Proof of Equivalence

Pete Gautam

March 17, 2023

In this document, we give a proof of equivalence between TMs and TML programs. This is done in many steps that involve:

- proving that a TM can be converted to a complete TML program;
- proving that a complete TML program can be converted to a TM; and
- proving that a valid TML program can be converted to a complete TML program.

1 Complete TML Programs

When we defined execution of a valid TML program on a tape in the specification, 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 establish in this section.

Consider the following complete program.

```
alphabet = {0, 1}
module isOdd {
    // move to the end
    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
    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 now illustrate this process with an example. Assume we first have the following program.

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

In step 1 of the completion process, we create a module for each block. In this program, there are two basic blocks- at lines 3-4 and 5-6. So, after applying the first step, we get the following program.

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

In this case, we have two basic blocks at lines 3-5 and 8-9. So, in step 2, we convert them into switch blocks and get the following program.

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

Finally, we add all the default values in step 3 and get the following program.

```
alphabet = {a, b}
module simple1 {
    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 {
    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 1.1. *Let P be a valid TML program. Then, P and its completion P^+ execute on every 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 TML 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. □

2 Equivalence of TMs and TMLs

In this section, we will show that there is an equivalence between TMs and valid 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 TML program, and show the equivalence in this case as well.

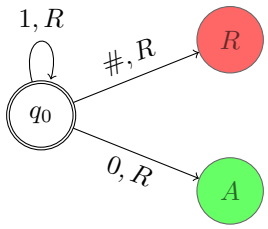

Figure 1: A TM that accepts binary strings containing 0

We will first illustrate how to convert a TM to a (complete) TML program. So, consider the TM at Figure 1. Then, its corresponding TML program is the following:

```
alphabet = {0, 1}
module has0 {
    if 1 {
```

```
        changeto 1
        move right
        goto has0
    } if 0 {
        changeto 0
        move right
        accept
    } if blank {
        changeto blank
        move left
        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 an *if* case in the *switch* command corresponding to letter σ with the following commands:
  - `changeto` σ'
  - `move` *dir*
  - 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 2.1. *Let M be a TM, and let P be the corresponding program for M. Then, M and P execute on every 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 TML 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 TML 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. □

Next, we construct a TM for a TML program. This process is essentially the inverse of the one we saw converting a TML program to a TM. In particular, 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.

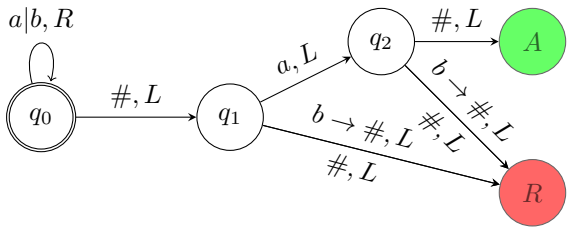

Figure 2: The TM corresponding to the program above. The state q_0 corresponds to the module `moveToEnd`; the state q_1 corresponds to the module `checkAFirst`; and the state q_2 corresponds to the module `checkASecond`.

We now illustrate this process with an example. So, consider the following TML program:

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

```
        changeto blank
        move left
        accept
    } if b, blank {
        changeto blank
        move left
        reject
    }
}
```

Then, its corresponding TM is given in Figure 2.

Theorem 2.2. *Let P be a TML program, and let M be the corresponding TM for P. Then, P and M execute on every 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 TML 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. Without loss of generality, assume that P is complete. 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 2.1, 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. □

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

3 TML as a model of computation

Since TML programs and TMs are equivalent, this implies that TML programs are a model for computation. Note that while we have given a proof for equivalence for TMs, this representation is based on accepting and rejecting programs only. Not all TMs are of this form. In particular, it is also possible for a TM to halt instead of accepting or rejecting.

Although the equivalence is limited to a subclass of TMs, we can add another flow command `halt` that mimics the halting behaviour. However, this is not necessary- we can use the accept state (or equally the reject state) to mimic the behaviour of halting. Since accepting or rejecting results in the program halting, we can simply disregard the final result, and possibly read the output from the tape to infer the actual result.