
adamy@mit.edu
aaronson@csail.mit.edu

Since the definition of the busy beaver function by Rado in 1962, an interesting open question has been the smallest value of n for which $BB(n)$ is independent of Zermelo–Fraenkel set theory with the axiom of choice (ZFC). Is this n approximately 10, or closer to 1 000 000, or is it even larger? In this paper, we show that it is at most 7910 by presenting an explicit description of a 7910-state Turing machine Z with one tape and a two-symbol alphabet that cannot be proved to run forever in ZFC (even though it presumably does), assuming ZFC is consistent. The machine is based on work of Harvey Friedman on independent statements involving order-invariant graphs. In doing so, we give the first known upper bound on the highest provable busy beaver number in ZFC. To create Z, we develop and use a higher-level language, Laconic, which is much more convenient than direct state manipulation. We also use Laconic to design two Turing machines, G and R, that halt if and only if there are counterexamples to Goldbach's conjecture and the Riemann hypothesis, respectively.

1. Introduction

1.1 Background and Motivation

Zermelo–Fraenkel set theory with the axiom of choice, more commonly known as ZFC, is an axiomatic system invented in the twentieth century that has since been used as the foundation of most of modern mathematics. It encodes arithmetic by describing natural numbers as increasing sets of sets.

Like any axiomatic system capable of encoding arithmetic, ZFC is constrained by Gödel's two incompleteness theorems. The first incompleteness theorem states that if ZFC is *consistent* (it never proves both a statement and its opposite), then ZFC cannot also be *complete* (able to prove every true statement). The second incompleteness theorem states that if ZFC is consistent, then ZFC cannot prove its own consistency. Because we have built modern mathematics on top of ZFC, we can reasonably be said to have assumed ZFC's consistency. This means that we must also believe that ZFC cannot prove its own consistency. This fact carries with it certain surprising conclusions.

In particular, consider a Turing machine Z that enumerates, one after the other, each of the provable statements in ZFC. To describe how such a machine might be constructed, Z could iterate over the axioms and inference rules of ZFC, applying each in every possible way to each conclusion or pair of conclusions that had been reached so far. We might ask Z to halt if it ever reaches a contradiction; in other words, Z will halt if and only if it finds a proof of $0 = 1$. Because this machine will enumerate every provable statement in ZFC, it will run forever if and only if ZFC is consistent.

It follows that Z is a Turing machine for which the question of its behavior (whether or not it halts when run indefinitely) is equivalent to the consistency of ZFC. While we will talk about ZFC throughout this paper, rather than simple ZF set theory, this is simply a convention. For our purposes, the axiom of choice is irrelevant: the consistency of ZFC is equivalent to the consistency of simple ZF set theory [1], and ZFC and ZF prove exactly the same arithmetical statements (which include, among other things, statements about whether Turing machines halt) [2]. Therefore, just as ZFC cannot prove its own consistency (assuming ZFC is consistent), ZFC also cannot prove that Z will run forever. In other words, the statement “Z will run forever” is independent of ZFC.

This is interesting because, while the undecidability of the halting problem tells us that there cannot exist an algorithmic method for determining whether an arbitrary Turing machine loops or halts, Z is an example of a specific Turing machine whose behavior cannot be proven one way or the other using the foundation of modern mathematics. Mathematicians and computer scientists think of themselves as being able to determine how a given algorithm will behave if given enough time to stare at it; despite this intuition, Z is a machine whose behavior we can never prove without assuming axioms more powerful than those generally assumed in modern mathematics.

1.2 Turing Machines

There are many slightly different definitions of Turing machines. For example, some definitions allow the machine to have multiple tapes; others only allow it to have one; some allow an arbitrarily large alphabet, while others allow only two symbols, and so on. In most research regarding Turing machines, mathematicians do not concern themselves with which of these models to use, because any one can simulate the others (usually efficiently). However, because this work is concerned with upper-bounding the exact number of states required to perform certain tasks, it is important to define the model precisely. The model we choose here is traditional for the busy beaver function.

Formally, a k-state Turing machine is a 7-tuple $M = (Q, \Gamma, a, \Sigma, \delta, q_0, F)$, where:

Q is the set of k *states* $\{q_0, q_1, \ldots, q_{k-2}, q_{k-1}\}$

$\Gamma = \{a, b\}$ is the set of *tape alphabet symbols*

a is the *blank symbol*

$\Sigma =$ is the set of *input symbols*

$\delta = Q \times \Gamma \to (Q \cup F) \times \Gamma \times \{L, R\}$ is the *transition function*

q_0 is the *start state*

$F = \{\text{HALT, ERROR}\}$ is the set of *halting states*.

A Turing machine's states make up the Turing machine's easily accessible, finite memory. The Turing machine's state is initialized to q_0.

The tape alphabet symbols correspond to the symbols that can be written on the Turing machine's infinite tape.

In this work, all Turing machines are run on the all-a input.

The transition function encodes the Turing machine's behavior. It takes two inputs: the current state of the Turing machine (an element of $Q \cup F$) and the symbol read off the tape (an element of Γ). It outputs three instructions: what state to enter (an element of $Q \cup F$), what symbol to write onto the tape (an element of Γ) and what direction to move the head (an element of $\{L, R\}$). A transition function specifies the entire behavior of the Turing machine in all cases.

The start state is the state that the Turing machine is in at initialization.

A halting transition is a transition to a halting state, which causes the Turing machine to halt. While having three possible halting transitions is not necessary for our purposes, being able to differentiate between different types of halting (HALT and ERROR) is useful for testing.

1.3 The Busy Beaver Function

Consider the set of all Turing machines with k states, for some positive integer k. We call a Turing machine B a *k-state busy beaver* if when run on the empty tape as input, B halts, and B also runs for at least as many steps before halting as all other halting k-state Turing machines [3].

In other words, a busy beaver is a Turing machine that runs for at least as long as all other halting Turing machines with the same number of states. Another common definition for a busy beaver is a Turing machine that writes as many ones on the tape as possible; because the number of ones written is a somewhat arbitrary measure, it is not used in this work.

The *busy beaver function*, written $BB(k)$, equals the number of steps it takes for a k-state busy beaver to halt. The busy beaver function has many striking properties. To begin with, it is not computable; in other words, there does not exist an algorithm that takes k as input and returns $BB(k)$, for arbitrary values of k. This follows directly from the undecidability of the halting problem. Suppose an algorithm existed to compute the busy beaver function; then given a k-state Turing machine M as input, we could compute $BB(k)$ and run M for $BB(k)$ steps. If, after $BB(k)$ steps, M had not yet halted, we could safely conclude that M would never halt. Thus, we could solve the halting problem, which we know is impossible.

By the same argument, $BB(k)$ must grow faster than any computable function. (To check this, assume that some computable function $f(k)$ grows faster than $BB(k)$ and substitute $f(k)$ for $BB(k)$ in the rest of the proof.) In particular, the busy beaver grows even faster than (for instance) the Ackermann function, a well-known fast-growing function.

Because finding the value of $BB(k)$ for a given k requires so much work (one must fully explore the behavior of all k-state Turing machines), few explicit values of the busy beaver function are known. The known values are [4, 5]:

$$\begin{aligned} BB(1) &= 1 \\ BB(2) &= 6 \\ BB(3) &= 21 \\ BB(4) &= 107. \end{aligned}$$

For $BB(5)$, $BB(6)$ and $BB(7)$, only lower bounds are known [6–8]:

$$\begin{aligned} BB(5) &\geq 47\,176\,870 \\ BB(6) &> 7.4\times 10^{36\,534} \\ BB(7) &> 10^{10^{10^{10^{10^{7}}}}}. \end{aligned}$$

Additionally, $BB(22)$ is known to be larger than Graham's number (a famous huge number from Ramsey theory, obtained by iterating the Ackermann function 64 times) [9]. Researchers have worked on pinning down the value of $BB(5)$ exactly, and some consider it to be possibly within reach.

Another way to discuss the busy beaver sequence is to say that modern mathematics has established a lower bound of 4 on the highest provable busy beaver value. In this paper, we prove the first known upper bound on the highest provable busy beaver value in ZFC; that is, we give a value of k, namely 7910, such that the value of $BB(k)$ cannot be proven in ZFC.

