CS 243 Homework 1 Solutions

January 31, 2024

Directions:

- Before starting this assignment, read textbook section 9.2.6 (pp. 610–614) to learn about the Available Expressions analysis. This will be needed for both Problem 3 and the Gradiance quizzes.
- Submit written answers via Gradescope.
- Complete the corresponding Gradiance quizzes by the due date.
- You have **two late days** on assignments for the entire quarter. See course website for details. **There are no late days for Gradiance.**
- This is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.

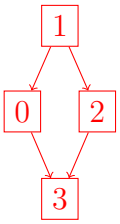
Problem 1. Indicate which of the following operator–domain pairs defines a semilattice. If so, also define the **top** ($\top$) and **bottom** ($\bot$) elements of the associated semilattice, if they exist. If not, justify your answer by listing at least one property of semilattice that fails to hold.

1. Set intersection over $\{\{1\}, \{2\}, \{1, 2\}\}$
2. Set union over the power set of $\mathbb{Z}^+$ (all positive integers)
3. Boolean implication ($\implies$) operator over boolean values $\{T, F\}$
4. The GCD (Greatest Common Divisor) function over $\{2, 4, 8\}$
5. Geometric mean ($a \wedge b = \sqrt{a \cdot b}$) over $\mathbb{R}^+$ (all positive real numbers)
6. Longest common prefix over S_5: the set of all strings of length $= 5$.
7. The meet operator $\wedge$, defined on $\{0, 1, 2, 3\}$, is given by the following table (the row and column headers specify the first and second operands, respectively):

$\wedge$	**0**	**1**	**2**	**3**
0	0	0	3	3
1	0	1	2	3
2	3	2	2	3
3	3	3	3	3

Answer:

1. No. It is not closed. $\{1\} \cap \{2\} = \emptyset$ which is not an element of the given set.
2. Yes. $\top = \emptyset, \bot = \mathbb{Z}^+$.
3. No. Idempotence does not hold, since $F \implies F = T$. Commutativity does not hold, since $T \to F = F$ but $F \to T = T$. Associativity does not hold, since $(F \to T) \to F = F$, but $F \to (T \to F) = T$.
4. Yes. $\top = 8, \bot = 2$.
5. No. Associativity does not hold, since $\sqrt{\sqrt{1 * 2} * 3} \neq \sqrt{1 * \sqrt{2 * 3}}$.
6. No. It is not closed. The longest common prefix of *aaaaa* and *bbbbb* is the empty string, which is not in S_5.
7. Yes. $\top = 1, \bot = 3$. The semilattice is shown in the following figure:



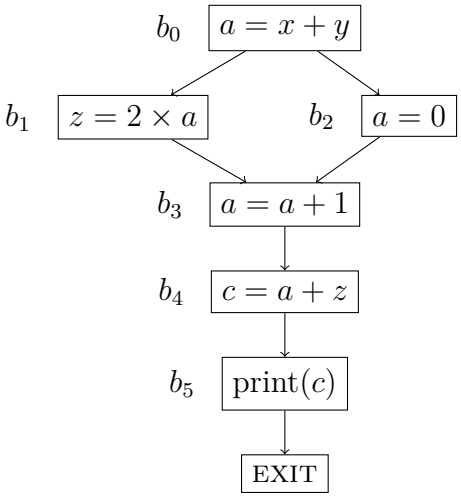
Problem 2. Live Range Analysis.

A path is *definition-free* with respect to a variable y if there does not exist a definition of variable y along that path. The live range of a definition d: $y = x + z$ that defines variable y includes all the program points p such that:

1. There is a path from d to p that is definition-free with respect to y, and
2. There is a path from p to q, a statement that uses (i.e., reads) the variable y, that is definition-free with respect to y.

To clarify, for any statement d within basic block b, there is no path from d to entry(b) (unless b participates in a cycle in the CFG), but there is a path from d to exit(b).

Intuitively, the live range of a definition consists of points along all subsequent paths until the variable defined is guaranteed never to be used before redefinition (or exit) along all paths. This concept is applicable to register allocation: two definitions can be assigned to the same register if their live ranges do not intersect.



