Lecture 8 – Inter-procedural Analysis

SWS128: Software Analysis

Jihyeok Park

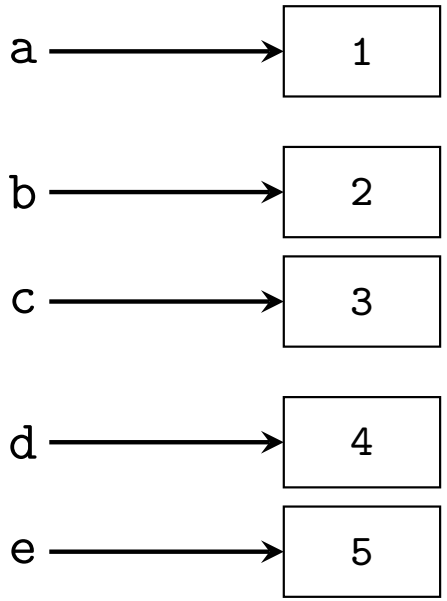


2026 Fall

Recall – Allocation Sites

Real Python, run `main()`:

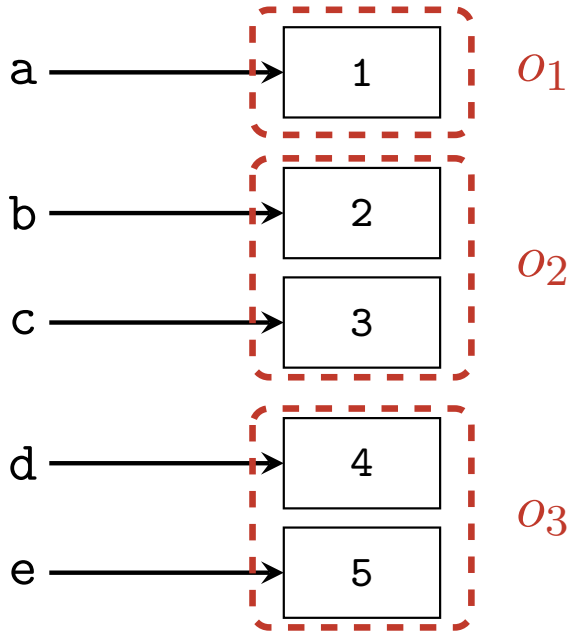
```
def f(x):
    t = [x]        # o2
    return t
def g(y):
    u = [y]        # o3
    return u
def main():
    a = [1]        # o1
    b = f(2)       # c1
    c = f(3)       # c2
    d = g(4)       # c3
    e = g(5)       # c4
```



Recall – Allocation Sites

Real Python, run `main()`:

```
def f(x):
    t = [x]         # o2
    return t
def g(y):
    u = [y]         # o3
    return u
def main():
    a = [1]         # o1
    b = f(2)        # c1
    c = f(3)        # c2
    d = g(4)        # c3
    e = g(5)        # c4
```



- One object per `[...]`: `b` and `c` are both o_2, so $\text{pt}(o_2[0]) = \{2, 3\}$. `b[0]` may be 3.

Contents

1. Crossing the Call

2. Context Sensitivity

3. Selective Context Sensitivity

4. Putting It Together

Contents

1. Crossing the Call

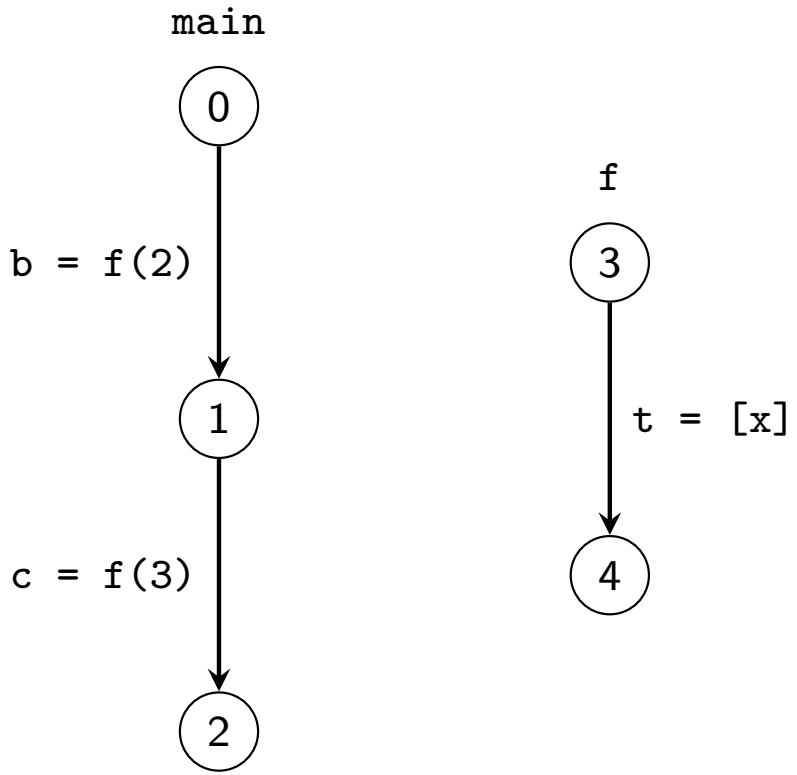
2. Context Sensitivity

3. Selective Context Sensitivity

4. Putting It Together

One Graph for the Program

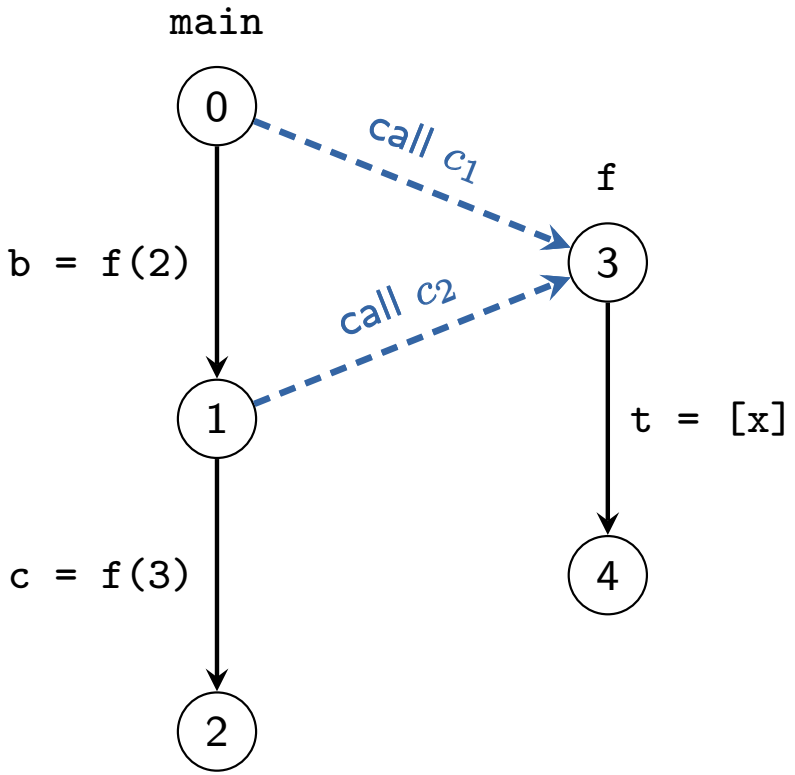
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



One CFG per function. In `main`, a call is one edge, as before.

