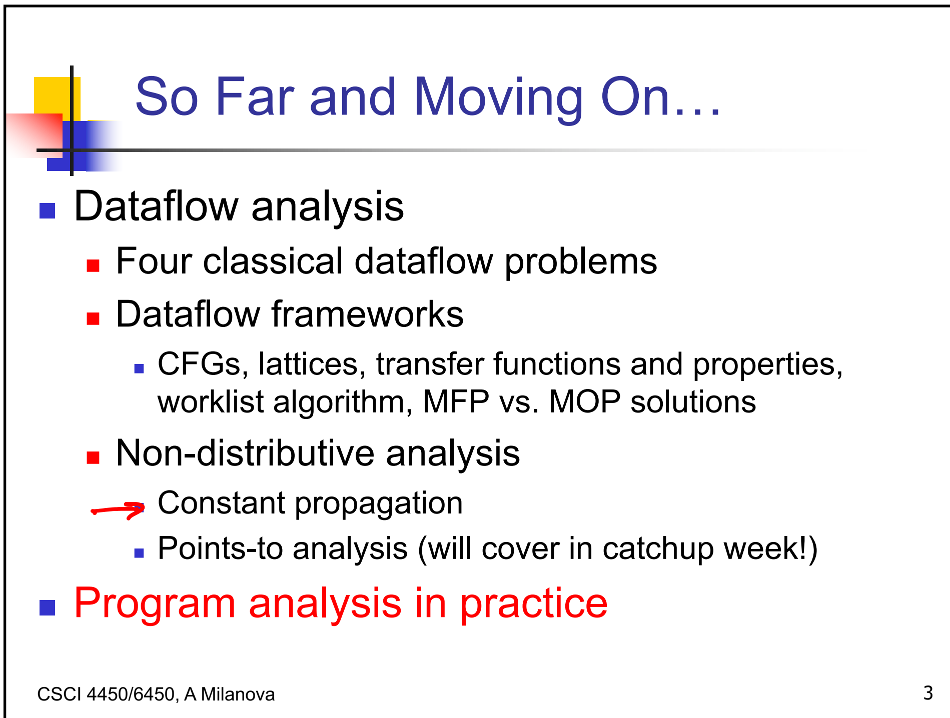
Dataflow Analysis in Practice: Program Analysis Frameworks, Analysis Scope and Approximation

1

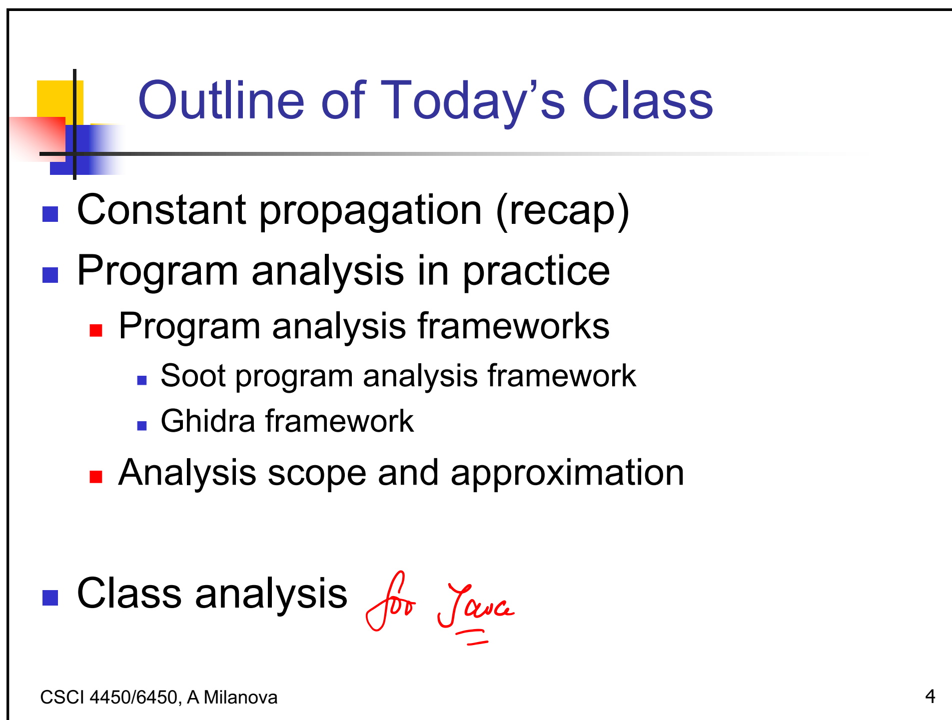
Announcements

- HW1 due today
- HW2 posted
  - Your task is to set the infrastructure locally
  - Start as soon as you can
  - Ask questions on Submitty, in class, in office hours
  - Run Soot on toy programs and study Jimple IR

