2.6 Variations on Turing Machines

Before we proceed further with our exposition of Turing Machines as language acceptors, we will consider variations on the basic definition of Slide 10 and discuss, somewhat informally, how these extensions do not affect the computing power of our Turing Machines. The variations however give us more flexibility when we discuss future aspects of Turing Machines.

2.6.1 Turing Machines with Two-Sided Infinite Tapes

The first variation we consider is that of having a Turing Machine with a tape that is infinite on both sides as shown on Slides 45 and 46 (recall that in Def. 1, the tape was bounded on the left). Thus, conceptually, if the tape locations in Def. 1 could be enumerated as

$$l_0, l_1, l_2, \ldots, l_n, \ldots$$

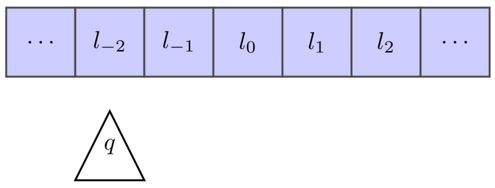
we now can have the enumeration

$$\ldots, l_{-n}, \ldots, l_{-2}, l_{-1}, l_0, l_1, l_2, \ldots, l_n, \ldots$$

It turns out that such an additional feature does not increase the the computational power of our Turing Machine. This fact is stated as Theorem 30 on Slide 45.

Unbounded Tapes on Both Sides

Infinite on Both Sides



Theorem 30. *A Turing Machine with a double-sided infinite tape has equal computational power as a Turing Machine with a single-sided inifinite tape.*

Slide 45

Proof for Theorem 30. We are required to show that:

1. Any Turing Machine with a single-sided infnite tape can be simulated on a two-sided infinite tape Turing Machine.
2. Any two-sided infinite tape Turing Machine can be simulated on a Turing Machine with a single-sided infnite tape.

Double-sided infinite tapes can express any Turing Machine with a single-side infinite tape by leaving the Turing Machine definition unaltered and simply not using the "negative index" side of the tape. If the Turing Machine with a single-side infinite tape relies on *hanging* for its behaviour by moving one step after the leftmost tape location, we can still model this on our double-sided infinite tape Turing Machine. We simply use a new symbol as part of our alphabet (say \$), and place it at location l_{-1} to delimit the end of the tape. We then add a "hanging transition rule" to our Turing Machine (*i.e.*, we transition to a stuck state) whenever we hit this special symbol (so as to model hanging in a single-side infinite tape Turing Machine).

The more involving part of the proof is showing that a Turing Machine with a single-side infinite tape is expressive as a Turing Machine with a double-sided infinite tape. Slide 46 outlines one proposal for an encoding to model a two-sided infinite tape Turing Machine M_D on a single-sided infinite tape. The idea there is to encoded the two-sided infinite tape locations by interspersing the negative indexed locations with their respective positively indexed locations. Thus we encode

$$\ldots,\, l_{-n},\, \ldots,\, l_{-2},\, l_{-1},\, l_0,\, l_1,\, l_2,\, \ldots,\, l_n,\, \ldots$$

as

$$l_0,\, l_{-1},\, l_1,\; l_{-2},\, l_2,\, \ldots,\, l_{-n},\, l_n,\, \ldots$$

The transitions of M_D of the form $(q_1, a) \mapsto (q_2, b)$ (*i.e.*, those that write on the tape) remain (more or less) the same in the new Turing Machine with one-sided infinite tape. The transitions that require most change in our encoding are those of the form $(q_1, a) \mapsto (q_2, \mathbf{R})$ and $(q_1, a) \mapsto (q_2, \mathbf{L})$ *i.e.*, those transitions that move the head of the tape. The changes required are (mainly) of two kinds:

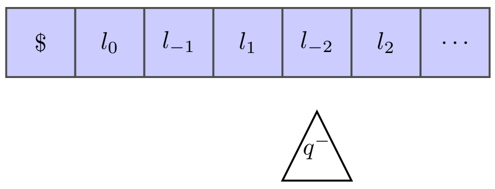
1. we need to model the atomic transitions of moving to the right (*resp.* left) by performing two moves to the right (*resp.* left), skipping the interspersed location of the opposite polarity.

