CSE 200 Spring 2008 Homework 1 answer key

Kirill Levchenko

April 30, 2008

Problem 1: Turing Machine with Left Reset

We will show how a left-reset Turing machine (LRTM) can simulate the move-left operation of a standard Turning machine (TM). The idea is to intercept the move-left commands and provide a way for the LRTM machine to reposition it's tape head one tape cell to the left of it's previous position. To do this, the LRTM will shift the entire tape to the right one cell and reposition itself at its previous position, which is now contains the contents of the cell to the left. We augment the tape alphabet Γ of the TM with a mark and a special "blank" symbol # which we assume is not in the original alphabet Γ, so that the tape alphabet of the LRTM is

$$\Gamma' = \{\textsc{mark}, \textsc{none}\} \times \Gamma \cup \{\#\}.$$

To move one cell left, the LRTM marks it's current position as MARK and resets to the start of tape. It then copies the tape one position to the right by writing # in the first position and copying its symbol to the second cell, the contents of the second cell to the third cell, and so on. When it encounters the mark, it *leaves it in it's current position*; the cell at that position now contain the contents of the cell one position to the left of where the mark as placed. The LRTM then resets again and advances to the marked cell. It is easy to verify that, ignoring the special # symbols at the front of the tape and the mark, the LRTM is now in the same configuration as if the head had moved one position left. When moving right, the LRTM erases the mark and moves one position right normally.

Note that simulating each move-left operation takes $O(s_0 + s_1)$ steps, where s_0 is the number of #-blanks written at the front of the tape, and s_1 is the number of cells used by the TM. If a TM takes time $t(n)$ to recognize a word, it uses at most $t(n)$ tape cells, and does at most $t(n)$ move-left operations, so both s_0 and s_1 are upper bounded by $t(n)$. Thus, each move-left operation takes $O(t(n))$ time. There are at most $t(n)$ move-left operations, so the running time of the LRTM is $O(t^2(n))$. For polynomial $t(n)$, $O(t^2(n))$ is also polynomial.

To simulate an LRTM on a TM, we use a mark to mark the first cell on the tape so that we can simulate a reset-left operation by simply moving left until we reach the mark. Each reset-left operation requires moving left, which

takes time linear in the length of the used portion of the tape. If a LRTM takes time $t(n)$ to recognize a word, it uses at most $t(n)$ tape cells, so each step of the LRTM takes at most $t(n)$ time on the TM, for a $O(t^2(n))$ total time. For polynomial $t(n)$, $O(t^2(n))$ is also polynomial.

Problem 2: Addition on One-Tape TMs

To decide the language on a one-tape TM, we can do standard binary addition using a three-tape TM by copying x to the second tape and y to third tape, and then verifying the answer z on the first tape. This takes linear time. As shown in class, any multi-tape TM can be emulated in $O(n^2)$ time by a one-tape machine.

To prove the lower bound, we will use the $\Omega(n^2)$ lower bound proven in class for the language

$$L_{\text{EQ}} = \{x\#x \mid x \in \Sigma^*\}$$

To show that L takes $\Omega(n^2)$ time on a one-tape TM, we will show how a Turing machine that decides L can be used to decide L_{EQ}. On input $x\#y$, our machine will prepend the string "0+" to the beginning of the input (the input will need to be shifted right two cells, which can be done in linear time). It then replaces # with =, moves to the left end of the tape (which can be found because it contains "0+"), and starts the machine to decide L. Note that the time taken by our machine is $O(n)$ for pre-processing plus the time taken to decide L, say t. From our lower bound, we know that altogether our machine must take $\Omega(n^2)$, so $t \in \Omega(n^2)$.

Problem 3: FP Bit-by-Bit

Note. Let L be the language described in the problem. That is,

$$L_f = \{(x, i, b) \mid i \leq |f(x)| \text{ and } \big(f(x)\big)_i = b\}$$

The forward direction ($f \in \mathbf{FP} \Rightarrow L_f \in \mathbf{P}$) is easy. Given input (x, i, b), first compute $y = f(x)$. This takes time polynomial in $|x|$ and therefore polynomial in the input (x, i, b). Also, we know that $|y|$ must be polynomial in $|x|$ because a polynomial-time computation can only produce polynomial-size output. Now, check if $i \leq |y|$, and finally if $y_i = b$. This takes linear time in the RAM model.

The converse ($L_f \in \mathbf{P} \Rightarrow f \in \mathbf{FP}$) requires us to compute $y = f(x)$ in polynomial time using a machine M that decides L_f. We will use this machine as a subroutine. The following machine computes $f(x)$ on input x. We give the description in a slightly higher-level pseudo-code, which can be translated to RAM machine operations in straightforward manner. We write ϵ to mean the empty string and $\circ$ to be the string concatenation operator.

