University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Turing Machine Language

Pete Gautam

October 26, 2022

Abstract

Acknowledgements

Education Use Consent

Contents

1 Introduction

2 Background

In this chapter, we consider the Turing Machine Language (TML) as an alternative to Turing Machines (TM). A TM can be executed on a tape, one step at a time, until we reach a terminating state on the TM. We will define the syntax of a TML program and how it can be executed on a tape in a similar manner. We will then prove that there is an equivalence between TML programs and TMs in terms of their execution on a tape.

2.1 Turing Machines

In this section, we review the definition of Turing Machines and executing a TM on a valid tape.

A *Turing Machine* (TM) is a collection (Q, Σ, δ, q_0), where:

- Q is a set of states, including the accept state q_Y and reject state q_N;
- Σ is the set of letters, which does not include the `blank` symbol;
- $\delta\colon Q \setminus \{q_Y, q_N\} \times \Sigma^+ \to Q \times \Sigma^+ \times \{\texttt{left}, \texttt{right}\}$, where $\Sigma^+ = \Sigma \cup \{\texttt{blank}\}$, is the transition function; and
- $q_0 \in Q$ is the starting state.

We can represent a TM as a directed graph, with vertices as states and edges as transitions. For example, the following is a TM:

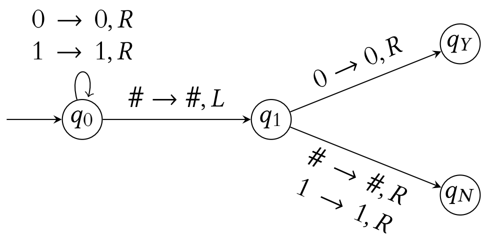


Figure 2.1: *A Turing Machine that accepts binary numbers divisible by 2.*

In this case, the alphabet $\Sigma = \{0, 1\}$. The blank symbol is denoted by #. The initial state is denoted by q_0; the accept state q_Y and the reject state q_N. Every edge corresponds to an evaluation of the transition function δ, e.g. $\delta(q_0, a) = (q_1, \texttt{blank}, \texttt{right})$.

Let Σ be an alphabet. A *tape T on Σ* is a function $T\colon \mathbb{Z} \to \Sigma^+$. In particular, the tape has infinite entries in both directions. Moreover, T is a *valid tape* if only finitely many symbols on T are not `blank`, and all the values that can be non-`blank` are non-`blank`. That is, there exist integers a, b such that for all $x \in \mathbb{Z}$, $T(x)$ is not `blank` if and only $x \geq a$ and $x \leq b$. We will represent a tape using a figure. For instance, let $\Sigma = \{0, 1\}$, and let T be the tape on Σ given below:

$$T(x) = \begin{cases} 0 & x \in \{0, 2, 3\} \\ 1 & x \in \{1\} \\ \texttt{blank} & \text{otherwise.} \end{cases}$$

Then, the following figure represents the tape T:

$$\underline{}\ \ \underline{\texttt{0}}\ \ \underline{\texttt{1}}\ \ \underline{\texttt{0}}\ \ \underline{\texttt{0}}\ \ \underline{}$$

We will assume that the first non-blank value is at index 0.

A TM can be executed on a tape. Let M be a TM with alphabet Σ, and let T be a (valid) tape on Σ. We execute M on T inductively, as follows:

- At any point during execution, we maintain 3 objects- a tape on Σ, a state in M and an index in the tape (called the *tapehead index*).
- At the start, the tape is T; the tapehead index is 0; and the state is the initial state q_0.
- At some point during the execution, assume that we have the tape S, tapehead index j, with *tapehead value* $T(j) = t$, and a non-terminating state q (i.e. not q_Y or q_N). Denote $\delta(q, t) = (q', t', \texttt{dir})$. Then, the next state is q', and the next tape S' and the next tapehead index j' are given by:

  $$S'(x) = \begin{cases} t' & x = i \\ S(x) & \text{otherwise,} \end{cases} \qquad j' = \begin{cases} j+1 & \texttt{dir} = \texttt{right} \\ j-1 & \texttt{dir} = \texttt{left}. \end{cases}$$

  If the state q' is not a terminating state, then the execution continues with these 3 objects. Otherwise, execution is terminated with terminating state q'.

