

# Variants of Turing Machines

Teodor Rus

`rus@cs.uiowa.edu`

The University of Iowa, Department of Computer Science

# Robustness

- Robustness of a mathematical object (such as proof, definition, algorithm, method, etc.) is measured by its invariance to certain changes;
- To prove that a mathematical object is robust one needs to show that it is equivalent with its variants.

**Question:** is the definition of a Turing machine robust?

# TM definition is robust

- Variants of TM with multiple tapes or with nondeterminism abound;
- Original model of a TM and its variants all have the same computation power, i.e., they recognize the same class of languages.

Hence, robustness of TM definition is measured by the invariance of its computation power to certain changes.

# Example of robustness

**Note:** transition function of a TM in our definition forces the head to move to the left or right after each step. Let us vary the type of transition function permitted.

- Suppose that we allow the head to stay put, i.e.;  
 $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$  where  $S$  stands for stationary.
- Does this feature allow TM to recognize additional languages? **Answer: NO.**

**Sketch of proof:**

1. An  $S$  transition can be represented by two transitions: one that move to the left followed by one that moves to the right;
2. Since we can convert a TM which stay put into one that has no this facility the answer is No.

# Equivalence of TMs

To show that two models of TM are equivalent we need to show that we can simulate one by another.

# Multitape Turing Machines

A multitape TM is like an ordinary TM with several tapes.

- Each tape has its own head for reading/writing;
- Initially the input is on tape 1 and other are blank;
- Transition function allow for reading, writing, and moving the heads on all tapes simultaneously, i.e.,

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

where  $k$  is the number of tapes.

# Formal expression

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L)$$

means that:

“if the machine is in state  $q_i$  and heads 1 through  $k$  are reading symbols  $a_1$  through  $a_k$  the machine goes to state  $q_j$ , writes  $b_1$  through  $b_k$  on tapes 1 through  $k$  respectively and moves each head to the left or right as specified by  $\delta$ ”

# Theorem (3.8) 3.13

*Every multitape Turing machine has an equivalent single tape Turing machine.*

**Proof idea:**

1. Show how to convert a multitape TM  $M$  into a single tape TM  $S$ ;
2. Then show how to simulate  $M$  with  $S$ .

# Simulating $M$ with $S$

Assume that  $M$  has  $k$  tapes:

- $S$  simulates the effect of  $M$  using  $k$  tapes by storing their information on  $S$ 's single tape;
- $S$  uses a new symbol  $\#$  as a delimiter to separate the contents of different tapes;
- $S$  keeps track of the location of the heads by marking with a  $\bullet$  the symbols where the heads would be.

# Example simulation

Figure 1 shows how to represent a machine  $M$  with 3 tapes by a machine  $S$  with one tape.