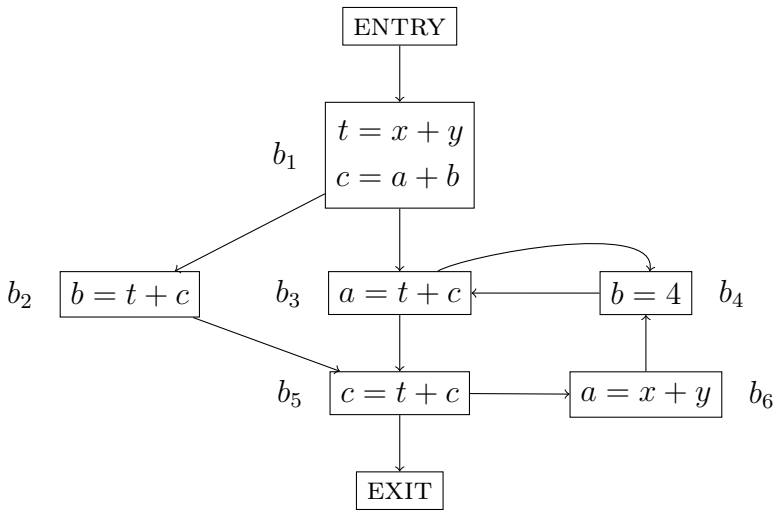
In the above example, the live range of definition b_0 is exit(b_0), entry(b_1), exit(b_1), and entry(b_3). Notably, entry(b_2) is *not* part of the live range of definition b_0 (think of why this is the case). Similarly, the live range of the definition b_4 is exit(b_4), entry(b_5). The two live ranges do not intersect, so c can reuse a's register.

Briefly describe an analysis that computes the live range for each definition in a program. *Hint: you may use algorithms discussed in class.*

Answer: Informally, a program point p is in the live range of definition $d : x = \cdots$ if (1) x is a live variable at p, and (2) d is a reachable definition at p. This gives rise to the following algorithm. First, run Reaching Definitions and Live Variables analyses on the program. Let IN_{RD} and OUT_{RD} be the result of Reaching Definitions and let IN_{LV} and OUT_{LV} be the result of Live Variables. Then for every definition $d : x = \cdots$, the live range of d can be computed as

$$\{\text{entry}(b) | x \in \text{IN}_{RD}[b] \wedge d \in \text{IN}_{LV}[b]\} \cup \{\text{exit}(b) | x \in \text{OUT}_{RD}[b] \wedge d \in \text{OUT}_{LV}[b]\}.$$

Problem 3. Read textbook section 9.2.6 (pp. 610–614) to learn about the Available Expressions analysis. Compute the available expressions on entry and exit for each basic block in the following flow graph.



Available expressions:

$$\begin{aligned}
\text{IN}[b_1] &= \emptyset, & \text{OUT}[\text{ENTRY}] &= \emptyset,\\
\text{IN}[b_2] &= \{x+y, a+b\}, & \text{OUT}[b_1] &= \{x+y, a+b\},\\
\text{IN}[b_3] &= \{x+y\}, & \text{OUT}[b_2] &= \{x+y, t+c\},\\
\text{IN}[b_4] &= \{x+y\}, & \text{OUT}[b_3] &= \{x+y, t+c\},\\
\text{IN}[b_5] &= \{x+y, t+c\}, & \text{OUT}[b_4] &= \{x+y\},\\
\text{IN}[b_6] &= \{x+y\}, & \text{OUT}[b_5] &= \{x+y\},\\
\text{IN}[\text{EXIT}] &= \{x+y\}. & \text{OUT}[b_6] &= \{x+y\},
\end{aligned}$$

Problem 4. True or False? Briefly justify your answer.

1. As discussed in lecture 3, $\text{MOP}(b) = \wedge f_{p_i}$ for all paths p_i reaching block b. Consider a program where the maximum path length among any of these paths p_i is N. You are applying the iterative data flow algorithm discussed in class to a monotone data flow framework. Let $\text{IN}_N[b]$ be the value of $\text{IN}[b]$ after N iterations of the algorithm, $\text{IN}_N[b] \leq \text{MOP}(b)$ holds true for all blocks b.

2. Let S be the set $\{a, b, c, d, e\}$. Consider a semi-lattice where the partial order is defined by the subset relation $\subseteq$ over the power set of S, $\mathcal{P}(S)$. Let $f : \mathcal{P}(S) \to \mathcal{P}(S)$ be the complement function, that is, $f(A) = A^c$. The data flow framework with f and the partial order $\subseteq$ is a distributive framework.

3. A distributive data flow framework is also a monotone data flow framework.

4. If the semilattice of a data flow framework has an infinite domain, then the iterative algorithm cannot converge to a fixed point solution.

5. Suppose $f : S \to S$ and $g : S \to S$ are monotone functions with respect to some partial order $\leq$. Then $f \circ g$ is also monotone (where $(f \circ g)(x) = f(g(x))$).

