Alana – A Turing Machine Simulator

Markus Triska (0225855)[1]

December 8, 2003

Abstract

This paper presents *Alana*[2] – a program simulating the execution of any standard Turing machine, encoded in a suitable description language. It also provides a short overview of the theory behind Turing machines.

1 Theoretical discussion

1.1 Definition of a Turing machine

A Turing machine (or TM for short) is an abstract device named after its designer, Alan Turing. Its components are a *storage unit*, a *control unit* and a *read-write head*. We can think of the storage unit as a one-dimensional array of cells that extends indefinitely in both directions. At any time, each cell can hold exactly one symbol. In analogy to magnetic tapes used in typewriters and computers, the storage device is also called a *tape*, and the read-write head is also called *tape head*. The tape head can move left or right on the tape. In each move, it can read and write the symbol of the cell that it is placed on. Figure 1 schematically depicts the device, which we shall call a *standard* TM.

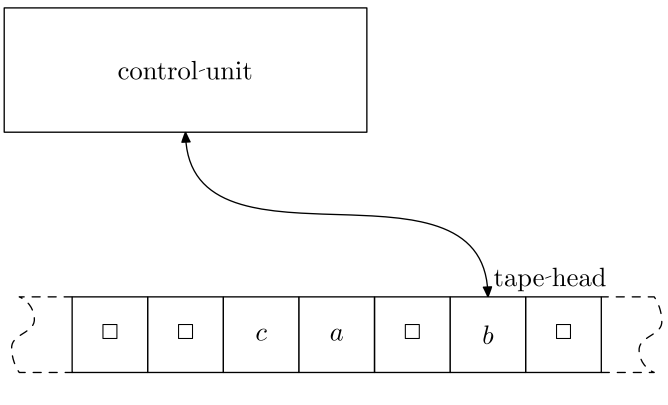


Figure 1: Schematic depiction of a Turing machine

[1]This project was done as "Projektpraktikum (mit Bakk.arbeit)", advisor G. Salzer
[2]*Alana* stands for "**A**lana **l**acks **a** **n**atural **a**cronym"

Formally, a Turing machine M is defined by a 7-tuple,

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

where

Q is the set of internal *states*,

Γ is the *tape alphabet* (symbols the tape head can read from and write to the tape),

Σ denotes the *input alphabet* (symbols that the tape can contain initially, it is often assumed that $\Sigma \subseteq \Gamma$),

δ is the *transition function*,

$\square \in \Gamma$ is the special *blank* symbol,

$q_0 \in Q$ is the *starting* or *initial state*,

$F \subseteq Q$ is the set of *final*, *halt* or *accepting* states.

The blank symbol ('$\square$' in this paper, 'B', 'ε' or 'λ' in some others) is special because every cell of the tape is assumed to contain this symbol if nothing else is specified. Therefore, it is usually assumed that $\square \notin \Sigma$, because one could not reliably find the end of the input otherwise – there could always be some character left on the tape, after an arbitrary number of blank symbols, and we would look for it forever if the tape is all blank. A similar restriction is necessary if we agree upon an ending delimiter for the input, because this delimiter obviously must not occur in the input in this case.

At any time, we know two things about a TM:

- the current tape content (= content of every cell of the tape), and in particular, the content of the cell under the tape head
- the current state of the machine

This is called the *configuration*, or *instantaneous description*, of the TM.

1.2 Transitions and computation

Initially, a TM is in the starting state, and any optional initial tape content is located to the right of the tape head, with the latter being positioned over the first symbol. If no initial tape content is specified, the tape head is positioned over any cell.

Given the current state and the symbol under the tape head, we can determine what to do next via the transition function δ, which is defined as

$$\delta\colon\, Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

A single transition of the form

$$\delta(q_1, a) = (q_2, b, R)$$

means that if the machine is in state q_1 and the symbol under the tape head is 'a', it will (in this order):

1. switch to state q_2,
2. write the symbol 'b' at the current position of the tape head
3. move the tape head one cell right, according to the move symbol, 'R'. The tape head would shift left if we specified 'L' as the move symbol.

