Interprocedural Analysis and Context Sensitivity

Announcements

- Graded HW2
  - Fix issues before moving on to HW3
  - You can submit HW2 in Submitty

- One issue in virtual call **y.m()**
  - **static_type_of_y**
    - It is not target.getDeclaringClass()
    - It is ((RefType) args.get(0).getType()).getSootClass() in virtualCallStmt

y

So Far

- Four classical dataflow problems
  - **Intra**procedural
  - Flow-sensitive
- Class analysis: RTA, XTA, 0-CFA, and PTA
  - **Inter**procedural
  - Flow-insensitive and context-insensitive analysis

- Interprocedural analysis and context sensitivity

Outline of Today's Class

- 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

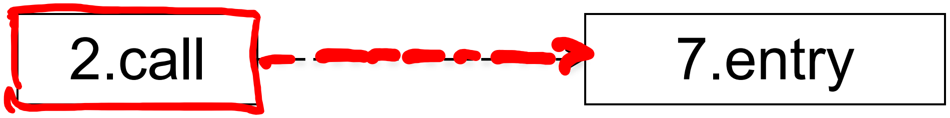
Outline of Today's Class

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

- Reading
  - Chapter 12.1-3 Dragon book

Interprocedural Control Flow Graph (ICFG)

- Add procedure entry node and exit node
- At each procedure call add
  - A call node and a call-entry edge



  - A return node and an exit-return edge

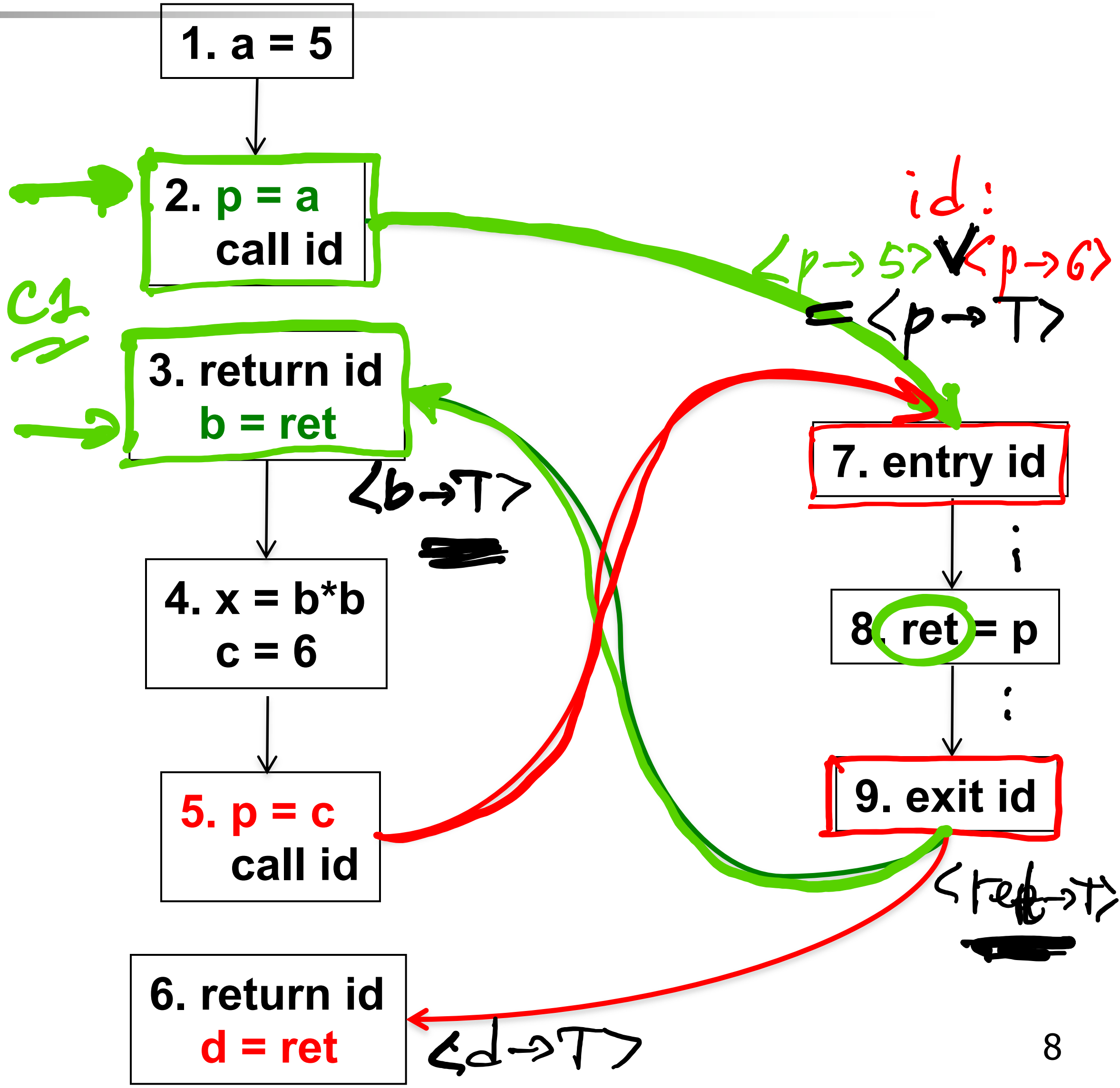
Context-Insensitive Analysis

- Add explicit assignments at call and return
  - E.g., **x = id(y)**
  - **p = y** models flow from actual argument **y** to formal parameter **p**
  - **x = ret** models flow from return to left-hand-side
- Treat ICFG as one big CFG
  - Can be flow-sensitive or
  - Flow-insensitive XTA, 0-CFA, PTA, RTA
    - E.g., Andersen's points-to analysis for C