2. when we are in the negative locations, we need to invert the direction of the move ie substitute a right move with (two) left move(s).

We also have to provide for a third aspect, that of changing the polarity of the soluation of our location on the tape whenever, in the Turing Machine being modelled M_D, we move from the positive indices $0, 1, 2, \ldots$ to the negative indices $-1, -2, \ldots$ and viceversa.

Unbounded Tapes on Both Sides (Proof)

Encoding on Infinite on One Side



We augment our set of Turing Machine states with:

- Polarity: q^+ and q^-.
- Intermediary states for two-step moves: $q\mathbf{R}^+$, $q\mathbf{R}^-$, $q\mathbf{L}^+$, $q\mathbf{L}^-$.

Slide 46

In order to handle this new functionality:

1. For every state $q \in Q_{M_D}$ we add the states q^+, q^-, $q\mathbf{R}^+$, $q\mathbf{R}^-$, $q\mathbf{L}^+$, $q\mathbf{L}^-$, $q?^-, q\mathbf{L}?^+$ to the Turing Machine encoding with the one-sided infinite tape.[4] These "replicated" states will be used as follows:
   (a) The polarity on the states, *e.g.,* q^+, $q^-, \ldots$ indicate whether we are in the positive side of the double-sided infinite tape or on the negative side. In particular, if we are in a negative-polarity state, we model left moves as right moves (and viceversa).
   (b) For every state q^+ (*resp.* q^-) we have two *intermediary* states $q\mathbf{R}^+$ and $q\mathbf{L}^+$ (*resp.* $q\mathbf{R}^-$ and $q\mathbf{L}^-$); these intermediary states are used for the required double hops modelling a single atomic (left or right) move in Turing Machine M_D.
   (c) We also have two auxilliary states with ? in them. These states are also intermediary states and they check whether in the case of the modelling (single-sided) Turing Machine, we have reached the end of the tape on the left. In our encoding, this implies that in the doubly-infinite sided Turing Machine being modelled we are about to transition from positive indices to negative ones (or viceversa). Hence, this alerts us to change the polarity of the current state. Note that we only need to perform these checks when we are in a positive-polarity state, *e.g.,* q^+, and we are about to perform the first hop modelling a left move in M_D, and when we are in a negative-polarity state , *e.g.,* q^+, and we are about to perform a second hop modelling a right move in M_D.

2. We define a new alphabet symbol \$ as part of the single-sided infinite tape machine where $\$ \notin \Sigma_{M_D}$. We then shift the tape encoding in the new Turing Machine up by one and place \$ at the leftmost position of the one-sided infinite tape, as shown on Slide 46. Thus our tape encoding now looks like

   $$\$,\, l_0,\, l_{-1},\, l_1,\, l_{-2},\, l_2,\, \ldots,\, l_{-n},\, l_n,\, \ldots$$

   The new symbol acts as an end of tape marker and signals the Turing Machine encoding that it needs to switch the polarity of its state from q^+ to q^- (or from q^- to q^+).

3. For every transition of the form $(q_1, a) \mapsto (q_2, b)$ in the former Turing Machine we have $(q_1^+, a) \mapsto (q_2^+, b)$ and $(q_1^-, a) \mapsto (q_2^-, b)$ *i.e.,* writing for both positive and negative polarity states.

