CSE211: Compiler Design

Homework 2: Optimization and Flow Analysis
Assigned Oct. 25, 2023(by midnight)
Due Nov. 13, 2023(by midnight)

- This assignment has two parts: part 1 implements the local value numbering optimization. Part 2 implements a live variable analysis for a subset of Python code; the analysis will be used to find potential uses of uninitialized variables. Both parts are weighted equally for your grade: 50% each.
- There are likely typos in the homework (writeup and code), especially as I am adapting the homework for the larger class and pair programming! Please let me know if you see any.
- Please ask questions on Piazza; if they are not about your submission specifically, then please ask them so the whole class can participate.
- This homework is designed to be done as a pair. I encourage you to utilize pair programming techniques, that is: both people should be present for any work on the homework. If this is on zoom, then a screen should be shared at all times. Only one computer should be working on the homework at a time. Please take this chance to help each other understand the homework; explain the concepts you are coding and make sure you both understand what is going on completely before moving on.
- If you have any issues with your partner, please let us know ASAP.
- Do not work with anyone apart from your partner on this homework (except for what is explicitly mentioned above).

1 Local value numbering

This part of the assignment implements local value numbering. We will be iterating over a basic block that consists of a series of arithmetic operations, and replacing redundant arithmetic instructions with assignment instructions. Your goal is to replace as many arithmetic instructions as possible.

1.1 Assignment skeleton

Find the assignment skeleton at `homework2_code/part1`. Your implementation is constrained to `skeleton.py`. The code takes in a `cpp` file, see the `test_cases` directory for examples. These test cases have a basic block delimited by comments for example:

```
// Start optimization range
c = h - f;
c = e + h;
a = f + h;
a = f - e;
...
// End optimization range
```

I have implemented code that filters this series of statements out from the program. It is your job to perform local value numbering on this set of expressions. You should assume commutativity of +. Each test file has a comment at the top stating the expected number of arithmetic expressions that you should be able to replace with assignment operators.

1.2 Implementation details

You will need to iterate over each statement in the basic block. You can assume statements are limited to the following form:

var = var op var;

where var is a single lower-case character and op is either plus or minus (+ or -).

The code that your program prints should be able to be compiled and executed. You should create as many extra variables as you need (i.e., for the numbered variables), but you are responsible for declaring the variables.

Please also print out a comment at the end of your output file that states how many instructions were replaced. For each test case, the expected replaced instructions is shown in a comment at the top.

1.3 Testing

You can test your code on the provided test cases. I have written a bash script to help you do this. Simply run:

```
bash test.sh test_cases/test0.cpp
```

This will run some tests for the `test0.cpp` test case. It will compile the original code and execute it. The original code will print out a hash of the variables along with some timing information. It will then run your skeleton, compile the resulting code and execute it. This will also print a hash and some timing information. Your hashes should match (or at least be very very close because floating point addition is not actually commutative). Additionally, the timing information allows you to see if your approach gave any timing improvements.

The test case `test0.cpp` is the simplest test case, and you can use it for debugging. The others are quite long and are more for stress testing your implementation.

1.4 Evaluation

1. Test your implementation. All of the provided test cases should pass (e.g. have a nearly equal hash). Additionally, the number of instructions you replace should be equal to the number given at the top of each test case file.

2. Record the timing difference between the original code and your optimized code.
3. Document your code and make it readable so that we can evaluate it.
4. Provide a README file in your submission that records the timing differences and the number of instructions replaced for each of the provided test cases.

Note that the tests I have provided you are not guaranteed to be the exact tests that I will use for grading.

1.5 Challenge

As a challenge, you can try to implement copy propagation: `https://en.wikipedia.org/wiki/Copy_propagation`

If you do this, you should see considerably more speedup in your optimized programs.

2 Uninitialized variables

In this part of the assignment, you will use flow analysis to find the LiveOut set of a CFG. You will then use this information to detect potentially uninitialized variable accesses in Python code. For background, I suggest you review slides from class and look at section 9.2 in the EAC book (available for free online from the libary, there is a link on the course page).