шсети

Context Insensitivity

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

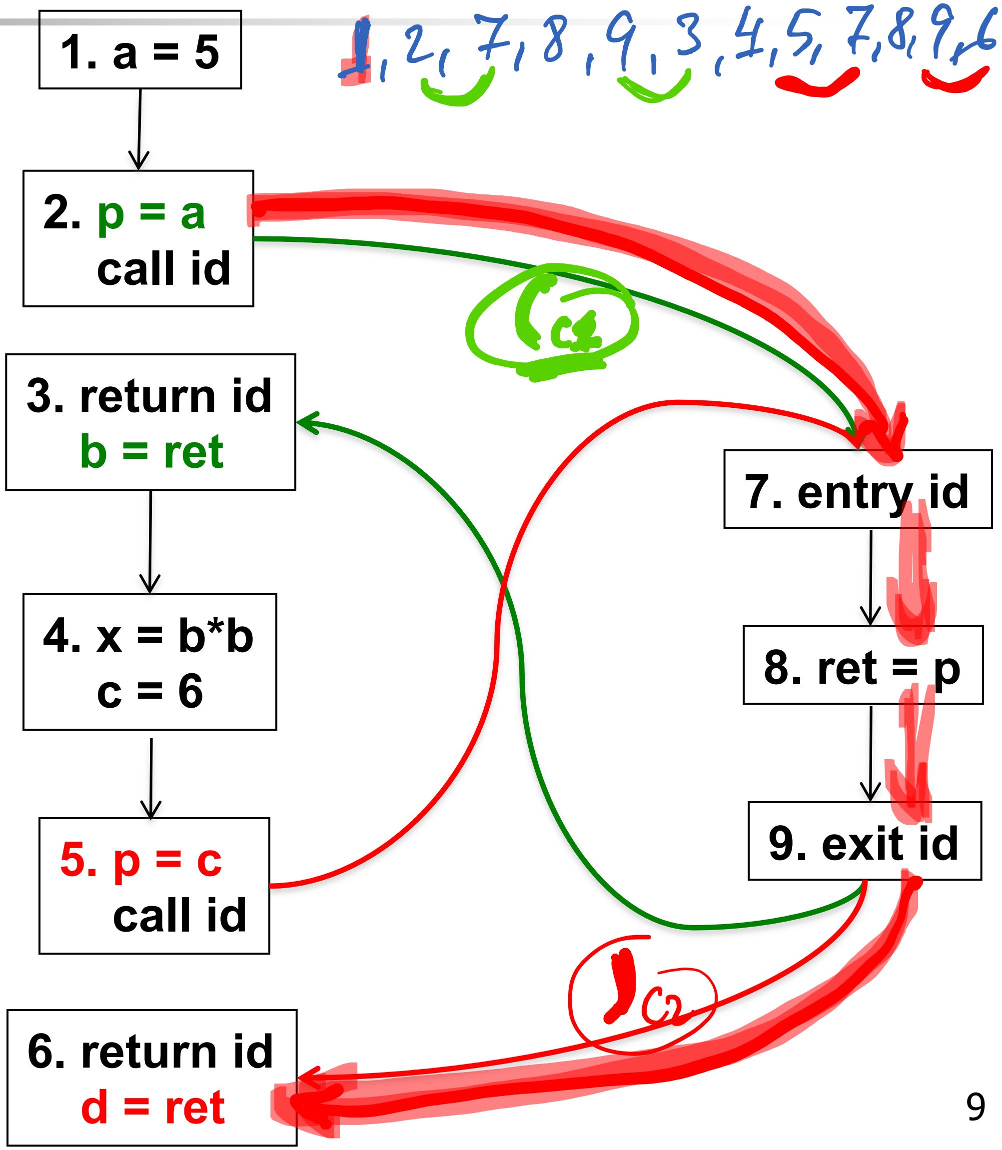


Unrealizable Paths

1, 2, 7, 8, 9, 6 → unrealizable path

1, 2, 7, 8, 9, 3, 4, 5, 7, 8, 9, 6

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



1. a = 5

2. p = a
call id

3. return id
b = ret

4. x = b*b
c = 6

5. p = c
call id

6. return id
d = ret

7. entry id

8. ret = p

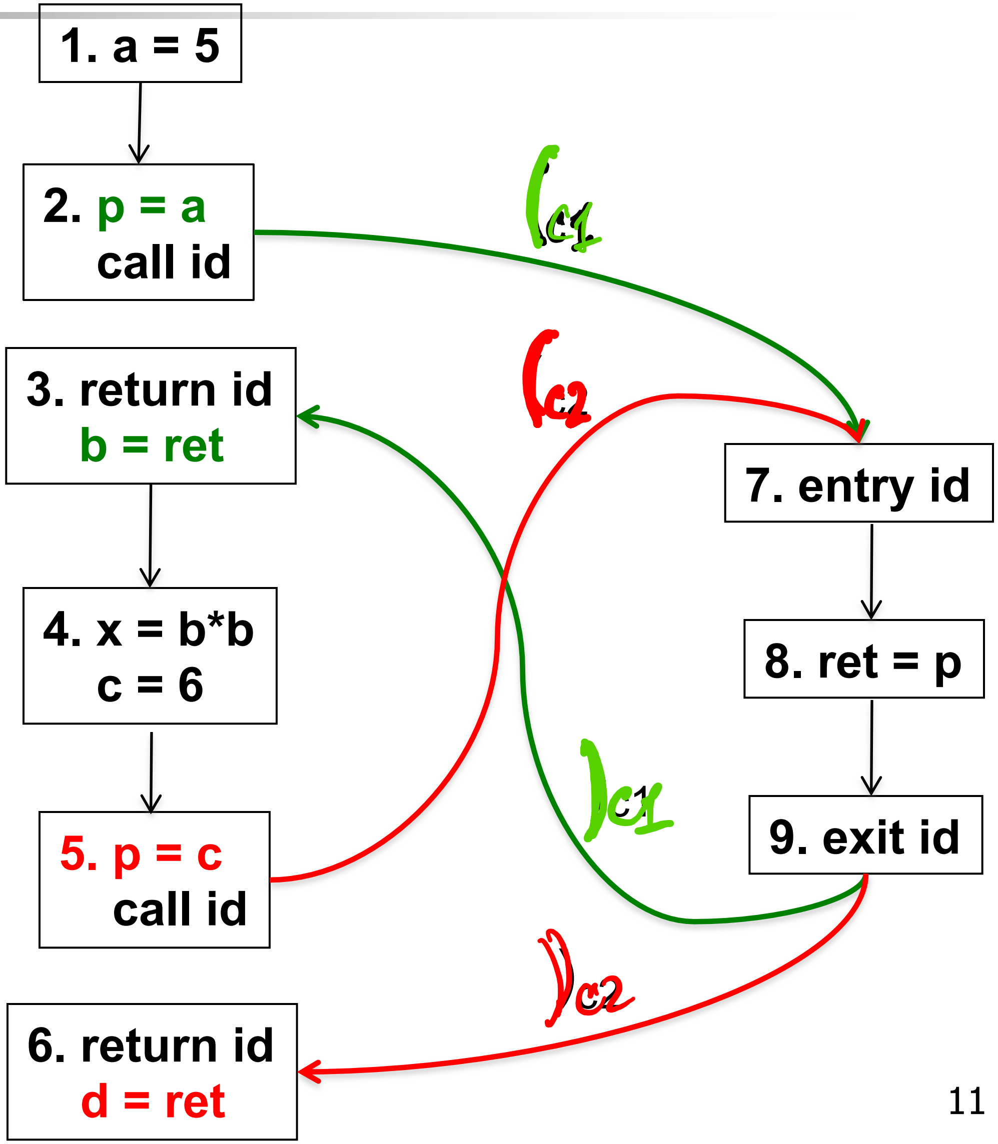
9. exit id

Context-Insensitive Analysis

- Problem with context-insensitive analysis: propagates data along “unrealizable paths”
- Goal of context-sensitive analysis is to propagate data along “realizable paths”