2

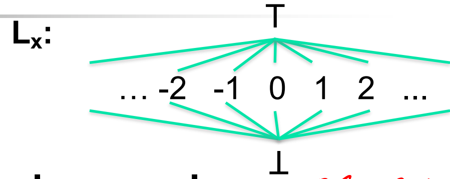


3



4

Constant Propagation fits into Monotone Dataflow Framework



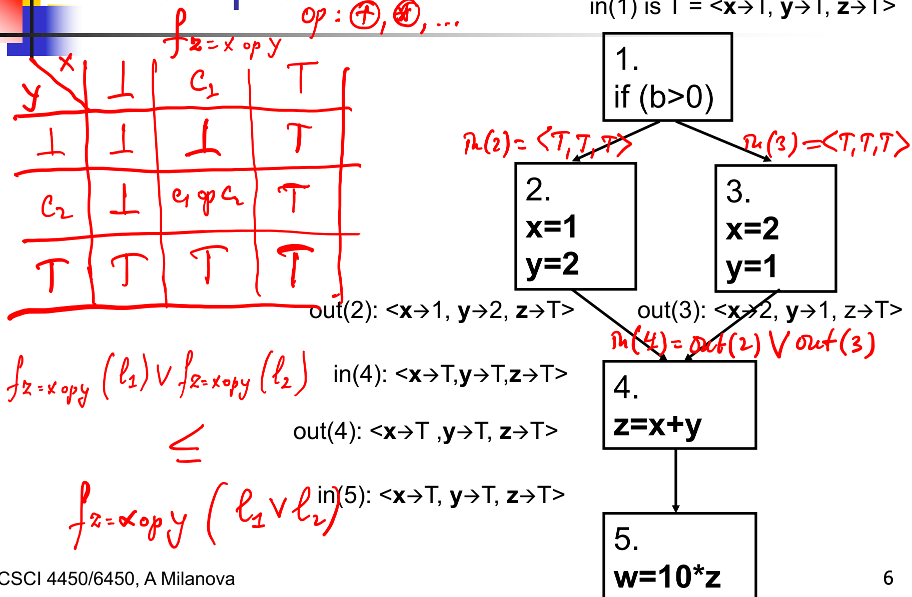
- Property space
  - Product lattice **L = L_x x L_y x … x L_z**
  - **L satisfies the ACC**

  and

0≤1 ? NO
1≤T ? YES

- Function space **F: L→ L** is monotone
- Thus, analysis fits into the monotone dataflow framework and can be solved using the worklist algorithm

Example

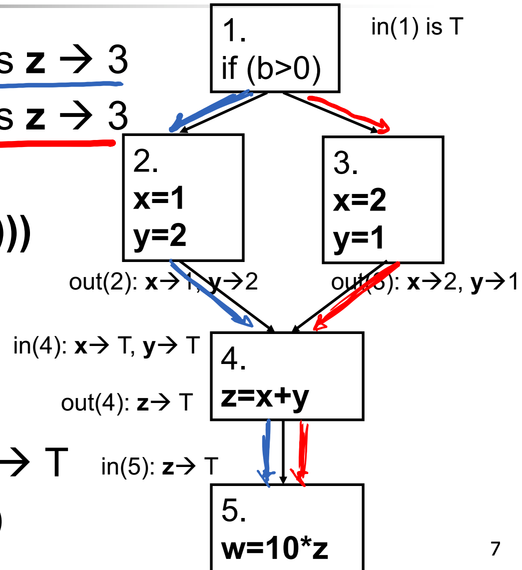


Constant Propagation is Monotone but Not Distributive!

- $f_4(f_2(f_1(T)))$ computes **z** → 3
- $f_4(f_3(f_1(T)))$ computes **z** → 3
- Thus, MOP at 5

$f_4(f_2(f_1(T))) \vee f_4(f_3(f_1(T)))$

computes **z** → 3



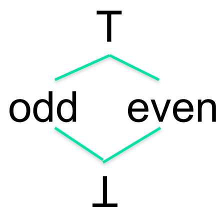
MFP at 5 computes **z** → T

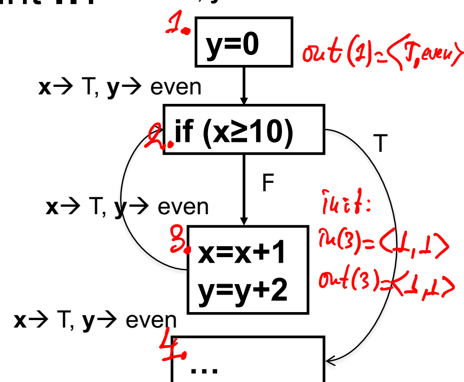
(i.e., **z** is NOT a const)

