TML Documentation

Pete Gautam

October 18, 2022

1 Specification

1.1 TML rules

In this section, we define the rules that a valid program in TML should obey. First, the following is the specification of the TML in EBNF:

$$\begin{aligned}
\mathit{program} &= \mathit{alphabet}\ \mathit{module}^{+} \\
\mathit{alphabet} &= \texttt{alphabet = }\{\ \mathit{seq\text{-}val}\ \} \\
\mathit{module} &= \texttt{module}\ \mathit{id}\ \{\ \mathit{block}^{+}\ \} \\
\mathit{block} &= \mathit{basic\text{-}block} \mid \mathit{switch\text{-}block} \\
\mathit{switch\text{-}block} &= \texttt{switch tapehead}\ \{\ \mathit{case\text{-}block}^{+}\ \} \\
\mathit{case\text{-}block} &= \mathit{if\text{-}block} \mid \mathit{while\text{-}block} \\
\mathit{if\text{-}block} &= \texttt{if}\ \mathit{seq\text{-}val}\ \{\mathit{block}^{+}\} \\
\mathit{while\text{-}block} &= \texttt{while}\ \mathit{seq\text{-}val}\ \{\ \mathit{core\text{-}com}^{+}\ \} \\
\mathit{basic\text{-}block} &= (\mathit{core\text{-}com} \mid \mathit{flow\text{-}com})^{+} \\
\mathit{core\text{-}com} &= \texttt{move}\ \mathit{direction} \mid \texttt{changeto}\ \mathit{value} \\
\mathit{flow\text{-}com} &= \texttt{goto}\ \mathit{id} \mid \mathit{terminate} \\
\mathit{terminate} &= \texttt{reject} \mid \texttt{accept} \\
\mathit{direction} &= \texttt{left} \mid \texttt{right} \\
\mathit{seq\text{-}val} &= (\mathit{value}\texttt{,})^{*}\ \mathit{value} \\
\mathit{value} &= \texttt{blank} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{z} \mid \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{id} &= (\texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \ldots \mid \texttt{Z})^{+}
\end{aligned}$$

Next, we analyse the rules of a TML program.

Rule 1.1.1. A valid TML program is composed of the *alphabet*, followed by one or more *modules*.

Rule 1.1.2. A module contains a collection of a *blocks* (a specific sequence of commands). There are two types of blocks- *basic blocks* and *switch blocks*.

Rule 1.1.3. 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.

Example 1.1.4. The following is a simple TML program:

```
1  alphabet = {"a", "b"}
2  module simple {
3      changeto blank
4      move right
5      changeto a
6      move left
7      accept
8  }
```

It is composed of a single module, called `simple`. The module has 2 basic blocks- lines 3-4, and lines 5-7.

Remark 1.1.5. In the example above, we could have also said that there were 5 basic blocks, one in each line from line 3 to line 7. If it is possible for us to break a module into blocks in different ways, we always choose the one that ends up with the fewest number of blocks. In this case, we cannot have just one block since lines 3-5 do not form a basic block- there are two *changeto* commands. So, we have 2 basic blocks.

Rule 1.1.6. 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.

Rule 1.1.7. The body of an *if* command can be composed of multiple blocks. These blocks can be both basic blocks and switch blocks.

Rule 1.1.8. The body of a *while* command must be composed of a single basic block. The basic block cannot have a *flow* command. Further, a switch block must be the final block present; it cannot be followed by any other block, basic or switch.

Example 1.1.9. The following is another TML program:

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

```
 7             move left
 8             switch tapehead {
 9                 if 0 {
10                     accept
11                 } if 1, blank {
12                     reject
13                 }
14             }
15         }
16     }
17 }
```

It is composed of a single module `isEven`. The module has:

- a switch block at lines 3-16;
- a basic block at line 5;
- a basic block at line 7;
- a nested switch block at lines 8-14;
- a basic block at line 10; and
- a basic block at line 12.