Realizable Paths

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



1. a = 5

2. p = a
call id

3. return id
b = ret

4. x = b*b
c = 6

5. p = c
call id

6. return id
d = ret

7. entry id

8. ret = p

9. exit id

Another Example (p3 from HW3)

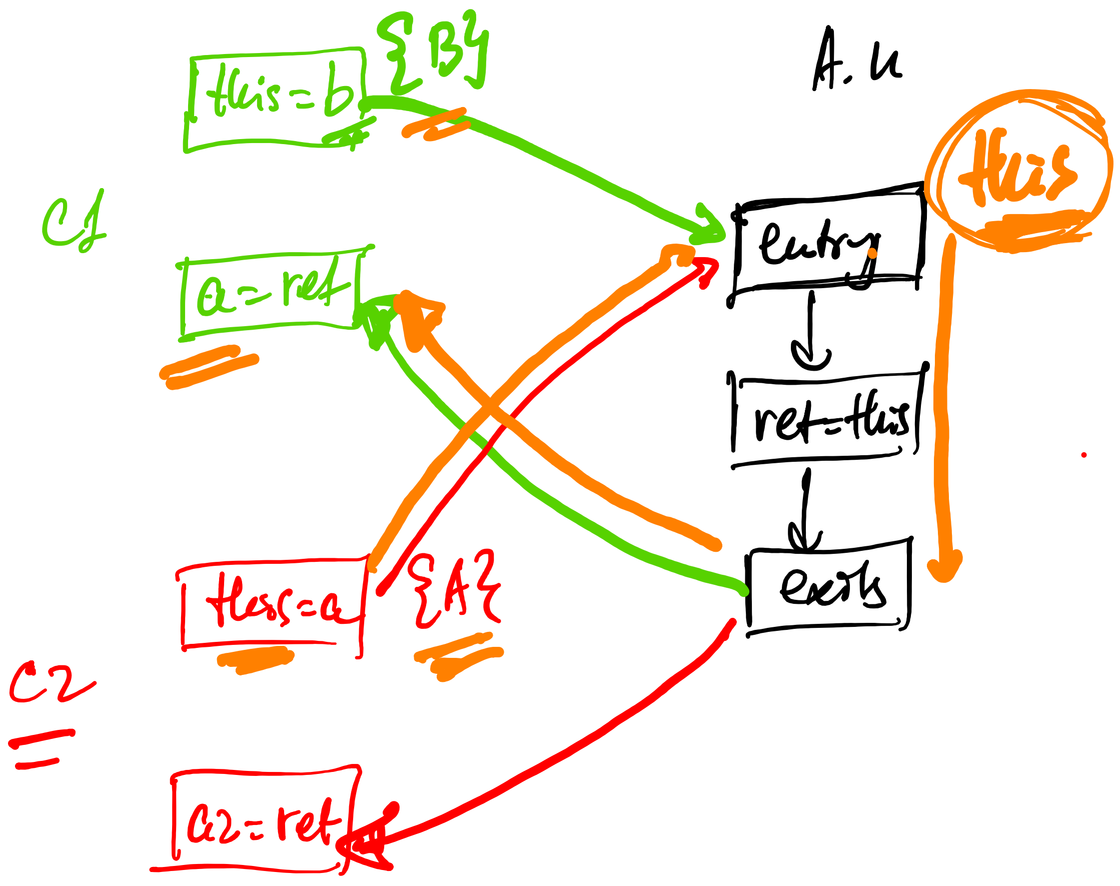
```
class A {
  main() {
    B b = new B();
    b.m();

    A a = b.n();
    a.m();
  }
  A n() { return this; }
}

class B extends A {
  void m() {
   A a = new A();
   A a2 = a.n();
  }
}
```

