**University of Stuttgart**

A Project Report on

**Dynamic Slicing**

Course: Program Analysis (Winter Semester 2023/24)

By

Parth Dineshchandra Panchal

MSc. Computer Science

(Matriculation No: 3697397)

1. **Introduction**

The goal of this project is to implement dynamic backward slicing [1] for Python. Given a Python program as input and a slicing criterion (denoted by the comment # slicing criterion), the final output of the analysis should keep only the lines needed for the slice. The input Python code is instrumented and analyzed using DynaPyt [2] and the AST of the source code is manipulated using LibCST [3]. The code was tested by running several test cases using the PyTest [4] Framework.

1. **Algorithm**

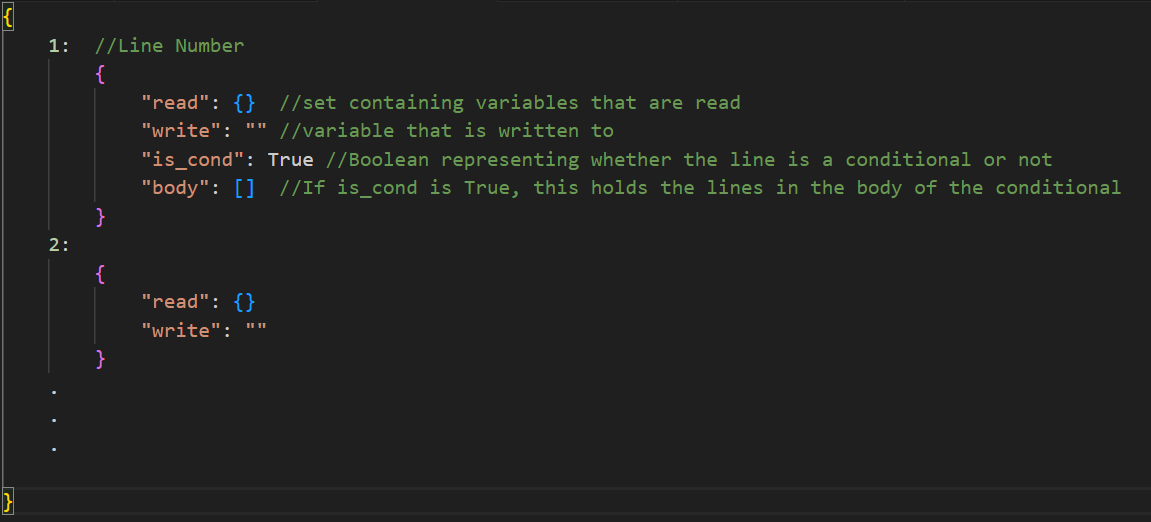
Before going through the high-level algorithm used, here is the structure of the “datastore” which is a dictionary used to keep track of reads, writes, and control structures throughout the analysis.

Fig.1: Structure of datastore

**Input:** program.py (with #slicing criterion)

**Output:** sliced.py

**Start**

1. SET datastore={},

SET lines\_to\_keep=[],

SET target\_vars={}

1. READ program.py
2. GET the line number for #slicing criterion and append it to lines\_to\_keep. Add all variables in this line to target\_vars.

Also, GET lines corresponding to function and class definitions (other than slice\_me()) and append those to lines\_to\_keep.

1. Using DynaPyt hooks:

// For Data Flows:

TRACK reads and writes of variables and add them to datastore.

// For Control Flows:

TRACK control structures and update datastore by adding the respective information to datastore (i.e. SET “is\_cond”=True and append lines in the body of the control structure to “body”)

1. // Analysis Step

For each entry in REVERSED datastore:

// For Data Flow:

5.1 If variable in “write” is in target\_vars:

* + Append line to lines\_to\_keep.
  + Append variables in “read” to target\_vars.

// For Control Flow:

5.2 For each line in the “body” of control structure:

If variable in “write” of body line is in target\_vars:

* + Append body line and control structure line to lines\_to\_keep.
  + Append variables in “read” to target\_vars.

1. REMOVE the lines not in lines\_to\_keep by traversing the AST and removing the corresponding nodes.
2. WRITE the updated program to sliced.py.

**End**

While the general algorithm followed remains the same, the actual implementation has adjustments added to accommodate different edge cases that one might come across. One such example of such adjustment is that the type of AST node at a statement can be of different types, and based on the type, the tree is traversed differently.

