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

Adam Yedidia

March 10, 2016

Abstract

Since the definition of the Busy Beaver function by Radó in 1962, an interesting open question has been what the smallest value of n for which $BB(n)$ is independent of ZFC. Is this n approximately 10, or closer to 1,000,000, or is it unfathomably large? In this paper, we show that it is at most 7,641 by presenting an explicit description of a 7,641-state Turing machine Z with 1 tape and a 2-symbol alphabet whose behavior cannot be proved in ZFC, assuming ZFC is consistent. The machine is based on work of Harvey Friedman on independent statements involving order-invariant graphs. [1] In doing so, we give the first known upper bound on the highest provable Busy Beaver number in ZFC. We also present an explicit description of a 4,888-state Turing machine G that halts if and only if there's a counterexample to Goldbach's conjecture, and an explicit description of a 5,372-state Turing machine R that halts if and only if the Riemann hypothesis is false. In the process of creating G, R, and Z, we make use of a higher-level language, Laconic, which is much more convenient than direct state manipulation.

1 Introduction

This paper is devoted to demonstrating the extreme difficulty of finding the values of the Busy Beaver function beyond a certain point. We cannot ever prove an upper bound on $BB(x)$ for any $x > 4,888$ or $x > 5,372$ without simultaneously providing a proof (or disproof) of the Goldbach or Riemann hypotheses, respectively. And we cannot ever prove an upper bound on $BB(x)$ for any $x > 7,641$ without assuming axioms more powerful than the axioms of ZFC, assuming ZFC is consistent.

We demonstrate these results by presenting explicit descriptions of Turing Machines, G, R, and Z whose *behavior* (whether or not they halt) implies the truth or falsehood of the Goldbach and Riemann hypotheses and the consistency of ZFC, respectively. If one knew a proof that $\mathrm{BB}(7,641) < u_G$ for some u_G, one could find out if G halts or loops by running G for u_G steps and seeing if it had halted by then; if it had not, we would be certain that G will never halt. Thus, an execution history of G for u_G steps would constitute a proof of the truth or falsehood of Goldbach's conjecture. The same logic holds for R, $BB(5{,}372)$, and the Riemann hypothesis.

Moreover, if one knew a proof that $\mathrm{BB}(7,641) < u_Z$, one could find out if Z halts or loops by running Z for u_Z steps Z for u_Z stepes and seeing if it had halted by then; if it had not, we would be certain that Z will never halt. Thus, an execution history of Z for u_Z steps would constitute a proof in ZFC (since ZFC can encode arithmetic, and therefore Turing Machine execution histories) of the consistency or inconsistency of ZFC. By Gödel's second incompleteness theorem, such a proof cannot exist if ZFC is consistent. Thus, no proof in ZFC exists of an upper bound on the value of $BB(7{,}641)$, assuming ZFC is consistent.

2 Laconic

The Laconic language is a programming language designed to be both user-friendly and easy to compile down to parsimonious Turing Machine descriptions. Laconic is so named because it's useful for demonstrating the simplicity of an algorithm by writing the shortest program possible that implements the algorithm. A short program will, in turn, compile to a Turing Machine with few states.

Laconic can also be interpreted. This will cause the Laconic program to be run line-by-line. Interpreting a Laconic program before compiling it down to a Turing Machine is highly recommended in order to verify the program's correctness.

Laconic is a strongly-typed language. It supports recursive functions.