C1

C2

GT: a : {B}

0-CFA : a: {A,B}



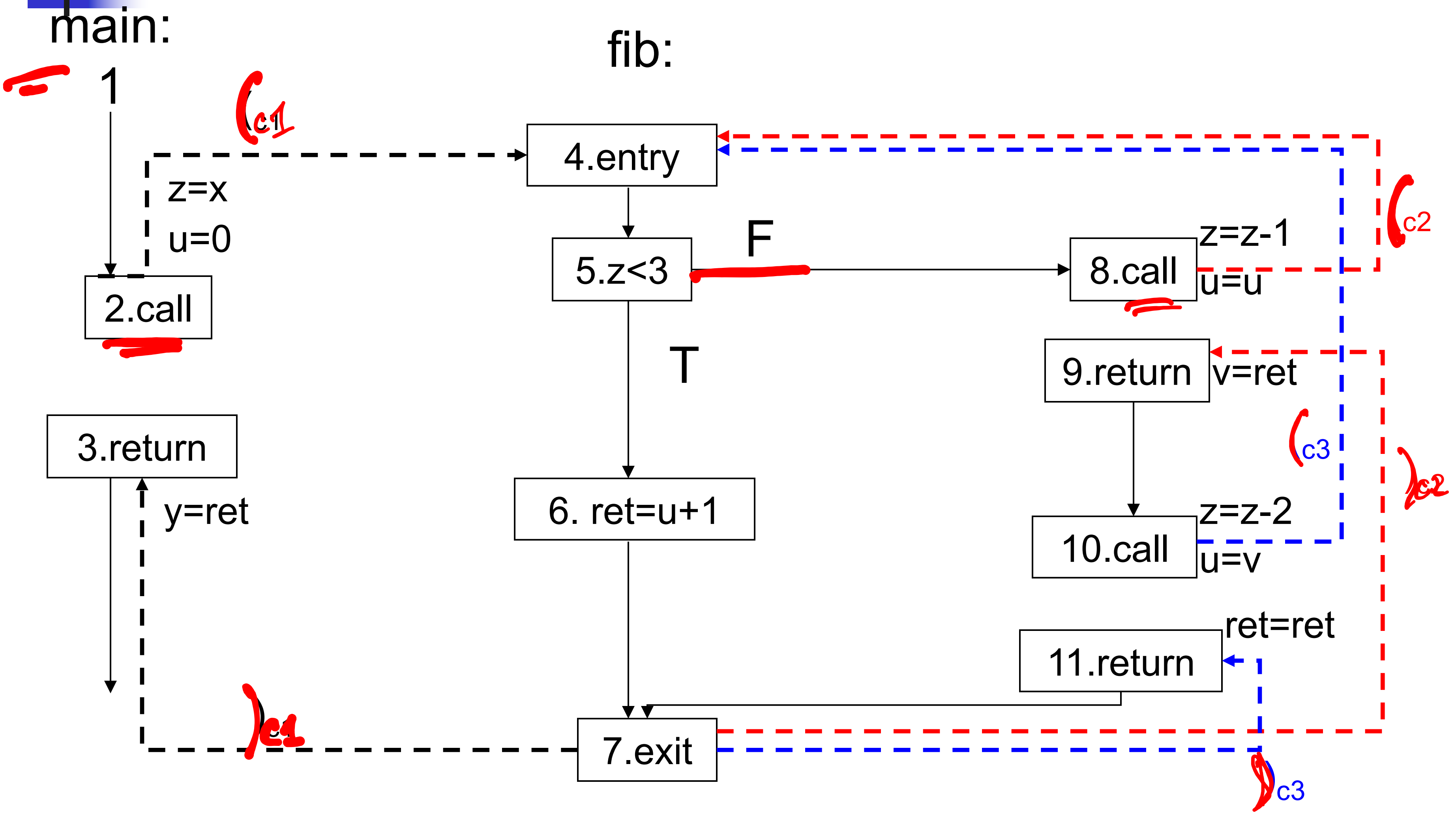
0-CFA propagates A to a.m() in main along the unrealizable orange path.

Another Example

```
int fib(int z, int u) {
    if (z<3) {
        return u+1;               /* ret = u+1;
    } else {                       auxiliary variable ret
        c2: v = fib(z-1,u);        holds the return values */
        c3: return fib(z-2,v)
    }
}
 …
c1: y = fib(x,0);
 …
```

What does **fib** compute? Here **z** and **u** are formal parameters; **ret** is the special variable holding the return value.

Another Example



Realizable Paths (RP)

- Context-free grammar!
- Grammar describes same-level path (**SLP**):

$M ::= \mathbf{e}$ **e** denotes intraprocedural edge

$\mid (_{ci}\ M\ )_{ci}$ path from call to return

$\mid M_1\ M_2$

  - An **intra**procedural edge is annotated with **e**
  - Call-entry edge that originates at call site $\mathbf{c_i}$ is $\mathbf{(_{ci}}$
  - Corresponding exit-return edge is $\mathbf{)_{ci}}$
- A path **p**, from **m** to **n**, is in $\mathbf{SLP_{m,n}}$ iff string along **p** is in language described by *M*

Realizable Paths (RP)

- What about paths with outstanding calls (calls that have not yet returned)?
- Another grammar:

$C ::= (_{ci} \mid M\ (_{ci} \mid C\ (_{ci} \mid C\ M$

- A path from entry node 1 to node **n** is in $\mathbf{RP}_{1,n}$ iff the string from 1 to **n** is in the language generated by either *M* or *C*
  - E.g., in Constant prop example, 1,2,7,8,9,3 is in $\mathbf{RP}_{1,3}$ but 1,2,7,8,9,3,4,5,7,8,9,3 is NOT in $\mathbf{RP}_{1,3}$

$(_{c1}$ $)_{c1}$

$(_{c1}$ $)_{c1}$ $(_{c2}$ $)_{c1}$

cannot call at c2 and return to c1.