Intuitively, one might expect that while no algorithm may exist to compute $BB(k)$ for all values of k, we could find the value of $BB(k)$ for any specific k using a procedure similar to the one we used to find the value of $BB(k)$ for $k \leq 4$. The reason this is not so is closely tied to the existence of a machine like the Gödelian machine Z, as described in Section 1.1. Suppose that Z has k states. Because Z's behavior (whether it halts or loops) cannot be proven in ZFC, it follows that the value of $BB(k)$ also cannot be proven in ZFC; if it could, then a proof would exist of Z's behavior in ZFC. Such a proof would consist of a *computation history* for Z, which is an explicit step-by-step description of Z's behavior for a certain number of steps. If Z halts, then a computation history leading up to Z's halting would be the entire proof; if Z loops, then a computation history that takes $BB(k)$ steps, combined with a proof of the value of $BB(k)$, would constitute a proof that Z will run forever.

In this paper, we construct a machine like Z, for which a proof that Z runs forever would imply that ZFC was consistent. In doing so, we give an explicit upper bound on the highest busy beaver value provable in ZFC, assuming the consistency of a slightly stronger set theory. Our machine, which we shall refer to as Z hereafter, contains 7910 states. Therefore, we will never be able to prove the value of $BB(7910)$ without assuming more powerful axioms than those of ZFC. This upper bound is presumably very far from tight, but it is a first step.

Even to achieve a state count of 7910, we will need three nontrivial ideas: Friedman's order-theoretic statements, *on-tape processing* and *introspective encoding*. Without all three ideas, we found that the state count would be in the tens of thousands, hundreds of thousands or even millions. We briefly introduce these ideas in the following subsection and explore them in much greater detail in Section 8. The implementation of these ideas constitutes this paper's main technical contribution.

1.4 Parsimony

In most algorithmic study, efficiency is the primary concern. In designing Z, however, parsimony is the only thing that matters. One historical analog is the practice of "code golfing": a recreational pursuit adopted by some programmers in which the goal is to produce a piece of code in a given programming language, using as few characters as possible. Many examples of code golfing can be found at [10]. The goal of designing a Turing machine with as few states as possible to accomplish a certain task, without concern for the machine's efficiency or space usage, can be thought of as code golfing with a particularly low-level programming language.

Part of the charm of Turing machines is that they give us a "standard reference point" for measuring complexity, unencumbered by the details of more sophisticated programming languages. Also, with Turing machines, there can be no suspicion that we engineered a programming formalism just for the purpose of code golfing, or for making the concepts we want artificially simple to describe. This is why we prefer Turing machines as a tool for measuring complexity; not because they are particularly special, but simply because they are so primitive that their specifics will interfere minimally with what we mean by an algorithm being "complicated."

In this paper, we use three ideas for generating parsimonious Turing machines: Friedman's mathematical statements, on-tape processing and introspective Turing machines. The last of these ideas was proposed, under a different name and with some variations, by Ben-Amram and Petersen in 2002 [11]. These three ideas are explained in more detail in Subsections 3.1, 8.1 and 8.3, respectively, but we summarize them very briefly here.

The first idea is simply to use the research done by Friedman [12] into finding simple-to-express statements that are equivalent to the consistency of various axiomatic systems. In particular, we use a statement discovered by Friedman to be equivalent to the consistency of a set theory stronger than ZFC (and whose consistency, therefore, would imply the consistency of ZFC) [13]. (Admittedly, it is not obvious that using Friedman's current statements does decrease the state count of the Turing machines. It is possible that one could do as well or better by directly searching for contradictions in ZFC, and indeed, recent unpublished work by O'Rear has given some evidence for that [14]. On the other hand, Friedman's statements can be translated into code without using the apparatus of first-order logic, which arguably gives us a conceptual simplification. In addition, statements like Friedman's seem like the most plausible path forward for further reductions in the state count, beyond whatever lower limit one hits when one needs to encode the ZFC axioms explicitly.)

The second idea, on-tape processing, is a way to encode high-level commands into a Turing machine parsimoniously. Instead of converting commands to groups of states directly, which incurs a multiplicative overhead based on how large these groups need to be, on-tape processing begins by writing the commands onto the tape, using as efficient an encoding as possible. Then, once the commands are on the tape, the commands are processed by a single group of states that understands how to interpret them.

The third idea, introspective Turing machines, is a way to write long strings onto the tape using as few states as possible. The idea is to encode information in one of each state's transitions, instead of encoding information in each state's write field. This is advantageous because there are many choices for which state to point a transition to, but only two choices for which bit to write. Therefore, more information can be encoded in each state using this method.

1.5 Implementation Overview

To generate descriptions of Turing machines with nice mathematical properties entirely by hand is a daunting task. Rather than approach the problem directly, we created tools for generating parsimonious Turing machines while presenting an interface that is comfortably familiar to most programmers (and to us!).

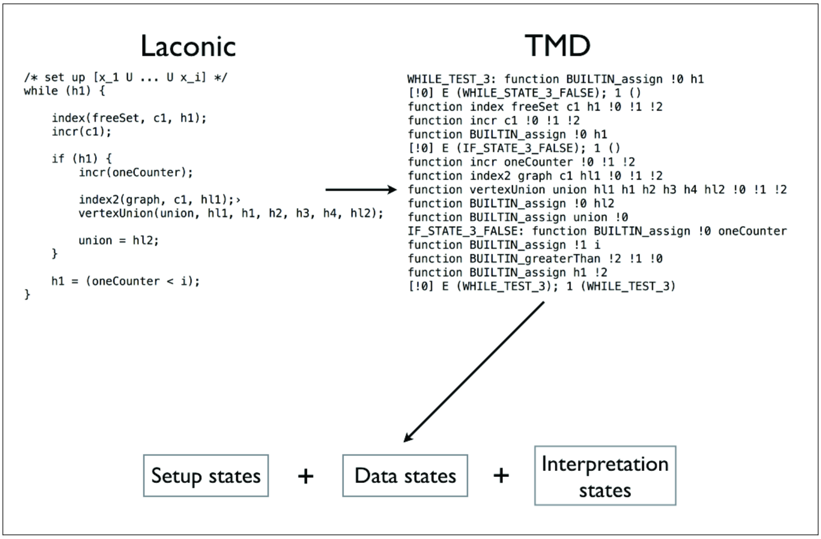
We created two tools. At the top level is the Laconic programming language, whose syntax and capabilities are similar to those of most programming languages, such as Java or Python. Beneath it we created a lower-level language called Turing Machine Descriptor (TMD). TMD is quite unlike most programming languages and is better thought of as a convenient way to describe a multi-tape, three-symbol Turing machine plus a function stack. The style of multi-tape Turing machine used in TMD is the commonly used "one-tape-at-a-time" abstraction: only one tape at a time can be interacted with, for reading, writing and moving the head. Laconic compiles down to a TMD program, and TMD compiles down to a description of a single-tape, two-symbol Turing machine. This process is illustrated in Figure 1.

We recommend that programmers hoping to use our tools to generate their own encodings of mathematical statements or algorithms as Turing machines use Laconic. Laconic's interface is perfect for somebody hoping to write in a "traditional" language. On the other hand, if the programmer wishes to improve upon Laconic's compilation process, writing code directly in TMD is likely to be the better option.

