CS4120/4121/5120/5121—Spring 2022
Homework 4
More Types and Program Analysis
Due: Monday, April 25, 11:59PM

0 Updates

1 Instructions

1.1 Partners

You may work alone or with *one* partner on this assignment. But remember that the course staff is happy to help with problems you run into. Use Piazza for questions, attend office hours, or set up meetings with any course staff member for help.

1.2 Homework structure

All problems are required of all students.

1.3 Tips

You may find the Dot and Graphviz packages helpful for drawing graphs. You can get these packages for multiple OSes from the Graphviz download page.

2 Problems

1. Preventing the billion-dollar mistake through program analysis

Accesses to null pointers are a frequent source of bugs and security vulnerabilities. To protect against these accesses, one option is to rely on hardware memory protection to prevent these accesses, but that protection is probably not available on embedded platforms. Another option, which you have already explored on the exam, is to use a more defensive type system to ensure null pointer accesses do not occur at runtime. In this problem, we explore yet another alternative. You will design a dataflow analysis that ensures memory accesses do not go to memory address zero, by conservatively computing the set of variables at each program point that may contain zero. Accesses to memory location $[x]$, where x is a variable, can then be prevented at a program point where x might be zero.

(a) What is the top element $\top$ for this dataflow analysis?

(b) Define the ordering and the meet operator for elements in this lattice (including $\top$).

(c) Give dataflow equations for this analysis for each of the possible kinds of IR nodes. Recall that we had five IR node types: $x \leftarrow e$, $[e_1] \leftarrow e_2$, `if` e, `start`, `return` e. For simplicity, we will use a simpler syntax in which expressions can only occur as right-hand side of an assignment to a variable: $x \leftarrow e$, `[`x_1`]` $\leftarrow x_2$, `if` x, `start`, `return` x where e can only take the forms n (constant), x, $x_1 + x_2$, and $[x]$.

Also, note that this is an analysis where, as with conditional constant propagation, it is sometimes helpful to propagate different information along different exiting edges from an `if` node.

2. Defending against zombies with dataflow analysis

Let us define "undead" code as code that depends on a variable that is *always* uninitialized. When such undead code is removed, additional program regions may become undead due to the disappearance of variable declarations. The goal of this exercise is to remove all undead code from a function using only a single analysis pass. No variables will be assumed to be live-in at the start of the CFG.

(a) Design a dataflow analysis that can be used for cascading undead-code removal. Describe its ordering, the meet operator, the top element, as well as the flow function. Where necessary, be conservative. You only need to specify the flow function for assignments $x = expr$.

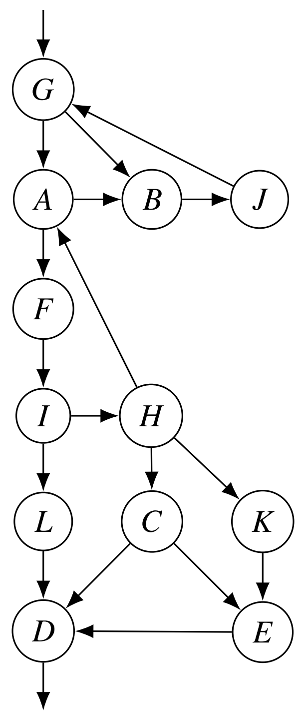
(b) Show that the flow functions you defined are monotonic, and either show that they are distributive or construct a counterexample.

(c) Show that one run of your analysis leads to the removal of the following grayed-out undead code (remember that meets are used at merge points in the CFG):

```
a = 1
if (f(a) > 0) {
  c = c+1
  d = 5
}
[d] = a+c
g = a+d
```

3. Control-flow analysis

For the control-flow graph below, give the dominator tree, with back edges added as dashed edges. Identify the loops and the control tree, and for each loop indicate its set of nodes, its header node, and its exit edges.



4. Handling Potentially Irreducible Control Flow

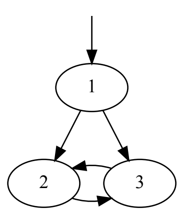
So far in the course, we have been working primarily with *natural loops*. Recall that formally defined, a natural loop is the smallest set of nodes which includes the head and tail of a back edge, and which has no predecessors outside the set other than the predecessors of the header. Put another way, we may construct a natural loop about a back edge as long as the loop header dominates all loop exits.