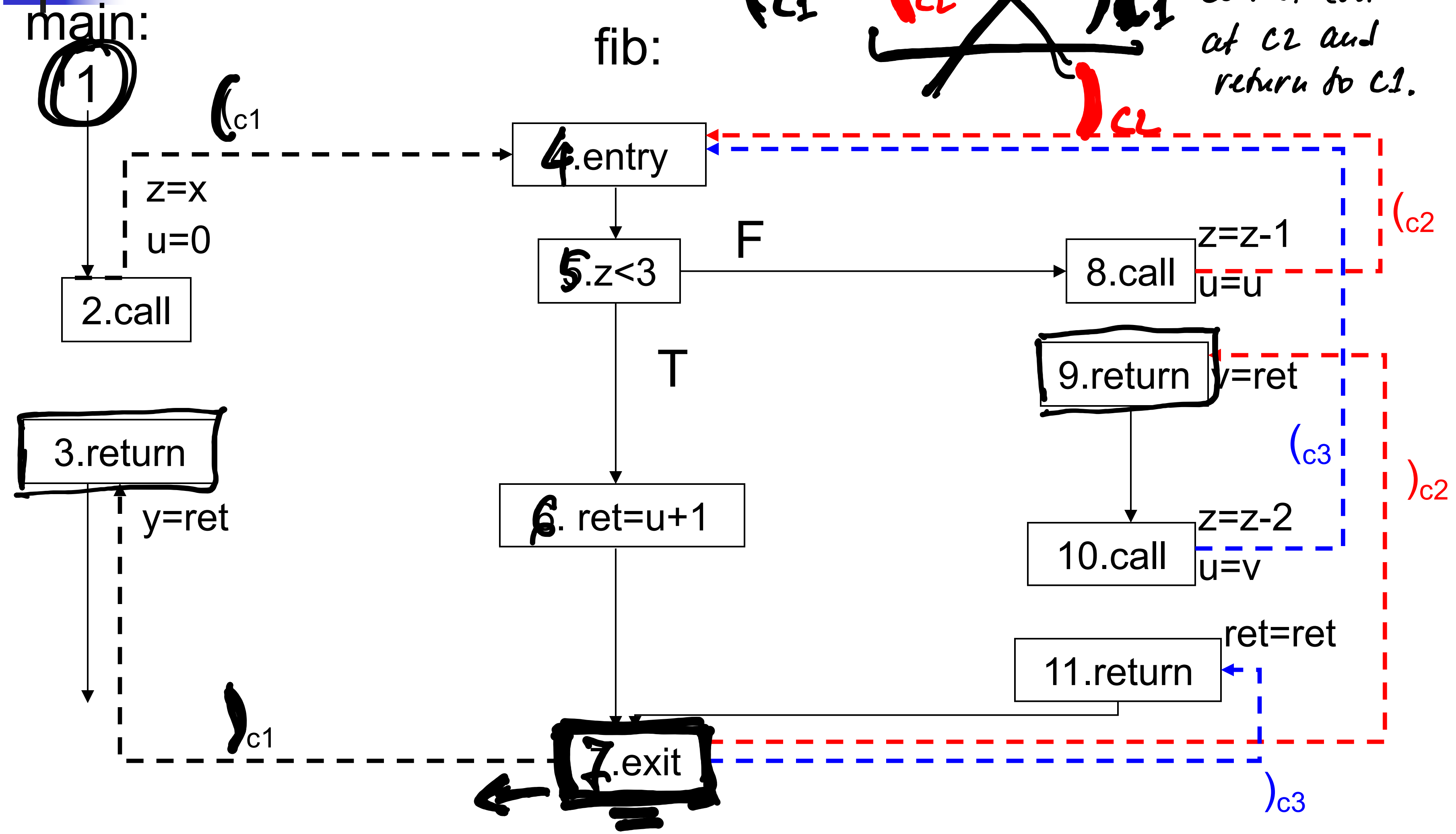
Is p1 = 1,2,4,5,6,7 in $RP_{1,7}$? YES

(c1

Is p2 = 1,2,4,5,8,4,5,6,7,3 in $RP_{1,3}$? NO

(c1 (c2)c1)c2

Cannot call at c2 and return to c1.



Meet Over All Realizable Paths (MORP)

- **MORP (n)** = $\bigvee \mathbf{f_{n_k} \circ f_{n_{k-1}} \circ \ldots f_{n_2} \circ f_1}(\text{init})$

  $p=(1,n_2\ldots n_k,n)$ is a path in $RP_{1,n}$

(∘ denotes function composition)

  - Also called MVP (meet over all **valid** paths) or just MRP

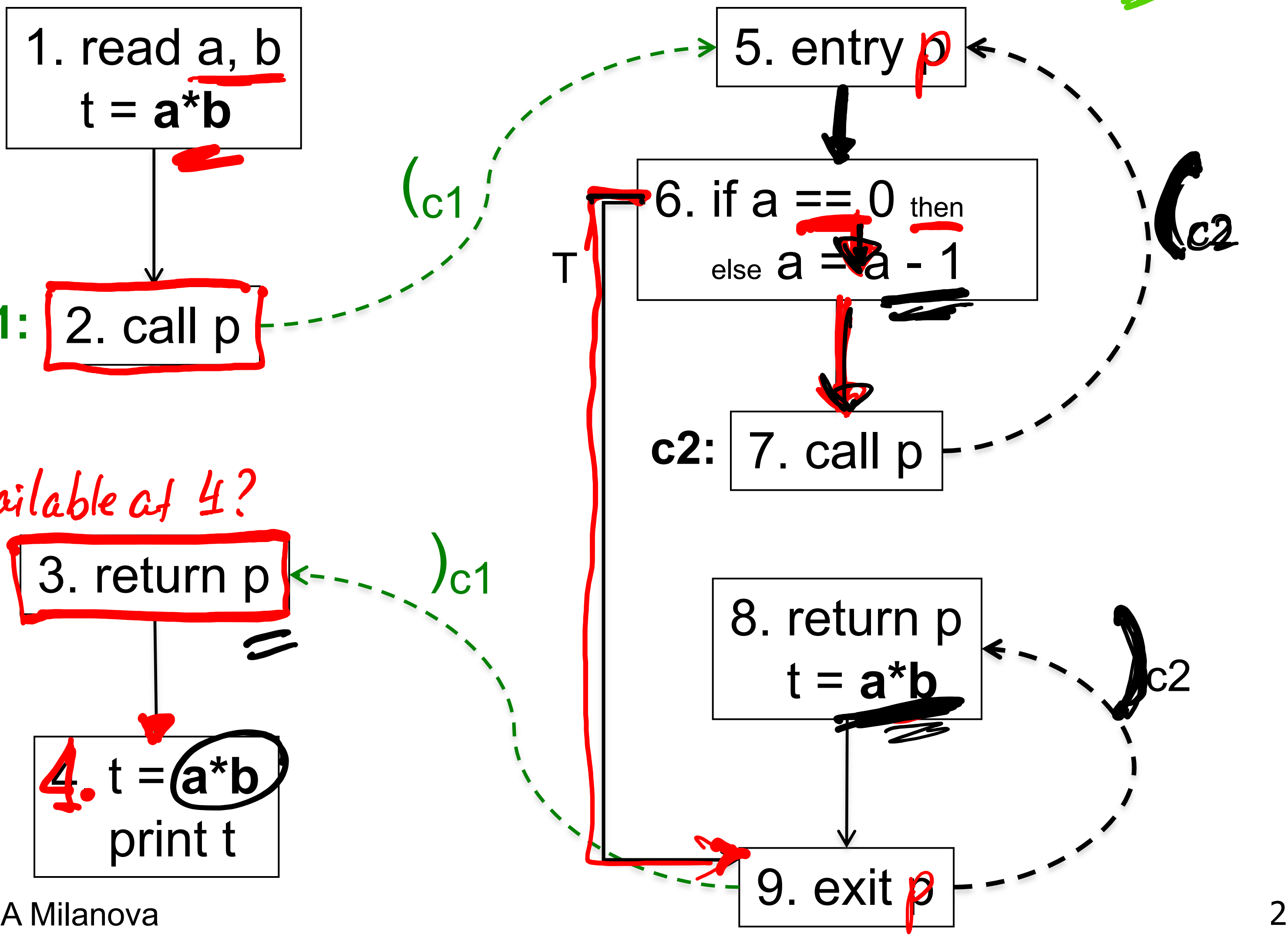
- MORP(n) ≤ MOP(n). Why?
- May be undecidable, even for lattices of finite height
- Goal: encode context and restrict flow over realizable paths, as much as possible

Outline of Today's Class

- 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

Sharir and Pnueli Example (Available Expressions)

unrealizable path:
1, 2, 5, 6, 7, 5, 6, 9, 3
(c2)c1



1. read a, b
t = **a*b**

c1: 2. call p

(c1

3. return p

)c1

4. t = **a*b**
print t

5. entry p

6. if a == 0 then
else a = a - 1

T

c2: 7. call p

(c2

8. return p
t = **a*b**

)c2

9. exit p

Q: Is a*b available at 4?

