Static Analysis

Announcements

- No class Wednesday 4/30
- No OH this week
- Lab today

Overview

- Reachability analysis review
- Detecting numerical properties via static analysis
- Course takeaways

Dataflow Analysis

- Use cases:
  - Deadcode identification and elimination
  - Compiler optimizations:
    - Constant propagation
    - Common subexpression elimination
  - Taint analysis

- How does it work?
  - At every program point, we track properties about variables
  - Each point has and IN and OUT set
    - IN: facts that are true at the entry of the point
    - OUT: factors that are true at the exit of the point

Dataflow Analysis

Dataflow Analysis applications

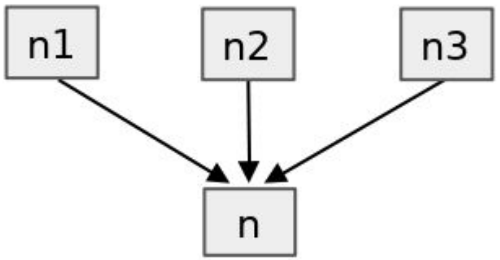
- Compilers and IDEs use data flow analysis to perform optimizations
    - Deadcode eliminations
    - Constant propagation
        - If we know x is always equal to 5 at if (x == 5) we can optimize away the check
    - Common subexpression elimination
        - Reuses previously computed statements

- Taint analysis
    - Tracks whether untrusted or sensitive input can reach sensitive operations
    - SQL injections
    - Log4J bug

Reachability Analysis

- For any dataflow analysis, we must define *transfer functions* for how IN and OUT change

$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$



$$IN[n] = OUT[n1] \cup OUT[n2] \cup OUT[n3]$$

Reachability Analysis

- For any dataflow analysis, we must define *transfer functions* for how IN and OUT change

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

n: $\boxed{b\ ?}$

$$GEN[n] = \emptyset$$
$$KILL[n] = \emptyset$$

n: $\boxed{x = a}$

$$GEN[n] = \{ <x, n> \}$$
$$KILL[n] = \{ <x, m> : m \mathrel{!=} n \}$$

Worklist Algorithm

for (each node n):

$$IN[n] = OUT[n] = \varnothing$$

repeat:

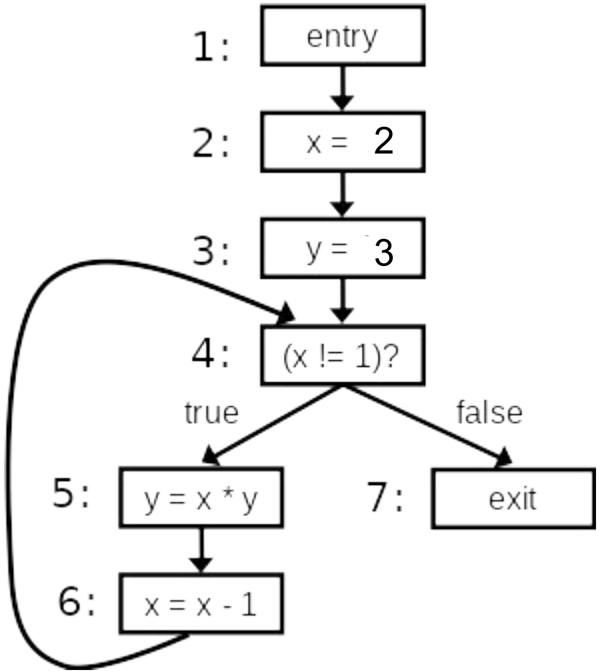
for (each node n):

$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

until IN[n] and OUT[n] stop changing for all n

Worklist Algorithm



Iter 1:

IN[1] = ∅, OUT[1] = ∅

IN[2] = ∅, OUT[2] = {x → 2}

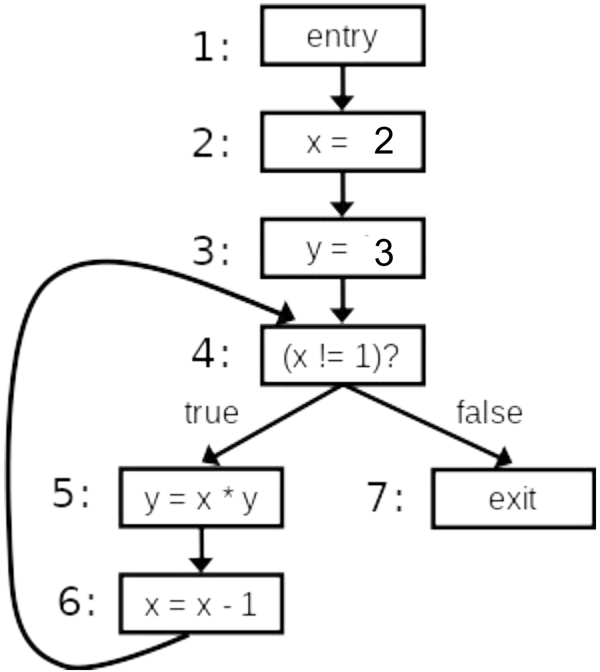
IN[3] = {x → 2}, OUT[3] = {x → 2, y → 3}

IN[4] = {x → 2, y → 3}, OUT[4] = {x → 2, y → 3}

IN[5] = {x → 2, y → 3}, OUT[5] = {x → 2, y → 6}

IN[6] = {x → 2, y → 6}, OUT[6] = {x → 1, y → 6}

Worklist Algorithm



Iter 2:

IN[1] = ∅, OUT[1] = ∅

IN[2] = ∅, OUT[2] = {x → 2}

IN[3] = {x → 2}, OUT[3] = {x → 2, y → 3}

IN[4] = [{x → 2, y → 3}, {x → 1, y → 6}]

OUT[4] = [{x → 2, y → 3}, {x → 1, y → 6}]

IN[5] = {x → 2, y → 3}, OUT[5] = {x → 2, y → 6}

IN[6] = {x → 2, y → 6}, OUT[6] = {x → 1, y → 6}

Detecting Numeric Properties

Static Analysis - Numeric Properties

```
//x is a parameter

  if (x == 0) {            // p1
     x++;                  // p2

  } else if (x > 0) {      // p3

     x = x * 20;           // p4

  } else if (x < 0) {      // p5

     x = x * (-10);        // p6

  }
  assert(x > 0);           // p7
```

How could we analyze if the following program satisfies the assertion?

1. Testing - execute the program with random values of x and check the value of x at p7
2. SymExc - collect path constraints and check if `x <= 0` at p7 is SAT
3. Verification - deductive reasoning over all paths and statements
4. Static analysis - track “facts” about values as they flow through the program

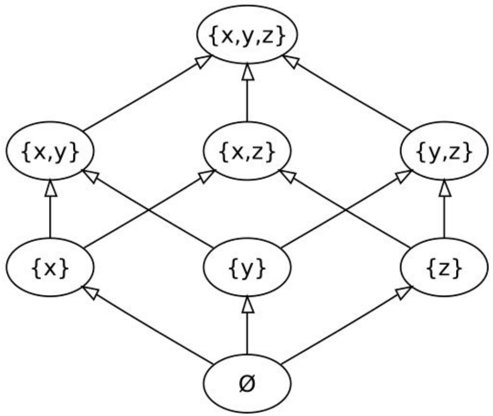
Abstraction

- Most interesting properties of programs are undecidable (halting problem), and even those that are not may be very expensive to compute.