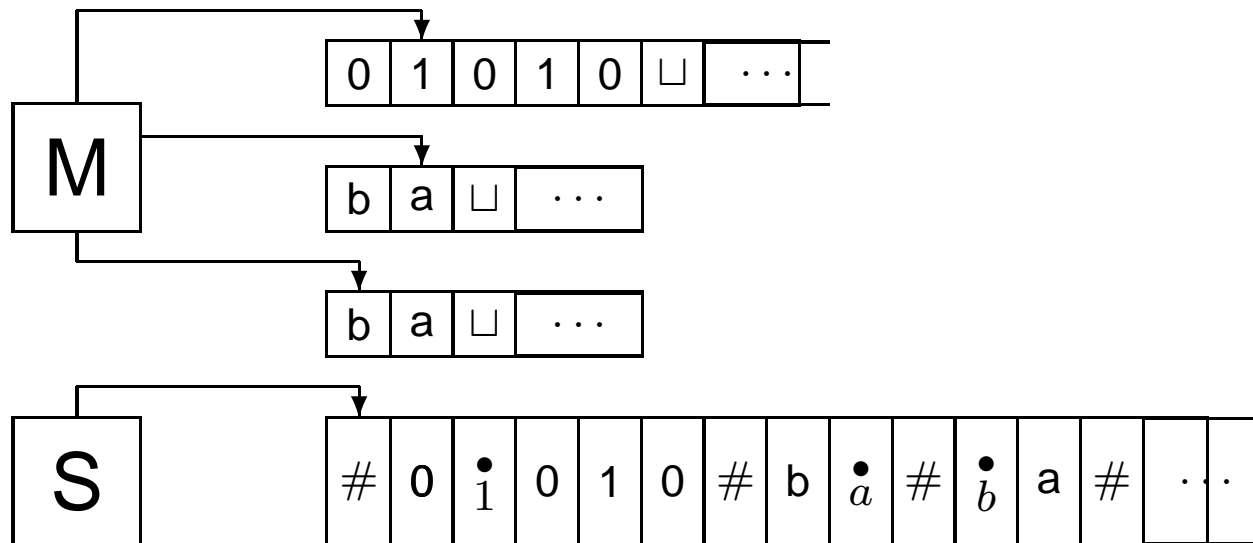


Figure 1: Multitape machine simulation

# General construction

$S =$  "On input  $w = w_1w_2 \dots w_n$

1. Put  $S(\text{tape})$  in the format that represents  $M(\text{tapes})$ :

$$S(\text{tape}) = \# \overset{\bullet}{w}_1 \dots w_n \# \overset{\bullet}{\square} \# \dots \# \overset{\bullet}{\square} \#$$

2. Scan the tape from the first  $\#$  (which represent the left-hand end) to the  $(k + 1)$ -st  $\#$  (which represent the right-hand end) to determine the symbols under the virtual heads.

Then  $S$  makes a second pass over the tape to update it according to the way  $M$ 's transition function dictates.

# Moving onto unread-tape

3. If at any time  $S$  moves one of the virtual heads to the right of  $\#$  it means that  $M$  has moved on the corresponding tape onto the unread blank portion of that tape.

## Implementation:

- (a)  $S$  shifts the tape contents from this cell until the rightmost  $\#$  one unit to the right;
- (b) Then  $S$  writes a  $\square$  on the free tape cell thus obtained.
- (c) Then  $S$  continues to simulate as before".

# Corollary 3.15

A language  $L$  is Turing recognizable iff some multitape Turing machine recognizes it

**Proof:**

- **if:** a Turing recognizable language  $L$  is recognized by an ordinary TM. But an ordinary TM is a special case of a multitape TM.

Hence, if  $L$  is Turing recognizable then it is recognized by a multitape TM.

- **only if:** If  $L$  is recognized by a multitape TM then it is also recognized by the equivalent single tape TM.

That is, if  $L$  is recognized by a multitape  $TM$  then  $L$  is Turing recognizable.

# Nondeterministic TM

- A NTM is defined in the expected way:  
at any point in a computation the machine may proceed according to several possibilities

**Formally:**  $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$

- Computation performed by a NTM is a tree whose branches correspond to different possibilities for the machine;
- If some branch of the computation tree leads to the *accept* state, the machine accepts the input.

# Theorem 3.16

Every nondeterministic Turing machine,  $N$ , has an equivalent deterministic Turing machine,  $D$ .

**Proof idea:** show that  $N$  can be simulated with  $D$ .

**Simulation:**

1.  $D$  tries all possible branches of  $N$ 's computation;
2. If  $D$  ever finds the accept state on one of these branches then it accepts. Otherwise  $D$  simulation will not terminate.

# More on NTM $N$ simulation

- $N$ 's computation on an input  $w$  is a tree,  $N(w)$ ;
- Each branch of  $N(w)$  represents one of the branches of the nondeterminism;
- Each node of  $N(w)$  is a configuration of  $N$ ;
- The root of  $N(w)$  is the start configuration.

## Implementation:

$D$  searches  $N(w)$  for an accepting configuration.

# A Tempting Idea

Design  $D$  to explore  $N(w)$  by a *depth-first search*.

**Note:**

- A depth-first search goes all the way down on one branch before backing up to explore next branch;
- Hence,  $D$  could go forever down on an infinite branch and miss an accepting configuration on an other branch.

# A Better Idea

Design  $D$  to explore the tree by using a *breadth-first search*.

**Note:**

- This strategy explores all branches at the same depth before going to explore any branch at the next depth;
- Hence, this method guarantees that  $D$  will visit every node of  $N(w)$  until it encounters an accepting configuration.

# Formal proof

$D$  has three tapes, Figure 2:

1. Tape 1 (input tape) contains the input and is never altered;
2. Tape 2 (simulation tape) maintains a copy of  $N$ 's tape on some branch of its nondeterministic computation;
3. Tape 3 (address tape) keeps track of  $D$ 's location in  $N$ 's nondeterministic computation tree.

