CS4120/4121/5120/5121—Spring 2020
Homework 4
More types and program Analysis
Due: Monday, April 27, 11:59PM

0 Updates

- None yet; watch this space.

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 types

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 not always available on embedded platforms, and in any case programs crash when they attempt to dereference null. Tony Hoare called the addition of a `null` value to the Algol language his "billion-dollar mistake", for the damage it has caused.

An excellent alternative to null values is to have a "maybe" or "option" type constructor in the language, as in OCaml, Haskell, and Kotlin. The type `maybe` t, where t is some type, either represents a value of type t or the absence of any value. Maybe types, which date back to the programming language CLU, are a more principled way of handling the absence of a value, because the type system itself helps the programmer remember to test whether a value is present. There are many presentations of the idea of maybe types; let's figure out typing rules for one of them. The following terms should be supported by the language:

- `none` should have the type `maybe` t for any type t.
- Any expression of type t should be usable as an expression of type `maybe` t.
- (OCaml-like) The expression `match` e `with some` $x \to e_1$ | `none` $\to$ e_2 evaluates e to a value of type `maybe` t and, if the value is not `none`, evaluates e_1 with variable x bound to the value of type t. Otherwise, it evaluates e_2. The result of the `match` expression is the value of whichever of e_1 or e_2 is evaluated.
- (Kotlin-like) A function call $e_1(e_2)$ is permitted where e_2 evaluates to a `maybe` t, but e_1 evaluates to a function expecting a t and returning a t'. If the argument is `none`, the function is not called, and the result of the expression is `none`. Otherwise the result of the expression is the result of the function, viewed as a maybe.

Write typing rules that describe formally how to type-check these expressions. Also give a safe but permissive subtyping rule for relating two `maybe` types.

2. 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. In this problem, 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 `=` e, `[`e_1`]` `=` 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 `=` e, `[`x_1`]` `=` 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.

3. 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
```

4. Control-flow analysis

For the control-flow graph in Figure 1, 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.

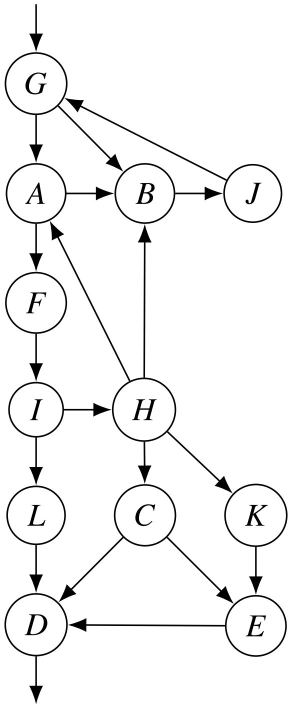
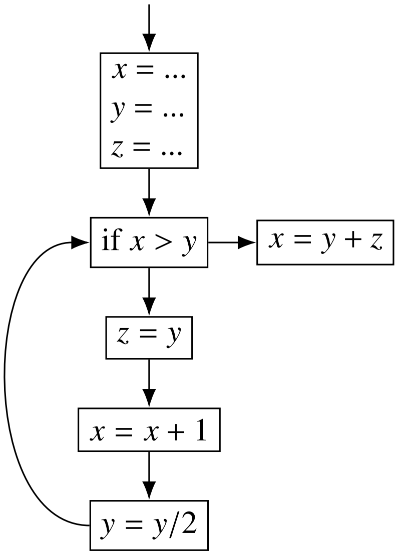


Figure 1: The control-flow graph for Problem 4

5. Single Static Assignment

Consider the following control-flow graph:



Convert to SSA using the least number of φ functions possible and draw the result.

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.