6. Suppose we have a partial order defined by the superset ($\supseteq$) relation over the power set of $\mathbb{Z}$, $\mathcal{P}(\mathbb{Z})$. We define function $f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ as $f(S) = ((S \cap \{1, 2, 3, 4, 5\}) \cup \{6\})$. f is a monotone function with respect to this partial order. *Hint: Use your answer from the previous part.*

7. The Live Variable analysis discussed in class can be applied to functions that use both local and global variables without needing any modifications.

Answer:

1. True.

   Two possible proof ideas: one showing that every block b has converged to the MFP after N_b iterations (where N_b is the maximum path length for that block), and another directly showing that $\text{IN}_N[b] \leq \text{MOP}(b)$. (WLOG this is a forward dataflow; for a backward dataflow, we can reverse the CFG.)

   For the first, we proceed via induction on the maximum path length to a block. The only block with maximum path length 0 is the entry block, but IN[Entry] is undefined, so we must show this for $k = 1$. If a block has maximum path length 1, then its only predecessor can be the entry block. Then $\text{IN}_1[b] = \text{OUT}_0[\text{Entry}]$ is fixed as OUT[Entry] never changes. Next, assuming the inductive hypothesis that all blocks with maximum path length k have reached a fixed point after iteration k, we must show that all blocks for $k + 1$ at iteration $k + 1$. Let a_i be the predecessors of a block b with maximum path length $k + 1$. Then all a_i must have maximum path length at most k, so it has been fixed the previous iteration. By definition of a fixed point, the value after this iteration remains the same, so $\text{IN}_{k+1}[a_i] = \text{IN}_k[a_i]$. Therefore the value we read in the $k + 1$th iteration is equal to $\text{OUT}_{k \textbf{ or } k+1} = f_i(\text{IN}_k[a_i])$ (where the subscript depends on if a_i is traversed before or after b). Then to show that we have reached a fixed point, we note that $\text{IN}_{k+2}[a_i] = \text{IN}_{k+1}[a_i] = \text{IN}_k[a_i]$, and so all values we read in the

next iteration will be the same. Therefore, b has reached a fixed point after $k+1$ iterations. Since we know from class that MFP $\leq$ MOP, we have that $\text{IN}_N[b] \leq \text{MOP}(b)$ for all b.

For the second, we perform strong induction on k to show that $\text{IN}_k[b] \leq \text{MOP}_k(b)$ for all k, where MOP_k is the meet-over-paths for paths of length at most k, for b having maximum path length at most k. The base case is essentially the same as before. For the inductive step, consider a block b with maximum path length $k+1$. We know $\text{IN}_{k+1}[b] = \bigwedge_i \text{OUT}_{k \textbf{ or } k+1}[a_i] = \bigwedge_i f_i(\text{IN}_{k \textbf{ or } k+1}[a_i])$, and due to monotonicity, we have $\text{IN}_{k+1}[b] \leq \bigwedge_i f_i(\text{IN}_{k_i}[a_i])$, where $k_i \leq k$ is the maximum path length for a_i, as $\text{IN}_{i+1} \leq \text{IN}_i$ for all i, since the only operations performed are meet and a monotonic transfer function, and the initial value of interior nodes is $\top$ which is greater than everything else. By the inductive hypothesis, we have $\text{IN}_{k+1}[b] \leq \bigwedge_i f_i(\text{IN}_{k_i}[a_i]) \leq \bigwedge_i f_i(\text{MOP}_{k_i}[a_i]) = \bigwedge_i f_i(\text{MOP}_k[a_i]) \leq \text{MOP}_{k+1}[b]$, where the last inequality holds via monotonicity and associativity, as each path to b can be partitioned on the previous block:

$$\begin{aligned}
\text{MOP}_{k+1}[b] &= \bigwedge_{\text{paths } p \text{ having length } l \,|\, p \text{ ends with } b} f_{p_l}(f_{p_{l-1}}(\dots(f_{p_1}(\text{OUT}_0[\text{Entry}])))) \\
&= \bigwedge_i \bigwedge_{\text{paths } p \,|\, p \text{ ends with } a_i} f_i(f_{p_{l-1}}(\dots(\text{OUT}_0[\text{Entry}]))) \\
&\geq \bigwedge_i f_i\left(\bigwedge_{\text{paths } p \,|\, p \text{ ends with } a_i} f_{p_{l-1}}(\dots)\right) \\
&= \bigwedge_i f_i(\text{MOP}_k[a_i]).
\end{aligned}$$

2. False. Consider $A_1 = \{a\}, A_2 = \{a, b\}$. Since the partial order is imposed using the subset relation $\subseteq$, we know that $A_1 \leq A_2$. Now, $f(A_1) = A_1^c = \{b, c, d, e\}$ and $f(A_2) = A_2^c = \{c, d, e\}$. Clearly, $f(A_1) \not\subseteq f(A_2) \implies f(A_1) \not\leq f(A_2)$. Therefore, f is not monotonic, and cannot therefore be distributive (as distributivity implies monotonicity).