If a program contains only natural loops, it is said to have *reducible* control flow. A simple way to ensure that a program's control flow is reducible is to remove every back edge (edges $A \mapsto B$ such that B dominates A), and check that what remains is a directed acyclic graph with every node reachable from the entry point. A program that violates this condition is said to have *irreducible* control flow.

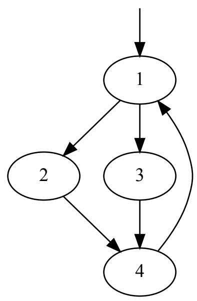
Most programming languages are designed so that they only produce reducible control flow. Standard programming constructs such as `for` and `while` loops combined with `if-else` branching and even `break` and `continue` statements always keep the control flow reducible. This ensures that all of our loop-based program analyses and transformations, which work correctly only on reducible control flow, are safe and viable program optimizations. However, older languages such as C still support arbitrary control flow by means of `goto` statements. Likewise, some domain-specific languages such as P4 include syntax for defining finite state machines (FSMs) which may cause irreducible control flow in the general case. In these settings, an optimizing compiler wishing to do loop optimizations must be able to check for irreducible control flow.

For each of the following example CFGs, indicate whether the control flow is reducible or irreducible. If it is reducible, give a brief explanation and list each set of nodes forming a natural loop. If it is irreducible, name a specific edge in the CFG and explain why the edge you've named causes the control flow not to be reducible.

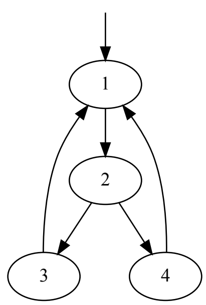
(a)



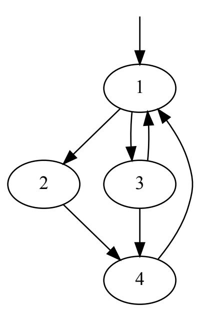
(b)



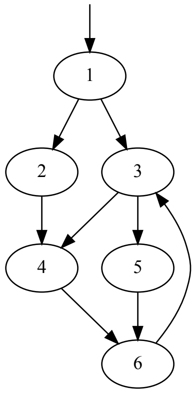
(c)



(d)

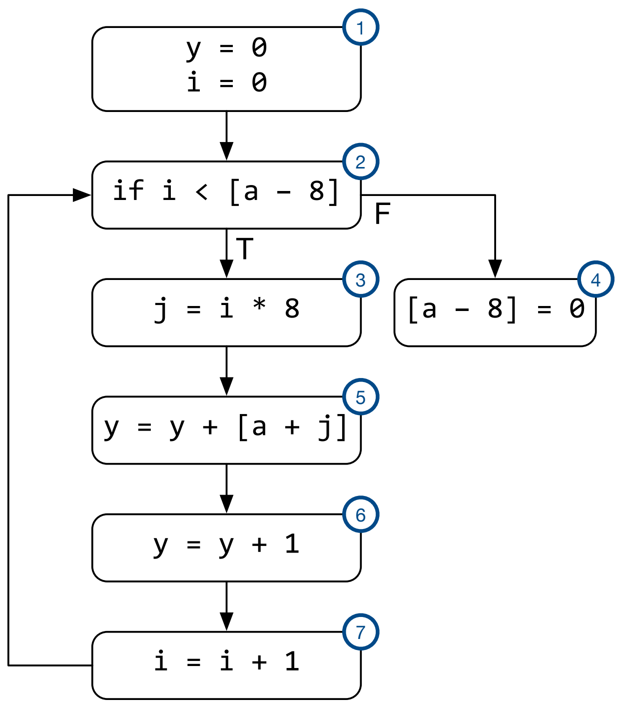


(e)



5. Induction variables

Consider the following loop:



a) Identify all induction variables in this code and whether they are basic, linear, or derived (or some combination). Do not include any loop-invariant variables.

b) Identify all loop-invariant expressions suitable for hoisting.

c) Perform loop unrolling on the loop, with an unrolling factor of 2. Show the resulting CFG.

3 Submission

Submit your solution as a PDF file on CMS. This file should contain your name, your NetID, all known issues you have with your solution, and the names of anyone with whom you have discussed the homework.