CS 243 Homework 1 Solutions

Winter 2023

Due: January 25, 2023 at 11:59pm

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 5a 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 semilattice axiom that fails to hold.

1. Set union over the power set of $\mathbb{Z}$
2. Set intersection over $\{\emptyset, \{1\}, \{2\}\}$
3. Set symmetric difference (i.e., $(A - B) \cup (B - A)$) over the power set of $\mathbb{Z}$
4. Boolean OR operator over boolean values $\{T, F\}$
5. The LCM (Least Common Multiple) function over $\mathbb{Z}^+$ (all positive integers)
6. The GCD (Greatest Common Divisor) function over $\{2, 4, 8\}$
7. Arithmetic mean ($\frac{a+b}{2}$) over $\mathbb{R}$

1. Yes, top is $\emptyset$ and bottom is $\mathbb{Z}$. Note that with set intersection as the operator instead, the top and bottom elements would be flipped.
2. Yes, top does not exist and bottom is $\emptyset$.
3. No, idempotence not satisfied (the symmetric difference of $\{1\}$ and itself is $\emptyset$).
4. Yes, top is $F$ and bottom is $T$. Note with Boolean AND as the operator instead, the top and bottom elements are flipped.
5. Yes, top is 1 and bottom does not exist.
6. Yes, top is 8 and bottom is 2.
7. No, associativity not satisfied:

$$\frac{1 + \frac{2+3}{2}}{2} = 1.75 \neq 2.25 = \frac{\frac{1+2}{2} + 3}{2}.$$ 

\footnote{Set difference $A - B$ is defined as the set $\{v \mid v \in A \text{ and } v \notin B\}$. It is sometimes also written as $A \setminus B$.}
Problem 2. True or False? Briefly justify your answer.

1. A monotone framework is also a distributive framework.
2. If the semilattice of a dataflow framework has a finite domain, then the iterative algorithm must converge to some fixed point solution.
3. Every nonempty finite semilattice has a bottom element.
4. A semilattice can have multiple distinct top elements.
5. Suppose \( f : S \to S \) and \( g : S \to S \) are monotonic functions with respect to some partial order \( \leq \). Then \( f \circ g \) is also monotonic (where \( (f \circ g)(x) = f(g(x)) \)). Hint: a function \( f : X \to Y \), \( f \) is monotonic if and only if \( \forall a, b \in X, a \leq b \implies f(a) \leq f(b) \).
6. Suppose we have a partial order defined by the subset (\( \subseteq \)) relation over the power set of \( \mathbb{Z} \), \( P(\mathbb{Z}) \). We define function \( f : P(\mathbb{Z}) \to P(\mathbb{Z}) \) as \( f(S) = (S \cup \{3\}) - \{4\} \). \( f \) is a monotonic function with respect to this partial order. Hint: Use your answer from the previous part.

1. False. Obvious from definition.
2. False. Using the iterative algorithm for dataflow analysis, convergence is only guaranteed with a monotone framework and finite descending chains. Example: consider the following forward dataflow framework: the domain is \( \{\emptyset, \{1\}\} \), set union as the meet operator, two blocks \( b_1, b_2 \) leading to each other, \( \text{OUT}[b_i] = \{1\} \) – \( \text{IN}[b_i], i = 1, 2 \). The iterative algorithm will never converge regardless of the initialization of internal blocks/entry block.
3. True. Suppose the domain is \( S \), let \( \bot = \bigwedge_{x \in S} x \). Then for all \( y \in S \),
   \[
   y \land \bot = y \land \left( y \land \bigwedge_{x \in S, x \neq y} x \right) \quad \text{(definition of \( \bot \))}
   = (y \land y) \land \bigwedge_{x \in S, x \neq y} x \quad \text{(associativity)}
   = y \land \bigwedge_{x \in S, x \neq y} x \quad \text{(idempotence)}
   = \bot. \quad \text{(definition of \( \bot \))}
   
   By definition of the bottom element, \( \bot \) is the bottom.
4. False. Suppose we have distinct \( T_1, T_2 \). From the property of the top element, \( T_1 \land T_2 = T_2 \) and \( T_2 \land T_1 = T_1 \). Then from communitivity, we have \( T_1 = T_2 \) which leads to a contradiction.
5. True. Without loss of generality, assume \( a \leq b \). Then we have \( g(a) \leq g(b) \) since \( g \) is a monotonic function. Similarly, since \( f \) is monotonic, \( \forall x, y \in S, x \leq y \implies f(x) \leq f(y) \). This means that \( g(a) \leq g(b) \implies f(g(a)) \leq f(g(b)) \). Therefore, \( a \leq b \implies f(g(a)) \leq f(g(b)) \).
6. True. Denote \( g(x) = x \cup \{3\} \), \( h(x) = x - \{4\} \). Then \( f(x) = h(g(x)) \). Since \( g, h \) are both monotonic, \( f(x) \) is also monotonic by conclusion from last question. The monotonicity of \( g \) can be proven by discussing the two cases of \( 3 \in x \) and \( 3 \notin x \), respectively. The similar reasoning goes with \( h \).
Problem 3. Live Range Analysis.