2. Related Work

Gregory Chaitin raised the problem of proving a version of our result in his book *The Limits of Mathematics* [15]. He wrote:

> I would like to have somebody program out Zermelo–Fraenkel set theory in my version of LISP, which is pretty close to normal LISP as far as this task is concerned, just to see how many bits



```
/* set up [x_1 U ... U x_i] */
while (h1) {

    index(freeSet, c1, h1);
    incr(c1);

    if (h1) {
        incr(oneCounter);

        index2(graph, c1, hl1);›
        vertexUnion(union, hl1, h1, h2, h3, h4, hl2);

        union = hl2;
    }

    h1 = (oneCounter < i);
}
```

```
WHILE_TEST_3: function BUILTIN_assign !0 h1
[!0] E (WHILE_STATE_3_FALSE); 1 ()
function index freeSet c1 h1 !0 !1 !2
function incr c1 !0 !1 !2
function BUILTIN_assign !0 h1
[!0] E (IF_STATE_3_FALSE); 1 ()
function incr oneCounter !0 !1 !2
function index2 graph c1 hl1 !0 !1 !2
function vertexUnion union hl1 h1 h2 h3 h4 hl2 !0 !1 !2
function BUILTIN_assign !0 hl2
function BUILTIN_assign union !0
IF_STATE_3_FALSE: function BUILTIN_assign !0 oneCounter
function BUILTIN_assign !1 i
function BUILTIN_greaterThan !2 !1 !0
function BUILTIN_assign h1 !2
[!0] E (WHILE_TEST_3); 1 (WHILE_TEST_3)
```

Figure 1. A visual overview of the compilation process.

> of complexity mathematicians normally assume … If you programmed ZF, you'd get a really sharp incompleteness result. It wouldn't say that you can get at most $H(ZF) + 15\,328$ bits of [Chaitin's halting probability] Ω, it would say, perhaps, at most 96 000 bits! We'd have a much more definite incompleteness theorem.

We did not program ZF set theory in LISP, but we programmed it in an even simpler language—thereby answering Chaitin's call for an explicit number of bits to attach to the complexity of ZF set theory. (As many as required to fully describe our Turing machine—or more precisely, 157 819.)

This paper is not the first to attempt to quantify the complexity of arithmetical statements. Calude and Calude [16, 17] define a register machine of their own design and provide quantifications of the complexity of Legendre's conjecture, Fermat's last theorem, Goldbach's conjecture, Dyson's conjecture, the Riemann hypothesis and the four color theorem. (Because Fermat's last theorem and the four color theorem have been proved, their "complexity" is now known to be 1—the minimum number of states in a Turing machine that runs forever.) In addition, Koza [18] and Pargellis [19] each invent instruction sets that are particularly well suited to representing self-reproducing programs simply and show that starting from a "primordial soup" of such instructions distributed about a large memory, along with an increasing number of program threads, a rich ecosystem of increasingly efficient self-reproducing programs starts to dominate the "landscape."

This paper differs from the previous work in two ways: first, it is the first to give explicit, relatively small machines whose behavior is provably independent of the standard axioms of modern mathematics. Second, to our knowledge, this paper is the first concrete study of parsimony to use Turing machines themselves as the model of computation—rather than (for example) a new programming language proposed by the authors, or a complex on-tape description of Turing machines! We consider it important to use the weakest and most common model of computation for complexity comparisons across different mathematical statements. This is because the more powerful and complex the model of computation used, the more of the complexity of the algorithm can be “shunted” onto the model of computation, and the greater the potential distortion created by the choice of model. As a reductio ad absurdum, we could imagine a programming language that included “test the Riemann hypothesis” and “test the consistency of ZFC” as primitive operations. By using the “weakest” model of computation that is commonly known, we hope to avoid this pitfall and make it easier to interpret our results in a model-independent way.

Also related to the work of this paper is the famous search for the smallest universal Turing machine, which has a relatively long history. A survey is available at [20]. Here a *universal Turing machine* is a Turing machine that can simulate any other Turing machine, when a description of the latter is provided on its input tape. The smallest known universal Turing machine has only two states and a three-symbol alphabet. It was found and conjectured to be universal by Wolfram [21] and then proved to be universal by Smith [22] in 2007. The search for the smallest universal Turing machine is closely related to the smallest Turing machine that is independent of ZFC, in that both constitute a search for simplicity according to some rigorous metric. From the perspective of this paper, however, the problem is that the known small universal Turing machines achieve their small size only at the cost of an extremely complicated description format for the input machine. That is, most of the complexity gets “shunted” from the Turing machine itself to the input encoding format. By contrast, with small Turing machines to test Con(ZFC), such as the Riemann hypothesis, Goldbach’s conjecture, or others, and which run on an initially blank tape, there is no analogous trick for hiding the statement’s complexity.

Finally, let us mention that, after we circulated a preprint of this work, O’Rear [23] created a different 1919-state Turing machine whose behavior is equivalent to the consistency of ZFC. O’Rear was directly inspired to do this by our result; his result, however, is stronger than ours in two ways. First, his machine is substantially smaller than ours, yielding a tighter upper bound on the lowest busy beaver number whose value is independent of ZFC. Second, the non-halting of his machine is directly equivalent to Con(ZFC), whereas proving the non-halting of our machine seems to require assuming the consistency of a stronger system than ZFC (known as “stationary Ramsey property”).

In order to create his machine, O’Rear adapted our Laconic language, creating a slightly different language which he called Not Quite Laconic (NQL). O’Rear then wrote a short NQL program that directly iterates through all theorems of a formal system called Metamath, which is known to have the same consistency strength as ZFC.

We hope that future work will manage to tighten the upper bound still further.

3. A Turing Machine That Cannot Be Shown to Run Forever Using ZFC

We present a 7910-state Turing machine whose behavior is independent of ZFC; it is not possible to prove that this machine halts or does not halt using the axioms of ZFC, assuming that a stronger set theory is consistent. It is therefore impossible to prove the value of $BB(7910)$ to be any given value without assuming axioms more powerful than ZFC, assuming that ZFC is consistent.

For an explicit listing of this machine, see Appendix C.

We call this machine Z. One way to build this machine would be to start with the axioms of ZFC and apply the inference rules of first-order logic repeatedly in each possible way so as to enumerate every statement ZFC could prove, and to halt if ever a contradiction was found. While this method is conceptually simple, to actually construct such a machine would lead to a huge number of states, because it would require writing a program to manipulate the axioms of ZFC and the inference rules of first-order logic and then compiling that program all the way down to Turing machine states.

3.1 Friedman’s Mathematical Statement

Thankfully, a simpler method exists for creating Z. Friedman [13] was able to derive a graph theoretic statement whose truth implies the consistency of ZFC and that is false if ZFC is inconsistent. Moreover, like most such conditional statements about the consistency of formal systems, Friedman’s theorem could itself be formalized and proved in a fragment of Peano arithmetic, so we can talk about it in the same theory-independent terms with which we talk about (say) the prime number theorem or any other result in elementary number theory. In fact, Friedman’s statement is equivalent to the consistency of SRP ("stationary Ramsey property"), which is a system of axioms more powerful than ZFC. Because SRP is strictly more powerful than ZFC (it in fact consists of ZFC plus some additional axioms), the consistency of SRP implies the consistency of ZFC, and the inconsistency of ZFC implies the inconsistency of SRP. Here is Friedman's statement (the notation will be explained in the rest of this section):

Statement 1. For all $k, n, r > 0$, every order invariant graph on $[\mathbb{Q}]^{\leq k}$ has a free $\{x_1, \ldots, x_r, \text{ush}(x_1), \ldots, \text{ush}(x_r)\}$ of complexity $\leq (8knr)!$, each $\{x_1, \ldots, x_{(8kni)!}\}$, for $i > 0$ and $(8kni!) \leq r$, reducing $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$ [13].

If s is a set, the operation $(.)^{\leq k}$ refers to the set of all subsets of s with size at most k.

A graph on $[\mathbb{Q}]^{\leq k}$ is an irreflexive symmetric relation on $[\mathbb{Q}]^{\leq k}$. In other words, it can be thought of as a graph whose vertices are elements of $[\mathbb{Q}]^{\leq k}$, and whose edges are undirected, connected pairs of vertices. These edges never connect vertices to themselves.

A *free* set is a set such that no pair of elements in that set are connected by an edge.

A number of *complexity* at most c refers to a number that can be written as a fraction a / b, where a and b are both integers with absolute value less than or equal to c. A set has complexity at most c if all the numbers it contains have complexity at most c.

An *order invariant graph* is a graph containing a countably infinite number of nodes. In particular, it has one node for each finite set of rational numbers. The only numbers relevant to the statement are numbers of complexity $(8knr)!$ or smaller. In every description of nodes that follows, the term *node* refers both to the object in the order invariant graph and to the set of numbers that it represents.

