Interprocedural Analysis and Context Sensitivity, conclusion

Announcements

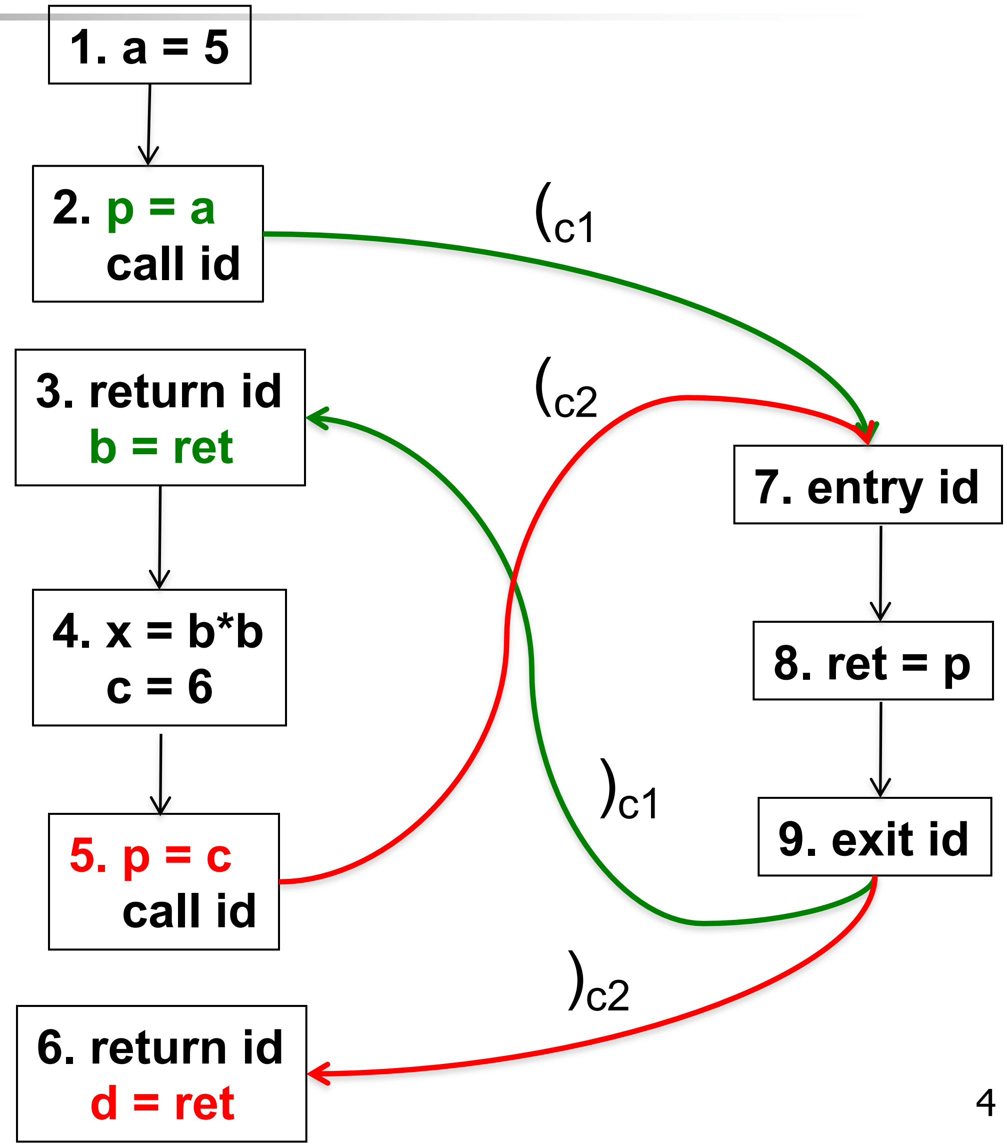
- Quiz 3
- HW3 and HW4?
- More office hours coming up starting tomorrow

So far on interprocedural analysis

- Interprocedural control-flow graph (ICFG)
  - Realizable paths
  - Meet over all realizable paths (MORP)
- Classical ideas in interprocedural analysis
  - Functional approach
  - Call string approach
- Reading
  - Chapter 12.1-3 Dragon book

Realizable Paths

```
int id(int p) {
  return p;
}
      a = 5;
c1: b = id(a);
      x = b*b;
      c = 6;
c2: d = id(c);
```



