Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 4 (10 Points)

Due: 21. December 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 4-1: Do You Remember? (5 Points)

Do you remember *Exercise 1-2*? Please read this exercise again. You implemented a dynamic tracer and manually injected logging statements. Your goal for this exercise is to leverage a `NodeTransformer` to instrument the code and achieve the same results. Implement your solution in **exercise_1.py**.

As suggested by the debugging book, an alternative (but more efficient) way to implement a tracer is to statically inject code into the traced program. Now, you need to reimplement the `RecursiveTracer` class, but via AST transformation. You should first read this section to get a better understanding of how ASTs are transformed in Python. Your task is to implement a `Transformer` class that extends `NodeTransformer` from Python's `ast` module:

In [2]:
```python
from ast import FunctionDef, NodeTransformer

def log(*objects: Any):
    print(*objects)

class Transformer(NodeTransformer):
    def visit_FunctionDef(self, node: FunctionDef) -> FunctionDef:
        self.ori_name = node.name
        self.traced_name = node.name + '_traced'
        # TODO: your implementation
```

Call the provided `log` function for output in your transformed code. Please **don't** call `print` because the grading script will modify the body of `log` (but the type signature is kept). Also, **don't** modify `self.ori_name` and `self.traced_name` because the grading script will use them.

To figure out which AST transformations are required for tracing recursive calls, you may take an example function and do the transformation manually. In the following, we introduce one such possible method for your reference, using the `fib` function as an example. Here is the transformed version `fib_traced` :

In [3]:
```
def returned(return_val: Any, level: int) -> Any:
    log('  ' * level + f"return {repr(return_val)}")
    return return_val

def fib_traced(n: int, level: int = 0) -> int:
    log('  ' * level + f"call with n = {n}")

    if n == 0:
        return returned(0, level)
    if n == 1:
        return returned(1, level)
    return returned(fib_traced(n - 1, level + 1)
                    + fib_traced(n - 2, level + 1), level)
```

In [4]:
```
_ = fib_traced(4)
```

```
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
```

Compared with `fib`, the following changes are introduced:

1. An additional argument `level` (with default value `0`) is used to keep track of the indentation level, or equivalently, the call stack depth.
2. At the beginning of the function body, a `log` statement (this `log` function provided by us) is inserted to output the calling arguments (following the format we defined earlier).
3. Every recursive call to `fib` is changed to `fib_traced` and the increased level `level + 1` is passed as the additional argument.
4. Every return value (or expression) is wrapped with `returned`, a helper function that outputs the return value (following the format we defined earlier).

Now, it's your turn to generalize the above changes into the needed transformation on the AST of the original function. You may use functions `parse_expr` and `parse_stmt` (see **exercise_1.py**) to directly create ASTs from Python source code:

In [5]:
```
from ast import expr, stmt
from exercise_1 import parse_expr, parse_stmt
from ast import dump
```

```python
e: expr = parse_expr('level + 1')
print(dump(e, indent=2))

s: stmt = parse_stmt('return foo(x)')
print(dump(s, indent=2))

# Note in Python, an expression `e` is also a statement `Expr(e)`
s1: stmt = parse_stmt('foo(x)')
print(dump(s1, indent=2))
```

```
BinOp(
  left=Name(id='level', ctx=Load()),
  op=Add(),
  right=Constant(value=1))
Return(
  value=Call(
    func=Name(id='foo', ctx=Load()),
    args=[
      Name(id='x', ctx=Load())],
    keywords=[]))
Expr(
  value=Call(
    func=Name(id='foo', ctx=Load()),
    args=[
      Name(id='x', ctx=Load())],
    keywords=[]))
```

Exercise 4-2: Slice It in Half! (5 Points)

In this exercise, you will reduce the source code of a program or items based on the slice of executions to produce a functional part of the code that reproduces the same results for the dependencies in this slice.

As an example, consider the `middle()` function below.

In [10]:
```python
def middle(x, y, z):  # type: ignore
    if y < z:
        if x < y:
            return y
        elif x < z:
            return y
    else:
        if x > y:
            return y
        elif x > z:
            return x
    return z
```

When calling this function with `middle(2, 1, 3)`, we would retrieve the following slice:

In [12]:
```python
with Slicer(middle) as slicer:
    middle(2, 1, 3)
slicer.code()
```

```
*     1 def middle(x, y, z):  # type: ignore
*     2     if y < z:  # <= z (1), y (1)
*     3         if x < y:  # <= x (1), y (1); <- <test> (2)
      4             return y
*     5         elif x < z:  # <= z (1), x (1); <- <test> (3)
*     6             return y  # <= y (1); <- <test> (5)
      7     else:
      8         if x > y:
      9             return y
     10         elif x > z:
     11             return x
     12     return z
```

The reduced function would then look something like this.

```python
def middle(x, y, z):
    if y < z:
        if x < y:
            pass
        elif x < z:
            return y
```

To reduce the code, you need to implement the `ReduceSlicer.reduce(self, sources: Dict[str, Tuple[str, int]]) -> str` method in the `ReduceSlicer` class. `ReduceSlicer.reduce(self, sources: Dict[str, Tuple[str, int]]) -> str` gets a dictionary of item names to the source code of an item and the line the source begins (you need both and remember the `__name__` field on the first exercise). Your task is to find the essential lines based on the data and control-flow dependencies collected by the slicer; we recommend looking at the representation of said dependencies. You can access the dependencies with `self.dependencies().data` and `self.dependencies().control` inside the `ReduceSlicer.reduce(self, sources: Dict[str, Tuple[str, int]]) -> str` method.

We provide you with a `NodeTransformer` in the `Reducer` class that you can leverage to build the final code. You can use the `Reducer` in your `ReduceSlicer.reduce(self, sources: Dict[str, Tuple[str, int]]) -> str` method like the following:

```python
result = ''

# For each item
reducer = Reducer('set of line numbers that are essential for this item')
# The signature of this constructor is Reducer.__init__(lines: Set[int])
tree = ast.parse('source of the current item')
# You need to put the actual source string of the item here that you get in sources
new_tree = reducer.visit(tree)
result += ast.unparse(new_tree)
```

The output of `ReduceSlicer.reduce(self, sources: Dict[str, Tuple[str, int]]) -> str` should be a single string that comprises all items reduced with the help of the slice.

The reduced part only needs to produce results for the execution of the slice and should result in `None` for other executions.

You may encounter empty blocks when reducing an item, but do not hesitate; the `Reducer` will automatically add a `pass` statement in those blocks to make your code executable.

Implement your solution in **exercise_2.py**.

The `ReduceSlicer` will be handled in the same way as a `Slicer`:

```python
with ReduceSlicer(middle) as rs:
    m = middle(2, 1, 3)
```

The following code reduces the code:

```
_, start = inspect.getsourcelines(middle)
f = rs.reduce({middle.__name__: (inspect.getsource(middle), start)})
```

With `exec(f)` we can overwrite the existing items and use the newly reduced items instead.

Currently, there is a bug in the `debuggingbook` pip package within the `Slicer` module. If you encounter the following error:

```
NameError: name '_data' is not defined
```

You need to add the following fix to the python file where the `Slicer` is used:

```
from debuggingbook.Slicer import DependencyTracker
_data = DependencyTracker()
```

We already added these lines to the beginning of **exercise_2.py** to enable you a safe start.