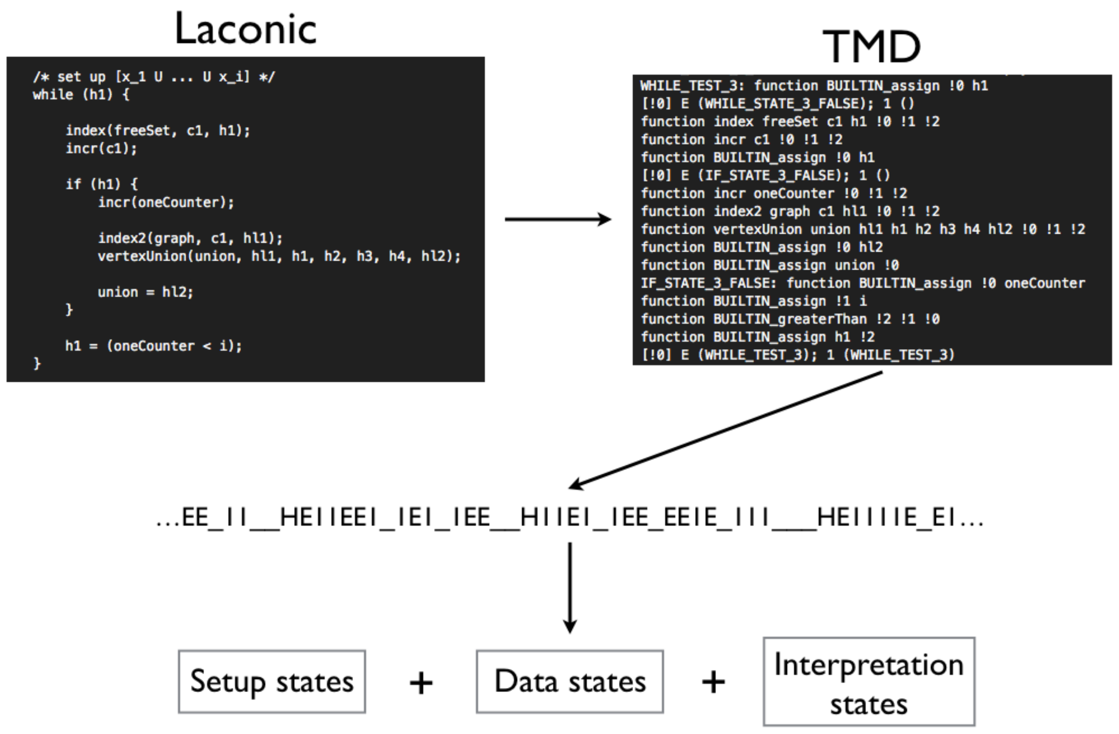
When Laconic is compiled, it is transformed into a TMD program, which is spread across many files (with one file per defined function). The TMD program is meant to represent a sequence of commands that could be given to a multi-tape, 3-symbol Turing Machine, using the Turing Machine abstraction that allows the machine to read and write from one head at a time.

For an example of a Laconic program, see Appendix [?]. For a visual illustration of the compilation process, see Figure 1

3 TMD

The TMD language is a programming language designed to help the user describe the behavior of a multi-tape, 3-symbol Turing Machine with a function stack. It allows the user to use three symbols: `_`, `1`, and `E`. The empty 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 $\_^\infty(1|E)^+\_^\infty$; in other words, each tape must always contain an infinite number of copies of the `_` symbol, followed by a string of `1`'s and `E`'s of size at least 1, followed by another infinite 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 each tape. Then, each Laconic command which manipulates the value of one or more variables compiles


Laconic
TMD
...EE_II__HEIIEEI_IEI_IEE__HIIEI_IEE_EEIE_III___HEIIIIE_EI...
Setup states
+
Data states
+
Interpretation states

```
/* 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: This figure gives a visual overview of the compilation process.

down to a TMD function call which 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 `a` to the value of `b*c`. It compiles down to the following TMD function call:

```
function BUILTIN_multiply a b c
```

This function call will result in `BUILTIN_add` 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 are 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. Moreover, when this top function returns, the Turing machine halts.
2. The `initvar` file. This file contains the non-_ symbols that start in each register at initialization.
3. Any files used to describe TMD functions. These files will all end in a `.tfn` extension and will only have any relevance to the compiled program if they show up in the functions file.

4 Compilation and Interpretation

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:

1. The *initializer* sets up the basic structure of the variable registers and the function stack.
2. The *programmer* 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 the programmer to the interpreter. In other words, initializer states point only to initializer states or to programmer states, programmer states point only to programmer states or to interpreter states, and interpreter states point only to interpreter states or the `HALT` state.

When discussing the layout of the tape symbols and patterns, there are two ways to think about it: one is with a 4-symbol alphabet ($\{\_, \mathtt{1}, \mathtt{H}, \mathtt{E}\}$, empty symbol _), and one is with a 2-symbol alphabet ({`a`, `b`}, empty symbol `a`). Naturally, the 2-symbol alphabet version is the one that is ultimately used for the results in this paper, since we advertised a Turing machine that made use of only two symbols. However, in nearly all parts of the Turing machine, the 2-symbol version of the machine is a direct translation of the 4-symbol version, according to the following mapping:

- `_` $\leftrightarrow$ `aa`
- `1` $\leftrightarrow$ `ab`
- `H` $\leftrightarrow$ `ba`
- `E` $\leftrightarrow$ `bb`

Additionally, the sections that follow may make reference to the `ERROR` state. Transitions to the `ERROR` state are stand-ins for transitions that will never be taken under any circumstances; as will be obvious in the sections that follow, there may be circumstances under which this situation can arise (although it is often indication that the Turing machine was not designed as parsimoniously as it could have been).

4.1 The Initializer

The initializer starts by writing a counter onto the tape which encodes how many registers there will be in the program. Using the value in that counter, it creates each register, with demarkation patterns in between registers, and unique identifiers for each register. Each register's value begins with the pattern of non-`_` 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 programmer.

4.2 The Programmer

4.2.1 Specification

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

Figure 3 is a detailed diagram describing the tape's state when the programmer passes control to the processor.

4.2.2 Introspection

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

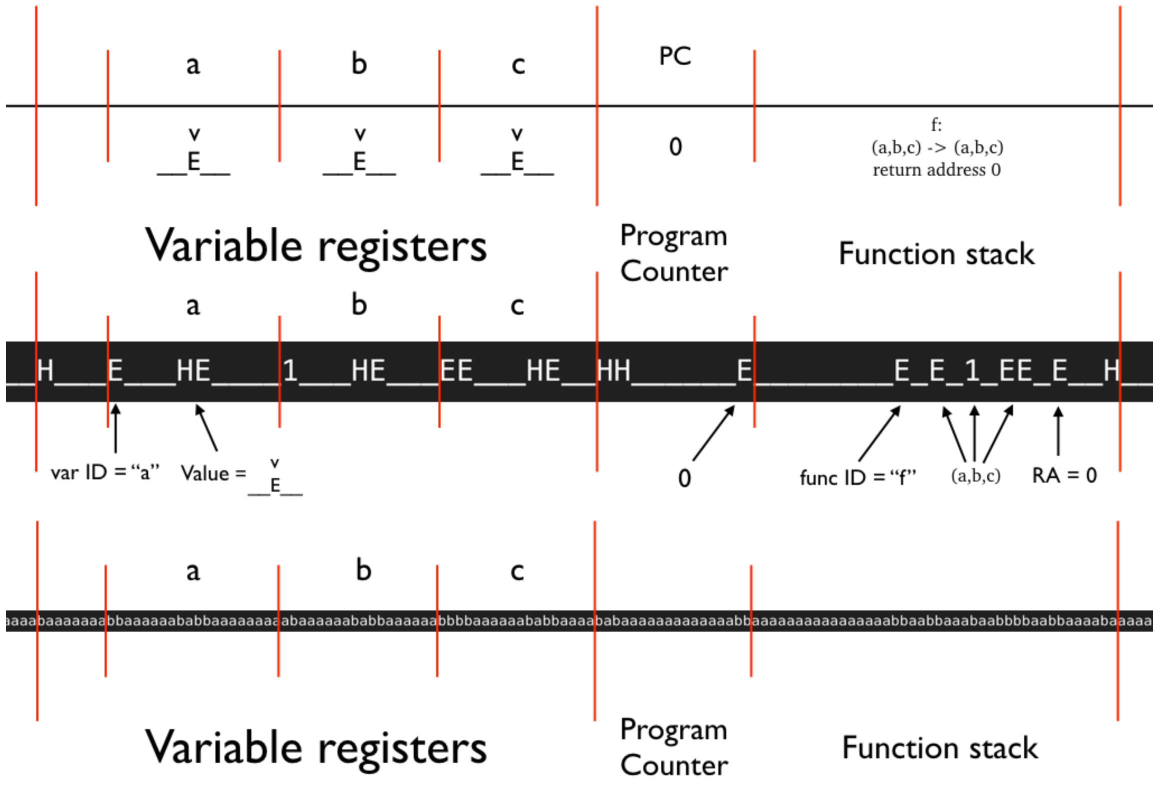


Figure 2: This figure shows the state of the Turing machine tape after the initializer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of what each part of the Turing machine tape represents. The middle bar is an encoding of the tape in the standard 4-symbol alphabet; the bottom bar is simply the translation of that tape into the 2-symbol alphabet. This figure is only meant to give the reader a sense for what the machine really looks like when written onto the Turing machine's tape. For a more detailed explanation of how to interpret the tape patterns, see [**?**].

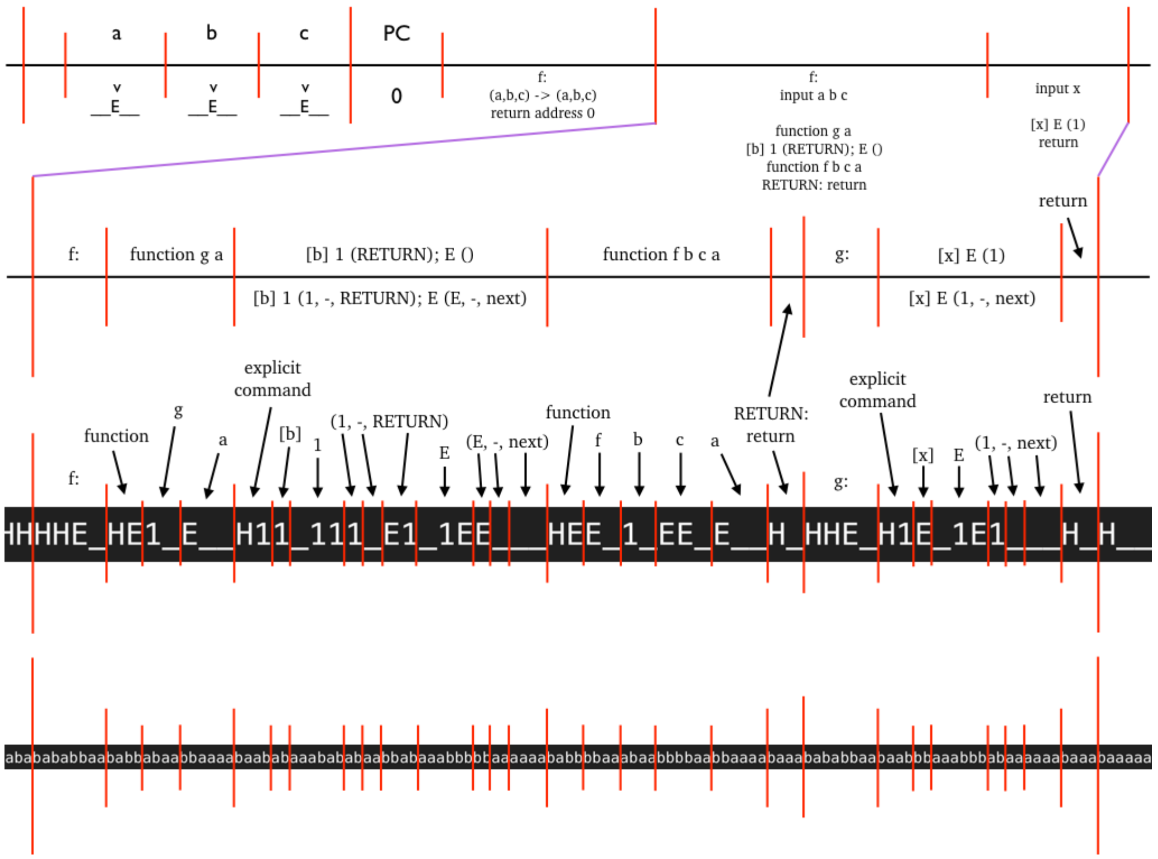


Figure 3: This figure shows the state of the Turing machine tape after the programmer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of the entire tape; unfortunately, at this point there are so many symbols 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 4-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 2-symbol alphabet. For a more detailed explanation of the encoding of TMD into tape symbols, see [?].