Outline of Today's Class

- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Summary-based context sensitivity

- Reading
  - Chapter 12.1-3 Dragon book

Context-Sensitive Analysis In Practice

- Transfer functions are not distributive
- Local variables, flow of values from actual arguments to formal parameters, and from return to left-hand-side
- Procedures have side effects!
- Sometimes there is no call graph!
  - Function pointers, virtual calls, functions as first-class values
- Parameter passing mechanisms

Context-Sensitive Analysis In Practice

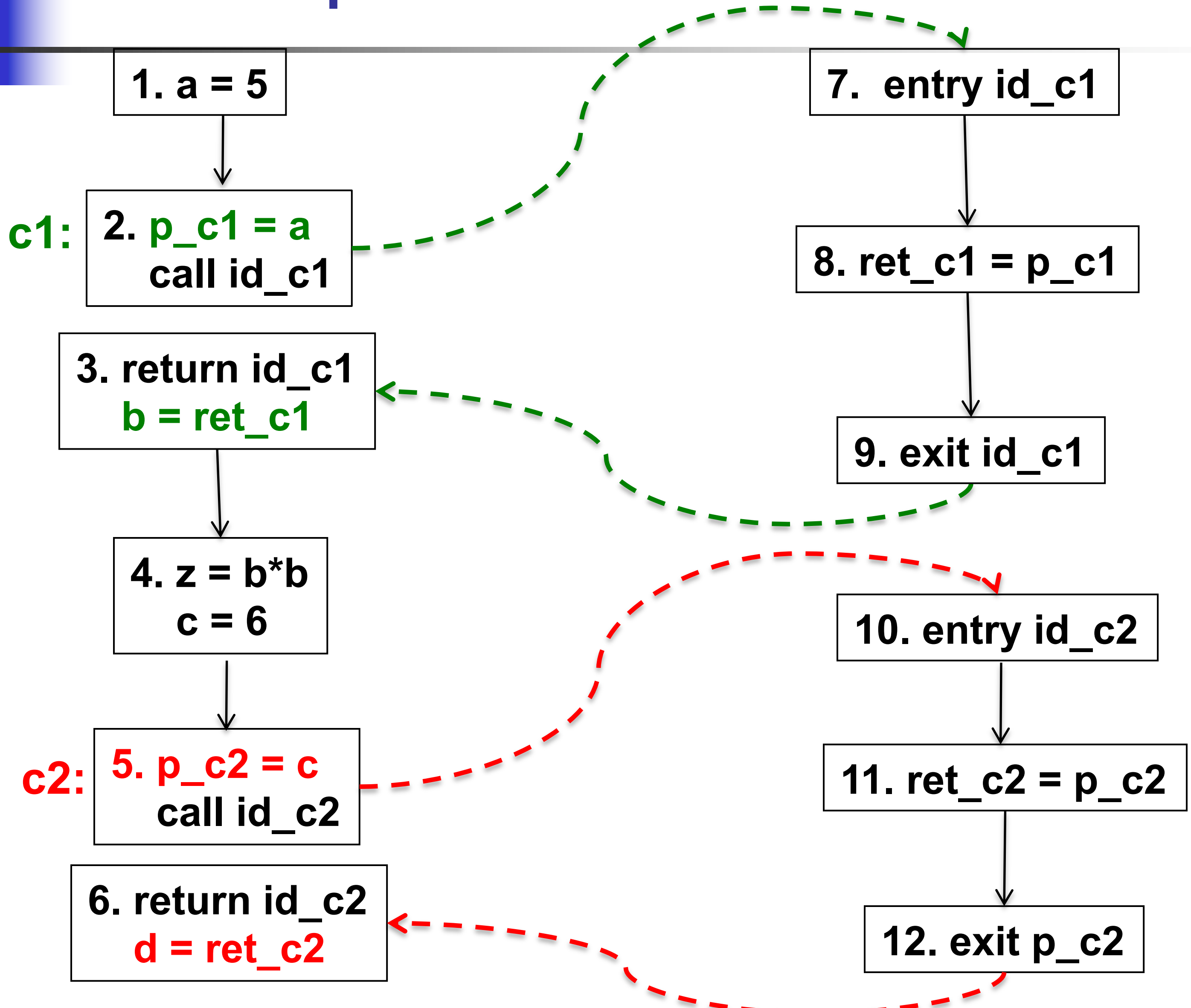
- Ad-hoc adaptation of Sharir and Pnueli's call string or functional approach
- Call-string-based approaches
  - More intuitive than functional one
  - Nearly universally applicable, widely used
- Functional approaches
  - More difficult to implement
  - Not always applicable
  - Better precision and better scalability, in general

