CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 1 (10 Points)

Due: 20. November 2022

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 1-1: Eww, a Bug! (4 Points)

In this exercise you need to detect bugs in a faulty implementation of a conversion from HEX colors to RGB values, which you can find in **exercise_1.py**.

a. Off by One (2 Points)

1. The implementation of the conversion produces in some cases an unwanted behavior when handling `AssertionError`. Find the conditions under which this error is unwantedly triggered or not triggered and provide a value for the argument `hex_str` of `convert_to_rgb()`. We provide a small specification for you:
   A. The code should trigger an `AssertionError` if the length of `hex_str` is not equal to six.
   B. The code should trigger an `AssertionError` if a non-hexadecimal character is included in the string, i.e., it should only accept `A-F`, `a-f`, and `0-9`.
   C. The code should not trigger an `AssertionError` if the length and characters of `hex_str` are valid.
2. Besides the unwanted behavior with `AssertionError` the function seems to provide incorrect results in some cases. Provide a value for the argument `hex_str` that triggers this functional error.

Please provide the arguments in **exercise_1a.py** where `hex_str_assertion_error` should reveal the additional or missing `AssertionError` and `hex_str_functional_error` should return an incorrect result when feeding it to `convert_to_rgb()`

b. Kill it with Fire! (2 Points)

Fix both unwanted behaviors and provide a correct version of the program. Implement your fix in **exercise_1b.py**. Try to make as few changes as possible.

Exercise 1-2: A Little Tracer for Recursion (6 Points)

Recursive functions are ubiquitous in programming. One useful method of debugging such functions is to keep track of the input argument(s) and the return values of all recursive calls. In this exercise, let's consider only non-mutual recursive functions, i.e., this function `f` does not call another function `g` that may call `f`.

The well-known Fibonacci function belongs to this kind (`exercise_2.py` contains the definition):

In [9]:
```
def fib(n: int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

A slightly complicated example is merge sort (`exercise_2.py` contains the definition):

In [10]:
```
def merge_sort(arr, l, r): # main function
    if l < r:
        m = l + (r - l) // 2

        merge_sort(arr, l, m)        # <--- trace this
        merge_sort(arr, m + 1, r)    # <--- trace this
        merge(arr, l, m, r)          # <--- do not trace this!

    return arr
```

a. RecursiveTracer (4 Points)

The `Tracer` class you learned from this section of the debugging book is already out-of-the-box. Your job is to extend the `Tracer` class into a `RecursiveTracer` class, which takes an additional argument `func: Callable` -- the recursive function we are tracing -- and outputs an *indented* log of tracing the calls of `func` (but not any other function):

1. Each time a recursive call is entered, increase the indentation and print out the arguments (format `call with <formal parameter 1> = <argument value>, <formal parameter 2> = <argument value>, ...`);
2. Each time a recursive call is returned, decrease the indentation and print out the return value (format `return <return value>`);
3. Each level of indentation contains **2 whitespaces**.

Note that the order of the arguments must be the same with the declaration order in the original function. You may call this helper function to obtain the names of the formal parameters of a function as a string list:

In [11]:
```
def param_names(func: Callable):
    return inspect.getfullargspec(func).args
```

In [12]:
```
# example usages
print('params of fib:', param_names(fib))
print('params of merge_sort:', param_names(merge_sort))
```

```
params of fib: ['n']
params of merge_sort: ['arr', 'l', 'r']
```

By the way, getting the name of a function (of type `Callable`) is rather simple:

In [13]:
```
merge_sort.__name__
```

Out[13]:
```
'merge_sort'
```

Implement the `RecursiveTracer` class in `exercise_2a.py`. Run the test cases by typing `python3 exercise_2a.py` in your shell. The expected output logs are also included in this Python file; compare your outputs against these. You can also find an example output below:

```
call with n = 4
  call with n = 3
    call with n = 2
      call with n = 1
      return 1
      call with n = 0
      return 0
    return 1
    call with n = 1
    return 1
```

```
    return 2
    call with n = 2
      call with n = 1
      return 1
      call with n = 0
      return 0
    return 1
  return 3
```

b. Back in my Days (2 Points)

Dynamic tracing is a nice feature of Python but is usually not available in other programming languages. Outside of the Python world, such tracing is accomplished by injecting code into the program, i.e., adding new statements that perform the logging. In this exercise, you need to inject static statements to the `fib()` and `merge_sort()` functions manually to produce the identical logging from above. You can modify the code to your liking; you can add new instructions and modify existing ones as long as the functions produce the same result and logging. However, try to make as few changes as possible. In addition, check this section from the Debugging Book to get a feeling where to add instructions. . Please implement your solution in **exercise_2b.py**. Consider the following hints:

1. Leverage the `level` variable and its corresponding functions `increase_level()` and `decrease_level()` in **exercise_2b.py** to keep track of the level.
2. Make sure to increase and decrease the level when required consistently.
3. You need to add statements in multiple locations, not only at the beginning of the function.
4. You can add functions that help you with the output.
5. Sometimes, you should modify statements to assign intermediate results to temporary variables.

Use the `log()` function in **exercise_2b.py** to produce the output. You must not modify this function because we will use it for evaluation.