2.2 Turing Machine Language Program

In this section, we will define the syntax of the Turing Machine Language (TML). We will first give a program in TML, and then analyse the syntax. We will then define execution of a valid TML program on a tape in a similar manner to the execution of a TM.

Consider the following TML program.

```
alphabet = {"0", "1"}
module isEven {
   switch tapehead {
      while 0, 1 {
         move right
      } if blank {
         move left
         switch tapehead {
            if 0 {
               changeto blank
               accept
            } if 1, blank {
               changeto blank
               reject
            }
         }
      }
   }
}
```

A program in TML will be used to execute on a tape, so the syntax used guides us in executing the program on a tape. We will see that later. For now, we consider the rules of the TML program:

- A valid TML *program* is composed of the *alphabet*, followed by one or more *modules*. In the example above, the alphabet of the program is $\{0, 1\}$, and the program has a single module called `isEven`.
- A module contains one or more *blocks* (a specific sequence of commands). There are two types of blocks- *basic blocks* and *switch blocks*.
- A basic block consists of *basic commands* (*changeto*, *move* or *flow* command). A basic block consists of at least one basic command, but it is not necessary for a basic block to be composed of all the basic commands. If multiple commands are present in a basic block, they must be in the following order- *changeto*, *move* and *flow* command. In the program above, there is are many basic blocks, e.g. at lines 5, 7, 10-11 and 13-14. We do not say that line 10 is a basic block by itself; we want the basic block to be as long as possible.
- A *switch block* consists of a single switch statement. A switch statement must contain precisely one case (*if* or *while* command) for each of the letter in the alphabet, including the `blank` letter. The first block of a case block must be a basic block. In the program above, there is a switch block at lines 8-16.
- The body of an *if* command can be composed of multiple blocks. These blocks can be both basic blocks and switch blocks. We can see this at lines 6-17; the *if* block has a basic block at line 7 and then a switch block.
- The body of a *while* command must be composed of a single basic block. The basic block cannot have a *flow* command. This is because when we execute a *while* block, the next block to run is the switch block it is in; we cannot accept, reject or go to another module.
- A switch block must be the final block present; it cannot be followed by any other block, basic or switch.

The EBNF for the TML is given in the appendix.

We will now consider how to execute a tape on a valid TML program. Let P be a TML program with alphabet Σ and let T be a tape on Σ. We execute P on T inductively, as follows:

- At any point during execution, we maintain 3 objects- a tape on Σ, a block of P and the tapehead index.
- At the start, the tape is T; the tapehead index is 0; and the block is the first block in the first module in P.
- At some point during the execution, assume that we have the tape S, tapehead index j, with tapehead value $T(j) = t$, and a block b. We define the next triple as follows:
  - if b is a switch block, we take the first block from the case corresponding to the tapehead value- because the program is valid, this is a basic block; we will now refer to this block as b.
  - if b has a *changeto* `val` command, the next tape T' is given by

    $$T'(x) = \begin{cases} \texttt{val} & x = i \\ T(x) & \text{otherwise.} \end{cases}$$

    If the *changeto* command is missing, then the tapehead $T' = T$.
  - if b has a *move* `dir` command, the next tapehead index is given by:

    $$i' = \begin{cases} i + 1 & \texttt{dir} = \texttt{right} \\ i - 1 & \texttt{dir} = \texttt{left}. \end{cases}$$

    If the *move* command is missing, then $i' = i - 1$.
  - we either terminate or determine the next block b' to execute (in decreasing precedence):

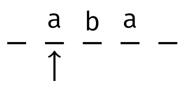
- if the block is the body of a while case block, then the next block $b' = b$, i.e. we execute this switch block again (not necessarily the same case block);
- if the block contains a terminating *flow* command, execution is terminated and we return the terminated state (`accept` or `reject`);
- if the block contains a *goto* `mod` command, then b' is the first block of the module `mod`;
- if the block is not the final block in the current module, then b' is next block in this module;
- otherwise, execution is terminated and we return the state `reject`.

If execution is not terminated, execution continues with the next triplet.