One Graph for the Program

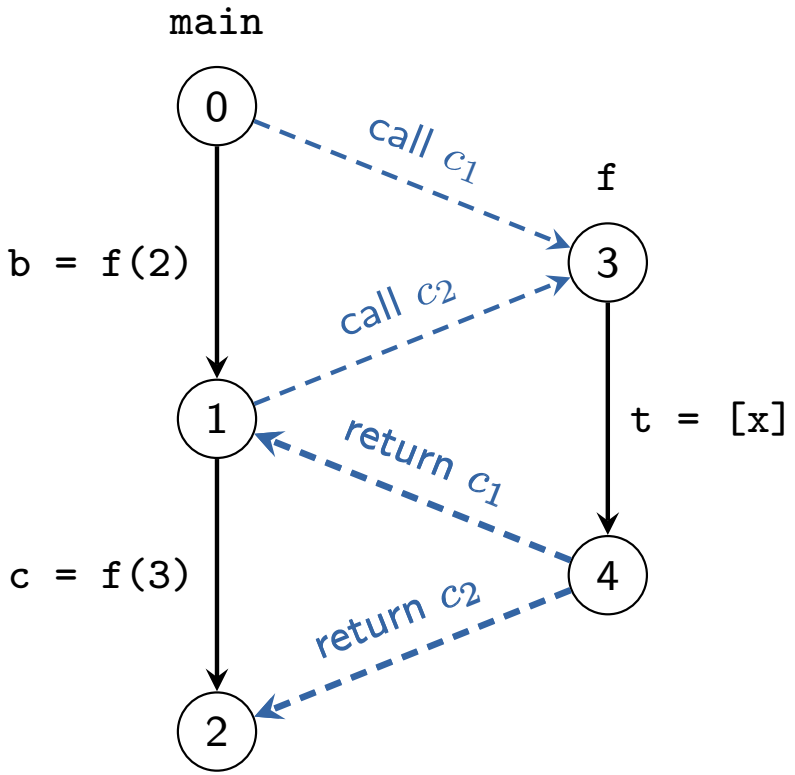
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



A **call edge** jumps from the call to the callee's entry, passing `x = 2` or `x = 3`.

One Graph for the Program

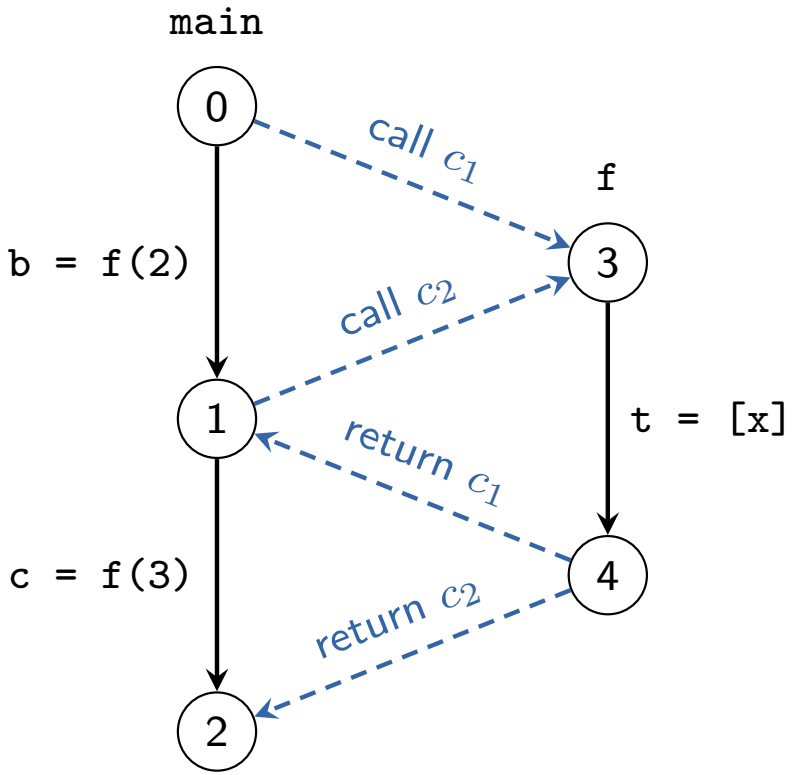
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



A **return edge** jumps from the callee's exit to the point after the call, passing `b = t` or `c = t`.

One Graph for the Program

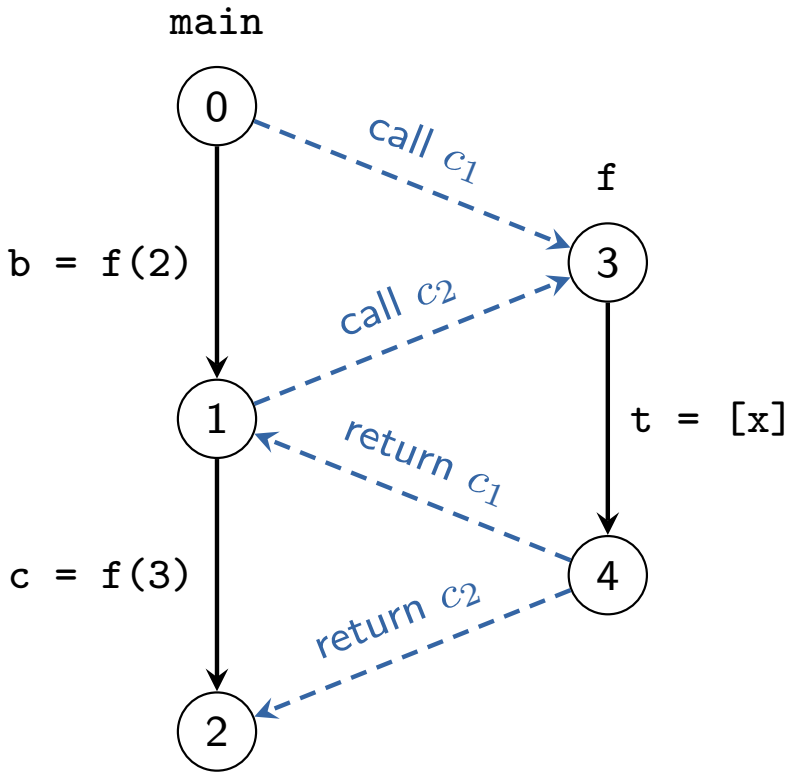
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Call edges go to the callee's entry with the argument;
return edges come back to the point after the call with the result.

Invalid Paths

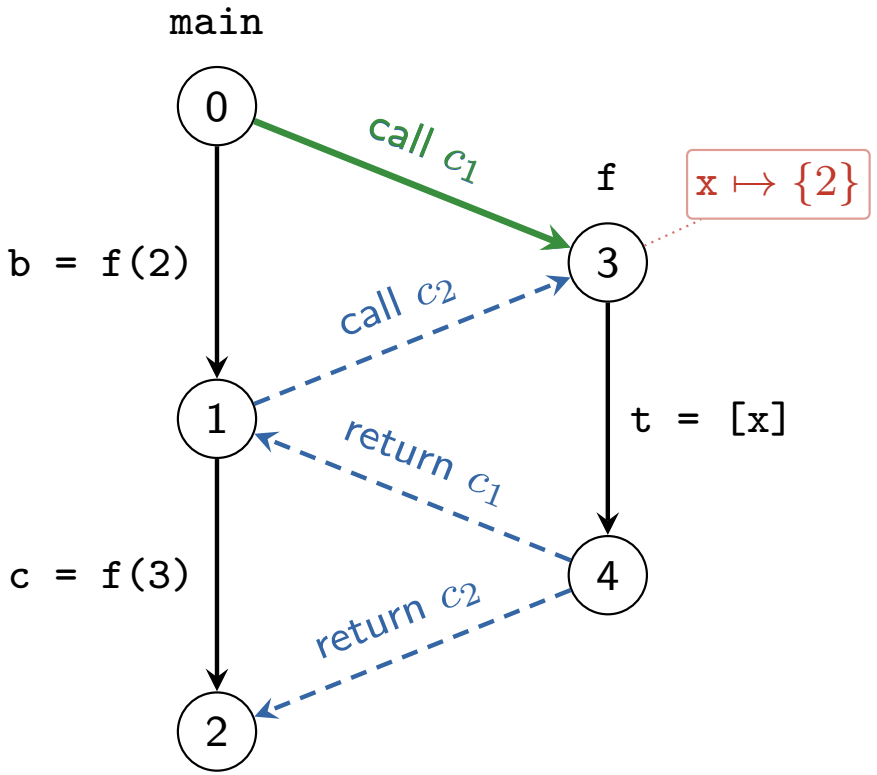
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Points-to at every point, **flow-sensitive**, heap included.