4. For every transition of the form $(q_1, a) \mapsto (q_2, \mathbf{R})$ in the former Turing Machine we have the following:
   - $(q_1^+, a) \mapsto (q_2\mathbf{R}^+, \mathbf{R})$ *i.e.,* the first hop
   - for all $x \in \Sigma_{M_D}$ we add a transition $(q_2\mathbf{R}^+, x) \mapsto (q_2^+, \mathbf{R})$ *i.e.,* the second hop
   - $(q_1^-, a) \mapsto (q_2\mathbf{L}^-, \mathbf{L})$ *i.e.,* the first hop, but in reverse direction
   - for all $x \in \Sigma_{M_D}$ we add a transition $(q_2\mathbf{L}^-, x) \mapsto (q_2?^-, \mathbf{L})$ *i.e.,* the second hop, but in reverse direction. Note however that we move to a ? state, *i.e.,* $q_2?^-$, instead of to state q_2^- directly, as in the case before. This is because moving to the left while in the negative polarity can land us on the location with the \$ symbol.
   - add transition $(q_2?^-, \$) \mapsto (q_2^+, \mathbf{L})$ for the case when we hit the \$ marker (notice the change in polarity) and, for all $x \in \Sigma_{M_D}$, add also the transition $(q_2?^-, x) \mapsto (q_2^-, x)$ for all remaining cases where we do not come across the \$ marker.

5. For every transition of the form $(q_1, a) \mapsto (q_2, \mathbf{L})$ in the former Turing Machine we have the following:
   - $(q_1^+, a) \mapsto (q_2\mathbf{L}?^+, \mathbf{L})$ *i.e.,* the first hop. Note that we move to a ? state, *i.e.,* $q_2\mathbf{L}?^+$, instead of to state $q_2\mathbf{L}^+$ directly because moving left to the intermediary state while in the positive polarity can land us on the location with the \$ symbol.
   - add transition $(q_2\mathbf{L}?^+, \$) \mapsto (q_2\mathbf{R}^-, \mathbf{R})$ for the case when we hit the \$ marker and, for all $x \in \Sigma_{M_D}$, add also the transition $(q_2\mathbf{L}?^+, x) \mapsto (q_2\mathbf{L}^+, x)$ for all remaining cases (when we do *not* hit the \$ marker.)

[4]Note that some of these states may never be used in reality. However, a generic, automated translation inevitably foces us to add redundant states.

- for all $x \in \Sigma_{M_D}$ we add transitions $(q_2\mathbf{L}^+, x) \mapsto (q_2^+, \mathbf{L})$ *i.e.,* the second hop.
- $(q_1^-, a) \mapsto (q_2\mathbf{R}^-, \mathbf{R})$ *i.e.,* the first hop in reverse order.
- for all $x \in \Sigma_{M_D}$ we add transitions $(q_2\mathbf{R}^-, x) \mapsto (q_2^-, \mathbf{R})$ *i.e.,* the second hop in reverse order.

□

2.6.2 Turing Machines with k-Worktapes

Another form of Turing Machine variant we will consider are Turing Machines with fixed, but arbitrary number k of tapes. These variants also have k independent heads (one per tape) and typically use one tape as the input/output tape and use the remaining $k-1$ tapes as a "scratchpad" to record and compute on intermediate computations that are unrelated to the input and the output functions. This facility is appealing because it allows us to write programs more easily. However we will see that, despite their convenience, k-tape Turing Machines do *not* increase the class of functions that can be computed by ordinary Turing Machines.

A k-tape Turing Machine works like a a normal Turing Machine, except that, at each computational step, k-heads independently perform either a move to the left, or a move to the right, or else write at the current tape location. This means that the only thing that changes from Def. 1 is the transition function, as outlined on Slide 47. There, the notation used for the domain can be "expanded" to more familiar notation $Q \times \underbrace{\Sigma \times \ldots \times \Sigma}_{k}$; the same applies for the codomain notation used.

k-tapes Turing Machine

k-tape Transition Function (Partial)

$$\delta_k \ : \quad Q \times \Sigma^k \rightharpoonup (Q \cup \{q_H\}) \times (\Sigma \cup \{\mathbf{L}, \mathbf{R}\})^k$$

Domain: a (non-final) state and k tape symbols pointed to by $k$heads (one per tape).

Co-domain: a state (possibly final) and k instructions (write, move left, move right) for k heads.

Synchronised: heads move/write at the same time.

Everything else from Def. 1 remains the same.

