CS6004-A4

Hardik Rajpal (200050048) and Sanchit Jindal (200020120)

April 2024

1 Introduction

We have implemented constant propagation. Our implementation also propagates constants across "pure functions."

1.1 Constant Propagation

We have implemented this analysis as a forward data flow analysis, extending the `ForwardFlowAnalysis` class from Soot to map each local in a function to a data flow value from $\{\perp, \top\} \cup \mathbb{R}$.

1.2 Pure Methods

Pure methods methods that:

- Have a return value that depends only on their arguments.
- Don't read or write to non-local variables (static or field variables).
- Don't call any impure methods.

We use reflection to execute calls to such methods with constant arguments and propagate the result further in the program.

1.3 Transformation

We process each unit and replace any uses of locals by their constant data flow value as discovered by the analysis. We also delete units that assign to locals whose values are discovered to be constant. This also includes deletion of pure method calls if the arguments themselves are constant.

2 Results on Testcases

The numerical outputs of the test cases is given in the file `a4/output.txt`. The analysis works by counting the number of times while the interpreter runs, operations such as add, subtract, multiply, divide and the number of function calls are made. We don't make distinction between the different type of arithmetic operations depending on datatype, or the different ways to call a function

- **T1:** The number of function calls reduce by one as the one function call `T1.foo(b,c)` is a pure call and can be substituted, The number of arithmetic calls also reduce due to the constant propagation
- **T2:** Same as T1 but using a double data type instead of a int datatype,
- **T3:** This test case runs a factorial function and by optimizing we can remove 7 function calls, This is because the argument to the function is a constant and as the function is pure its value can be directly substituted instead of recursively calling the function
- **T4:** This is again a factorial function but the runtime alternates between two identical functions named `factorial1` `factorial2`, so its runtime is similar to the third testcase, and the number of function calls are reduced by same amount
- **T5:** In this testcase we add a print statement inside the factorial function due to which the function no longer remains pure. So no optimization is done
- **T6:** In this example a member function of another class is called so the algorithm does not make any optimizations

3 Setup Instructions

1. Clone the repository from: `https://github.com/sanchit1053/javaOptimization_CS6004/`
2. Switch to the `master` branch.
3. Compile and run the java program in `a4`:

   (a) `cd a4`

   (b) `javac -cp '.:../soot.jar' *.java` (Replace : (Unix) by ; on Windows).

   (c) `java -cp '.;../soot.jar' PA4`

   This should produce `.class` for the `testcases` in the `a4/sootOutput` folder.
4. To run the analysis on a file
   - Build openj9 using the instructions given at `https://github.com/eclipse-openj9/openj9/blob/master/doc/build-instructions/Build_Instructions_V8.md`
   - The assignment is done on the the `0.26` version branch

- You also need to make patches as was discussed on piazza
- `openj9-java -Xint <file name> > output.txt`. openj9-java is the location where the java binary will be built
- `./findStatistics.sh <function_name> output.txt` to parse out the statistics corresponding to the function name