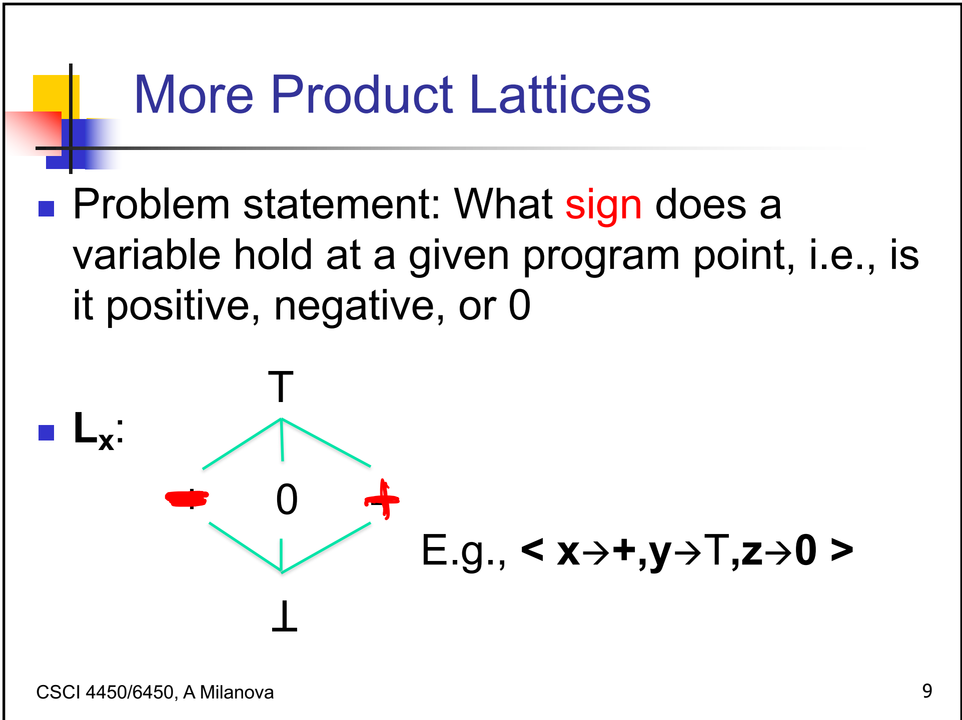
More Product Lattices

- Problem statement: Is integer variable **x** odd or even at program point **n**?

- L_x:







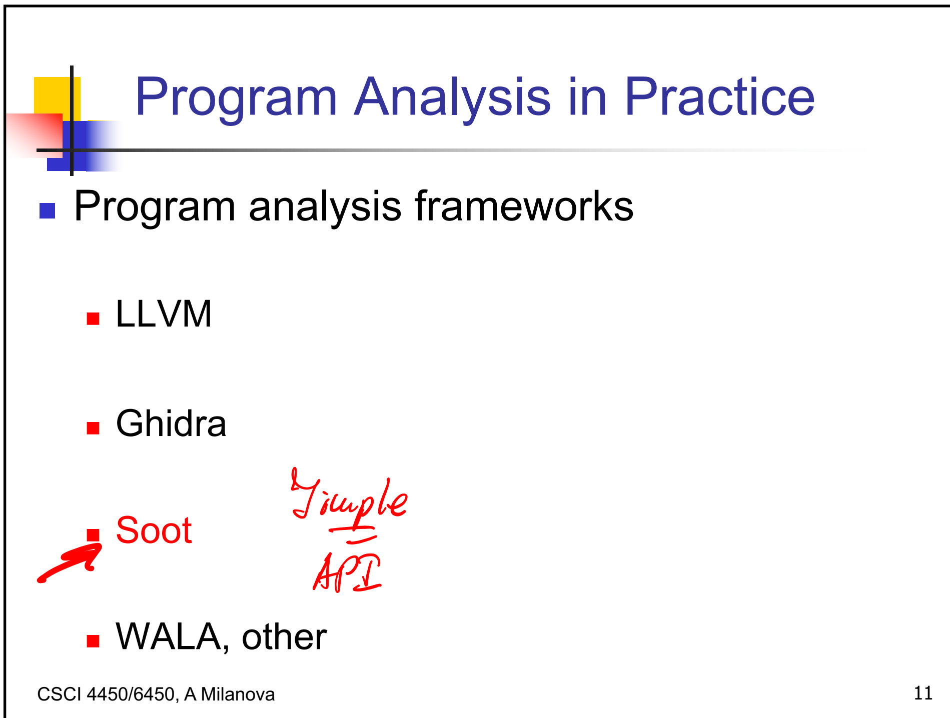
9

So far and moving on

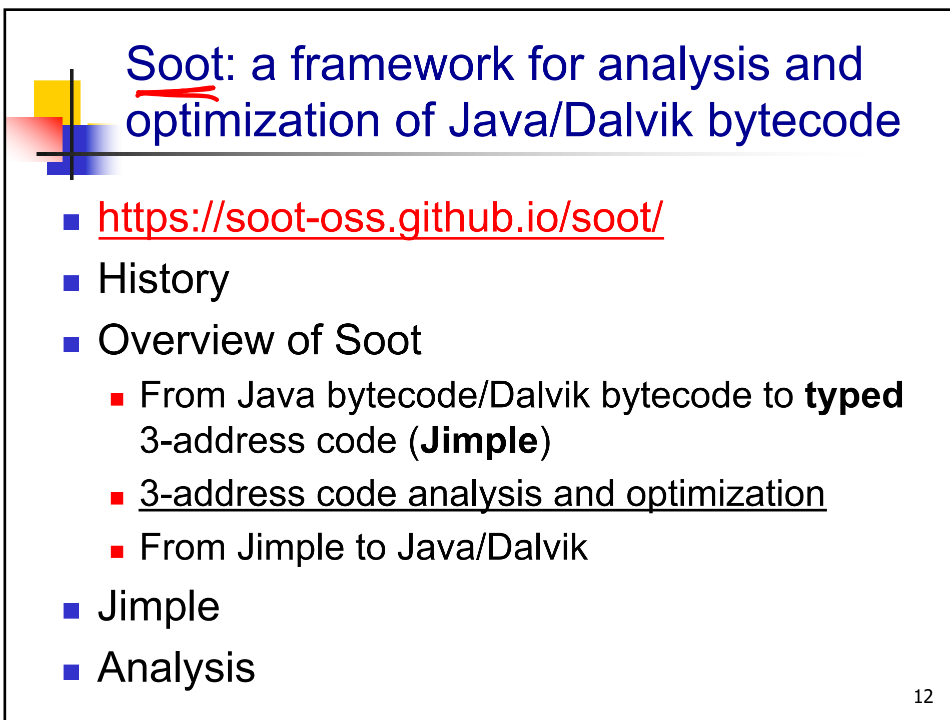
- **Intra**procedural dataflow analysis
  - CFGs, lattices, transfer functions, worklist algorithm, etc.
  - Classical analyses
- **Inter**procedural analysis
- Analysis scope and approximation