Invalid Paths

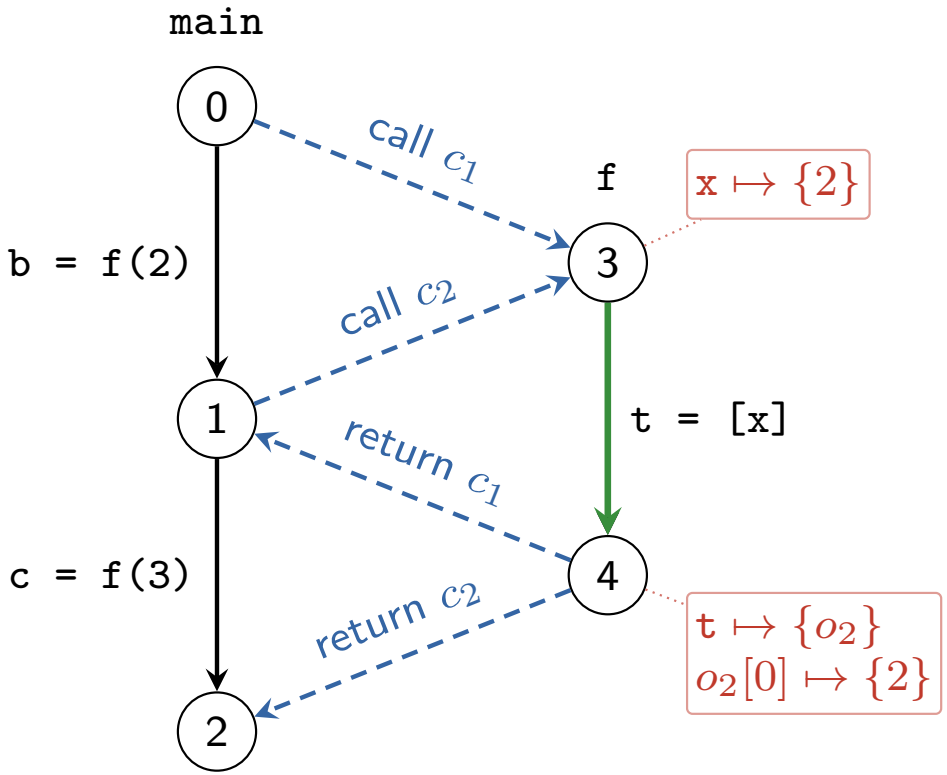
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Call c_1 passes x = 2.

Invalid Paths

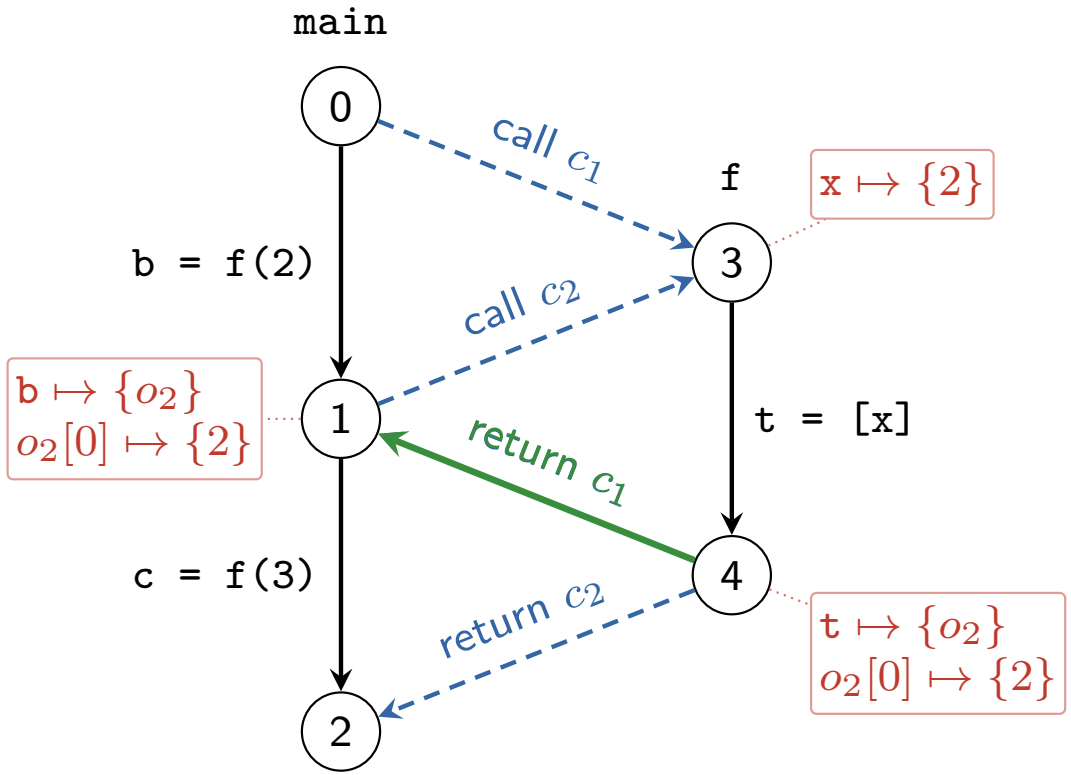
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



`t = [x]` allocates o_2 and stores `x` into it.