We will use PyCFG (see: `https://pypi.org/project/pycfg/`), which generates a simple CFG for Python code. The library is quite fragile, but the subset of the language we will use seems to be robust. A key difference in PyCFG from the CFGs we've seen in class is that the PyCFG is limited to single-instruction nodes. This makes large graphs, but the analysis per node easy. The Python subset we will constrain ourselves to is:

- Variables are any sequence of lower-case letters
- Variable-to-variable assignment: e.g. `x = y`
- Input-to-variable assignment: e.g. `x = input()`
- Simple `if` statements, where the condition is a single variable. The if can be followed by an else. e.g.

```
if x:
  y = z
else:
  x = input()
```

- Simple `while` statements, where the condition is a single variable. e.g.

```
while x:
  x = input()
```

The project is to identify variables that are potentially accessed before initialization. For example, the following program may access `x` before it is set, i.e. if the `else` branch is taken:

```
z = input()
if z:
  x = input()
else:
  w = input()
y = x
```

A LiveOut analysis will find `x` is live at the start (and thus, has the potential to be accessed uninitialized).

2.1 Assignment skeleton

Find the assignment skeleton at `homework2_code/part2`.

Please look at the various files in `test_cases` to see examples of the python language subset we will be analyzing. The `solutions.py` file will show for each test case, the set of potentially uninitialized variables you should be finding.

Your assignment is constrained to `skeleton.py`. Read through this code and to understand the structure and what you will be implementing. I have outlined several functions that you can use as a starting point.

If you want to develop locally, I installed the following packages to the docker image:

```
pip install astunparse
apt-get install python-dev graphviz libgraphviz-dev pkg-config
pip install pygraphviz
```

2.2 Technical work

1. Implement functions to create the per-node sets of UEVar and VarKill and the compliment of VarKill. you will need some way how to parse the instructions from `get_node_instruction`. You should be able to get everything you need using regular expressions. Remember, the language is highly constrained as described above. However, your parsing should be robust to different amounts of whitespace inside a single instruction.

2. Implement `compute_LiveOut`. This is the iterative flow analysis algorithm. It should look similar to figure 8.14 in the EAC book (in the most recent edition; for the edition available online, this is the right-hand side of figure 9.2).

3. As we discussed in class, flow algorithms can be optimized depending on the order that nodes are traversed. We will now investigate this.

   - record how many iterations each test case takes to converge using the default order given by pycfg.
   - replace the default node order with a reverse postorder (rpo) traversal through the nodes. Record how many iterations each test case takes to converge.

- replace the default node order with the rpo computed on the reverse CFG (see section 9.2.2 of EAC). Record how many iterations each test case takes to converge.

Write your observations as comments at the end of the file.

If you want to visualize CFGs, you can use the `print_dot.py` file, which takes in a python file as an input, e.g. `python print_dot test_cases/1.py`. It will produce a png file of the CFG: `test_cases/1.py.png`.

2.3 Evaluation

1. Test your implementation. You can run your script standalone with one of the test case files as an argument, i.e. running: `python skeleton.py test_cases/1.py` and it will report the uninitialized values found.
2. You can test your implementation using my testing script: `tester.sh`. I had to use a bash script this time to reinitialize the PyCFG module for each file. It will run the 8 test cases and compare the results to solutions I have computed using a reference. You can see the solutions in `test_cases/solutions.py`.
3. You are expected to pass all the provided tests.
4. There is no tester for the traversal order part of the assignment. Please write your observations as comments at the end of the file. It does not matter which traversal order your submitted code uses.
5. As noted before, PyCFG can be quite fragile. It is not required, but I would be interested in any additional test cases you develop. If you want to include them, simply put them in `test_cases` give them some kind of distinguishing name (e.g. `new_tests_0.py`).

You will be graded on several components:

- **Correctness**: I will run a suite of tests on your implementation and check that they are equivalent to a reference implementation.
- **Comments and clarity**: Please document your code. You don't need tons of comments, but your code should be readable. This includes comments describing your observations about how the traversal order changes the number of iterations for the algorithm.

Please note that the tests I have provided you are not guaranteed to be the exact tests that I will use for grading.

3 Submission

Please zip up the `homework2_code` directory containing all your work and submit this zip file to Canvas under Homework 2.