Slide 47

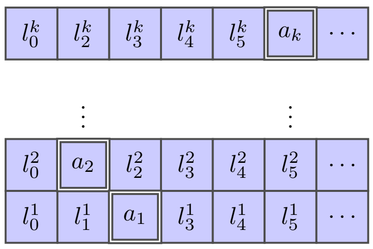
The fact that heads may perform a different action at each state (*i.e.,* either move left, or right or else perform a write at the current location) means that, even though they may start at the same location position on their respective tape, these heads may be at a different location index at different stages of the computation (*cf.* diagram on Slide 48). Because of this, k-tape Turing Machines require an extended notion for configurations, as shown on Slide 48. In this straightforward extension, we record not only the contents of every tape, but also the position of every head.

Despite appearing clumsy initially, we will use k-tapes to structure our Turing Machine programs better in terms of memory access. More generally, k-tape Turing Machines allow us to depart from the sequential memory access of single-tape Turing Machines: we can use the $k-1$ work-tapes as registers for the $k-1$ concurrent access of intermediary results of complex computations. It is important to stress that the recording

k-tapes Turing Machine

k-tape Configurations

$$\langle q,\ x_1\underline{a_1}y_1, \ldots, x_k\underline{a_k}y_k\rangle$$



Slide 48

of intermediary results *can* be performed in a single-tape Turing Machine. This would however require us to record these intermediary results sequentially on the same tape, perhaps separated by some special symbol such as #. This has two main disadvantages:

- when an intermediate results increase in size, we need to shift the other intermediate results recorded after it to the right. This shifting around can sometimes be avoided by introducing "padding" between intermediary results but it is, in general, not possible to know in advance how much padding we need.
- at the end of our computation, we would need to "clean up" the intermediate calculations, so that they do not form part of our output.

k-Tape Turing Machines help us circumvent these problems if we assign a separate tape for every intermediate computation; moreover, the contents of the $k-1$ tapes that are not the input/output tape act as *background tapes* and need not be emptied at the end of the computation.

To be able to establish a proper form of comparison between the computational power of a k-tape Turing Machine and the single-tape counterpart, we need to express what does computation mean in both cases. We have already established this for single-tape Turing Machines in Def. 8 and have already outlined informally how this would be done in a k-tape setting. Def. 31 formalises this on Slide 49. Note that the definition requires that (the encoding of) the input of the function is located on the first tape, and that the other $k-1$ (work)tapes are properly initialised to empty tapes as well. More importantly, Def. 31 only imposes constraints on the first tape for the terminal configuration. This means that we are allowed to leave 'junk' on the work-tapes when we finish computing and thus use the first tape as the input/output tape.

This last point of Def. 31 is crucial, as it streamlines our comparison between the two forms of Turing Machines by limiting simulation up to the contents of the first tape (in the case of the k-tape Turing Machine). Simulation between k-tape Turing Machines and single-tape Turing Machines consists in producing, for every input $\underline{\#}x\#$ a corresponding output $\#y\underline{\#}$ upon halting. In addition, the simulating Turing Machine must not halt whenever the Turing Machine being simulated hangs or diverges for a particular input.

Lemma 33. *Any computation that can be carried out on a single tape Turing Machine can be also performed on a k-tape Turing Machine.*

k-tape Turing Computability

Definition 31. Let M be a k-tape Turing Machine over Σ, and let Σ_1 and Σ_2 be alphabets included in $\Sigma \setminus \{\#\}$. Then M computes a function $f : \Sigma_1^* \to \Sigma_2^*$ if for any $x \in \Sigma_1^*$ such that $f(x) = y$ we have

$$\langle q_0,\ \underbrace{\underline{\#}x\#, \underline{\#}\#, \ldots, \underline{\#}\#}_{k}\rangle \vdash_M^* \langle q_H,\ \#y\underline{\#}, x_2\underline{a_2}y_2, \ldots, x_k\underline{a_k}y_k\rangle$$