1. $i \leftarrow 0$
2. $y \leftarrow \epsilon$
3. **loop**
4. $\quad i \leftarrow i + 1$
5. $\quad$ **if** $M(x, i, 0)$ accepts **then**
6. $\quad\quad y \leftarrow 0 \circ y$
7. $\quad$ **else if** $M(x, i, 1)$ accepts **then**
8. $\quad\quad y \leftarrow 1 \circ y$
9. $\quad$ **else**
10. $\quad\quad$ output y
11. $\quad$ **end if**
12. **end loop**

The result y is built up one bit a time by by running the decider machine to determine if the i-th bit of y is 0 or 1. The i-th bit of y is the i-th bit of $f(x)$, by definition of L_f, and the length of y is exactly $|f(x)|$. Since $|f(x)|$ is polynomial in $|x|$, i is $O(\log |x|)$, so each invocation of M takes time polynomial in $|x|$. We execute the loop at most $2|f(x)|+2$ times, so the machine described above runs in time polynomial in $|x|$ as desired.

Problem 4: Mystery Language

Let L be the language given in the problem statement. We first show that L is not recursive by reducing HALT, the halting language (p. 188 in the textbook), to L. We will reduce HALT to L via a mapping reduction (See Sec. 5.3 in textbook), although for decidability we could also use a Turing reduction. The mapping reduction requires that we transform an an instance (M, ω) of the halting problem to an instance (M', ω') for a (hypothetical) machine that decides L, so that

$$(M, \omega) \in \text{HALT} \;\Leftrightarrow\; (M', \omega') \in L \tag{1}$$

Given M and ω the reduction constructs the following machine M':

1. Ignore input y
2. Run M on ω
3. **if** M accepts **then**
4. $\quad$ output 1
5. **else**
6. $\quad$ output 0
7. **end if**

Also set $\omega' = 1$. We need to show that (1) holds. Consider $(M, \omega) \in$ HALT. Since M accepts ω, the machine M will also halt and output 1 if run long enough. Thus, there must be some y (which depends on M and ω) so that after $100|y|$ steps M' will halt and output $1 = \omega'$. Thus $(M', \omega') \in L$ only if M halts on ω. To show the other direction, note that if M' accepts ω after some time $100|y|$, then M must accept ω, so $(M, \omega) \in$ HALT. Thus, the language L is not recursive.

We claim that L is recursively enumerable. We will show this directly. The following machine M'' recognizes L:

1. **for** $y \in \Sigma^*$ **do**
2. Run M on y for $100|y|$ steps
3. **if** M halts and outputs ω **then**
4. accept
5. **end if**
6. **end for**

Machine M'' tries all possible y (e.g., in lexicographic order); it accepts if only if there is an input y for which M halts in $100|y|$ steps and outputs ω, as desired. We could have also reduced L to HALT using a mapping reduction (but *not* a Turing reduction) by constructing M'' from M in the reduction itself, which would show L is recursively enumerable by Theorem 5.28 (p. 209 in textbook).

Since L is recursively enumerable but not recursive, by Theorem 4.22 (p. 181 in textbook), it can't be co-recursively enumerable.

Problem 5: Neither RE nor coRE

Theorem 5.30 (p. 210) in the textbook gives such a language. Here is another way to solve the problem.

Let HALT be the halting language (p. 188 in the textbook). We know that HALT $\notin$ **REC**, in other words, HALT is undecidable. Theorem 4.22 (p. 181) tells us that an language is in **REC** if and only if it is in **RE** and **coRE**. Since HALT is recursive but not decidable, it follows that HALT $\notin$ **coRE**. We also know that the complement of HALT (denoted $\overline{\text{HALT}}$) is not decidable, since **REC** is closed under complement. By the same argument as above, $\overline{\text{HALT}} \notin$ **RE**. To construct our language that is neither in **RE** nor in **coRE**, we take the disjoint union of HALT and $\overline{\text{HALT}}$:

$$L = \{0\} \times \text{HALT} \cup \{1\} \times \overline{\text{HALT}}.$$

To show that L is neither **RE** nor **coRE**, we reduce HALT $\leq_{\text{m}} L$ and $\overline{\text{HALT}} \leq_{\text{m}} L$ in the obvious way. Since HALT $\notin$ **coRE** and $\overline{\text{HALT}} \notin$ **RE**, by Corollary 5.29 (p. 210) in the textbook, it follows that L is neither recursively enumerable not co-recursively enumerable, as desired.