Invalid Paths

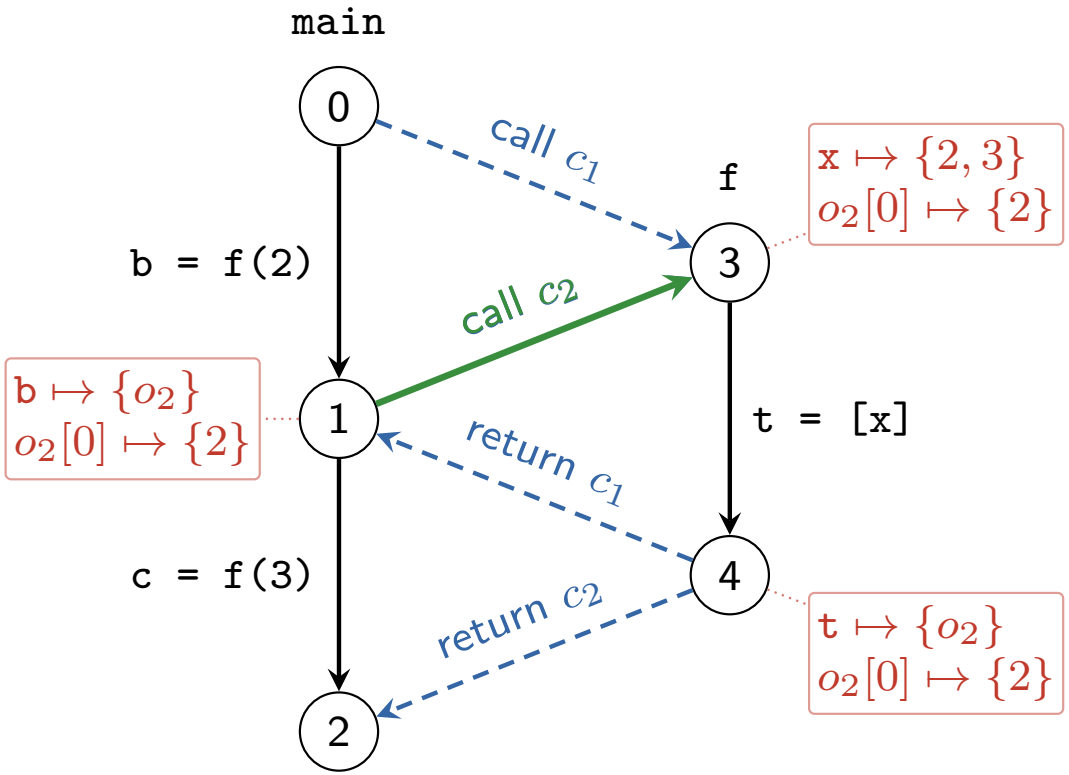
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Return c_1: `b = t`. At 1, `b[0]` is 2. Correct.

Invalid Paths

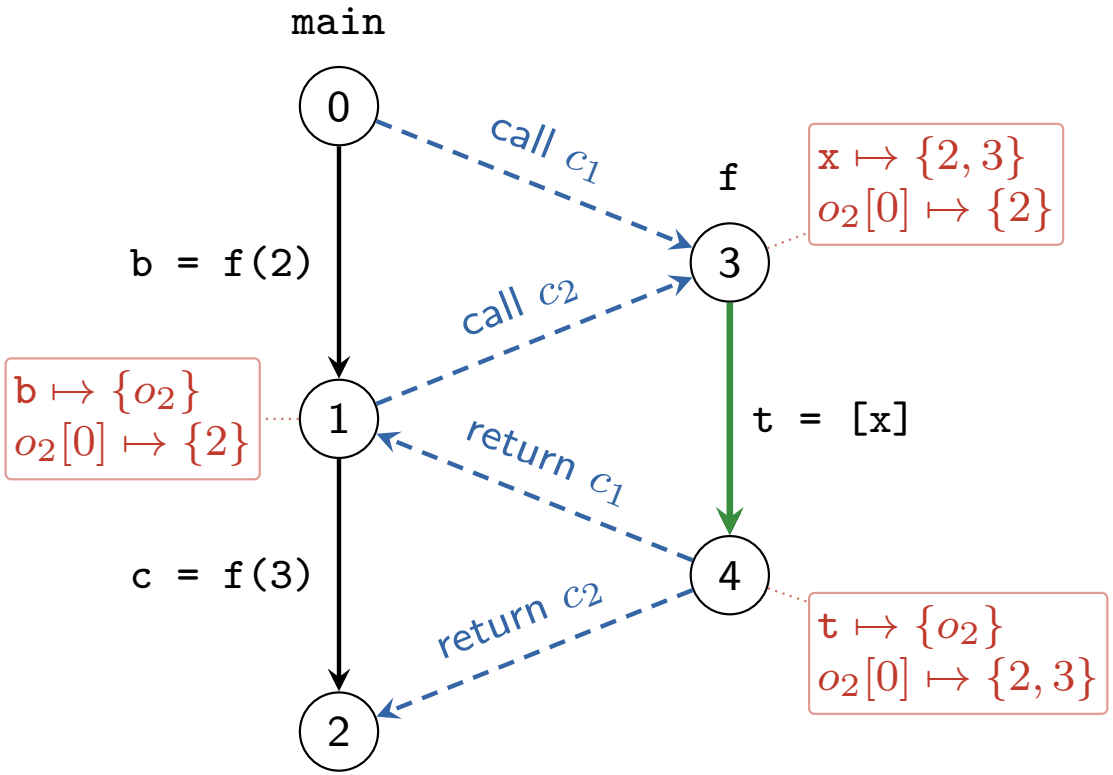
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Call c_2 passes `x = 3`. Point 3 is **shared**: the join keeps both.

Invalid Paths

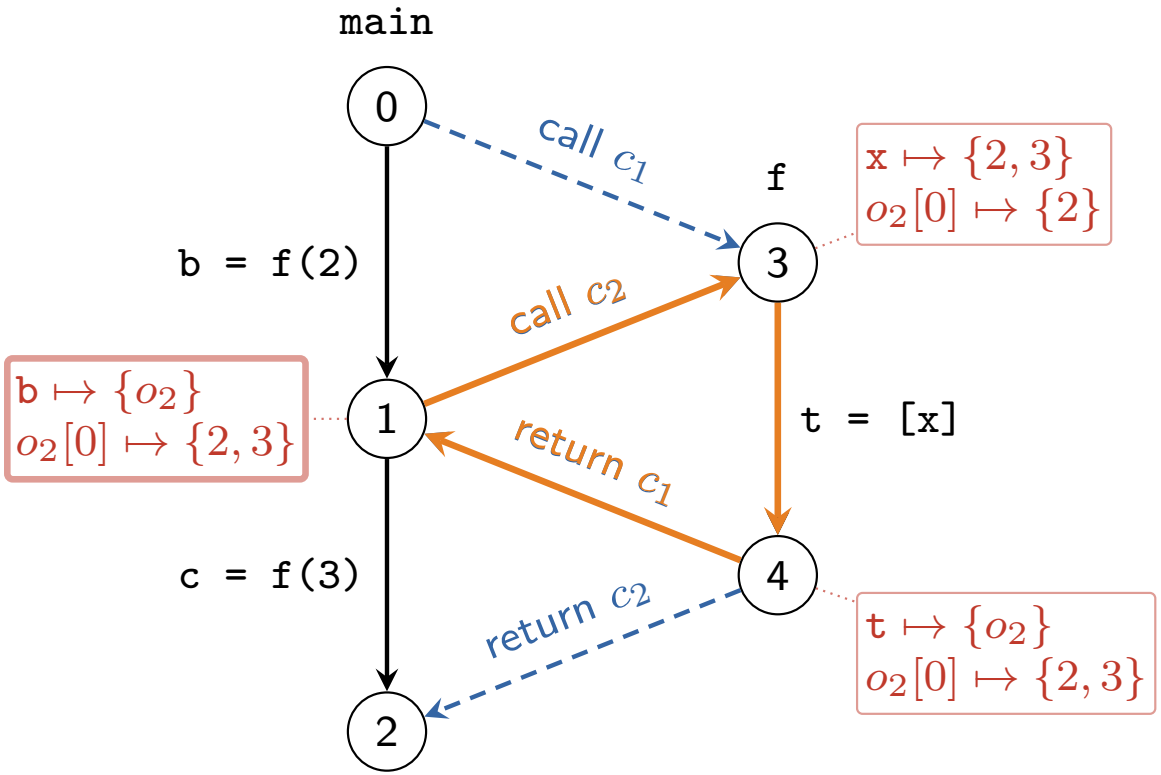
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



So at 4, $o_2[0] \mapsto \{2, 3\}$, and point 4 returns to **both** sites.