The machine halts when there is no transition defined for the current configuration. The input is said to be *accepted* by the machine if the machine halts in an accepting state, *not accepted* otherwise. A sequence of transitions leading to an accepting state is called a *computation*. After the input is accepted by the TM, the final tape content is called the *result* of the computation. The process of computation is deterministic in the sense that for every possible configuration, the transition function can define at most one 3-tuple of next state, symbol and direction.

1.3 Graphical representation of Turing machines

A TM is naturally depicted as a graph, where states are represented by circles. The starting state is drawn using a thicker pen, and accepting states are indicated by double-circles.

A line going from state q_0 to state q_1 with a label of the form "$a/b/D$" represents a part of the transition function and means $\delta(q_0, a) = (q_1, b, D)$. When each transition that switches to state q moves the tape head in the same direction D, we can write D in place of the state's name and print the latter somewhere nearby the respective circle. We can thus avoid printing the direction in the label of each line that enters said state q. Labels of the (reduced) form "a/a" can then be abbreviated as 'a'.

Figure 2 gives an example of a simple TM's graph. This TM performs addition of two numbers in unary notation. This particular encoding scheme is often very useful to avoid more complicated calculations: 0 is represented by '□', 1 is represented by '1', 2 by "1 1", 3 by "1 1 1", and so on. Of course, any other symbol could be used to indicate the value of the number as well. To add two numbers specified in unary encoding, separated by '+', we simply replace '+' by '1' and delete the last '1' on the tape. After that, we rewind the tape and place the tape head on the first cell of the result.

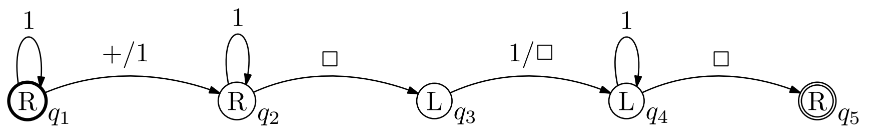


Figure 2: Adding two numbers in unary encoding

1.4 Textual notation of transitions

It is convenient to introduce a textual notation to describe instantaneous descriptions and transitions, in which

$$s_1 s_2 \cdots s_{k-1} q s_k s_{k+1} \cdots s_n$$

means that the tape content is $s_1 s_2 \cdots s_n$, the machine is in state q and the tape head is over the symbol immediately following q, in this case s_k. Figure 3 shows a graphical representation of this configuration. To avoid ambiguity, this notation requires that Q and Γ be disjoint, i. e., $Q \cap \Gamma = \emptyset$. The unspecified part of the tape is assumed to contain all blanks, which can of course also occur in the instantaneous description.

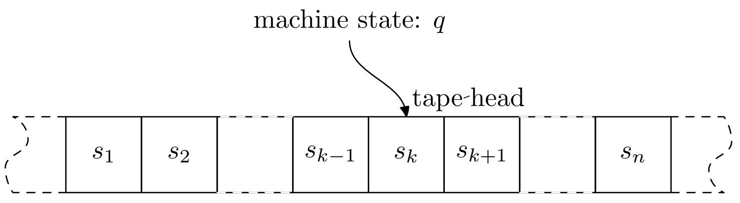


Figure 3: Graphical representation of the stated configuration

The operator $\vdash$ is used to indicate a move from one configuration to another. It can be subscripted like $\vdash_M$ to distinguish between several machines.

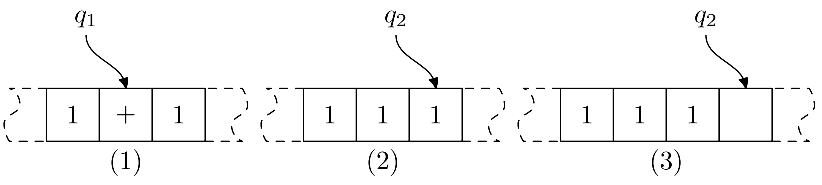


Figure 4: A sequence of transitions

The sequence of transitions depicted in Figure 4 can thus be described as

$$1q_1{+}1 \vdash 11q_2 1 \vdash 111q_2\ .$$

Incidentally, these are exactly the transitions that the TM in Figure 2 would carry out. Its next steps would be

$$111q_2\Box \vdash 11q_3 1 \vdash 1q_4 1 \vdash q_4 11 \vdash q_4\Box 11 \vdash q_5 11\ .$$