Call-String-Based Context Sensitivity

- Call-string-based context-sensitivity uses a _static_ call string as abstraction of the stack

- k-CFA: distinguishes context by **k** most recent call sites that lead to **p**
  - Make a "copy" of procedure **p** for each call string of length **k** in the original program
- 1-CFA: "inline" **p** at each call site of **p** in the original program

Example: 1-CFA



Problems?

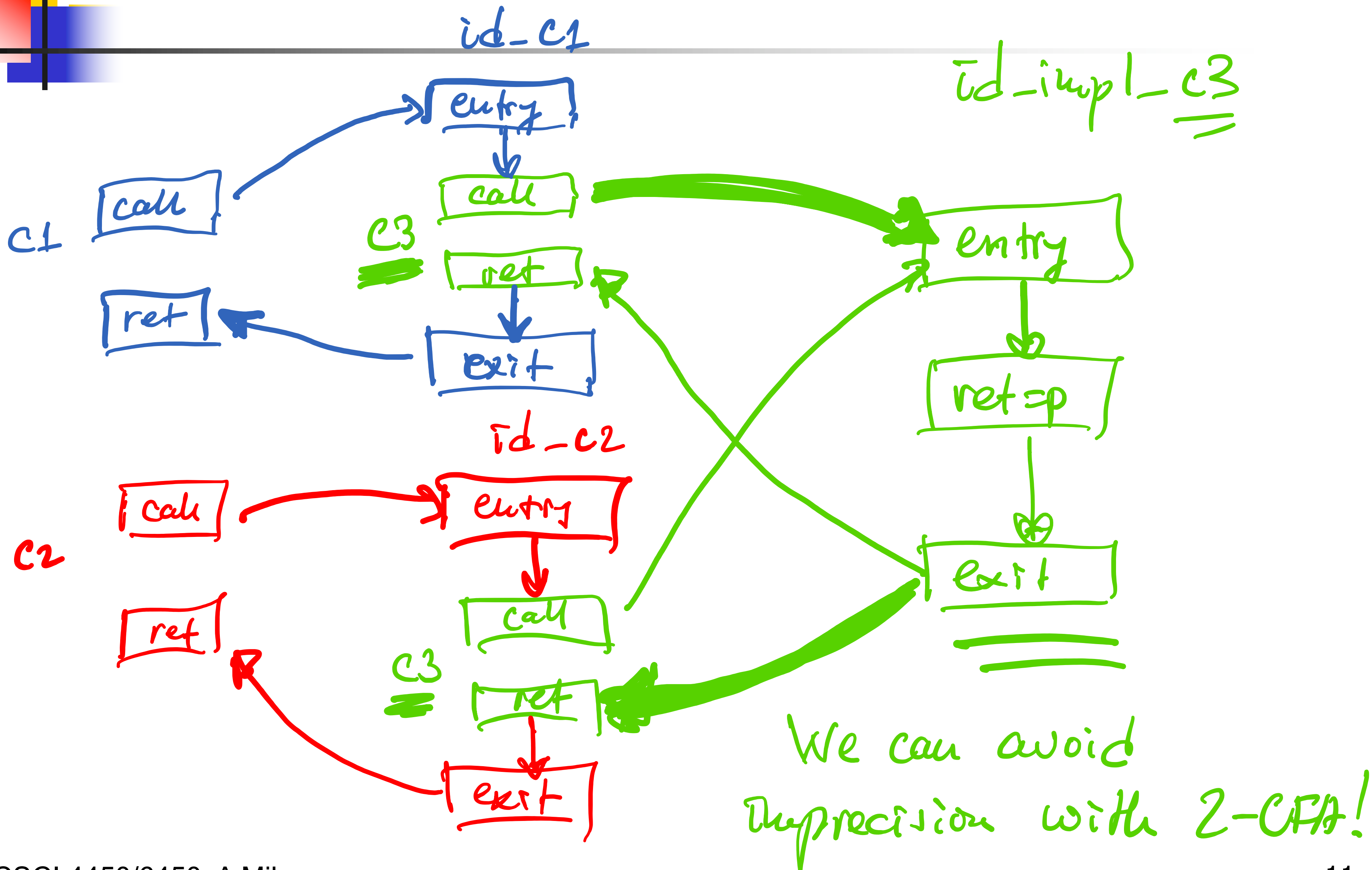
main:

```
   …
   a = 5;
c1: b = id(a);
   z = b*b;
   c = 6;
c2: d = id(c);
   …
```

