# Program Analysis

## – Project Description, Winter Semester 2023/24 –

Prof. Dr. Michael Pradel, Aryaz Eghbali, Beatriz Souza

November 4, 2023

## 1 Introduction

Program slicing is a type of program analysis that has multiple applications, one of which is locating the root cause of a bug. In slicing, a subprogram is extracted based on some slicing criterion to reduce the size of the code, which facilitates the downstream tasks. There are multiple slicing algorithms, some of which are discussed during the lecture.

## 2 Project Description

The goal of this project is to implement **dynamic backward slicing** for Python. The exact choice of the slicing algorithm is up to you, and in particular, you are not restricted to algorithms discussed in the lecture. The project will be based on DynaPyt[1], a general purpose framework for dynamically analyzing Python. Your implementation should also be in Python. Given a Python program and a slicing criterion, the final output of your analysis should keep only the code needed for the slice, e.g., by removing subtrees from the AST that are not part of the slice. To reduce the given source code into the slice, libraries like LibCST[2] will be helpful for parsing, manipulating, and generating code.

### 2.1 Scope and Limitations

To help students focus on the main analysis and avoid the hassle of dealing with all of the language features of Python, we narrow down the scope of programs that your implementation should be able to handle:

- You will implement an intra-procedural analysis, which means your analysis is limited to the execution inside a function. You can assume that any function that is called during the execution of the analyzed function has the following side-effects:
  - There is a data flow from any argument to the return value. E.g., `z = foo(x)` introduced a data flow from `x` to `z`.
  - If the call is a method call, then there is a data flow from the object on which the method is called to the return value, from the arguments to the object on when the method is called, and from the arguments to the return value. E.g., `z = y.foo(x)` introduces data flows `y` $\rightarrow$ `z`, `x` $\rightarrow$ `y`, and `x` $\rightarrow$ `z`.
  - There are no other side-effects, i.e., a called functionis assumed to not change the values of any non-local variables or objects.
- Calls to `exec` and `eval`, and `with` statements are considered out of the scope of this project, and will not appear in our test cases.
- The left-hand side of every assignment (or augmented assignment) is either a variable (`foo=`), an attribute access (`foo.bar=`), or an index access (`foo[bar]=`).

[1] `https://github.com/sola-st/DynaPyt`
[2] `https://github.com/Instagram/LibCST`

- Your analysis should be able to handle complex objects and arrays, but you can assume that any modifications to the object or array causes a dataflow edge to any read of attributes or any index accesses of the object or array.
- When slicing the function, all function arguments remain intact, even if an argument is not needed in the slice.
- There are no definitions of other functions or classes inside the analyzed function.

# 3 Milestones

The project is broken down into three milestones for better guidance throughout the semester. Each milestone has a progress meeting, in which each student meets with her/his mentor, presents the deliverable, and seeks help if needed. Please create a **private** GitHub repository for your project and give read access to your mentor. Some code templates have been prepared for this project, on top of which you should implement your solution. The template is provided as a Python package name `dynamicslicing`. Please adhere to the output format shown in the provided test cases, as the output of your program will be automatically tested.

## 3.1 Milestone 1 (Progress Meeting in Week of Nov 20–24, 2023)

In the first part of the project you will familiarize yourself with DynaPyt by implementing a simple dynamic analysis. Furthermore, you will also implement utility methods that are required for manipulating the source code and for printing the final result. On the progress meeting of this milestone, course participants are expected to have implemented the following:

1. A dynamic analysis in DynaPyt that each time a variable is being written to, prints the value being written. This analysis must be implemented in `dynamicslicing/trace_writes.py`. For example, in the following program

   ```
   def slice_me():
       x = 5
       print("Hello World")
       if x < 10:
           x += 5
       y = 0
       return y

   slice_me()
   ```

   the output (including both analysis output and program output) should be

   ```
   5
   Hello World
   10
   0
   ```

   As a starting point, we suggest working through this tutorial.

2. A function that, given the path of a Python file and a set of code lines to keep, removes nodes from the syntax tree that are not on these lines. To this end, the function should parse the code, e.g., with LibCST, manipulate the syntax tree, and then return the new code. This function should be implemented in `dynamicslicing/utils.py` as the `remove_lines` function. For example, given the above source code and lines `[1, 2, 4, 5, 9]`, the function should produce

   ```
   def slice_me():
       x = 5
       if x < 10:
           x += 5

   slice_me()
   ```

Note that the correctness evaluation is based on LibCST's `deep_equals` on the root of the syntax trees. For your reference, there is a sample code that uses LibCST to negate the condition of if statements on odd lines. This sample is implemented in `dynamicslicing/utils.py` in the `OddIfNegation` class and the `negate_odd_ifs` function. There is a naive solution to this part of Milestone 1, which is to remove the lines directly from the string representation of the code. However, this naive can result in syntax errors in the resulting code, which is why you should implement a syntax tree-based solution.

