CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 8 (10 Points)

Due: 3. February 2023

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you must not delete any provided ones. You can verify whether your submission is valid by calling python3.

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise: Running in the Deep (10 Points)

The debugging approaches we've learned so far are more or less based on the assumption that the execution ends in finite time. But this is not always true: dead loops make your code run forever -- computer scientists call it **nontermination**. Detection of such dead loops during the development is indispensable. In this exercise, we are going to extend the `PerformanceDebugger` to allow programs that may loop forever, and use this debugger to help us detect nontermination.

a. Limit the Power (4 Points)

The very first problem of debugging nonterminating programs is: they don't terminate! Thus, our tracers and profiles run forever, too! We can break this situation by supplying limited power to the runner: Set up an upper bound as the *maximum* number of steps the program is allowed to execute. When the bound exceeds, *immediately abort* the execution; the profiling results will be computed based on the recorded execution steps.

To realizes this functionality, extend the `HitCollector` and the `PerformanceDebugger` class in `debugger.py`. For the `HitCollector` class:

1. Support an optional argument `limit: int` in the constructor, meaning the maximum number of steps we allow the program to execute; set the default value to `100000`.
2. Maintain a counter (initialized by `limit`) that decreases by `1` each time `collect` is invoked. When the counter reaches zero, immediately raise an `OverflowError`.

In this way, the `HitCollector` will raise an `OverflowError` if the profiling program is nonterminating.

For the `PerformanceDebugger` class:

1. Catch this `OverflowError` (in the `__exit__` function) so that we don't pass this intentionally-raised exception to the main thread -- otherwise you will not be able to use the instance `PerformanceDebugger` when the code finishes executing.
2. Implement an instance method `def is_overflow(self) -> bool` that returns whether the `OverflowError` was raised or not.

Test your implementation against a trivial nonterminating function:

```
def loop(x: int):
    x = x + 1
    while x > 0:
        pass

with PerformanceDebugger(HitCollector) as debugger:
    loop(1)

assert debugger.is_overflow()
print(debugger)
```

The debugger should print out the profiling results, like this:

```
   1   0% def loop(x: int):
   2   0%     x = x + 1
   3  49%     while x > 0:
   4  49%         pass
```

b. Testing and Reasoning (6 Points)

It's time to apply the debugger you implemented in (a) and the testing skills you've learned so far on several example functions containing loops (you can also find them in `examples.py`):

In [3]:
```
def ex_1(i: int):
    while i < 0:
        i = i + 1

def ex_2(i: int):
    while i != 1 and i != 0:
        i = i - 2

def ex_3(i: int, j: int):
    while i != j:
        i = i - 1
        j = j + 1

def ex_4(i: int):
    while i >= -5 and i <= 5:
        if i > 0:
            i = i - 1
        if i < 0:
            i = i + 1

def ex_5(i: int):
    while i < 10:
        j = i
        while j > 0:
            j = j + 1
        i = i + 1

def ex_6(i: int):
    c = 0
    while i >= 0:
        j = 0
        while j <= i - 1:
            j = j + 1
            c = c + 1
        i = i - 1
```

Your task is to judge if each function terminates **on all inputs** (i.e., all integers). You don't have to understand the code at the very beginning, but instead test them against a number of inputs (say, randomly-generated) and use your debugger to profile how often each line has been executed. Then make educated guesses based on your observations. Finally, double check your guesses via "human static analysis".

Report your answers in `report.md`, and explain the reasons -- specifically,

- for functions that you think are terminating on all inputs: explain why the loop condition will eventually not hold at any input;
- for functions that you think are not terminating:
  1. provide a **concrete** input as an evidence for nontermination,
  2. show the frequencies of each line collected by your debugger,
  3. point out which loop(s) does not terminate (if the function indeed contains multiple loops), and

4. explain why the loop condition will hold forever after a certain number of iterations.