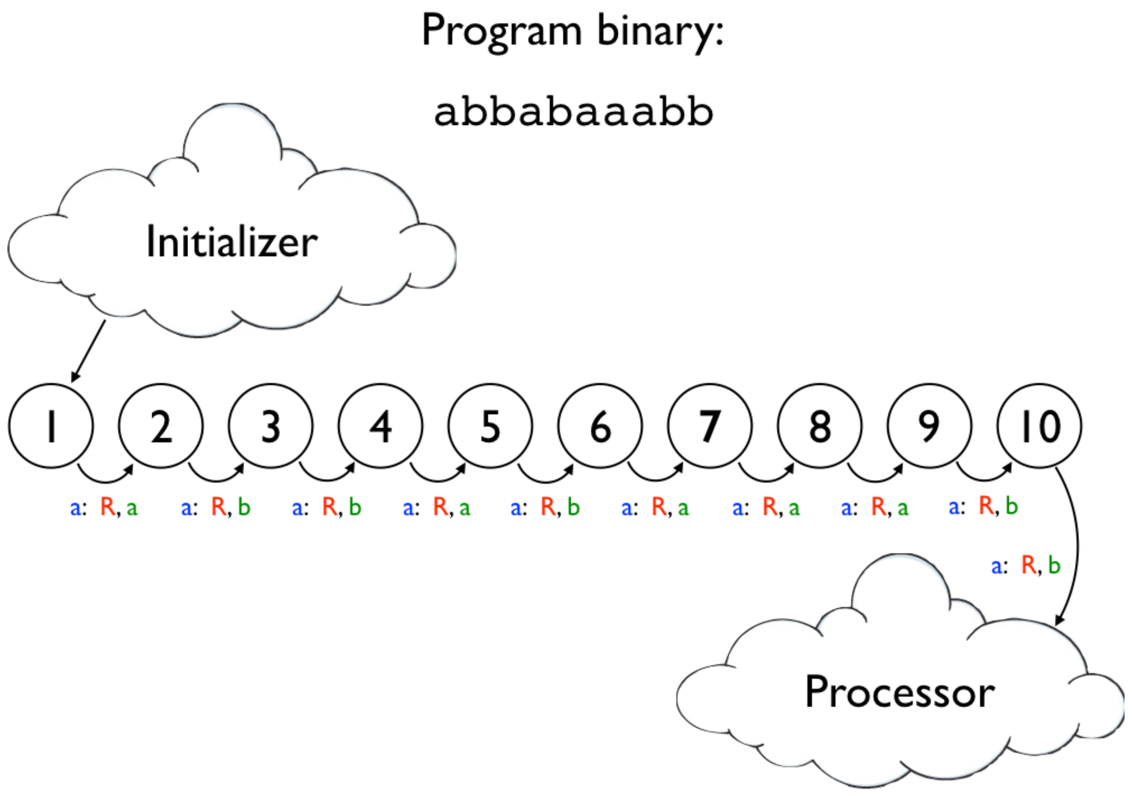


Figure 4: This figure shows a naive implementation of the programmer. In this example, the hypothetical program is ten bits long, and the programmer uses ten states, one for each bit. In the diagram, the blue symbol is the symbol that is read on a transition, the red letter indicates the direction the head moves, and the green symbol indicates the symbol that it written. Note the lack of transitions on reading a `b`; this is because in this implementation, the programmer will only ever read the empty 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 1-10 (and therefore `b` transitions are presumed to lead to the `ERROR` state)

This idea has the advantage of being very straightforward to implement and to understand, and if you don't think about it too hard it even appears to be optimal. Upon closer examination, however, it is apparent that 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 with each state. In a machine that contains n states, each state could contain $2(\log(n) + 1)$ bits of information: for each of its two transitions could point to any of the n states, and 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 will be difficult.

What we propose here is a scheme inspired by Luke Schaeffer (Is "inspired" good? Should I have said "invented"? "Proposed"? I want to make sure I'm properly attributing credit.). It does not achieve the optimal theoretical encoding described above, but is relatively simple to implement and understand, and is within a factor of 2 of optimal for large binary strings. He named Turing machines that use this idea *introspective*.

It works as follows. If the binary string contains k bits, then let w be the *word size*. w takes the largest value it can such that $w2^w \leq k$. We can split the binary string into $n_w = \left\lceil \frac{k}{w} \right\rceil$ different *words* of size w bits each (we can pad the last word with copies of the empty symbol). In our scheme, each word in the bit-string will be represented by a *data state*. Each data state will point to the state representing the next word in the sequence for its `a` transition, but which state the `b` transition points will encode the next word. Every `b` transition will point to one of the last 2^w data states, thereby encoding w bits of information.

Of course, the encoding is useless until it is specified 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 $\texttt{a}^{i-1}\texttt{ba}^{\infty}$ (a string of $i-1$ `a`'s 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 $\texttt{b}^{i-1}\texttt{ab}^{r}\texttt{a}^{\infty}$, 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 which removes one `b` at a time and increments the counter appropriately. Then, afterward, the extractor reverts the tape back to the form $\texttt{a}^{i}\texttt{ba}^{\infty}$, shifts the whole thing 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 has a visual description of the introspection algorithm.

How much have we gained by using an introspective technique for encoding the program binary, instead of the naive approach? Well, it depends on how large the program binary is. By using introspection, we incur 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.) But in return, we save a *multiplicative* factor of w (which scales with $\log k$) on the number of data states needed.

Is this worth it? Well, plainly not for the 10-bit example binary shown in Figs. 4 and 5. For that binary, we require 69 additional states for the extractor in order to save 5 states on the data states.

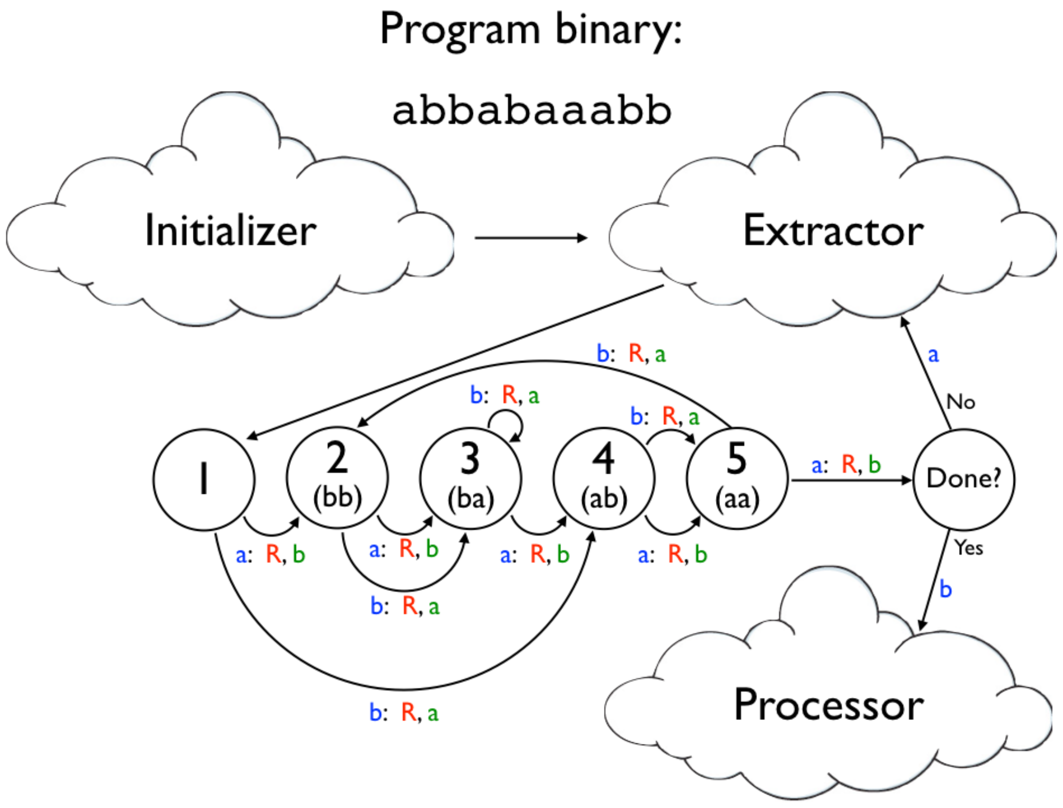


Figure 5: This figure shows an introspective implementation of the programmer. In this example, the hypothetical program is $k = 10$ bits long, and so the word size must be 2 (since $w = 2$ is the largest w such that $w2^w \leq 10$). There are therefore $n_w = \left\lceil \frac{k}{w} \right\rceil = 5$ 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 you mean to encode if you point to them.