## 3.2 Milestone 2 (Progress Meeting in Week of Dec 11–15, 2023)

The expectation for this milestone is that each student has implemented dynamic program slicing based on data-flow dependencies.
This analysis must be implemented in `dynamicslicing/slice_dataflow.py` as the `SliceDataflow` class. For the following example,

```
def slice_me():
    x = 1
    y = 2
    x = x + y
    y += 2
    return y # slicing criterion

slice_me()
```

the correct output is

```
def slice_me():
    y = 2
    y += 2
    return y # slicing criterion

slice_me()
```

This output should be written to a file named `sliced.py` in the same directory as the input file. Note that you should not change the signature of the constructor of `SliceDataflow`.

Each input for this milestone consists of an input Python file, which is the program to be sliced, containing a comment, which gives the slicing criterion. Note that all variables used in the line containing the comment are part of the slicing criterion. You can safely assume that, in this milestone, the input code consists only of functions with a single control flow path, i.e., no branches.

You can test your code using `pytest tests --only milestone2`. You can, and are strongly encouraged to, add your own test cases to the `tests/milestone2` directory. Each test case consists of one program file and one expected output file, bundled in a directory, similar to the provided test cases.

## 3.3 Milestone 3 (Progress Meeting in Week of Jan 15–19, 2024)

It is expected that each course participant has implemented the complete slicing algorithm, including both data-flow and control-flow dependencies.
This analysis must be implemented in `dynamicslicing/slice.py` as the `Slice` class. For example, the output of your implementation for

```
def slice_me():
    x = 1
    y = 2
    z = 3
    if x < 4:
        y += 2
    if x > 0:
        z -= 5
    return y # slicing criterion
```

```
slice_me()
```

should be

```
def slice_me():
    x = 1
    y = 2
    if x < 4:
        y += 2
    return y # slicing criterion

slice_me()
```

This output should be written to a file named `sliced.py` in the same directory as the input file. Similar to the second milestone, the input of your analysis is a Python file containing a comment, which indicate the program to be sliced and the slicing criterion, respectively. Note that similar to the previous milestone, you can test your analysis with PyTest and you should not change the signature of the constructor of the analysis class.

# 4 Installation Guidelines

Please refer to DynaPyt's tutorial and documentation for installation instructions. To install your analysis as a package, run the following from the root directory of the project:

```
pip install -r requirements.txt
pip install -e .
```

# 5 Running and Testing Your Code

## 5.1 Implementation & Execution

You should implement your analyses in the appropriate files described above, and must not change any function signature from the provided skeleton code. Your implementation for all milestones should generate a file named `sliced.py` beside the input file with the output required in that milestone. You can use Python versions between 3.6 and 3.10 (inclusive) for your implementation. We suggest using a virtual environment to install the dependencies and run your code.

## 5.2 Testing

For the first milestone, you are supposed to run DynaPyt's instrumentation script and analysis using its command line interface (CLI). For the second and third milestones, you can use `pytest` to run your analysis on the test cases provided in the `tests` directory.

To run the test cases in the `tests` directory, you can use the following command:

```
pytest tests
```

To run the tests for one of the milestones only, run:

```
pytest tests --only tests/milestoneX
```

where `milestoneX` can be `milestone2` or `milestone3`. In addition to the provided test cases, we strongly recommend to test your code with additional test cases. To add test cases, you should create an input file, named `program.py`, and an expected output file, named `expected.py`. Your implementation will be evaluated with test cases that are not provided to you.

For consistency and to ease the task, all input code files follow this convention:

- There is a single function definition, with the function called `slice_me`, which is the function to be sliced.
- The last statement in the file is a call to the `slice_me` function.

The examples given above follow this convention.

# 6 Mentoring and How to Submit

Each student has a mentor, either Aryaz Eghbali (aryaz.eghbali@iste.uni-stuttgart.de) or Beatriz Souza (beatrizbzsouza@gmail.com). Students must meet their mentor at least three times during the semester, on the dates indicated for the milestones. In addition, students may consult their mentor to resolve any questions, to ensure progress in the right direction, and to help you submit a successful project in time. We also recommend to ask general questions in the Ilias forum.

The deadline for submitting the project is at 11:59 PM on February 1, 2024. This deadline is firm. The submission will be via Ilias and should include:

- A scientific report of at most four pages that summarizes the approach taken, the results obtained, and any shortcomings you are aware of.
- A .zip file of the `dynamicslicing` directory (the top directory which includes `src`, `tests`, etc.) with your implementation. Your implementation will be evaluated automatically, by installing it and running PyTest. Please do not change the interface of the analysis classes.

Each person must present the project on the week of February 5–9, 2024, in a short talk, followed by a question and answer session.

The project is individual, i.e., students must work by themselves and not share their solution with anyone else. Any form of collaboration that results in similar or identical solutions being submitted will be punished according to the usual rules for plagiarism.

Grading will be based on the following criteria:

| Criterion | Contribution |
|---|---|
| Progress meetings and final presentation (progress made, clarity, illustration, quality of answers) | 25% |
| Implementation (completeness, documentation) | 25% |
| Results (soundness, reproducibility) | 25% |
| Report (clarity, illustration, discussion and interpretation of the results) | 25% |