Invalid Paths

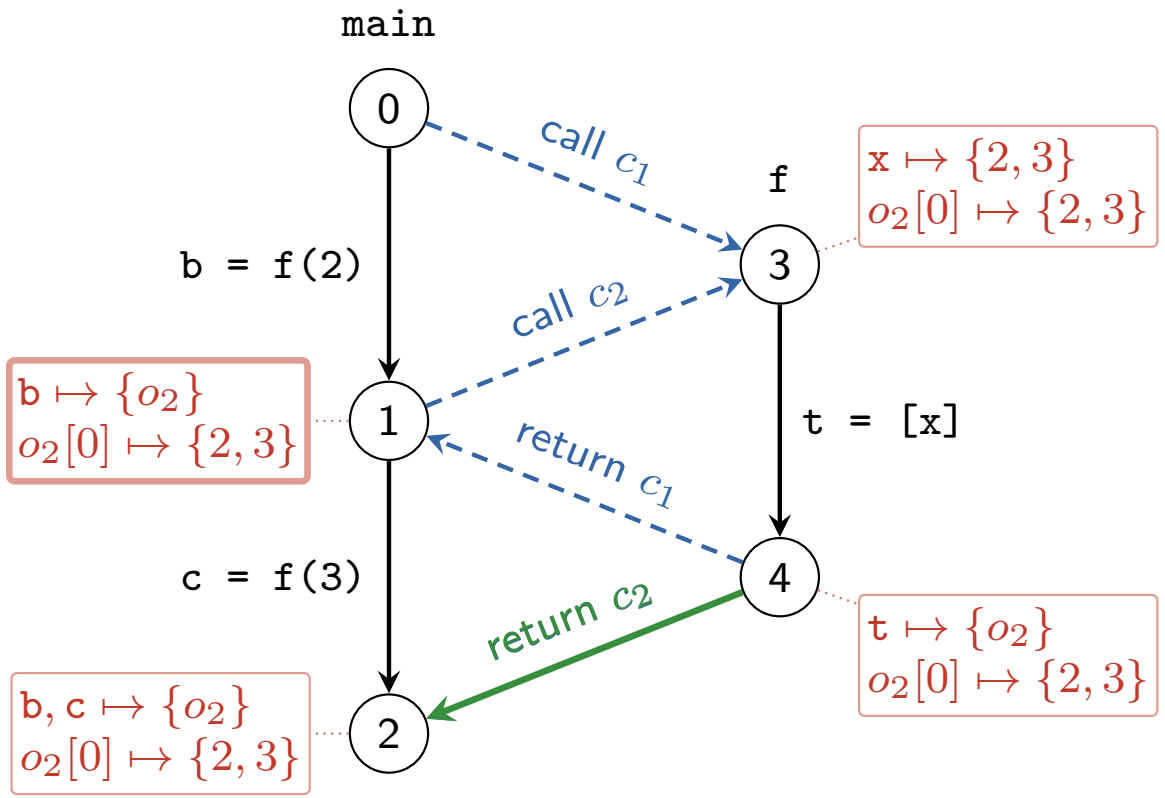
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



The 3 came in by call c_2 and goes out by return c_1:
an **invalid path**. At 1, b[0] may be 3.

Invalid Paths

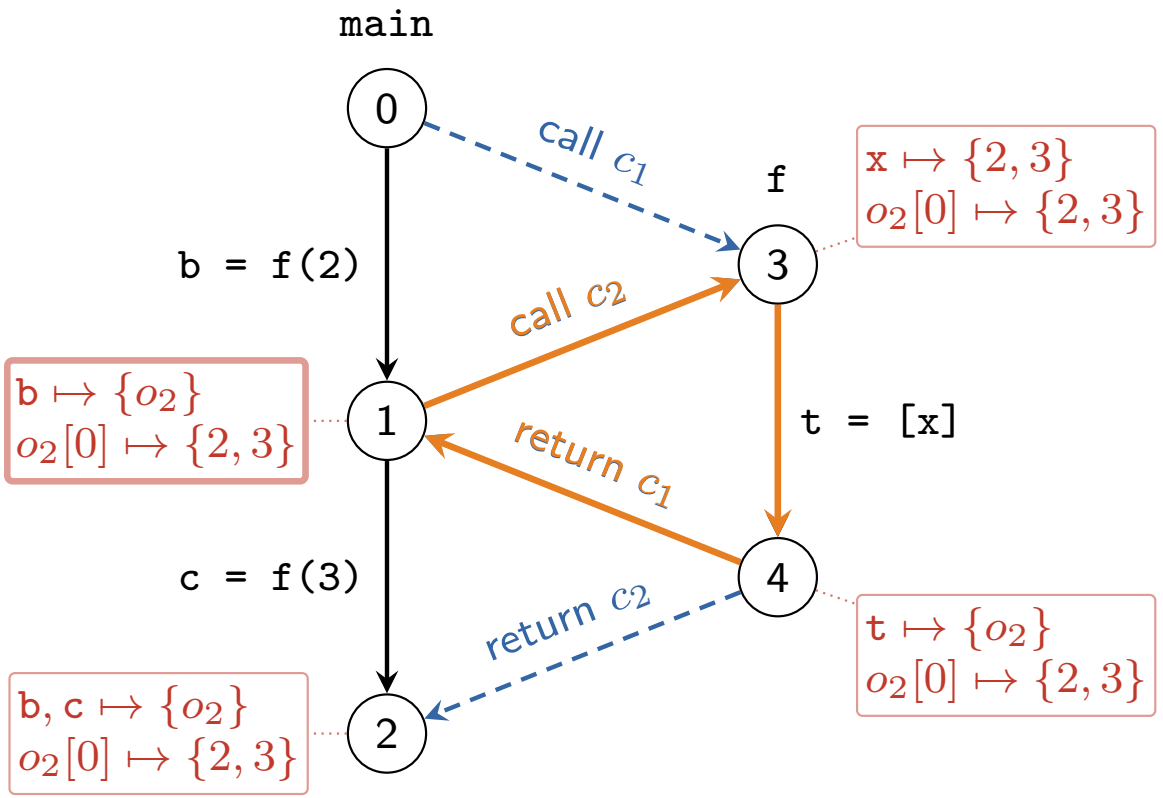
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Return c_2: `c = t`.

