TML EQUIVALENCE PROOF

PETE GAUTAM

Abstract. This document proves the equivalence of the TML with TMs. We do this by constructing an equivalent TML program for a TM, and vice versa. In the first section, we define complete TML programs, which are very close to TMs, and describe how they can be mapped into TMs. In the second section, we define formally how complete TML programs can be transformed into TMs, and vice versa. Finally, the third section describes how we can transform a valid TML program into a complete TML program.

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 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 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:

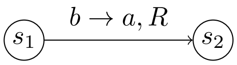

FIGURE 1. The TM subpart of the given Turing Machine complete block.

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

Definition 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 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 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:

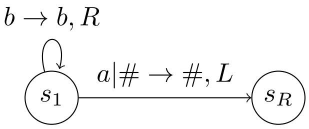

FIGURE 2. The sub-TM of the given Turing Machine complete module.

Remark 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 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 1.8. The second condition (called *valid reference*) is required for any TML program.

Remark 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 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:

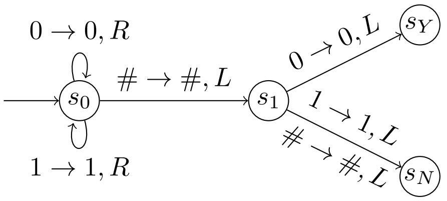

FIGURE 3. The TM of the given TML program. The state s_0 corresponds to the module `isDivTwo` and the state s_1 corresponds to the module `isDivTwoCheck`.

Remark 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 1.12. Consider the following TM:

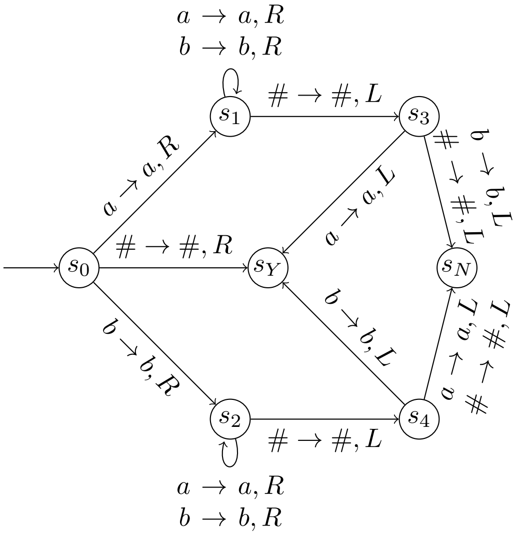

FIGURE 4. A Turing machine

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 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. 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 2.1. 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. 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 2.3. 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 $\delta(q_0, t) = (q_1, t', \texttt{dir})$. 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:

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

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 2.4. 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 $\delta(q, \sigma) = (q', \sigma', \texttt{dir})$. 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 2.5. For any TM M, its corresponding program P will be complete. By definition, every module in P is complete.

Example 2.6. Consider the following TM:

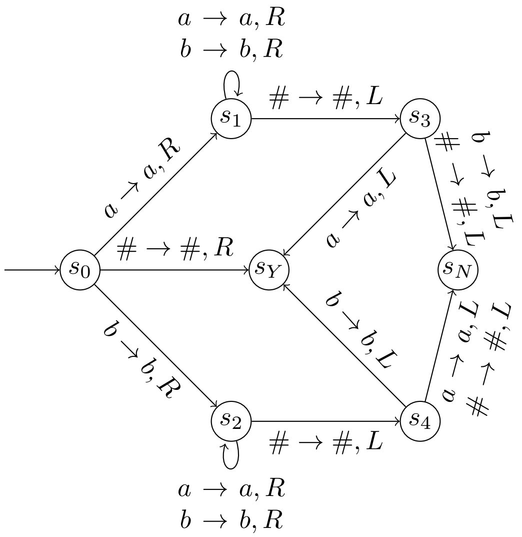

FIGURE 5. A Turing machine

Then, its corresponding TM 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.

Theorem 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. 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. Now, let σ_n be the letter at index $i_n = j_n$ on the tape $T_n = S_n$. Compute $q(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \texttt{dir})$. 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:

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

So, $i_{n+1} = j_{n+1}$. Now, 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 $n + 1$. Then, the result follows from induction. □

Definition 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 2.9. 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:

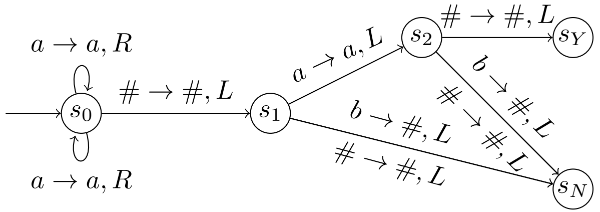

FIGURE 6. The TM corresponding to the program above. The state s_0 corresponds to the module `moveToEnd`; the state s_1 corresponds to the module `checkAFirst`; and the state s_2 corresponds to the module `checkASecond`.

Theorem 2.10. *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 by induction as well. 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. Now, let σ_n be the letter at index $j_n = i_n$ on the tape $S_n = T_n$. 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 **2.7**, 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 $\delta(q_n, \sigma_n) = (q_n, \sigma_{n+1}, \texttt{dir})$, where σ_{n+1} is the corresponding state to the *flow* command present. Here too, we have $T_{n+1} = S_{n+1}$, $i_{n+1} = j_{n+1}$. 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}. So, the result follows from induction. □

3. Valid to complete programs

Definition 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 3.2. For any TM program P, its completion P^+ is complete by construction.

Remark 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 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 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}. Also, 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 block and the next module, or both the program terminate with the same status. □