Example 1.1.10. Consider the following TML program:

```
 1 alphabet = {"0", "1"}
 2 module isEven {
 3     switch tapehead {
 4         while 0, 1 {
 5             move right
 6             accept
 7         } if blank {
 8             move left
 9             switch tapehead {
10                 if 0 {
11                     accept
12                 } if 1, blank {
13                     reject
14                 }
15             }
16         }
17     }
18 }
```

This program is not valid because it contains a terminating command in the *while* block at lines 4-6. This does not logically make sense- if we want to terminate, it should be an *if* block; there is no way a *while* block can continue after rejecting the program in the first iteration.

1.2 TML execution

In this section, we discuss how a valid TML program can execute a tape. The tape we will be using has infinite spaces in both directions.

Definition 1.2.1. Let Σ be an alphabet. A *tape T on Σ* is a function $T : \mathbb{Z} \to \Sigma^+$, where $\Sigma^+ = \Sigma \cup \{\texttt{blank}\}$, where `blank` is a symbol not present in Σ.

Remark 1.2.2. A valid TML program always specifies the alphabet. This corresponds to the alphabet Σ used in the tape.

Although there are many possible tapes, we will only be interested in tapes that obey some rules.

Definition 1.2.3. Let Σ be an alphabet and let T be a tape on Σ. Then, 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 $x \leq b$.

Remark 1.2.4. In a valid tape, there are only finitely many values that are non-`blank` and there is no gap between any two non-`blank` values.

Remark 1.2.5. We require the tape to have finitely many non-`blank` entries so that we start execution with the tapehead at the first non-`blank` entry. Moreover, this ensures that going from the start of the tape to the end does not create an infinite loop- this is a very common procedure. If we allowed for infinitely many non-`blank` entries, we would need to specify where the initial location of the tapehead is. Moreover, there would be much fewer programs that we could write which are guaranteed to terminate.

Remark 1.2.6. A valid tape can always be represented as an illustration. For instance, consider the following valid tape T on $\{0, 1\}$:

$$T(x) = \begin{cases} 0 & x \in \{1, 3, 4\} \\ 1 & x \in \{2\} \\ \texttt{blank} & \text{otherwise.} \end{cases}$$

Then, an illustration of the tape is:

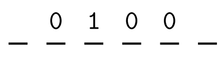


Figure 1: The tape as a diagram.

Only the non-`blank` values are represented. Blank ones are represented by empty lines.

Remark 1.2.7. More than one tape definition can result in the same figure. In the case above, the following tape S on $\{0, 1\}$ also corresponds to the same figure.

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

Example 1.2.8. Consider the following TML program:

```
alphabet = {"a", "b"}
module isSecondValueA {
    move right
    switch tapehead {
        if a {
            accept
        } if b, blank {
            reject
        }
    }
}
```

The following is a valid tape for the program:

$$\underline{}\;\underline{\texttt{a}}\;\underline{\texttt{b}}\;\underline{\texttt{b}}\;\underline{\texttt{a}}\;\underline{}$$

Now, we will define how we can execute a TML program on a valid tape.

Definition 1.2.9. Let P be a TML program, let T be a valid tape for the program with i the smallest integer such that $T(i)$ is not `blank` (i is the initial *tapehead index* and $T(i)$ the initial *tapehead value*). We *execute P on T* by constructing (countable) different tapes until execution is terminated. We first take the given tape and the tapehead index and execute it using the first block b in the first module of P to construct the next tape T' and next tapehead index i'. This is done as follows:

- if the block is a switch block, we take the first block from the case corresponding to the tapehead value- now, we must have a basic block;
- if there is a *changeto* `val` command in the basic block, 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 there is a *move* `dir` command in the basic block, 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 $b' = b$, i.e. we execute this switch block again (not necessarily the same case block);
  - if the block contains a terminating *flow* command, we terminate and 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, we terminate and return the state `reject`.

If execution is not terminated, we execute the block b' with the new tape T' and a new tapehead index i', and continue until we terminate (which not happen).

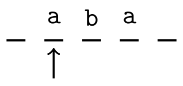
Remark 1.2.10. By construction, for a valid program, precisely one of the 5 possible step applies when choosing the next block to execute or terminate.

