Turing Machine Analysis

Eric C.R. Hehner

Turing Machine Definition

Here is a Turing Machine [5] written in ProTem [0]. I have taken some of it from Darcy Otto [4]. I have numbered the lines so I can refer to them in the explanation of the program, which follows.

```
0    new symbols:= 0,..5.  new states:= 0,..8. `the number of symbols and states must be finite
1    new tape: ∞*symbols:= ∞*0. `infinite tape, initially all 0s
2    new position: nat:= 0.  `the position of the read-write-head, initially  0
3    new state: states:= 0.  `the current state, initially  0
4    new time: nat:= 0. `time measured as the number of actions performed
5    new program:=  `the state transition matrix;  this is the TM program
6            `  Row  r  means we are in state  r .
7            `  Column  c  means the tape at the read-write-head position is showing  c .
8            `  At row  r  and column  c , the item is a triple  [n; h; s] .
9            `       n  is the symbol (number) to be printed.
10           `                to erase, print space;  to leave alone, print the same symbol.
11           `       h  is an integer indicating read-write-head movement.
12           `            h>0  means move right  h  squares
13           `            h=0  means do not move
14           `            h<0  means move left  –h  squares
15           `       s  is the next state.
16     `      0 " "        1 "("        2 ")"        3 "0"        4 "1"          tape symbols
17     [ [  [1; 1; 1];     [1; 0; 0];   [2; 0; 0];   [3; 0; 0];   [4; 0; 0]  ];   `state 0
18       [  [3; 2; 2];     [1; 0; 1];   [2; 0; 1];   [3; 0; 1];   [4; 0; 1]  ];   `state 1
19       [  [3; –2; 3];    [1; 0; 2];   [2; 0; 2];   [3; 0; 2];   [4; 0; 2]  ];   `state 2
20       [  [0; 0; 3];     [1; 0; 3];   [2; 0; 3];   [3; 0; 5];   [4; 1; 4]  ];   `state 3
21       [  [2; –3; 3];    [1; 0; 4];   [2; 0; 4];   [3; 0; 4];   [4; 0; 4]  ];   `state 4
22       [  [4; –1; 6];    [1; 0; 5];   [2; 0; 5];   [3; 2; 5];   [4; 2; 5]  ];   `state 5
23       [  [0; –2; 6];    [1; 1; 7];   [0; 1; 5];   [3; 0; 6];   [4; 0; 6]  ];   `state 6
24       [  [3; –2; 3];    [1; 0; 7];   [2; 0; 7];   [3; 2; 7];   [4; 2; 7]  ] ].  `state 7
25
26   for r: states
27   ⟦for c: symbols
28    ⟦if –((program r c 0 : symbols)  ∧  (program r c 2 : states)) ⟦!"error 0".  stop⟧⟧⟧.
29
30   loop ⟦new action:= program state (tape_position).
31         if (action 0 = position)  ∧  (action 1 = 0)  ∧  (action 2 = state) ⟦!"halt".  stop⟧.
32            tape:= tape◂position▸action 0  `"print" a symbol on the tape
33         ‖ if position + action 1 ≥ 0 ⟦position:= position + action 1⟧  `move head
34            else ⟦!"error 1".  stop⟧`attempt to move left of the start of the tape
35         ‖ state:= action 2.  `next state
36         time:= time+1.  loop⟧
```

Line 0 defines *symbols* to be the numbers $0, 1, 2, 3, 4$ (x,..y includes x but excludes y), and defines *states* to be the numbers from (including) 0 to (excluding) 8 . Following these definitions there is a comment saying that the number of symbols and states must be finite.

Line 1 defines *tape* to be a variable whose type is an infinite string of *symbols* , and whose initial value is an infinite string of 0s. Turing's tape was an infinite string of squares, and each square holds a symbol from a finite alphabet, and initially each square holds a blank space. I have changed the symbols to natural numbers because I will want them to index the *program* array; we will come to it in a moment. This change does not affect the nature of the machine. I refer either to the numbers on the tape, or equivalently to the symbols they represent; 0 represents a blank space " " , 1 represents a left bracket "(" , 2 represents a right bracket ")" , 3 confusingly represents the digit "0" , and 4 confusingly represents the digit "1" .

Line 2 defines *position* to be a natural number variable, initially 0 . It will be used to index *tape* , telling which square the read-write-head is over.