A path is \textit{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 \( \text{entry}(b) \) (unless \( b \) participates in a cycle in the CFG), but there is a path from \( d \) to \( \text{exit}(b) \).

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.

\[
\begin{align*}
& b_0 & a = x + y \\
& b_1 & z = 2 \times a \\
& b_2 & a = 0 \\
& b_3 & a = a + 1 \\
& b_4 & b = a + z \\
& b_5 & \text{print}(b) \\
& \text{EXIT}
\end{align*}
\]

In the above example, the live range of definition \( b_0 \) is \( \text{exit}(b_0) \), \( \text{entry}(b_1) \), \( \text{exit}(b_1) \), and \( \text{entry}(b_3) \). Notably, \( \text{entry}(b_2) \) is \textit{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 \( \text{exit}(b_4) \), \( \text{entry}(b_5) \). The two live ranges do not intersect, so \( b \) can reuse \( a \)'s register.

Describe an analysis that computes the live range for each definition in a program. \textit{Hint: you may use algorithms discussed in class.}

Informally, a program point \( p \) is in the live range of definition \( d : x = \ldots \) 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 \( \text{IN}_{\text{RD}} \) and \( \text{OUT}_{\text{RD}} \) be the result of Reaching Definitions and let \( \text{IN}_{\text{LV}} \) and \( \text{OUT}_{\text{LV}} \) be the result of Live Variables. Then for every definition \( d : x = \ldots \), the live range of \( d \) can be computed as

\[
\{ \text{entry}(b) \mid x \in \text{IN}_{\text{LV}}[b] \land d \in \text{IN}_{\text{RD}}[b] \} \cup \\
\{ \text{exit}(b) \mid x \in \text{OUT}_{\text{LV}}[b] \land d \in \text{OUT}_{\text{RD}}[b] \}.
\]
Problem 4. 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:

\[
\begin{align*}
\text{IN}[b_1] &= \emptyset, \\
\text{IN}[b_2] &= \{x+y, 2*a\}, \\
\text{IN}[b_3] &= \{x+y, 2*a\}, \\
\text{IN}[b_4] &= \{x+y\}, \\
\text{IN}[b_5] &= \{x+y, 2*a\}, \\
\text{IN}[\text{EXIT}] &= \{x+y, 2*a, x-t\}.
\end{align*}
\]

\[
\begin{align*}
\text{OUT}[\text{ENTRY}] &= \emptyset, \\
\text{OUT}[b_1] &= \{x+y, 2*a\}, \\
\text{OUT}[b_2] &= \{x+y\}, \\
\text{OUT}[b_3] &= \{x+y, 2*a\}, \\
\text{OUT}[b_4] &= \{x+y, 2*a\}, \\
\text{OUT}[b_5] &= \{x+y, 2*a, x-t\}.
\end{align*}
\]
Problem 5. Initial Values in Dataflow Analysis.
This question asks you to think about how changes to initial values in a dataflow analysis can affect the result. Recall that an answer to a dataflow problem is considered “safe” if it is no greater than the ideal solution.

Part a. Recall the Live Variables (LV) analysis, a backward dataflow algorithm covered in class. The version of LV presented in class initializes all interior points to $\emptyset$ (i.e., $\top$), and the boundary condition in $\text{IN}[\text{EXIT}]$ to $\emptyset$ as well.

We would now like to extend the LV analysis to work with both local and global variables. Every program variable is either local or global. You may assume that local variables after the exit block are never going to be used, while no such assumption can be made for global variables. Carry out this extension by only changing the initial values.

1. What should be the initial value of the interior points?
2. What should be the initial value of the boundary condition?
3. Suppose we have a list of program variables, but do not know which ones are global and which ones are local. How should we set the interior points and boundary condition?
4. Reconsider the version of LV presented in class. Does it treat all variables as global, treat all variables as local, or neither?