3. True. A framework $(F, V, \wedge)$ is monotone iff $f(x \wedge y) \leq f(x) \wedge f(y)$. On the other hand, it is distributive iff $f(x \wedge y) = f(x) \wedge f(y)$. Since $f(x \wedge y) = f(x) \wedge f(y) \implies f(x \wedge y) \leq f(x) \wedge f(y)$, a distributive framework is also a monotonic framework.

4. False. The iterative algorithm may converge to a fixed point solution. For example: in constant propagation, the domain may be infinite, but owing to the finite descending chain of the semilattice and monotonicity of the data flow framework, we converge to a fixed point solution. Note: if dataflow does not converge, then at least one point must continue to change; consider the sequence of its values (excluding duplicates), which is guaranteed to be descending from monotonicity. Then this forms an infinite descending chain; if the semilattice has finite descending chains (has bounded height), this cannot happen.

5. True. Let's assume that $a \leq b$. Since we know that g is a monotonic function, by the property of monotonic functions, we conclude that $g(a) \leq g(b)$. Similarly, since f is monotonic, we can say that $\forall x, y \in S, x \leq y \implies f(x) \leq f(y)$. This means that $g(a) \leq g(b) \implies f(g(a)) \leq f(g(b))$. Therefore, $a \leq b \implies f(g(a)) \leq f(g(b))$. Hence, $f \circ g$ is monotonic.

6. True. Let's denote $g(S) = S \cap \{1, 2, 3, 4, 5\}$ and $h(S) = S \cup \{6\}$. Then, $f(S) = h(g(S))$.

Now, we can prove that both g and h are monotonic. Here is the reasoning for monotonicity of g. Let us consider $X, Y \in \mathcal{P}(\mathbb{Z})$ such that $X \leq Y \implies X \subseteq Y$. This means that $\forall x \in \mathbb{Z}, x \in X \implies x \in Y$. This implies that $\forall x \in \{1,2,3,4,5\}, x \in X \implies x \in Y$. This implies that $X \cap \{1,2,3,4,5\} \subseteq Y \cap \{1,2,3,4,5\} \implies X \cap \{1,2,3,4,5\} \leq Y \cap \{1,2,3,4,5\} \implies g(X) \leq g(Y)$. Therefore, g is monotonic. Similar reasoning can be given for h. Since both g and h are monotonic, $f = h \circ g$ is also monotonic.

7. False. Since data flow analysis deals with intra-procedural code and we generally look at one function at a time, we do not know the usage of the global variables outside the function. Therefore, we generally need to assume that the global variables are live at exit of the flow graph. Hence, we should initialize the boundary condition, IN[EXIT], to be the set of global variables in the program.

Problem 5. Initial Values in Data flow Analysis.

This question asks you to think about how changes to initial values in a data flow analysis can affect the result. Recall that an answer to a data flow problem is considered "safe" if it is no greater than the ideal solution.

Suppose you have defined a forward data flow algorithm that has a monotone framework and finite descending chains. You accidentally initialized OUT[B] to $\perp$ for all nodes other than ENTRY.

1. Will your algorithm still give a safe answer for all flow graphs? If so, please explain. If not, provide a counterexample.

2. Will your algorithm give the MOP solution for all flow graphs? If so, please explain. If not, provide a counterexample.

3. If not, will it give the MOP solution for some flow graphs? If it will, provide an example.

Answer:

1. Yes, it will give a safe answer for all flow graphs. Let IN'[b] and OUT'[b] be the values of IN and OUT for basic block b for the original case and IN[b] and OUT[b] be the values for the case where we accidentally initialized OUT[b] to $\perp$.

   Use mathematical induction.

   Let $\text{OUT}_k[b]$, $\text{OUT'}_k[b]$, $\text{IN}_k[b]$, $\text{IN'}_k[b]$ be the corresponding values at the end of the k^{th} iteration.