id:

```
int id(int p) {
  c3: return id_impl(p);
}
…
int id_impl(int p) {
  return p;
}
```

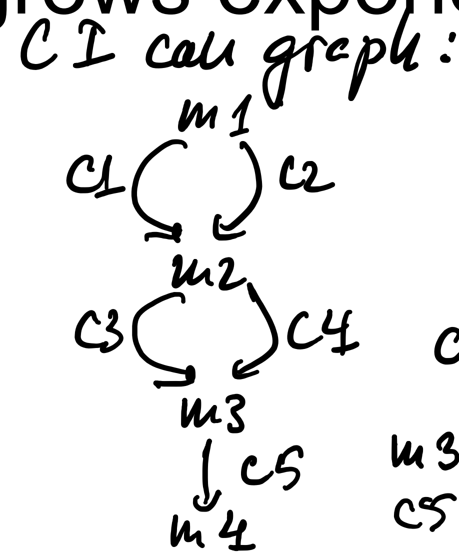
Problems with 1-CFA?

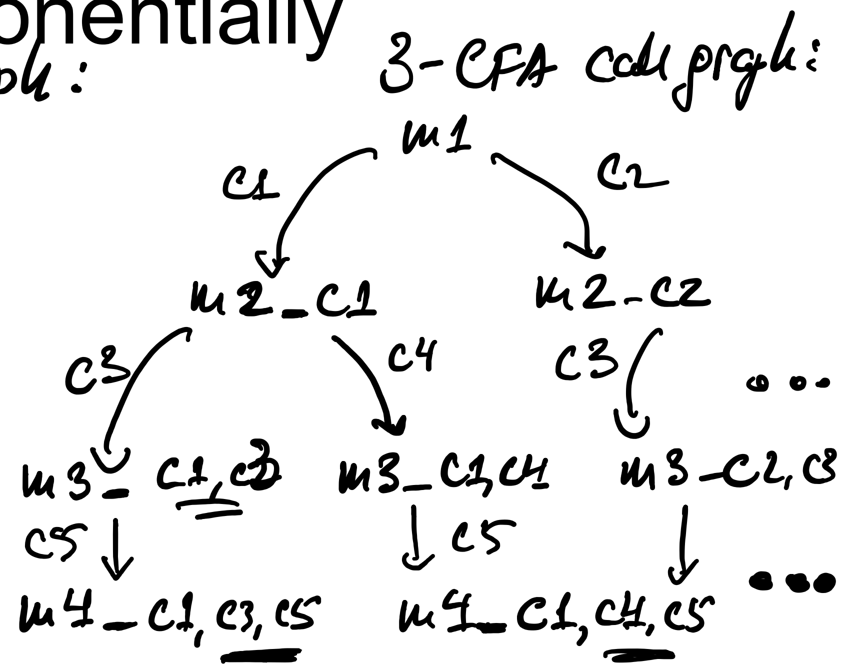


Problems with k-CFA?

- 1-CFA may not be enough
- Program size grows exponentially

```
m1() {
  c1: m2()
  c2: m2()
}
m2() {
  c3: m3()
  c4: m3()
}
m3() { c5: m4(); }
```





- In practice, 2-CFA and 3-CFA are popular approaches

Recall: Points-to Analysis for Java (PTA)

- Saw in context of class analysis framework
- Context-insensitive, flow-insensitive analysis
- Syntax

Object allocation:	**a_i: x = new A // o_i**
Assignment:	**x = y**
Field Write:	**x.f = y**
Field Read:	**x = y.f**
Virtual call:	**c_i: x = y.m(z)**

