

Turing Machine Documentation

Adam Yedidia

April 27, 2016

This document explains the structure of the Turing machines used in this project. It is intended for users who are curious about the algorithm. Note that this document also appears nearly verbatim in Section 8 (Compilation and Processing) at:

`parsimony/tex/busybeaver/busybeaver.pdf`

There are two ways to think about the layout of the tape symbols: with a 4-symbol alphabet ($\{_, 1, H, E\}$, blank symbol $_$), and with a 2-symbol alphabet ($\{a, b\}$, blank symbol a). The 2-symbol alphabet version is the one that's 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 2-symbol version of the machine is a direct translation of the 4-symbol version, according to the following mapping:

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

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.

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 2-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*. 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.

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

2 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 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 1 is a detailed diagram describing the tape’s state when the initializer passes control to the printer.

3 The Printer

3.1 Specification

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

Figure 2 shows the tape’s state when the printer passes control to the processor.

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 3.

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 machine with n states, each state could contain $2(\log_2(n) + 1)$ bits of information, because each of its two

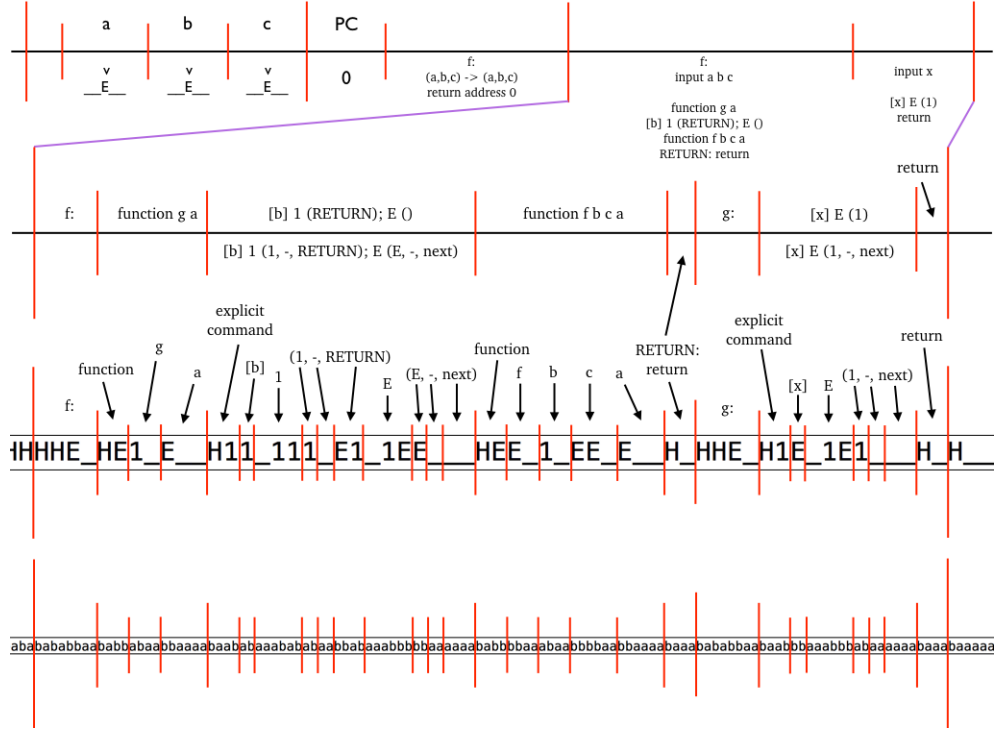


Figure 2: The state of the Turing machine tape after the printer completes. The top bar is a high-level description of the entire tape; unfortunately, at this point there are so many symbols on the tape that it is impossible to see everything at once. For a detailed view of the first two-thirds of the tape (registers, program counter, and stack), see Figure 1. 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.

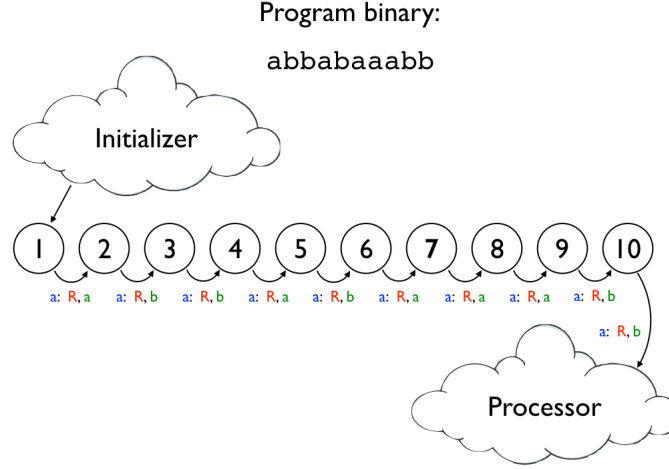


Figure 3: A naïve implementation of the printer. In this example, the hypothetical program is ten bits long, and the printer uses ten states, one for each bit. In the diagram, the blue symbol is the symbol that is read on a transition, the red letter indicates the direction the head moves, and the green symbol indicates the symbol that it written. Note the lack of transitions on reading a b; this is because in this implementation, the 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 1-10 (and therefore b transitions are presumed to lead to the ERROR state)

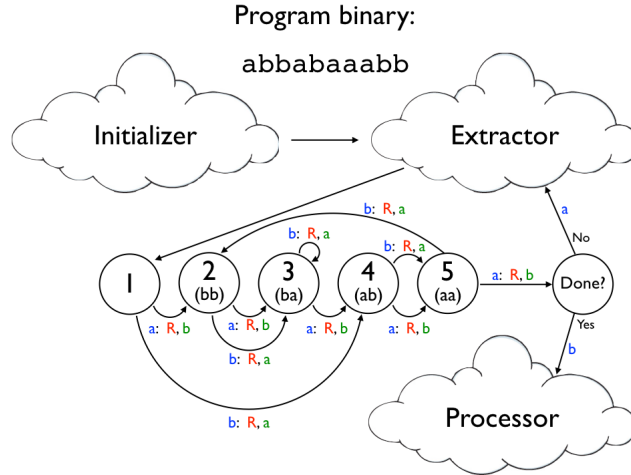


Figure 4: An introspective implementation of the printer. In this example, the hypothetical program is $k = 10$ bits long, and so the word size must be 2 (since $w = 2$ is the largest w such that $w2^w \leq 10$). There are therefore $n_w = \left\lceil \frac{k}{w} \right\rceil = 5$ data states, each encoding 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 we mean to encode if we point to them.

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 therefore 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 and refined further and suggested to us by Luke Schaeffer. 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. 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 $w2^w \leq k$. We can split the binary string into $n_w = \left\lceil \frac{k}{w} \right\rceil$ *words* of w bits each (we can pad the last word with copies of 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 $a^{i-1}ba^\infty$ (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 $b^{i-1}ab^ra^\infty$, assuming that the i^{th} data state pointed to the r^{th} -to-last data state. Thus, what we’re 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 $a^i ba^\infty$, 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 4 shows the whole procedure.

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’s possible to build a constant-size extractor, but it’s 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 Figs. 3 and 4. For that binary, we require 69 additional states for the extractor in order to save 5 data states. For real programs, however, it is worth it, as can be seen from the following table.

| Program | Binary Size | w | n_w | Extractor Size | States (Naïve) | 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 | 38,956 | 11 | 3,542 | 127 | 38,956 | 3,621 |

One minor detail concerns 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 blank 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 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.