CSE 401 - Section 9 – Data Flow & SSA - Solutions

1. **Reaching Definitions**

   Consider the following small program that we used as a dataflow example for live variable analysis in lecture. This time all the statements are labeled individually, and we want to compute reaching definitions.

   L0: a = 0
   L1: b = a + 1
   L2: c = c + b
   L3: a = b * 2
   L4: if a < N goto L1
   L5: return c

   The reaching definitions dataflow problem is to determine for each variable definition which other blocks in the control flow graph could potentially see the value of the variable that was assigned in that definition. **To simplify things, we will treat each individual statement above as a separate block, and use the statement labels as the names of both the blocks and the definitions in them**. So, for example, reaching definition analysis would allow us to determine that definition L0, which assigns to *a*, can reach block L1.

   A definition *d* in block *p* reaches block *q* if there is at least one path from *p* to *q* along which definition *d* is not redefined.

   Dataflow sets for this task:

   GEN(*b*): the definitions assigned and not killed in block b
   KILL(*b*): the definitions of variables overwritten in block b
   IN(*b*): the definitions that are reaching upon entering block b
   OUT(*b*): the definitions that are reaching upon exiting block b

   Equations for IN(*b*) and OUT(*b*) in terms of other sets and other basic blocks:

   $IN(b) = \bigcup_{p \in pred(b)} OUT(p)$
   $OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$

a) Compute the reaching definitions for the blocks in the given program, treating each statement as a separate block. In the following table, compute the GEN and KILL sets for each block, and then use those answers to compute successive iterations of the IN and OUT sets until there are no more changes to be made.

Note that this is a forward dataflow analysis problem, so the answer will converge faster if you compute from beginning to end (i.e. starting with L0).

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0			L0		L0
L1	L1		L0	L0, L1	L0, L1, L2, L3	L0, L1, L2, L3
L2	L2		L0, L1	L0, L1, L2	L0, L1, L2, L3	L0, L1, L2, L3
L3	L3	L0	L0, L1, L2	L1, L2, L3	L0, L1, L2, L3	L1, L2, L3
L4			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3
L5			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3

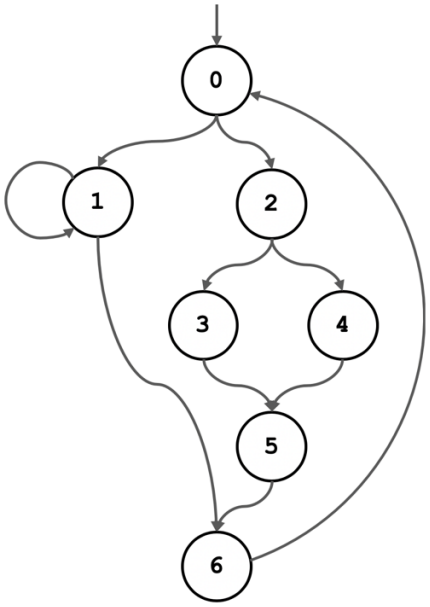
b) Now that we have completed our dataflow analysis, we want to apply optimizations to the code. After noticing that the definition L0 of the variable *a* is a constant value, we wonder if it is possible to use constant propagation to replace uses of the variable *a* with the constant 0.

Is it possible to replace the use of *a* in block L1 with the constant 0? Justify your answer using evidence from the sets that you computed during dataflow analysis.

No, it is not possible. To determine this, we would look at the IN set for block L1 -- the fact that the IN set contains two definitions of ‘a’ (L0 and L3) means we cannot perform this constant propagation. In other words, more than one definition of ‘a’ is a reaching definition to block L1, and therefore performing constant propagation would only preserve one possible value of ‘a’ and the generated code would not be equivalent.

Single Static Assignment Conversion

a. Consider the following simplified control flow graph. For each node in the graph, fill in the table with the set of nodes that are strictly dominated by that node and the set of nodes in its dominance frontier. Recall: node **X** dominates **Y** iff every path from the CFG entry point to **Y** includes **X**. Node **X** strictly dominates **Y** iff **X** dominates **Y** and **X ≠ Y**. Finally, node **Y** is in the dominance frontier of node **X** if **X** dominates an immediate predecessor of **Y** but **X** does not strictly dominate **Y**.

