CS 243 Homework 2

Winter 2024

Due: January 31, 2024, 11:59 pm

Directions:

- This is a programming assignment. Download the starter code from the course website, complete the task (see "Your Task" below), and submit your code using Gradescope.
- You are expected to work on all programming assignments in pairs, though working alone is acceptable also. If you are working in a pair, be sure to add your partner to your group on Gradescope.
- Get started early. This project will give you the experience of working with a real-world compiler, which may take you some time. If you wait until the last minute, you will not be able to finish.
- You can watch the video posted on Canvas to get started with the assignment.

1 Introduction

Your assignment this week is to implement a basic dataflow framework for the JoeQ system. We will provide the interfaces that your framework must support. You will write the iterative algorithm for any analysis matching these interfaces, and also phrase Reaching Definitions and Faint Variable in terms that an implementation of the solver can understand.

2 Environment

We have set up an environment for this assignment on the Stanford Myth Cluster. To access Myth, ssh from your machine as follows, using your SUNet ID password when prompted:

```
$ ssh <YOUR_SUNET_ID>@myth.stanford.edu
```

Make sure your code compiles and runs on Myth, since your code will be tested and graded on Myth.

2.1 Developing on Your Own Computer (at your own risk)

Due to the strict Java version requirements of this assignment, we highly recommend using Myth as shown above. **Use the following instructions at your own risk.**

If you prefer using your own machine, you may download a JDK that is compatible with Java 5–8. JoeQ runs best on Java 5 (1.5 in `java -version`), but we provide a work-around to run JoeQ on Java 6–8 (aka 1.6–1.8 in `java -version`). As a side effect of the work-around, you may get an error like:

```
Warning: vm vendor [...] is not yet supported, trying default sun13_
```

You can ignore this error. It is sent to standard error, so it will not interfere with diff-ing your output for correctness.

Here are some shortcuts for installing a correct version of JDK:

- Linux/Windows: Download JDK 5.0 from the Oracle website.
- macOS (Intel): `$ brew install openjdk@8`
- macOS (Apple M1/M2):

```
$ brew tap homebrew/cask-versions
$ brew install temurin8
```

3 Starter Code

We have supplied you with some starter code that bundles JoeQ along with files for you to fill in. This directory is organized as follows:

`Makefile`	Makefile for compiling your code and testing
`id.txt`	Write down your SUNet ID(s) in this file
`build/`	build directory (generated by make, not part of the zip file)
`lib/joeq.jar`	packaged JoeQ
`lib/rt.jar`	Java 5 Object.class, for compatibility with later versions
`src/examples`	Examples of using JoeQ
`src/flow`	The dataflow framework and interfaces
`src/test`	Tests for your solution

You only need to modify

1. src/flow/MySolver.java
2. src/flow/ReachingDefs.java and
3. src/flow/Faintness.java

.

4 Getting Started

To set up the environment and compile the source code, do the following:

```
myth:~$ unzip hw2.zip
myth:~$ cd hw2
myth:~/hw2$ source hw2.env    # See notes below if not using Myth.
myth:~/hw2$ java -version     # Make sure this prints a supported
                                Java version (1.5-1.8).
myth:~/hw2$ make
```

Notes:

1. If you are using your own machine, you need to modify the `hw2.env` file to use the correct file path for your Java installation. Examples:

   - macOS (Intel): Assuming you ran `$ brew install openjdk@8` earlier:
     `export JAVA_HOME=/usr/local/opt/openjdk@8/libexec/openjdk.jdk/Contents/Home`
   - macOS (Apple M1/M2): Assuming you ran `$ brew install temurin8` earlier:
     `export JAVA_HOME=/Library/Java/JavaVirtualMachines/temurin-8.jdk/Contents/Home`

2. The third line (source hw2.env) sets environment variables `JAVA_HOME` and `PATH` in your current shell. Therefore, if you exit the current shell and reopen one, you would have to run `source hw2.env` again.

   To avoid manually doing this every time you log in to a shell, you can modify your `/.bashrc` (or `/.bash_profile`, `/.zshrc`, etc., depending on your machine configuration) in order to do this automatically during login. You may wish to undo the changes after this course concludes.

Visualizing Quads. To "visualize" how JoeQ breaks down a Java program into Quads, you can run the PrintQuads example with the following command:

```
myth:~/hw2$ ./run.sh examples.PrintQuads examples.ExprTest
```

In the example above, `examples.PrintQuads` is the Java program executed, and `examples.ExprTest` is the argument that `examples.PrintQuads` takes. You may take a look at the main function of `examples/PrintQuads.java` for better understanding.

Running dataflow solver. Our main task is to implement a dataflow solver in `flow/MySolver.java`. To run the solver with constant propagation in `flow/ConstantProp.java` over a test program called `test.Test`, invoke the `run.sh` script on these four classes:

```
myth:~/hw2$ ./run.sh flow.Flow flow.MySolver flow.ConstantProp test.Test1
```

We also wrote a shortcut for you (here `<test_name>` corresponds to "Test1" in the command above, and `<flow_name>` corresponds to "ConstantProp" in the command above):

```
myth:~/hw2$ FLOW=<flow_name> make <test_name>.check
```

Until you write `flow.MySolver`, neither of these commands will compute anything useful. (See section 7 for additional information on testing.)

