CS 243 Homework 4

Winter 2024

Due: February 21, 2024 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 Gradescope.
- You have **two late days** on assignments for the entire quarter. See course website for details.
- 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.
- Because of the upcoming midterm, this homework is due in two weeks instead of the usual one week. You are not expected to complete the homework ahead of the midterm.

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.

Unlike Homework 2 which is compatible with Java 7/8, you will have to develop on a machine with Java 5. The starter code zip file can be found here.[1] We will be testing your code on the Myth cluster. **Make sure your code compiles and runs on Myth using Java 5**.

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:

[1] `https://cs243.stanford.edu/hws/hw4.zip`

`build/`	build directory (generated by `make`, not part of the zip file)
`run.sh`	used to run your program (Same as HW2)
`hw4.env`	used to set the environment (Same as HW2)
`Makefile`	Makefile for compiling your code
`lib/joeq.jar`	packaged JoeQ
`src/examples`	Examples of using JoeQ (same as HW2)
`src/flow`	The dataflow framework
`src/submit`	The code you will submit (only make modifications here!)
`src/test`	Tests for your solutions

3 Getting Started

Download the starter code zip file from here and the reference implementation of `MySolver.java` from Canvas. After copying the `hw4.zip` and `MySolver.java` to Myth, do the following:

```
myth:~$ unzip hw4.zip
myth:~$ cp MySolver.java hw4/src/flow/MySolver.java
myth:~$ cd hw4
myth:~/hw4$ source hw4.env
myth:~/hw4$ java -version      # Make sure this prints version 1.5.0
myth:~/hw4$ make
```

4 Your Task

1. Implement the `FindRedundantNullChecks` class in `src/submit/` that finds all redundant `NULL_CHECK`s.
2. Implement the `Optimize` class in `src/submit/` that transforms the input control flow graph by deleting all redundant `NULL_CHECK`s.
3. Extra Credit: See the "Extra Credit" section below.

5 Redundant `NULL_CHECK`s

`NULL_CHECK` is a JoeQ operator that checks if the source register is `null`, and throws an exception if so. As an example, consider the following Java program and corresponding JoeQ quads:

```
int access(int[] arr, int i) {
    return arr[i];
}
1   NULL_CHECK              T-1 <g>,    R1 int[]
2   BOUNDS_CHECK            R1 int[],   R2 int, T-1 <g>
3   ALOAD_I                 T3 int,     R1 int[],       R2 int, T-1 <g>
4   RETURN_I                T3 int
```

In this function, register R1 corresponds to the parameter arr. The array access arr[i] will first emit a NULL_CHECK to make sure that arr is not null (quad 1), before indexing by i (quad 3). NULL_CHECK also has a destination register – in this case the T-1 pseudoregister. However, for the purpose of this homework, you may ignore the destination register.[2]

However, many NULL_CHECKs can be redundant. Consider instead the following function:

```
int add(int[] arr, int i, int j) {
    return arr[i] + arr[j];
}
1   NULL_CHECK          T-1 <g>,    R1 int[]
2   BOUNDS_CHECK        R1 int[],   R2 int, T-1 <g>
3   ALOAD_I             T4 int,     R1 int[],       R2 int, T-1 <g>
4   NULL_CHECK          T-1 <g>,    R1 int[]
5   BOUNDS_CHECK        R1 int[],   R3 int, T-1 <g>
6   ALOAD_I             T5 int,     R1 int[],       R3 int, T-1 <g>
7   ADD_I               T4 int,     T4 int, T5 int
8   RETURN_I            T4 int
```

By the time execution reaches the second NULL_CHECK (quad 4), it is guaranteed that R1 (arr) is not null, since otherwise quad 1 would have thrown an exception. The second NULL_CHECK is thus redundant. We included this example in src/examples/, you can run the following command:

```
myth:~/hw4$ ./run.sh examples.PrintQuads examples.NullCheckExample
```

The task. In this section, you will write a program that identifies all redundant NULL_CHECKs. Formally, we consider a NULL_CHECK on register x **redundant** at point p, if x successfully passed a NULL_CHECK along all possible paths to p. For example, in the add() example above, your analysis must find quad 4 to be redundant. However, the analysis does not have to reason about copies of values to or from other previously or subsequently NULL_CHECKed registers. So with the following program:

```
1   NULL_CHECK          T-1 <g>,    T1 String
2   MOVE                T2 String,  T1 String    // move T1 into T2
3   NULL_CHECK          T-1 <g>,    T1 String
4   NULL_CHECK          T-1 <g>,    T2 String
```

your analysis should only report quad 3 as redundant, *not* quad 4.