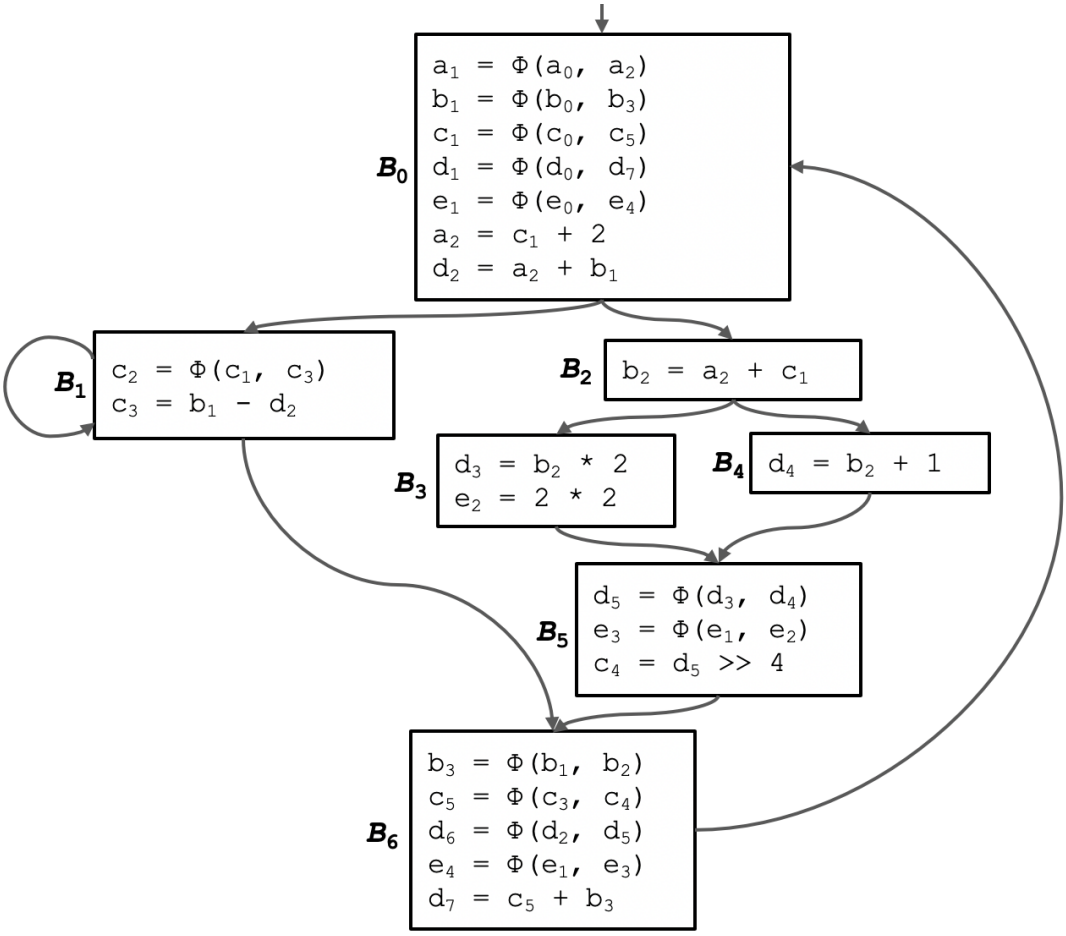


NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	ø	1, 6
2	3, 4, 5	6
3	ø	5
4	ø	5
5	ø	6
6	ø	0

For a more thorough walkthrough of this problem, see the section 10 slide deck which depicts the relevant sets graphically for each node.

b. Now, you will complete the conversion to SSA. Suppose the control flow graph from part (a) contains the following code. Convert this code to Single Static Assignment form. Remember that you can use the dominance frontiers computed in part (a) to determine which variables need to be merged (using phi functions) in each block.

The final SSA control flow graph becomes:



The steps taken to compute this final control flow graph are as follows. For a more graphical breakdown of these steps, consult the section 10 slide deck.

1. **We will need the sets of nodes that are strictly dominated by each node and in the dominance frontier of each node, as calculated in part (a). Doing this is important because we will use the dominance frontiers to determine where the phi functions need to be placed.**

2. **Initial sets of variables are computed that will be merged at each point. Recall that if a block Y is in the dominance frontier of block X, then any variable which is defined in block X will be given a phi function in block Y (the intuition here is that block Y may have been reached without going through block X, requiring a merge). Therefore, we determine that block 0 needs to merge a and d; block 1 needs to merge c; block 5 needs to merge d and e; and block 6 needs to merge c and b.**

3. **However, each merge of a variable will itself be a definition storing the result of a phi function. Therefore, if there are variables in the need-to-merge sets for any nodes, they also have to be copied to their dominance frontiers. For example, since d and e are now going to be merged in block 5, and**

block 6 is in the dominance frontier of block 5, we must add d and e as need-to-merge variables in block 6 (ending up with c, b, d, and e) This repeats until there are no more changes, which in this case means one more iteration to add e as a need-to-merge variable in block 0 (ending up with a, b, c, d, and e).

4. **Finally, we can produce the SSA code. Every definition of a variable must be distinct, and we simply increment the subscript with each new definition. Phi functions appear at the top of their block. We assume that all variables are defined as "a_0" or "b_0" before entering the control flow graph. If there is a loop backward, it is entirely possible that a definition will have a phi function that contains as an argument a definition with a greater subscript.**

2.