Advanced Programming Languages
Homework Assignment 5

The previous homework assignment was largely theoretical. This homework assignment is largely practical.

In class we have covered operational semantics (large- and small-step), axiomatic semantics (including verification condition generation) and some abstract interpretation. You are now qualified to pull ideas from many of those techniques together and create a non-trivial program analysis.

This analysis will target off-the-shelf C programs. We will use the `CIL` library to parse and process C programs into an IMP-like OCaml representation. We will use the `Z3` automated theorem prover of de Moura and Bjorner to reason about infeasible paths or otherwise decide questions of logic.

Our analysis will automatically generate test inputs that will force the subject program to cover all of its branches. This is undecidable in general (by direct reduction with the halting problem). Automated test input (and test case) generation is a research problem that is receiving quite a bit of attention; see the papers on the course webpage for more information. This problem is also known as *program reachability*, the same issue we discussed in the context of software model checking.

I have provided an introductory analysis. It performs flow-sensitive, path-sensitive, context-insensitive, intraprocedural path enumeration, symbolic execution, and constraint generation to create test inputs. Unlike previous assignments, for this assignment you *may change anything* in the main file you like.

Initial steps:

1. Compile `tigen`, our test-input generation program. See the file `README-GradPL.txt` in the code archive for details. Reproducing the systems research results of others is a key part of modern research. Getting the code up and running is explicitly part of this assignment and is *your responsibility.* Talk to your friends, post on the forum, scour the Internet — do whatever it takes to make it happen.

2. Take a look at some of the included small subject programs. Think about how you would generate test inputs to reach and cover all of

their branches. Then run tigen and inspect the test cases it actually creates.

As a sanity check, on my machine, the report.txt output is:

```
      array-1.i.c : reached   0 /   4
      array-2.i.c : reached   8 /   8
      array-3.i.c : reached   0 /  10
      array-4.i.c : reached   1 /   4
     balanced.i.c : reached   4 /   8
    bsearch-1.i.c : reached   4 /   8
    bsearch-2.i.c : reached   6 /   8
   bubblesort.i.c : reached   5 /   6
       conway.i.c : reached   6 /   6
      dotprod.i.c : reached   2 /   2
     eq-index.i.c : reached   6 /   6
         fibo.i.c : reached   6 /   6
      float-1.i.c : reached   2 /   6
      float-2.i.c : reached   7 /   8
      float-3.i.c : reached   5 /   8
      float-4.i.c : reached   4 /  14
      float-5.i.c : reached   1 /   4
          gcd.i.c : reached   4 /   4
    hailstone.i.c : reached   5 /   6
        leven.i.c : reached   7 /  10
many-features.i.c : reached   0 /   8
       matmul.i.c : reached   3 /   6
  matrix-path.i.c : reached   0 /   8
      mymin-1.i.c : reached   7 /   8
        roman.i.c : reached  21 /  28
      selsort.i.c : reached   6 /   6
     simple-1.i.c : reached   2 /   2
     simple-2.i.c : reached   4 /   4
     simple-3.i.c : reached   8 /   8
     simple-4.i.c : reached   5 /   6
       string.i.c : reached   0 /   8
     struct-1.i.c : reached   3 /  10
     struct-2.i.c : reached   0 /   8
     struct-3.i.c : reached   0 /  18
 struct-float.i.c : reached   0 /  16
    subseqsum.i.c : reached   0 /  12
```

3. Read some of the papers associated with this homework on the course webpage.

Exercise 1: This exercise is open-ended. You must do something to convince me that you have an integrated understanding of the theory and practice of using PL research techniques to analyze programs. More concretely, you must modify `tigen.ml` so that it is "better" in a way of your choosing. As a rough estimate, I would expect a `diff` of your modified source to indicate at least 200 changed lines. Then you must write up a formal three- or four-paragraph explanation of what you did and why it was worthwhile. Your explanation should motivate your changes and explain why the problem you tackled is important.

Any of the following could suffice:

- Modify tigen so that it handles string-valued data by integrating with the DPrle external decision procedure.

- Modify tigen so that it handles loops in an intelligent manner. For example, you might use a dataflow-style join – if it is to possible reach the loop head knowing $x = 0 \wedge y = 55$ and it is also possible to reach the loop head knowing $x = 5 \wedge y = 55$, you should process the loop in a state where $y = 55$ (or, better yet, $x \geq 0 \wedge y = 55$).

- Modify tigen so that it handles arrays. Note that Z3 already has built-in handling for the McCarthy select and update axioms, but you'll have to integrate it.

- Modify tigen so that it handles the heap (i.e., dynamic pointers) more precisely. For example, you might introduce an explicit handling of `malloc` (which either returns 0 or a new non-zero address that is distinct from all previous addresses) and `free`.