# Deterministic simulation of $N$

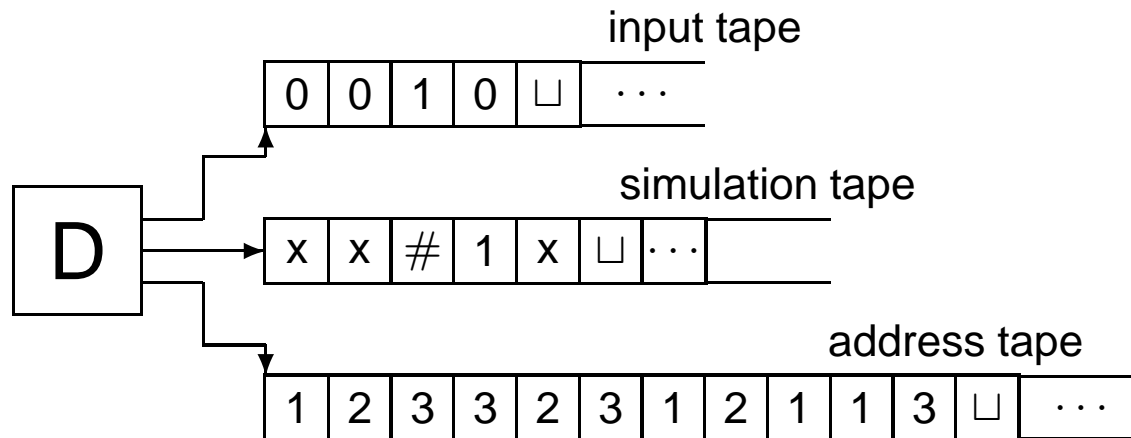


Figure 2: Deterministic TM  $D$  simulating  $N$

# Address tape

- Every node in  $N(w)$  can have at most  $b$  children, where  $b$  is the size of the largest set of possible choices given by  $N$ 's transition function;
- Hence, to every node we assign an address that is a string in the alphabet  $\Sigma_b = \{1, 2, \dots, b\}$ .

## Example:

the address  $\sqcup 231$  represents the  $N(w)$ 's node reached by starting at the root  $\sqcup$ , going to its second child and then going to that node's third child and then going to that node's first child.

# Facts

1. Each symbol in a node address tells us which choice to make next when simulating a step in one branch in  $N$ 's nondeterministic computation;
2. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. **In that case the address is invalid and doesn't correspond to any node;**
3. Tape 3 contains a string over  $\Sigma_b$  which represents a branch of  $N$ 's computation from the root to the node addressed by that string, unless the address is invalid;
4. The empty string (denoted  $\sqcup$  here) is the address of the root.

# The description of $D$

$D =$  "For any input  $w$ :

1. Set tape 1 to contain  $w$  and tape 2 and 3 to empty;
2. Copy tape 1 over tape 2
3. Use tape 2 to simulate  $N$  with input  $w$  on one branch of its nondeterministic computation:
  - (a) Consult the next symbol on tape 3 to determine which choice to make among those allowed by  $N$ 's transition function;
  - (b) If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4;
  - (c) If a rejecting configuration is reached go to stage 4; if an accepting configuration is encountered, *accept*.
4. Replace the string on tape 3 with the lexicographically next string and simulate the next branch of  $N$ 's computation by going to stage 2".

# Corollary 3.17

A language  $L$  is Turing-recognizable iff some nondeterministic TM recognizes it.

**Proof:**

- **if:** If a language is Turing-recognizable it is recognized by a DTM. Any DTM is automatically an NTM. Hence, if  $L$  is Turing-recognizable it is recognized by an NTM.
- **only if:** If a language is recognized by a NTM then it is Turing-recognizable. If  $L$  is recognized by a NTM  $N$  then  $L$  is also recognized by the TM  $D$  equivalent with  $N$ . That is,  $L$  is Turing-recognizable.

# Corollary 3.19

A language is decidable iff some NTM decides it.

Sketch of a proof:

- **if:** If a language  $L$  is decidable, it is decided by a DTM. Since a DTM is automatically a NTM, it follows that:

if  $L$  is decidable it is decidable by a NTM.

- **only if:** If a language  $L$  is decided by a NTM  $N$  it means that  $N$  halts on any input  $w \in \Sigma$ . Construct the TM  $D'$  that decides  $L$  by:

$D'$  = On any input  $w \in \Sigma$ :

1. Runs the TM  $D$  that simulates  $N$  on  $w$  (see the proof of theorem 3.16);
2. Reject if all branches of nondeterminism of  $N$  are exhausted.

That is, if  $L$  is decided by a NTM  $N$  it is decidable.