The blank symbol was explicitly stated in this case to clarify the situation. The symbol $\vdash^*$ is used to denote an arbitrary number of transitions, including 0. We could therefore also write

$$1q_1{+}1 \vdash^* q_5 11$$

for the sequence of transitions that we looked at. A transition

$$s_1 s_2 \cdots s_{k-1} q_1 s_k s_{k+1} \cdots s_n \vdash s_1 s_2 \cdots s_{k-1} t q_2 s_{k+1} \cdots s_n$$

is admissible if and only if

$$\delta(q_1, s_k) = (q_2, t, R)$$

and analogously for moving left.

1.5 Turing machines and contemporary computers

Now that we have found a TM to perform integer addition, the question arises whether other functions and algorithms can be expressed in terms of a TM. It is easy to see that we can devise TMs to perform subtraction, multiplication and comparisons, but what about more complicated functions?

Let us first define a class of functions:

> A function f with domain D is called *(Turing-)computable* if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ such that
>
> $$q_0 w \vdash_M^* q_f f(w), \quad q_0 \in Q, q_f \in F,$$
>
> for all $w \in D$.

Note that the domain D plays a critical role in this definition. For a function to be computable, there must be a TM that computes the function on the whole of its domain. If we expect only "yes" or "no" as the result of the function, we use the terms *decidable* and *undecidable* instead of *computable* and *uncomputable.*

As outlined in [Davis 2000] and [Davis et al. 1994], all the common mathematical functions, no matter how complex, are Turing-computable. In fact, Turing machines are so powerful that the definition of Turing-computable functions is widely believed broad enough to include any existing *mechanical computation*, including all calculations of a typical contemporary digital computer. This is known as the *Church-Turing-Thesis.*

Alternative models have been proposed to define "mechanical computation", but none of them are more powerful than the Turing machine model, and it is unlikely that a simpler model that is equally powerful can be found. Examples of how to translate concepts of contemporary high-level languages – such as variables, conditions, loops and function calls – to a TM can be found in [Denning et al. 1978], [Hopcroft et al. 2000] and [Linz 2001]. Using these features, one can also simulate the register architecture of today's personal computers using a TM, thus proving that a PC can not be more powerful than a TM.

1.6 Universal Turing machines

Notice that every TM is computable itself, because one can specify a transition function for some other TM that expects the specification of the initial TM and its input as its input (this is no mistake!) in some reasonable encoding and then interprets the encoded form of the initial TM to execute exactly the operations that *this* TM would have executed on the original input.

Such a TM that simulates other TMs is called a universal Turing machine or UTM. *Alana* itself is an example of a UTM, because we can give it a specification of a TM (using a somewhat reasonable encoding described below) and it will simulate the execution of this given TM. Yet, one could translate the operation of *Alana* to a TM, because the language that *Alana* is written in (Tcl/Tk) is no more powerful (in computational terms) a concept than Turing machines are. Translating *Alana* to a standard TM is cumbersome, but achievable. Have a look at [Denning et al. 1978], where an example of a UTM is presented.

1.7 Other definitions of Turing machines

There exist many other essentially equivalent definitions of Turing machines. They differ only in specification details, not in expressiveness or computation power. For example, some of them allow the tape head not only to shift right or left, but also to stay in place. This can of course be achieved in our model by introducing additional states that shift the tape head back to where it came from. Some machines use multiple tapes. This can be achieved in our model by using more than one letter per cell and interpret the n-th letter as the content of the n-th tape. For example, "gxa" could be interpreted as 'g' on tape 1, 'x' on tape 2 and 'a' on tape 3.

Also, without loss of generality, we may assume that the tape extends infinitely only to the right (or left), because we can interpret (for example) every odd-numbered cell as extending to the left and every even-numbered cell as extending to the right.

