CSE211: Compiler Design

Homework 2: Optimization and Flow Analysis
Assigned Oct. 18, 2021
Due Nov. 1, 2021

- This assignment has two parts: part 1 implements several variants of 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.
- All required software has been pushed to the class docker. Please do a `docker pull` command as described on the class webpage to get required software.
- Skeletons for this assignment can be found in two places: either in a zip from class webpage at: https://sorensenucsc.github.io/CSE211-fa2021/assignment_data/homework2_code.zip, or in the docker image with base path `assignment_data/homework2_code`. I suggest you copy the skeleton directory to a working directory.
- Please read the instructions in their entirety before asking questions.
- There are likely typos in the homework (writeup and code)! Please let me know if you see any. This is still a new course and I am evolving the homework constantly. Along these lines, if you have ideas for the homework, please let me know!
- I have created a google doc for questions located here: https://docs.google.com/document/d/1KPV7mWqXAFw5AqG-QwZz5LlcFT4o34lEQBc6ntY0Yq4/edit?usp=sharing. Please feel free to ask and answer questions there; I will try to keep it updated with questions from slack.

1 Local value numbering

This part of the assignment requires 4 different implementation of 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, under the constraints given for each implementation variant. I will compare your results to a reference implementation; each variant should be specified enough so that optimal results are deterministic.

1.1 Assignment skeleton

Find the assignment skeleton at `homework2_code/part1`.

Your implementation is constrained to `skeleton.py`. Read through the code to understand the variable, instruction and basic block objects and how to use them. I have implemented the function that numbers variables in a basic block as a reference.

1.2 Implementation variants

You will implement 4 variants of local value numbering, each with different constraints.

I have implemented the function that numbers variables in a basic block. You can look at this function to see how to use the objects and the member functions. There are four functions for you to implement: `replace_redundant` with numbers: 1,2,3,4. I have outlined the structure of the code, but you will need to fill in the rest.

Each function should return the new block, along with an integer indicating how many arithmetic instructions were replaced with assignment instructions. I have provided a rough draft for each implementation. It is your job to finish the implementations.

1. Part 1: implement `replace_redundant_part1`. This is around line 154. I have implemented the basic structure for you, but you will need to do the rest. You can assume the input block has been numbered.

   In this variation, you can not assume any commutative operations. You can assume numbered variables are distinct. That is, `a0` is different from `a1`.

2. Part 2: implement `replace_redundant_part2`. This is around line 194. Part 2 is the same as part 1 except you can use the commutative property of + and *.

3. Part 3: implement `replace_redundant_part3`. This is around line 243. This is the same as part 2, with the new constraint that you CANNOT assume that numbers create fresh variables. That is, `a0` is NOT different from `a1`. If you drop the numbers from the variables, the program must still execute the same as the original.

   This means that your replacement check needs to determine if the variable has been assigned a new value more recently. For example, consider the program:

   ```
   a = b + c
   a = x + y
   z = b + c
   ```

   You cannot replace `z = b + c` because `a` no longer contains `b + c`, it was overwritten. In order to grade this, I am assuming that the oldest value remains in the hashtable. That is, if you replace an arithmetic operation `a = b + c` with an assignment operation `a = e`, the hash table will keep `b + c` mapped to `a`, rather than updating it to `e`.

4. Part 4: implement `replace_redundant_part4`. This is around line 290. This part is the same as Part 3, except you should track a set of possible replacements rather than just one. For example, consider this program:

   ```
   a = b + c
   e = b + c
   a = x + y
   ```

```
z = b + c
```

Like in part 3, you cannot replace the fourth instruction with `a` because `a` has been replaced with `x + y`. However, you could replace it with `e`, as this one has not been overwritten.

1.3 Evaluation

1. Test your implementation. You can run your script standalone, i.e. running: `python skeleton.py` and it will run the test basic blocks at the end of the file. These should be simple and straightforward to debug.

2. You can test your implementation using my testing script `tester.py`. It will run 128 randomly generated basic blocks (in `test_cases.py`) and compare the number of replaced expressions with my values obtained from my reference implementation. It may be of interest to see how many total instructions were replaced with each approach. It should not be surprising that part 2 replaces much more than part 1. Please think about the values for part 3 and 4 as well.

1.4 Submission

Please zip up the directory containing `skeleton.py` (including all testing files) and submit to canvas.

You will be tested on several components:

- **Correctness**: I will run a suite of tests on your implementation and check that they are equivalent to a reference implementation I have. Both the number and of optimized instructions, as well as the returned basic block must match.

- **Comments and clarity**: Please document your code. You don't need tons of comments, but your code should be readable.

Please note that I the tests I have provided you are not guaranteed to be the exact tests that I will use for grading. For example, they only test the number of replaced instructions and not the returned basic block. The tests are provided to get you started.

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 written code to parse the CFG produced by PyCFG and several functions that you will need for the assignment.

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 the functions to create the sets UEVar and VarKill. These are at lines 111 and line 121, respectively. See the implementation of `get_VarDomain` as an example of how to iterate through nodes and get variables.

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). Keep in mind that you will need VarDomain to compute the complement of VarKill. Additionally, I have provided a function that iterates through the successors of a graph node.

3. In section 9.2.2 of EAC, it discusses that many 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.
   - 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 7 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. 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.

4. 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`).

2.4 Submission

Please zip up the directory containing `skeleton.py` (including all testing files) and submit to canvas.

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