Invalid Paths

```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



Through the invalid path, `b[0]` may be 3 at 1. A return must go back to its own call: keep the calls apart, each in its own **context**.

Contents

1. Crossing the Call

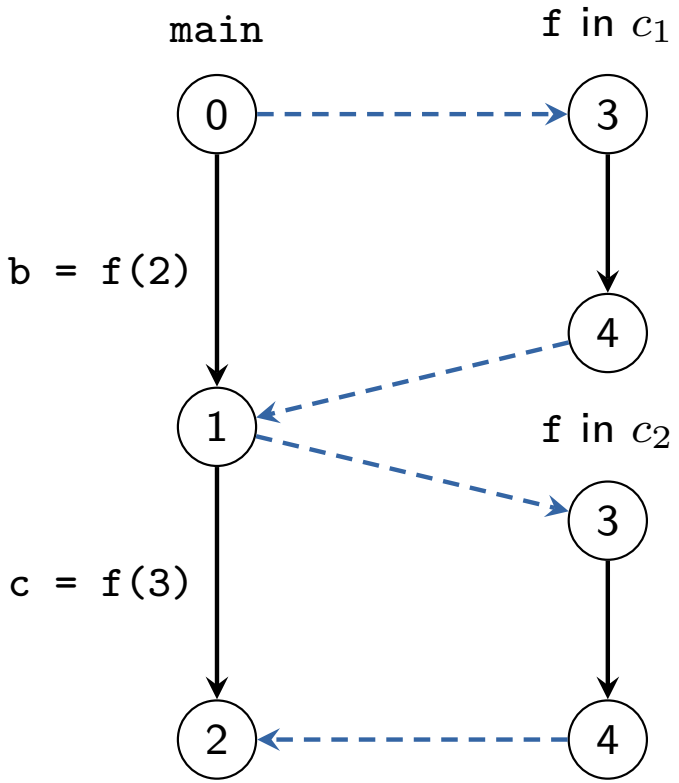
2. Context Sensitivity

3. Selective Context Sensitivity

4. Putting It Together

Call-Site Sensitivity

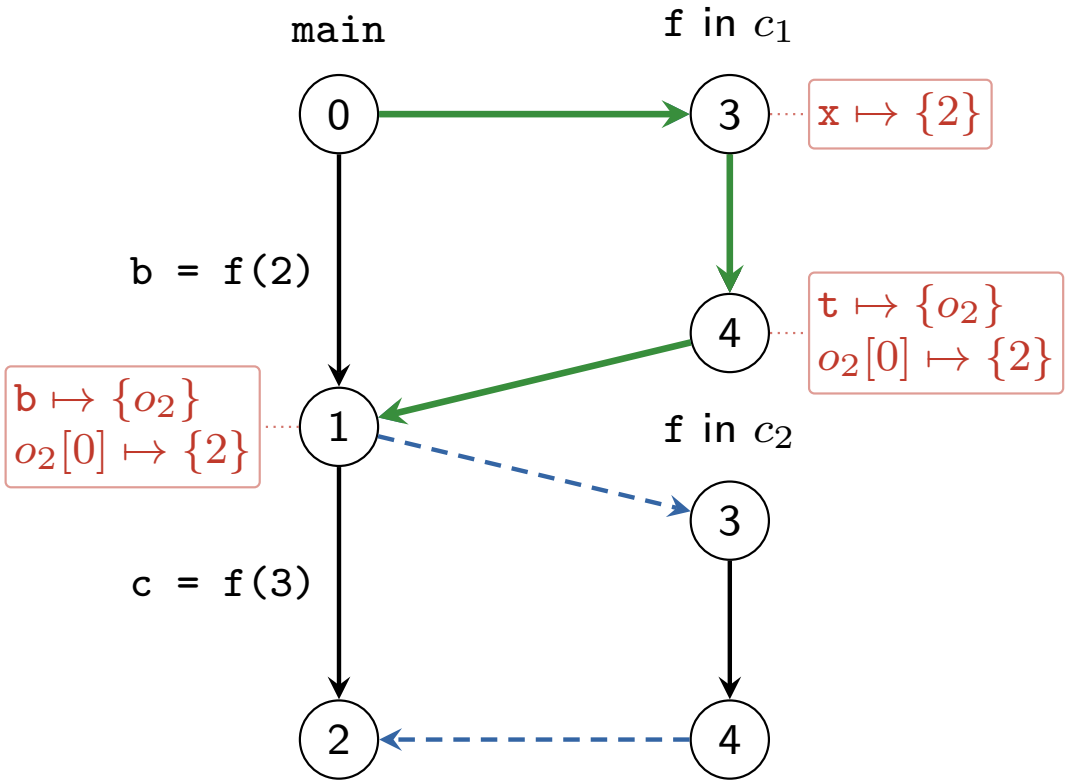
```
def f(x):
    t = [x]        # o2
    return t
def g(y):
    u = [y]        # o3
    return u
def main():
    a = [1]        # o1
    b = f(2)       # c1
    c = f(3)       # c2
    d = g(4)       # c3
    e = g(5)       # c4
```



Analyze f once per **call site**. A fact now belongs to a point **and** a context.

Call-Site Sensitivity

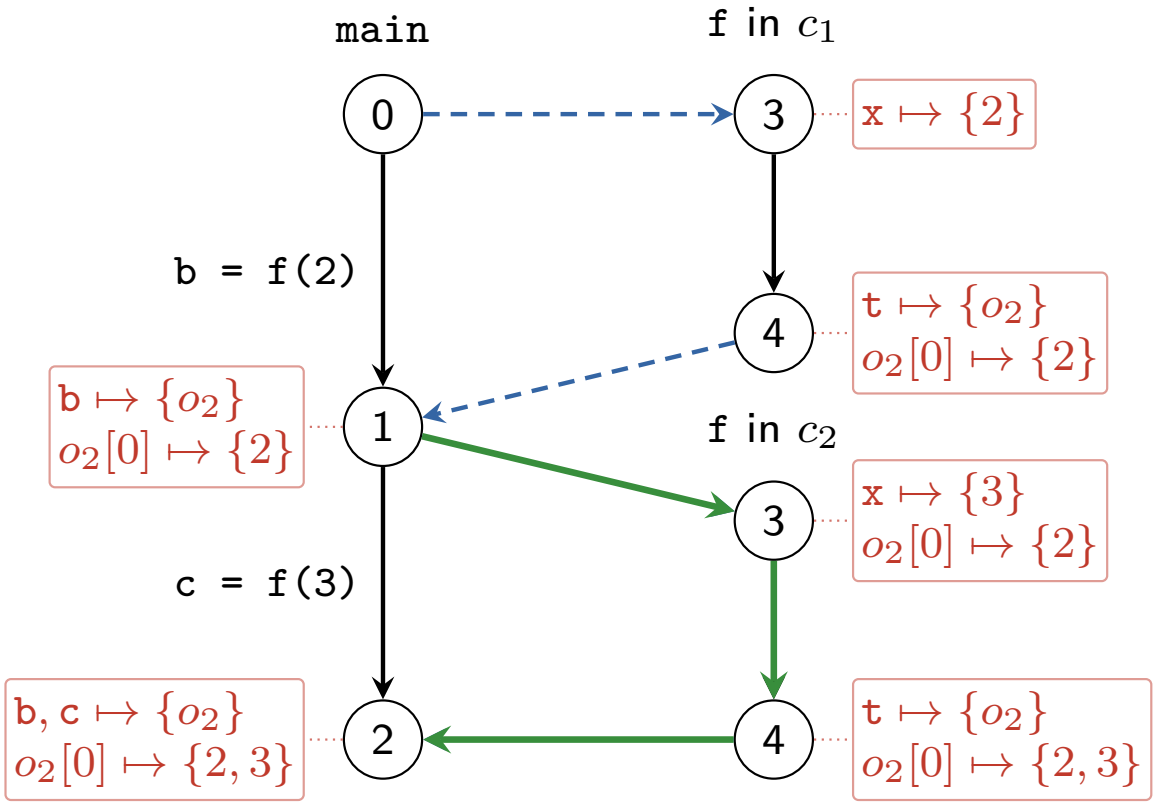
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



In context c_1: x $\mapsto \{2\}$, back to 1 only. At 1, b[0] is 2.

Call-Site Sensitivity

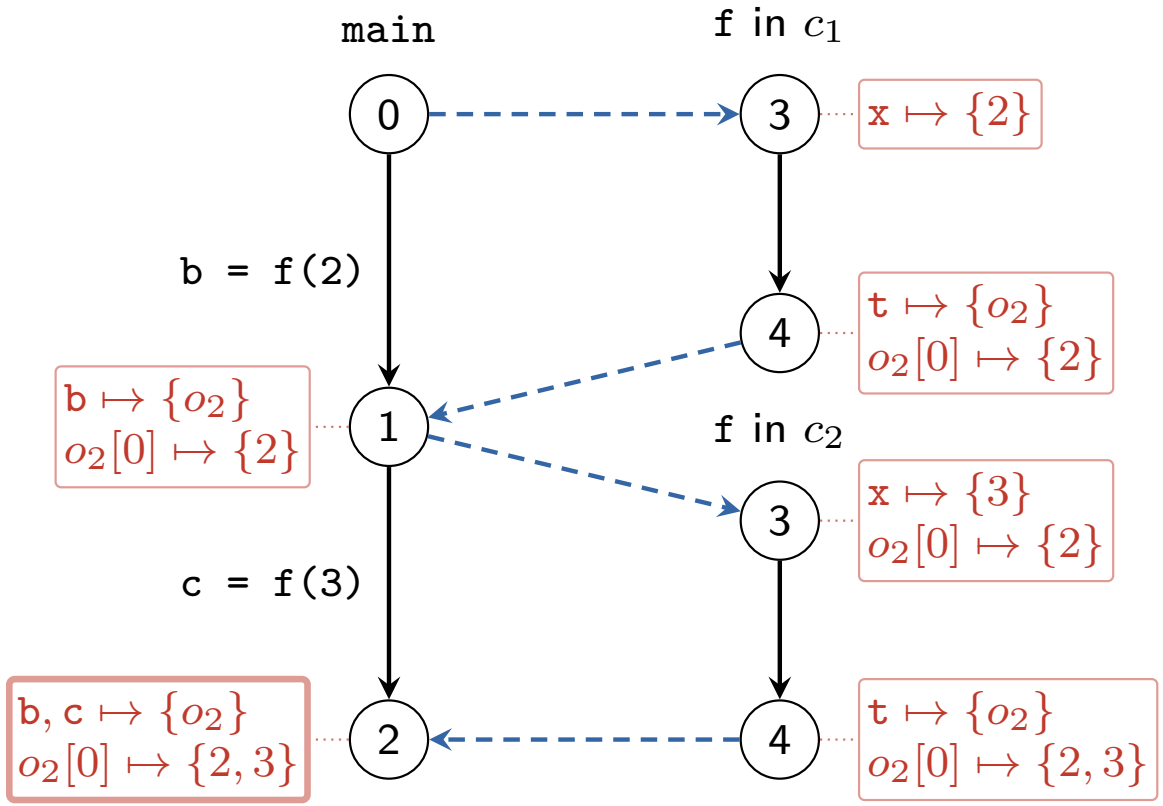
```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