Recall: PTA

- Next, define the analysis semantics
- Constraints over syntax
  - E.g., Allocation **x = new A // o_i**

for each reachable method **m**

    for each Allocation site **i: x = new A // o_i** in **m**

        $\{ o_i \} \sqsubseteq Pt(x)$

  - Note: **Pt(x)** denotes the points-to set of **x**
- Progression: RTA => XTA => 0-CFA => PTA

Recall: PTA Constraints

a_i: **x = new A // o_i** $\{ o_i \} \sqsubseteq Pt(x)$

x = y $Pt(y) \sqsubseteq Pt(x)$

x.f = y for each **o** in **Pt(x)**. $Pt(y) \sqsubseteq Pt(o.f)$

x = y.f for each **o** in **Pt(y)**. $Pt(o.f) \sqsubseteq Pt(x)$

c_i: **x = y.m(z)**

for each **o** in **Pt(y)**

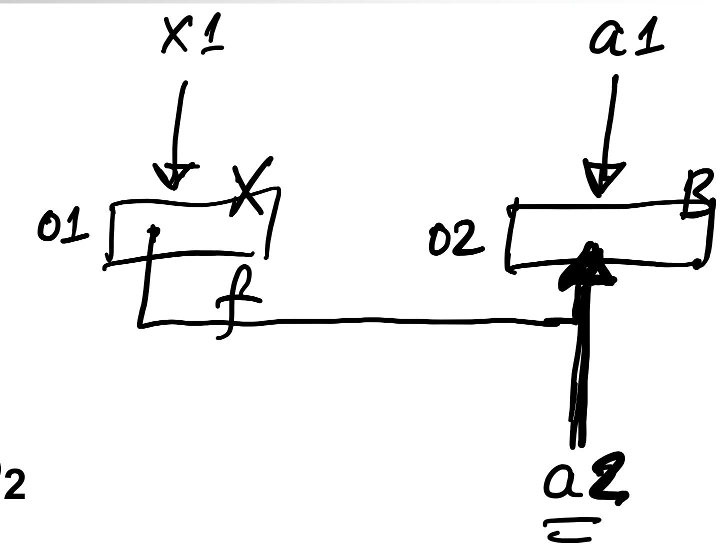
let **m'(this,p,ret) = resolve(o,m)** in

→ $\{ o \} \sqsubseteq Pt(this)$

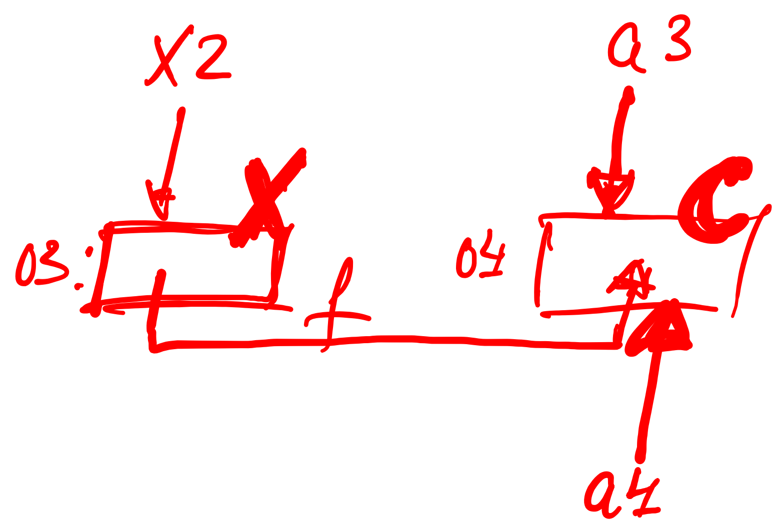
→ $Pt(z) \sqsubseteq Pt(p)$ $Pt(ret) \sqsubseteq Pt(x)$

PTA Example

```
public class A {
   public static void main() {
      X x1 = new X();    // o1
      A a1 = new B();    // o2
      x1.f = a1;  // o1.f points to o2
      A a2 = x1.f;  // a2 points to o2
      a2.m();        a2 : { B }
   }
}
```




```
X x2 = new X(); //o3
A a3 = new C(); //o4
x2.f = a3;
a4 = x2.f;
```