In complexity theory, *nondeterministic* TMs, where δ is a relation instead of a function, play an important role, but standard TMs can be devised to handle nondeterministic behaviour deterministically. [Linz 2001] contains a comprehensive overview of different definitions of TMs and their relations.

1.8 The halting problem

Consider the Turing machine M defined by

$$M = (\{q_0\}, \emptyset, \{\Box\}, \delta, q_0, \Box, \emptyset)$$

where $\delta(q_0, \Box) = (q_0, \Box, R)$. It is obvious that this machine, once execution starts, will never halt (much less accept any input, because there is not a single accepting state), but move to the right indefinitely. In general, however, we can not decide if a given Turing machine halts by simply looking at its definition. This was proved by Alan Turing, and the proof is given below. Because every contemporary computer program (and computer) can be reduced to a Turing machine, we can outline the proof in the more convenient notation of Pascal and C-style programs. A more mathematical proof without the need to resort to programming languages is given in [Denning et al. 1978].

Theorem 1 *The problem whether a given Turing machine (or program) P will halt on input I in finite time (the "halting problem") is undecidable.*

Proof: Let `halt(program p, input I)` be a function that expects the digital presentation `p` of a computer program and some input `i` that is given to that program as its arguments and returns (in finite time) `yes` if program `p` halts on this input in finite time, `no` otherwise.

We can use this function in any program. Let program `S` be the following program:

```
program S(program M, input N):
        if (halt(M, N) == yes) {
                infinite_loop;
        } else {
                print "That program never halts."
        }
```

Program `S` expects the digital representation of a computer program as its argument and checks if *that* program will halt on the given input `N` by calling said function `halt`.

Now since program `S` is a program like any other program that our computer can execute, it surely has a digital representation itself. The question now is what does

```
halt(S, S)
```

return? (That is program `S` with the digital representation of itself as its argument.) We know that the function `halt` can only return either `yes` or `no`.

Case 1 If `halt(S, S)` returns `yes`, it means that the call `halt(M, N)` in program `S` also returned `yes` (because its arguments were also `S` and `S`). We know that program `S` would then have branched into an infinite loop, so our assumption that `yes` is returned must be wrong and `yes` can not be returned.

Case 2 If `halt(S, S)` returns `no`, it means that the call `halt(M, N)` in programs `S` also returned `no`. Since programs `S` would simply print out "`That program never halts`" and exit in this case, the call to `halt(S, S)` cannot possibly return `no`, because we know that program `S` halts nearly immediately in this case.

Since neither return value can be returned if function `halt` works correctly, we conclude that our assumption must be wrong and such a function cannot exist. For an overview of other undecidable problems, consult [Hopcroft et al. 2000].

1.9 Busy Beavers

Related to the halting problem is the quest for *Busy Beavers*. The problem specification is this:

> Given a fixed number n of states, a blank tape, and a finite tape alphabet Γ, construct a transition function δ so that the resulting Turing machine writes the maximum number of '1's on the tape and then halts.

You can freely choose the starting state and the set of accepting states. It is usually implicitly assumed that $\Gamma = \{\Box, 1\}$.

The crux of the problem lies in the last requirement, namely that the machine has to halt! It is of course possible to examine every Turing machine with n states that could be constructed via arbitrary transition functions over the alphabet Γ (because it is finite), but the number of possible machines grows rapidly with the number n of states, and for every candidate machine, it cannot be seen beforehand whether it will halt eventually or simply loop forever.

Due to the undecidability of the halting problem, the Busy Beaver function $B(n)$, returning the number of '1's written on the tape by an n-state Busy Beaver, is uncomputable. Therefore the predicate "an n-state Busy Beaver" is sometimes incorrectly used as a synonym for "a (former) candidate for an n-state Busy Beaver". Figure 5 shows a candidate for a 5-state Busy Beaver, not counting the accepting state. Amazingly, this machine – when started on a blank tape in state q_0 – halts with 4,098 '1's written on the tape after 47,176,870 transitions.

