Problem 1.

Prove that the class of Turing-acceptable languages is closed under union, intersection, and reversal. For each property, give a detailed sketch of the proof, by saying how you would build a Turing machine that accepts the resulting language, given the Turing machine(s) that accept the original language(s).

Solution

Union

Proof. Let L_1, L_2 be Turing-acceptable languages, then define their corresponding Turing machines as M_1, M_2 respectively.
We will then define a new machine M_U which operates on the union of Q, Σ for M_1, M_2. We will distinguish each state by the source machine, thus $Q_1 \cap Q_2 = \emptyset$. The initial state of M_U will be the initial state of M_1.
On the input of a string w, M_U will behave exactly as M_1. If we reach a halting and accepting state of M_1, then M_U will also halt and accept. Otherwise, rather than halting and accepting, M_U will then run M_2 on w, once M_2 halts, M_U accepts if M_2 does, and rejects otherwise. Note that M_I will halt provided that both M_1 and M_2 each halt.
Now assume $w \in L_1 \cup L_2$, then either $w \in L_1$ or $w \in L_2$, if $w \in L_1$ w will be accepted by M_U after the initial run of M_1 (as M_1 accepts w). Otherwise if $w \in L_2$, then after M_U runs M_1 where M_1 rejects w, M_U will continue with M_2 which accepts w and thus so does M_U.
If $w \notin L_1 \cup L_2$ then $w \notin L_1$ thus M_U continues onto M_2, but as $w \notin L_2$, L_2 will reject w and thus so will M_U.

□

Intersection

Proof. Let L_1, L_2 be Turing-acceptable languages, then define their corresponding Turing machines as M_1, M_2 respectively.
We will then define a new machine M_I which operates on the union of Q, Σ for M_1, M_2. We will distinguish each state by the source machine, thus $Q_1 \cap Q_2 = \emptyset$. The initial state of M_U will be the initial state of M_1.
On the input of a string w, M_I will behave exactly as M_1. If we reach a halting and rejecting state of M_1, then M_I will halt and reject, if M_1 halts and accepts, M_I will continue on and run M_2 on w. If M_2 accepts w, then so will M_I otherwise w is rejected by M_I. Note that M_I will halt provided that both M_1 and M_2 each halt.

Now assume $w \in L_1 \cap L_2$, then $w \in L_1$ and $w \in L_2$, because of this, w will run through both stages of M_I and pass each, thus M_I clearly accepts w. However, if $w \notin L_1 \cap L_2$, then $w \notin L_1$ (in which case w is rejected in the first stage of M_I) or $w \notin L_2$ (in which case w is rejected by the second stage of M_I). In either case, w will be rejected by M_I, therefore M_I represents $L_1 \cap L_2$. □

Reversal

Proof. Let L be a Turing-acceptable language, then define a corresponding Turing machine M which. Define a new Turing machine M_R using the same alphabet, and set of states as M. However let the initial state of M_R be the final accepting state of M, similarly, let the final state of M_R be the initial state of M.

We can then run M backwards on w (note we are starting on the final state), if we can then reach the initial state of M we have successfully read the reverse of a string $w \in L$, thus the initial state of M will be the final accepting state of M_R. If running M backwards halts but does not accept, then M_R should reject the input. In this sense, the δ function of M_R is the inverse of the δ function for M.

To show this accepts reversal, assume $w \in L$ then M accepts w and $w^R \in L^R$. Running M_R on w^R is equivalent to running w on M, which accepts w thus M_R accepts w^R. If $w \notin L$ then M does not accept w, thus $w^R \notin L^R$. Running M^R on w^R is again equivalent to running w on M, however as $w \notin L$, we know that M will not reach an accepting state, therefore running δ backwards from the accepting state cannot possibly reach the initial state (the final accepting state of M^R) therefore M^R will not accept w^R. □

Problem 2.

Prove or disprove that the set of Turing-acceptable languages is closed under concatenation.

Solution

Proof. Let L_1, L_2 be Turing-acceptable languages, then define their corresponding Turing machines as M_1, M_2 respectively.
We will then define a new machine M_C which operates on the union of Q, Σ for M_1, M_2. We will distinguish each state by the source machine, thus $Q_1 \cap Q_2 = \emptyset$. The initial state of M_U will be the initial state of M_1. We will also include a set of 'marked' characters in our alphabet, which correspond each possible state of $Q_1 \times \Sigma_1$ (note this additional set of characters is finite), these are unique from other characters in our alphabet.
Our new machine M_C will first run M_1 until it reaches any arbitrary accepting state, note M_1 need not have halted. M_C then marks on the tape with one of the 'marked' symbols which corresponds to the state of M_1 and the current character at that point. We know the string up to this marked point is accepted by M_1 thus we then run M_2 on the remainder of the string, if M_2 accepts the remainder, then we halt and accept the string. If M_2 does not accept the remainder, we enter a new set of states which take us back to the marked location, rewrites the original character (remember that it is encoded into our marked character), restores the state of M_1 and repeats the process until the next accepting state of M_1 is reached. If we continue and run out of characters in the string for which M_1 could accept, we halt and reject the string.
By performing this process, M_C divides an input string w into two substrings $w = xy$, where x is accepted by M_1, the machine then checks if y is accepted by M_2 if so then w must be in the concatenation by definition. If not, we grow x until it is again accepted by M_1, and perform our check once again. Clearly then if w is in the concatenation of L_1, L_2 M_C will accept it by testing every possible combination until a division is found that works. If w is not in the concatenation of L_1, L_2, then we must eventually reach a point where no more strings can be accepted by M_1 (we are trying to feed a string longer than our initial string into M_1) at this point our machine will halt and reject the string.