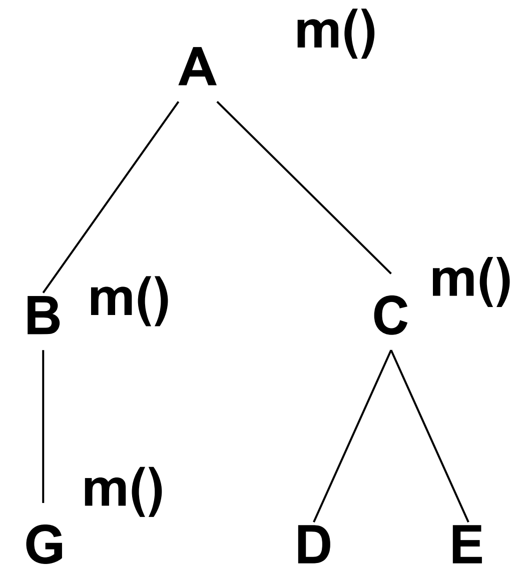
0-CFA vs. PTA Example

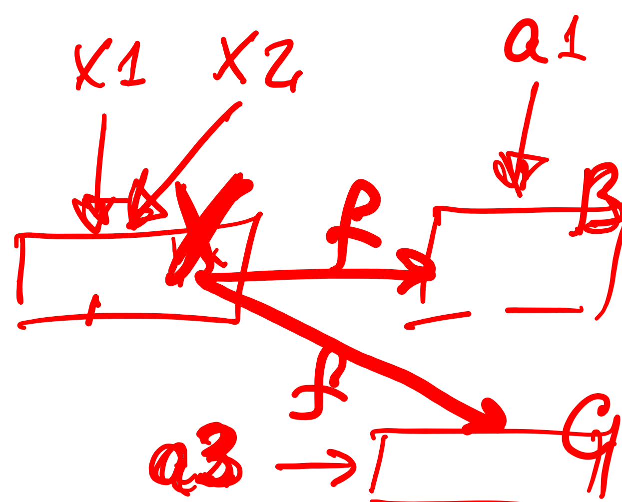
```
public class A {
   public static void main() {
      X x1 = new X();    // o1
      A a1 = new B();    // o2
      x1.f = a1;  // o1.f points to o2
      A a2 = x1.f;  // a2 points to o2
      a2.m();

      X x2 = new X();    // o3
      A a3 = new C();    // o4
      x2.f = a3;  // o3.f points to o4
      A a4 = x2.f; // a4 points to o4
      a4.m();
   }
}
```

PTA: a2: {B}

0-CFA: a2: {B, C}





Another PTA Example

X x1 = new X(); // o_1

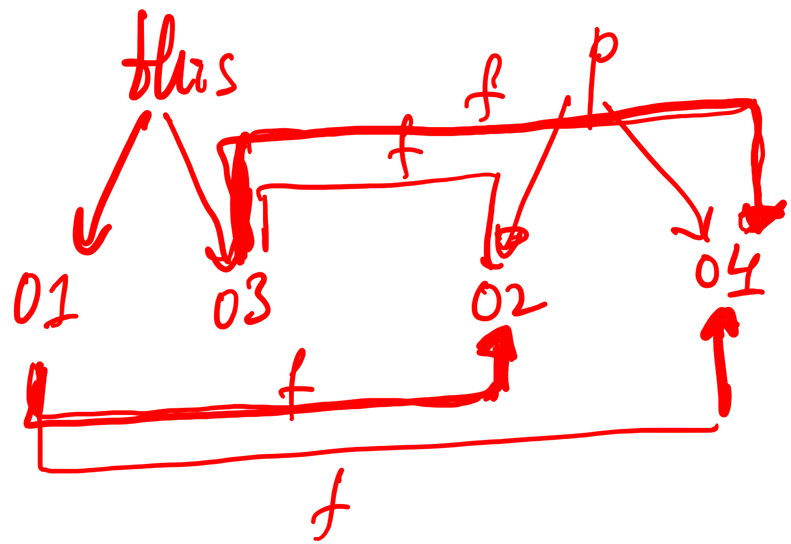
A a1 = new B(); // o_2

c1: x1.set(a1);

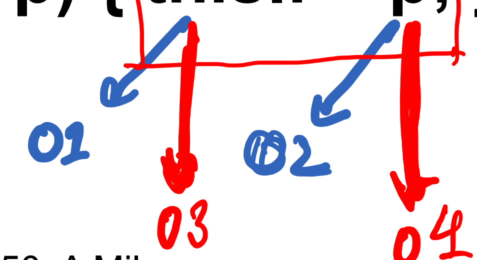
X x2 = new X(); // o_3

A a2 = new C(); // o_4

c2: x2.set(a2);