In context c_2: x $\mapsto \{3\}$, back to 2 only. No more invalid path.

Call-Site Sensitivity

```
def f(x):
    t = [x]          # o2
    return t
def g(y):
    u = [y]          # o3
    return u
def main():
    a = [1]          # o1
    b = f(2)         # c1
    c = f(3)         # c2
    d = g(4)         # c3
    e = g(5)         # c4
```



But `t = [x]` makes o_2 again, the same object as `b`'s list:
at 2, `b[0]` may still be 3.

Heap Cloning

Real Python:

```
def f(x):
    t = [x]       # o2
    return t
def g(y):
    u = [y]       # o3
    return u
def main():
    a = [1]       # o1
    b = f(2)      # c1
    c = f(3)      # c2
    d = g(4)      # c3
    e = g(5)      # c4
```

object	$o[0]$
o_1	$\{1\}$
o_2	$\{2, 3\}$
o_3	$\{4, 5\}$

Contexts split `x`, but objects are still one per `[...]`.

Heap Cloning

Real Python:

```
def f(x):
    t = [x]        # o2
    return t
def g(y):
    u = [y]        # o3
    return u
def main():
    a = [1]        # o1
    b = f(2)       # c1
    c = f(3)       # c2
    d = g(4)       # c3
    e = g(5)       # c4
```

object	$o[0]$
o_1	$\{1\}$
o_2	$\{2, 3\}$
o_3	$\{4, 5\}$

b and c are two lists from one site: o_2 holds both 2 and 3.

Heap Cloning

Real Python:

```
def f(x):
    t = [x]        # o2
    return t
def g(y):
    u = [y]        # o3
    return u
def main():
    a = [1]        # o1
    b = f(2)       # c1
    c = f(3)       # c2
    d = g(4)       # c3
    e = g(5)       # c4
```

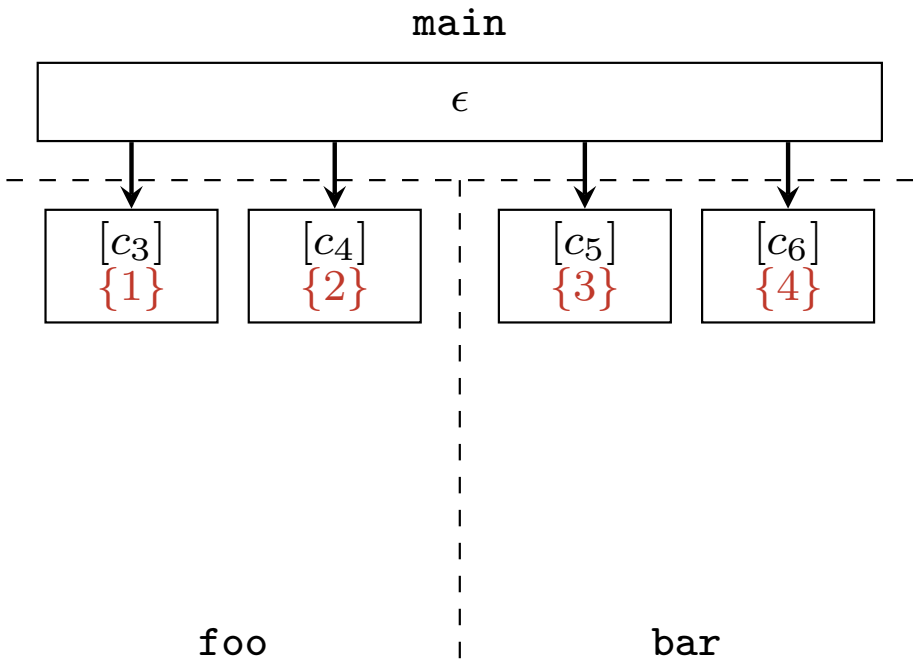
object	$o[0]$
(o_1, ϵ)	$\{1\}$
(o_2, c_1)	$\{2\}$
(o_2, c_2)	$\{3\}$
(o_3, c_3)	$\{4\}$
(o_3, c_4)	$\{5\}$

Heap cloning: name an object by its site **and** the context that ran it. Every list is apart.