1. Set all interior points to $\emptyset = \top$.
2. Initialize the boundary condition to the set of global variables.
3. For safety, assume all program variables are global. Initialize the boundary condition to the universal set (i.e., $\bot$). This is safe since $\bot$ is no greater than any other set, including the actual set of global variables.
4. The version of LV presented in class initializes the boundary condition to $\emptyset$, and so assumes the set of global variables is $\emptyset$. In other words, it treats all variables as local.
Part b. Suppose you have defined a backward dataflow algorithm that has a monotonic transfer function and finite descending chains. You accidentally initialized $\text{IN}[^{\text{EXIT}}]$ to $\perp$.

1. Will your algorithm give a safe answer for all flow graphs?
2. If not, will it give a safe answer for some flow graphs? If it will, give an example.
3. Will your algorithm give the MOP solution for all flow graphs, with the exception of $\text{IN}[^{\text{EXIT}}]$?
4. If not, can it give the MOP solution for some flow graphs, with the exception of $\text{IN}[^{\text{EXIT}}]$? If it will, give an example.

Hint: While these questions relate to all backward dataflow algorithms, thinking about these questions in terms of a concrete analysis, like Live Variables, might help.

1. Yes. Run the iterative algorithm with the correct initial values and the modified initial values in parallel. At each iteration, by monotonicity of $f$, the values at each node in the modified run will remain $\leq$ that in the correct run. Do not stop until both runs reach FP (the run converged first will just remain its fixed results). Then by induction, when both algorithms stop, the values under the modified algorithm (Modified-FP) will still be $\leq$ the FP values under the correct algorithm (MFP). And since we know $\text{MFP} \leq \text{IDEAL}$, we have Modified-FP $\leq \text{IDEAL}$ as well.

2. N/A

3. No. Any nontrivial example would act as a counterexample.

4. Yes. Consider the Live Variables analysis with the following flow graph:

```
ENTRY

b_1 \quad x = x

EXIT
```

The MOP solution is

\[
\text{OUT}[^{\text{ENTRY}}] = \text{IN}[b_1] = \text{OUT}[b_1] = \{x\} = \perp,
\]

\[
\text{IN}[^{\text{EXIT}}] = \emptyset = \top,
\]

while with the modified algorithm it would be exactly the same except $\text{IN}[^{\text{EXIT}}] = \perp$. 
Problem 6. Detecting Use-After-Free

Use-after-free bugs occur when memory is accessed after it has been freed. This can lead to crashes and undefined behavior, and may make the program vulnerable to memory attacks. Detecting memory bugs is difficult because of aliases where multiple variables may point to the same location. We eliminate the complexity of this problem with the following simplified instruction set (p is a pointer variable and v is a non-pointer variable in the program):

- p = allocate();
- *p = v;
- v = *p;
- free(p);
- Other local code that does not allocate memory, free memory, or access pointers.

Your task is to warn programmers of dereferences in the program that refer to locations that MAY have been freed. (1) Define a dataflow analysis to solve this problem by filling in the table below, and (2) Specify how you use the dataflow results to issue warnings on the specific vulnerable memory access statements.

There are other errors such as dereferencing an uninitialized pointer. You may ignore such errors in this analysis. You may treat each instruction as a basic block.
<table>
<thead>
<tr>
<th>Direction of your analysis (forward/backward)</th>
<th>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lattice elements and meaning</td>
<td>Set of pointers that are freed</td>
</tr>
<tr>
<td>Lattice diagram</td>
<td>We didn’t ask for the diagram this time, but this might still be helpful.</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Lattice Diagram" /></td>
</tr>
<tr>
<td>Meet operator</td>
<td>Set union</td>
</tr>
<tr>
<td>Is there a top element?</td>
<td>Yes, the empty set</td>
</tr>
<tr>
<td>If yes, what is it?</td>
<td></td>
</tr>
<tr>
<td>Is there a bottom element?</td>
<td>Yes, the universal set of all pointers</td>
</tr>
<tr>
<td>If yes, what is it?</td>
<td></td>
</tr>
</tbody>
</table>
| Transfer function                           | \[ OUT[b] = \begin{cases} 
\text{IN}[b] - \{p\} & \text{if } b \text{ is of form } p = \text{allocate}(); \\
\text{IN}[b] \cup \{p\} & \text{if } b \text{ is of form } \text{free}(p); \\
\text{IN}[b] & \text{otherwise.} 
\end{cases} \] |
| Boundary condition                          | Either universal set or empty set. (If we want to error on unallocated pointers, the universal set. Otherwise, the empty set.) |
| Interior points                             | The empty set |

Issue warning for basic block \( b \) if \( b \) contains \(*p\) (either read or store) and \( p \in \text{IN}[b] \). (Note: we also accepted answers that warned if \( b \) contains \( \text{free}(p) \) – that (a double-free) is also a memory error, though technically not a use-after-free.)