Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.820 Foundations of Program Analysis

Problem Set 4

Out: October 22, 2015
Due: Nov 6, 2015 at 5:00 PM

In this problem set, you will be writing your own verification system based on the Z3 SMT solver. As before, collaboration is encouraged when thinking about the pset, but everyone must submit their own code. If you have questions or comments about the pset, please post on piazza.

Problem 1 Working with Z3 (10 points)

For the first part of the pset, the goal will be for you to acquire some experience using the Z3 SMT solver. As a first step, you need to download the solver from here: https://github.com/Z3Prover/z3/releases

The Z3 solver can establish the validity of a logical formula involving terms from different theories, including integer arithmetic, arrays, and uninterpreted functions. Z3 takes as input problems in a standard format called SMT-LIB. This format is supported by all major SMT solvers, so generating formulas in this format will allow you to try different solvers on your problem and pick the one that works best. For this pset, though, we will be sticking with Z3.

Part a: (5 points) Your goal is to use Z3 to answer the following puzzle.

Joe has a sister named Mary; Mary is 5 years younger than Joe and 8 years older than her cousin Ted. In 1997, one of the three surprised everyone by going to college. One of the three had just turned 20 a few years earlier, but he was not the one who caused the surprise. The person who caused the surprise was born after 1985, so nobody expected college so soon.

Given the facts stated above, use Z3 to determine who was the one who gave the surprise. Specifically, you want to create a Z3 input with variables for the age of each of the three characters in 1997 and a variable to indicate which of the three surprised everyone by going to college. Write your solution in a file called `prob41a.z`.

Part b: (5 points) Create a second Z3 input to determine whether the solution you found in the previous subproblem is unique. Could someone else have caused the surprise? Write your solution in a file called `prob41b.z`.

Problem 2 Verification conditions by hand (10 points)

Consider the piece of code below. Your goal is to hand-code the verification condition for that problem in SMT-LIB and feed it to Z3 to tell whether the code is correct or not with respect to its pre/post-conditions.

```
{y=y0, k=k0, t=y0− k0 }
while(t>0){ // Invariant = {t=y−k0 ,k=k0}
          y = y−1;
          t = t−1;
}
{y <= k0 }
```

Your solution should be in a file named `prob42a.z`.

Problem 3 Writing your own verifier (65 points)

Your goal for this part of the assignment is to develop a verifier which given a program with pre/post-conditions and invariants, produces a verification condition that can be fed into Z3 to determine whether a program is correct.

As a starting point, you have been given the following files:

- `lexer.mll`
- `parser.mly`
- `implang.ml`
- `main.ml`
- `Makefile`

Together, `lexer.mll` and `parser.mly` define the syntax of the language you have to support. The lexer and parser defined in these files will read the program as input and convert it into an AST as defined in the `implang.ml` file. The file `main.ml` contains a driver that currently parses the input from stdin, calls the `stmtToStr` function to pretty print the AST into a string and then prints the string into the screen. After you run `make`, you will get an executable `./a.out` that will read from stdin and print to the screen, so if you call

```
  > cat test1.c | ./a.out
```

you will see the result of pretty printing the program contained in `test1.c`. After you implement the verification condition generator, running the same command as above should output a verification condition in Z3's input format so I can pipe it directly into Z3. When the program verifies, Z3 should output 'unsat', and when it does not, Z3 should produce a model. So overall, you are expected to submit a tar file such that I can run

```
> tar −xvf yourfile.tar
> make
> cat mytest.c | ./a.out | z3 −in
```

You want to make sure your executable does not produce additional output that may confuse Z3; while developing, you may find it useful to precede any degut output with a semicolon, which is what SMTLIB uses to indicate single line comments.

A few notes

- You will want to use the `Int` datatype for all integer variables. Z3 will automatically determine whether to use the decision procedures for linear or non-linear arithmetic depending on your constraints.
- We will not impose any restrictions on where pre/postconditions will appear, so you really want to treat Preconditions as Assumes and Postconditions as Asserts as will be discussed in lecture.
- Note that the grammar uses a single '&' and '|' as and and or operators respectively. Assignment is a single equals sign `=` and equality comparison is a double equals sign `==`.
- Keep in mind that your submission will also be tested on programs with wrong invariants or pre/postconditions.

One data-structure that you may find useful is a map. You don't have to use maps to complete the assignment, but you may find they make your life easier. In ocaml, you use a map as follows: First, you need to define the type of map you want, for example, if I want a map whose keys will be strings I have to define it as follows:

```
module StringMap = Map.Make(String);;
```

`StringMap` has to be capitalized. Now, if you want to create an empty StringMap you can just do

```
let m = StringMap.empty;;
```

and m is now an empty map. You can then add things to a map, or remove them, or even apply a function to every key-value pair in the map. One important thing to keep in mind about ocaml maps, though, is that they are immutable, so you can never actually modify a map. For example, if you use the add function to add something to a map you are really creating a new map that has everything the old map had, plus the new stuff. If you want to have a single variable that you use to refer to a map as it evolves, you can use a reference. For example, consider the following code:

```
let m = ref StringMap.empty;;
```

This makes m a reference to a map. By executing

```
m := StringMap.add "hello" "world" !m
```

you are updating the reference m to now point to a new map, one that is equal to the old map m with the key-value pair ("hello", world") added to it.

You can find details of how maps work in this tutorial: http://ocaml.org/learn/tutorials/map.html including information on how to print them.

Part a: (20 points) To get the first 20 points for this assignment, your verifier needs to work correctly for straight-line programs that involve only the following kinds of expressions

```
        Binary of binop * expr * expr
        | Unary of unop * expr
        | Var of string
        | Num of int;;
```

and only the following kinds of statements

```
type stmt =
          Skip
        | Post of expr
        | Pre of expr
        | Assign of string * expr
        | Seq of stmt * stmt
```

In other words, in you can get the first 20pts for this assignment just by suporting straignt-line programs with simple arithmetic and pre/post conditions.

Part b: (20 points) To get the next 20 points, your program must additionally support conditionals, so you need to be able to correctly generate verification conditions for programs that use the construct.

```
  | Ifthen of expr * stmt * stmt
```

Part c: (25 points) In order to get full credit for the assignment, your code also needs to correctly support loops.

```
   | Whileloop of expr * expr* stmt
```

Note that the constructor for the loop includes an expression corresponding to the loop invariant in addition to the usual loop exit condition and loop body.

Part d: (Bonus 10 points) You can get 10 extra bonus points if in addition to the core language, your verifier also supports array reads and writes using the expression

```
| Arr of string * expr
```

and the statement

```
| ArrAssign of string * expr * expr
```

To claim these extra 10 bonus points, you need to provide 3 test programs that exercise the array functionality.

Part e: (Bonus 10 points) You can get an additional 10 bonus points if your system also supports universally quantified predicates for invariants, pre-conditions and post-conditions for programs involving arrays. Within your predicate, it will be assumed that any variable that starts with an underscore is universally quantified. So for example, your system should be able to prove the following program:

```
Pre( !( _i  < n) |   A[_i] == 0 );
i = 0;
while(i  < n){
Inv( i<=n & (!(_i < i) | A[_i] == _i) );
A[i] = i;
}
Post( !( _i  < n) |   A[_i] == _i );
```

Problem 4 Testing your verifier (15 points)

In addition to your verifier, you need to submit 5 tests that exercise all the constructs in your language.

MIT OpenCourseWare
http://ocw.mit.edu

6.820 Fundamentals of Program Analysis
Fall 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.