If such a Turing Machine M exists, then function f is *k-tape Turing-computable.*

Slide 49

k-tape Turing Computability

Theorem 32. *Any k-tape Turing Machine has equal computational power to a Turing Machine with a single tape.*

Proof. This requires us to show that:

1. Any computation that can be carried out on a single tape Turing Machine can be also performed on a k-tape Turing Machine. (Lemma 33)
2. Any computation that can be carried out on a k-tape Turing Machine can be also performed on a single-tape Turing Machine. (Lemma 34)

□

Slide 50

Proof. The proof is immediate because we can use a k-tape Turing Machine as a single tape Turing Machine and never employ the additional $k-1$ tapes. The modification of the 1-tape transitions into the k-tape transitions is straightforward. □

Simulating a k-tape Turing Machine on a single tape Turing Machine is less trivial. At the heart of this simulation lies an appropriate encoding of the k-tapes and their respective head positions on a single tape with one head whereby for each k-tape transition, the single-tape will attempt to recreate a corresponding configuration (using one or more transitions) according to the encoding.

We choose the following encoding:

- We encode the contents on k tapes as contiguous k tuples on a single tape whereby the tuple i contains the contents at location i for each tape. Since these tuples are always fixed in size, they can easily be represented on a single tape as k-sized blocks of contiguous tape symbols. For example the tuple $\langle a_0^1, a_0^2, \ldots, a_0^k\rangle$ represents the contents at location l_0 for every tape between 1 and k.
- We encode the position of the k tape-heads by extending the size of the tuples to $2k$, meaning that there

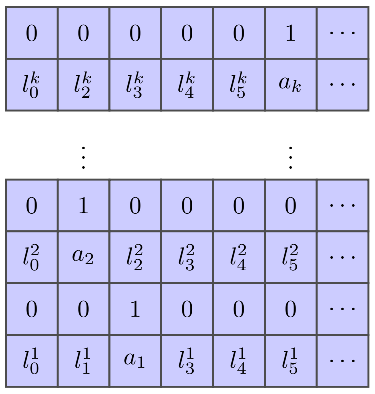
will be two locations dedicated to every location in the k-tape Turing Machine. The first location of the encoding will record the contents of the respective location as already stated. The second location will act as a boolean value recording whether the head of the tape is at the current location or not. Extending our previous example, the tuple $\langle a_0^1,\ 0,\ a_0^2,\ 1,\ ,a_0^3,\ 0, \ldots, a_0^k,\ 1\rangle$ represents the contents at location l_0 for every tape between 1 and k whereby, in the case of tape 2 and tape k, the head is also at position 0. This is shown pictorially on Slide 51 (for some other k-tape configuration).

- Since we are free to choose any alphabet for our single-tape Turing Machine M simulating the k-tape Turing Machine, we chose the following:

$$\Sigma_M = \Sigma \cup (\Sigma \times \{0,1\})^k \cup \{\$\}$$

  The key elements in this alphabet are the new symbols in the subset $(\Sigma \times \{0,1\})^k$: there we define new symbols, each representing a unique tuple of the form $\langle a_0^1,\ 0,\ a_0^2,\ 1,\ ,a_0^3,\ 0, \ldots, a_0^k,\ 1\rangle$ discussed earlier. Note that this subset, although large, is finite because the alphabet of the the k-tape Turing Machine, Σ, is also finite. We choose to include Σ as well in Σ_M in order to mimic the contents of the input/output tape of M_k directly (rather than with an encoding of it). The final symbol is \$ and is called the *end-marker*. It is written at the leftmost location on M's tape and allows us to realise when we have reached the end of the tape.

k-tape Encoding Turing Machine



0 0 0 0 0 1 ···
l_0^k l_2^k l_3^k l_4^k l_5^k a_k ···
⋮ ⋮
0 1 0 0 0 0 ···
l_0^2 a_2 l_2^2 l_3^2 l_4^2 l_5^2 ···
0 0 1 0 0 0 ···
l_0^1 l_1^1 a_1 l_3^1 l_4^1 l_5^1 ···

