Problem Set 8: CS103

Katherine Cheng, Richard Davis, Marty Keil

June 3, 2015

Problem 1: Closure Properties of RE

i.
```
bool inL1nL2(string w) {
    return inL1(w) && inL2(w);
}
```

This method, which models the language L1 *cap* L2, satisfies the properties of a recognizer. When $w \in L1 \text{cap} L2$ the method will return true, and the method returns false or loops infinitely otherwise.

ii. Since the RE languages are closed under union, we would expect L1 $\cup$ L2 to be recognizable. By definition of a recognizable language, this means that the method should return true for any string $w \in L1 \cup L2$. Because the line `return inL1(w) || inL2(w);` evaluates left to right, if `inL1(w)` enters an infinite loop, the statement will never evaluate the second function, `inL2(w)`. If `inL2(w)` evaluates to true, the recognizer should return true - however, since this portion of the statement is never evaluated, this method not properly model the behavior of a recognizer.

iii.
```
bool imConvincedIsInL1uL2(string w, string c) {
    return imConvincedIsInL1(w, c) || imConvincedIsInL2(w, c);
}
```

As opposed to the method in part ii of this problem, this new method `imConvincedIsInL1uL2()` will not get caught in an infinite loop. This is because both `imConvincedIsInL1(w, c)` and `imConvincedIsInL2(w, c)` always return a true or false value, so the return statement evaluates both sides of the 'or' properly.

Problem 2: Password Checking

1. meta level - by contradiction that password checking language is RE/verifiable. Then we can create this. now (i). there is some certificate c where imConvinced will return true, we accept(). If we accept, the language of the machine is (we have 2 levels of machines, (1) candidate password checkers, (2) looks at machines and says if it's a password checker) higher machine has said you're only going to

accept p. Now, reach a contradiction. If we are a passwordchecker, just accepts always regardless of the input. input of the machine is everything.

contradiction: if it is a password checker then it accepts all inputs

2. it'll infinitely loop

   we're guaranteed the pass checker returns p and only p

   if we loop infinitely on every string,

3. input equal to p?

i. We begin with the assumption that this program is a valid password checker. If this is the case, this means the program only accepts a single string p. However, because of the way this program is constructed, we see that if it is a valid password checker we will be able to find some certificate c such that the program will always accept no matter the input string. This is a contradiction.

ii. We begin with the assumption that this program is not a valid password checker. In this case, the program is constructed in a way that no matter what the input string is the program never accepts. This is consistent with the program not being a valid password checker.

iii. The modified program is as follows:

```
bool imConvincedIsPasswordChecker(string program, string certificate) {
/* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    bool actualAnswer = (input == p);

    for (int i = 0 to infinity) {
        for (each string c of length i) {
            if (imConvincedIsPasswordChecker(me, c)) {
                accept(); // Once the certificate is found, accept any input
            } else {
                if (actualAnswer) {
                    accept();
                } else {
```

```
                    reject();
                }
            }
        }
    }
}
```

This program leads to a contradiction regardless of whether it is a password checker. If the program is a password checker, by the same argument as in part i. we can show that this leads to a contradiciton. If the program is not a password checker, we know that `imConvincedIsPasswordChecker` returns false for all values of c. However, the program is constructed such that for all values of c, the program behaves like a password checker, only accepting input strings that are equal to p. In other words, if the program is not a password checker (as determined by `imConvincedIsPasswordChecker`) the program always behaves as a password checker. This is a contradiction.

iv.

Theorem. $L \notin \boldsymbol{RE}$.

Proof. By contradiction; assume that $L \in \mathbf{RE}$. Then there is some verifier V for L. This verifier has the property that if M is a TM that is a password checker, there is a certificate c such that V accepts $\langle M, c\rangle$, and if M is not a password checker, V will never accept $\langle M, c\rangle$ for any certificate c.

Given this, we could then construct the following TM:

M =On input w:

Have M obtain its own description, $\langle M\rangle$. For all strings c:

If V accepts $\langle M, c\rangle$, accept.

Otherwise, if V does not accept $\langle M, c\rangle$, accept if the input string is equal to p and return false otherwise.

Choose any string w and trace through the execution of the machine. If V ever accepts $\langle M, c\rangle$, we are guaranteed that M only accepts p, but in this case we find that M accepts any input, a contradiction. If V never accepts $\langle M, c\rangle$, then we are guaranteed that M is not a password checker, but in this case we find that M always behaves as a password checker, a contradiction.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $L \notin \mathbf{RE}$. □

Problem 3: Equivalent TMs

```
bool imConvincedAreEqual(string p1, string p2, string certificate) {
  // Returns true if p1 and p2 are both turing machines that have the same language, false
```

```
}

bool simulateTM(string tm, string inp) {
  // simulate the tm run on the string inp and return whatever tm returns
}

int main() {
  string me = mySource();
  string m2 = getOtherTMSource();
  string input = getInput();

  for (i = 0 to infinity) {
    for (each string c of length i) {
      if (imConvincedAreEqual(me, m2, c)) {
        return !simulateTM(m2, input);
      } else {
        return simulateTM(m2, input);
      }
    }
  }
}
```

Problem 4: The Big Picture

m1 takes an input, it will accept/reject/loop m2 has some other language if imConvinced true, then language does not match what you have coded

