CSE491/596, Fall 2020 Problem Set 4 Due Mon. Oct. 26, 11:59pm
Plus topics for presentations on Oct. 22–23

Reading: Picking up Debray's notes in section 9, Monday's lecture will pick up the example in Theorem 9.1 but relate it to ALL_{TM}. *Skim* the discussion of Post's Correspondence Problem—it arises in a different way as a presentation option below. The first part of section 10, Theorem 10.2, will be covered later in-tandem with Theorem 13.12 when we begin complexity theory. **Skip** page 30 with the subject of "oracles"—IMHO it has limited value until late in the complexity theory unit where *binary search* becomes a quintessential example of an oracle process. But page 31 picks up with useful examples of the kind also slated for the Monday 10/19 lecture, when also Rice's Theorem will be proved by applying the all-or-nothing switch idea as a *filter*, pretty much as Debray does it on page 32. From there until the end of section 12, the material becomes *presentation options* as detailed further below. This all will enable us to connect section 13 to what has been going on in sections 8–10. Likewise the first half of section 14—but stop before Theorem 14.9 on page 46 of the notes.

In brief, the reading is through Rice's Theorem by Monday and through page 45 (jumping over skim/skip parts) by Friday of next week. *Presentation options are after the problems.*

(1) Prove by reduction from A_{TM} (or K_{TM}) that the following decision problem is undecidable:

If-Else

Instance: A program P in Java (or some other high-level programming language) and a particular if-then-else statement S in P.

Question: Does there exist an input x such that when $P(x)$ is run, there is a point in the computation where the "else" branch is taken?

Important note: it is assumed that the condition between `if` and `then` is decidable—i.e., that the if-else statement itself always exits, as required of a flowchart diamond.

Also answer: is the language of the problem c.e.? Justify briefly. (12+6 = 18 pts.)

(2) With reference to Problem (1), suppose we are given a program P such that we know in advance that a particular one of the following conditions holds. For each of the conditions, say whether knowing about it in advance and being able to assume it enables giving a definite yes/no answer for the If-Else problem about that particular P.

(a) For all inputs x, the computation reaches the statement S at least once.

(b) For all inputs x, the computation reaches the statement S exactly once, whereupon both the "yes" branch and the "no" branch halt after one more statement.

(c) On input $x = 10010110$ (150 in binary), the computation reaches the statement S at least once, and we only need to decide whether *that* computation ever takes the "else" branch.

(d) On input $x = 10010110$ (150 in binary), the computation reaches the statement S exactly once, and we only need to decide whether *that* computation takes the "else" branch.

In each case, if the answer is *no*, then you will be able to do the reduction in problem (1) in a way that the mapped programs $P = P_{M,w}$ also always satisfy the extra condition—regardless of whether M accepts w or not. (Perhaps you have already done so.) If you say *yes* to any condition, justify why. (24 pts. total)

(3) Prove via the "all-or-nothing switch" that the following two problems are undecidable:

PAL1

INSTANCE: A deterministic Turing machine with input alphabet $\Sigma = \{0, 1\}$.

QUESTION: Is there a palindrome x such that M accepts x?

PAL2

INSTANCE: A deterministic Turing machine with input alphabet $\Sigma = \{0, 1\}$.

QUESTION: Does M accept all palindromes?

Also say which of these languages is c.e., if any. (24 pts. total)

(4) Recall that PAL stands for the language of palindromes over the alphabet $\Sigma = \{0, 1\}$. Prove that the language

$$\text{PAL3} = \{\langle M\rangle : \text{PAL} = L(M)\}$$

is neither c.e. nor co-c.e. (24 pts.) Then answer: What about the language

$$\text{PAL4} = \{\langle M\rangle : \text{PAL} \subseteq L(M)\}?$$

Is it c.e.? co-c.e.? neither-nor? (12 pts., for 36 total on the problem and 102 pts. on the written part of the problem set)

Presentation Options For Thu–Fri. 10/22–23.

(The first two or three can be done by a team.)

(A) Any workable system of logic F gives rise to a *proof predicate* $R_F(S, \pi)$ meaning π is a proof of the theorem S in the system F. The system is *effective* if R_F is decidable—and usually R_F is polynomial-time decidable. Then the set of *theorems of* F, namely $\{S : (\exists \pi) R_F(S, \pi)\}$, is always c.e. but not necessarily decidable. The system F is *sound* if every theorem T is true under a natural interpretation of what it says. An important body of theorems are all statements of the form

$$T(M, x, \vec{c}),$$

where $\vec{c}$ is an encoding of a sequence of IDs giving a valid accepting computation of the Turing machine M on input x, that are factually true. This so-called *Kleene T-predicate* is also decidable—and in polynomial, indeed linear, time. Using it, we can define the language E_{TM} along lines shown in lecture:

$$E_{TM} = \{\langle M\rangle : (\forall x)(\forall \vec{c}) \neg T(M, x, \vec{c})\}.$$