Slide 51

Lemma 34. *Any computation that can be carried out on a k-tape Turing Machine can be also performed on a single-tape Turing Machine such that:*

- *whenever the computation below occurs in M_k:*

$$\langle q_0,\ \underbrace{\underline{\#}x\#, \underline{\#}\#, \ldots, \underline{\#}\#}_{k}\rangle \vdash_M^* \langle h,\ x_1\underline{a_1}y_1, x_2\underline{a_2}y_2, \ldots, x_k\underline{a_k}y_k\rangle$$

  then the following computation occurs in M:

$$\langle q_0,\ \underline{\#}x\#\rangle \vdash_M^* \langle h,\ x_1\underline{a_1}y_1\rangle$$

- *whenever M_k rejects input x, then M rejects it as well.*

Proof. Central to this proof is the encoding of the memory (the k-tapes) of M_k in M, as discussed above. Thus, M performs the following procedures:

Initialisation: Assume the input string x has length n. M rewrites its tape contents to a representation of the first n tape locations of the k-tapes using the encoding discussed above. Note that the encoding assumes that the contents on the tapes $2..k$ are empty, thus this initialisation phase is just a function dependent on the input string x. Note also that in the resulting initialised string in M, beyond the n initialised locations, the tape contents are blank. The procedure can be decomposed as follows:

1. move the tape head to the rightmost location of the string) *i.e.,* until we hit a #symbol on the tape.
2. Start moving towards the left and, for each content on the tape at location i, write the corresponding encoding of that symbol (and the assumed $k-1$ blank symbols on the other tapes) to location $i+1$. Shifting the string by one location leaves a space at the leftmost position where to write the special symbol \$. At the end of this process, we should obtain an encoding of the k-strings and heads
$$\underbrace{\underline{\#}x\#,\underline{\#}\#,\ldots,\underline{\#}\#}_{k}$$
as a single string, shifted by one location to the right.
3. write \$ at the leftmost tape location, *i.e.,* l_0.

Simulation: The main idea of the simulation procedure is that the simultaneous operations of the k-heads of M_k (*i.e.,* write, move left or move right) can be sequentialised, by doing what each tape does in order. The procedure can be broken down as follows:

1. We first need to know what are the contents of the locations pointed to by the heads in our encoding. This requires us to start at the leftmost position and scan the entire contents of the encoded string, updating to some "intermediate" state at each location scan that tells us the contents of some location where the respective head is *i.e.,* whenever we find a tuple with 1 markings.
2. At the end of this scan, we should have the knowledge of all the current location contents pointed to by the (encoding of the) $k - heads$. Thus we can consult the transition function of the original k-tape Turing Machine so as to determine what next step to take (in terms of head operations (write, move left or move right for k-heads and the next state to transition to, say q).
3. move to the head to the leftmost position (*i.e.,* till you hit the \$ symbol) and start making the head changed to the encoding at the respective locations as dictated by the transition function of the original k-tape Turing Machine in sequential fashion.
4. When finished, move to the new state q, as dictated by the transition function of the original k-tape Turing Machine.

Halting: When M_k is due to halt, M performs the following procedure to mimic the output tape of M_k (the contents of the other tapes do not matter):

1. It scans the contents of the tape, noting where the head of the first tape currently is.
2. it rewrites the contents of the tape, "decoding" every location to hold the symbol at tape 1, *i.e.,* the input/output tape using only symbols from Σ.
3. it moves the head to the position noted in step 1.

□

***k*-tape Simulation**

Initialisation Encode the input in the appropriate format.

Simulation Replicate simultaneous operations in sequential fashion on the encoding.

Halting Decode and move head to appropriate position.

Slide 52

2.6.3 Non-Deterministic Turing Machines

Non-deterministic Turing Machines are normal Turing Machines as in Def. 1 with the only difference being that the transition function may give us a number of possibilities that we can chose for the next step to take. This makes execution *non-deterministic* (because we have this internal choice that we can make) and our Turing Machine programs a relation rather than a function. Mathematically, this is expressed by having the transition functions of non-deterministic Turing Machines be *total mappings to the powerset of possible state and tape operation*, as shown in Slide 53.

