CS243 Homework 7

Winter 2017

Due: March 16, Thursday, 3pm

Directions:

- This is an **individual** assignment.
- This assignment has two parts:
  1. **The first part if a written question**. Write the solution to this question, and hand it over at the beginning of class on Thursday, March 16, if you are not using late days. If you are using late days, drop your assignment in the CS243 drop box. We will collect assignments at 3pm from the drop box on Friday and Saturday.
  2. The second part requires you to generate an SMT formula, and run it with Z3 online, to find bugs in a given C snippet. **Copy this formula into `func.smt` in a submit directory on `myth`**, and run the submit script, `/usr/class/cs243/submit`. Enter the Lab number as 4 for this part. Note that this part is an individual assignment, unlike previous coding assignments.
- Both parts are due on March 16, at 3pm.
- The total number of late days will be the maximum of late days used for first part, and late days used for the second part.

Problem 1. Pointer Analysis on a Java snippet

Perform pointer analysis on the following java snippet, and answer the following questions.

```
public class Foo {
    public Foo a;
    public Foo b;
    public static main(String [] args) {
        Foo x = new Foo(); // h1
        Foo y = new Foo(); // h2
        Foo z = new Foo(); // h3

        bar(x, y);
        z.a = x;
        bar(y, z);
    }
```

```
        public static void bar(Foo p, Foo q) {
            if(p.b == q.a) return;
            baz(p, q);
            qux(p, q);
        }

        public static void baz(Foo c, Foo d) {
            c.a = d;
            if(c != d.b) qux(c, d);
        }

        public static void qux(Foo m, Foo n) {
            n.b = m;
            if(m.a != n) baz(m, n);
        }
}
```

1. What are the `hP` tuples inferred from this code in a context insensitive analysis? To get you started heres one: `hP(h1, a, h2)`.
2. Draw the call-graph of the original program and the call-graph after running the cloning-based algorithm for context-sensitive analysis. Do not make clones for recursive calls.
3. How many copies of each procedure are created (for `bar`, `baz` and `qux`)?
4. What are the `hP` tuples inferred from this code in a context sensitive analysis?

Problem 2. Satisfiability Modulo Theories

While this problem is submitted electronically like a lab, you need to complete it individually as if it were a written assignment.

In this problem you will use an SMT solver to find test cases exhibiting a bug in the following C function:

```
int func(int x, int[] data, int N) {
  int v, z;
  if (0 <= x && x < N) {
    if (x <= N/2) {
      x = x + 1;
    }
    v = data[x]; // line 7
    if (0 <= v && v < N) {
      z = data[v]; // line 9
    } else {
      z = 0;
    }
    return data[z]; // line 13
  }
  return 0;
}
```

Assume we care about 'crashes' due to array out-of-bounds accesses (even if they don't cause the program to segfault), and that `data` has length `N`. First you will use the phi variable technique to translate this function into a formula. Then you will run a solver and interpret its output. We will use the Z3 tool available at rise4fun.com/z3/?menu=0. There is a introductory guide at rise4fun.com/Z3/tutorial/guide which also serves as a language reference sufficient for our purposes.

Your submission will consist of the file `func.smt` which contains the input to Z3 (copy and pasted from the web editor), along with comments explaining your interpretation of the result. Follow these steps to complete this problem:

1. **SSA Construction.** The first step is to write the function in SSA form by assigning all definitions a unique suffix. You need to generate phi variables for each branch. At each join point in the CFG you will need to introduce new definitions for the variables that are defined on either path. At this point, the program is still imperative code, but each variable is defined exactly once.

2. **Translation to SMT.** The second step is to translate into an SMT formula. An assignment `x3 = e` becomes an assertion `(assert (= x3 E))` in SMT, where `E` is the translation of `e`. In particular, you need to translate `int` operations at the C level into *bit vector* operations at the SMT level. Do not use `Int` in Z3, as these model mathematical integers rather than 32bit 2's complement arithmetic. For example, the translation of `x = y + 1` is:

   ```
   (declare-const x (_ BitVec 32))
   ```

```
(declare-const y (_ BitVec 32))

(assert (= x (bvadd y #x00000001)))

(check-sat)
(get-model)
```

Z3 responds with:

```
sat
(model
  (define-fun y () (_ BitVec 32)
    #x00000000)
  (define-fun x () (_ BitVec 32)
    #x00000001)
)
```

Here the program is satisfiable with model `x = 1`, `y = 0`. Note that the variables x and y are given as functions of no arguments (which must be constants because there are no side effects), and the constants themselves are hexadecimal.

To translate arrays, you will use variables of the sort `(Array (_ BitVec 32) (_ BitVec 32))`. Array dereferences like `data[i]` become `(select data i)` when translating as an expression.

3. **Bounds checks.** The final step is to add an assertion to check each of the three particular array accesses. You need to check that the signed value of the index is in bounds. Further, not all accesses are accessible on all paths, so you need to *guard* the assertion for a particular access by using the phi variables. The assertion should express "the path reaches this access, and it is out of bounds." Note that this will possibly constrain some path variables if the access is nested inside an if statement, for example.

   You should use the sequence `(push)(assert C)(check-sat)(pop)` for each access, where C is the check for that access. Push/pop allows us to add C to our set of assertions, check satisfiability, and then remove it to add a different C, because adding all such assertions together will find a path which crash all points simultaneously rather than just at least one.

4. **Intepretation.** Once you find (a) bug(s), add `(get-model)` after `(check-sat)` within the push/pop sequence for each satisfiable assertion, to print the model found for that bug. In comments by each section, interpret the results. Does the result indicate a crash can occur on some concrete path? If not, does this mean there can be no crash for this access? If there is a crash, translate the model into a concrete input represented by a call `data = {...}; func(...)` which causes a crash *at the corresponding access*. Hint: what happens if the program has more than one bug, and how should the model be interpreted in that case?

5. **Submission.** Copy this Z3 input to a file `func.smt` and submit it using the script on `myth`, lab 4. Your file should include variable definitions, assertions defining each assignment and phi variable, a push/assert/check-sat/get-model?/pop sequence labeled by line number for each of the three accesses, and comments for each section from part 4.