CS243 Review Session 1 Notes

1 Taint Analysis

Data from untrusted sources could cause security vulnerability in programs. For example, in C, a hacker can potentially take over the control of a program by passing well crafted strings into a format function (e.g. `printf()`).

Consider a simplified language whose variables can only take string values and has the following kinds of statements:

```
ASSIGN: a = "constant string"
COPY:   a = b
CONCAT: a = b+c
INPUT:  a = read()
PRINT:  print(a)
```

A variable is tainted if it is defined by an INPUT statement, or a CONCAT or COPY of other tainted variables. This problem has two parts:

- Use data flow analysis to issue warnings whenever a PRINT statement may write out a tainted variable.
- Indicate for each warning the INPUT statements that may be responsible for the warning.

For example, for the following code:

```
L1: a = read ()
L2: c = read ()
L3: if (a)
L4:  a = c
L5: b = a + "str"
L6: print (c)
L7: print (b)
```

Warnings should be issued on both L6 and L7.

- L6 warning: input statements that may be responsible: {L2}
- L7 warning: input statements that may be responsible: {L1, L2}

You can design one or more data flow analyses to answer the two parts of this problem.

2 More on the MFP

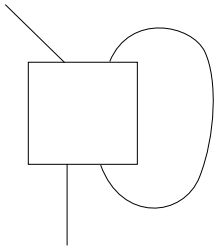
In class we observed that IDEAL $\geq$ MOP $\geq$ MFP $\geq$ FP, where MOP is meet over paths, MFP is maximal fixed point, and FP is any fixed point of the dataflow equations. We said that any solution that is less than the IDEAL solution is still "safe", but not as precise as we may want.

Let's take a deeper look at what *safe* means. Consider liveness analysis. Liveness is used to know when we can stop keeping track of a variable because it will no longer be used. We can use this for register allocation, because if a variable is no longer needed (i.e. not live) then we can reuse its register to store another value. Remember from lecture that liveness has a top element of $\emptyset$. Thus, smaller element in the lattice are actually larger sets! If we chose a smaller set in the lattice then we'll get all the variables we had before, but add in some extra ones. These extra variables could hurt optimization: in this case they might force us to unnecessarily spill registers to the stack because we mistakenly believe we'll use them again.

This is true of many analyzes we'll look at in this class, because basically all interesting properties of a program are undecidable. (See Rice's Theorem.) However, by using the dataflow framework, we can be assured our solution is always a safe, if perhaps imprecise, answer.

2.1 MOP versus MFP

One thing we have emphasized is the dataflow equations set up a mathematical problem that we then use the iterative algorithm to solve. Remember that the dataflow equations don't necessarily give us the MOP solution, but we never wrote out what the MOP solution was. Consider this graph:



The dataflow equations are:
IN = INIT $\wedge$ OUT
OUT = f(IN)
Or equivalently:
IN = INIT $\wedge$ f(IN)
If we let F(X) = INIT $\wedge$ f(X), then we see that IN = F(IN), i.e. the solution to the dataflow equations is the fixpoint of a particular function F (which depends on the graph and transfer functions).

However, the MOP solution for IN is different:

$$\text{IN} = \text{INIT} \wedge f(\text{INIT}) \wedge f(f(\text{INIT})) \wedge ... = \bigwedge_{i=0}^{\infty} f^n(\text{INIT})$$

These two definitions are equivalent if the framework is distributive. The proof of this fact is too complicated, so we'll look at something else.

2.2 MFP versus FP

We're going to take a better look at the solutions to the dataflow equations. The tuple of all basic block entry points admits a product semi-lattice, which induces a partial order on solutions to the entire graph. Our problem creates a function F this set of tuples to itself which is composed of transfer functions and meet operators. (Assume forward analysis.) We saw such a function for the trivial graph in the previous part. It turns out this function is monotone, just like the transfer functions.

Any fixed point of F is a solution to the dataflow equations. But remember we are in a semi-lattice, which has a partial order rather than a total order. What is the relation between all the different fixed points? Is there always a unique maximal fixed point that is greater than all others?

Consider the sequence:

$$\top,\ F(\top),\ F(F(\top)),\ F^3(\top),\ ...$$

We know this sequence is descending and what the iterative algorithm computes. Further, we know that at some point it will converge to a fixed point of F. Consider any other fixed point of F, $d = F(d)$. Then $\top \geq d$, by definition of $\top$. Thus $F(\top) \geq F(d)$ by monotonicity. But d is a fixed point so $F(\top) \geq F(d) = d$. Thus $F(\top) \geq d$. If we apply F any number of times, by induction d is always less than $F^n(\top)$. Notice that if we initialize our iterative algorithm with something other than $\top$ we don't get this result because the induction doesn't have a base case!

This tells us that not only does the iterative algorithm arrive at a fixed point, it finds a fixed point that is larger than all other fixed points. Thus we can in fact say that the maximal fixed point is both well defined and unique.

Aside Remember f monotone does not imply $x \leq f(x)$. Can you construct a counterexample?