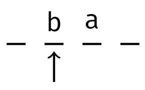
We will now illustrate the execution with an example. Consider the following TML program:

```
alphabet = {"a", "b"}
module palindrome {
   switch tapehead {
      if blank {
         accept
      } if a {
         changeto blank
         move right
         switch tapehead {
            while a, b {
               move right
            } if blank {
               move left
               switch tapehead {
                  if blank, a {
                     changeto blank
                     move left
                     goto restart
                  } if b {
                     reject
                  }
               }
            }
         }
      } if b {
         changeto blank
         move right
         switch tapehead {
            while a, b {
               move right
            } if blank {
               move left
               switch tapehead {
                  if blank, b {
                     changeto blank
                     move left
                     goto restart
                  } if a {
                     reject
                  }
               }
            }
         }
44         }
45      }
46   }
47   module restart {
48      switch tapehead {
49         while a, b {
50            move left
51         } if blank {
52            move right
53            goto palindrome
54         }
55      }
56   }
```

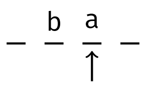
We will execute the program on the following tape.



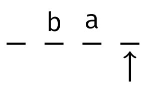
The arrow points to the tapehead value. We first execute the switch block at `palindrome`. Since the tapehead value is `a`, we execute the basic block at lines 7–8. So, we change the tapehead value to `blank`, and the tapehead moves to the right by one step. Since this is an *if*-block, without a flow command, and there is a block following this one, the next block to be executed is the switch block at lines 9–24. Now, the current tape is the following.



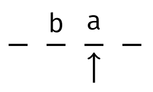
The current block is a switch block. The tapehead value is `b`, so we are at the *while* command at line 11. The basic block here only contains a *move* command. So, we leave the tape as is, and the tapehead moves to the right once. This is a *while* command, so the next block to execute is still this switch block. The current tape state is the following.



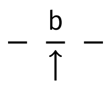
The current block is still a switch block. The tapehead value is `a`, so we execute the same *while* command at line 11. Moreover, the next block to execute is still the switch block. Now, the current tape state is the following.



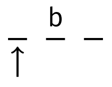
For the third time, we are executing the same switch block. Now, however, the tapehead value is `blank`, so we execute the first block of the *if* command at line 13. Since this is an *if* command and this is not the last block in the if command, the next block to execute is the switch block at lines 14–22.



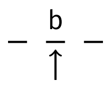
Now, the current block is a switch block. The tapehead value is `a`, so we take the basic block at lines 16-18. The value of the tapehead becomes `blank`, and the tapehead moves to the left. The next block to execute is the switch block in `restart`.



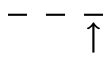
Since the current tapehead value is `b`, we execute the basic block at line 50. So, we move to the left, and the tape is left as is. Moreover, since this is a while block, the next block to execute is still the switch block.



Since the current tapehead state is `blank`, we execute the basic block at lines 52-53. So, we move to the right, and the tape is left as is. The next block to execute is the switch block at `palindrome`.



Since the current tapehead state is `b`, we execute the basic block at lines 26-27. So, we change the tapehead value to `blank`, move to the right. This is a while block, so the next block to be executed is still the switch block.



At this point, the tapehead index moves between the blank values as we move to the basic block at line 5. Then, the execution terminates and we accept the tape.

2.3 Complete TML Programs

When we defined execution of a valid TML program on a tape above, we said that a basic block need not have all 3 types of commands (*changeto*, *move* and a *flow* command), but in the execution above, we have established some 'default' ways in which a program gets executed. In particular,

- if the *changeto* command is missing, we do not change the value of the tape;
- if the *move* command is missing, we move left;
- if the *flow* command is missing, we can establish what to do using the rules described above- this is a bit more complicated than the two commands above.

Nonetheless, it is possible to include these 'default' commands to give a *complete* version of the program. This is what we will now establish.

