CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 5 (10 Points)

Due: 04. January 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 3-1: Is this True or Can I Clean Up? (10 Points)

Up to this point we investigated statistical debugging for coverage feedback. However, statistical debugging can also correlate the outcome of predicates to the occurence of failures.

In this exercise, you will introduce predicates for function parameters of type `int` and `float`, to compare them with `0` and against each other.

In [1]:
```
class Predicate:

    def __init__(self, rpr: str, failing_true: int = 0,
                 successful_true: int = 0, true: int = 0, failing_observed:
                 successful_observed: int = 0, observed: int = 0):
        self.rpr = rpr
        self.failing_observed = failing_observed
        self.successful_observed = successful_observed
        self.observed = observed
        self.failing_true = failing_true
        self.successful_true = successful_true
        self.true = true

    def __repr__(self):
        return (f'{self.rpr}(F(p)={self.failing_true}, S(p)={self.successfu
                f'F(p observed)={self.failing_observed}), S(p observed)={se

    def __str__(self):
        return self.__repr__()

    def __hash__(self):
        return hash(self.rpr)
```

```
    def __eq__(self, other):
        return isinstance(other, Predicate) and self.rpr == other.rpr
```

a. Measuring the World (3 Points)

To accomplish the implementation of predicates, we need a new metric that can handle the three different outcomes that we can observer for predicates. These three outcomes are `True`, `False`, and not observed. For this exercise, implement the following functions in **exercise_1a.py** based on the provided formulas.

- failure(p: Predicate) -> float:

$$\frac{F(p)}{F(p) + S(p)}$$

- context(p: Predicate) -> float:

$$\frac{F(p \text{ observed})}{F(p \text{ observed}) + S(p \text{ observed})}$$

- increase(p: Predicate) -> float:

$$\text{failure}(p) - \text{context}(p)$$

Where $F(p)$ donates to the number of failing runs for which p is observed to be `True` and $S(p)$ to the number of successfull runs in which p is `True`. $F(p \text{ observed})$ and $S(p \text{ observed})$ donate the number of failing and successfull runs where p is observed (can be either `True` or `False`). You will get all information needed to calculate all three functions from the `Predicate` class.

In cases where a division by `0` can occur, return `0` instead.

In [2]:
```
def failure(p: Predicate) -> float:
    return 0  # TODO: implement

def context(p: Predicate) -> float:
    return 0  # TODO: implement

def increase(p: Predicate) -> float:
    return 0  # TODO: implement
```

b. Do You Wanna See My... Collection? (4 Points)

In this exercise, you need to implement the `collect()` function of a `Collector` that compares all `int` and `float` parameters of a function when called against each other based on the following comparisons:

- `<`: checks whether one argument is smaller than another.
- `>`: checks whether one argument is greater than another.
- `==`: checks whether one argument is equal to another.

In addition, you need to check all `int` and `float` parameters for the value `0`, i.e., compare all parameters with all three comparators against `0`.

To accomplish this task, you must implement the `collect(self, frame: FrameType, event: str, arg: Any) -> None` of the `PredicateCollector` class in **exercise_1b.py**. The class introduces a dictionary `self.predicates` in its constructor, where you should add the collected predicates. The dictionary maps from a string representation of the predicate to the predicate itself. The string representation should look like the following:

```
function_name(arg_name op (arg_name|0))
```

Since you cannot determine whether the run is successful or failing at this point, you should set `Predicate.p` to `1` if the predicate evaluates to true and `Predicate.observed` to `1` if you observe the predicate. Note that you do not need to count how often you encounter the predicate or it evaluates to true.

We make an example based on the following function:

In [6]:
```python
def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))
```

When collecting from the execution of `ackermann(0, 1)`, the `PredicateCollector.predicates` should look like this after the execution:

```
{'ackermann(m == 0)': ackermann(m == 0)(F(p)=0, S(p)=0), ,F(p observed)=0),
S(p observed)=0, p=1, p observed=1)),
 'ackermann(m < 0)': ackermann(m < 0)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=0, p observed=1)),
 'ackermann(m > 0)': ackermann(m > 0)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=0, p observed=1)),
 'ackermann(m == n)': ackermann(m == n)(F(p)=0, S(p)=0), ,F(p observed)=0),
S(p observed)=0, p=0, p observed=1)),
 'ackermann(m < n)': ackermann(m < n)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=1, p observed=1)),
 'ackermann(m > n)': ackermann(m > n)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=0, p observed=1)),
 'ackermann(n == 0)': ackermann(n == 0)(F(p)=0, S(p)=0), ,F(p observed)=0),
S(p observed)=0, p=0, p observed=1)),
 'ackermann(n < 0)': ackermann(n < 0)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=0, p observed=1)),
 'ackermann(n > 0)': ackermann(n > 0)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=1, p observed=1)),
 'ackermann(n == m)': ackermann(n == m)(F(p)=0, S(p)=0), ,F(p observed)=0),
S(p observed)=0, p=0, p observed=1)),
 'ackermann(n < m)': ackermann(n < m)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=0, p observed=1)),
 'ackermann(n > m)': ackermann(n > m)(F(p)=0, S(p)=0), ,F(p observed)=0), S(p
observed)=0, p=1, p observed=1))}
```

Note that the dictionary can be in a different order when you run your collector.

Here are some tips that you should consider:

- `inspect.getargvalues(frame)` gives you a dictionary of arguments to their values.
- You should include semantic duplicates, like `ackermann(m < n)` and `ackermann(n > m)`.
- You should exclude comparisons of a variable with itself.
- You can approach the implementation as follows:
  - Iterate over all arguments and their values and verify their types.
  - Add the predicates that compare against `0` for this argument.

- Make a nested iteration of all parameters.
- Check whether the first and second arguments are different and check for the types.
- Add the predicates that compare these two arguments.
- Party 🥳

c. Debugging with Predicates (3 Points)

Now it is time to combine the collected predicates. Implement the `all_predicates(self) -> Set[Any]` of the `PredicateDebugger` class in **exercise_1c.py** that returns a set of predicates.

You only need to consider collectors whose outcome is `PredicateDebugger.pass` or `PredicateDebugger.fail`. Combine all predicates of the collectors in the following way: Iterate the collectors and get the predicates. For each predicate, create a new predicate if you do not already have the predicate in your result. If the run is passing, increase the predicate's in the result `successful_true` and `successful_observed` based on `true` and `observed` of the current predicate. If the run is failing, increase the failing variable instead. Also, increase `true` and `observed` of the predicate in the result.

Consider the following example:

```
pd = PredicateDebugger()

with pd.collect_pass():
    ackermann(3, 3)
with pd.collect_pass():
    ackermann(0, 0)
with pd.collect_fail():
    ackermann(0, 1)
```

`pd.all_predicates()` should return the following set:

```
{ackermann(m < 0)(F(p)=0, S(p)=0), ,F(p observed)=1), S(p observed)=2, p=0, p
observed=3)),
 ackermann(m < n)(F(p)=1, S(p)=1), ,F(p observed)=1), S(p observed)=2, p=2, p
observed=3)),
 ackermann(m == 0)(F(p)=1, S(p)=2), ,F(p observed)=1), S(p observed)=2, p=3, p
observed=3)),
 ackermann(m == n)(F(p)=0, S(p)=2), ,F(p observed)=1), S(p observed)=2, p=2, p
observed=3)),
 ackermann(m > 0)(F(p)=0, S(p)=1), ,F(p observed)=1), S(p observed)=2, p=1, p
observed=3)),
 ackermann(m > n)(F(p)=0, S(p)=1), ,F(p observed)=1), S(p observed)=2, p=1, p
observed=3)),
 ackermann(n < 0)(F(p)=0, S(p)=0), ,F(p observed)=1), S(p observed)=2, p=0, p
observed=3)),
 ackermann(n < m)(F(p)=0, S(p)=1), ,F(p observed)=1), S(p observed)=2, p=1, p
observed=3)),
 ackermann(n == 0)(F(p)=0, S(p)=2), ,F(p observed)=1), S(p observed)=2, p=2, p
observed=3)),
 ackermann(n == m)(F(p)=0, S(p)=2), ,F(p observed)=1), S(p observed)=2, p=2, p
observed=3)),
 ackermann(n > 0)(F(p)=1, S(p)=1), ,F(p observed)=1), S(p observed)=2, p=2, p
observed=3)),
 ackermann(n > m)(F(p)=1, S(p)=1), ,F(p observed)=1), S(p observed)=2, p=2, p
observed=3))}
```