CS 6110 — Assignment 7

Gavin Gray u1040250

1. (**20 points**) The derivation of typing constraints is given in slide 27 of `2-type-analysis.pdf`. Following this method, do the Exercise on Slide 13 which is to generate typing constraints for this program:

```
main() {
    var x, y, z;
    x = input;
    y = alloc 8;
    *y = x;
    x = *y;
    return x;
}
```

Student Response: *Note : I will use '&' to represent the arrow (pointer) data type.*

```
-- For this program... --
main() {
    var x, y, z;
    x = input;
    y = alloc 8;
    *y = x;
    x = *y;
    return x;
}
------ ...the type constraints are: -------
[x] = [x]
[y] = [y]
[z] = [z]
[x] = [input] = int
[y] = &[y]
[*y] = [x] = int
[x] = [*y] = int // coherent with the above constraints
-----------------------------------------
```

2. (**10 points**) In Slide 26, suppose the complicated program has two changes: `*q = p - 1` instead of `*q = (*p) - 1`, and `return foo(n, foo).` instead of `return foo(&n, foo)`. Indicate how these changes will cause type-checking errors. Mention where the constraint checking will fail.

   Student Response: *The altered program is as follows:*

```
foo(p,x) {
    var f, q;
    if (*p == 0) { f=1; }       // ERROR [p] = int not &int
    else {
        q = alloc 0;
        *q = p - 1;
        f = (*p) * (x(q,x));    // ERROR [p] = int not &int
    }                           //  [q] = &int not int
    return f;
}
main() {
    var n;
    n = input;                  // [n] = [input] = int
    return foo(n, foo);         // [foo] = (int, [foo]) -> int
}
```

   As the comments in the above code snipped indicate, there are a few places where type constraints aren't met in this altered program. The first is when we try to dereference the variable 'p'. We can see starting in 'main' that `[n] = int = [p]`. This is also violated later at the statement `f = (*p) * (x(q, x))`. Additionally, on this same line, the type constraint for 'q' is not met because `[q] = &int` however the function signature for 'foo' requires that the first argument is of type `int`.

3. (**10 points**) Why are the two diagrams in Slide 13 of `3-lattices-and-fixpoints.pdf` not lattices? For each diagram, mention which definitions of a lattice are affected.

Figure 1: Example bad lattices from slides

Student Response: In the left lattice of 1 we can see that the main definition affected is that for every pair of vertices $a, b \in L \; \exists \; glb(a, b)$ which is not true for the top three vertices. In other words, it does not have a unique largest element.

In the right lattice of 1 the main definition affected by the structure is that for all $a, b \in L$ there does not exist a unique $glb(\{a, b\})$. For example, the two vertices on the third row of the lattice have a common glb which are both the vertices on the second row of the lattice[1].

4. (**10 points**) In `3-lattices-and-fixpoints.pdf`, Slide 21, the abstract interpretation of signs is given. What is the final answer (the value of `[exit]`) in this example?

Student Response: In this program `[exit]` will have the value of *top*. The reason for this, is both 'a' and 'b' are mapped to positive (+). Input produces a random variable that could either execute the first or second branch of the 'if'.

Given this, `[exit] = [c = a+b] U [c = a-b]` we can see that it gets boiled down to `[exit] = (+) U T => T`.

[1] These vertices are the ones circled in red on the slide if my description of "rows" was unclear.

5. (**10 points**) Listen to this Youtube video explaining how Facebook's Sparta system implements its abstract interpretation: `https://youtu.be/_fA7vkVJhF8`. You'll derive the most high-level insights by the 17th minute mark (but why not listen to the full video which is not much longer). Note down five key points from this presentation which tells you about what purposes this abstract interpreter serves. Each point may please be about two sentences.

   Student Response:

   - The first main concept of the abstract interpreter is the mapping from concrete values to something more abstract. In the first few examples the presenter shows mapping integers to simple intervals, this mapping allows for a reduction of state space.
   - To make things faster they have broken the analysis into stages, starting from the innermost loops then continuing towards the outermost loops. This can be achieved using Bourdoncle's algorithm which at a high level will identify nested strongly connected components.
   - Storing the program state at every basic block is very expensive (this is needed for fixedpoint analysis). To do this more efficiently they are using *immutable* Patricia Trees which only store the difference between basic blocks.
   - To handle arbitrarily complex loops a technique called widening is used. Widening is making large jumps after termination hasn't completed for a few iterations. These jumps will typically go to the "top of the lattice" which is the abstract value that encompasses all those below it (in the diagram given it was representative of all possible integers).
   - At a high level all of the previous concepts go into making the abstract interpreter. The goal of this project is to approximate the semantics of a program and verify it against certain classes of bugs (i.e. reaching the bottom of the lattice).

