CS243 Homework 1
Winter 2018
Due: January 24, 2018

Directions:

- Complete the following problems and hand them in at the beginning of class with your name and SUID at the top.
- Remember to complete the corresponding Gradiance quizzes by the start of class on the due date. There are no late days for Gradiance.
- If you need to use one or more late days, hand in your assignment by 4:30pm Thursday or Friday in Gates 407. Slide it under the door if it is locked.
- 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. If so, 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 universe set
2. Set difference no, not commutative and not idempotent
3. Component-wise minimum for tuples of natural numbers (partial order) yes, bottom is the zero vector, top does not exist
4. Lexicographic minimum for tuples of natural numbers (total order) yes, bottom is the zero vector, top does not exist
5. Addition over complex numbers no, not idempotent
6. Arithmetic mean over real numbers no, not associative
7. AND over booleans yes, bottom is the all-false vector, top is the all-true vector
8. XOR (exclusive OR) over booleans no, not idempotent

Rubric: 1 point for each answer (0.5 correct answer, 0.5 correct explanation).

Problem 2. True or False? Briefly justify your answer in 1 to 5 lines.
1. There is at most one fixed point solution for a dataflow problem.

   False in general: there are multiple fixed points which depend on the initialization condition. E.g. initializing all program points to ⊥ is always a fixed point, but rarely the only or the useful fixed point.

   IDEAL is not a fixed point solution: indeed, any solution higher than MFP cannot be a fixed point solution, by definition.

2. There is always at least one fixed point solution for a monotone dataflow problem with finite descending chains.

   True, because the solution must decrease at every step, and it cannot be lower than bottom.

3. A semi-lattice can have at most one bottom element.

   True. Let $b_1$ and $b_3$ the candidate bottom elements. Let $b_3 = b_1 \land b_2$. By definition of meet, $b_3 \preceq b_1$; by definition of bottom, $b_1 \preceq b_3$. Therefore $b_3 = b_1$. By a similar argument, $b_3 = b_2$. Therefore $b_1 = b_2$.

4. A bounded semi-lattice (one that has top and bottom) has necessarily a finite domain.

   False. The semi-lattice over sets of natural numbers, with intersection as the meet operator, admits a top element (the set of all natural numbers) and a bottom element (the empty set), but the domain is not finite.

**Rubric**: 1 point for each answer (0.5 correct answer, 0.5 correct explanation).

**Problem 3. Live Range Analysis**

We say that a path is *definition free* with respect to a variable $y$ if there is no definition of variable $y$ along that path. The live range of a definition $d : y = x + z$ that defines variable $y$ consists of all the program points $p$ such that both of the following hold:

1. There is a path from $d$ to $p$ that is definition free with respect to $y$.

2. There is a path from $p$ to $q$ that is definition free with respect to $y$, where $q$ is a statement that uses the variable $y$.

Intuitively, the live range of a definition includes points along all subsequent paths until either it is overwritten by a new definition or the variable is no longer used along that path. The concept of live ranges can be used in register allocation: if live ranges of two definitions do not intersect, they can be assigned to the same register. Consider the following program:
In this 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 exit\((b_4)\), 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, in which case, you don’t have to describe those algorithms again, but be specific on how you compute live range.

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.

Rubric: 2 points for the reaching definitions part, and 2 points for liveness part. Other solutions graded ad-hoc. Full marks to all correct solutions.

Problem 4. Detecting uninitialized variables.

Consider the following program:

There are no other instructions in this program. If the program follows the path \( b_0, b_1, b_3, b_4 \), the variable \( z \) is used without initialization. Conversely, if the program follows the path \( b_0, b_1, b_3, b_4 \), the variable \( z \) is initialized and then used. Unlike a program that always uses an uninitialized variable, this program is not necessarily incorrect: whether \( z \) is used uninitialized depends on the conditions on those branches. For this reason, compilers that warn
about uninitialized variables usually have two options: one to warn about variables that are
definitely uninitialized, and one to warn about variables that are maybe uninitialized (i.e.
the compiler cannot prove that the variable is initialized).

In this problem, we’ll consider how to implement those warnings using the dataflow
framework.

1. What semi-lattice(s) can be used to implement those warning passes? Define the domain,
meet operator and bottom element.

This solution uses two passes, one that computes “definitely uninitialized” variables
and one that computes “definitely initialized” variables. For both passes, we use the
lattice of sets of variables. The meet operator is intersection, and the bottom element
is the empty set.

For the definitely uninitialized case, a variable is in the set at a program point if it’s
not initialized before at any program point. For the maybe uninitialized (definitely
initialized) case, a variable is in the set at a program point if it is initialized along all
paths of the program.

2. What are the boundary conditions?

All variables are uninitialized at the beginning of the program.

For the definitely uninitialized, this means OUT(entry) = U. For the maybe unini-
tialized, OUT(entry) = ∅.

3. What are the initialization values for a maximum fixed point algorithm?

The initialization value is always T for the lattice, so the universe set of all variables.

4. What is the transfer function on each instruction?

For the definitely uninitialized, an instruction that sets variable x removes x from the
set of (uninitialized) variables.

For the maybe uninitialized, the same instruction adds x to the set of (initialized)
variables.

5. Once the dataflow algorithm completes, how do we find which instruction should gen-
erate a warning?

For the definitely uninitialized case, warn if an instruction uses a variable in the set at
that program point. For the maybe uninitialized, warn if an instruction uses a variable
not in the set at that program point.

6. Would your dataflow passes warn on the program given above? If so, and assuming
that the program was indeed correct, can you think of a refined algorithm which gives
no warning?

That pass would warn that z is maybe uninitialized at point b₄, but not definitely
uninitialized. To refine the analysis for a correct program, one needs a path-sensitive
analysis that takes into account the branching conditions. If the two if-statements
branch in the same way, the incorrect path is impossible. This is what static analyzers will do in practice, and can be sound and complete for loop-free programs with no IO or memory (but NP hard). IO and memory force you to give up soundness (so some warnings will be spurious) or completeness (so some buggy programs will go undetected); loops and recursion make the problem undecidable and you usually give up both.

This answer was considered correct as long you mentioned path sensitivity, branch conditions, or equivalently enumerating all paths. Conversely, there is no dataflow-based answer to this question, because dataflow ignores the branch conditions.

**Rubric:** for each of 1 to 5, 1 point for each correct analysis (i.e. 2 if you got both analysis right, 1 if you had one and not the other). One point for the last question.

**Alternative solution for the whole problem**

Use reaching defs, but add a special \( x = \text{UNINIT} \) definition for all variables. The boundary condition is that all variables have value UNINIT at the entry block. Define the rest of the dataflow as reaching defs. If the only definition reaching a point that uses a variable is UNINIT, the value is definitely uninitialized. If any definition is UNINIT, the variable is maybe uninitialized.

Solutions that used this algorithm (or any hybrid where they used this for one pass and the variable-based one for the other) received full marks, as long as both cases were accounted in question 5.