□

Problem 3.

Consider a new type of *deterministic* machine, having one read-only input tape and two stacks. The tape is read-only, it cannot be written, but the head can move left, right, or do nothing. Each stack operates (independently of the other) as in a deterministic pushdown automaton.

$$M = (K, \Sigma, \Gamma_1, \Gamma_2, z_1, z_2, \delta, s)$$

where K is a finite set of states, Σ is a finite input alphabet, Γ_1 and Γ_2 are two finite stack alphabets, $z_1 \in \Gamma_1$ and $z_2 \in \Gamma_2$ are the initial symbols for the two stacks, $s \in K$ is the initial state. h is a special halting state not in K, just like a Turing machine.

(a) Give an appropriate definition for the transition function δ, for a configuration of this machine, for the "yields in one step" operator, and for the language accepted by this machine.

(b) These machines can accept the same languages as a class of automata you already know: deterministic pushdown automata, pushdown automata, or Turing machines? Prove your answer formally.

Solution

Part (a)

The δ function on input will know it's current state, the current character under it's head, as well as the character at the top of each of it's stacks. On output it will produce either a new state or halting state, the head can move left/right/stay, and it can push/pop a character onto/off either of it's stacks. Note this transition function is basically the transition function of NPDA glued together with the transition function of Turing machines, with the exception of the absence of the output character which the Turing machine writes back to the stack.

$$\delta : (K \times \Sigma \times \Gamma_1 \times \Gamma_2) \to ((K \cup \{h\}) \times \{L, R, \epsilon\} \times \Gamma_1^* \times \Gamma_2^*)$$

Part (b)

This machine is equivalent to a Turing machine.

Proof. Let T be a Turing machine which represents some language L. We will construct an equivalent machine M using our new type of machine to T. Let Σ_T be the alphabet used by the Turing machine. Then our machine M will have $\Sigma = \Sigma_T, \Gamma_1 = \Sigma_T, \Gamma_2 = \Sigma_T$, that is both our tape and both stacks have the same alphabet as the Turing machine tape. Our set of states will be equivalent to those of the Turing machine, with the exception of two additional setup states which I will denote S_M and S_C. S_M will serve as the initial state of M.

SETUP The tape for machine M is initialized to the same state as the Turing machine T. When in state S_M, the delta function will move the head to the left (remember that the machine starts at the end of the input) and will repeat this process until it reaches either the end of the tape or the first input character (if this is an infinite tape, we could overstep, read nothing, then step back). The machine then enters state S_C, in which it will read a single character, push that character on to it's first stack, then move one character to the right. Once it reads an empty character (has reached the end of its input) it enters the initial state of the Turing machine.

CLAIM Stack 1 represents characters to the left of the head in the Turing machine, and Stack 2 represents characters to the right (in the initial state of the Turing machine). This is clear to see, since we read from left to right while pushing onto the stack, Stack 1 must necessarily the characters to the right of the Turing machine head (with nearest being closer to the top of the stack), the Turing machine has nothing to its right therefore Stack 2 (which is empty save z_2) is equivalent. Note that the top of Stack 2 is considered by my convention to by the location of the head.
For the following operations on Turing machine T, I will define an operation on δ which I claim will be equivalent. In all cases, machine M enters it's equivalent state that T does. Note that for move operations if we have an empty stack (save z_i) we can ignore the pop and for what I describe as the "same value" below, instead use the empty character for the corresponding push. Also note that once we have read the tape, because of the fact that our stack now represents our tape, we never have to move the actual head we will instead simulate with the stacks as shown below.

T **WRITES TO HEAD** M pops from Stack 2 and pushes the write value to Stack 2. Note that Stack 1 does not change, the Turing machine does not move, therefore the left of the head does not change. The Turing machine has a new value at it's head. By convention, this is top of Stack 2. When the top is removed, the value overwritten is removed now the item to the right of the head is at the top of the stack. After writing the top of Stack 2 now points to the new written value the same at the head of the Turing machine. Thus these two operations are equivalent.

T **MOVES LEFT** M pops from Stack 1 and pushes the same value to Stack 2.
The Turing machine has one less character to the left of it, what was 2 characters away is now 1 character away. After popping from Stack 1, M has one less character to its left and what was 2 characters away is now 1 character away.
What was under the Turing machine head is now to its right, what was to the left of the Turing machine is now under the head. After pushing the head of Stack 1, what was to the left of M is now top of Stack 2 (that is, underneath the head of M). Further what was at the top of stack 2 is now one character away (as we pushed a new value).

T **MOVES RIGHT** M pops from Stack 2 and pushes the same value to Stack 1.
Same as above, one less character to the right in T, where popping from Stack 2 corresponds in one less character to the right in M. Pushing the same value to Stack 1 is equivalent to having the character under the head now to the left of T after moving. Again, note that if Stack 2 only contains z_2 do not pop from Stack 2, instead push the empty character onto Stack 1. By the above it is clear that this will be equivalent, as an empty stack implies that there was nothing (infinitely many empty characters) to the right of the head.

T **HALTS AND ACCEPTS/REJECTS** M also halts and accepts/rejects (same as T)
T and M are in equivalent states, thus when T halts, M can therefore also halt with the same result as T.

Therefore in all situations, the machine we have defined represents the tape of T through use of its two stacks. Further, after completing its setup, M will always be in an equivalent state to T. Therefore, given the same input both T and M will perform the same operations and produce the same result. With the only difference being that M is using stacks to represent the tape (initialized from its actual tape) where as T can perform all its operations directly on the actual tape.

$\square$