- Modify tigen so that it uses computed alias information. CIL comes with John Kodumal's implementation of Manuvir Das' One-Level Flow alias analysis to aid in reasoning about pointers, but it is not currently used in this project. As a hint, alias analysis information leads directly to "distinctness" constraints. This would be a relatively short change, so you should also do something else and/or provide compelling examples to show that the alias analysis really helps.

- Modify tigen so that its performance and scalability are non-trivially improved. This typically requires more "engineering" than "theory", but getting an analysis to run on millions of lines of code (e.g., the Linux kernel, SQL Server) is very difficult. Your modified version should run significantly faster on large benchmarks of your choosing.

- Modify or post-process tigen so that the performance of its generated test inputs is non-trivially improved. That is, perform *test suite selection* or *test suite reduction* or even *time-aware test suite prioritization.* Ideally we would like the smallest number of test cases that require the smallest amount of wall-clock time to execute but still covert the greatest fraction of the subject programs. This is a reasonable project if you are more interested in CS theory than in systems hackery.

- Modify tigen to handle record data types (e.g., structures and/or unions).
- Modify tigen to handle floating-point data types. In practice, this ends up being insufficient for full credit for most students, as they "merely" use Z3's Real datatype without considering the differences between real numbers in mathematics and IEEE floating point numbers. Think carefully before you select this option.
- Modify tigen to accept additional constraints provided by the user (e.g., pre- and post-conditions on the subject program, an external constraint language that you parse at the beginning, or whatever you like). For example, you may want to specify that you're only interested in test inputs involving negative numbers. Part of a larger project might be to have tigen output multiple *diverse* test inputs that cover the same path.
- Modify tigen to be context-sensitive. You might compute the call graph and analyze the functions in reverse dependency order. You might do a full-blown CFL reachability analysis. Or you might just start in the target function and take very long paths through the entire reachable program. Handling recursion is a related topic.
- Modify tigen so that it implements key ideas from Godefroid *et al.*'s DART project. This would presumably involve some notion of random test input generation (see their paper).
- Modify tigen so that it implements key ideas from Sen *et al.*'s CUTE project. This would presumably involve some notion of concolic testing (see their paper).
- Modify tigen so that it implements key ideas from Lakhotia *et al.*'s AUSTIN project. This would presumably involve some of the empirical optimizations and best practices described in their paper.

You should submit

1. your modified `uniquename-tigen.ml` file
2. a PDF text file containing your multi-paragraph report
3. as well as any other compelling figures or charts that you feel back up your case

4. two (2) new subject programs in the style of the subject programs already included. Your homework should perform well on those new subject programs.

5. a PDF text file containing your email to a tool author (see below)

Recall that you should demonstrate that you did something useful with respect to this homework's goals of using program analysis techniques either (1) in your research or (2) to understand programs or (3) to find bugs or (4) to verify properties of programs or (5) to make related tools more usable.

Exercise 2: Compose a brief (one- or two-paragraph) email to one of the authors of the tools or papers you used in this homework and include the text of it in your submission. In addition, indicate whether you are willing to use your name or whether you would like to be portrayed as an anonymous student in my class. I will check off the fact that you wrote something and potentially forward it along. You can comment on any aspect of your experience with their work — your comments need not be positive. For example, you might ask Bjorner why Z3 doesn't handle multiplication, complain to Kodumal or Das that OLF isn't precise enough for C programs, or tell Necula or McPeak that you find CIL's memory lvalue semantics unintuitive. You might even write to Lakhotia and ask him how he managed to scale to large programs given all the difficulties you observed when wrestling with C. If you do offer criticism, strive to make it constructive by commenting on what you would have liked to have seen instead or how you might like to see things improved if the time were available. If you absolutely cannot think of anything to say, thank them for making their tools available and let them know that you used them with success. Even minor comments about documentation or a fresh-eyed perspective on usability can be helpful.

The purpose of this non-standard exercise is two-fold.

- First, I have observed multiple instances in this class of a student being unwilling to contact the author of some publicly-available project. While I realize that you don't want to be known as a whiny grad student who didn't bother to read the manual, it's also not worth wasting your time to try to decipher a research prototype when the author is only an email away. I think it would legitimately be good practice for many of you to correspond with a random researcher. You may not get a response, but the sky won't fall. (In addition, I know the people involved in all of this software and they are all quite friendly.)

- Second, internships are not the only way to build up contacts and networks. It is entirely reasonable to grow a friendship or collaboration with someone over time, starting with a lowly email about research, moving on to chatting a conferences, and eventually working together on new research. You're rarely certain of exactly where you will end up or what you will be working on, so it behooves you to know as many people out there as possible.