In an order invariant graph, two nodes (a, b) have an edge between them if and only if each other pair of nodes (c, d) that is order equivalent with (a, b) has an edge between them. Two pairs of nodes (a, b) and (c, d) are *order equivalent* if a and c are the same size and b and d are the same size and if for all $1 \leq i \leq |a|$ and $1 \leq j \leq |b|$, the i^{th} element of a is less than the j^{th} element of b if and only if the i^{th} element of c is less than the j^{th} element of d.

To give some trivial examples of order invariant graphs: the graph with no edges is order invariant, as is the complete graph. A less trivial example is a graph on $[\mathbb{Q}]^{\leq 2}$, in which each node corresponds to a set of two rational numbers of a given complexity, and there is an edge between two nodes if and only if their corresponding sets a and

b satisfy $|a| = |b| = 2$ and $a_1 < b_1 < a_2 < b_2$. (Because edges are undirected in order invariant graphs, such an edge will exist if either assignment of the vertices to a and b satisfies the inequality above.)

The ush() function takes as input a set and returns a copy of that set with all non-negative numbers in that set incremented by 1.

For vertices x and y, $x \leq_{\text{rlex}} y$ if and only if $x = y$ or $x_{|x|-i} < y_{|y|-i}$, where i is the least integer such that $x_{|x|-i} \neq y_{|y|-i}$. (Friedman recommended in private communication that we use the $\leq_{\text{rlex}}$ comparator to compare vertices, instead of comparing their maximum elements as described in his manuscript.) (The $\leq_{\text{rlex}}$ operation creates a lexicographic ordering over the vertices, weighting the last and largest elements of those vertices most heavily. Like with lexicographic orderings, if the two vertices are identical but one is longer, the shorter one comes first.)

Finally, a set of vertices X *reduces* a set of vertices Y if and only if for all $y \in Y$, there exists $x \in X$ such that either $x = y$ or $x \leq_{\text{rlex}} y$ and an edge exists between x and y.

3.2 Implementation Methods

To create Z, we needed to design a Turing machine that halts if Statement 1 is false and loops if Statement 1 is true. Such a Turing machine's behavior is necessarily independent of ZFC, because the truth or falsehood of Statement 1 is independent of ZFC, assuming the consistency of SRP [13]. SRP is an extension of ZFC by certain relatively mild, large cardinal hypotheses and is widely regarded by set theorists as consistent. For more information about SRP, see [24].

To design such a Turing machine, we wrote a Laconic program that encodes Friedman's statement, then compiled the program down to a description of a single-tape, two-symbol Turing machine. What follows is an extremely brief description of the design of the Laconic program; for the documented Laconic code itself, along with a detailed explanation of the full compilation process, see [25].

Our Laconic program begins by looping over all non-negative values for k, n and r. For each trio (k, n, r), our program generates a list N of all numbers of complexity at most $(8knr)!$. These numbers represent the vertices in our putative order invariant graph. Because Laconic does not support floating-point numbers, the list is entirely composed of integers; it is a list of all numbers that can be written in the form $(((8knr)!)!)(i/j)$, where i and j are integers satisfying $-(8knr)! \leq i \leq (8knr)!$ and $1 \leq j \leq (8knr)!$. (Note that any number that can be expressed in this form is necessarily an integer, because of the large scaling factor in front.)

After we generate N, we generate the nodes in a potential order invariant graph by adding to N all possible lists of k or fewer numbers from N. We call this list of lists V.

We iterate over all binary lists of length $|V|^2$. Any such list E represents a possible set of edges in the graph. To be more precise, we say that an edge exists between node i and node j (represented by V_i and V_j respectively) if and only if $E_{i|V|+j}$ is 1.

For any graph (V, E), we say that it is "valid" if the following three conditions hold:

1. No node has an edge to itself.
2. If an edge exists between node i and node j, an edge also exists between node j and node i.
3. The graph has a free $\{x_1, \ldots, x_r, \text{ush}(x_1), \ldots, \text{ush}(x_r)\}$, each $\{x_1, \ldots, x_{(8kni)!}\}$ reducing $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$.

For each list of nodes V, we loop over every possible binary list E, and if no pair (V, E) yields a valid graph, we halt.

When verifying the validity of a graph, checking the first two conditions is trivial, but the third merits further explanation. In order to verify that a given graph (V, E) has a free $\{x_1, \ldots, x_r, \text{ush}(x_1), \ldots, \text{ush}(x_r)\}$, each $\{x_1, \ldots, x_{(8kni)!}\}$ reducing $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$, we look at every possible subset of the nodes in V. For each subset, we verify that it has length r, that $\text{ush}(x_1), \ldots, \text{ush}(x_r)$ all exist in V, and for each i such that $(8kni)! \leq r$, that $\{x_1, \ldots, x_{(8kni)!}\}$ reduces $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$. Once we have found such a subset, we know that the third condition is satisfied.

4. A Turing Machine That Encodes Goldbach's Conjecture

We present a 4888-state Turing machine that encodes Goldbach's conjecture; in other words, to know whether this machine halts is to know whether Goldbach's conjecture is true. It is therefore impossible to prove the value of $BB(4888)$ without simultaneously proving or disproving Goldbach's conjecture. (Note that our tools were primarily meant to encode complex statements into Turing machines, such as Statement 1. Because Goldbach's conjecture is so simple, it is feasible in that case to make dramatically smaller Turing machines through a more direct approach. Indeed, after a preprint of this paper was circulated online, "Jared S" and "code golf addict" created Turing machines for Goldbach's conjecture with 47 and 31 states, respectively [14].)

Recall that Goldbach's conjecture is as follows:

Statement 2. Every even integer greater than 2 can be expressed as the sum of two primes.

Because Goldbach's conjecture is so simple to state, the Laconic program encoding the statement is also quite simple. It can be found in Appendix A. A detailed explanation of the compilation process, documentation for the Laconic language and an explicit description of this Turing machine are available at [25].

5. A Turing Machine That Encodes the Riemann Hypothesis

We present a 5372-state Turing machine that encodes the Riemann hypothesis; in other words, to know whether this machine halts is to know whether the Riemann hypothesis is true. An explicit description of this machine can be found at [25].

The Riemann hypothesis is traditionally stated as follows:

Statement 3. The Riemann zeta function has its zeros only at the negative even integers and the complex numbers with real part $1/2$.

5.1 Equivalent Statement

Instead of encoding the Riemann zeta function into a Laconic program, it is simpler to use the following statement, which was shown by Davis, Matijasevic and Robinson [26] to be equivalent to the Riemann hypothesis:

Statement 4. For all integers $n \geq 1$,

$$\left(\left(\sum_{k \leq \delta(n)} \frac{1}{k}\right) - \frac{n^2}{2}\right)^2 < 36n^3.$$

The function $\delta(n)$ used in Statement 4 is defined as follows:

$$\eta\,(j) = p \text{ if } j = p^k,\ p \text{ is prime, } k \text{ is a positive integer}$$
$$\eta\,(j) = 1 \text{ otherwise}$$
$$\delta\,(x) = \prod_{n<x} \prod_{j \leq n} \eta(j).$$

5.2 Implementation Methods

Statement 4 is equivalent to the following statement, which involves only positive integers:

$$l(n) < r(n)$$

for all positive integers n, where

$$\begin{aligned}
l(n) &= a(n)^2 + b(n)^2 \\
r(n) &= 36n^3(\delta(n)!)^2 + 2a(n)b(n) \\
a(n) &= \sum_{k\leq\delta(n)} \frac{\delta(n)!}{k} \\
b(n) &= \frac{n^2\delta(n)!}{2}.
\end{aligned}$$

Although it is not immediately obvious, $\delta(n)! \big/ k$ is necessarily an integer for all $k \leq \delta(n)$, and $\delta(n)! \big/ 2$ is an integer for all $n > 1$.

To check the Riemann hypothesis, our program computes $a(n)$, $b(n)$, $l(n)$ and $r(n)$, in that order, for each possible value of n. If $l(n) \geq r(n)$, our program halts.