GT: YES!

CI: NO!

CS: YES!

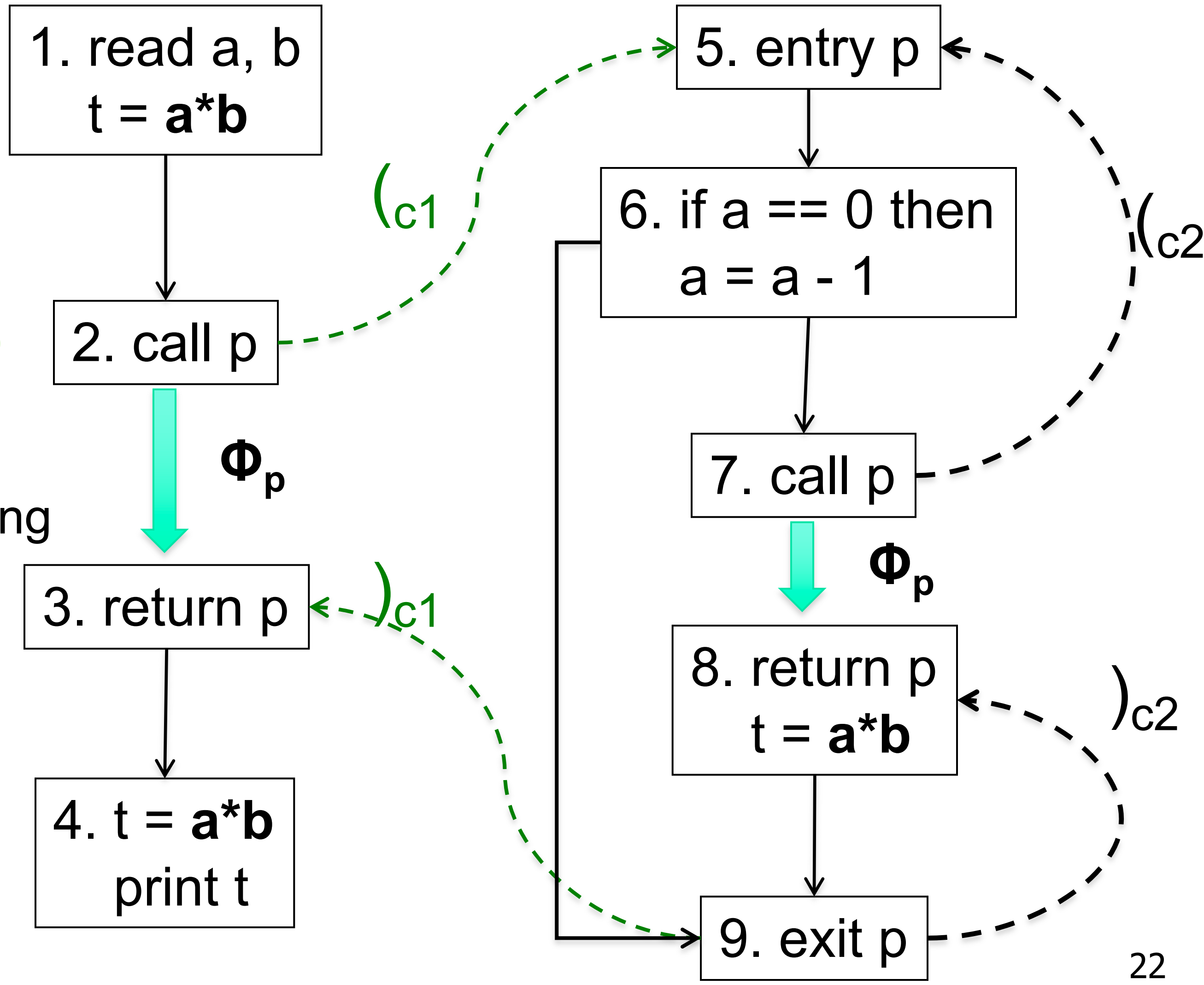
Functional Approach

- Operates on unchanged property space
- Computes summary transfer functions Φ_p that summarize the effect of procedure **p**

- Reduces problem to intraprocedural case:
  - in(**return p**) = Φ_p(in(**call p**))
  - thus, avoids propagation from callee along the **exit p** $\dashrightarrow$ **return p** edge!