But a 10-bit binary is unrealistically small; we only made it so small for illustrative purposes. What about for actual programs, such as the small example TMD program used in Figs. 2 or 3 and presented in full in Appendix B? Or what about the compiled Laconic programs that encode statements independent of Goldbach's conjecture, Riemann's hypothesis, and ZFC?

The following table shows the performance of the naive and introspective approaches on each of those four programs.

Program	Binary Size	w	n_w	Extractor Size	States Naive	States Introspective
Example TMD	116	4	29	57	116	86
Goldbach	4,964	9	552	107	4,964	659
Riemann	9,532	10	1,024	117	9,532	1,141
ZFC	35,906	11	3,265	127	35,906	3,392

As is apparent from the table, using introspective techniques for the programmer creates a big improvement, particularly for large programs. Even for programs so small as to be useless, such as the example TMD program used for illustration in this paper, introspection still improves, if only slightly, over the naive approach.

One very minor but notable detail is the numbers presented for the Riemann program. Ordinarily, with a binary of size 9,532, we would opt to split the program into 1,060 words of 9 bits each plus a 107-state extractor, since 9 is the greatest w such that $w2^w <$9,532. But because 9,532 is so close to the "magic number" 10,240, it's actually more parsimonious to pad the program with copies of the empty symbol until it's 10,240 bits long, and split it into 1,024 words of 10 bits each plus a 117-state extractor.

4.3 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 following this 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.

4.4 Cost Analysis

Before concluding this section, it is worthwhile to analyze the relative contributions of the initializer, the programmer, and the processor to the machine's final state count. Obviously, which of these contributions is greatest depends heavily on the size of the program being compiled. We have created a table containing the number of states in each of the initializer, programmer, and processor for each of the four different TMD programs we have been analyzing.

Program	Initializer	Programmer	Processor	Total
Example TMD	349	86	3,860	4,295
Goldbach	369	659	3,860	4,888
Riemann	371	1,141	3,860	5,372
ZFC	389	3,392	3,860	7,641

As can be seen from this table, the processor makes the larges contribution to every one of the four programs presented in this paper. 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 programmer will grow almost linearly but the cost of the processor will stay the same. Improving the programmer, therefore, and with it the TMD and Laconic languages, is probably the best approach for improving performance for very large and complex programs.

5 A Turing Machine Whose Behavior is Independent of ZFC

We present a 7,641-state Turing Machine that is *independent of ZFC*; it would not be possible to prove that this machine would halt or wouldn't halt using the axioms of ZFC, assuming that ZFC is consistent. It is therefore impossible to prove the value of $BB(7{,}641)$ to be any given value without assuming axioms more powerful than ZFC, assuming that ZFC is consistent.

For an explicit description of this machine, see 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 of ZFC repeatedly in each possible way so as to enumerate every statement ZFC could prove, and to halt if ever a contradiction was found. Such a machine's *behavior* (whether or not it ultimately halts) would necessarily be independent of ZFC, because to a proof of the machine's behavior would necessarily imply a proof about the consistency or inconsistency of ZFC. While the idea for this method is simple, to actually construct such a machine would be very involved, because it would require creating a language in which to encode the axioms of ZFC that could be stored on a Turing machine tape.