Line 3 defines *state* to be a variable that can be assigned to any value in *states* , and is initially assigned to 0 . It tells what state the machine is in. Turing used letters in a strange font. I use natural numbers because I need them to index the *program* array. The "state of the machine" is not just the value of variable *state* ; it is the combination of *tape* , *position* , and *state* ; Turing called this combination the "configuration". The configuration determines the current action and all future actions and configurations.

Line 4 defines *time* to be a natural number variable, initially 0 . It counts the number of actions performed. This variable is not used to control the machine; it is used for analysis.

Line 5 declares *program* to be an array constant. Lines 6 through 15 are comments describing *program* . On line 16 there is a comment saying, for each column, what number corresponds to what symbol. The array constant is on lines 17 through 24. This is the Turing Machine program. Its rows are indexed by *state* . Its columns are indexed by *tape_position* , which is the symbol (number) in the current square of the tape. The item at row r and column c specifies the action to be taken when the machine is in state r and the read-write-head is reading c . The item is a triple $[n; h; s]$, where n is the symbol to be printed at the current tape location, h is the movement of the read-write-head, and s is the next state. In addition to printing, Turing included an erase action, which is unnecessary because it just means printing a blank space. Turing also allowed no printing to take place; for us, that is printing the symbol that is already there. If $h>0$ the read-write-head moves right h places; If $h=0$ the head does not move; if $h<0$ the head moves left $-h$ places. The machine is initially in state 0 ; after printing and head movement, the state changes to s , and the cycle repeats.

Lines 26 through 28 are a safety check that in action $[n; h; s]$, n is a symbol and s is a state. If not, "error 0" is printed and execution stops. In all correct programs, including the above example program and all Turing's programs, the problem does not arise.

Line 30 defines *loop* to be a named-program, which is then called in line 36, creating a loop. In some other programming language, these lines would be written **while** *true* **do** ... **od** . Line 30 continues by defining *action* to be *program* indexed by *state* and *tape_position* ; this makes *action* be the triple $[n; h; s]$ indicating the action to be performed.

Line 31 checks whether the current action is to halt (explained below), and if so, prints "halt", and halts.

Line 32 says print symbol *n* on the tape. In some other programming language, this line would be written *tape*(*position*):= *action*(0).

Line 33 moves the read-write-head. Turing did not say what would happen if the machine were directed to move the head to the left of the start of the tape. In a correct program, including the above program and all Turing's programs, the problem does not arise. When it arises, on line 34, "error 1" is printed, and execution stops.

Line 35 updates the *state* variable. Lines 32, 33, 34, and 35 are executed in parallel, but they could just as well be executed in sequence.

Line 36 adds 1 to the *time* variable, and then takes us back to the start of the loop on line 30 for the next iteration.

The messages printed on lines 28, 31, and 34 could be much more informative. With more printing, we could watch the execution.

Each of Turing's example programs generates the digits of a computable number. These programs do not have any input. Turing always starts with a blank tape. Theoretically, programs do not need input because the first part of the program can define (or generate) any constants instead of input. Our example, which is one of Turing's examples, works this way. But as a practical matter, input is useful, so if you would like to have input, you can put it on line 1 as the initial value of the tape.

Turing wrote a program as a list; each item in the list is a pair (state, tape-symbol) followed by the action triple (print symbol, head movement, next state) to be performed. But he did not give an action for all (state, tape-symbol) pairs. If the machine is in a state and looking at a tape symbol for which no action is given, that means halt. The machine in this paper works slightly differently.

The *program* array has an action for every (state, tape-symbol) pair. There are six possible outcomes of execution.

- If an action says that the symbol to be printed is not one of the symbols on this machine, or that the next state is a number larger than any state, execution prints an error message and stops. In the example program, there are no such actions.
- If the action moves the read-write-head to the left of the start of the tape, an error message is printed, and execution stops. That does not happen when the example program is executed.
- Execution reaches a halting configuration, after which nothing more happens. In the example program, look at row 0 column 1: the action is [1; 0; 0]. That says print 1, which is the symbol "(" that was already there, so that is no change. The action then says move the head 0 squares, so that is no change. Then go into state 0, which is the state it was already in, so that is no change. There are 25 such actions, shown in red. If such an action were to be executed, there would be no visible change; the machine is effectively halted. Our machine simulator detects such configurations, and prints "halt", and ends execution. Execution of the example program never reaches a halting configuration.
- A nonprinting loop can involve more than one action, moving the read-write-head but never changing the tape symbols. There are no such loops in the example program.

- Execution continues producing more output forever. That is what happens in the example program. The output is counting in unary:
  (0 0 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0
  and so on.
- Execution continues forever, but the output on the tape reaches a finite maximum length (to the right of which the tape remains blank forever), and all further output makes changes to the already written portion of the tape. This does not happen in the example program.

