TMD Documentation

Adam Yedidia

April 6, 2016

This document is intended for users who wish to learn how to program in TMD. For users who only want to know how to run or compile a TMD directory, without needing to create one of their own, information on how to do that is available in the `tmd_readme.pdf` file. The video tutorial at the root of this repository also explains how to run or compile TMD directories.

The top-level representation is a program written in the TMD language, which is a language designed to give the user a simple interface with which to program a multi-tape, 3-symbol Turing Machine with a function stack. Each of the many tapes in the execution of the program is infinite in one direction.

TMD code can be processed in two ways. First, it can be *interpreted*; that is, it can be directly evaluated line-by-line. This is generally done to verify a program's correct behavior, and to correct errors which would result in thrown exceptions in the interpreter but might lead to undefined behavior in the compiled Turing machine (because the compiled Turing machine is optimized for parsimony, whereas the interpreter need not be).

Second, TMD code can be *compiled* down to a description of a single-tape, 2-symbol Turing machine. It is highly recommended, however, to first interpret any piece of TMD code before compiling it, because the interpreter is much better for catching programming errors. The compiler is general-purpose and not restricted to compiling the programs discussed in this thesis. It is optimized to minimize the number of states in the resulting Turing machine, not to make the resulting Turing machine time- or space-efficient.

A TMD program is contained in a directory. It is a collection of *functions*, each of which contains a sequence of *commands*. Each function is

given its own file. Commands are separated by newlines. Each command is given its own line.

A TMD program also contains two special files which contain important information about the multi-tape Turing Machine: a file named `functions.tff` and a file named `initvar`.

What follows is a brief description of the syntax of the TMD language. A much more detailed description is available at [**?**]. The brief description contained here should be enough for a reader to gain a reasonable understanding of the programs written for this paper; any reader wishing to write her own programs is strongly encouraged read the more detailed documentation before proceeding.

1 TMD Files

Most of the logic contained within a TMD program lies in its TMD files, each of which describes one function in the program. TMD functions can make reference to the tapes in the machine and read the symbol under the head of each tape, much as a standard multi-tape TM can. In addition, TMD functions can call other TMD functions, pushing those functions to the top of the stack. Finally, TMD functions can return, popping themselves from the function stack.

A TMD function file is composed of four kinds of lines: an input line, calls to other functions, tape commands, and return statements.

TMD function files must use the extension `.tmd`.

1.1 The Input Line

Every TMD function file begins with an input line. TMD function files cannot contain more than one input line. The input line describes what tapes are going to be passed in as arguments to the function, and will therefore be readable and writeable within the function. An example of an input line might be:

```
input x y z
```

1.2 Labels

Any line in a TMD function other than the input line may be preceded by a *label*. A label takes the following form:

[alphanumeric string]**:** [line of code]

A label is an indicator attached to a line of code. Using tape commands, the line of code can be referenced in order to jump to the labelled line of code. The label can also be given a descriptive name that helps to explain the purpose of the line. This is perfectly legitimate even if the programmer has no intention of jumping to the line later; the addition of a label results in no additional states in the compiled program.

1.3 Function Calls

Function calls call a TMD function defined within the directory, placing the newborn function call at the top of the function stack with a new program counter of 0 (indicating that the first line of the called function should be the first to be read). The return address of the function call is the line following the function call.

Function calls have the following form:

`function` [function name]([argument name])*

When a function call is executed, the function that is described by the file [function name]`.tmd` is executed, starting with its top line. If no file [function name]`.tmd` exists within the program directory, an error is thrown at compile-time or upon interpreting the function call.

An example of a function call might be:

```
function add x y
```

1.4 Tape Commands

Tape commands begin with a marker that indicates which tape is going to be used in the command. They then explain what to do if each possible symbol is read.