Consider the following complete program.

```
alphabet = {"0", "1"}
module isOdd {
   // move to the end
   switch tapehead {
      while 0 {
         changeto 0
         move right
      } while 1 {
         changeto 1
         move right
      } if blank {
         changeto blank
         move left
         goto isOddCheck
      }
   }
}
module isOddCheck {
   // accept if and only if the value is 1
   switch tapehead {
      if 0, blank {
         changeto 0
         move left
         reject
      } if 1 {
         changeto blank
         move left
         accept
      }
   }
}
```

Now, we consider the rules that a complete TML program obeys:

- A basic block in a complete program has all the necessary commands- if the basic block is inside *while* case, it has a *changeto* command and a *move* command; otherwise, it also has a *flow* command.
- A module in a complete program is composed of a single switch block.

We will now construct a complete TML program for a valid TML program.

1. We first break each module into smaller modules so that every module has just one basic/switch block- we add a *goto* command to the next module if it appeared just below this block.
2. Then, we can convert each basic block to a switch block by just adding a single case that applies to each letter in the alphabet.
3. Finally, we add the default values to each basic block to get a complete TML program.

This way, we can associate every block in the valid program with a corresponding block in the complete program. The complete version is always a switch block and might have more commands than the original block, but it still has all the commands present in the original block.

We will illustrate the process with an example. Assume we first have the following program.

```
alphabet = {"a", "b"}
module simpleProgram {
   changeto b
   move left
   move right
   accept
}
```

After applying step 1 of completion, we get the following program.

```
alphabet = {"a", "b"}
module simple1 {
   changeto b
   move left
   goto simple2
}
module simple2 {
   move right
   accept
}
```

After applying step 2, we have the following program.

```
alphabet = {"a", "b"}
module simple1 {
   switch tapehead {
      if a, b, blank {
         changeto b
         move left
         goto simple2
      }
   }
}
module simple2 {
   switch tapehead {
      if a, b, blank {
         move right
         accept
      }
   }
}
```

Finally, after applying step 3, we get the following program:

```
alphabet = {"a", "b"}
module simple1 {
   switch tapehead {
      if a {
         changeto b
         move left
         goto simple2
      } if b {
         changeto b
         move left
         goto simple2
      } if blank {
         changeto b
         move left
         goto simple2
      }
   }
}
module simple2 {
   switch tapehead {
      if a {
         changeto a
         move right
         accept
      } if b {
         changeto b
         move right
         accept
      } if blank {
         move right
         accept
      }
   }
}
```

This program obeys the definition of a complete program.

Theorem 1. *Let P be a valid TM program. Then, P and its completion P^+ execute on every valid tape T in the same way.*

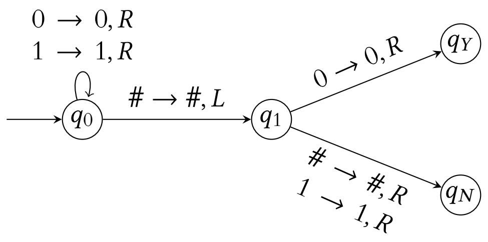
The proof of the theorem is given in the appendix.

Because of the equivalence between valid and complete programs, we will assume that every valid program is complete from now on.

2.4 Equivalence of TMs and TMLs

In this section, we will show that there is an equivalence between TMs and valid (complete) TML programs. We will first construct a valid TML program for a TM and then show that it has the same behaviour as the TM. Later, we will construct a TM for a complete TML program, and show the equivalence in this case as well.

We will first illustrate how to convert a TM to a (complete) TML program. So, consider the following TM: Consider the following TM:



Then, its corresponding TML program is the following:

```
alphabet = {"0", "1"}
module q0 {
   switch tapehead {
      while 0 {
         changeto 0
         move right
      } while 1 {
         changeto 1
         move right
      } if blank {
         changeto blank
         move left
         goto q1
      }
   }
}
module q1 {
   switch tapehead {
      if 0 {
         changeto 0
         move right
         accept
      } if 1 {
         changeto 1
         move right
         reject
      } if blank {
         changeto blank
         move right
         reject
      }
   }
}
```

In general, we convert each (non-terminating) state in the TM M to a TML module. The following is how we create the module:

- the module contains a single *switch* command;
- for each letter σ in the alphabet Σ^{+}, denote $\delta(q, \sigma) = (q', \sigma', \texttt{dir})$. We add a case in the *switch* command corresponding to letter σ (an *if* case if $q' \neq q$, otherwise a *while* case) with the following commands:
  - `changeto` σ'
  - `move` *dir*
  - in the case of an *if* block, if q' is `accept`, then the command `accept`; if q' is `reject`, then the command `reject`; otherwise, `goto` q'.

