CS243 Homework 1

Winter 2021

Due: January 27, 2021 at 4:00 pm

Directions:

- This homework includes a Gradiance quiz (please see the website) and the following questions, to be submitted via Gradescope.
- You may use up to two of your remaining late days for this assignment, for a late deadline of January 29, 2021 at 4:00 pm. However, you need to complete the Gradiance quiz by the start of class on the due date. **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 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 Union. Yes, bottom is the universal set, top is the empty set.

2. Set symmetric difference (i.e. $(A \setminus B) \cup (B \setminus A)$). No, not idempotent.

3. Component-wise addition of tuples of natural numbers. No, $x \wedge x \neq x$.

4. Or over boolean values ($\{T, F\}$). Yes, top is F, bottom is T.

5. Mediant ($\frac{a}{b} \oplus \frac{c}{d} = \frac{a+c}{b+d}$) over positive rational numbers (including 0) where a, b, c, d are non-negative integers. *Hint: Be sure to first check that this operation is well-defined and closed over the positive rational numbers!* No, not well-defined. $\frac{0}{1} \oplus \frac{1}{1} = \frac{1}{2}$, but $\frac{0}{2} \oplus \frac{1}{1} = \frac{1}{3}$. Therefore, $0 \oplus 1$ is not well-defined.

6. Arithmetic mean over real numbers (for $a, b \in \mathbb{R}$, define $a \wedge b = \frac{a+b}{2}$). Consider $(2 \wedge 4) \wedge 6$, where $a \wedge b = \frac{a+b}{2}$. $2 \wedge 4 = 3$, and $3 \wedge 6 = 4.5$. Now consider $2 \wedge (4 \wedge 6)$. $4 \wedge 6 = 5$, but $2 \wedge 5 = 3.5$. Therefore, $(2 \wedge 4) \wedge 6 \neq 2 \wedge (4 \wedge 6)$ so this operation is not associative.

   If finite-precision real numbers are assumed, then idempotence is not guaranteed either, since with any floating point representation, there will be gaps between representable values.

7. The GCD (Greatest Common Divisor) function on non-negative, non-zero integers. Yes, no top 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. Using the iterative algorithm for data flow analysis, convergence is only guaranteed with a monotone framework and finite descending chains.

3. A semi-lattice can have multiple top elements. False. Suppose T_1 and T_2 are top elements, in a semi-lattice. This means that $x \wedge T_1 = x$ and $x \wedge T_2 = x$; however, according to the first relation, $T_2 \wedge T_1 = T_2$ and according to the second relation, $T_1 \wedge T_2 = T_1$. Since the semi-lattice operator is commutative, this means that $T_2 = T_2 \wedge T_1 = T_1 \wedge T_2 = T_1$. Therefore, $T_1 = T_2$ which implies that there can only be one top element.

4. 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.

   Hint: a function $f : X \to Y$, f is monotonic if and only if $\forall a, b \in X, a \leq b$ implies $f(a) \leq f(b)$. True. Consider $a \leq b$ according to the partial order. Then $g(a) \leq g(b)$ since g is a monotonic function. Additionally, since f is monotonic, then for any $x, y \in S$, $x \leq y \implies f(x) \leq f(y)$. Therefore, this means that $g(a) \leq g(b) \implies f(g(a)) \leq f(g(b))$. Therefore, $a \leq b \implies f(g(a)) \leq f(g(b))$.

5. Suppose we have a partial-order defined by the subset ($\subseteq$) relation over all sets of integers, $\mathcal{P}(\mathbb{Z})$. We define function $f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ as $f(S) = (S \cup \{1\}) \setminus \{2\}$. f is a monotonic function with respect to this partial order.

   Hint: Use your answer from the previous part. True. Consider two sets $S_1, S_2 \in \mathcal{P}(\mathbb{Z})$ such that $S_1 \subseteq S_2$. Let $g(S) = S \cup \{1\}$ and $h(S) = S \setminus \{2\}$.

   If both S_1 and S_2 contain 1, then $g(S_1) = S_1$ and $g(S_2) = S_2$, thus $g(S_1) \subseteq g(S_2)$. If neither S_1 nor 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, $g(S_2) = S_2$ and $g(S_1) = S_1 \cup \{1\}$ must still be a subset of S_2.