In TMD, there are three legal tape symbols; they are `1`, `E`, and `_`. `_` is the empty symbol (the only one which can appear an infinite number of times on any tape). Furthermore, in TMD, it is a requirement that each tape always have the form $\texttt{\_}(\texttt{1}|\texttt{E})^{+}(\texttt{\_})^{\infty}$ (here, the $(.)^{+}$ operation denotes repetition any positive number of times, and the $(.)^{\infty}$ operation denotes repetition an infinite number of times). This requirement is for ease of compilation down to a single-tape TM; later, when the TMD is compiled down a Turing machine and that Turing machine is searching the registers for a register that matches a specific ID, the search will assume that the register organization will alternate (identifier, value, identifier, value) and that a value will be represented as a "chunk" of 1's and E's. This is why the above requirement must be respected.

Tape commands have the following form:

`[`[variable name]`]` ([symbol read] `(` [list of reactions] `)`)*

The list of reactions is a series of words separated by commas and encompassed by parentheses. Different reactions are separated by semicolons. These reactions can include what direction to move, what symbol to write, and what line to jump to after execution.

The symbols `1`, `E`, and `_` are used to denote what symbol to write on the tape after reading. The symbols `R`, `L`, and `-` are used to denote what direction the head should move after writing (with `R` indicating a move rightward, `L` indicating a move leftward, and - indicating a command to remain stationary). Any other string denotes a label, indicating a jump to the relevant label. Therefore, TMD programmers should not name their labels R (for example)! This will confuse the compiler. Additionally, TMD programmers should not put two conflicting reactions in the same command, such as two different move commands.

Any type of reaction that is excluded from a parenthetical will be presumed to be the default. The default for each type is:

-Symbol written: whatever symbol was read.
-Head move direction: - (in other words, no move).
-Jump: go to the next line.

As an example, the reaction `1 (E, -, LABEL)` would indicate that

in response to reading a `1` on the tape, an `E` should be written, the head should not move, and the next line of code to be executed should be labelled LABEL. It would have identical behavior to the reaction `1 (E, LABEL)`, because – is the default behavior for which direction to move the head.

An example of a full tape command might be the following:
`FIND_E: [x] 1 (R, FIND_E); E ()`

This tape command would indicate that on the tape named `x`, if a `1` is read off the tape, then a `1` should be written back onto the tape (because that is the default), the head should move right, and the line of code should be re-executed. If an `E` is read off the tape, then an `E` should be written back onto the tape, the head should not move, and execution should proceed to the next line of code (because those are the defaults). Because no reaction is provided in the case where a `_` is read off the tape, an error will be thrown if that is the symbol that is read, both in interpretation and in the Turing machine that results from compilation.

It should be noted that while there is no difference between explicitly writing out the default and leaving it implied for head moves and for symbol writes (so, for example, a TMD program including the line `[x] 1 (1)` will compile to an identical machine as a program including the line `[x] 1 ()`), there *is* a difference between explicitly writing out the default and leaving it implied for jumps. This difference manifests itself only in terms of parsimony and not in terms of behavior, and therefore is only visible when compiling the program, not when interpreting it (since during interpretation, only behavior matters). This is because while in the case of an explicit jump, the jump address must be written out onto the tape, there is a much shorter abbreviation for the default if it is simply a jump to the next line. Figure **??** shows this case in more detail.

1.5 Return Statements

Return statements indicate the end of a function's execution. After a function returns, that function's call is popped off the function stack, and execution continues at the line of code following the line that called the function that just returned. If the function stack is empty, and there is no function to return to, the program halts. This will happen when the first function to be called finally returns.