Implement your analysis in the submit.FindRedundantNullChecks class. The main() method of this class takes an array of names of classes that should be analyzed, and should print 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:

[2]For additional information about T-1, you may check out Example 2 under the Program Representation section in JoeQ guide.

```
myth:~/hw4$ ./run.sh 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` method, quad ID 5 is redundant in `sample` method, and no quads are redundant in `<init>`.

Testing. The `test` package contains three test classes named `test.NullTest`, `test.SkipList`, and `test.QuickSort`. The outputs that should be generated by running `submit.FindRedundantNullChecks` over the first two of these classes are in `src/test/NullTest.basic.out` and `src/test/SkipList.basic.out`.

We provide a quick command to test if your `submit.FindRedundantNullChecks` is correct:

```
myth:~/hw4$ make check
Running FindRedundantNullChecks on NullTest!
Output is correct!
Running FindRedundantNullChecks on SkipList!
Output is correct!
```

6 Removing Redundant NULL_CHECKs

After finding all redundant NULL_CHECKs, write code to perform an optimization pass that removes all redundant NULL_CHECKs. Do so by filling in the `optimize()` method in the `submit.Optimize` class.

Much like the last section, the `optimize()` method of `submit.Optimize` takes a list of class names to analyze and optimize. You may disregard the `nullCheckOnly` boolean parameter for now; it is only relevant for the next section.

Strategies for getting started. We have already provided code that iterates through all the classes to optimize, and loads each of them into JoeQ. You should first reuse the code you wrote in the last section to identify redundant quads within each method. After that, you can create a `QuadIterator` object that would go through each quad in the CFG, and call `QuadIterator.remove()` to remove a quad from the program.

To create a `QuadIterator` object, you need access to the `ControlFlowGraph` corresponding to a method; that object is available to you in the `preprocess()` and `postprocess()` methods of a `Flow.Analysis`.

Testing. To test your code, you can use the provided `run.OptimizeHarness` class. `OptimizeHarness` takes a list of classes to optimize as well as one class to run, which could be one of the two we provided (`test.SkipList` and `test.QuickSort`) or your own program. It also requires the parameter to pass into the test program. Example invocations below:

```
# Run test.SkipList.main with argument [20] WITHOUT optimizations
myth:~/hw4$ ./run.sh run.OptimizeHarness --run-main test.SkipList \
```

```
            --run-param 20
# Run your optimizations on SkipList; then run test.SkipList.main with
# argument [20]
myth:~/hw4$ ./run.sh run.OptimizeHarness --run-main test.SkipList \
            --optimize test.SkipList --run-param 20
```

`OptimizeHarness` prints out the number of quads actually executed, so you can make sure that your optimizations are actually working. You should expect to see fewer quads executed with the optimization than without. For reference, here are example outputs of running the TAs' solution on `test.SkipList` and `test.QuickSort`:

```
myth:~/hw4$ ./run.sh run.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)

myth:~/hw4$ ./run.sh run.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)
```

7 Extra Credit

As an extra credit exercise, perform any other sound optimizations that speed up the `test.SkipList` program (the skip list implementation is from here[3]).

The extra credit points awarded will range from 0 to 100, depending 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. You are free to discuss on Ed how many quads your optimized code achieves.

To determine the performance-based score, we will use only the `test.SkipList` program. Thus, we encourage you to look closely at that particular program, in order to identify optimization opportunities that would bring the most benefit. On the other hand, any optimization you create must be **sound**: applying the same transformations to any other program should not break those programs. For example, if you remove even one necessary null or bounds check, or falsely copy a constant, you will receive no extra credit. Also, you must describe the design of your extra credit optimizations in the `design.txt` file in the `src/submit` directory.

Modify the `optimize()` function within the `Optimize` class to perform the extra optimizations, but only if the second argument `nullCheckOnly` is false. Take a look at the `QuadIterator` documentation to learn how to add and remove quads. To change the values of `Operand`s, the `Operator` class contains static methods to set the appropriate

[3] https://www.mathcs.duq.edu/drozdek/DSinJava/SkipList.java

argument. For example, `Operator.Move` has methods `setDest()` and `setSrc()`. To modify the `ControlFlowGraph`, use `ControlFlowGraph.createBasicBlock()` to construct a `BasicBlock` and use the `add`, `append`, `remove`, and `replace` methods in `BasicBlock` to modify the list of quads. Please refer to the JoeQ Javadoc[4] for more details.

Test your extra credit implementation using the `--extra-credit` flag:

```
myth:~/hw4$ ./run.sh run.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)
```

8 Submission

To submit your assignment, follow the following steps:

1. Run `make submission` in the homework directory.
2. Use `scp` (or tools such as FileZilla) to download `submission.zip` from the server.
3. Upload this file to the Gradescope HW4 assignment.
4. 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

- Again, get started early.
- Compared to Homework 2 and the first part of Homework 4, you need to transform your code instead of just doing some analyses. Transforming code (especially, if you want to modify control flows) could be tricky and may involve some more JoeQ Javadoc reading.

[4] https://cs243.stanford.edu/joeq/javadoc/