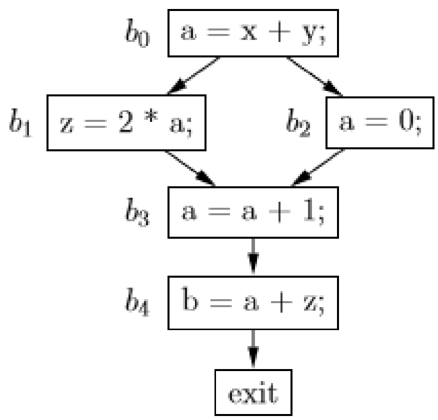
   Now if S_1 and S_2 both do not contain 2, then $h(S_1) = S_1$ and $h(S_2) = S_2$; therefore, $h(S_1) \subseteq h(S_2)$. If both S_1 and S_2 contain 2, then it still holds $S_1 \setminus \{2\} \subseteq S_2 \setminus \{2\}$. Finally, if only one of the sets contains 2, then it must be that S_2 contains 2, since $S_1 \subseteq S_2$. But this means that $h(S_1) = S_1$ (from above), and since $2 \notin S_1$, this still means that $h(S_1) = S_1 \subseteq S_2 \setminus \{2\}$. Therefore $h(S_1) \subseteq h(S_2)$.

   Using the previous part, since g and h are monotonic, it must also mean that their composition is monotonic, which means that $f = h \circ g$ is also monotonic.

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.

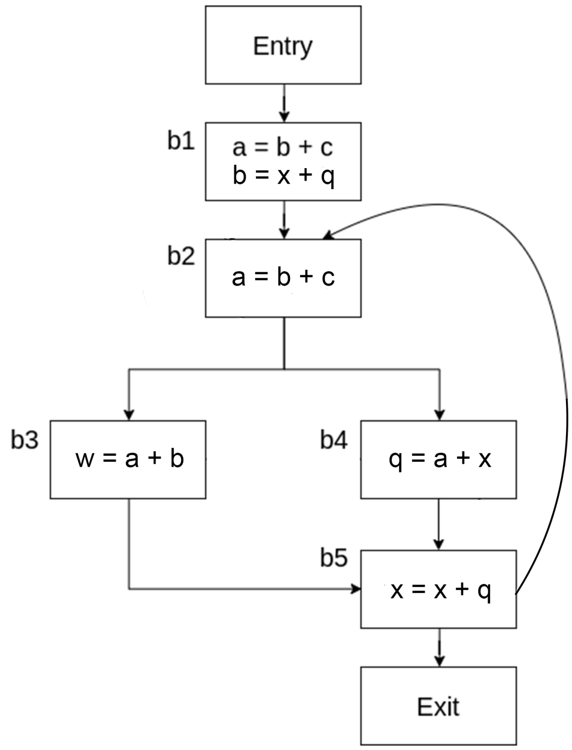


In the above example, the live range of definition $a = x + y$ is exit(b_0), entry(b_1), exit(b_1) and entry(b_3). Similarly, the live range of the definition $b = a + z$ is empty. The two live ranges do not intersect, so b can reuse a's register.

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:



out[entry]: $\emptyset$
in[b1]: $\emptyset$
out[b1]:{x + q}
in[b2]: $\emptyset$
out[b2]: {b + c}
in[b3]: {b + c}
out[b3]: {b + c, a + b}
in[b4]: {b + c}
out[b4]: {b + c, a + x}
in[b5]: {b + c}
out[b5]: {b + c}
in[exit]: {b + c}

Problem 5. Understanding 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 bigger 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. Yes. Run algorithm with the correct values and the modified 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 when the modified run reaches an FP – pretend it's still running in parallel. Then by induction, when the correct algorithm stops, the FP values it hold (MFP) will be greater than whatever value the modified algorithm holds, which will be in turn $\geq$ whatever value the modified algorithm will hold at its FP.

2. Will your algorithm give the MFP solution for all flow graphs? If so, please explain. If not, provide a counterexample. 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] will contain both d1 and d2 due to being initialized to bottom = {d1, d2}, whereas it should be just d1.

