CS243 Homework 1

Winter 2020

Due: January 22, 2020 at 4:30 pm

Directions:

• Submit via Gradescope.

• You may use up to two of your remaining late days for this assignment, for a late
deadline of January 24, 2020 at 4:30 pm.

• Remember to complete the corresponding Gradience quizzes by the start of class on
the due date. There are no late days for Gradience.

• 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 operators defines a semi-lattice. Define the top and bottom element of the associated semi-lattice, if they exist. If not, justify your answer by listing the properties that fail to hold.

1. Set intersection. Yes, bottom is the empty set, top is the universal set.
2. Set symmetric difference (i.e. \((A \setminus B) \cup (B \setminus A)\)). No, not idempotent.
3. Component-wise maximum for tuples of natural numbers. Yes, top is zero vector, bottom does not exist.
4. Product mod 2 (on the domain \{0, 1\}). Yes, top is 1, bottom is 0.
5. Product mod 3 (on the domain \{0, 1, 2\}). No, not idempotent.
6. Arithmetic mean over real numbers. No, not associative.
7. The GCD (Greatest Common Divisor) function (on integers). No, not idempotent: \(\text{GCD}(-1, -1) = 1\). We also accepted “yes” as it can be argued that \(\text{GCD}(-1, -1) = -1\), in which case top is 0 (or does not exist depending on interpretation) and bottom is 1.

8. Cross product on three-dimensional \((\mathbb{R}^3)\) vectors, defined as follows:
\[
\begin{pmatrix}
a_1 \\ a_2 \\ a_3 \\
\end{pmatrix} \times \begin{pmatrix}
b_1 \\ b_2 \\ b_3 \\
\end{pmatrix} = \begin{pmatrix}
a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \\
\end{pmatrix}
\]
No, not idempotent, not associative, not commutative.

**Problem 2.** True or False? Briefly justify your answer.

1. A monotone framework is also a distributive framework. False. Given a data-flow framework with transfer function \(f\) and meet operator \(\wedge\), the framework is monotone if and only if \(f(x \wedge y) \leq f(x) \wedge f(y)\). The framework is distributive if and only if \(f(x \wedge y) = f(x) \wedge f(y)\). Therefore, a distributive framework is always monotone, however the converse is not true.

2. If the semi-lattice of a data-flow framework has a finite domain, then the iterative algorithm must converge to some fixed point solution. False. The framework must also be monotone. Consider the following framework: the domain is \{1; 2\}, the meet operator is \(\text{min}\), the bottom is 1, and the transfer function is \(f_{\text{switch}}(x) = 3 - x\). In this case, \(f_{\text{switch}}(\text{min}(1, 2)) = 2\). But \(\text{min}(f_{\text{switch}}(1), f_{\text{switch}}(2)) = 1\).

3. If a data-flow framework is monotone and the semi-lattice of the framework has finite descending chains, then the iterative algorithm must converge to some fixed point solution. True. Assuming the meet and transfer function is well defined over the semi-lattice of the data-flow framework, the algorithm must terminate: after each iteration, the values of IN and OUT of each basic block can either decrease or stay the same. If, for the sake of contradiction, there exists a basic block whose OUT decreases after
every iteration, then the semi-lattice of the framework does not have a finite-descending
chain. Therefore, the values of OUT of all the basic blocks must converge after some
number of iterations.

4. A semi-lattice has exactly one top element. False, a semi-lattice does not have to have
a top element. For instance, the semi-lattice with the domain of all natural numbers
and minimum as the meet operator.

5. Suppose we have a partial-order defined by the subset (⊆) relation over all sets of
integers, \( \mathcal{P}(\mathbb{Z}) \). We define function \( f : \mathcal{P}(\mathbb{Z}) \rightarrow \mathcal{P}(\mathbb{Z}) \) as
\( f(S) = S \cup \{1\} \). \( f \) is a
monotonic function with respect to this partial order.

Hint: a function \( f : X \rightarrow Y \), \( f \) is monotonic if and only if \( \forall a, b \in X, a \leq b \) implies
\( f(a) \leq f(b) \). True. Consider two sets \( S_1, S_2 \in \mathcal{P}(\mathbb{Z}) \) such that \( S_1 \subseteq S_2 \). If both \( S_1 \)
and \( S_2 \) contain 1, then \( f(S_1) = S_1 \) and \( f(S_2) = S_2 \), thus \( f(S_1) \subseteq f(S_2) \). If neither \( S_1 \)
or \( S_2 \) contains 1, then it naturally follows that \( S_1 \cup \{1\} \subseteq S_2 \cup \{1\} \). If one of the two sets
contains 1, it can only be that \( S_2 \) contains 1, since \( S_1 \subseteq S_2 \). Thus, \( f(S_2) = S_2 \) and
\( f(S_1) = S_1 \cup \{1\} \) must still be a subset of \( S_2 \).
Problem 3. 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 the variable \( y \), that is definition free with respect to \( y \).

Intuitively, the live range of a definition consists of points along all subsequent paths until either the variable is no longer used along that path or a new definition overwrites it. 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 \( a = x + y \) is \( \text{exit}(b_0), \text{entry}(b_1), \text{exit}(b_1) \) and \( \text{entry}(b_3) \). Similarly, the live range of the definition \( b = a + z \) is \( \text{exit}(b_4), \text{entry(\text{EXIT})} \). 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. You may use algorithms discussed in class.