CSCI 4450/6450, A Milanova 10

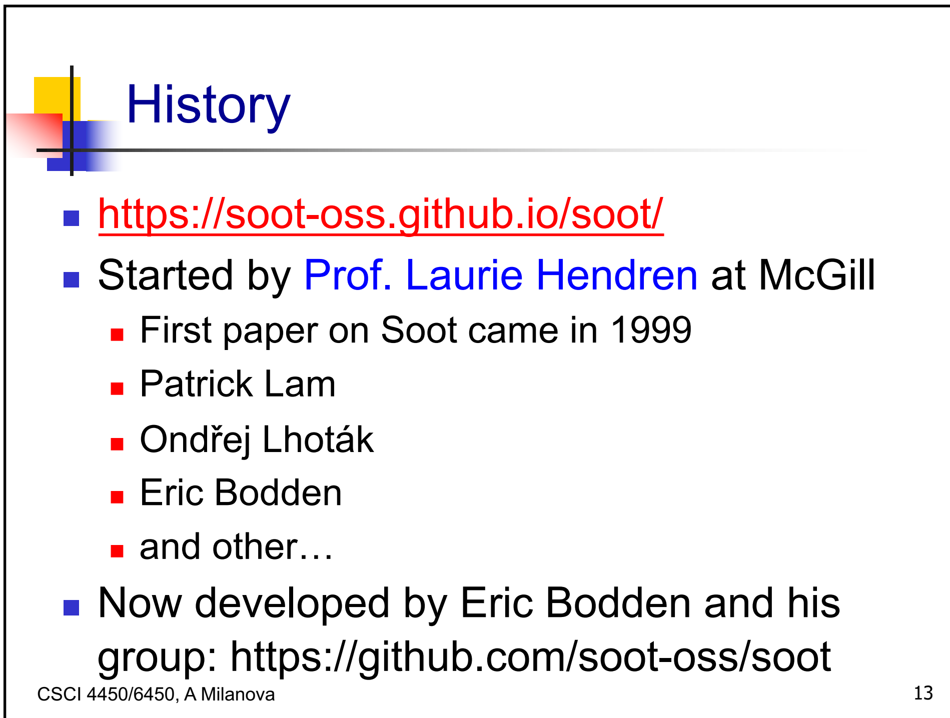
10



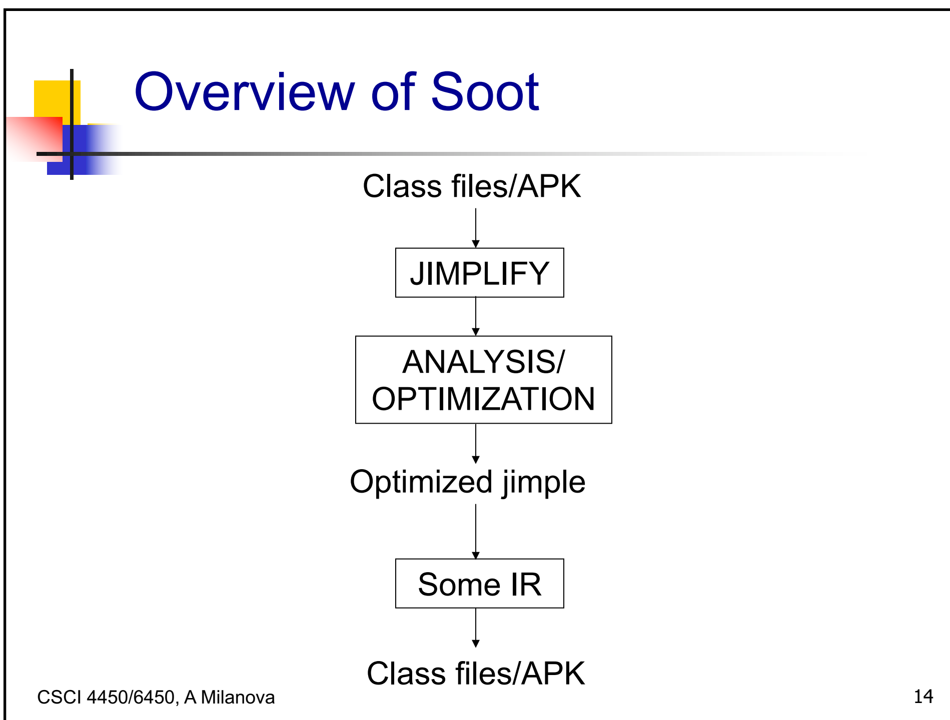
11



12



13



14

Advantages of Jimple and Soot

- Jimple
  - Typed local variables
  - 16 simple 3-address statements (1 operator per statement). Bridges gap from analysis abstraction to analysis implementation
- Soot provides
  - Itraprocedural dataflow analysis framework
  - Points-to analysis for Java
  - IR from Dalvik and taint analysis
  - Other analyses and optimizations

15

15

Jimple

⇒ .java —Soot→ Jimple
.class —Soot→

- Run soot: java soot.Main –jimple A (need paths)

Java:

```
public class A {
   main(String[] args) {
      A a = new A();
      a.m();
   }
   public void m() {
   }
}
```

