CS 6620
assigned: Sept. 18, 2019
12 points **Homework 2** due: ≤23:59 EST Oct. 2, 2019

This assignment can be completed individually or in a pair. All assignments in the class can be completed this way, but you cannot repeat any pairing during the course. Please clearly indicate the names of the people submitting the solution on the two attached files.

Ths instructor will answer questions on this assignment up to 23:59 EST on Sept. 30, 2019 – 48 hours before the due date.

Part 1 (5 points)

Complete the TIP data flow analysis for live variables. This involves completing the definition of the `LiveVarsAnalysis` in `.../src/analysis/LiveVarsAnalysis.scala` and then providing convincing evidence that your analysis implementation is correct.

1. Implement the undefined portions of the analysis implementation, marked as `<--- Complete here`. Note that you do not have to implement the `case _ => ???` case for assignment – this handles the situation where you are assigning through a pointer, but this simple analysis doesn't support that.

2. You will also need to implement the `<--- Complete here` lines for class `PowersetLattice` in `.../src/lattices/GenericLattices.scala`.

3. Provide convincing evidence that your analysis implementation is correct. The typical way to do this is to define a comprehensive test suite on which you have confirmed the correct behavior of the analysis. The test suite should be comprised of the files from `examples/*.tip`, but you may supplement those files if you choose.

Part 2 (5 points)

TIP does not include a reaching definitions analysis. You are to implement one. Here you are starting from scratch, but you can leverage the fact that data flow analyses are variations on a pattern and there are other analyses defined for TIP, e.g., `AvailableExpAnalysis.scala`, and you just implemented `LiveVarsAnalysis`. These analyses will define much of the boilerplate for your implementation, but you will have to adapt that code to compute reaching definitions.

1. Implement `ReachingDefAnalysis` in the file `.../src/analysis/ReachingDefAnalysis.scala`

2. Implement the undefined behavior marked with `<--- Complete here` in `.../src/analysis/FlowSensitiveAnalysis.scala` to create the solvers.

3. Provide convincing evidence that your analysis implementation is correct. The typical way to do this is to define a comprehensive test suite on which you have confirmed the

correct behavior of the analysis. The test suite should be comprised of the files from `examples/*.tip`, but you may supplement those files if you choose.

Part 3 (2 points)

Pick one of the two analyses above and identify an example that reveals the difference between the simple fixpoint solver and the worklist solver. Explain the differences and describe how the features of the example reveal those differences.

Submission

Upload to the course collab site the following three files: `LiveVarsAnalysis.scala`, `ReachingDefAnalysis.scala`, and a PDF file describing how you designed the reaching definition implementation, your analysis of the different solver algorithms, and how you validated both of the analyses. If you included test cases of your own design, please include those as well.

Hints

You will want to become familiar with the following elements of the TIP analysis implementation:

- `.../src/cfg/CfgNode.scala` defines the types of CFG nodes. Each node contains a `data` field which holds a reference to the associate AST node.
- `.../src/cfg/CfgOps.scala` defines a number of convenience methods that access information from CFG node `data` fields, for example.
- `.../src/ast/AstOps.scala` defines a number of convenience methods that access information from AST nodes. Not suprisingly many of these are implemented using `DepthFirstAstVisitor`. If you didn't take the chance to understand and appreciate the visitor pattern in HW1, you should do so now. This is a very valuable concept when manipulating and analyzing programs.
- Note that much of the boilerplate to allow you to run the analyses is already defined, e.g., the command line option is `-reaching` and the parsing and invocation of the analysis is already setup.
- run with the `-verbose` option turned on to see intermediate calculations of your analysis runs. You don't need to add any debugging statements since the infrastructure already provides what you need in the generic logging code.