Jhonny Gonzalez

CS 4390/5390 Fall 2025

Advanced Compilers

Shirley Moore, Instructor

25 points

## **Data Flow Analyses using the Worklist Algorithm**

For this assignment, you will answer a couple of questions about dataflow analysis and then implement two data flow analyses using the generic worklist algorithm. Please use the existing implementation of the worklist algorithm in examples/df.py from the Bril repository at <https://github.com/sampsyo/bril/> for tasks 3 and 4 below.

1. Fill in the table below with the necessary information for each type of analysis.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Domain | Direction | Init | Merge | Transfer |
| Reaching Definitions | Sets of Defs | Forward | (Empty Set) | Union | f(inb) = genb (inb – killb) |
| Live Variables | Sets of Vars | Backward | (Empty Set) | Union | f(outb) = useb(outbkillb) |
| Constant Propagation | Valuation or T | Forward | (variables unknown) | Meet ( ) | if all reaching defs assign same const to v, else if conflict, else inb |
| Available Expressions | Sets of Expressions | Forward | (Empty Set) | Intersect |  |

1. Write a convincing argument that the worklist algorithm is guaranteed to converge to a solution, given a certain condition. Be sure to state that condition. You do not need to use lattice theory -- you may if you wish but you can also just give a convincing logical argument.

The worklist algorithm is guaranteed to converge to a solution because it uses the combination of monotonicity and finite lattice. For convergence to be reached, we should see that the transfer function for every node in the CFG is monotonic. A monotonic function is described as one that if its input does not decrease, or it increases within the lattice’s ordering, then its output will also not decrease. With this, we are certain that as we process nodes and update information, the values computed can only move in one direction through the lattice. This will either be up or down, which prevents cycles where values oscillate endlessly. What we want to see for convergence begins with the finite height. We want to see that the dataflow values form a lattice with a finite height. The height of the lattice is the length of the longest path taken from bottom to the top. In a set, this is limited by the number of possible elements which makes it so that the maximum number of times you can add a new element is a finite set. Then when the algorithm continues by processing a node, it applies the transfer function to the node’s input to produce a new output. Since we have mentioned it is a monotonic function, we will see that the output value for the node will be larger or smaller based on whether the movement is up or down the lattice. It can also not reverse the direction. As mentioned before, every update to a node’s output, prompted by a change in its input, must move the value with a non-reversable step in that single direction. As noted, the lattice will have a finite height, which limits the number of steps that can be taken. There cannot be an element added to a set an infinite number of times if the set has a finite maximum size. You cannot refine a constant propagation map an infinite number of times if the number of variables and possible constant states is finite. In the end, the algorithm should reach a state where any node’s input causes noticeable output change. Here the worklist becomes empty and the algorithm terminates with a stable solution.

1. Implement reaching definitions analysis

For this function, we want it to track which variable definitions can reach each program point without being overwritten. A couple of functions were added such as rd\_transfer. This function has kill and gen, where kill will remove all previous definitions of variables that get redefined in this block and gen creates new definitions for variables define in the block using unique ids. It will return a set of variable and definition\_id tuples. The rd\_merge function will take union of definitions from the predecessor blocks. We know that a definition reaches a point if it comes from any path.

1. Implement available expressions analysis.

This function is able to track expressions that are computed on all paths to a program point that haven’t been invalidated by variable reassignments. Key components added so that this works were the ae\_transfer and the ae\_merge, where ae\_transfer will use kill and gen to remove expressions that use variables being redefined and adds new expressions computed in this block. On ae\_merge takes the intersection of expressions from predecessor blocks where the expression is only available if it's computed on all paths. There are also helper functions get\_expressions, kill\_expressions, and uses\_variable.

1. Construct a test set and use it to thoroughly test your implementations using Turnt.

In addition to your answers to 1 and 2 above, please turn in your code files, your test cases, and a README file. The easiest way to do this is to create a github repository for the class and create a directory for this assignment with a subdirectory called test for your test cases. Your README file should include a description of the code along with usage instructions.

Grading: Each analysis question or implementation is worth 5 points. The README and test cases make up the remaining 5 points.