Moreover, we can construct the program P with:

- the alphabet Σ;
- modules corresponding to every state q in M;
- the module corresponding to the initial state q_0 placed at the top.

We say that P is *the corresponding program for M*.

Theorem 2. *Let M be a TM, and let P be the corresponding program for M. Then, M and P execute on every valid tape T in the same way.*

The proof of the theorem is given in the appendix.

Next, we construct a TM for a (complete) TML program. First, we illustrate with an example. So, consider the following complete TM program:

```
alphabet = {"a", "b"}
module moveToEnd {
   switch tapehead {
      while a {
         changeto a
         move right
      } while b {
         changeto b
         move right
      } if blank {
         changeto blank
         move left
         goto checkAFirst
      }
   }
}
module checkAFirst {
   switch tapehead {
      if a {
         changeto blank
         move left
         goto checkASecond
      } if b, blank {
         changeto blank
         move left
         reject
      }
   }
}
module checkASecond {
   switch tapehead {
      if a {
         changeto blank
         move left
         accept
      } if b, blank {
         changeto blank
         move left
         reject
      }
   }
}
```

Then, its corresponding TM is the following:

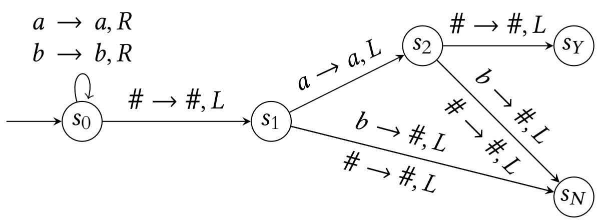


Figure 2.2: *The TM corresponding to the program above. The state s_0 corresponds to the module* `moveToEnd`*; the state s_1 corresponds to the module* `checkAFirst`*; and the state s_2 corresponds to the module* `checkASecond`*.*

In general, for each module m in P, we construct the state q as follows- for each letter σ in Σ^+, we define $\delta(q, \sigma) = (q', \sigma', \texttt{dir})$, where:

- the value σ' is the letter given in the *changeto* command within m;
- the value `dir` is the direction given in the *move* command within m;
- if the *flow* command in m corresponding to σ is `accept`, then q' is the `accept` state; if it is `reject`, then q' is the `reject` state; if we are in a *while* block, then $q' = q$; otherwise, q' is the state corresponding to the module given in the *goto* command.

Then, the TM with all the states q, the same alphabet Σ, the transition function δ and initial state q_0 corresponding to the first module in P is the *corresponding TM for P*.

Theorem 3. *Let P be a complete TM program, and let M be the corresponding TM for P. Then, P and M execute on every valid tape T in the same way.*

The proof of the theorem is given in the appendix.

Hence, we have established that for any valid TML program, there is a TM, and vice versa.

3 Requirements

4 | Design

5 Implementation

6 Evaluation

7 Conclusion

A | TML EBNF

The following is the EBNF for the Turing Machine Language.

program = *alphabet* *module*$^+$

alphabet = `alphabet = {` *seq-val* `}`

module = `module` *id* `{` *block*$^+$ `}`

block = *basic-block* | *switch-block*

switch-block = `switch tapehead {` *case-block*$^+$ `}`

case-block = *if-block* | *while-block*

if-block = `if` *seq-val* `{`*block*$^+$`}`

while-block = `while` *seq-val* `{` *core-com*$^+$ `}`

basic-block = (*core-com* | *flow-com*)$^+$

core-com = `move` *direction* | `changeto` *value*

flow-com = `goto` *id* | *terminate*

terminate = `reject` | `accept`

direction = `left` | `right`

seq-val = (*value*`,`)* *value*

value = `blank` | `a` | `b` | `c` | ... | `z` | `0` | `1` | ... | `9`

id = (`a` | `b` | `c` | ... | `z` | `A` | `B` | `C` | ... | `Z`)$^+$

B Proofs of the Theorems

Theorem 1. *Let P be a valid TML program. Then, P and its completion P^+ execute on every valid tape T in the same way. That is,*

- *for every valid index n, if we have tape T_n, tapehead index i_n and module m_n with executing block b_n for the TM program P, and we have tape S_n, tapehead index j_n and module t_n, then $T_n = S_n$, $i_n = j_n$, and t_n is the corresponding complete module block of b_n;*
- *P terminates execution on T if and only if P^+ terminates execution on T, with the same final status (`accept` or `reject`).*

Proof. We prove this by induction on the execution step (of the tape).