# Detailed description of $D'$

$D'$  = "On any input  $w$ :

1. Set tape 1 to contain  $w$  and tape 2 and 3 to empty;
2. Copy tape 1 over tape 2;
3. Simulate  $N$  with input  $w$  on one branch of its nondeterministic computation:
  - (a) Consult the next symbol on tape 3 to determine which choice to make among those allowed by  $N$ 's  $\delta$  ( $\sqcup$  represents the root);
  - (b) If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4;
  - (c) If a rejecting configuration is reached go to stage 4; if an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the lexicographically next string and simulate the next branch of  $N$ 's computation by going to stage 2.
5. *Reject* if all branches of  $N$  are exhausted.

# $D'$ is a decider for $L$

To prove that  $D'$  decide  $L$  we use the following theorem:

**Tree theorem:** if every node in a tree has finitely many children and any branch of the tree has finitely many nodes then the tree itself has finitely many nodes.

**Proof:**

1. If  $N$  accepts  $w$ ,  $D'$  will eventually find an accepting branch and will accept  $w$  as well;
2. If  $N$  rejects  $w$ , all of its branches halt and reject because  $N$  is a decider. Consequently each branch has finitely many nodes, where each node represents a step in  $N$ 's computation along that branch;
3. According to the tree theorem, entire computation tree is finite, and thus  $D'$  halts and rejects when the entire tree has been explored.

# Enumerators

- An enumerator is a variant of a TM with an attached printer;
- The enumerator uses the printer as an output device to print strings;
- Every time the TM wants to add a string to the list of recognized strings it sends it to the printer.

## Note:

1. Some people use the term *recursively enumerable* language for languages recognized by enumerators;
2. See the lecture notes "Algorithms and recursive functions".

# Formal definition

An enumerator is a 7-tuple  $E = (Q, \Sigma, \Gamma, \delta, q_0, q_{print}, q_{halt})$  where  $Q, \Sigma, \Gamma$  are as in the definition of a TM and

1.  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} \times \Sigma_\epsilon$ ;
2.  $q_0 \in Q$  is the start state;
3.  $q_{print} \in Q$  is the print state;
4.  $q_{halt} \in Q$  is the reject state,  $q_{print} \neq q_{halt}$ .

# Computation of an enumerator

- An enumerator starts with a blank input tape;
- If the enumerator does not halt it may print an infinite list of strings;
- The language recognized by the enumerator is the collection of strings that it eventually prints out.

**Note:** an enumerator may generate the strings of the language it recognizes in any order, possibly with repetitions.

# Formal

The computation of the enumerator  $E$  is defined as in the case of an ordinary TM, except for the following difference:

1.  $E$  has two tapes, a work tape and a print tape, both initially blank.
2. At each step the machine may write a symbol from  $\Sigma$  on the print tape or do nothing, as defined by  $\delta$ .
3. If  $\delta(q, a) = (r, b, L, c)$  it means that in state  $q$ , reading  $a$ , the enumerator  $E$  enters state  $r$ , writes  $b$  on the work tape, move the work tape head left (or right, if L had been R), write  $c$  on the print tape, and move the print tape head to the right if  $c \neq \epsilon$ .
4. Whenever state  $q_{print}$  is entered, the print tape is printed-out and then it is reset to blank and its head returns to the left-end;  $E$  halts when  $q_{halt}$  is entered.

$$L(E) = \{w \in \Sigma^* \mid w \text{ appears on the print tape if } q_{print} \text{ is entered}\}$$

# Theorem 3.21

A language  $L$  is Turing-recognizable iff some enumerator enumerates it.

**Proof:**

- **if:** If  $L$  is recognizable it means that there is a TM  $M$  that recognizes  $L$ . Then we can construct an enumerator  $E$  for  $L$ . For that consider  $s_1, s_2, \dots$ , the list of all possible strings in  $\Sigma^*$ , where  $\Sigma$  is the alphabet of  $M$ .

$E =$  "Ignore the input.

1. Repeat for  $i = 1, 2, 3, \dots$ :
2. Run  $M$  for  $i$  steps on each input  $s_1, s_2, \dots, s_i$ ;
3. If any computation *accepts*, prints out the corresponding  $s_j$ ."

**Question:** how can we construct the list  $s_1, s_2, \dots$ ?

# Fact

If  $M$  accepts  $s$ , eventually it will appear on the list generated by  $E$ .

**Note:**

In fact  $s$  will appear infinitely many times because  $M$  runs from the beginning on each string for each repetition of step 1.

