CS 510 Homework 4

4/11/17 (Due 4/29/17)

Problem 1 (40p) Program equivalence checking using SMT/
In this assignment, you are asked to convert two different implementations for the same task into SSA form and prove that they are not semantically equivalent. Next, you will correct the buggy version of the program (student.c) and prove that it is semantically equivalent to the correct version (teacher.c). The code needed for this project as well as the documentation for z3py can be downloaded at:

http://web.ics.purdue.edu/~perry74/equivalence_assignment.zip

Program Descriptions:
In the project folder you will find two separate C programs: teacher.c, and student.c. Both of these programs take in an integer as input and print out each digit in the number from least significant to most significant. However, student.c is buggy and will not have the same output as teacher.c in many cases. For this particular homework assume that all the inputs given to the programs are between 100 and 999 or between -999 and -100.

Part 1: Convert to SSA
Edit teacher.c and student.c so that variables in the programs are in SSA form. This will require the creation of new variables and loops will have to be unrolled.

Part 2: Prove the Programs are Not Semantically Equivalent
Use the python wrapper for the Z3 theorem prover from Microsoft Research to model the semantic behavior of the two programs. Additionally, add constraints that prove that the two programs do not always behave the same way when given the same inputs. Once you have completed the script Z3 will report a concrete instance of when the programs differ in behavior.

Part 3: Correct student.c and Prove the Updated Version is Semantically Equivalent to teacher.c
Make modifications to student.c so that it always exhibits the same behavior as teacher.c on inputs between 100 and 999 or between -999 and -100. After fixing the program, model the updated version using Z3 and show that after your updates the two programs are semantically equivalent.

Suggested Steps:

1.) Install Z3 and the corresponding python wrapper. The source code and installation guide can be found at: https://github.com/Z3Prover/z3 (Make sure to read the Python section in the README file before starting the build process.)

2.) Make sure that Z3 has been correctly built and installed for python. To test this simply make a python script that says: “from z3 import *”. If there are no complaints when running the script then z3 is installed correctly. (The default installation process makes is so the z3 api only works when running

python scripts in the z3/build directory. If you are having trouble making it work make sure you are in this directory when running your scripts)

3.) Read through the z3py documents included in the examples directory. In particular, read through the files guide-examples.htm and advanced-examples.htm (open them in a browser). These files describe the syntax of the various features available in z3 and also have some interesting example problems.

4.) Experiment with the z3py tool by creating integer variables with constraints involving conditionals and basic arithmetic and find solutions to them using the solver. This should allow you to get an understanding of how the tool works and give you a better idea of what you'll need to do for the project. An example of doing so is given in the attachment.

5.) Begin modeling the programs!

Important Notes:
1.) Use the If(condition, then, else) construct when modeling the program. For some reason this syntax is not included in the original z3py documentation. However, it is quite useful when modeling programs.

2.) When modeling variables and their corresponding updates be careful. Remember that the variables for z3 can only be assigned to once. (HINT: Think about SSA)

3.) Modeling loops can often be difficult. Think carefully about how you can handle them. (HINT: what do the bounds on program inputs infer about each loop's behavior?)

Example:
Consider the following simple C program:

```
int x, y;

x = atoi(argv[1]);
y = 1;

if(x > 10) {
  y = 3;
}

return 0;
```

And this z3py script that proves that if the value of the first input argument is greater than 10, y must have a value of 3 at the point the program returns.

```
from z3 import *

# Define some ints for the program/ssa variables
x = Int('x')
y_0 = Int('y_0')
y_1 = Int('y_1')

# Model the assignment of y as an equivalence
stmt1 = y_0 == 1
# Model the if stmt and updates to y
stmt2 = If(x > 10, y_1 == 3, y_1 == y_0)

# Create a goal and add the stmt's
g = Goal()
g.add(And(stmt1, stmt2))

# Add the conditions your proving
g.add(And(x > 10, y_1 != 3))
solve(g)
```

The returned solution from Z3 will be UNSAT. This is because we added conditions that stated when x is greater than 10, y_1 (the version of y before the program returns) does not equal 3. Therefore, we are able to prove that if the input argument is greater than 10 y must equal 3.

Grading:
Your grade will be based on four deliverables:

1.) SSA form of student.c and teacher.c
2.) Python script that uses z3 to prove student.c and teacher.c are not equivalent on the given bounds
3.) Corrected version of student.c
4.) Python script that uses z3 to prove corrected_student.c and teacher.c are equivalent on the given bounds

The first part will be graded based on how correct your SSA implementation is. The second and fourth parts will be graded on how accurately your models represent the real programs and how well it proves/disproves their semantic equivalence. The third part will be graded on how well you corrected student.c

2 (30p) Predicate Abstraction

In order to mitigate state explosion in explicit state model checking, predicate abstraction is often used to reduce state space. Counter-example guided refinement may be needed during the process.

```
void main (void)
{
   int a, b;
   a=1;
   b=1;
   if (a > b) {
     a--;
   } else {
     a++;
   }
   assert(a>b);
}
```

(a) Starting with predicate a>b, apply predicate abstraction to the above program.
(b) Perform explicit state model checking on the abstract program, present your execution tree and the counter example, if there is one.
(c) If there is a counter example in (b), test if it is a counter example in the original program.
(d) If the counter example is bogus, refine your abstraction so that you either find a real counter example or show the correctness of the program.