- At the start, we have the same tape T for both P and P^+, with tapehead index 0. Moreover, the corresponding (completed) module of the first block in the first module of P is the first module of P. So, the result is true if $n = 0$.
- Now, assume that the result is true for some integer n, where the block b_n in the TML program P does not end with a terminating *flow* command. Let σ_n be the letter at index $i_n = j_n$ on the tape $S_n = T_n$.
  - If the *changeto* command is missing in b_n for σ_n, then the next tape $T_{n+1} = T_n$. In the complete module m_n, the case for σ_n will have the command `changeto` σ_n. So, the next tape is given by:
    $$S_{n+1}(x) = \begin{cases} S_n(x) & x \neq j_n \\ \sigma_n & \text{otherwise} \end{cases}.$$
    Therefore, we have $S_{n+1} = S_n$ as well. So, $T_{n+1} = S_{n+1}$. Otherwise, we have the same *changeto* command in the two blocks, in which case $T_{n+1} = S_{n+1}$ as well.
  - If the *move* command is missing in b_n for σ_n, then the next tapehead index $i_{n+1} = i_n - 1$. In the complete module m_n, the case for σ_n will have the command `move left`, so we also have $j_{n+1} = j_n - 1$. Applying the inductive hypothesis, we have $i_{n+1} = j_{n+1}$. Otherwise, we have the same *move* command, meaning that $i_{n+1} = j_{n+1}$ as well.
  - We now consider the next block b_{n+1}:
    - ⋆ If the block b_n is a *switch* block with a *while* case for σ_n, then this is still true in the module m_n. So, the next block to be executed in P is b_n, and the next module to be executed in P^+ is m_n. In that case, the corresponding module of the block $b_{n+1} = b_n$ is still $m_{n+1} = m_n$.
    - ⋆ Instead, if the block b_n has no *flow* command for σ_n, and is not the last block, then the next block to execute is the block just below b_n, referred as b_{n+1}. By the definition of P^+, we find that the case block in the module m_n has a *goto* command, going to the module m_{n+1} which corresponds to the block b_{n+1}.
    - ⋆ Now, if the *flow* command is missing for σ_n and this is the last block, then execution is terminated with the status `reject` for the program P. In that case, the case for σ_n in the module m_n has the `reject` command present, so the same happens for P^+ as well.

- ⋆ Otherwise, both P and P^+ have the same flow command, meaning that there is either correspondence between the next module to be executed, or both the program terminate with the same status.

In that case, P and P^+ execute on T the same way by induction. □

Theorem 2. *Let M be a TM, and let P be the corresponding program for M. Then, M and P execute on every valid tape T in the same way. That is,*

- *for every valid index n, if we have tape T_n, tapehead index i_n and module m_n for the TM program P, and we have tape S_n, tapehead index j_n and state q_n for the TM M, then $T_n = S_n$, $i_n = j_n$ and m_n is the corresponding module for q_n;*
- *M terminates execution on T if and only if P terminates execution on T, with the same final status (`accept` or `reject`).*

Proof. We prove this by induction on the execution step.

- At the start, we have the same tape T for both M and P, with tapehead index 0. Moreover, the first module in P corresponds to the initial state q_0. So, the result is true if $n = 0$.
- Now, assume that the result is true for some integer n, where the TM state q_n is not `accept` or `reject`. In that case, $T_n = S_n$, $i_n = j_n$ and m_n is the corresponding module for q_n. Let σ_n be the letter at index $i_n = j_n$ on the tape $T_n = S_n$. Denote $q(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \texttt{dir})$. In that case,

$$T_{n+1}(x) = \begin{cases} T_n(x) & x \neq i_n \\ \sigma_{n+1} & \text{otherwise,} \end{cases} \qquad i_{n+1} = \begin{cases} i_n - 1 & \texttt{dir} = \texttt{left} \\ i_n + 1 & \texttt{dir} = \texttt{right}, \end{cases}$$

  and the next state is q_{n+1}.