Non-Deterministic Turing Machine

Transition Function (Total)

$$\delta \ : \quad Q \times \Sigma \ \rightarrow \ \mathcal{P}((Q \cup \{q_H\}) \times (\Sigma \cup \{\mathbf{L}, \mathbf{R}\}))$$

Domain: a (non-final) state and a tape symbols pointed to by the head.

Co-domain: a *set of* states (possibly final) and instructions (write, move left, move right) for the head.

Everything else from Def. 1 remains the same.

Slide 53

Non-determinism poses a conundrum in terms of language acceptance. This is because, for any particular string x, it may well be the case that a non-deterministic Turing Machine ends up both accepting *and* rejecting x (and also diverging on x for that matter). Thus we modify the definition of acceptance slightly for non-deterministic Turing Machines as in Def. 35, shown on Slide 54. The new acceptance is based on the *existence* of an accepting execution path. This is sometimes refered to as the *angelic* interpretation of non-determinism; broadly speaking, this means that when the computation is faced with a choice then, whenever available, the (right) one leading to a successful termination (halting) is chosen. With angelic non-determinism, rejection becomes hard to show for non-deterministic Turing Machines (see Slide 55) because we require that for a particular string x:

- *all* computation sequences eventually terminate (they do not diverge).

- *all* terminal configurations hang.*i.e.,* reject.

Acceptance in NTMs

Reachability is not a function anymore. Thus, for any x, we can have acceptance

$$\langle q_0, \underline{\#}x\#\rangle \vdash_M^* \langle q_H, y_1\underline{a}y_2\rangle$$

but also rejection or divergence.

Angelic Interpretation

Definition 35. For any non-deterministic Turing Machine M and string $x \in \Sigma^*$ M accepts x iff there *exists* a computation sequence

$$\langle q_0, \underline{\#}x\#\rangle \vdash_M^* \langle q_H, y_1\underline{a}y_2\rangle$$

for some $a \in \Sigma$, $y_1, y_2 \in \Sigma^*$

Slide 54

Rejection in NTMs

Rejection is harder to show in a non-deterministic setting.

Angelic Interpretation

Definition 36. For any non-deterministic Turing Machine M and string $x \in \Sigma^*$ M rejects x iff *for all* a computation sequences we have

$$\langle q_0, \underline{\#}x\#\rangle \vdash_M^* \langle q, y_1\underline{a}y_2\rangle$$

for some $a \in \Sigma$, $y_1, y_2 \in \Sigma^*$, $q \in Q$ (*i.e.,* $q \neq q_H$)

Slide 55

Turing Machines are as expressive as their non-deterministic counterparts.

Lemma 38. *Any computation that can be carried out on a deterministic Turing Machine can be also performed on a non-deterministic Turing Machine.*

Proof. The proof is immediate because every deterministic Turing Machine can be expressed as a non-deterministic by changing the (partial) transition function of the former into one of the latter with a singleton set for transitions that exist and empty sets for undefined transitions. □

Proving the other way round is a little bit more involving and uses a technique called *Dove-tailing.* This means that we simulate multiple executions of the same Turing Machine at the same time, interleaving their executions as a sequential single execution. Using this technique, if *one* execution accepts, then the

Non-Deterministic Turing Computability

Theorem 37. *Any non-determinismtic Turing Machine has equal computational power to a deterministic Turing Machine.*

Proof. This requires us to show that:

1. Any computation that can be carried out on a deterministic Turing Machine can be also performed on a non-deterministic Turing Machine. (Lemma 38)
2. Any computation that can be carried out on a non-deterministic Turing Machine can be also performed on a deterministic Turing Machine. (Lemma 40)

□

Slide 56

entire execution accepts and logically, if *all* executions hang, the entire execution hangs. Alternatively, one can think of the set of possible execution paths as a tree with the root being the start configuration, and dove-tailing being a breadth-first search along this tree searching for an accepting configuration.