Example 1.2.11. Consider the following TML program:

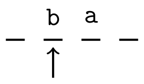
```
alphabet = {"a", "b"}
module palindrome {
    switch tapehead {
        if blank {
            accept
        } if a {
            changeto blank
            move right
            switch tapehead {
                while a, b {
                    move right
                } if blank {
        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

```
26             changeto blank
27             move right
28             switch tapehead {
29                 while a, b {
30                     move right
31                 } if blank {
32                     move left
33                     switch tapehead {
34                         if blank, b {
35                             changeto blank
36                             move left
37                             goto restart
38                         } if a {
39                             reject
40                         }
41                     }
42                 }
43             }
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 }
```

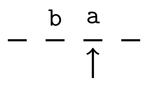
We will execute the program on the following tape.



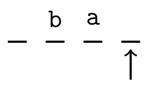
The arrow points to the tapehead value. We first execute the switch block at `palindrome`. Since the tapehead value is `a`, we execute the basic block at lines 7-8. So, we change the tapehead value to `blank`, and the tapehead moves to the right by one step. Since this is an *if*-block, without a flow 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.



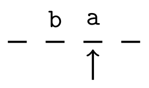
The current block is a switch block. The tapehead value is `b`, so we are at the *while* command at line 11. The basic block here only contains a *move* command. So, we leave the tape as is, and the tapehead moves to the right once. This is a *while* command, so the next block to execute is still this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



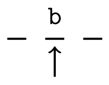
The current block is still a switch block. The tapehead value is `a`, so we execute the same *while* command at line 11. Moreover, the next block to execute is still the switch block. Now, the current tape state is the following.



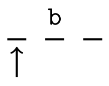
For the third time, we are executing the same switch block. Now, however, the tapehead value is `blank`, so we execute the first block of the *if* command at line 13. Since this is an *if* command and this is not the last block in the if command, the next block to execute is the switch block at lines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



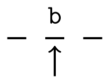
Now, the current block is a switch block. The tapehead value is `a`, so we take the basic block at lines 16-18. The value of the tapehead becomes `blank`, and the tapehead moves to the left. The next block to execute is the switch block in `restart`.



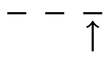
Since the current tapehead value is `b`, we execute the basic block at line 50. So, we move to the left, and the tape is left as is. Moreover, since this is a while block, the next block to execute is still the switch block.



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. At this point, we accept the tape.

2 Proof of Equivalence

2.1 Complete TML programs

Complete programs in the TML are a specific type of TML programs that are very detailed and obey different properties. As such, it is very easy to construct the TM corresponding to a complete program. In this section, we will build to the definition complete programs from complete blocks and modules.

Definition 2.1.1. Let P be a valid TML program, and let B be a basic block in P. We say that B is a *complete block* if it is composed of all the 3 commands: a *changeto* command, a *move* command, and a *flow* command. If the *flow* command is missing, we say that B is a *partially complete block*.

Complete blocks contain all the information required to transition from one state to another. This is because of the following:

- A complete block lists the next value of the tapehead; we assume the original value of the tapehead to be known outwith the block.
- A complete block states which direction the tapehead is moving- left or right.
- A complete block determines precisely what the next state is for the block- it is either a terminating state or we are going to the initial block of another module.

We can convert any basic block with a *flow* command to a complete block by adding the default commands (i.e. moving left and changing the tapehead value to the current value). Equally, it is quite straightforward to convert a complete block into a subpart of a Turing Machine.

Example 2.1.2. Consider the following complete block.

```
changeto a
move right
goto s2
```

If we are at the state s_1, and the block applies when the tapehead value is b, then the following is the corresponding subpart of the Turing Machine:

$$\begin{tikzcd} s_1 \arrow[r, "b \to a, R"] & s_2 \end{tikzcd}$$

Figure 2: The TM subpart of the given Turing Machine complete block.

Using complete and partially complete blocks, we can define complete modules.

Definition 2.1.3. Let P be a valid TML program, and let M be a module in P. We say that M is a *complete module* if all of the following hold:

- it is composed of a single switch block;
- the body of each *if* command is a complete block; and
- the body of each *while* command is a partially complete block.

Remark 2.1.4. For a *while* command, the body must be partially complete because the corresponding edge in the TM is always a loop.

It is quite easy to map a single module to a sub-Turing machine. It precisely corresponds to having an initial state and edges going to other states as dictated by each case. Because the program is valid, we know that there is a case for each letter in the alphabet, including the `blank` letter.

Example 2.1.5. Consider the following complete module.

