CS 675A: Midterm

Instructor: Pramod Subramanyan

22 February 2020

Introduction

This examination has three questions. Question 1 only has programming while questions 2 and 3 have both written and programming components. The questions may not all take the same amount of time, so carefully consider in what order you'd like to answer the questions. Total time for the exam is **3:00 hours** and the total marks are **150** points.

Rules

Please read all the rules carefully.

DOs

- You are allowed to consult all reference material, including making Google searches.
- Posting public questions on Piazza is allowed and we will try to answer them ASAP. Private questions will be disabled for the duration of the exam. Looking at old posts on Piazza is also acceptable.
- If you are unable to get the code working correctly, please write your solution approach and we will try to give you partial credit.
- There will be points for good tests. If you feel you have added meaningful new tests, let us know and we will consider them during our grading.

DON'Ts

- You are not allowed to communicate with any other students. So, we are banning usage of email, chat and other communication apps like WhatsApp and Skype.
- No copying of code or solutions is allowed. Any attempt to cheat by communicating with other students, including those who are not taking the class, is prohibited.

Submission

You can submit the written component in two ways:

(a) Create a file called `solution.txt` which contains your answers.

(b) Write it out in the answer booklet. In this case too, you must submit `solution.txt` and write that you are submitting an answer booklet in this file. Don't forget to write your name and roll number on the answer booklet.

For the programming component, please submit through the homework server in the midterm folder. The files you need to upload are:

1. `solution.txt`
2. `DPSolver.scala`
3. `DPSolverEx.scala`
4. `ProofChecker.scala`
5. `Relations.scala`
6. `CryptArithmetic.scala`

1 SAT Solving (50 points)

(a) Implement the Davis-Putnam algorithm for SAT solving. We have provided a skeleton code in `CNF.scala` and `DPSolver.scala`. The class `Clause` in `CNF.scala` contains a lot of useful methods which you probably want to use in your implementation.

Start by implementing the `Solver.resolve` method. Then implement the `addClause` and `solve` methods in the class `Solver`. There are some tests in the `MidtermTests.scala` you can use to check your code. (20 points)

(b) Modify your solver to produce a resolution proof of unsatisfiability. The data structures you have to use are provided in `DPSolverEx.scala`. You will need modify the `solve` method of the `DPSolverEx` class. (20 points)

(c) Implement a verifier for resolution proofs. You will to modify the method `verify` in `ProofChecker.scala`. (10 points)

2 Properties of a Relations via SAT (40 points)

Suppose we have a program takes as input two variables and produces only output. Assume all inputs and outputs are of the same type and that the set of values that variables of this type can take is T. If we assume that the program always terminates, then its input/output behavior can be represented by a relation $R : T \times T \times T$. A particular tuple of values (x, y, z) belongs to this relation if the program produces output z when given inputs x and y.

For example, if I have the following function.

```
int foo(int a, int b)
{
    return a + b;
}
```

This can be modeled by the relation $R \subseteq \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ where $R = \{(x, y, z) \mid x + y = z\}$. $\mathbb{Z}$ is the set of integers. This relation can be represented in a SAT/SMT solver through as an indicator function, which is a function $F(x, y, z)$ such that $F(x, y, z) = \mathit{true}$ iff $(x, y, z) \in R$. For the above example, $F(x, y, z) \equiv (x + y = z)$. Note the $=$ in the last formula represents equality not assignment. You are given the definition of an indicator function of a relation as an object of type Expr and sets of input and output variables of this relation.

(a) How would determine if the given relation is a function? Recall that a function is a relation where every input has exactly one output. Write the query you would pose to the SAT solver. Then implement the code in the function `isFunction` in `Relations.scala`. It may be helpful (but is not necessary) to implement the function `subst` first. (20 points)

(b) How would you determine if a given relation is transitive? Write the query you would pose to the SAT solver. Then implement the corresponding code in the function `isTransitive` in `Relations.scala`. (20 points)

3 CryptArithmetic Revisited (60 points)

Recall that we created a solver for the cryptarithmetic problem SEND + MORE = MONEY. In this problem, you are asked to formulate more such puzzles. We are giving you a small list of randomly-chosen words in the file `WordList.scala`. Your job is to construct an SMT instance that finds a valid puzzle of length greater than equal to *minLen* consisting of the words in this list. The length of a puzzle is the sum of the lengths of the words that comprise the puzzle. (In other words, the length of the puzzle SEND + MORE = MONEY is 4 + 4 + 5 = 13.)

The rules for a valid puzzle are as follows.

1. It must be of the form WORDA + WORDB = WORDC.
2. All the words must occur in the set `words` in the object `WordList`.
3. Assigning a value between 0 and 9 to each of the letters must produce a valid sum.
4. All letters in the words must be assigned distinct values.

5. (**Optional/Extra-Credit)** The words must not have leading zeros.

You must formula a *single* SMT instance to get full-credit. A solution that calls the SMT solver repeatedly in a loop will get only partial credit.

(a) Briefly describe your algorithm in the write-up. (20 points)

(b) Implement the function `findPuzzle` in `CryptArithmetic.scala`. (40 points)

Hints: this problem is challenging so make sure you start it only after the other two problems are done. There are some theories we have learned in class that will be a good fit for this problem. You can try changing the number of words extracted from the list by changing the argument to the `take` function in `WordList.scala` to see if that makes it easier for the solver to find a solution.