// set(X p) { this.f = p; }



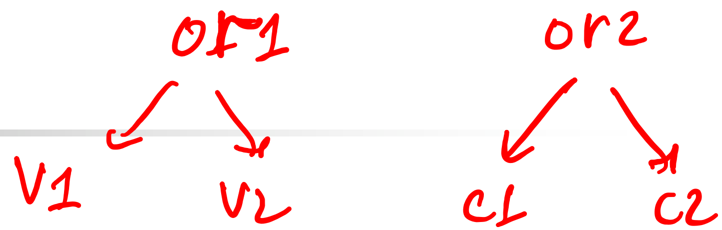
1-CFA PTA Example

```
    X x1 = new X();  // o1
    A a1 = new B();  // o2
c1: x1.set(a1);
    X x2 = new X();  // o3
    A a2 = new C();  // o4
c2: x2.set(a2);

// set(X p) { this.f = p; }
```

this_c1.f = p_c1 and this_c2.f = p_c2

Boolean Expression Hierarchy: PTA




```
main() {
  Context theContext = new Context();

  BoolExp or1 = new OrExp(new VarExp("X"),            // or1
                          new VarExp("Y"));

  BoolExp or2 = new OrExp(new Constant(true),         // or2
                          new Constant(false));

  boolean result1 = or1.evaluate(theContext);
  boolean result2 = or2.evaluate(theContext);
}
```

Boolean Expression Hierarchy: PTA

```
public class OrExp extends BoolExp {
   private BoolExp left; private BoolExp right;

   public OrExp(BoolExp left, BoolExp right) {
      this.left = left;
      this.right = right;
   }

   public boolean evaluate(Context c) {
      private BoolExp l = this.left;
      private BoolExp r = this.right;
      return l.evaluate(c) || r.evaluate(c);
   }
}
```

1-CFA PTA works!

Pt(this) = { $\mathbf{or_1}$, $\mathbf{or_2}$ }
Pt(left) = { $\mathbf{v_1}$, $\mathbf{c_1}$ }, Pt(right) = { $\mathbf{v_2}$, $\mathbf{c_2}$ }
Pt($\mathbf{or_1}$.left) = Pt($\mathbf{or_2}$.left) = { $\mathbf{v_1}$, $\mathbf{c_1}$ } !!!

Pt(l) = { $\mathbf{v_1}$, $\mathbf{c_1}$ }
Pt(r) = { $\mathbf{v_2}$, $\mathbf{c_2}$ }

Boolean Expression Hierarchy: 1-CFA

What If We Changed Boolean Expression Hierarchy? 1-CFA?

```
public abstract class BinaryExp extends BoolExp {
   private BoolExp left;
   private BoolExp right;

   public BinaryExp(BoolExp left, BoolExp right) {
      this.left  = left;
      this.right = right;
   }
}
public class OrExp extends BinaryExp {
   public OrExp(BoolExp left, BoolExp right) {
      c5: BinaryExp.BinaryExp(left,right); // call to super
   }
}
```

What If We Changed Boolean Expression Hierarchy: 1-CFA?

```
main() {
  Context theContext = new Context();




  c1: BoolExp or1 = new OrExp(new VarExp("X"),              // or1
                              new VarExp("Y"));

  c2: BoolExp or2 = new OrExp(new Constant(true),           // or2
                              new Constant(false));

  c3: boolean result1 = or1.evaluate(theContext);
  c4: boolean result2 = or2.evaluate(theContext);
}
```

What If We Changed Boolean Expression Hierarchy: 1-CFA?

2-CFA?