Jimple:

```
public class A extends java.lang.Object
  {
   public void <init>() {
     A r0;   <- ref. variable
     r0 := @this: A;
     specialinvoke r0.
        <java.lang.Object: void <init>()>();
     return;
   }
   …
   (continues on next slide…)
```

CSCI 4450/6450, A Milanova

16

16

Jimple

Java:

```
public class A {
   main(String[] args) {
      A a = new A();
      a.m();
   }
   public void m() {
   }
}
```

Jimple:

```
…
public void m() {
      A r0;
      r0 := @this: A;
      return;
   }
…
```

instance method — *typed reference variable*

Jimple

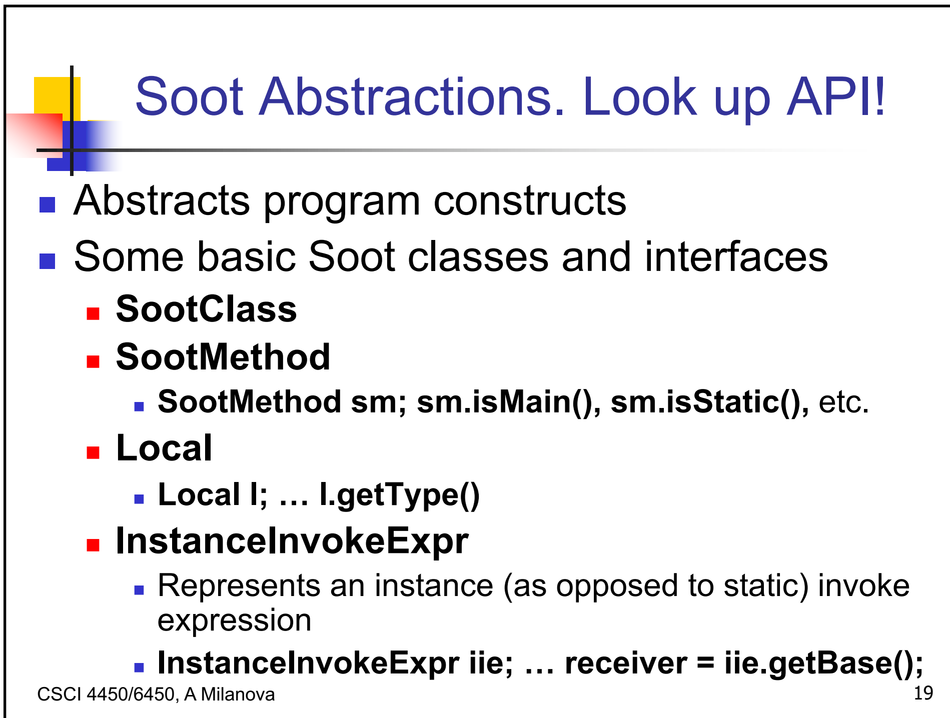
Java:

```
public class A {
   main(String[] args) {
      A a = new A();
      a.m();
   }
   public void m() {
   }
}
```

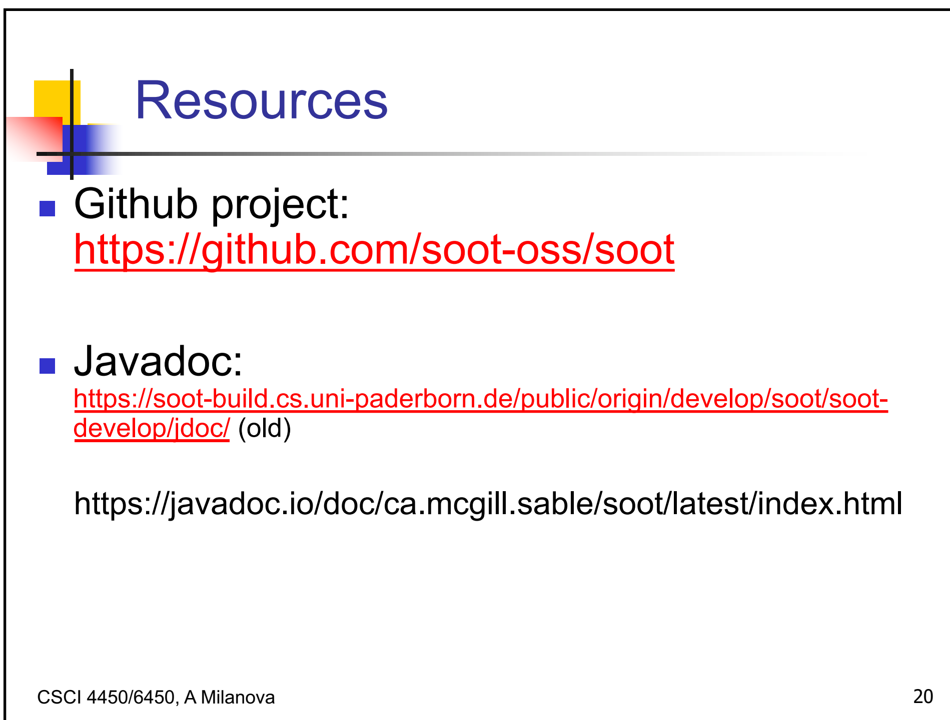
Jimple:

```
…
main(java.lang.String[]) {
    java.lang.String[] r0;
    A $r1, r2;
    r0 := @parameter0: java.lang.String[];
    $r1 = new A;
    specialinvoke $r1.<A: void <init>()>();
    r2 = $r1;
    virtualinvoke r2.<A: void m()>();
    return;
 }
}
```