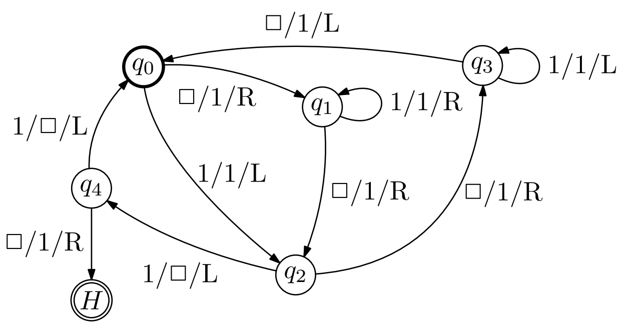


Figure 5: A candidate for a 5-state Busy Beaver

2 The program *Alana*

2.1 Goals of the project

The main goal during the design and development stages of *Alana* was *usability*, that is ease and comfort to use and work with Alana. Although there do exist other Turing machine simulators on the Internet, they are difficult to use at best. Most of them have serious or highly impractical limitations: [TM 1] only provides 2,001 tape cells to work with, [TM 2] lets you only use a very narrow range of tape symbols, [TM 3] works only with 3 preloaded examples, [TM 4] requires a complicated and redundant input format, and [TM 5] is only available for a fee. None were found that display the next step of the computation, or provide back stepping functionality.

2.2 The user interface

Figure 6 shows a screen-shot of *Alana* that helps to explain the layout of the program's major components. The reader will immediately recognize a few cells of the tape, buttons to move the tape head, and a text entry box to define transitions. Also, a menu button is provided to change the current state of the machine on-the-fly, and there are buttons to make the machine perform the next step of its computation, which is always displayed beforehand in the bottom right corner of the main window. Alternatively, a user can advise the program to enter *run-mode*, i.e., to execute step after step without the need to click on the button that would normally trigger this action. The scrollbar at the bottom is used to adjust the speed of the tape head.

2.3 Syntax of *Alana*

To specify the transition

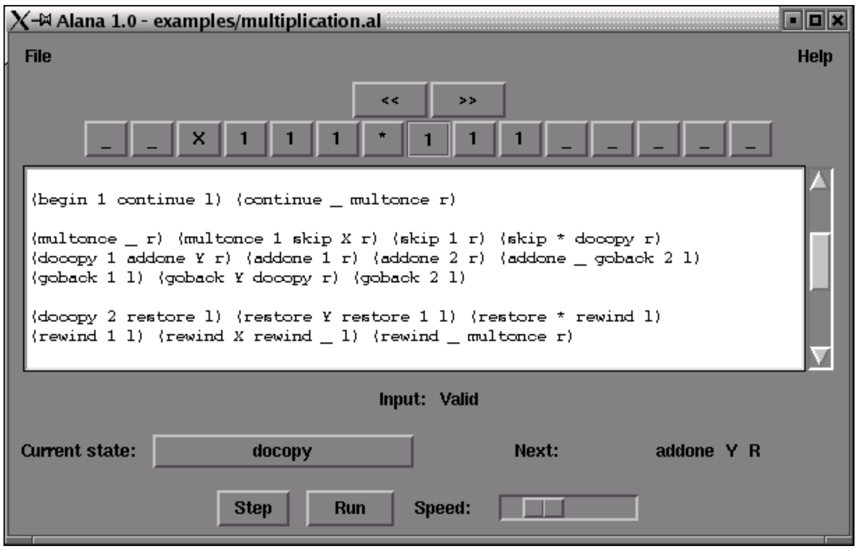
$$\delta(q_1, a) = (q_2, b, R)$$

in *Alana*'s syntax, enter

$$(q_1\ a\ q_2\ b\ R)$$

in the transition entry box. For transitions of the form

$$\delta(q_1, a) = (q_2, a, R)$$


Alana 1.0 - examples/multiplication.al
File
Help
<<
>>
_ _ X 1 1 1 * 1 1 1 _ _ _ _ _
Step
Run
Speed:

```
(begin 1 continue l) (continue _ multonce r)

(multonce _ r) (multonce 1 skip X r) (skip 1 r) (skip * docopy r)
(docopy 1 addone Y r) (addone 1 r) (addone 2 r) (addone _ goback 2 l)
(goback 1 l) (goback Y docopy r) (goback 2 l)

(docopy 2 restore l) (restore Y restore 1 l) (restore * rewind l)
(rewind 1 l) (rewind X rewind _ l) (rewind _ multonce r)
```