To compute the live range, first compute all reaching definitions and all live variables at all program points. The live range of a definition is then the set of all program points that are reached by that definition where the associated variable is live.
Problem 4. Compute the available expressions (Chapter 9.2.6 in ALSU) on entry and exit for each basic block in the following flow graph:

\[
\begin{align*}
\text{in}[b1]: & \emptyset \\
\text{out}[b1]: & \{b + c, a + b\} \\
\text{in}[b2]: & \{b + c\} \\
\text{out}[b2]: & \{b + c\} \\
\text{in}[b3]: & \{b + c\} \\
\text{out}[b3]: & \{b + c, a + b\} \\
\text{in}[b4]: & \{b + c\} \\
\text{out}[b4]: & \{b + c, q + r\} \\
\text{in}[b5]: & \{b + c\} \\
\text{out}[b5]: & \{b + c, p + q\}
\end{align*}
\]
Problem 5.

Part A. Detecting Use-After-Free

Use-after-free bugs occur when memory is accessed after it has been freed. This can lead to crashes, 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 stripped down instruction set (p is a pointer variable and v is a non-pointer variable in the program):

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

You may treat each instruction as a basic block.

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

There are other errors such as dereferencing an undefined pointer. You may ignore such errors in this analysis.
<table>
<thead>
<tr>
<th><strong>Direction of your analysis</strong> (forward/backward)</th>
<th>Forwards.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lattice elements and meaning</strong></td>
<td>Pointers in the program. They represent which pointers may have been freed.</td>
</tr>
<tr>
<td><strong>Meet operator or lattice diagram</strong></td>
<td>Union meet operator.</td>
</tr>
<tr>
<td><strong>Is there a top element?</strong>&lt;br&gt;<strong>If yes, what is it?</strong></td>
<td>Empty set.</td>
</tr>
<tr>
<td><strong>Is there a bottom element?</strong>&lt;br&gt;<strong>If yes, what is it?</strong></td>
<td>Universal set (all pointers).</td>
</tr>
<tr>
<td><strong>Transfer function of a basic block</strong>&lt;br&gt;<code>\quad p = \text{allocate}();</code>&lt;br&gt;<code>\quad *p = v;</code>&lt;br&gt;<code>\quad v = *p;</code>&lt;br&gt;<code>\quad \text{free}(p);</code>&lt;br&gt;<code>\quad \text{other}</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b] - \{p\}</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b]</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b]</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b] \cup \{p\}</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b]</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b]</code>&lt;br&gt;<code>\quad \text{out}[b] = \text{in}[b]</code></td>
<td></td>
</tr>
<tr>
<td><strong>Boundary condition initialization</strong></td>
<td>Empty set.</td>
</tr>
<tr>
<td><strong>Interior points initialization</strong></td>
<td>Empty set.</td>
</tr>
</tbody>
</table>

For each basic block containing `*x;`, check if `x \in \text{in}[b]`. If so, warn on this access.
Part B. Detecting Memory Leaks

A memory leak occurs when memory is never freed after its last use. Assume again the instruction set from Part A. Your task is to warn of any potential memory leaks in a program.

1. Describe the conditions under which a memory leak may occur.

2. Describe one or more data flow analyses to detect potential memory leaks (fill out the same table as in Part A, one per analysis).

3. Describe how the data-flow results can be used to issue warnings on the precise statements where the leak may occur.

The two classes of memory leaks, given the limited operations, are the following:

1. Overwriting of a live pointer. This occurs when an `x = allocate();` can reach an `x = allocate();` without a `free(x);` in the path.

2. Exiting with a live pointer. This occurs when an `x = allocate();` can reach the program exit without encountering a `free(x);` in the path.

Define the framework as follows:

<table>
<thead>
<tr>
<th>Direction of your analysis (forward/backward)</th>
<th>Forwards.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lattice elements and meaning</td>
<td>Pointers in the program. They represent which pointers may be allocated.</td>
</tr>
<tr>
<td>Meet operator or lattice diagram</td>
<td>Union meet operator.</td>
</tr>
<tr>
<td>Is there a top element?</td>
<td>Empty set.</td>
</tr>
<tr>
<td>If yes, what is it?</td>
<td></td>
</tr>
<tr>
<td>Is there a bottom element?</td>
<td>Universal set (all pointers).</td>
</tr>
<tr>
<td>If yes, what is it?</td>
<td></td>
</tr>
<tr>
<td>Transfer function of a basic block</td>
<td></td>
</tr>
<tr>
<td><code>p = allocate();</code></td>
<td><code>out[b] = in[b] ∪ \{p\}</code></td>
</tr>
<tr>
<td><code>*p = v;</code></td>
<td><code>out[b] = in[b]</code></td>
</tr>
<tr>
<td><code>v = *p;</code></td>
<td><code>out[b] = in[b]</code></td>
</tr>
<tr>
<td><code>free(p);</code></td>
<td><code>out[b] = in[b] - \{p\}</code></td>
</tr>
<tr>
<td>other</td>
<td><code>out[b] = in[b]</code></td>
</tr>
<tr>
<td>Boundary condition initialization</td>
<td>Empty set.</td>
</tr>
<tr>
<td>Interior points initialization</td>
<td>Empty set.</td>
</tr>
</tbody>
</table>
Warn the programmer that you may exit with live variables if $\text{in[exit]}$ is non-empty. If $x \in \text{in[b]}$ for any basic block $b$ that contains $x = \text{allocate}();$, warn the programmer that this allocation may overwrite a live variable.