ALLOCATION STMT — *CALL STMT* — *static target*



19



20

4 Kinds of Calls[1]

- Constructor/Super Call:

A a = new A(); → $r1 = new A; specialinvoke $r1.<A: void <init>()>();

- Virtual Call:

a.m(); → virtualinvoke r2.<A: void m()>();

- Static Call:

sm(); → staticinvoke <A: void sm()>();

- Interface Call:

x.m(); → interfaceinvoke r0.<pack2.X: void m()>();

1. We should not need to worry about dynamicInvoke. (Soot does support it.)

Outline of Today's Class

- Program analysis in practice
  - Program analysis frameworks
    - Soot program analysis framework
    - Ghidra framework
  - Analysis scope and approximation
- Overview of class analysis framework (HW2)
- Class analysis

Analysis Scope

- **Intra**procedural analysis
  - Scope is the CFG of a single subroutine
  - Assumes no call and returns in routine, or models calls and returns
  - What we did so far
- **Inter**procedural analysis
  - Scope of analysis is the ICFG (**Inter**procedural CFG), which models flow of control between routines

CSCI 4450/6450, A Milanova 23

23

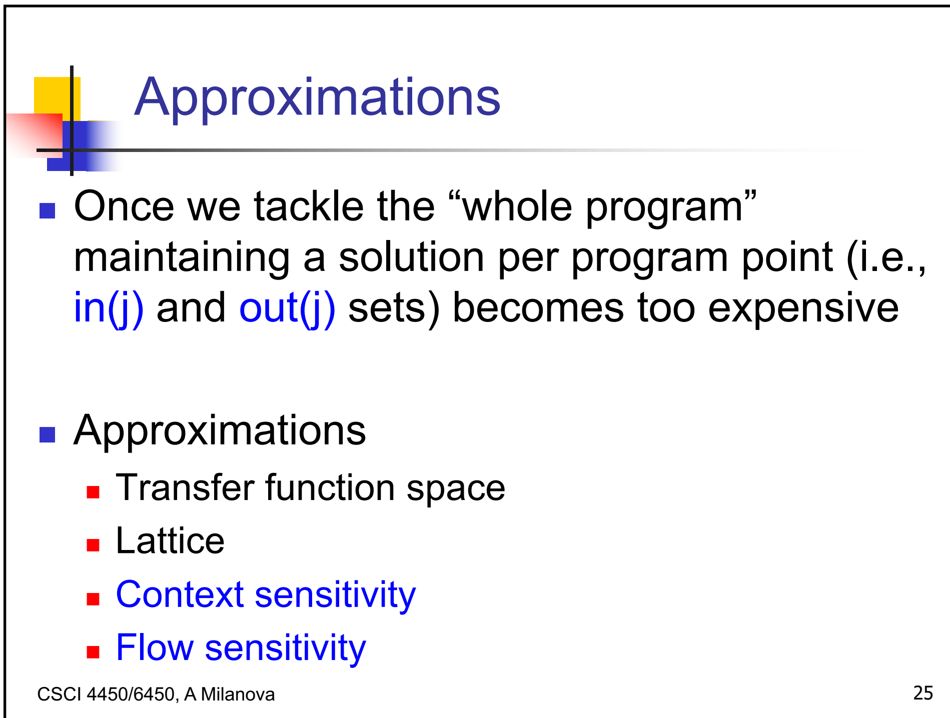
Analysis Scope

E.g. App → Android SDK → JDK

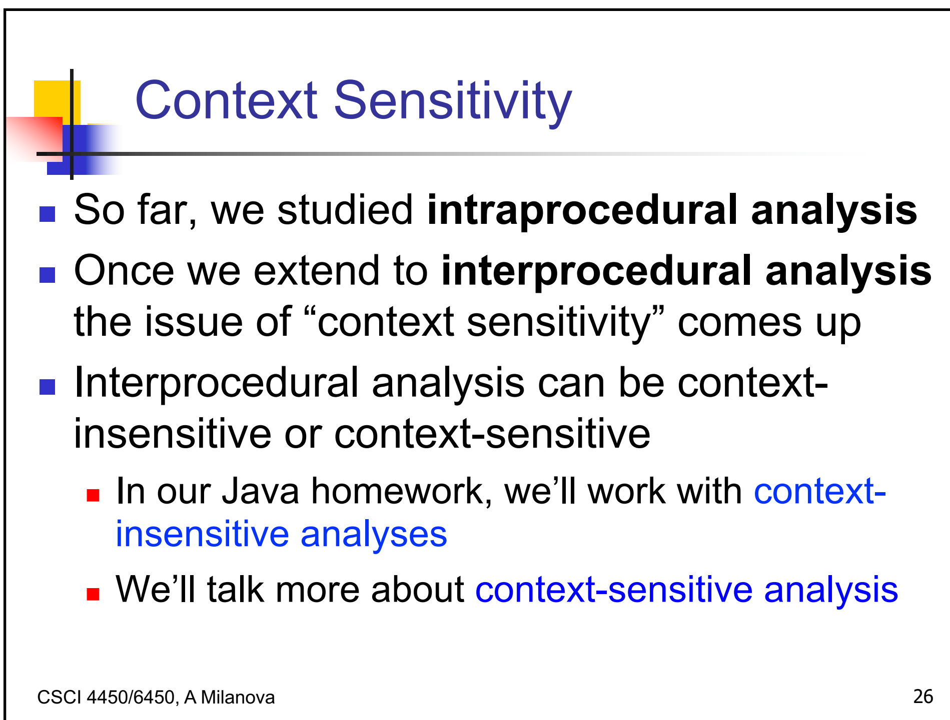
- **Whole-program** analysis
  - Usually, assumes entry point "main"
  - Application code + libraries
    - Intricate interdependences, e.g., Android apps
- **Modular** analysis
  - Scope either a library without entry point
  - or application code with missing libraries
  - … or a library that depends on other missing libraries