Input: Valid

Current state: docopy   Next: addone Y R

Figure 6: A screen-shot of *Alana*

the shorter form

$$(q_1\ a\ q_2\ R)$$

can be used. For transitions of the form

$$\delta(q_1, a) = (q_1, a, R)$$

you can use the even shorter version

$$(q_1\ a\ R)\ ,$$

but you can of course also use the longer versions $(q_1\ a\ q_1\ R)$ or $(q_1\ a\ q_1\ a\ R)$ if you want.

To specify a set of accepting states, enter them delimited by curly braces, for example:

$$\{q_5\ q_0\ q_1\}$$

The starting state is delimited by brackets, like $[q_3]$. The blank symbol is denoted by '_' (underline). Every string of letters and digits not enclosed by parentheses, braces or brackets is written directly onto the tape, followed by a right shift of the tape head. After the input is read in completely, the tape head is positioned on the first non-blank symbol on the tape if some initial tape content was specified, and otherwise on any blank symbol. For example, the 5-state Busy Beaver candidate of Figure 5 was encoded for *Alana* as

```
(q0 _ q1 1 R) (q0 1 q2 L) (q1 _ q2 1 R) (q1 1 R)
(q2 _ q3 1 R) (q2 1 q4 _ L) (q3 _ q0 1 L) (q3 1 L)
(q4 _ H 1 R) (q4 1 q0 _ L) {H} [q0]
```

There is no need to specify the set Q of states or the tape alphabet Γ, because these can both be deduced from the transition function. The blank symbol can occur in the input and will be written on the tape just like every other symbol.

2.4 Included examples

Alana is distributed with a set of examples:

a2n.al a TM accepting only strings of the form a^{2^n}, $n \geq 0$

anbncn.al a TM accepting only strings of the form $a^n b^n c^n$, $n \geq 0$

binaryadd.al addition of two numbers in binary encoding

busy5.al a former candidate for a 5-state Busy Beaver writing 501 '1's and halting after 134,467 steps

copyinput.al copy a given string of '0's and '1's to the tape

divisibility.al divisibility testing of two numbers in unary encoding

multiplication.al multiplication of two numbers in unary encoding

primality.al test if a given number (in unary encoding) is prime

substract.al a TM subtracting two numbers in unary encoding

unaryadd.al a very simple example performing addition in unary encoding

3 Bibliography and resources

[**Davis 2000**] Martin Davis, *The Universal Computer: The Road from Leibniz to Turing*, W. W. Norton & Company, 2000

[**Davis et al. 1994**] Martin D. Davis, Ron Sigal, Elaine J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, 2nd edition, 1994

[**Denning et al. 1978**] Peter J. Denning, Jack B. Dennis, and Joseph E. Qualitz, *Machines, Languages, and Computation*, Prentice Hall, 1978

[**Hopcroft et al. 2000**] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing, 2nd edition, 2000

[**Linz 2001**] Peter Linz, *An introduction to formal languages and automata*, Jones and Bartlett Publishers, 3rd edition, 2001

[**TM 1**] *http://userpages.wittenberg.edu/bshelburne/Turing.htm*

[**TM 2**] *http://math.hws.edu/TMCM/java/labs/xTuringMachineLab.html*

[**TM 3**] *http://www.turing.org.uk/turing/scrapbook/tmjava.html*

[**TM 4**] *http://www.cs.binghamton.edu/˜software/tm/tmdoc.html*

[**TM 5**] *http://www-csli.stanford.edu/hp/Version-turing-mac.html*

This document was typeset in LaTeX. Figures 1, 3 and 4 were created using MetaPost. Figures 2 and 5 were created using *Finomaton*. *Alana* can be obtained from

http://stud4.tuwien.ac.at/˜e0225855/alana/alana.html.