Homework 8
Posted Tuesday April 17, Due Tuesday May 1
(Optional. Homework can be completed individually or in teams of two.)
50 points

Note: If you plan to work on a different version of this homework, please send me a paragraph describing your proposal. When done, email me a zip file with your code along with a readme file with instructions on how to run it. I won’t be setting up a Submitty page. Also, there will be no extensions on this homework since we are at the end of the semester.

1. A Simple Symbolic Execution Engine

In this homework you will build a symbolic execution engine on top of IMP. The program comes with a postcondition (assertion) and your task is similar to what we did with backwards reasoning — produce a set of inputs that violate the postcondition. Use the starter code from HW7 and add your code in implang.ml.

1.1. Concrete evaluation. Start by coding functions evalAExp, evalBExp and step that work on concrete values:

```
let rec evalAExp e sigma = ...

takes an arithmetic expression e and a store sigma, and evaluates e in sigma.
```

```
let rec evalBExp e sigma = ...

takes a boolean expression e and a store sigma, and evaluates it. Finally
```

```
let rec step sigma label = ...
```

is the evaluation step. It takes a store sigma and a label and produces a new store and label emulating the small-step operational semantics we discussed earlier in class.

1.2. Symbolic expressions. Next, extend the evaluation and step functions to work on symbolic expressions. Add symbolic expressions

```
type expr = ... | SymAVar of string | SymBVar of string | SymAPlus ...
```

where strings represent symbolic variable names. SymAVar are arithmetic, i.e., integer variables, SymBVar are boolean variables, etc. You can add expressions as you see fit.

Next, extend concrete evalAExp and evalBExp to work with symbolic expressions:

```
let rec evalAExp e sigma = ... match e with
| SymAVar var -> ...
| SymAPlus(op1,op2) -> symPlus (evalAExp sigma op1) (evalAExp sigma op2)
| ...
```
let rec evalBExp b sigma = ... match b with
| SymBVar var -> ...
| SymBLeq(op1,op2) -> symLeq (evalAExp sigma op1) (evalAExp sigma op2)
| ...

To extend step you would first need to extend the store so that it holds the label and path condition in addition to the mapping from variables to (symbolic and concrete) values. This is because the symbolic execution engine forks execution and maintains more than one path at a time:

```ocaml
type symbolic = {
  sigma : ...
  condition : ...
  label : ...
}
```

When processing an if-then-else/while statement step must decides whether a branch is feasible. To do so, combine the path condition carried by current symbolic state and the branch condition and ask Z3 whether the branch is feasible. (You will need to interface into Z3 from your OCaml code.) For while loops, set a limit on the number of times the symbolic engine enters the loop. You may need to modify the record that models symbolic state. In general, you can modify the skeleton I have specified as you see fit.

1.3. Running the engine. To run the engine, run a worklist algorithm in main! Create an initial symbolic state setting label = 0, condition = true and sigma = empty, and push this state on the worklist. While the worklist is not empty, remove a symbolic state st. If st is a final state, ask Z3 whether the postcondition (assertion) holds at this state and print the model if it does not. Otherwise, take a step of evaluation arriving at a new state st’, then store st’ on the worklist.