6. (**20 points**) This is to help you explore the `eva` plug-in and the `acsl` plug-in and add to my understanding of these features. There is no correct answer. Anything you find in these areas and report with certainty will add to our collective knowledge of static analysis (SA) features in Frama-C; that is the main goal of this exercise.

   You may get help as follows:

   ```
   frama-c -e-acsl-h -- obtains help
   frama-c -eva-h    -- obtains help
   ```

   In particular, I tried these. Please go through a similar exercise to understand these commands (or if you find better SA features, let me know):

   ```
   frama-c -eva arroob.c
   -- seems to catch array out of bounds. Plz confirm for me!

   frama-c -e-acsl first.i -then-last -print -ocode monitored_first.c
   -- generates code to run-time monitor assertions. Plz confirm for me!

   frama-c -eva -warn-invalid-pointer  warn-invalid-ptr.c
   -- Seems to warn invalid pointers. Plz onfirm for me!
   ```

   Student Response:

   - For the first example it does appear that 'frama-c' catches the invalid memory access. From the output of the program we can see

     ```
     [eva:alarm] arroob.c:5:
         Warning: accessing out of bounds index. assert i < 10;
     ```

     At the bottom of the output we can see the alarm info `1 access out of bounds index`.
   - For the second example it seems that `-e-acsl` generates run-time assertions that get injected into a C program. One thing that confuses me here is the inclusion of the assert $x == 1$. In the source file `first.i` there is a comment:

     ```
     /*@ assert x == 0; */
     /*@ assert x == 1; */
     ```

but I am unsure as to why we want to assert $x == 1$. Update, I realized that this assertion is to show how they are expanded and checked during execution. If we run the compiled binary there is in fact an assertion failure.

- In the third exercise we are testing the 'eva' plug-in's ability to detect invalid use of a pointer. Here we are incrementing a pointer which only points to a single integer value. Eva correctly raises an alarm against this because it is out-of-bounds.

7. (**20 points**)

```
frama-c -wp -wp-rte find_min_proved.c
```

This is a fully annotated program that is proven correct by Frama-C.

```
/*@ requires length > 0 && \valid(a+(0..length-1));
    assigns \nothing;
    ensures 0<=\result<length &&
      (\forall integer j; 0<=j<length ==> a[\result]<=a[j]);*/
int find_min(int* a, int length) {
  int min, min_idx;
  min_idx = 0;
  min = a[0];
  /*@ loop invariant 0<=i<=length && 0<=min_idx<length;
      loop invariant \forall integer j; 0<=j<i ==> min<=a[j];
      loop invariant a[min_idx]==min;
      loop assigns min, min_idx, i;
      loop variant length - i; */
  for (int i = 1; i<length; i++) { //*bug1* : change < to <=
    if (a[i] < min) {
      min_idx = i; //*bug2 : forget to set min_idx
      min = a[i];
    }
  }
  return min_idx;
}
```

1. Explain the annotations in the given program `find_min_proved.c`
2. Introduce the above-indicated bugs and study the manner in which Frama-C fails to verify
3. (I've not tried this) Modify `find_min_proved.c` to return "1" if max-min is greater than 4; else, return "0".

Student Response:

1. At the top level (function level) the annotations are stating
   - length must be greater than zero and 'a' must be at least 'length' long.
   - We aren't changing any of the values of 'a' or length.
   - The value returned is in the given length range and the value at that index is less than or equal to every other element in 'a'.

   At the loop level the annotations are saying
   - 'i' remains within the bounds of the array (except at the last iteration) and more importantly 'min_idx' is strictly within the bounds of the array.
   - For the elements "seen so far" it is guaranteed that 'min' is the minimum of them.
   - The value of 'min' is getting updated with 'min_idx'. i.e. if one of them changes the other one must as well.
   - The assigns clause tells frama-c that these variables could change in the loop body.
   - The variant clause is telling frama-c that 'i' is increasing.
2. When running the verification with the introduced bugs frama-c says that verifying the invariants "timed out". I find this output interesting as compared to what the dafny verifier gave us. One thing that makes me curious, is if it's possible to write a correct program that fails to verify in the allotted time (thus triggering a verification bug when there isn't one).
3. I sadly ran out of of time to finish this problem but I look forward to using frama-c in the future.