3. If your answer to 2 is no, will it give the MFP solution for some flow graphs? If it will, provide an example.

   Yes. It will give the MFP on any acyclic flow graph.

   All flow graphs where all blocks other than the entry and exit blocks have exactly one predecessor. For programs that have loops with a bounded number of iterations, one can generate a flow graph that enumerates all the possible paths through the program such that each basic block has only one predecessor. This is equivalent to simulate all the possible paths (MOP), but the flow graph is exponentially in size.

Problem 6. Detecting Errors in Interrupt Handlers

Part A. Warn on Un-elevated Uses of Privileged Instructions

Interrupt handlers are blocks of code that are run in response to events detected by a processor that need software attention. These handlers are run in a special "interrupt" execution context with elevated access to hardware, allowing them to execute privileged instructions. Consider a simplified language with the following constructs:

1. `begin_interrupt`: this enters an interrupt context.

2. `end_interrupt`: this exits an interrupt context.

3. `read_processor_flags`: this is an privileged instruction. It must be run from inside an interrupt execution context.

4. all other instructions that are not privileged instructions and do not enter or exit interrupt contexts.

You may treat each instruction as a basic block.

Your task is to warn programmers about any potential use of the privileged instruction `read_processor_flags` outside of an interrupt context. (1) Define a data flow analysis to solve this problem by filling out the table below, and (2) Specify how you use the data flow results to issue a warning on each potentially invalid use of `read_processor_flags`. Since invalid uses of this instruction can completely halt your processor, you want to warn whenever there is a potential path that could result in un-elevated uses of `read_processor_flags`.

There are other errors like nested entry of interrupt contexts (which may be erroneous depending on the hardware), or calling `end_interrupt` without a corresponding `begin_interrupt`, but you may ignore such errors for this analysis.

<table>
<tr><td>Direction of your analysis
(forward/backward)</td><td>Forwards.</td></tr>
<tr><td>Lattice elements and meaning</td><td>Boolean values. False indicates the program is outside of an interrupt context, and true indicates the program is inside of an interrupt context.</td></tr>
<tr><td>Meet operator or lattice diagram</td><td>And.</td></tr>
<tr><td>Is there a top element?
If yes, what is it?</td><td>True.</td></tr>
<tr><td>Is there a bottom element?
If yes, what is it?</td><td>False.</td></tr>
<tr><td>Transfer function of a basic block</td><td><table><tr><td><code>begin_interrupt</code></td><td>out[b] = <code>true</code></td></tr><tr><td><code>end_interrupt</code></td><td>out[b] = <code>false</code></td></tr><tr><td>other</td><td>out[b] = in[b]</td></tr></table></td></tr>
<tr><td>Boundary condition initialization</td><td>False.</td></tr>
<tr><td>Interior points initialization</td><td>True.</td></tr>
</table>

For each basic block `b` containing `read_processor_flags`, check if `in[b]` = `true`. If not, warn on this instruction.

Part B. Detecting Nested Interrupt Contexts

Now you would like to warn programmers about nested entry into interrupt contexts as this is not supported by your processor. In other words, warnings are given if there are any potential multiple `begin_interrupt`'s in a row, without intervening `end_interrupts`. (1) Define a data flow analysis to solve this problem by filling out the table below, and (2) Specify how you use the data flow results to issue a warning on each potentially extraneous `begin_interrupt` command. Since this error can also cause your processor to halt, you want to warn on any *potential* spurious `begin_interrupt`'s.

You may treat each instruction as a basic block.

Direction of your analysis (forward/backward)	Forwards.
Lattice elements and meaning	Boolean values. False indicates the program is outside of an interrupt context, and true indicates the program is inside of an interrupt context.
Meet operator or lattice diagram	Or.
Is there a top element? If yes, what is it?	False.
Is there a bottom element? If yes, what is it?	True.

Transfer function of a basic block	`begin_interrupt`: out[b] = `true`
	`end_interrupt`: out[b] = `false`
	other: out[b] = in[b]

Boundary condition initialization	False.
Interior points initialization	False.

For each basic block b containing `begin_interrupt`, check if `in[b]` = `true`. If so, warn on this instruction.