5.1 Friedman's Mathematical Statement

Thankfully, a simpler method exists for creating Z. Friedman [1] was able to derive a graph-theoretical statement whose truth cannot be proved in ZFC if ZFC is consistent, and which is false if ZFC is not consistent. These two properties are what we ultimately need to prove an upper bound on the highest provable Busy Beaver value in ZFC, and they are true of Friedman's statement, which follows:

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

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 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.

Also, 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) have 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 $[Q]^2$, in which each node corresponds to a set of two real numbers, and there is an edge between two nodes if and only if their corresponding sets a and b satisfy $a_1 < b_1 < a_2 < b_2$. (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.

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 $x \leq_{rlex} y$ and an edge exists between x and y. $x \leq_{rlex} y$ if and only if $x = y$ or $x_i < y_i$ where i is least such that $x_i \neq y_i$. [10]

5.2 Implementation Methods

In order to create Z, we needed to design a Turing Machine that would halt if Statement 1 was false, and would loop if Statement 1 was true. Such a Turning Machine's behavior would necessarily be independent of ZFC, because the truth or falsehood of Statement 1 is itself independent of ZFC. [1]

To design such a Turing machine, we wrote a Laconic program which encoded Friedman's statements, then compiled the program down to a description of a single-tape, 2-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 very detailed explanation of the full compilation process, please see [**?**].

Our Laconic program begins by looping over all non-negative values for k, n, and r. For each trio of values (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 $(((8knr)!)!)((8kni)!)/((8knj)!)$, 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 $\{x_1, \ldots, x_r, ush(x_1), ..., ush(x_r)\}$, 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 $\{x_1, \ldots, x_r, ush(x_1), ..., ush(x_r)\}$, 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 $ush(x_1), ..., ush(x_r)$ 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 conditon is satisfied.

6 G

We present a 4,888-state Turing Machine that is *independent of Goldbach's conjecture*; in other words, to know whether or not this machine halts is to know whether or not Goldbach's conjecture is true. It is therefore impossible to prove the value of $BB(4{,}888)$ without simultaneously implying a proof of the truth or falsehood of Goldbach's conjecture.

Goldbach's conjecture is stated as follows:
Statement 2. Every even integer greater than 2 can be expressed as the sum of two primes.

6.1 Implementation Methods

Because Goldbach's conjecture is quite simply stated, the Laconic program encoding the statement is also quite simple. We very briefly describe it here; once again, the documented source code itself, a detailed explanation of the compilation process, and documentation for the Laconic language are available at [9].

In order to create G, we loop over all even integers $i > 2$. For each i, we loop over all positive integers j such that $0 < j < i$. If no such j exists such that both j and $i - j$ are prime, we halt.

To test the primality of a positive integer j, we first test to see if $j = 1$; if it is, we determine that j is not prime. Otherwise, we loop over each integer k such that $1 < k < j$. If a k exists such that $j\%k = 0$ (where % denotes the modulus operation) then we determine that j is not prime; otherwise, we determine that it is.

7 R

We present a 5,372-state Turing Machine that is *independent of Riemann's hypothesis*; in other words, to know whether or not this machine halts is to know whether or not Riemann's hypothesis is true. It is therefore impossible to prove the value of $BB(5{,}372)$ without simultaneously implying a proof of the truth or falsehood of Riemann's hypothesis.