CSCI 4450/6450, A Milanova 24

24



25



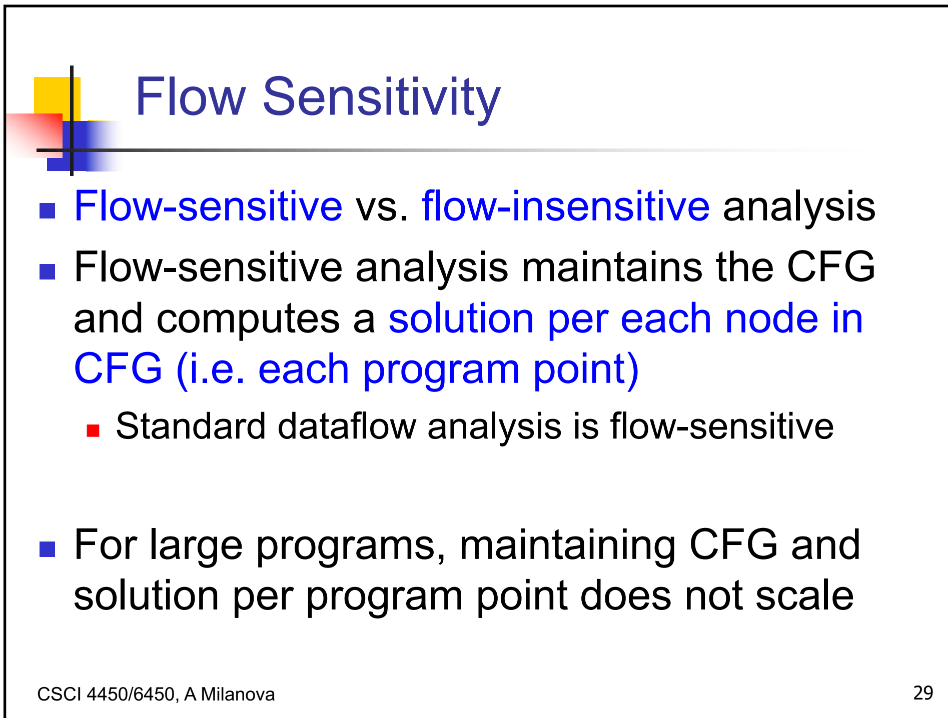
26

Context Insensitivity

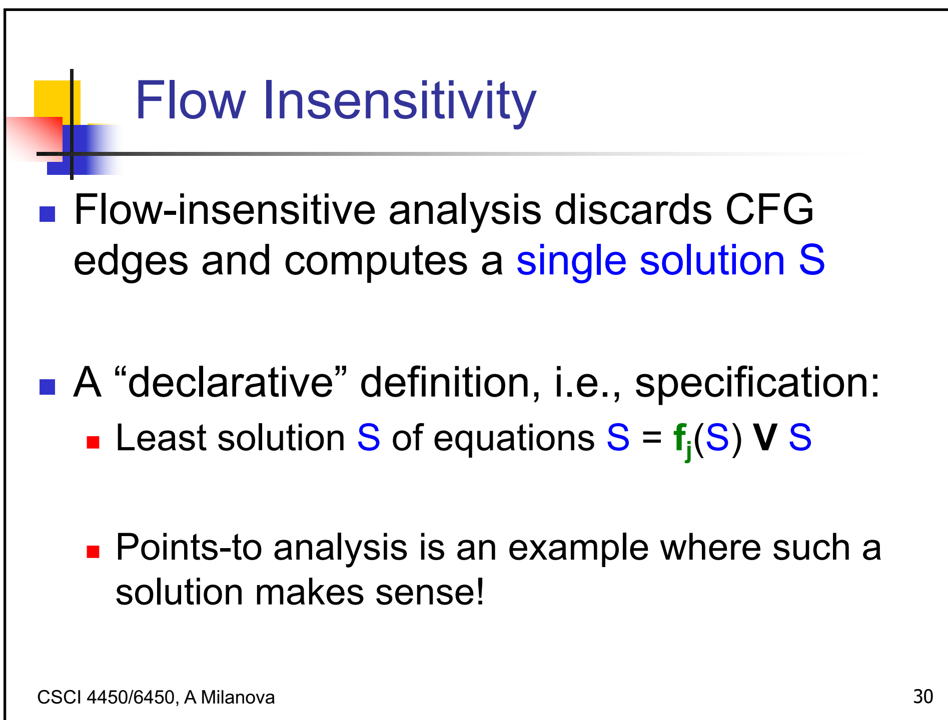
- Context-insensitive analysis makes one big CFG; reduces the problem to standard dataflow, which we know how to solve
- Treats implicit assignment of actual-to-parameter and return-to-left_hand_side as explicit assignment
  - E.g., **x = id(y)** where **id: int id(int p) { return p; }** adds **p = y** // flow of values from arg to param and **x = ret** // flow of return to left_hand_side
- Can be flow-sensitive or flow-insensitive