1. REG
2. ALL
3. REG
4. REG
5. REG (regular languages closed under union)
6. R (M-N so not REG, but CFG so R)
7. RE (?)

8. ALL (there is a TM where L={∅} will loop forever, can't be RE)
9. RE (n is the certificate) (what happens when it loops? don't worry about that)
10. ALL
11. ALL
12. RE

if in RE, if I give you a machine, deep down I know this is in the machine, machine will halt. R if not in language, will halt, which is not guaranteed. Not in RE, I can give you (M,n) that's in the language and you can't conclusively tell me it's in the language. ex) at most length 5, run on all strings on at most length 5, if in language, will accept it. I'm giving you something in the language, if you halt

Problem 5: 4-Colorability

i. 3COLOR ∈ NP because it is NP-complete and NP-complete is a subset of NP. Therefore if 3COLOR can be reduced to 4COLOR, then 4COLOR is also an element of NP. Take every 3COLOR graph, in which there exists a way to color each of the nodes one of 3 colors, such that no two nodes of the same color are connected by an edge. Each of these graphs is also by definition 4-colorable, just one of the 4 colors will go unused. Therefore, there is polynomial time reduction that shows that 4COLOR must also be in NP.

   I'm am not sure about this explanation. I actually think they might be looking for:

   We can guess different 4Colorings of a graph using a nondeterministic turing machine. We can then deterministically check whether the coloring option is a legal 4-coloring of G. This non-deterministic turing machine us in polynomial time, because it would only take the number of nodes, n, to check if this was a valid coloring for the graph.

ii. Take an arbitrary graph G and contrast Graph G' by adding a new node that is connected to all other nodes.

1. If the original graph G is 3-colorable, then the new graph G' must be 4-colorable because in the original graph G at most three colors were used so that no two nodes of the same color were connected by an edge. If a new node is added that connects to all other nodes, then this node must be of a new color. This new graph would require one additional color to prevent the same color from being connected by an edge, bringing the total to at most 4 colors, the definition of a 4-colorable graph.
2. If we take G' and complete the transformation that removes the node that connects to all other nodes, we have then reduced the colorable number by 1. This is because that node must have been a different

color than all other nodes, by definition of colorability. Since G' was 4-colorable(had at most 4 colors needed), G is therefore 3-colorable(has at most 3 colors needed).

3. This reduction can be completed in polynomial time n+1. The reduction must add one node to graph G, then make n (the number of nodes in graph G) edges between the new node and all previous nodes in G.

Problem 6: Resolving P $\stackrel{?}{=}$ NP

1. neither
2. neither
3. P = NP
4. P = NP
5. P $\neq$ NP
6. P $\neq$ NP
7. neither
8. neither
9. neither
10. neither
11. neither
12. neither
13. P = NP
14. P = NP
15. neither
16. neither

Problem 7: The Big Picture

i. $\{a^n \mid n \in \mathbb{N}\}$ is an example of a regular language. You can prove that this language is regular by creating a regular expression for it, namely a^*. We could also have proven the language is regular by creating a DFA or NFA for it.

ii. $\{a^n b^n \mid n \in \mathbb{N}\}$ is an example of a context-free language that is not regular. You can prove that it is context free by providing the context-free grammar, aka production rules, for deriving the language. The CFG for this example is $S \to aSb|\epsilon$. We can prove that this language is not regular using the Myhill-Nerode Theorem.

iii. $\{a^n \mid n \in \mathbb{N}\}$ is an example of a language in P. Since we know this language is a regular language, and regular languages $\subseteq$ P, it follows that the language is in P. In general, you can prove a language is in P by creating a Turing machine decider for the language that decides in polynomial time.

iv. INDSET $= \{\langle G, n\rangle \mid G$ is an undirected graph with an independent set of size at least $n\}$ is an example of a language in NP that is not known to be in P. You can prove INDSET is in NP by building a polynomial-time verifier for the language. For this language, the polynomial-time verifier checks whether S is an n-element independent set.

In general, you can prove that a language is in NP by building a polynomial-time verifier for the language. Specifically, you would create a NTM that nondeterministically guesses a certificate, then deterministically runs the verifier to check it. Then, you would argue that the NTM runs in nondeterministic polynomial time.

We don't know whether INDSET is in P because INDSET $\in$ NP-complete. All known algorithms for NP-complete problems run in worst-case exponential time, and are infeasible for reasonably-sized inputs. We do not know whether INDSET is in P because P$\stackrel{?}{=}$NP.

v. A_{TM}, the language of the Universal Turing machine, is an example of a language in RE not contained in R. You can prove that A_{TM} is RE by simulating the machine, which will accept if string w is in the language of machine M, and reject/loop otherwise. You can prove that A_{TM} is not contained in R by contradiction - you'd show that if there were a decider for A_{TM}, then you could construct a self-referential TM that flips its response based on the decider's answer, thus contradicting the decider in all cases.

vi. $\{ \langle M\rangle \mid M$ is a Turing machine and $L(M) = \emptyset\}$ is an example of a language that is not in RE. You can prove it is not contained in RE by constructing a case of a TM where the language cannot be verified - for example, a TM that loops infinitely and does not halt.