Riemann's 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.

7.1 Equivalent Statement

Instead of encoding the Riemann zeta function into a Laconic program, it is simpler to use the following statement, which has been shown to be equivalent to teh Riemann hypothesis: [11]

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:

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

7.2 Implementation Methods

The statement can be manipulated to contain only positive integers as follows:

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

$$\left(\left(\sum_{k\le\delta(n)}\frac{\delta(n)!}{k}\right)-\delta(n)!\frac{n^2}{2}\right)^2<36n^3(\delta(n)!)^2$$

$$\left(\sum_{k\le\delta(n)}\frac{\delta(n)!}{k}\right)^2-2\left(\sum_{k\le\delta(n)}\frac{\delta(n)!}{k}\right)\left(\delta(n)!\frac{n^2}{2}\right)+\left(\delta(n)!\frac{n^2}{2}\right)^2<36n^3(\delta(n)!)^2$$

$$\left(\sum_{k\le\delta(n)}\frac{\delta(n)!}{k}\right)^2+\left(\delta(n)!\frac{n^2}{2}\right)^2<36n^3(\delta(n)!)^2+2\left(\sum_{k\le\delta(n)}\frac{\delta(n)!}{k}\right)\left(\delta(n)!\frac{n^2}{2}\right)$$

Further reorganizing the statement:

$$a(n)=\sum_{k\le\delta(n)!}\frac{\delta(n)!}{k}$$

$$b(n)=\delta(n)!\frac{n^2}{2}$$

and let

$$l(n)=(a(n))^2+(b(n))^2$$
$$r(n)=36n^3(\delta(n)!)^2+2a(n)b(n)$$

.

When rewritten in terms of l and r, Riemann's hypothesis is equivalent to whether or not $l(n)<r(n)$ for all positive integers n.

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 $l(n)\ge r(n)$, our program halts.

Appendices

A Example Laconic Program: Goldbach's Conjecture

The following is an example Laconic program, which compiles down to the aforementioned Turing Machine G (which halts if and only 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 [9]. To find this file specifically, navigate to `parsimony/src/laconic/laconic_files/goldbach.lac` at [9]

B Example TMD Program

The following is an example TMD directory, which compiles down to a binary string to be written on a Turing machine's tape. It is the example that is used in illustrations throughout this paper, most notably in the example compilation shown in Figs. 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 [9]. To find this directory specifically, navigate to `parsimony/src/tmd/tmd_dirs/example_tmd_dir/` at [9]

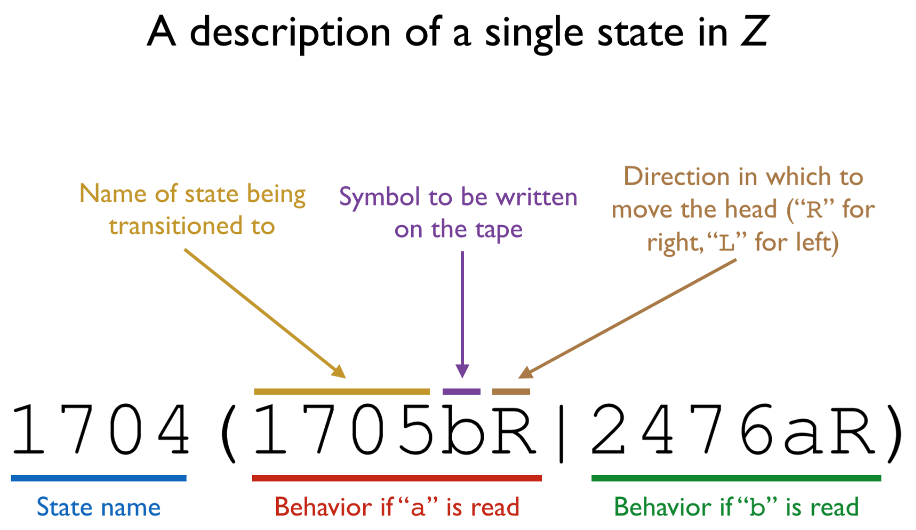

A description of a single state in Z
Name of state being transitioned to
Symbol to be written on the tape
Direction in which to move the head ("R" for right, "L" for left)
1704(1705bR|2476aR)
State name
Behavior if "a" is read
Behavior if "b" is read

Figure 6: This figure explains how to read a description of a single state. Note that "`ERROR-`" or "`HALT--`" denote transitions to the `ERROR` or `HALT` states, respectively (no further information is provided because what symbol is written and which direction the head moves are at that point irrelevant.)

C Explicit Description of Z

We present below an explicit description of Z. In order to prevent the Turing Machine from being outrageously long, our presentation is in a very compressed form. For a more easily readable version of Z, complete with descriptive state names, see [9].

Figure 6 presents useful information for how to interpret the description shown below. In addition, note the following:

1. The tape has a 2-symbol alphabet, with tape symbols $\{$`a`$,$ `b`$\}$ and empty symbol `a` (in other words, `a` is the only symbol that can appear an infinite times on the tape).
2. The start state of Z is state `0000`.
3. Z will never transition to the `ERROR` state. Any transition to the `ERROR` state could be replaced by a transition to any other state (including `HALT`) and the Turing Machine's behavior would remain identical.
4. Z contains only one transition to the `HALT` state, out of state `7593`.