Lemma 40. *Any computation that can be carried out on a non-deterministic Turing Machine can be also performed on a deterministic Turing Machine.*

(Outline). At the heart of the dove-tailing simulation of a non-deterministic Turing Machine lies an encoding once again. This time, the encoding concerns the possible configurations the non-deterministic Turing Machine can be in after k reduction steps. Thus, the alphabet of the simulating deterministic Turing Machine M is

$$\Sigma_M = \Sigma \cup Q \cup \{\texttt{:}, \texttt{>}, \texttt{*}\}$$

and we use this extended alphabet to record a configuration $\langle q,\, x\underline{a}y\rangle$ on tape as the string $q\texttt{:}\ x\texttt{>}ay$. The third special alphabet symbol `*` is used as a delimiter between multiple configurations written on tape *i.e.,* we can represent the list of configurations $\langle q_1,\, x_1\underline{a_1}y_1\rangle, \ldots, \langle q_n,\, x_n\underline{a_n}y_n\rangle$ as the string

$$q_1\texttt{:}\ x_1\texttt{>}a_1y_1\texttt{*}\ldots\texttt{*}q_n\texttt{:}\ x_n\texttt{>}a_ny_n$$

. Note that since Q is finite, then Σ_M is also finite.

For convenience, our simulating deterministic Turing Machine uses three tapes we call R and S, with R being the input/output tape and also the "main memory" tape and S being our scratchpad tape. The simulation proceeds as follows:

1. Initialise by taking the input on R and rewriting it in the encoding discussed above.
2. read the current configuration (*i.e.,* read and move left till you hit either `*` or #).
3. if the current state is the success state,
    - then (we have found a success configuration and so) terminate by decoding the configuration and writing it on R using the alphabet of the non-deterministic machine, erasing everything else and move the head to the appropriate position (as indicated by the encoded configuration last read).

Non-Deterministic Computability Implies Deterministic Computability

Lemma 39. *Any computation that can be carried out on a non-deterministic Turing Machine can be also performed on two-tape deterministic Turing Machine.*

Proof. Perform a breadth-first search along the tree of possible execution paths:

extended alphabet: $\Sigma_M = \Sigma \cup Q \cup \{:, >, *\}$

Encode Configurations: $\langle q, x\underline{a}y\rangle$ as $q\colon x{>}ay$.

2nd tape as scratchpad: Keep all reachable configurations in i computational steps as the list $q_1\colon x_1{>}a_1y_1 * \ldots * q_n\colon x_n{>}a_ny_n$

□

Slide 57

- else consult the transition function of the non-deterministic Turing Machine and append to the string on tape S a list encoding of all the possible configurations that we can transition to, separating each possible configuration in the list of options by $*$.

4. if the symbol in the next location on tape R is $*$
   - then (we have more configurations reachable in i steps to process and so) goto step 2.
   - else if the next symbol is #, it means that we have processed all possible configurations reachable from the start configuration in i steps. We can therefore proceed to process all reachable configurations in $i+1$ steps. All these configurations are on tape S. We first clear tape R. Then, if S is empty
     - then it means that there are *no* reachable configurations in $i+1$ steps. We simply decode the last configuration from round i, write it to tape R and terminate (not in a success state).
     - else we copy the contents of tape S to tape R, erase the contents of tape S, and go to step 2.

□

2.6.4 Exercises

1. Outline how you would convert transitions on a single-tape Turing Machine to simulate the same computation on a k-tape Turing Machine.

2. Assume that you have k Turing Machines $M_1 \ldots M_k$ accepting languages $L_1 \ldots L_k$ respectively. Using this information, outline how you would build a k-tape Turing Machine recognising the language $\bigcup_{i=1}^{k} L_i$. (*Hint:* attempting to accept a string $x \in L_i$ using some M_j where $i \neq j$ may result in M_j diverging).