A formal system F is *adequate for computation* if it proves that all true cases of T are true and that all false cases of T are false (the latter are needed to verify cases of E_{TM}), as well as handling logical quantifiers and basic numerical and string operations. The bulk of Kurt Gödel's famous 1931 paper with his first incompleteness theorem was the technical detail of showing how basic arithmetic is adequate. But we can get most of Gödel's theorem just by taking all the above for granted about a given F: *Show that there must exist Turing machines M such that $\langle M \rangle \in E_{TM}$ is true but not a theorem of F.* The only principle you need is that if a language A is c.e. and a language B is not c.e., then A can't equal B.

(B) With reference to (a) and the Wed. 10/14 lecture (near the end), note that we can formalize "M is total" by the statement

$$S_M = (\forall x)(\exists \vec{c}) T(M, x, \vec{c}).$$

OK, we changed the meaning of the T-predicate a little to have it say that M halts rather than accepts, but we can roll with that difference. (What Kleene actually did was use the halting version but he made an extra function $U(\vec{c})$ to give the output, so now "$T(M, x, \vec{c}) \wedge U(\vec{c}) = 1$" is the same as the accepting version of the T-predicate in (a).) Now define

$$D_F = \{\langle M, \pi \rangle : R_F(S_M, \pi) \wedge \langle M, \pi \rangle \notin L(M)\}.$$

Prove that assuming F is sound, (a) the language D_F is decidable, but (b) F cannot prove it. The moral is that no matter how strong a formal system F is, there will always be problems that are decidable but *not* by programs that F can verify.

Indeed, show that there does not exist a Turing machine Q such that $L(Q) = D_F$ and F proves the statement S_Q. Then ask yourself: wait, why can't Q be the Turing machine translation of the pseudocode I just gave in part (a) to show that D_F is decidable—?? (This is where this all yields a cut-down version of Gödel's second incompleteness theorem.)

(C) Consider any and all languages B that we can define in the form

$$x \in B \iff (\forall y)(\exists z) R(x, y, z)$$

where R is a decidable predicate—that is, the language of encoded triples $\langle x, y, z \rangle$ that make R true is decidable. Above we have shown that the language of total machines (which is often called TOT) can be defined this way, ditto the ALL_{TM} language. So can the E_{TM} language—because we can "throw away" the $(\exists z)$ quantifier by just making z an ignored variable in the predicate, and just using the $(\forall y)$ quantifier where "y" becomes the pair $\langle x, \vec{c} \rangle$ from above. [Note a "symbol shift": the "x" here is $\langle M \rangle$; the "y" is $\langle x, \vec{c} \rangle$, and the "$R$" is $\neg T(\cdots)$.] Moreover, we can do this for NE_{TM} by making the "$(\forall y)$" the throwaway and thinking $z = \langle x, \vec{c} \rangle$ instead, with $R = T$ rather than its negation.

The standard name for the class of languages B that are definable this way is $\prod_2$, because the outer $(\forall y)$ quantifier (when not ignored) sets the tone for the definition and there is a powerful analogy between a universal quantifier and a (possibly infinite) product. For one thing, a product over an empty domain defaults to 1, just as a statement that is universally

quantified over an empty domain defaults to true. Thus $\prod_2$ contains both RE and co-RE and also contains lots of languages that are neither c.e. not co-c.e. There is also the class of languages A having definitions of the form

$$x \in A \iff (\exists y)(\forall z)R'(x, y, z),$$

and this class is called $\sum_2$ because the leading existential quantifier sets the tone. Then $\prod_2 = co\text{-}\Sigma_2$ and vice-versa. An example of a language in $\sum_2$ is $FIN = \{\langle M\rangle : L(M) \text{ is finite}\}$, since:

$$L(M) \text{ is finite} \iff (\exists t)(\forall x, \vec{c}) : |x| \geq t \longrightarrow \neg T(M, x, \vec{c}).$$

This is using the "accepts" version of the T predicate; the point is that the portion after the colon : is decidable once M, t, x, and $\vec{c}$ are determined. The reason a single-barred $\longrightarrow$ is used in the body is that it is part of the program logic, whereas the double arrow $\iff$ is part of our outer analysis.

Finally getting to the presentation exercise—after you do a 5-minute preamble from the above: Show that the language TOT is *complete* for the class $\prod_2$ under mapping reductions. We've shown it belongs to $\prod_2$, so what you have to do is take a predicate $R(x, y, z)$ that defines a generic language $B \in \prod_2$ in the above manner, and create a Turing machine $M_{R,x}$ that is total if and only if $(\forall y)(\exists z)R(x, y, z)$ is true for x. You have to show that the mapping from x to $M_{R,x}$ is computable, where R is fixed but x can vary. Then explain how ALL_{TM} is likewise complete. If time allows, think about $INF = \{\langle M\rangle : L(M) \text{ is infinite}\}$, which is basically the complement of FIN... [But please stop short of the subject of "oracles" and how $\sum_2$ equals the class of languages that are "recursively enumerable in the Halting Problem"—I wish to focus on the "all" versus "exists" logical structure, and oracles tend to get away from that. What oracles are best for, IMHO, is computing functions via yes/no answers to languages.]

