Lecture 6: Turing Machines: Introduction and Examples

Michael Beeson

Analysis of mechanical computation

- a machine has “internal states”
- and communicates with the world by reading and writing symbols one at a time
- and changes its state according to prescribed rules, based on its current state and the symbol being read.
- So it proceeds in a sequence of steps, reading, writing, changing state.
- It can also not proceed further; that is called “halting”.

Modeling machinery by isolating fundamental operations

- The reading and writing of symbols is done one symbol at a time
- But the time could be short, so e.g. updating all the pixels on a big monitor could be considered as updating one pixel at a time.
- The simplest way to model this is that the symbols are written on a "tape" marked off into squares just big enough to hold one symbol.
- It can be shown rigorously that, e.g., two-dimensional models of data reading and writing are not "stronger."
- The idea is to isolate a few very basic operations and argue that more complex (computing) machines can be built out of these simple operations.
- This is a theory of symbol-manipulation machines, not a theory of robots or manufacturing.

The tape and tape head

- The tape extends infinitely in both directions.
- Or, it's finite, but there are attendants ready to attach more tape as required.
- This is not a theory about the use of computing resources such as time or space; it is about computability in principle (not in practice).
- There is a "read-write head" that, at any moment, is located on just one square.
- It can move right or left one square, and it can change the symbol on that square.
- "blank" is considered a symbol, so writing is not different than overwriting.

States

- There are finitely many states for each Turing machine.
- These have names or numbers to identify them.
- For example, to think of your laptop this way, its state at any moment is specified by the contents of each bit of memory (including ALL memory, RAM, video, keyboard chips, etc.) plus one more bit for whether the computer is on or off. If you have N bits of memory then the number of states is 2^{N+1}.
- In this view, your screen isn't really "output" as you're just viewing some of the contents of video memory.
- The machine moves from state to state, under the influence of its program (operating system) and your input

Computing functions

- We want to model the situation where you use your computer for a specific purpose. You type in some “input”, and then after a while it writes your “output” to the screen or printer.
- Then it has “computed” that output.
- If you can repeat this with different input, then the machine “computes a function”, namely the function that leads from an input x to an ouput $f(x)$.

Computing in binary or unary

When we talk about computing a function of integers, we have to write the input on the tape to start, and then put the machine in the start state reading the beginning of the input.

- It is supposed to run and leave the output on the tape starting at the final location of the head.
- We speak of “computing in unary” if input and output are written in base 1 notation (that is, 3 is 111 and 7 is 1111111).
- We speak of “computing in binary” or “computing in decimal” if input and output are to be written in binary or decimal notation.
- functions of several variables have the arguments separated by a single blank. Blank means ascii 32. Null is ascii zero. The part of the tape not used for input is initially filled with null.

Formal definition of Turing machines

- A *state* is an integer (of any size); the *start state* is 0.
- A *symbol* is an 8-bit integer; the *blank* is 0.
- A *direction* is either 0 ("stay"), 1 ("left"), or 2 ("right").
- An *instruction* is a 5-tuple of integers, interpreted as (*old-state*, *old-symbol*, *new-state*, *new-symbol*, *direction*).
- A Turing machine is a finite sequence of instructions.
- There is no difference between a "Turing machine" and a "Turing machine program"; we are not talking about physical machines, but abstract machines.

List or set of instructions?

- What if there are two or more instructions for what to do when reading a certain symbol in a certain state?
- Take the first applicable one in the list.
- Or, if TM is defined as a *set* of instructions, require that there be no contradictory instructions.
- Allowing a choice between contradictory instructions defines "non-deterministic Turing machine", which we do not consider further in the immediate future.

Some details that can vary in other presentations

- We follow Kleene, Chapter XIII, section 67.
- Two-way infinite tape (as opposed to one with a left end)
- Moves are L, R, C for left, right, and center (no move).
- A single move can change the symbol, then move L, R, or C, as opposed to having different kinds of instructions for read-write or move.
- We place a fixed limit on the size of the alphabet (8 bits). It turns out that a larger alphabet can't compute more functions, so this is not a serious restriction.
- However, you can't limit the possible number of states, so the names for states can't have a bounded length either.
- For practical (programming) purposes, we allow arbitrary strings of letters and digits (no commas, etc.) as state names.
- No special "halt" instruction. The machine halts when there is no applicable instruction.

Examples

There are many online TM emulators. We will practice writing TM code and learn some programming tricks. One writes TM programs in a text file, and pastes them into the emulator.

You have three examples to work out for homework. Test them in the emulator. Turn in a printout of your working, commented code. I am going to trust you that they run in the emulator; otherwise I would have to ask for a copy of the files.

An example: Erase

To describe the coding of more complex machines, we need to able to use somewhat higher-level instructions, such as “copy” or “erase”. For example, “erase every square from the scanned square to the first occurrence of the symbol $\#$”.
Turing machine programs are best written in a text file. Then you can debug them using an online emulator. We will discuss some examples in class.

Copy

In the “copy” routine, the source and destination of the copy need to be specified, and the terminating character. Then the contents of squares, starting with the source square and continuing to the right until the first occurrence of the terminating character (but not including it) are to be copied to the squares starting with the destination and continuing to the right. Whatever is currently on those squares will be overwritten. The source will not be changed. The source and destination can be specified in various ways; for example, one common need for copying has the destination as the scanned square, and the source as the first square to the left of the scanned square containing a specified symbol, for example $\#$. The behavior is unspecified should you be so foolish as to allow the source and destination to overlap.

Copying via Turing machine code

Copying has to proceed one character at a time; we move left, fetch one character, "carry" that character to the right until we reach the correct location to "dump" or write it. Then we move back to the left to fetch the next character. There are two problems to solve: (1) how do we know when to stop for the next character at the left, or to dump at the right, and (2) how do we remember the character being "transported"? As shown in Exercise 6.3, it is possible in a Turing machine to "remember" a fixed finite number of characters. That solves problem (2) directly, and indirectly solves problem (1) also: One can leave a "marker" (a symbol not otherwise used) to indicate "where we have so far copied up to", and similarly, a marker to indicate the next destination square. Exercise 6.1 addresses the issue of being sure that there are some unused symbols to use for markers.