CS243 Lab 3

Spring 2022

Due: April 26, 2022 at 11:59PM

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 Canvas.
- You may work alone or in pairs. If you are working in a pair, be sure to add your partner's SUNetID to your collaborators.txt file.

1 Introduction

Your assignment this week is to implement an optimization pass in JoeQ to remove redundant `NULL_CHECK`s. When generating quads out of Java bytecodes, safety checks are explicitly inserted into the control flow graph. In particular, every register is `NULL_CHECK`-ed before it is dereferenced. These safety checks are necessary to ensure memory safety, but result in substantial runtime overhead. In this assignment, you will design sound optimizations to eliminate redundant checks and to optimize the program in general. It's important for your code to be compatible with Java 1.5. And we will be testing your code on myth machine. So, **make sure your code compiles and runs on myth using its Java 1.5** (the environment is setup using `setup.sh`).

The starter zip file can be found in the course website.

2 Starter Code

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

- `lab3/classes/` : build directory (will be overwritten, not part of zip).
- `lab3/bin/` : contains `parun`, the project runner.
- `lab3/lib/joeq.jar` : packaged JoeQ.
- `lab3/Makefile` : Makefile for compiling your code.

- `lab3/src/examples` : Examples of using JoeQ.
- `lab3/src/flow` : The dataflow framework and interfaces.
- `lab3/src/submit` : The code you will submit (Only make modifications here!!!)
- `lab3/src/test` : Tests for your solution.

3 Getting Started

After copying the lab3.zip to myth, do the following:

```
myth:~$ unzip lab3.zip
myth:~$ cd lab3
myth:~/lab3$ source setup.sh
myth:~/lab3$ make
```

4 You Task

1. Implement the `FindRedundantNullChecks` class in `lab3/src/submit` that finds all redundant `NULL_CHECK`s.
2. Implement the Optimize class in `lab3/src/submit` that transforms the input control flow graph by deleting all redundant `NULL_CHECK`s.
3. Extra Credit: Perform any other sound optimization to the input class.

5 Redundant `NULL_CHECK`

A `NULL_CHECK` on register x is redundant at point p if x successfully passed a `NULL_CHECK` along all possible paths to p. For example, in the following code, your analysis must find quad 5 to be redundant but does not need to find any other quad to be redundant.

```
1.  MOVE T1 String, T0 String
2.  NULL_CHECK T-1 , T1 String
3.  MOVE T2 String, T1 String
4.  NULL_CHECK T-1 , T0 String
5.  NULL_CHECK T-1 , T1 String
6.  NULL_CHECK T-1 , T2 String
```

In other words, your analysis needs to find that a `NULL_CHECK` is redundant on a register only if that particular register was `NULL_CHECK`'ed along all possible paths to that `NULL_CHECK`. The analysis does not have to reason about copies of values to or from other previously or subsequently `NULL_CHECK`'ed registers.

The `submit.FindRedundantNullChecks.main(String[])` method takes an array of names of classes that should be analyzed for redundant `NULL_CHECK`s.

Fill in the submit.FindRedundantNullChecks.main(String[]) method so that it prints exactly one line for each method that contains the method name and a subset of the sorted quad ids of redundant NULL_CHECKs.

For example:

```
myth:~/optimize$ bin/parun submit.FindRedundantNullChecks test.SomeTest
main 4 17 19
sample 5
<init>
```

means that NULL_CHECKs with quad ids 4, 17, and 19 are redundant in main, quad ids 5 are redundant in sample, and no quads are redundant in <init>. The test package contains two test classes named test.NullTest and test.SkipList. The outputs that should be generated by running submit.FindRedundantNullChecks.main(String[]) over these two classes, are src/test/NullTest.basic.out and src/test/SkipList.basic.out.

6 Removing Redundant NULL_CHECKs

After finding all redundant NULL _CHECKs, perform an optimization pass that removes all redundant NULL_CHECKs in the test.SkipList and test.QuickSort programs. The submit.OptimizeHarness.main(String[]) method takes a list of names of classes that should be optimized, a run class that contains a static main(String[]) method, and a list of run parameters to be passed to the main method.