6. Laconic

Laconic is a programming language designed to be both user friendly and easy to compile down to parsimonious Turing machine descriptions.

Laconic is a strongly typed language that supports recursive functions. Laconic compiles to an intermediate language called TMD. TMD programs are spread across multiple files and grouped into directories. TMD directories are meant to represent sequences of commands that could be given to a multi-tape, three-symbol Turing machine, using the Turing machine abstraction that allows the machine to read and write from one head at a time.

For an example of a Laconic program, see Appendix A. For an illustration of the compilation process, see Figure 1.

7. Turing Machine Descriptor

TMD is a programming language designed to help the user describe the behavior of a multi-tape, three-symbol Turing machine with a function stack. Each tape is infinite in one direction and supports three symbols: `_`, `1` and `E`. The blank symbol is `_`; that is, `_` is the only symbol that can appear on the tape an infinite number of times. The tape must always have the form $\_?(1\,|E)^+\_^\infty$; in other words, each tape must always contain a string of `1`s and `E`s of size at least 1, possibly preceded by a `_` symbol and necessarily followed by an infinite number of copies of the `_` symbol.

What is the purpose of having a language like TMD as an intermediary between Laconic and a description of a single-tape machine? The concept of tapes in a multi-tape Turing machine and the concept of variables in standard imperative programming languages map to one another very nicely. The idea of the Laconic-to-TMD compiler is to encode the value of each variable on one tape. Then, each Laconic command that manipulates the value of one or more variables compiles down to a TMD function call that manipulates the tapes that correspond to those variables appropriately.

As an example, consider the following Laconic command:

```
a == b*c;
```

This Laconic command assigns the value of `b*c` to the variable `a`. It compiles down to the following TMD function call:

```
functionBUILTIN _ multiply a b c
```

This function call will result in `BUILTIN_multiply` being run on the three tapes `a`, `b` and `c`. This will cause the symbols on tape `a` to take on a representation of an integer whose value is equal to *bc*.

In turn, the TMD code compiles directly to a string of bits that is written onto the tape at the start of the Turing machine's execution.

A TMD directory consists of three types of files:

1. The `functions` file. This file contains a list of the names of all the functions used by the TMD program. The top function in the file is pushed onto the stack at initialization. When this top function returns, the Turing machine halts.
2. The `initvar` file. This file contains the non-blank symbols that start in each register (or tape) at initialization.
3. Any files used to describe TMD functions. These files all end in a .tfn extension and only have any relevance to the compiled program if they show up in the functions file.

8. Compilation and Processing

There are two ways to think about the layout of the tape symbols: with a four-symbol alphabet ({_, 1, H, E}, blank symbol _) and with a two-symbol alphabet ({a, b}, blank symbol a). The two-symbol alphabet version is the one that is ultimately used for the results in this paper, since we advertised a Turing machine that used only two symbols. However, in nearly all parts of the Turing machine, the two-symbol version of the machine is a direct translation of the four-symbol version, according to the following mapping:

_ ↔ aa
1 ↔ ab
H ↔ ba
E ↔ bb

The sections that follow sometimes refer to the ERROR state. Transitions to the ERROR state should never be taken under any circumstances and are useful for debugging purposes.

8.1 Concept

A directory of TMD functions is converted at compilation time to a string of bits to be written onto the tape, along with other states designed to interpret these bits. The resulting Turing machine has three main components, or *submachines*:

1. The *initializer* sets up the basic structure of the variable registers and the function stack.
2. The *printer* writes down the binary string that corresponds to the compiled TMD code.
3. The *processor* interprets the compiled binary, modifying the variable registers and the function stack as necessary.

The Turing machine's control flow proceeds from the initializer to the printer to the interpreter. In other words, initializer states point only to initializer states or to printer states, printer states point only to printer states or to interpreter states and interpreter states point only to interpreter states or the HALT state.

This division of labor, while seemingly straightforward, actually constitutes an important idea. The problem of the compiler is to convert a higher-level representation—a machine with many tapes, a larger alphabet and a function stack—to the lower-level representation of a machine with a single tape, a two-symbol alphabet and no function stack. The immediately obvious solution, and the one taught in every computability theory class as a proof of the equivalence of different kinds of Turing machines, is to have every "state" in the higher-level machine compile down to many states in the lower-level machine.

While simple, this approach is suboptimal in terms of the number of states. As is nearly always true when designing systems to be parsimonious, the clue that improvement is possible lies in the presence of repetition. Each state transition in the higher-level machine is converted to a group of lower-level states with the same basic structure. Why not instead explain how to perform this conversion exactly once, and then apply the conversion many times?

This idea is at the core of the division of labor described previously. We begin by writing a description of the higher-level machine onto the tape and then "run" the higher-level machine by reading what is on the tape with a set of states that understands how to interpret the encoded higher-level machine. We refer to this idea as *on-tape processing*.

In this paper, we use TMD as the representation of the higher-level machine. (Note that instead of TMD, the on-tape processing scheme could be used for any language, assuming the designer provides both a processor and an encoding for that language. We chose TMD because it made the interpreter easy to write, but other minimalist languages, like Unlambda [27], BF [28] or Iota and Jot [29], might be good candidates for parsimonious designs, with the additional advantage of being already known to some programmers! Thanks to Luke Schaeffer for this point.) The printer writes the TMD program onto the tape, and the processor executes it. As a result of using this scheme, we incur a constant additive overhead—we have to include the processor in our final Turing machine—but we avoid the constant multiplicative overhead required for the naïve scheme.

Is this additive overhead small enough to be worth it? We found that it is. Our implementation of the processor requires 3860 states. (See Section 8.5 for a detailed breakdown of the state cost by submachine.) In contrast to this additive overhead of 3860, the naïve approach incurs a large multiplicative overhead that depends in part on how many states must be used to represent each higher-level state transition, and in part on how efficient an encoding scheme can be devised for the on-tape approach. Table 1 compares the performance of on-tape processing to the performance of an implementation that used the naïve approach. The comparison is shown for three kinds of machines: a machine that halts if and only if Goldbach's conjecture is false, a machine that halts if and only if the Riemann hypothesis is false and a machine whose behavior is independent of ZFC.

As can be seen from Table 1, on-tape interpretation results in huge gains, particularly in large and complex programs.

The subsections that follow describe each of the three submachines—the initializer, the printer and the processor—in greater detail.

Program	States (Naïve)	States (On-Tape Processing)
Goldbach	7902	4888
Riemann	36146	5372
ZFC	340943	7910

Table 1. A comparison of Turing machine size with and without on-tape processing. On-tape processing leads to vastly more parsimonious Turing machines.

8.2 The Initializer

The initializer starts by writing a counter onto the tape that encodes how many registers there will be in the program. Using the value in that counter, it creates each register, with demarcation patterns between registers and unique identifiers for each register. Each register's value begins with the pattern of non-blank symbols laid out in the `initvar` file. The initializer also creates the program counter, which starts at 0, and the function stack, which starts out with only a single function call to the top function in the `functions` file.

Figure 2 is a detailed diagram describing the tape's state when the initializer passes control to the printer.

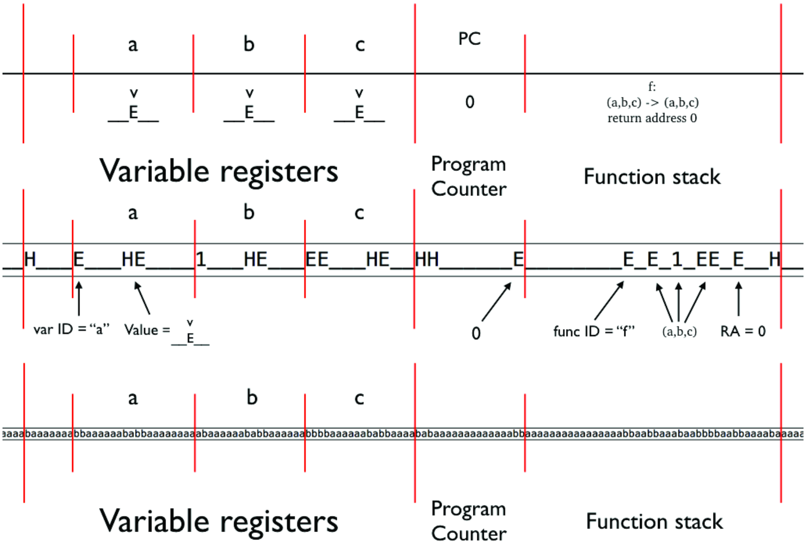


Figure 2. The state of the Turing machine tape after the initializer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of what each part of the Turing machine tape represents. The middle bar is an encoding of the tape in the standard four-symbol alphabet; the bottom bar is simply the translation of that tape into the two-symbol alphabet. For a more detailed explanation of how to interpret the tape patterns, see [25].