- static analysis usually involves some kind of *abstraction*

- For example, instead of keeping track of all of the values that a variable may have at each point in a program, we might only keep track of whether a variable's value is positive, negative, zero, or unknown

Lattices



- A *lattice* is a mathematical structure
- Partially ordered set
  - For certain pairs of elements one proceeds the other
  - Not every pair of elements needs to be comparable

- Each pair of elements has a Least Upper Bound (LUB)
  - Smallest element that is greater than both of them

Lattices:

```
    ⊤
   / \
odd   even
   \ /
    ⊥
```

```
   {a,b}
  /     \
{a}     {b}
  \     /
     ∅
```

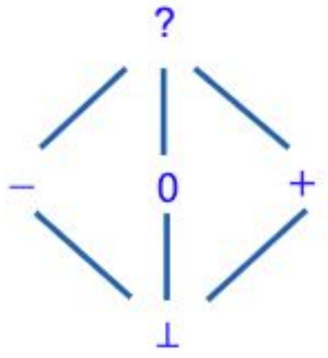
- Two example lattices:
  - One for even / odd
  - One for sets of values

- T (top) = { odd, even} (could be either)
- ⊥ (bottom) = unknown, uninitialized

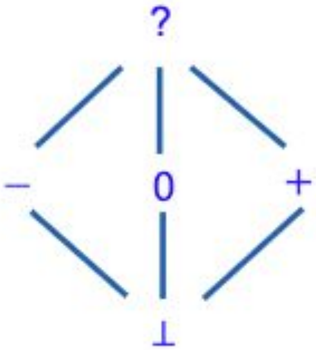
- An **edge** from a lower element X to a higher element Y indicates that X is *at least as precise as* Y
- Partial orders are transitive and anti-symmetric
  - Transitive: e.g., '⊥' is more precise than '?'
  - two different elements cannot be more precise than each other

An Abstract Domain for Signs

- Zero (0), representing the integer value 0;
- Minus (-), representing any negative integer value;
- Plus (+), representing any positive integer value;
- Top (?), representing any integer value; and
- Bottom (⊥), representing no integer value.



An **Abstract Domain** for Signs




```
//x is a parameter
  if (x == 0) {              // p1
     x++;                     // p2
  } else if (x > 0) {        // p3
     x = x * 20;             // p4
  } else if (x < 0) {        // p5
     x = x * (-10);          // p6
  }
  assert(x > 0);
```

Let's track the flow of x while representing it as a member of the abstract domain

X starts as ⊥
How does each program statement affect the sign?

We need to define this!

An Abstract Semantics for Signs

In the same way concrete semantics are defined for each java statement, we must define how different program constructs affect members of our abstract domain

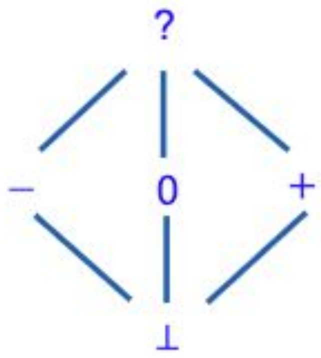
ADD	-	0	+	?
-	-			
0				
+				
?				

MULT	-	0	+	?
-				
0				
+				
?				

An **Abstract Domain** for Signs

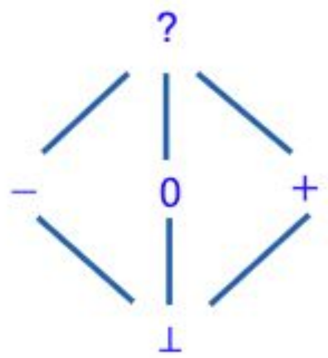
```
//x is a parameter

  if (x == 0) {          // p1
     x++;                 // p2

  } else if (x > 0) {    // p3

     x = x * 20;         // p4

  } else if (x < 0) {    // p5

     x = x * (-10);      // p6

  }
  assert(x > 0);
```

Now let’s track the flow of x while representing it as a member of the abstract domain using our abstract semantics



Example 2

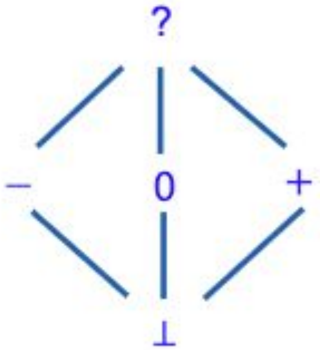
```
if (x == 0) {       // p1
    x++;            // p2
} else if (x > 0) { // p3
    x = x * (-10);  // p4
}
assert(x != 0);
```



False Positive!