For example:

```
myth:~/lab3$ bin/parun submit.OptimizeHarness --optimize test.SkipList
--run-main test.SkipList --run-param 20
14 6 21 ...  28 14 17
Result of interpretation:  Returned:  null (null checks:  24547 quad count:  106185)
```

applies your optimizations to test.SkipList, and then interprets test.SkipList with parameter 20. The interpreter prints out the number of quads executed.

The following is an example output of test.QuickSort:

```
myth:~/lab3$ bin/parun submit.OptimizeHarness --optimize test.QuickSort
--run-main test.QuickSort --run-param 200
10 18 20 ...  2838 2844 2878
Result of interpretation:  Returned:  null (null checks:  32017 quad count:  136210)
```

The submit.OptimizeHarness.main(String[]) method invokes the submit.Optimize.optimize(List<String>, boolean) method which should load the classes to be optimized and apply the control flow graph transformations. The transformed control flow graphs should automatically be stored by joeq.Compiler.Quad.CodeCache.

Read joeq.Compiler.Quad.CodeCache and joeq.Main.Helper carefully to understand how control flow graphs are cached. The submit.OptimizeHarness.main(String[]) method then interprets the run class with respect to the list of run parameters using the CodeCached control flow graphs.

Fill in the `submit.Optimize.optimize(List<String>, boolean)` method so that it applies your optimizations to the classes named by the list of String parameters. To remove quads from the program, use `QuadIterator.remove()` which will remove the most recently returned quad.

7 Extra Credit

Perform any other sound optimization that speeds up the `test.SkipList` program (the skip list implementation is from here).

The extra credit points awarded will range from 0 to 100. The number of points will depend on the number of quads executed by the optimized program, and will be applied after all grades are curved. If you work in a group of two, the same extra credit score is assigned to both members. All optimizations must be sound! For example, if you remove even one necessary null or bounds check, or falsely copy a constant, you will receive no extra credit. Your are free to discuss on Piazza how many quads your optimized code achieves.

Modify `Optimize(List<String>, boolean)` to perform extra optimizations when the second argument is false. Take a look at the `QuadIterator` documentation to learn how to add and remove quads. To change the values of `Operands`, the `Operator` class contains static methods to set the appropriate argument. For example, `Move` has methods `setDest` and `setSrc`. To modify the `ControlFlowGraph`, use `ControlFlowGraph.createBasicBlock` to construct a `BasicBlock` and use the `add` methods in `BasicBlock` to modify the list of quads. Please refer to the JoeQ Javadocs for more details.

Test your extra credit using the following command:

```
myth:~/lab3$ bin/parun submit.OptimizeHarness --extra-credit --optimize
test.SkipList --run-main test.SkipList --run-param 20
14 6 21 ...  28 14 17
Result of interpretation:  Returned:  null (null checks:  24547 quad count:  106185)
```

Describe the design of your extra credit optimizations in the `design.txt` file in the `src/submit` directory.

8 Submission

To submit your assignment, follow the following steps:

1. Type `make submission` in the lab directory.
2. Use `scp` to download `submission.zip` from the server.
3. Upload this file to Canvas.
4. If you discover a bug after submitting (and before the due date, you have late days though), simply run the submit again.

9 Hints

- Again, get started early. First, think about what are the biggest optimization opportunities. No matter how sophisticated your optimization is, if the maximum speedup from that is 1%, spending time on this is probably not worth it (Remember Amdahl's law). To do that, it may be a good idea to look at the code of SkipList (both Java source code and Quad representation generated by JoeQ).

- Compared to lab2 and the first part of lab3, you need to transform your code instead of just doing some analyses. Transforming code (especially, if you want to modify control flows) is much trickier and may involve more JoeQ Javadoc reading.

- When eliminating redundant `NULL_CHECK`s, note that sometimes JoeQ performs constant propagation which may lead to some `NULL_CHECK`s of constant values. If this occurs there will be no used registers for the `NULL_CHECK` quad. When determining whether to eliminate a `NULL_CHECK,` make sure that you handle this case properly as it has caused bugs in prior years (i.e. unsafe elimination of `NULL_CHECK AConst: null`).

- **(Extra Credit)** If you choose to perform dead code elimination, note that some operators have side effects (e.g. `NULL_CHECK`) which cause dependencies that are not visible through the defined and used registers of a quad, and thus **cannot** safely be removed by the results of a normal live variable or faintness analysis. You can check whether a quad has side effects by calling `quad.getOperator().hasSideEffects()`.