```
module basic {
    switch tapehead {
        while b {
            changeto b
            move right
        } if a, blank {
            changeto blank
            move left
            reject
        }
    }
}
```

If the alphabet is composed only of a and b, then the corresponding sub-TM is the following:

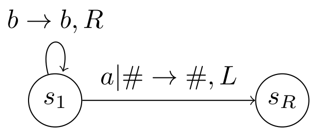


Figure 3: The sub-TM of the given Turing Machine complete module.

Remark 2.1.6. In the example above, we consider the state s_1 to be the *corresponding state* of the module. For every complete module, there is precisely one corresponding state.

Now, we can define complete programs.

Definition 2.1.7. Let P be a TML program. We say that P is *complete* if it is composed of one or more complete modules. We also require every *goto* command in a complete block to refer to an existing module.

Remark 2.1.8. The second condition (called *valid reference*) is required for any TML program.

Remark 2.1.9. In general, a complete program is not composed of a single complete module. We will see later that a relatively simple module can be broken down into a couple of complete modules, each of which refer to each other.

Every module in the program can be converted to a state, along with directed edges to other states. If the program is complete, then we ensure that the states connect to form a valid TM.

Example 2.1.10. Consider the following complete program.

```
alphabet = {"a", "b"}
module isDivTwo {
    switch tapehead {
        while 0 {
            changeto 0
            move right
        } while 1 {
            changeto 1
            move right
        } if blank {
            changeto blank
            move left
            goto isDivTwoCheck
        }
    }
}
module isDivTwoCheck {
    switch tapehead {
        if 0 {
            changeto 0
            move left
            accept
        } if 1, blank {
            changeto blank
            move left
            reject
        }
    }
}
```

Then, the corresponding TM is the following:

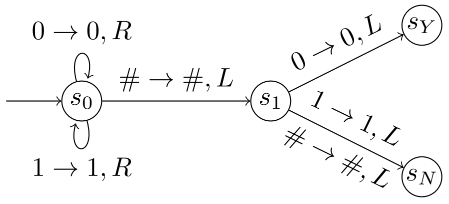


Figure 4: The TM of the given TML program. The state s_0 corresponds to the module `isDivTwo` and the state s_1 corresponds to the module `isDivTwoCheck`.

Remark 2.1.11. In the example above, we converted a complete TML program into a TM. It is equally possible to convert a TM into a complete TML program.

Example 2.1.12. Consider the following TM:

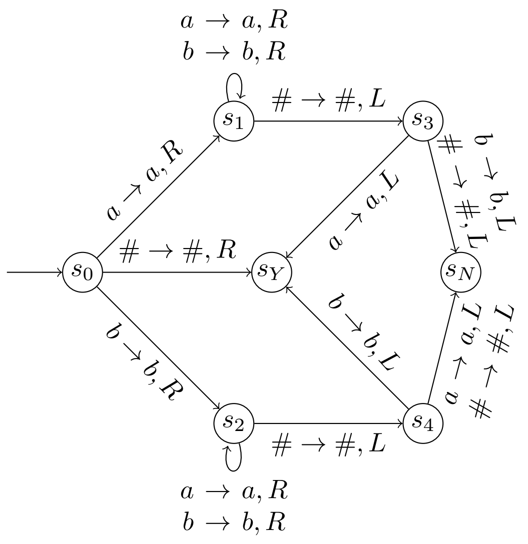


Figure 5: A Turing machine

Then, its corresponding TML program is:

```
alphabet = {"a", "b"}
module startsAndEndsSame {
    switch tapehead {
        if blank {
            accept
        } if a {
            changeto a
            move right
            goto startsAndEndsSameMoveA
        } if b {
            changeto b
            move right
            goto startsAndEndsSameMoveB
        }
    }
}
module startsAndEndsSameMoveA {
    switch tapehead {
        while a {
            changeto a
            move right
        } while b {
            changeto b
            move right
        } if blank {
            changeto blank
            move left
            goto startsAndEndsSameCheckA
        }
    }
}
module startsAndEndsSameCheckA {
    switch tapehead {
        if a {
            changeto a
            move left
            accept
        } if b {
            changeto b
            move left
            reject
        } if blank {
            changeto blank
            move left
            reject
        }
    }
}
module startsAndEndsSameMoveB {
    switch tapehead {
        while a {
            changeto a
            move right
        } while b {
            changeto b
            move right
        } if blank {
            changeto blank
            move left
            goto startsAndEndsSameCheckB
        }
    }
}
module startsAndEndsSameCheckB {
    switch tapehead {
        if a {
            changeto a
            move left
            reject
        } if b {
            changeto b
            move left
            accept
        } if blank {
            changeto blank
            move left
            reject
        }
    }
}
```

This is a complete program since TMs always include the required commands corresponding to *if* and *while* commands.

Remark 2.1.13. Although a TML program need not be complete, any valid TML program is equivalent to a complete one. So, a TML program that is not be complete is just a compact representation of its complete version.

2.2 Complete TML programs to TM

In this section, we will convert TML programs into TM. To do this, we first need to define TMs.

Definition 2.2.1. A *Turing Machine* 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, excluding the `blank` symbol;
- $\delta : Q \setminus \{\texttt{accept}, \texttt{reject}\} \times \Sigma^+ \to Q \times \Sigma^+ \times \{\texttt{left}, \texttt{right}\}$, where $\Sigma^+ = \Sigma \cup \{\texttt{blank}\}$, is the transition function; and
- $q_0 \in Q$ is the starting state.

Remark 2.2.2. The definition of a valid tape only depends on the alphabet. So, for a TM program with language set equal to Σ and a TM with alphabet Σ, the set of valid tapeheads is equivalent.

Like with TML programs, we can execute a TM on a valid tape.

Definition 2.2.3. Let M be a TM, and let T be a valid tape for the program, with i the smallest integer such that $T(i)$ is not `blank`. We *execute* M *on* T by constructing (countable) different tapes until execution is terminated. We first take the given tape T and the tapehead index i, and execute it using the initial state q_0. This is done by computing $\delta(q_0, t) = (q_1, t', \mathtt{dir})$. Then,

- the next tape T' is given by

$$T'(x) = \begin{cases} t' & x = i \\ T(x) & \text{otherwise;} \end{cases}$$

- the next state is q_1; and
- the next tapehead index is given by:

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

  If the next state q_1 is not a terminating state (`accept` or `reject`), then we execute the tape T' with the next state q_1 and the next tapehead index i'.

Definition 2.2.4. Let M be a TM. For each state q in M, we define the *corresponding module for* q, m, as follows:

- the module contains a single *switch* command;
- for each letter σ in the alphabet Σ^+, we find $\delta(q, \sigma) = (q', \sigma', \mathtt{dir})$. Then, we add a case in the *switch* command corresponding to letter σ (an *if* case if $q' \neq q$, otherwise a *while* case) with:
  - `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'.

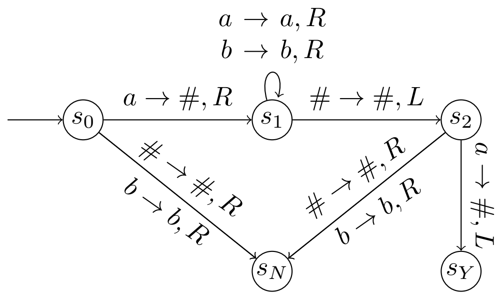
Let P be the program with

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

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

Remark 2.2.5. For any TM M, its corresponding program P will be complete. By definition, every module in P is complete.

Example 2.2.6. Consider the following TM:



Then, its corresponding TML program is the following:

```
alphabet = {"a", "b"}
module s0 {
    switch tapehead {
        if a {
            changeto blank
            move right
            goto s1
        } if b {
            changeto b
            move right
            reject
        } if blank {
            changeto blank
            move right
            reject
        }
    }
}
module s1 {
    switch tapehead {
        while a {
            changeto a
            move right
        } while b {
            changeto b
            move right
        } if blank {
            changeto blank
            move left
            goto s2
        }
    }
}
module s2 {
    switch tapehead {
        if a {
            changeto blank
            move left
            accept
        } if b {
            changeto b
            move right
            reject
        } if blank {
            changeto blank
            move right
            reject
        }
    }
}
```

Theorem 2.2.7. *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 of the tape. 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}.

Therefore, if the result holds for n, it holds for $n+1$. So, the result follows from induction. □

Definition 2.2.8. Let P be a complete TM program, and let Σ be its alphabet. For each module m in P, we define the *corresponding state for* m, 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 is `accept`, then q' is the `accept` state; if it is `reject`, then q' is the `reject` state; otherwise, q' is the state corresponding to the module given in the *goto* command within m.

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

Example 2.2.9. 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:

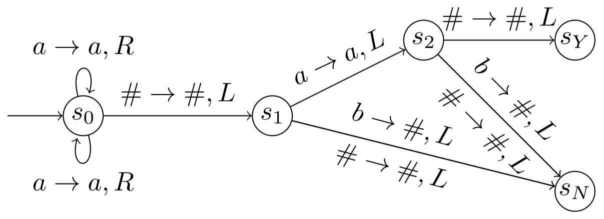


Figure 6: The TM corresponding to the program above. The state s_0 corresponds to the module `moveToEnd`; the state s_1 corresponds to the module `checkAFirst`; and the state s_2 corresponds to the module `checkASecond`.

Theorem 2.2.10. *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 block 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 **2.2.7**, 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. □

2.3 Valid to complete programs

In this section, we will show that for every valid program, there is a corresponding complete program. We will first define how to 'complete' a valid program, and then show that the completion executes on a tape in the same way as a valid program.

Definition 2.3.1. Let P be a TM program. We define the TM program P_1 to be the *first completion of P* as follows:

- the alphabet of P_1 is the same as the alphabet of P;
- for every block b in every module m of P, P_1 has a module with body the block b. If the block is not the final block in m, it has an additional flow command to the next block in the module. The order of the modules is the same as the order of the original blocks.

We define the TM program P_2 to be the *second completion of P* as follows:

- the alphabet of P_2 is the same as the alphabet of P;
- for every module m of P with a *basic* block, replace it with a *switch* block where every case is an *if* command with body the original basic block.

Finally, we define the TM program P^+ to be the *(third) completion of P* as follows:

- the alphabet of P^+ is the same as the alphabet of P;
- for every non-complete module m of P, replace it with a complete block by adding the default commands:
  - if the *changeto* command is missing, add the *changeto* command corresponding to the case value;

– if the *move* command is missing, add the command `move left`;
– if the *flow* command is missing, add the command `reject`.

Remark 2.3.2. For any TM program P, its completion P^+ is complete by construction.

Remark 2.3.3. Let P be a TM program. For every block b in P, there is a complete module m in the completion program P^+. We say that m is the *corresponding complete module* of b.

Example 2.3.4. Consider the following TM program:

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

Them, its first completion is the following program:

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

The second completion is 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, the completion of P is the following:

```
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
        }
    }
}
```

Theorem 2.3.5. *Let P be a valid TM 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. At the start, we have the same tape T for both P and P^+, with tapehead index 0. Moreover, the corresponding (complete) 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 TM 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 `changeto` σ_n. So, the next tape is given by:

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

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 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 `move left`, so we also have $j_{n+1} = j_n - 1$. So, we have $i_{n+1} = j_{n+1}$. Otherwise, we have the same *move* command, meaning that $i_{n+1} = j_{n+1}$.

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. □