1. **Results**

The implementation can handle all the provided test cases. Besides this, to test various edge cases and combinations of inputs, an additional 49 test cases were added. Some of them are listed below:

|  |  |  |
| --- | --- | --- |
| **Milestone** | **Test Case #** | **Description** |
| 2 | test\_11 | To test when slicing criterion has return x + y i.e. Multiple variables in the return statement |
| test\_14-18 | To test more with objects |
| test\_19-22 | To test cases with string concatenation & fstrings |
| test\_23 | To test overwritten values. E.g. x=1; y=2; x=4 |
| test\_29 | To test passing multiple arguments to a function e.g., a.Add(x, y) |
| test\_30-31 | To test other data types like sets and tuples |
| 3 | test\_11 | To test if..elif..else |
| test\_13-15 | To test multiple conditions in if joined with logical operators |
| test\_18 | To test whether the definition of variables in the condition that is evaluated to false is kept or not (for if else case) |
| test\_20-24 | To test nested if |
| test\_25-30 | To test loops |
| test\_31-32 | To test nested for and while |
| test\_33-34 | To test loops with break and continue |

1. **Shortcomings**

Although the implementation functions as expected for the test cases discussed above, there are a few limitations to keep in mind:

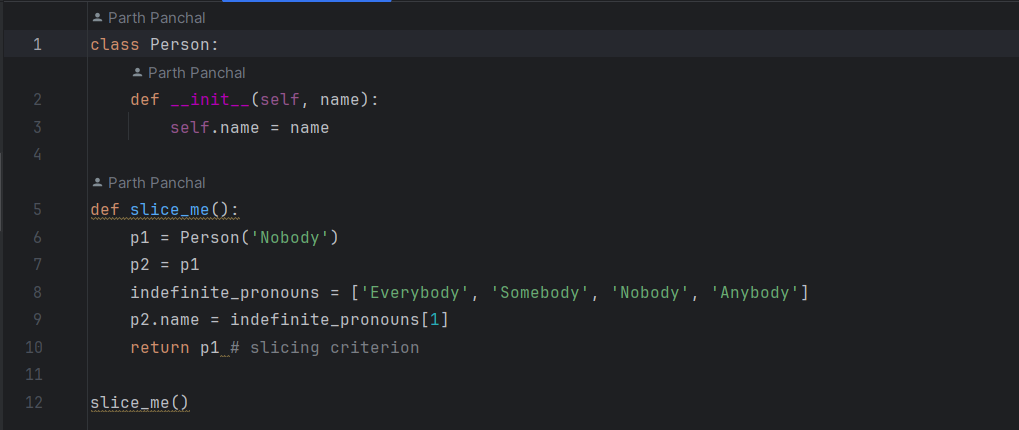
1. **Object Aliases:** Currently, object aliases are handled in a rudimentary fashion, where aliases are tracked by maintaining a dictionary. During the

Fig.2: milestone2/test\_32 program.py

analysis stage, this dictionary is used to include the alias variable in the slice if it meets certain criteria. For example, in milestone2/test\_32, because p2 is an alias of p1, and p2 is being used to modify the value of p1, the expected.py keeps all the lines in slice\_me(). This basic logic can however break quite easily when dealing with complex objects and their aliases. This has not been tested thoroughly.

1. **Match Case:** The analysis is not able to handle Python's match case (introduced in 3.10) due to it being not supported by DynaPyt.
2. **Loop-Else:** The for..else and while..else cases have not been handled yet.
3. **AST structures:** Since modifications to the code are based on ASTs, not all possible types of nodes and all possible combinations may be covered yet. Redundant code may be present in the implementation to cover all possible combinations seen so far from the test cases, which could be simplified. Hence, more testing is needed to cover all language features of Python
4. **References**

[1] Dynamic Program Slicing, Agrawal and Horgan, PLDI 199

[2] DynaPyt: A Dynamic Analysis Framework for Python <https://2022.esec-fse.org/details/fse-2022-research-papers/48/DynaPyt-A-Dynamic-Analysis-Framework-for-Python>

[3] LibCST: A Concrete Syntax Tree (CST) parser <https://github.com/Instagram/LibCST>

[4] PyTest: A Python testing framework <https://github.com/pytest-dev/pytest/>