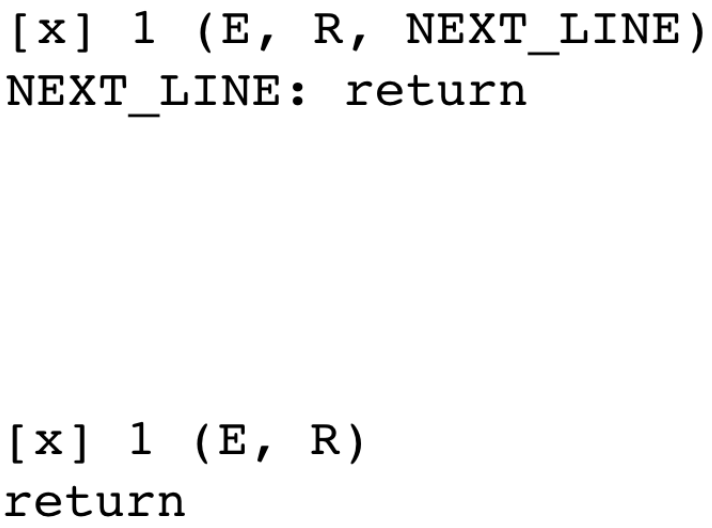

Figure 1: This figure shows two code snippets. The two snippets will have identical behavior when interpreted or compiled, because the default is to jump to the next line. However, the bottom snippet will compile to a more parsimonious Turing machine (albeit one that behaves identically) because the compiler will know to use the default abbreviation for "go to the next line," whereas in the top case the compiler will have to write out the jump address for the line labelled `NEXT_LINE`. Therefore, it is recommended that programmers concerned with parsimony use the approach shown by the bottom snippet.

Execution reaching the end of a function without returning will cause a runtime error.

2 The `functions` File

Every TMD directory must contain a file named `functions` (no extension). The `functions` file contains a list of every TMD function file in the directory that is used in the program. Note that not every TMD function file in the directory needs to be listed! If any such files are not listed, however, they must not be used in the program. If there is ever a call to a function not present in the `functions` file, there will be a runtime error (if the program is being interpreted) or a compilation error (if the program is being compiled).

When a TMD program is run, the first function to be called will be the function at the top of the `functions` file.

In the compiled TMD binary, the functions near the top of the `functions` file will be given shorter binary strings as names. Therefore, it is advantageous to put more frequently-called functions near the top of the `functions` file. (The Laconic-to-TMD converter does this automatically.)

3 The `initvar` File

Every TMD directory must contain a file named `initvar` (no extension). The `initvar` file contains a sequence of `1` and `E` symbols; this will be the sequence of symbols on each tape at initialization. So, for example, if the `initvar` file contains the string `11E1`, then each tape will be initialized with the values $\_\texttt{11E1}(\_)^{\infty}$.

At initialization, every tape's head will begin just above the first non-$\_$ symbol.

No `initvar` file may contain characters other than `1` and `E`. The `initvar` file cannot be empty.

4 Example

I have provided a simple example of a TMD directory to make it easier to understand what's going on. It can be found at `parsimony/src/tmd/tmd_dirs/example_tmd_dir`. Before reading on, please navigate to this directory—I will make reference to its contents in this section.

What happens when this program is run? Well, to know where the program starts, we look at the `functions` file. The `functions` file lists two functions, `f` and `g`. To know how the program starts, we mostly care about which function is at the top of the list; in this case, that's `f`. Therefore, execution is going to start at the top of the file describing `f`. At the top of the file describing `f` we see the line "`input a b c`"; because `f` is the first function, this tells us that our program execution will make use of three tapes, named `a`, `b`, and `c`, respectively.

We also need to know how each tape is initialized. In order to know that, we need to look at the `initvar` file. In this directory, the `initvar` file contains only the string "`E`"; that tells us that every tape in the program is initialized to the value $\_E\_^{\infty}$.

Now we are ready to begin the program. Our three tapes contain an `E` symbol and are otherwise covered in _'s. Our function stack begins with a single call to `f`. We look at the top command of `f.tmd`, ignoring the comment. It reads: "`function g a`". This indicates that the first thing we must do is call the function `g`, with the tape `a` as an argument.

This we do, placing a call to `g` at the top of the stack. Execution now proceeds from the top of `g`. We see from the input line of `g` that it takes one input, named `x`. We recognize that because `g` was called on the tape `a`, any references to `x` in the call to `g` are in fact going to read and modify `a`.

We look at the top command of `g`, ignoring the comment. It reads: "`[x] E (1)`". This indicates we are to read "`x`" (which we know to in fact refer to `a`) and read the symbol on it. If that symbol is an `E`, we will write a 1 onto the tape, keep the head stationary, and proceed to the next line; otherwise, we will throw an error (because no instructions are provided for how to handle symbols other than `E`).