  - We know that the module m_n in TM program P corresponds to the state q_n, so it has a `changeto` σ_{n+1} command for the case σ_n. In the case, the next tape for P is:

$$S_{n+1}(x) = \begin{cases} S_n(x) & x \neq i_n \\ \sigma_{n+1} & \text{otherwise.} \end{cases}$$

    So, $T_{n+1} = S_{n+1}$.
  - Similarly, the case also contains a `move dir` command. This implies that the next tapehead index for P is:

$$j_{n+1} = \begin{cases} j_n - 1 & \texttt{dir} = \texttt{left} \\ j_n + 1 & \texttt{dir} = \texttt{right}. \end{cases}$$

    Hence, $i_{n+1} = j_{n+1}$.
  - Next, we consider the value of q_{n+1}:
    - ⋆ If $q_{n+1} = q_n$, then the case block is a *while* block, and vice versa. So, the next module to be executed is m_n. In that case, m_{n+1} still corresponds to q_{n+1}.
    - ⋆ Otherwise, we have an *if* block.
      - · In particular, if q_{n+1} is the `accept` state, then the case for σ_n contains the *flow* command `accept`, and vice versa. In that case, execution terminates with the same final status of `accept`. The same is true for `reject`.
      - · Otherwise, the module contains the command `goto` m_{n+1}, where m_{n+1} is the corresponding module for q_{n+1}.

In that case, P and M execute on T the same way by induction. □

Theorem 3. *Let P be a complete TM program, and let M be the corresponding TM for P. Then, P and M execute on every valid tape T in the same way. That is,*

- *for every valid index n, if we have tape T_n, tapehead index i_n and module m_n for TM program P, and we have tape S_n, tapehead index j_n and state q_n for the TM M, then $T_n = S_n$, $i_n = j_n$ and q_n is the corresponding state for m_n;*
- *P terminates execution on T if and only if M terminates execution on T, with the same final status (`accept` or `reject`).*

Proof. We prove this as well by induction on the execution step of the tape.

- At the start, we have the same tape T for both P and M, with tapehead index 0. Moreover, the initial state q_0 in M corresponds to the first module in P. So, the result is true if $n = 0$.
- Now, assume that the result is true for some integer n, which is not the terminating step in execution. In that case, $S_n = T_n$, $j_n = i_n$ and q_n is the corresponding state for m_n. Let σ_n be the letter at index $j_n = i_n$ on the tape $S_n = T_n$. We now consider the single switch block in m_n:
  - If the block in m_n corresponding to σ_n is a *while* block, then we know that its body is partially complete, and so is composed of the following commands:
    * `changeto` σ_{n+1}
    * `move dir`

    So, we have $\delta(q_n, \sigma_n) = (q_n, \sigma_{n+1}, \texttt{dir})$. Using the same argument as in Theorem 2, we find that $T_{n+1} = S_{n+1}$ and $i_{n+1} = j_{n+1}$. Also, $q_{n+1} = q_n$ is the corresponding state for $m_{n+1} = m_n$.
  - Otherwise, we have an *if* command. In this case, the case body is complete, and so composed of the following commands:
    * `changeto` σ_{n+1}
    * `move dir`
    * `accept`, `reject` or `goto` m_{n+1}.

    So, we have $\delta(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \texttt{dir})$, where q_{n+1} is the corresponding state to the *flow* command present. Here too, we have $T_{n+1} = S_{n+1}$ and $i_{n+1} = j_{n+1}$ by construction.

  Now, we consider the flow command:
  - If we have an `accept` command in the body, then q_{n+1} is the accepting state, and vice versa. So, we terminate execution with the final status of `accept`. The same is true for `reject`.
  - Otherwise, the state q_{n+1} is the corresponding state to the module m_{n+1}.

  In all cases, there is a correspondence between the state for m_{n+1} and q_{n+1}.

So, the result follows from induction. □

Bibliography