I.e., it appears that  $E$  runs  $M$  in parallel on all possible input strings.

# Proof, continuation

- **only if:** If we have an enumerator  $E$  that enumerates a language  $L$  then a TM  $M$  recognizes  $L$ .  $M$  works as follows:

$M =$  "On input  $w$ :

1. Run  $E$ . Every time  $E$  outputs a string, compare it with  $w$ .
2. If  $w$  ever appears in the output of  $E$  *accept*."

Clearly  $M$  accepts those strings that appear on  $E$ 's list.

# Problems

Now we solve two problems from the textbook:

- **Problem 3.11:** show that a *Turing machine with double infinite tape* is equivalent to an ordinary Turing machine.
- **Problem 3.14:** show that a *queue automaton* is equivalent to an ordinary Turing machine.

# TM with double infinite tape

A *Turing machine with double infinite tape* is like an ordinary Turing machine but its tape is infinite in both directions, to the left and to the right.

## Assumptions:

- The tape is initially filled with blanks except for the portion that contains the input;
- Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward.

# Problem

Show that a TM  $D$  with double infinite tape can simulate an ordinary TM  $M$  and an ordinary TM  $M$  can simulate a TM  $D$  with double infinite tape.

# Simulating $M$ by $D$

A TM  $D$  with double infinite tape can simulate an ordinary TM  $M$  by marking the left-hand side of the input to detect and prevent the head from moving off of that end.

This is done by:

1. Mark the left-hand end of the input. Let this mark be  $\perp \notin \Gamma_D$ .
2. Each transition  $\delta_D(q, \perp) = (q', a, L)$  performs as follows:

$$q \perp v \vdash q \perp av$$

# Simulating $D$ by $M$

We show first how to simulate  $D$  with a 2-tape TM  $M$  which was already shown to be equivalent to an ordinary TM.

- The first tape of the 2-tape TM  $M$  is written with the input string and the second tape is blank.
- Then cut the tape of  $D$  into two parts, at the starting cell of the input string.
- The portion with the input string and all the blank spaces to its right appears on the first tape of the 2-tape TM.

The portion to the left of the input string appears on the second tape, in reverse order, Figure 3

# Replacing $D$ -tape by 2 $M$ -tapes

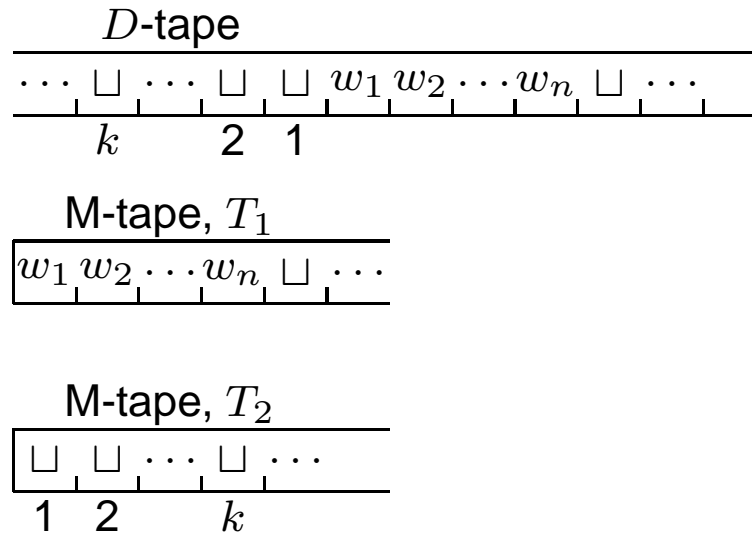


Figure 3: Representing  $D$  tape by 2  $M$ -tapes

# Deterministic Queue Automata

A DQA is like a push-down automaton except that the stack is replaced by a queue.

- A *queue* is a tape allowing symbols to be written only on the left-hand end and read only at the right hand-end;
- Each write operation (called a *push*) adds a symbol to the left-hand end of the queue;
- Each read operation (called a *pull*) reads and removes a symbol at the right-hand end.

**Note:** As with a PDA, the input of a DQA is placed on a separate read-only input tape, and the head on the input tape can move only from left to right.

# More on the DQA

- **Initial condition:** the input tape of a DQA contains a cell with a blank symbol following the input, so that the left-end of the input can be detected.
- **Computation:** A queue automaton accepts its input by entering a special accept state at any time.
- **Problem:** Show that a language can be recognized by a deterministic queue automaton, DQA, iff the language is Turing-recognizable.