5 Your Task

1. Fill in the `MySolver` class so it implements the interface `Flow.Solver`. Use it to run the constant propagation and liveness algorithms we provide.

   Note: After correctly completing this part, you should pass `FLOW=ConstantProp` make test and `FLOW=Liveness` make test.

2. Fill in `ReachingDefs` so that it executes reaching definitions analysis.

   Note: After correctly completing this and the previous part, you should pass `FLOW=ReachingDefs` make test.

3. Fill in `Faintness` so it executes faint variable analysis defined in the following section. Test your faintness implementation by adding test methods to `TestFaintness.java`. Make sure you test your implementation thoroughly. Note that the supplied `Test Faintness.Faintness.out` is only a very basic test.

6 Faint Variable Analysis

The lecture introduced the concept of live variables, which can be used to eliminate dead code in a program. Consider the following code:

```
int foo() {
  x = 1;
  y = x + 2;
  z = x + y;
  return y;
}
```

In this example only `x` and `y` are live; therefore it is safe to eliminate instruction 3 which sets a dead variable `z`. However, there are many cases where live-variable analysis fails to completely eliminate dead code. Consider the following programs:

```
    Case 1

    void foo() {
1.      x = 1;
2.      y = x + 2;
3.      z = x + y;
4.      return;
    }
```

```
    Case 2
    bool bar();
    int foo() {
       while(bar()) {
1.          x = x + 1;
2.          y = y + 1;
        }
3.      return y;
      }
```

In Case 1, `x` and `y` in instruction 1 and 2 are live because they are used in instruction 3. `z` is not live and can be eliminated. If you apply the liveness analysis again after removing instruction 3, you will find that `y` is actually dead and can be eliminated. In the third

iteration, you will find that `x` is also dead. A better dataflow analysis should be able to eliminate the three instructions in one pass. In Case 2, `x` and `y` are both live because they are used in statements that define them. However, since the final value of `x` is never used, instruction 1 could in fact be eliminated. These examples motivate the following definition of a "faint" variable. If we base the dead- code elimination algorithm on faint variables, we would be able to eliminate all dead code in one step.

Definition 1. A variable is *faint* if it is not live (dead) **or** it is only used in computing faint variables. In this assignment, we assume that the faintness of a left-hand-side variable will only propagate to the right-hand-side variable if the operator is either `Operator.Move` or `Operator.Binary` (referring to the classes in JoeQ). For example:

```
ADD_I  T12 int, R4 int, R5 int       # equivalent to T12 = R4 + R5
```

`ADD_I` is a binary operator (it inherits from `Operator.Binary`). Therefore, if `T12` is faint (and this is the only instruction that uses `R4` and `R5`), then `R4` and `R5` are also faint. Function calls may look like:

```
INVOKEVIRTUAL_I%  T8 int, M.Class1.Method(I)I, (R0 Class1, R1 int)
                                     # equivalent to T8 = R0.Method(R1)
```

`INVOKEVIRTUAL_I` is neither binary nor a move; therefore, faintness is not propagated. Function calls, writes to fields, and returns are all examples of things that do not propagate faintness, and can cause a variable to not be faint.

7 Testing Your Code

The `test` package contains a `Test1` and a `Test2` class that you can use to test your implementation. The `src/test/` directory contains `Test1.ConstantProp.out`, `Test1.Liveness.out`, and `Test1.ReachingDefs.out` which are output files for running the `ConstantProp`, `Liveness`, and `ReachingDefs` analyses over the `Test1` class, and similarly for `Test2`.

`diff` the output of running your `MySolver` with these out files to test your implementation. We also provide a quick testing command in `Makefile`:

```
FLOW=<flow_name> make test              # run all test cases specified
                                        # by TESTS in Makefile
FLOW=<flow_name> make <test_name>.test  # run one test case
```

where `<flow_name>` is one of `ConstantProp`, `Liveness`, `ReachingDefs`. If you make any modifications to the tests, this will no longer work because it is comparing the output of your solver to the reference output files we provide.

8 Submission

To submit your assignment, follow the following steps:

1. Write your (and your partner's) SUNet ID in `id.txt`, separated by newline.

2. Run `make submission` in the homework directory.
3. Use `scp` (or tools such as FileZilla) to download `submission.zip` from the server.
4. Upload this file to the Gradescope HW2 assignment.
5. If you discover a bug after submitting (and before the due date), simply run steps 1–3 again. Gradescope will take the latest version.

9 Hints

- Look at the `ConstantProp` and `Liveness` classes. You will find techniques that will be useful to you when you formulate your own analyses.
- Read the `JoeQ documentation` carefully, and make use of the `full Javadoc`.
- You are encouraged to refer to the pseudocode of the DFA iterative algorithm in the lecture slides. Be careful with handling the initialization and loop exit conditions.
- Study the `QuadIterator` interface carefully. It does a great deal of the work for you. But please take note: if you construct a backward `QuadIterator`, you must iterate over it with hasPrevious() and previous().
- `DataflowObjects` are mutable; when in doubt, copy them. When writing reaching definitions, be sure to implement an equals() method, or your code will loop forever.

  Note: the grading code has a strict timeout of 4 seconds, to detect code that loops.
- The iterator returned by `Quad.successors()` (resp. `Quad.predecessors()`) generates null to indicate that the current object has the exit node (resp. entry node) as a successor (resp. predecessor).