8.3 The Printer

8.3.1 Specification

The printer writes down a long binary string that encodes the entirety of the TMD program onto the tape.

Figure 3 shows the tape's state when the printer passes control to the processor.

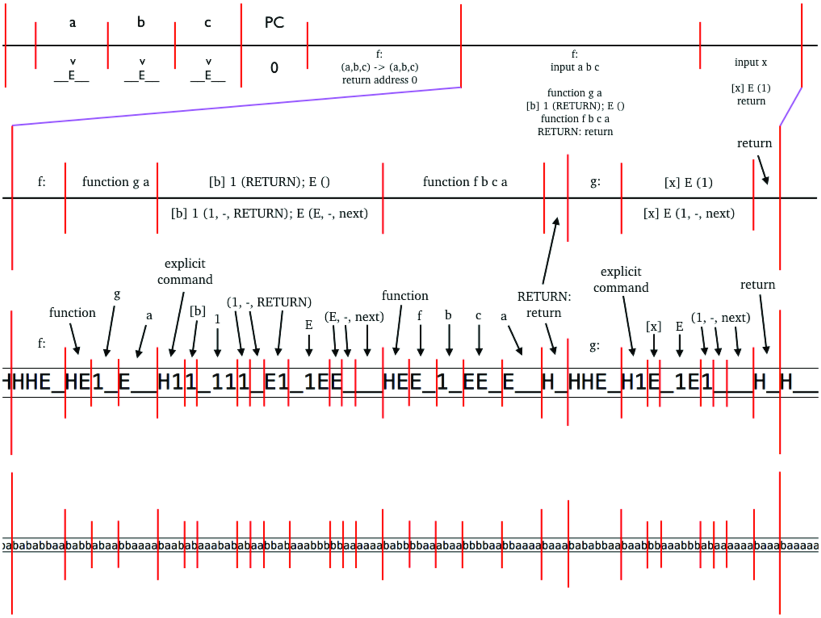


Figure 3. The state of the Turing machine tape after the printer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of the entire tape; unfortunately, at this point there are so many symbols on the tape that it is impossible to see everything at once. For a detailed view of the first two-thirds of the tape (registers, program counter and stack), see Figure 2. The bottom three bars show a zoomed-in view of the program binary. From the top, the second bar gives a high-level description of what each part of the program binary means; the third bar gives the direct correspondence between four-symbol alphabet symbols on the tape and their meaning in TMD; the fourth and final bar gives the translation of the third bar into the two-symbol alphabet. For a more detailed explanation of the encoding of TMD into tape symbols, see [25].

8.3.2 Introspection

Writing down a long binary string onto a Turing machine tape in a parsimonious fashion is not as straightforward as it might initially appear. The first idea that comes to mind is simply to use one state per symbol, with each state pointing to the next, as shown in Figure 4.

On closer examination, however, this approach is quite wasteful for all but the smallest binary files. Every a transition points to the next state in the sequence, and none of the b transitions are used at all! Indeed, the only information-bearing part of the state is the single bit contained in the choice of which symbol to write. But in theory, far more information than that could be encoded in each state. In a

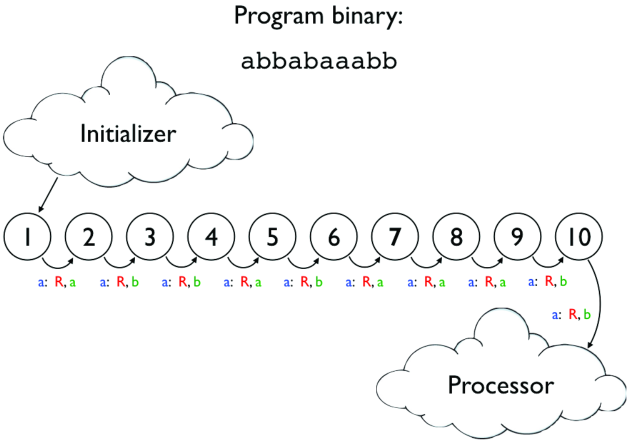


Figure 4. A naïve implementation of the printer. In this example, the hypothetical program is 10 bits long, and the printer uses 10 states, one for each bit. In the diagram, the blue symbol is the symbol that is read on a transition, the red letter indicates the direction the head moves and the green symbol indicates the symbol that is written. Note the lack of transitions on reading a b; this is because in this implementation, the printer will only ever read the blank symbol, which is a, since the head is always proceeding to untouched parts of the tape. It therefore makes no difference what behavior the Turing machine adopts upon reading a b in states 1 through 10 (and therefore b transitions are presumed to lead to the ERROR state)

machine with n states, each state could contain $2\big(\log_2(n)+1\big)$ bits of information, because each of the state's two transitions could point to any of the n states, and the machine will write either an a or a b onto the tape. Of course, this is only in theory; in practice, to extract the information contained in the Turing machine's states and translate it into bits on the tape is nontrivial.

We will use a scheme originally conceived by Ben-Amram and Petersen [11] and refined further and suggested to us by Luke Schaeffer. It does not achieve the optimal theoretical encoding described above, but it is relatively simple to implement and understand and is within a factor of 2 of optimal for large binary strings. Schaeffer named Turing machines that use this idea *introspective*.

Introspection works as follows. If the binary string contains k bits, then let w be the *word size*. The word size w takes the largest value it can such that $w2^w \le k$. We can split the binary string into $n_w = \lceil k / w \rceil$ *words* of w bits each (we can pad the last word with the blank symbol). In our scheme, each word in the bit string is represented by a *data state*. Each data state points to the state representing the next word in the sequence for its a transition, but which state the b transition points to encodes the next word. Every b transition points to one of the last 2^w data states, thereby encoding w bits of information.

Of course, the encoding is useless until we specify how to extract the encoded bit string from the data states. The extraction scheme works as follows. To query the i^{th} data state for the bits it encodes, we run the data states on the string $\mathtt{a}^{i-1}\mathtt{ba}^{\infty}$ (a string of $i-1$ as followed by a b in the i^{th} position). After running the data states on that string, what remains on the tape is the string $\mathtt{b}^{i-1}\mathtt{ab}^{r}\mathtt{a}^{\infty}$, assuming that the i^{th} data state pointed to the r^{th}-to-last data state. Thus, what we are left with is essentially a unary encoding of the "value" of the word in binary. Thus, the job of the extractor is to set up a binary counter that removes one b at a time and increments the counter appropriately. Then afterward, the extractor reverts the tape back to the form $\mathtt{a}^{i}\mathtt{ba}^{\infty}$, shifts all symbols on the tape over by w bits and repeats the process. Finally, when the state beyond the last data state sees a b on the tape, we know that the process has completed, and we can pass control to the processor. Figure 5 shows the whole procedure.