# Solution sketch

- Show that any DQA  $Q$  can be simulated with a 2-tape TM  $M$ ;
- Show that any single-tape deterministic TM  $D$  can be simulated by a DQA  $Q$ .

# Simulating a DQA $Q$ by a TM $M$

Use a two-tape TM  $M$ :

- The first tape of  $M$  holds the input, and the second tape of  $M$  holds the queue.
- To simulate reading  $Q$ 's next input symbol,  $M$  reads the symbol under the first head and moves to the right.
- To simulate a *push*  $a$ ,  $M$  writes  $a$  on the leftmost blank cell of the second tape.
- To simulate a *pull*,  $M$  reads the rightmost symbol on the second tape and shifts the tape one symbol leftward.

**Note:** Multitape TM-s are equivalent to single tape TM-s, so we can conclude that if a language is recognized by DQA is is recognized by a TM.

# Simulating a TM $M$ with a DQA $Q$

$$M = (Q_m, \Sigma, \Gamma_M, \delta_M, q_0^M, q_a^M, q_r^M)$$

$$Q = (Q_Q, \Sigma, \Gamma_M \cup \hat{\Gamma}_M, \delta_Q, q_0^Q, \{q_a^M, q_r^M\})$$

- For each symbol  $c$  of  $M$ 's tape alphabet  $\Gamma_M$ , the alphabet  $\Gamma_Q$  of  $Q$  has two symbols:  $c$  and  $\hat{c}$ .
- We use  $\hat{c}$  to denote  $c$  with  $M$ 's head over it.
- In addition  $\Gamma_Q$  has an left *end-of-tape* marker symbol denoted \$.

# The simulation

- $Q$  simulates  $M$  by maintaining a copy of the  $M$ 's tape in the queue.
- $Q$  can effectively scan the tape from right to left by pulling symbols from the right-hand end of the queue and pushing them back on the left-hand end side, until  $\$$  is seen.
- When a  $\hat{c}$  symbol is encountered,  $Q$  can determine  $M$ 's next move, because  $Q$  can record  $M$ 's current state in its control.

# Computation: $M$ moves to left

- If  $M$ 's tape head moves leftwards, the updating of the queue is done by writing the new symbol  $c$  instead of the old  $\hat{c}$  and moving the  $\hat{\cdot}$  one symbol leftwards.

**Formally:** if current configuration is  $u a \hat{b} t v$  and  $\delta(q, b) = (q', c, L)$  then the next configuration is  $u \hat{a} c t v$  and is obtained by:

pull  $v$ ; push  $v$ ;

pull  $t$ ; push  $t$ ;

pull  $\hat{b}$ ; push  $c$ ; pull  $a$ ; push  $\hat{a}$ ;

pull  $u$ ; push  $u$

# Computation: $M$ moves to right

- If  $M$ 's tape head moves rightward, the updating is harder because the  $\hat{c}$  must go to the right.
- By the time  $\hat{c}$  is pulled from the queue, the symbol which receives the  $\hat{c}$  has already been pushed onto the queue.

# Solution

The solution is to hold tape symbols in the control for an extra move, before pushing them onto the queue. This gives  $Q$  enough time to move the  $\hat{\phantom{t}}$  rightward if necessary.

**Formally:** if current configuration is  $u a \hat{b} t v$  and  $\delta(q, b) = (q', c, R)$  then the next configuration is  $u a c \hat{t} v$  and is obtained by:

```
pull v; push v;
pull t; hold (t);
pull hat(b); push hat(t); push(c);
pull a; push a;
pull u; push u;
```

# Equivalence with other models

- There are many other models of general purpose computation.

**Example:** recursive functions, normal algorithms, semi-thue systems,  $\lambda$ -calculus, etc.

- Some of these models are very much like Turing machines; other are quite different;
- All share the essential feature of a TM: unrestricted access to unlimited memory;
- All these models turn out to be equivalent in computation power with TM.

# Analogy

- There are hundreds of programming languages;
- However, if an algorithm can be programmed using one language it might be programmed in any other language;
- If one language  $L_1$  can be mapped into another language  $L_2$  it means that  $L_1$  and  $L_2$  describe exactly the same class of algorithms.

**Note:** a PL with this property is called "Turing-complete".

# Philosophy

- Even though there are many different computational models, the class of algorithms that they describe is unique.
- Whereas each individual computational model has certain arbitrariness to its definition, the underlying class of algorithms it describes is natural because it is the same for other models.

This has profound implications in mathematics.