Functional Approach

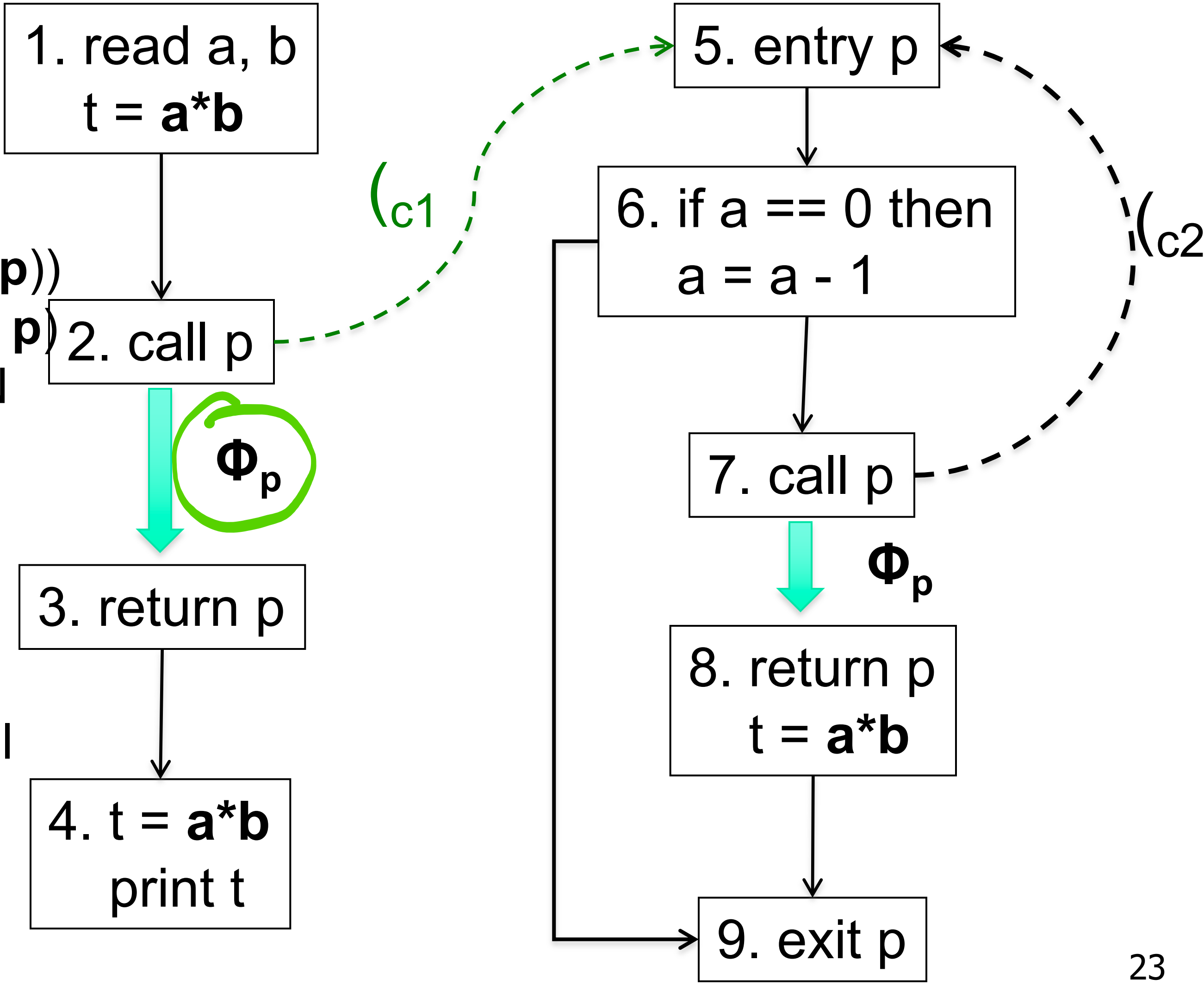
Phase 1:
Compute **summary transfer functions** Φ_p that capture effect of **p.**
In example Φ_p is the **identity function**: nothing gets generated and nothing gets killed (simplifying a bit)



Functional Approach

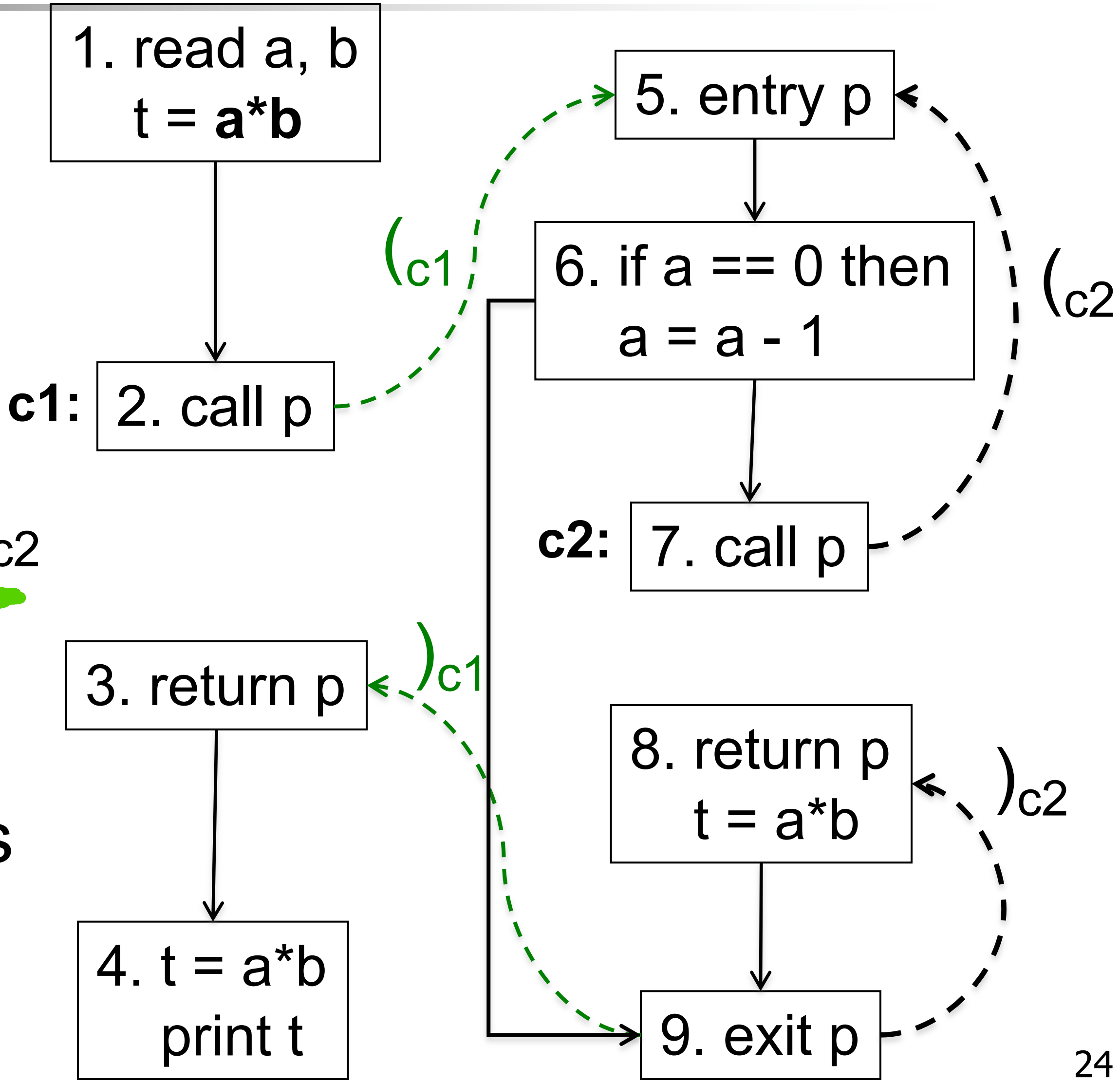
Phase 2:
Dataflow analysis:
•At **return p**
in(**return p**) = $\mathbf{\Phi_p}$(in(**call p**))
out(**return p**) = in(**return p**)
AVOIDS PROPAGATION
along exit-return edges!