   Initially: $\text{OUT}_0[b] \leq \text{OUT'}_0[b]$ for all b. $\text{IN}_0[b] = \wedge\text{OUT}_0[p_b]$ for all predecessors p_b of b. Since $\text{OUT}_0[b] \leq \text{OUT'}_0[b]$ for all b, by properties of the meet operator, we say that $\text{IN}_0[b] \leq \text{IN'}_0[b]$ for all b.

   After running iterations of the data flow algorithm: $\text{OUT}_k[b] = f_b(\text{IN}_{k-1}[b])$ for all b where f_b is the transfer function for block b. If $\text{IN}_{k-1}[b] \leq \text{IN'}_{k-1}[b]$, then by monotonic property of transfer functions, $\text{OUT}_k[b] \leq \text{OUT'}_k[b]$. Similarly, $\text{IN}_k[b] = \wedge\text{OUT}_k[p_b] \leq \wedge\text{OUT'}_k[p_b] = \text{IN'}_k[b]$. Therefore, we always get a safe answer.

2. No. It will fail on cyclic flow graphs. For example, the following reaching definition problem will do:

   `ENTRY - BLOCK A - BLOCK B - EXIT. Block A has a self loop.`

   `BLOCK A: d1:  x = 1.`

   `BLOCK B: d2:  y = 1.`

   OUT[A] = {d1, d2} due to being initialized to bottom, whereas it should be just {d1}.

3. Yes, it will give the MOP on any acyclic flow graph as the boundary condition will propagate and overwrite the correct values of IN and OUT as per MOP.

Problem 6. Detecting memory leaks in a toy instruction set

A common challenge associated with the use of pointers in code is the occurrence of memory leaks. When a pointer points to a particular memory location, any subsequent reassignment of the pointer to a different location or reallocation of the pointer may result in the abandonment of the old memory location without freeing it, leading to a memory leak. This concern becomes particularly significant in long-running programs as leaked data can accumulate and cause them to fail by running out of memory. Hence, it becomes imperative to proactively address potential memory leaks within a program.

For this problem, let us consider a toy instruction set where only local, scalar variables can be pointers. In other words, you cannot store pointers into allocated data, there are no complex data structures like linked lists and trees. The available instructions are:

1. `p = allocate()` : Allocate memory and hold the reference in pointer `p`.
2. `free(p)` : Free the memory pointed by pointer `p`.
3. `move(p, q)` : Transfer the memory location referred by `p` to `q`, so that now `q` holds the reference to the memory location and `p` stores null. In C/C++ syntax, this is essentially equivalent to: `q = p; p = null;`
4. No other instructions can allocate memory, free memory, change the values of pointer variables, or assign pointers to any variables.

In simple terms, a memory location will be pointed to by **at most one pointer** at a time.

Your task is to warn programmers of any instructions in the program that MAY cause a memory leak. Specifically, you need to flag any instruction that can potentially cause a memory location to never be freed.

(1) Define a data flow analysis to solve this problem by filling in the table below, and (2) Specify how you would use the data flow results to issue warnings on specific vulnerable statements. You may treat each instruction as a basic block.

There are other potential problems such as dereferencing an uninitialized pointer, use-after-free, etc. You may ignore such errors in this analysis.

Direction of your analysis (forward/backward)	
Lattice elements and meaning	
Is there a top element? If yes, what is it?	
Is there a bottom element? If yes, what is it?	
Meet operator	
Transfer function	
Boundary condition	
Interior points	

Issue warning for basic block b if:

Direction of your analysis (forward/backward)	Forward
Lattice elements and meaning	Set of pointers that are currently pointing to some location in memory
Is there a top element? If yes, what is it?	Yes, the empty set $\emptyset$
Is there a bottom element? If yes, what is it?	Yes, the universal set of all pointers
Meet operator	Union
Transfer function	$\text{OUT}[b] = \begin{cases} \text{IN}[b] \cup \{\texttt{p}\}, \text{ if } b \text{ is of form } \texttt{p = allocate()} \\ \text{IN}[b] - \{\texttt{p}\}, \text{ if } b \text{ is of form } \texttt{free(p)} \\ (\text{IN}[b] \cup \{\texttt{q}\}) - \{\texttt{p}\}, \\ \quad \text{if } b \text{ is of form } \texttt{move(p, q)} \text{ and } p \in \text{IN}[b] \\ \text{IN}[b], \text{ otherwise} \end{cases}$
Boundary condition	Empty set
Interior points	Empty set

Issue warning for basic block b if:

1. b is of the form `p = allocate()` and `p` $\in \text{IN}[b]$
2. b is of the form `move(p, q)` and `q` $\in \text{IN}[b]$