Context Insensitivity

```
int id(int p) {
  return p;
}
      a = 5;
  2: b = id(a);
      x = b*b;
      c = 6;
  5: d = id(c);
```



29



30

Flow Insensitivity

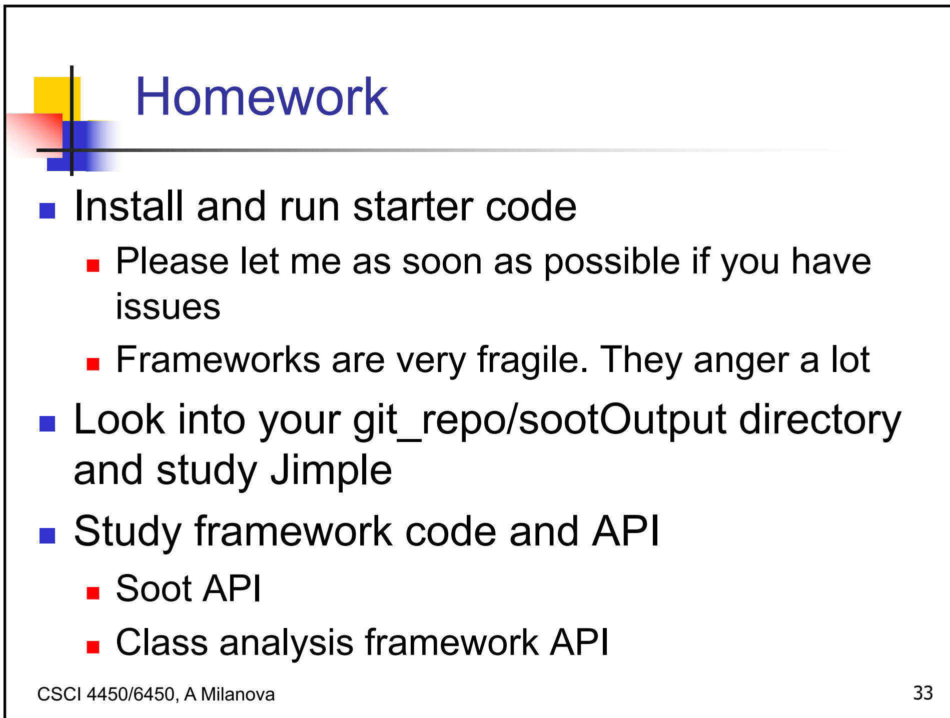
- An "operational" definition. A worklist algorithm:

```
S = 0, W = { 1, 2, … n } /* all nodes */
while W ≠ Ø do {
  remove j from W
  S = fj(S) V S
  if S changed then
    W = W U { k | k is "successor" of j }
}
```

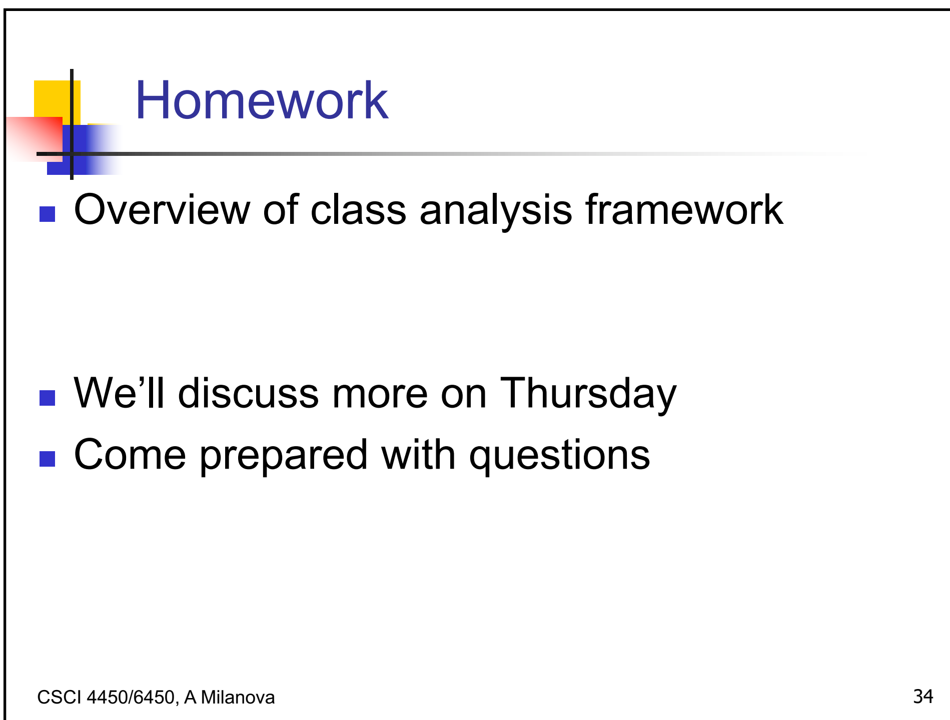
- "successor" is not CFG successor nodes, but more generally, nodes **k** whose transfer function $\mathbf{f_k}$ may be affected as a result of the change in S by **j**

Your Homework

- A bunch of flow-insensitive, context-insensitive analyses for Java
  - RTA, 0-CFA, PTA, other
  - Simple property space
  - Simple transfer functions
    - E.g., in fact, RTA gets rid of most CFG nodes, processes just 2 kinds of nodes!
  - Millions of lines of code in seconds



33



34