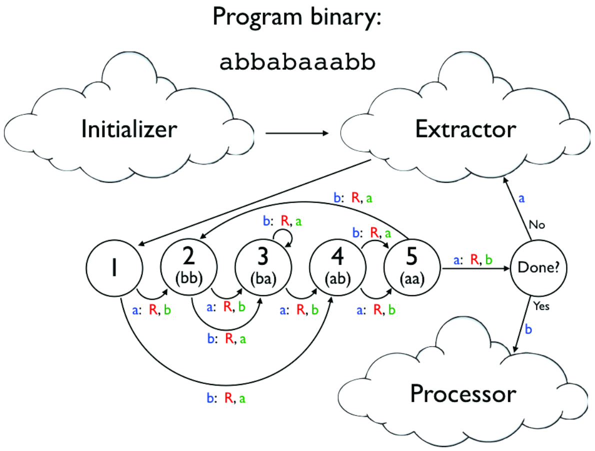


Figure 5. An introspective implementation of the printer. In this example, the hypothetical program is $k = 10$ bits long, and so the word size must be 2 (since $w = 2$ is the largest w such that $w 2^{w} \leq 10$). There are therefore $n_w = \lceil k / w \rceil = 5$ data states, each encoding two bits. The b transitions carry the information about the encoding; note that each one only points to one of the last four data states. The last four data states have in parentheses what word we mean to encode if we point to them.

How much have we gained by using introspection for encoding the program binary, instead of the naïve approach? It depends on how large the program binary is. Using introspection incurs an $O(\log k)$ additive overhead, because we have to include the extractor in our machine. (Our implementation of the extractor takes $10w + 17$ states. It is possible to build a constant-size extractor, but it is not worth it for our value of w.) In return, we save a multiplicative factor of w (which scales with $\log k$) on the number of data states needed.

This is plainly not worth it for the 10-bit example binary shown in Figures 4 and 5. For that binary, we require 69 additional states for the extractor in order to save five data states. For real programs, however, it is worth it, as can be seen from Table 2.

Program	Binary Size	w	n_w	Extractor Size	States (Naïve)	States (Introspective)
Example TMD	116	4	29	57	116	86
Goldbach	4964	9	552	107	4964	659
Riemann	9532	10	1024	117	9532	1141
ZFC	38864	11	3534	127	38864	3661

Table 2. Statistics relating to the printer, with and without using introspective techniques. Introspection leads to substantially more parsimonious Turing machines, particularly when the Turing machine is complex and the program binary is long.

One minor detail concerns the numbers presented for the Riemann program. Ordinarily, with a binary of size 9532, we would opt to split the program into 1060 words of nine bits each plus a 107-state extractor, since nine is the greatest w such that $w2^w < 9532$. But because 9532 is so close to the "magic number" 10 240, it is actually more parsimonious to pad the program with copies of the blank symbol until it is 10 240 bits long and split it into 1024 words of 10 bits each plus a 117-state extractor.

8.4 The Processor

The processor's job is to interpret the code written onto the tape and modify the variable registers and function stack accordingly. The processor does this by the following sequence of steps:

START:

1. Find the function call at the top of the stack. Mark the function f in the code whose ID matches that of the top function call.
2. Read the current program counter. Mark the line of code l in f whose line number matches the program counter.
3. Read l. Depending on what type of command l is, carry out one of the following three lists of tasks.

IF l IS AN EXPLICIT TAPE COMMAND:

1. Read the variable name off l. Index the variable name into the list of variables in the top function on the stack. This list of variables corresponds to the mapping between the function's local variables and the register names.
2. Match the indexed variable to its corresponding register r. Mark r. Read the symbol s_r to the right of the head marker in that register.
3. Travel back to l, remembering the value of s_r using states. Find and mark the reaction x corresponding to the symbol. See what symbol s_w should be written in response to reading s_r.
4. Travel back to r, remembering the value of s_w using states. Replace s_r with s_w.
5. Travel back to x. See which direction d the head should move in response to reading s_r.
6. Travel back to r, remembering the value of d using states. Move the head marker accordingly.
7. Travel back to x. See if a jump is specified. If a jump is specified, copy the jump address onto the program counter. Otherwise, increment the program counter by 1.
8. Go back to START.

IF l IS A FUNCTION CALL:

1. Write the function's name to the top of the stack.
2. For each variable in the function call, index the variable name into the list of variables in the top function on the stack. This list of variables corresponds to the mapping between the function's local variables and the register names. Push the corresponding register names in the order that they correspond to the variables in the function call.
3. Copy the current program counter to the return address of the newborn function call at the top of the stack.
4. Replace the current program counter with 0 (meaning "read the first line of code").
5. Go back to START.

IF l IS A RETURN STATEMENT:

1. Replace the current program counter with f's return address.
2. Increment the program counter by 1.
3. Erase the call to f from the top of the stack.
4. Check if the stack is now empty. If so, halt.
5. Go back to START.

8.5 Cost Analysis

It is worthwhile to analyze the relative contributions of the initializer, the printer and the processor to the machine's final state count. Table 3 lists the number of states in each submachine for each of the four different TMD programs under discussion.

Program	Initializer	Printer	Processor	Total
Example TMD	349	86	3860	4295
Goldbach	369	659	3860	4888
Riemann	371	1141	3860	5372
ZFC	389	3661	3860	7910

Table 3. State cost of each submachine. The cost of the processor is substantial but fixed; as the Turing machine becomes more complicated, the cost of the printer becomes increasingly important.

As can be seen from Table 3, the processor makes the largest contribution to all four programs. Improving the processor, therefore, is probably the best approach for improving upon the bounds we present. Equally clear, however, is that for programs more complicated than the ones presented here, the cost of the printer will grow almost linearly, but the cost of the processor will stay the same. The cost of the initializer grows very slightly with the complexity of programs because of the need to initialize additional registers.

Improving the printer, and with it the TMD and Laconic languages, is probably the best approach for reducing state count for very large and complex programs.

9. Future Work

This paper still leaves a three-orders-of-magnitude gap between the smallest n, namely 7910, for which $BB(n)$ is known to be independent of ZF set theory, and the largest n, namely 4, for which $BB(n)$ is known to be determinable. We regard it as a fascinating problem to pin down the truth here: for example, is it conceivable that $BB(10)$ or even $BB(6)$ might be independent of ZF? If so, that would arguably force a qualitative change in our understanding of the Gödel incompleteness phenomenon—showing that incompleteness from strong set theories rears its head for much simpler arithmetical questions than had previously been known.

A more immediate question is how much further Z's state count can be reduced. We are optimistic about the possibility of further reductions. For example, one could adapt the processor-printer model to use a better language than TMD. Ideally, one wants a language whose processor contains fewer states than TMD's, and whose typical programs are also shorter than TMD programs. A few ideas have been proposed for this [14], many of them related in some way to lambda calculus.

Other future work might involve further use of our Laconic language to upper-bound the "complexities" of mathematical statements and algorithms, in as standardized and model-independent a way as possible. Perhaps Laconic could be used to measure the complexity of other well-known conjectures, or even to compare different algorithms for solving the same problem (e.g., to try to quantify the notion that an insertion sort is simpler than a merge sort)!

Acknowledgments

We thank Harvey Friedman for having done the crucial theoretical work that made this project feasible. He was endlessly available over email and provided us with detailed clarifications when we needed them.

We thank Luke Schaeffer for his early help, as well as his help designing introspective Turing machines.

We thank Alex Arkhipov for introducing us to the term "code golfing."

We thank the commenters on Scott Aaronson's blog [14] for their ideas and suggestions.

Supported by an Alan T. Waterman Award from the National Science Foundation, under grant number 1249349.

Appendices

A. Example Laconic Program: Goldbach's Conjecture

The following is an example Laconic program, which compiles down to the Turing machine *G* mentioned in Section 4 (which halts if and only if Goldbach's conjecture is false).

```
func zero(x) {
    x = 0;
    return;
}

func one(x) {
    x = 1;
    return;
}
```

```
func incr(x) {
    x = x + 1;
    return;
}

/* Computes x modulo y */
func modulus(x, y, out) {
    out = x;

    while (out >= y) {
        out = out - y;
    }

    return;
}

func assignXtoYminusX(x, y) {
    x = y - x;
    return;
}

/* Figures out if x is prime, and puts the output in y */
/* Does not modify x, modifies y */
func isPrime(x, h, y) {
    if (x == 1) {
        zero(y);
        return;
    }

    y = 2;

    while (x > y) {
        modulus(x, y, h);

        if (h == 0) {
            zero(y);
            return;
        }
        incr(y);
    }

    return;
}

int evenNumber;
int primeCounter;
int isThisOnePrime;
int foundSum;
int h;

evenNumber = 2;
one(foundSum);
```

```
while (foundSum) {
    zero(foundSum);
    evenNumber = evenNumber + 2;
    one(primeCounter);

    while (primeCounter < evenNumber) {
        isPrime(primeCounter, h, isThisOnePrime);

        if (isThisOnePrime) {
            assignXtoYminusX(primeCounter, evenNumber);
            isPrime(primeCounter, h, isThisOnePrime);
            assignXtoYminusX(primeCounter, evenNumber);

            if (isThisOnePrime) {
                print evenNumber;
                print primeCounter;

                one(foundSum);
            }
        }

        incr(primeCounter);
    }
}

halt;
```

For detailed documentation of the Laconic programming language, see [25]. To find this file specifically, navigate to parsimony/src/laconic/laconic_files/goldbach.lac at [25].