(D) Suppose we have a single-tape DTM M that decides whether a given binary string x is a palindrome. Then M works correctly on strings of the form $x = v10^r1w$ where $r = |v| = |w|$: it will accept them only when $w = v^R$. Call the two cells occupied by the two 1s surrounding the central 0^r the two "mileposts." The TM M can be allowed to change the 1s in these cells but they are still the mileposts.

The computation $M(x)$ starts to the left of the left milepost and must reach a point where it reads the right milepost *and makes a right move* (in order to start reading the "w" part). Call the state after that right-move q_1. Now we care about is the next time (if any) that the sole tape head of M goes back left of the left milepost to re-consult the "v" region and then comes back right, crossing the right-milepost cell once again. The state after that move is q_2. Now M might criss-cross the right milepost cell many times in quick succession, but we only care about cases where it has gone into the leftmost region and come back. So if we have defined $q_1, \ldots, q_i$, then we only get q_{i+1} after it has traveled the distance between the mileposts, which means that q_{i+1} comes at least $2r$ steps after q_i ($2r + 4$ steps, to be more precise, but its being $\Theta(r)$ is what we care about and why we have the middle 0^r region to begin with). Let q_k be the last such state in the sequence before M halts. Note that states can be repeated in $(q_1, \ldots, q_k)$; the basic one-tape TM that recognizes PAL in quadratic time needs only one loop with two branches, one using a state r_0 to remember that the last char read was a 0 and otherwise r_1 to remember a 1. The sequencing is what matters—$(q_1, \ldots, q_k)$ is called a *crossing sequence* at the right milepost cell. The key fact is:

For any r and $v \in \{0,1\}^r$, with M fixed, the crossing sequence $(q_1, \ldots, q_k)$ of the accepting computation of M on input $x = v10^r1v^R$ uniquely determines v.

Convince yourself of this fact by considering how the computation works on a single tape, regardless of how the characters on that tape get altered. Then use this to justify two further deductions:

- For some strings $v \in \{0,1\}^r$, the corresponding k must be proportional to r (as r grows—note that the code of M stays fixed). Otherwise you would have a violation of the Pigeonhole principle, since there are 2^r distinct strings v but you wouldn't have enough crossing sequences to go around.
- It follows that the running time of M must be $\Omega(r^2)$, which is quadratic in $n = |x| = 3r + 2$.

Hence it is impossible to recognize palindromes in less than quadratic time on a single-tape TM. Lurking in the background are how this argument also lower-bounds the number of passes that a "streaming algorithm" in place of M must make, how the pigeonhole argument resembles that for $S = \{0,1\}^r$ being a PD set for PAL, and the notion of *Kolmogorov complexity* in the skimmed section of Debray's notes—but you especially need not go into the whirl of definitions leading to *Berry's Paradox* there.

(E) Unlike (D), this option is IMHO represented well by Debray's notes on pages 32–33 without going to excess. They are supplemented by my own notes called "a programmer's view of the S-m-n and Recursion Theorems" in the Optional Reading section of the course webpage. Debray calls this "the trippiest part of the class" and "getting weirder"; my program-style presentation tries to unroll the enigma but maybe it too gets trippy. Anyway, the goal is to show how to create a total program P such that $L(P) = \{\langle P \rangle\}$. Finish with some general remarks about why this doesn't cause any paradox.

(F) Define a *finite-state transducer* (FST) $T = (Q, \Sigma, \delta, s, F, \phi)$ to have instructions of the form $(p, c/u, q)$ where u is a string that is output during the transition. This allows $u = \epsilon$ whereupon output is paused in that step. In addition, the rule is that if T does not end in a state in F, then the entire computation is cancelled—so the function $T(x)$ it computes can be undefined at certain x. But if the final state f is in F, then there is a final string $\phi(f)$ that gets appended to the output, again possibly ϵ to do no further change.

Now say that a language A *regular reduces* to a language B (written $A \leq_m^{reg} B$) if there is an FST T such that for all x,

$$x \in A \iff T(x) \text{ is defined and } T(x) \in B.$$

Show that if A is not regular, then B is not regular. Use this to give some examples of non-regularity proofs by reduction rather than by Myhill-Nerode. Show, for instance, that $\{a^n b^n : n \geq 0\} \leq_m^{reg} \{x : \#a(x) = \#b(x)\}$.