The example program is reasonably faithful to Turing's presentation, with the exceptions as noted. We begin with a blank tape, as does Turing, and use program states 0, 1, and 2 to create the tape we want before entering a loop at state 3.

I now simplify the machine by initializing the tape and read-write-head position and state to the desired configuration, rather than creating the desired configuration in the program, saving 3 states. The result is shown below. There are still 13 halting actions that are never performed in the simplified machine; they are shown in red.

new *symbols*:= 0,..5. **new** *states*:= 0,..5. `the number of symbols and states must be finite
new *tape*: ∞**symbols*:= 1; 3; 0; 3; ∞*0. `initially "(0 0" followed by blank spaces
new *position*: *nat*:= 1. `the position of the read-write-head, initially 1
new *state*: *states*:= 0. `the current state, initially 0
new *time*: *nat*:= 0. `time measured as the number of actions performed
new *program*:= `the state transition matrix; this is the TM program

`	0 " "	1 "("	2 ")"	3 "0"	4 "1"		tape symbols
[[	[0; 0; 0];	[1; 0; 0];	[2; 0; 0];	[3; 0; 2];	[4; 1; 1]	];	`state 0
[	[2; –3; 0];	[1; 0; 1];	[2; 0; 1];	[3; 0; 1];	[4; 0; 1]	];	`state 1
[	[4; –1; 3];	[1; 0; 2];	[2; 0; 2];	[3; 2; 2];	[4; 2; 2]	];	`state 2
[	[0; –2; 3];	[1; 1; 4];	[0; 1; 2];	[3; 0; 3];	[4; 0; 3]	];	`state 3
[	[3; –2; 0];	[1; 0; 4];	[2; 0; 4];	[3; 2; 4];	[4; 2; 4]	]]	`state 4

Analysis

The problem is to write a program to analyze Turing Machine programs to see which of the six outcomes will be the result of execution. The program we write cannot be a Turing Machine program; that was proven impossible by Turing [5]. But it may be possible to write the program in another language (see [1]); I shall use ProTem [0]. If this ProTem program is written, it will not be possible to translate it into the Turing Machine language. (For examples of programs that cannot be translated from one Turing-Machine-equivalent language to another, see [2].)

I simplify my task by assuming the tape always starts blank, the read-write-head position always starts at 0, and the starting state is always 0, as in the first example machine before I simplified it. I also assume that the next state is never 0 (except for halting configurations on row 0, which are unreachable). These assumptions do not restrict the class of Turing Machine programs that can be written. Also, the analysis will determine whether the outcome is halting, or not (grouping the other five outcomes together, as in the original Halting Problem).

———————————UNFINISHED
Forward simulation: need to recognize when execution will not terminate.
Backward simulation: starting with each halting configuration, find all preceding configurations, repeatedly, and see if the starting configuration is among them.

The current configuration is *tape* and *position* and *state* . A preceding configuration is *pretape* and *preposition* and *prestate* if there is a row *r* and column *c* such that

*tape_*(*position* – *program r c* 1) = *program r c* 0
state = *program r c* 2

and the preceding configuration is

pretape = *tape_*(0;..*preposition*); *c*; *tape_*(*preposition*+1;..∞)
preposition = *position* – *program r c* 1
prestate = *r*

The number of nonblank symbols on the tape is at most the time *t* .
Look at *t*=0 , *t*=1 , and so on.
Suppose execution terminates; then for some *t* the search will find a starting state.
Suppose execution does not terminate; then, for each *t* , the search will find that there isn't a starting state. When can we stop increasing *t* ?
I believe that all well-specified problems are computable. But this problem may not be well-specified due to the lack of time bound on termination (see [3]).
— — — — — — — — — — —

References

[0] E.C.R.Hehner: ProTem: a Programming System, hehner.ca/PT.pdf, 2023
[1] E.C.R.Hehner: several papers on halting, hehner.ca/halting.html, 2013 to 2022
[2] E.C.R.Hehner: Objective and Subjective Specifications, WST Workshop on Termination, Oxford, hehner.ca/OSS.pdf, 2018
[3] E.C.R.Hehner: Observations on the Halting Problem, hehner.ca/OHP.pdf, 2014
[4] D.Otto: Turing: a Turing Machine Language, https://docs.racket-lang.org/turing/index.html, date unknown
[5] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* s.2 v.42 p.230-265, 1936; correction s.2 v.43 p.544-546, 1937

other papers on halting