•At **entry p**
in(**entry p**) = V in(**call p**)
(propagates facts from all callers to callee)



Call String Approach

- A call string records outstanding calls in a path
- E.g., call string $(_{c1}(_{c2}$ denotes that “we got there” on a path with outstanding calls at **c1** and at **c2**



1. read a, b
t = **a*b**

c1: 2. call p

3. return p

4. t = a*b
print t

5. entry p

6. if a == 0 then
a = a - 1

c2: 7. call p

8. return p
t = a*b

9. exit p

Call String Approach

- Tags solutions per program point with corresponding call string
- Multiple tagged solutions per program point **j** in **p**:
  - Sharir and Pnueli Example:
  - We have $<\{\, a*b \,\},\ (_{c1}>$, $<\{\ \},\ (_{c1}(_{c2}>$ at **6**
  - Meaning: **a*b** is available at **6** on paths with outstanding call string c1, but it is not available on paths with outstanding call string c1 c2

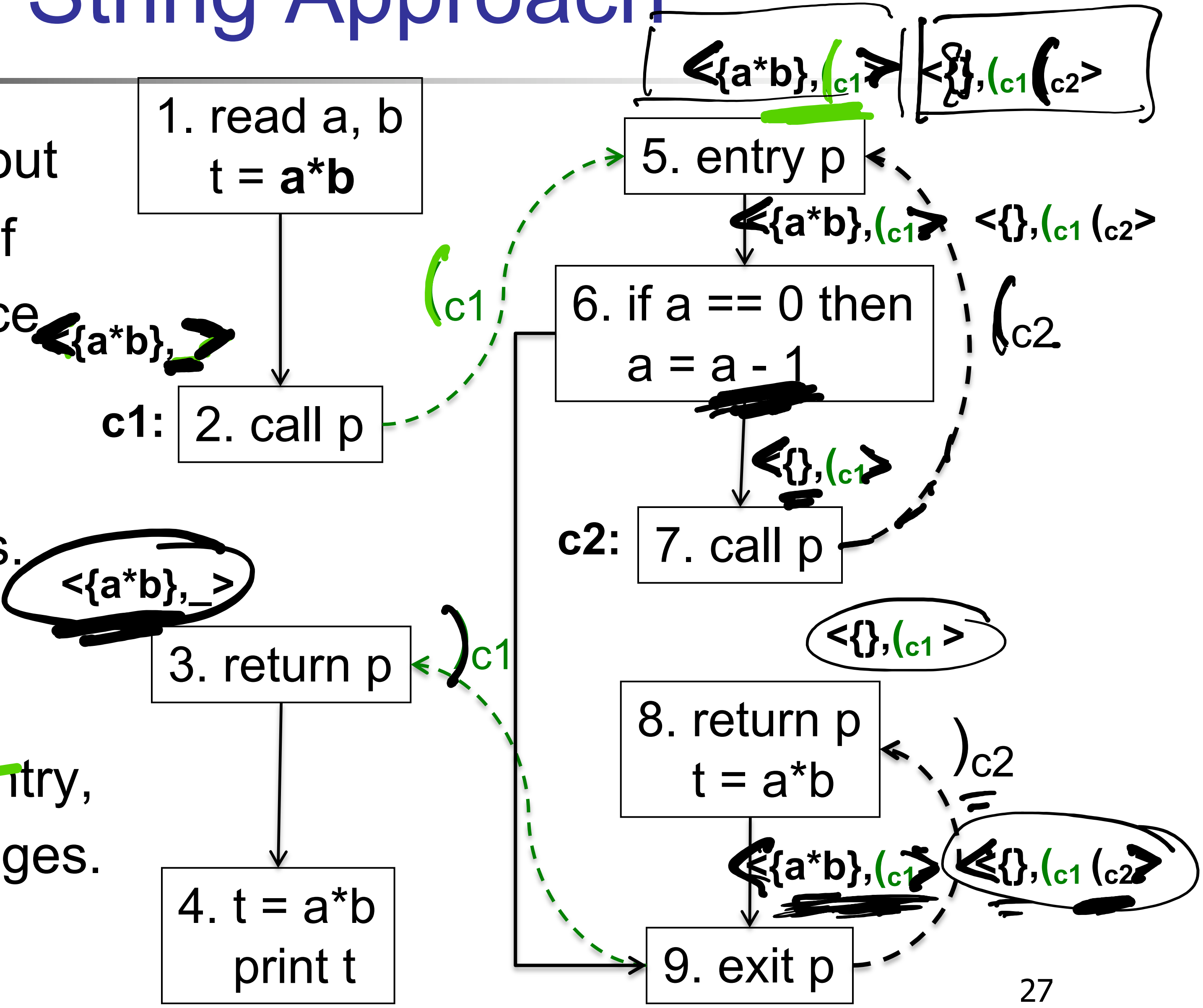
Call String Approach

- Apply original transfer functions point-wise on elements of the original, i.e., “intraprocedural” dataflow lattice
  - Elements: { **a*b** }, { **a*b**, **a+b** }, {}, etc.

- Extend to handle call-entry and exit-return
  - At call-entry, simply append $(_{ci}$
  - At exit-return, propagate only if $)_{ci}$ matches!

Call String Approach

1. Extend in/out sets to sets of “tagged” lattice elements.
2. Apply orig. transfer funcs. point-wise.
3. Extend to handle call-entry, exit-return edges.



1. read a, b
t = **a*b**

c1: 2. call p

3. return p

4. t = a*b
print t

5. entry p

6. if a == 0 then
a = a - 1

c2: 7. call p

8. return p
t = a*b

9. exit p

Sharir and Pnueli, Key Result

- $\mathbf{S_{FA}(j)}$ is the solution at **j** computed by the functional approach
- $\mathbf{S_{CS}(j)}$ is the solution at **j** computed by the call string approach
- For (certain) distributive functions and finite lattices

$$\mathbf{S_{FA}(j) = S_{CS}(j) = MORP(j)}$$

- Caveats?

Sharir and Pnueli, Key Result

- Caveats
  - Summary functions $\boldsymbol{\Phi_p}$ difficult to compute
  - With recursion, infinite call strings, $\mathbf{S_{CS}}$ is infinite
  - Even for distributive functions and finite lattices, $\mathbf{S_{FA}}$ and $\mathbf{S_{CS}}$ cannot be computed in general

  - Simple programming model
  - Only distributive analysis

Outline of Today's Class

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

  - We'll continue next time

- 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