In fact, all three tapes currently have a `E` symbol under the head, be-

cause that's how they were initialized. Thus, we read the `E` symbol from off `a`, replace it with a `1` symbol, and proceed to the next line.

The next line reads: "`return`". This tells us to pop the call to `g` off the stack and continue execution where we were in the function that called `g`. This leads us to the second command of `f`, which reads: "`[b] 1 (RETURN); E ()`". This tells us to read what is under the head on tape `b`, and if it's a `1` to go to the line of code labelled `RETURN` and otherwise to proceed to the next line of code. In either case, we don't change what's written on the tape, nor do we move the tape head.

Recall that at the moment, `a` has a `1` symbol, but both `b` and `c` are unchanged from when they were initialized. Therefore, we read a `E` symbol from off the tape; this tells us to proceed simply to the next line.

This is where things start to get a little bit confusing. The next line tells us to call `f` recursively, on a permutation of its own inputs. This we do, placing a new call to `f` on top of the old call to `f`. We look at the input line of `f`; it says "`input a b c`". This indicates to us that in this function call, the first argument to `f` will be called "`a`", the second "`b`", and the third "`c`". Recall that when we called `f` from within the original function, we called `f` on `b`, `c`, and `a`, in that order. This means that in the new function call, "`a`" will refer to the first argument of the function call, or `b`. Similarly, "`b`" will refer to `c`, and "`c`" to `a`.

With that in mind, let's begin executing the new call to `f`. The first command is a command to call `g` on "`a`", which in fact refers to `b`. The function `g` reads `b`, and finding an `E` under the head, replaces it with a `1` and returns. Our three tapes `a`, `b`, and `c` now contain a `1`, a `1`, and an `E`, respectively.

The next command tells us to read the symbol on "`b`", which in fact refers to `c`. Given that `c` still contains an `E`, we proceed simply to the next line. The next line tells us to make a *new* call to `f`, with "`b`" (actually `c`) as the first argument, "`c`" (actually `a`) as the second argument, and "`b`" (actually `c`) as the third argument.

Still following? To know how to interpret which names point to which tapes in the newborn call to `f`, we look its input line; it says (still!) "`input a b c`". This indicates to us that in this new function call, the first argu-

ment to f will be called "a", the second "b", and the third "c". But remember that in the function that called this one, c was the first argument, a was the second argument, and b the third! Therefore, in this newest function call, "a" will refer to c, "b" to a, and "c" to b.

That was one of the more confusing paragraphs that I've written in my lifetime, so let's take a step back. We have three tapes, a, b, and c, that contain 1, 1, and E, respectively. Our function stack contains three calls to f. Across those function calls, the names "a", "b", "c" refer to different things. In the bottom call, the names ("a", "b", "c") refer to (a, b, c). In the middle call, the names ("a", "b", "c") refer to (b, c, a). And in the top call, the names ("a", "b", "c") refer to (c, a, b).

Okay. Now we can start executing the top call to f. The top command tells us to run g on "a" (actually c). The function g reads c, and finding an E under the head, replaces it with a 1 and returns. Now our three tapes all contain a 1. Next, we execute the command "[b] 1 (RETURN); E ()". This tells us to read "b" (actually a). Upon reading a, we see that it contains a 1. We are told to jump to the line of code labelled RETURN. The line of code labelled RETURN says "return"; thus, we pop the top call to f off the stack, and we continue execution where we were in the function that called f.

The function that called f was itself a call to f, but with a different mapping from tape names to actual tapes. The line after the function call is the last line of f, which reads "return". Thus, this middle call to f is also popped off the stack, and we continue execution in the original call to f.

The line we were on in the original call to f was also the last line of f. It reads return. Therefore, we pop it off the stack too.

Now the stack is empty. If ever the stack is empty, execution halts. Therefore, we halt here, leaving the three tapes containing a 1.

For a pictorial representation of what happened here, navigate to parsimony/src/tmd/tmd_meta and enter the command python tmd_interpreter.py example_tmd_dir. Then, read the output file example_tmd_dir_history.txt, found in the parsimony/src/tmd/tmd_histories/ directory.