Example 2

```
if (x == 0) {       // p1
   x++;             // p2
} else if (x > 0) { // p3
   x = x * (-10);   // p4
}
assert(x != 0);
```



How could we make this domain *more precise* to avoid a FP?

False Positives

- Static analysis
  - Sound, but may have FPs
  - Ratio of acceptable FPs is 1:3

- Can testing have FPs?
  - Argument for **NO:** I have a crashing input. I can run it and show you that it crashes
  - Argument for **YES:** `createNumber(“agh/’}}31a”)` crashes but I don’t care because it’s not expected to be called like that!
    - “Precondition violations” are a problem in testing!

What makes a good program analysis?

- Soundness:
  - If there is a bug, it will report it
  - If the tool says SAFE for some property, the program will be safe
  - (only as good as the property)

- Completeness:
  - If it reports a bug, the bug exists

	Complete	Incomplete
Sound		
Unsound		

Where do the techniques we've seen fall in this chart?

Course Summary

Tools and techniques we learned

Techniques:

Tools:

Takeaways

1. Program analysis takeaways
    a. What should I do if I want to answer a question about code?
    b. Emerging technology in other CS areas requires constantly rethinking PL and SE foundation
        i. We used java as a target in this course, but every decade there is a new hot thing in CS. We'll always want to answer questions about this !

Takeaways

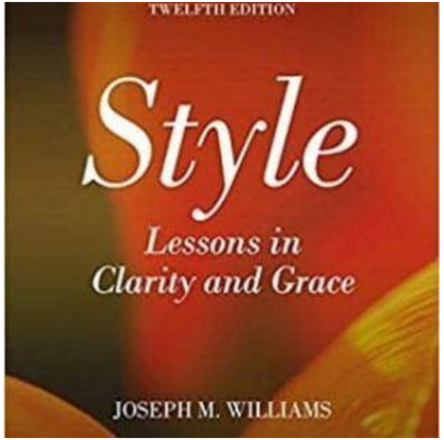
2. General CS takeaways
   a. When I’m solving a problem how can I approach it ?
      i. What do I care about? Speed? Correctness?

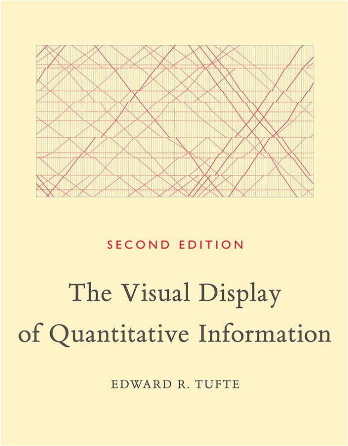
   b. We have a huge toolbox of ways to solve things
      i. Greedy, dynamic programming, all the algorithms you learned in 340
      ii. Techniques based on statistics
         1. LLMS and other ML techniques
      iii. Information retrieval!
      iv. Randomized and evolutionary algorithms
      v. Formal techniques based on logic and deductive reasoning

Takeaways

3. Technical writing

   a. Stop and think about the kernel of what you want to say and how you can clearly and concisely communicate that
   b. https://www.clc.hcmus.edu.vn/wp-content/uploads/2015/11/Style_-_Joseph_M._Williams_Joseph_Bizup.pdf





Takeaways

4. LLMS

a. They're cool (fast, NL interface, can pick up on hints like var names etc) but have no guarantees
b. energy usage

Model Size (Parameters)	Computational Resources	Training Duration (Hours)	Infrastructure	Training Energy (MWh)	Evaluation Energy (MWh)
7B	8 GPUs (NVIDIA V100)	336	Cloud (Efficient)	50	5
40B	64 GPUs (NVIDIA V100)	672	Cloud (Efficient)	200	10
100B+	1024 GPUs (NVIDIA A100)	1344	Cloud (Standard)	1,287	50

Thank you!

Have a great summer and good luck in finals!