```
0000(0001bR|ERROR-) 0001(0004bR|ERROR-) 0002(0003bR|ERROR-) 0003(0012aR|0012bR) 0004(0005bR|ERROR-) 0005(0006bR|ERROR-) 0006(0007aR|ERROR-) 0007(0008bR|ERROR-) 0008(0009bR|ERROR-) 0009(0010bR|ERROR-) 0010(0011bR|ERROR-) 0011(0002bR|ERRO
```

References

[1] Friedman, H. “Order Invariant Graphs and Finite Incompleteness.” https://u.osu.edu/friedman.8/files/2014/01/FIiniteSeqInc062214a-v9w7q4.pdf

[2] Rado, T. “On Non-Computable Functions.” Bell System Technical Journal, 41: 3. May 1962 pp 877-884.

[3] http://www.drb.insel.de/ heiner/BB/

[4] http://codegolf.stackexchange.com/

[5] http://cs.nyu.edu/pipermail/fom/1999-April/003014.html

[6] Marxen, H., Buntrock, J.“Attacking the Busy Beaver 5.”

[7] McLarty, C. “What Does It Take To Prove Fermat’s Last Theorem? Grothendieck and the Logic of Number Theory.” The Bulletin of Symbolic Logic, Volume 00, Number 0.

[8] Friedman, H. “Order Theoretic Equations, Maximality, and Incompleteness.” June 7, 2014. http://u.osu.edu/friedman.8/foundational-adventures/downloadable-manuscripts #78.

[9] https://github.com/adamyedidia/parsimony

[10] Personal communication between A. Yedidia and H. Friedman.

[11] Browder, F. “Mathematical Developments Arising from Hilbert Problems.” American Mathematical Society. Volume 28, Part 1.