B. Example Turing Machine Descriptor Program

The following is an example TMD directory, which compiles down to a binary string to be written on a Turing machine tape. It is the example used in illustrations throughout this paper, most notably in the example compilation shown in Figures 2 and 3. The program calls itself recursively three times until the starting symbol on each tape E is replaced with a 1, at which point the program halts.

This TMD directory is called example_tmd_dir and contains four files: f.tmd, g.tmd, initvar and functions.

```
f.tmd:
input a b c

// Recursively writes a 1 on every tape.

function g a
[b] 1 (RETURN); E ()
function f b c a
```

```
RETURN: return

g.tmd:
input x

// Writes a 1 on the input tape.

[x] E (1)
return

functions:
f
g

initvar:
E
```

For detailed documentation of the TMD programming language, see [25]. To find this directory specifically, navigate to parsimony/src/tmd/tmd_dirs/example_tmd_dir at [25].

C. Explicit Listing of *Z*

To find an explicit description of our Turing machine *Z*, please visit our repository at [25].

We ran this Turing machine for 10 000 000 000 steps (more than half a day on our simulators), and within that time it did not halt. We note, however, that *Z* was designed for parsimony rather than efficiency, and that this "experiment" is of little consequence! We similarly ran a Turing machine designed to test the conjecture that all perfect squares are less than 5, and it ran for 2 895 083 899 steps (a couple of hours on our simulator) before it found the counterexample 9 and halted.

References

[1] K. Gödel, *The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis with the Axioms of Set Theory*, Princeton, NJ: Princeton University Press, 1940.

[2] J. Schoenfield, "The Problem of Predicativity," in *Essays on the Foundations of Mathematics* (Y. Bar-Hillel et al., eds.), Jerusalem: Magnes Press, Hebrew University, 1961 pp. 132–142.

[3] T. Rado, "On Non-computable Functions," *The Bell System Technical Journal*, **41**(3), 1962 pp. 877–884.
doi:10.1002/j.1538-7305.1962.tb00480.x.

[4] A. H. Brady, “Solution of the Non-computable ‘Busy Beaver’ Game for $k = 4$,” in *Abstracts for: ACM Computer Science Conference*, Washington, DC, 1975, New York: Association for Computing Machinery, 1975 p. 27.

[5] S. Lin and T. Rado, “Computer Studies of Turing Machine Problems,” *Journal of the ACM*, **12**(2), 1965 pp. 196–212.
doi:10.1145/321264.321270.

[6] H. Marxen. “Busy Beaver.” (Oct 4, 2016) www.drb.insel.de/~heiner/BB.

[7] Wythagoras. “A Good Bound for S(7)?” (Sep 20, 2016)
googology.wikia.com/wiki/User_blog:Wythagoras/A_good_bound_for_S%287%29%3F.

[8] H. Marxen and J. Buntrock, “Attacking the Busy Beaver 5,” *Bulletin of the EATCS*, **40**, 1990 pp. 247–251.
www.drb.insel.de/~heiner/BB/mabu90.html.

[9] Deedlit11. “Okay, More Turing Machines.” (Sep 20, 2016)
googology.wikia.com/wiki/User_blog:Deedlit11/Okay,_more_Turing_machines.

[10] “Programming Puzzles & Code Golf.” (Sep 20, 2016)
codegolf.stackexchange.com.

[11] A. M. Ben-Amram and H. Petersen, “Improved Bounds for Functions Related to Busy Beavers,” *Theory of Computing Systems*, **35**(1), 2002 pp. 1–11. doi:10.1007/s00224-001-1052-0.

[12] H. Friedman. “Order Theoretic Equations, Maximality, and Incompleteness.” (Oct 12, 2016) u.osu.edu/friedman.8/foundational-adventures/downloadable-manuscripts #78.

[13] H. Friedman. “Order Invariant Graphs and Finite Incompleteness.” (Sep 20, 2016)
u.osu.edu/friedman.8/files/2014/01/FIiniteSeqInc062214a-v9w7q4.pdf.

[14] S. Aaronson, “The 8000th Busy Beaver Number Eludes ZF Set Theory: New Paper by Adam Yedidia and Me,” *Shtetl-Optimized* (blog). (Sep 20, 2016) www.scottaaronson.com/blog/?p=2725#comments.

[15] G. Chaitin, *The Limits of Mathematics*, p. 79. (Sep 20, 2016) archive.org/details/arxiv-chao-dyn9407003.

[16] C. S. Calude and E. Calude, “Evaluating the Complexity of Mathematical Problems: Part 1,” *Complex Systems*, **18**(3), 2010 pp. 267–285. www.complex-systems.com/pdf/18-3-1.pdf.

[17] C. S. Calude and E. Calude, “Evaluating the Complexity of Mathematical Problems: Part 2,” *Complex Systems*, **18**(4), 2010 pp. 387–401. www.complex-systems.com/pdf/18-4-1.pdf.

[18] J. Koza, "Spontaneous Emergence of Self-Replicating and Evolutionarily Self-Improving Computer Programs," in *Artificial Life III: Proceedings of the Workshop on Artificial Life, held June 1992 in Santa Fe, New Mexico* (C. G. Langton, ed.), Reading, MA: Addison-Wesley, 1994 pp. 225–262.

[19] A. N. Pargellis,"The Spontaneous Generation of Digital 'Life'," *Physica D: Nonlinear Phenomena*, **91**(1–2), 1996 pp. 86–96. doi:10.1016/0167-2789(95)00268-5.

[20] D. Woods and T. Neary, "The Complexity of Small Universal Turing Machines: A Survey," *Theoretical Computer Science*, **410**(4–5), 2009 pp. 443–450. doi:10.1016/j.tcs.2008.09.051.

[21] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002 p. 709.

[22] A. Smith. "Universality of Wolfram's 2, 3 Turing Machine." Submitted for the Wolfram 2, 3 Turing Machine Research Prize. (Sep 20, 2016) www.wolframscience.com/prizes/tm23/TM23Proof.pdf.

[23] S. O'Rear. "Metamath Turing Machines." (Sep 20, 2016) github.com/sorear/metamath-turing-machines.

[24] H. Friedman. "The Upper Shift Kernel Theorems." (Sep 20, 2016) u.osu.edu/friedman.8/files/2014/01/KernStruThm100910-1lu0b8v.pdf.

[25] A. Yedidia. "Parsimony." (Sep 20, 2016) github.com/adamyedidia/parsimony.

[26] M. Davis, Y. Matijasevic and J. Robinson, "Hilbert's Tenth Problem. Diophantine Equations: Positive Aspects of a Negative Solution," in *Mathematical Developments Arising from Hilbert Problems: Proceedings of Symposia in Pure Mathematics*, Vol. 28 (F. E. Browder, ed.), Providence: American Mathematical Society, 1976 pp. 323–378.

[27] D. Madore. "The Unlambda Programming Language." (Sep 20, 2016) www.madore.org/david/programs/unlambda.

[28] U. Müller. "Brainfuck: An Eight-Instruction Turing-Complete Programming Language." www.muppetlabs.com/~breadbox/bf.

[29] C. Barker. "Iota and Jot: The Simplest Languages?" (Sep 20, 2016) semarch.linguistics.fas.nyu.edu/barker/Iota.