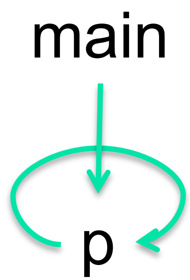
Outline of Today's Class

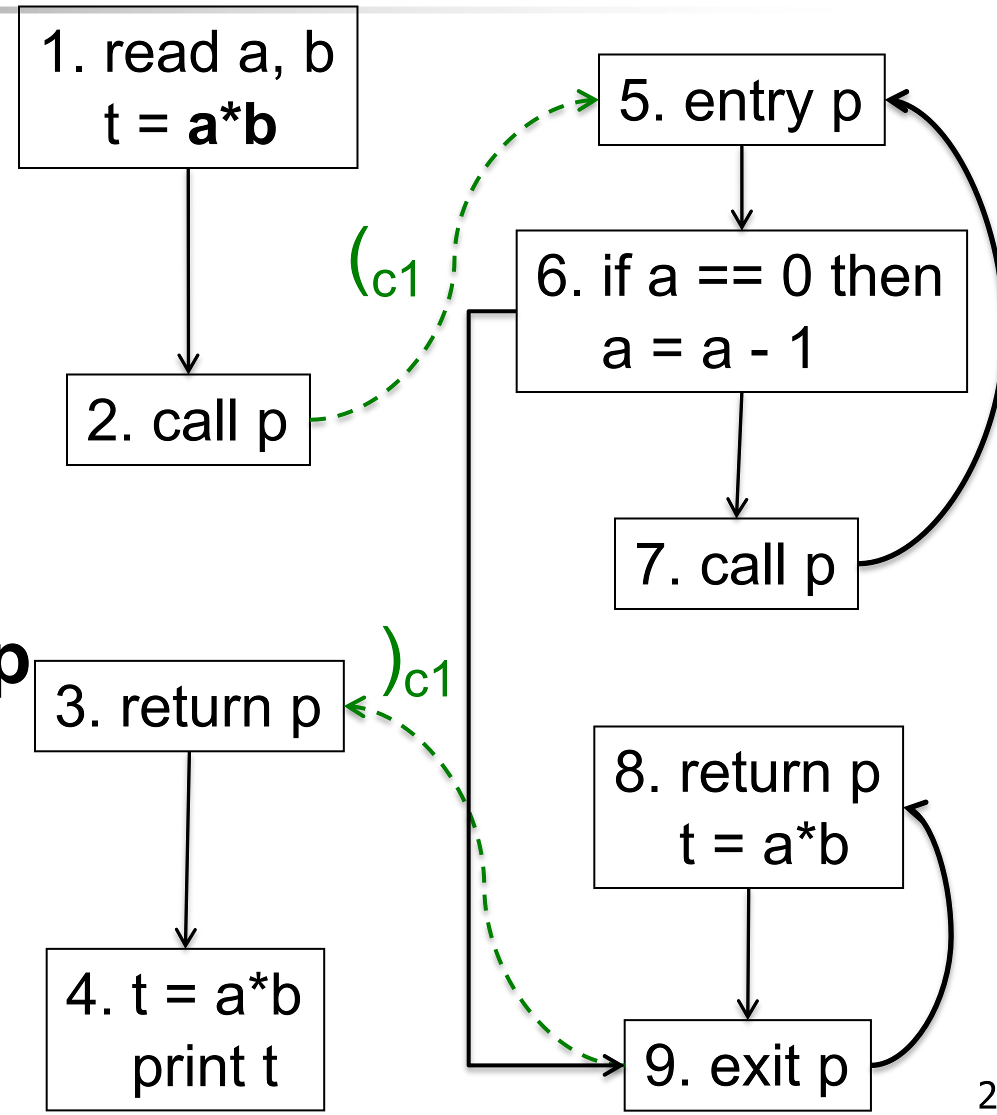
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Summary-based context sensitivity
- Reading
  - Chapter 12.1-3 Dragon book

Summary-based Context Sensitivity

- Compute summary transfer functions
  - **x = id(y)** applies "add **x**⟶**a** for each **y**⟶**a**" (points-to for C example)
  - **p()** applies the "identity function" (Sharir and Pnueli's Available expressions example)
  - **a.set(x)** "sets field **f** of all objects **a** points to to point to the objects **x** points to" (PTA example)
- Phase 1: compute summary transfer functions
  - Collapse into SCC on call graph, then compute summaries bottom up
- Phase 2: propagate values into callees

Strongly-Connected Components





- **p** forms a SCC.
- Compute summary of **p** treating SCC as single procedure
- Summary of **p** says **a*b** is NOT available ☹

Key Points

- Context-sensitive analysis is difficult
- Different approaches
  - Call-string-based, also known as k-CFA
    - 2-CFA and 3-CFA
    - Intuitive, easier to implement
  - Summary-based
    - Harder to design and harder to implement
    - Generally, more precise and more scalable