CSE211: Compiler Design

Homework 2: Optimization and Flow Analysis
Due Nov. 12, 2020

Oct. 22, 2020

- Read through instructions fully before beginning assignment; the testing scripts described later can help you while developing!
- These assignments are done with Python 3; you'll need to explicitly call `python3` on the server unless you switch it in your environment. The default `python` command defaults to version 2.7.

1 Four Variants of Local Value Numbering

This problem requires 4 different implementation of local value numbering and how to use the technique to remove redundant arithmetic operations. We will be iterating over a basic block that consists of a series of arithmetic operations, and replacing redundant arithmetic instructions with assignment instructions. You will be evaluated on the number of instructions you replace. I will compare your results to a reference implementation; each variant should be specified enough so that results are deterministic.

1.1 Preliminaries

1. Read through the Chapter 8.4 of EAC to refresh yourself on the local value numbering algorithm. You might also read this blog post: https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/global-value-numbering/
2. Find the assignment skeleton at `/home/tsorensen/public/homework2/part1`. Make a copy over to some homework1/part1 directory within your home directory. e.g. run the following:

   ```
   cp -r /home/tsorensen/public/homework2/part1/* ./
   ```

3. Your assignment is constrained to `skeleton.py`. Read through this code to understand the 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 Technical work

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; 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. This will include tracking values through hash tables (or dictionaries), etc.

1. Part 1: implement `replace_redundant_part1`. This is around line 160. 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, i.e. all variales are numbered.

   For part 1, you can not assume any commutative operations. You can assume numbered variables are new variables. That is, a0 is different from a1

   You should create a hash table from the rhs (op1,op,op2) to the lhs. For each instruction, you should check if the rhs is in hash table. If so, the arithmetic instruction can be replaced with a assignment op to the variable in the hashtable.

   It should return the new block, along with an integer indicating how many arithmetic instructions were replaced with assignment instructions

2. Part 2: implement `replace_redundant_part2`. This is around line 200. Part 2 is the same as part 1 except you can use commutativity of + and *. You should find a deterministic order for your operands when you hash them (e.g. based on the numbering)

3. Part 3: implement `replace_redundant_part3`. This is around line 252. This is the same as part 2, with the new constraint that you CANNOT assume that numbers create fresh variables. That is, a_0 is NOT different from a_1. If you drop the numbers from the variables, the program must still be correct.

   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. You should keep an extra hash table to track the most recent version of the variable. 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 `e`, rather than updating it to `a`. If you implement it differently and are failing the checks, please let me know.

4. Part 4: implement `replace_redundant_part4`. This is around line 301. This part is the same as Part 3, except you should track a set of possible replacements rather than just one. Consider this program that is similar to the program in part 3:

   ```
   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.

To do this, your hash table should be updated to return a set of possible values, rather than a single one. That is, at the fourth instruction, your hash table should map `b + c` to a set containing both `a` and `e`. Your analysis should then determine that you cannot use `a` because its been overwritten, but it CAN use `e`.

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 create a directory in your home directory called: `submissions`. Within that directory, create a directory called `homework2_part1`. Move your `skeleton.py` into that directory. Do not make this directory readable by anybody else. I have admin privileges that will allow me to copy from there.

1.5 Extra

If you want to experiment further, try to think of other heuristics you might implement. I can think of at least two others you could use to further reduce the number of arithmetic instructions.

2 Statically Finding Potentially Uninitialized Variables in Python

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.

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 has only single instruction nodes rather than basic block 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 can be accessed uninitialized).

Like in part 1, I have provided a skeleton for you. Please use this skeleton because it is set up in a way that I can automatically test.

2.1 Preliminaries

1. Review the flow analysis algorithm for LiveOut in section 8.6.1 of EAC. You can also review slides from Oct. 22.

2. Find the assignment skeleton at `/home/tsorensen/public/homework2/part2`. Make a copy over to some homework1/part2 directory within your home directory. Make sure to include the subdirectory for pycfg and the test cases. e.g. run the following:

   ```
   cp -r /home/tsorensen/public/homework2/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.

3. Your assignment is constrained to `skeleton.py`. Read through this code and to understand the structure and what you will be implementing. Specifically, I have written code to parse the CFG produced by PyCFG.

2.2 Technical work

1. Implement the functions to create the sets UEVar and VarKill. These are at lines 110 and line 120, 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. 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 page 481). Record how many iterations each test case takes to converge.

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

2.3 Evaluation

1. 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: `test_cases/1.py.png`. If you view this file (e.g. by copying it back to your machine), it will be the CFG for the input file.

2. 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.

3. 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`.

4. There is no tester for iteration count part of the assignment. Good luck!

2.4 Submission

Please create a directory in your home directory called: `submissions`. Within that directory, create a directory called `homework1_part2`. Move your `skeleton.py` into that directory. Do not make this directory readable by anybody else. I have admin privileges that will allow me to copy from there.