Recursion

Real Python:

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```

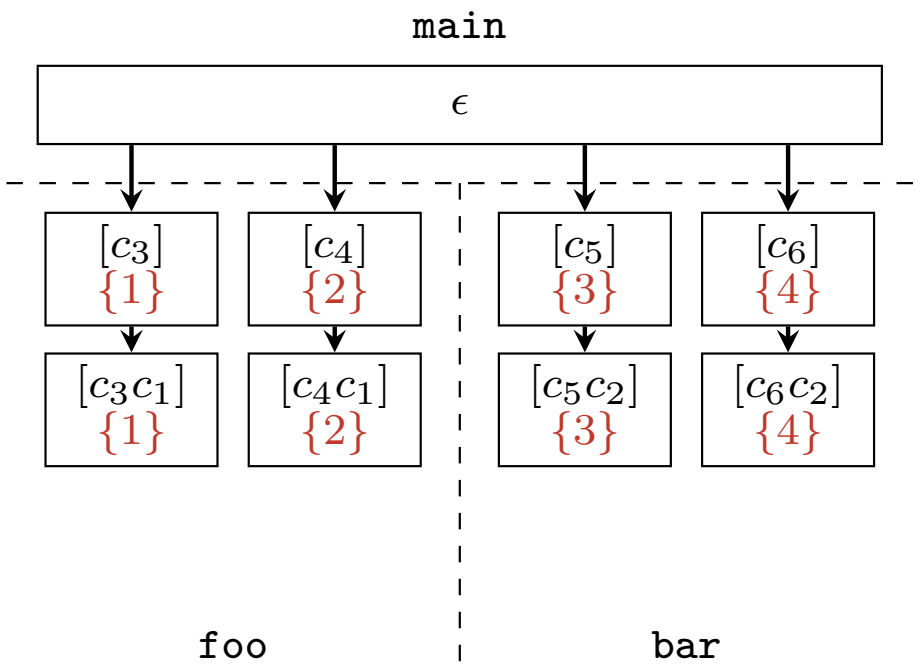


`main` calls `foo` at c_3, c_4 and `bar` at c_5, c_6: one context per call site.

Recursion

Real Python:

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```

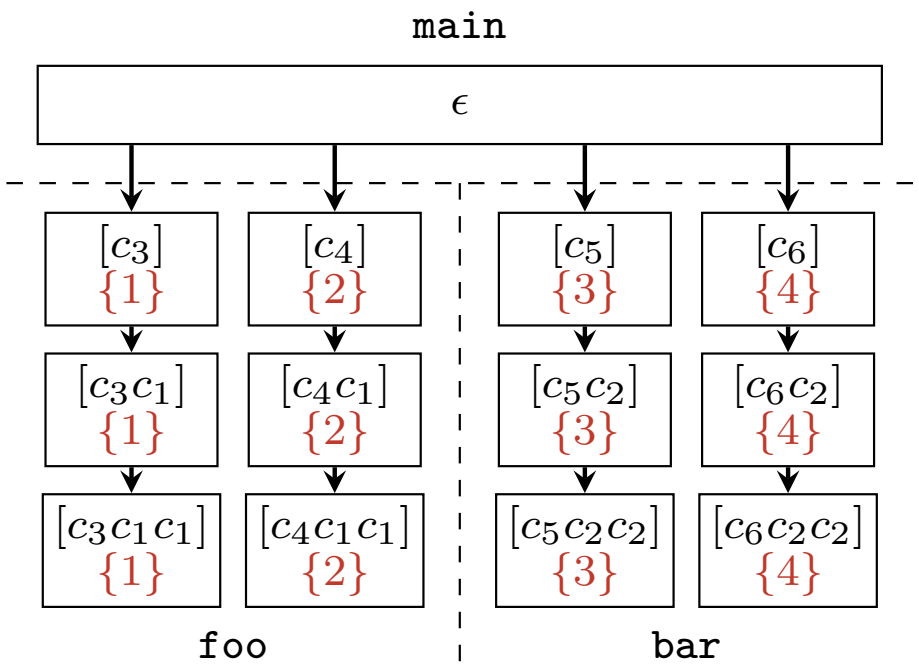


A recursive call adds its site: $[c_3c_1]$, $[c_4c_1]$, $[c_5c_2]$, $[c_6c_2]$.

Recursion

Real Python:

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```

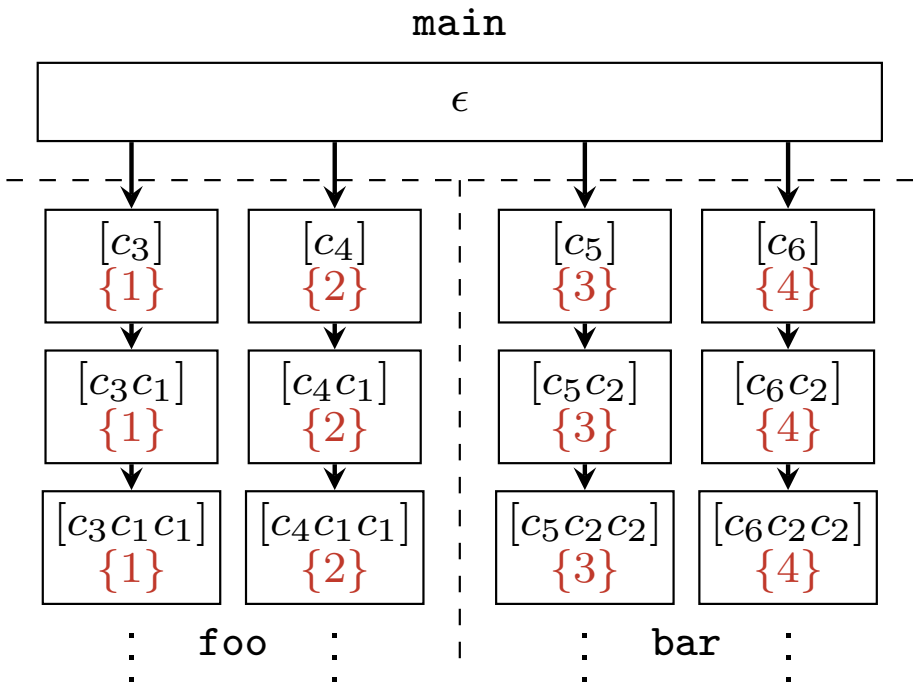


And again, one level deeper. Every value is still exact.

Recursion

Real Python:

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```

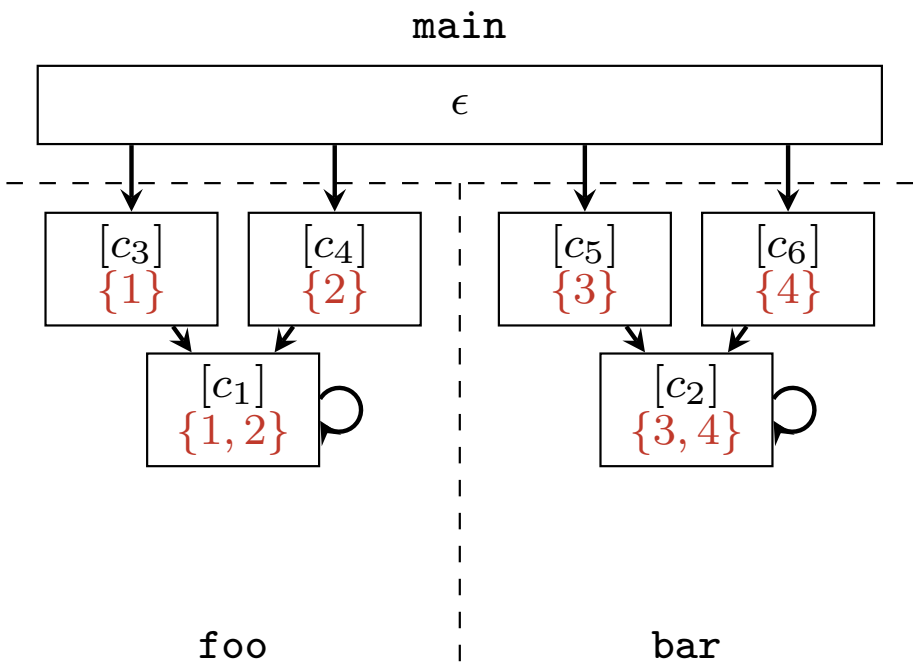


As deep as `input()` allows: unbounded. The fixpoint never stops.

1-Call-Site Sensitivity

Real Python:

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```



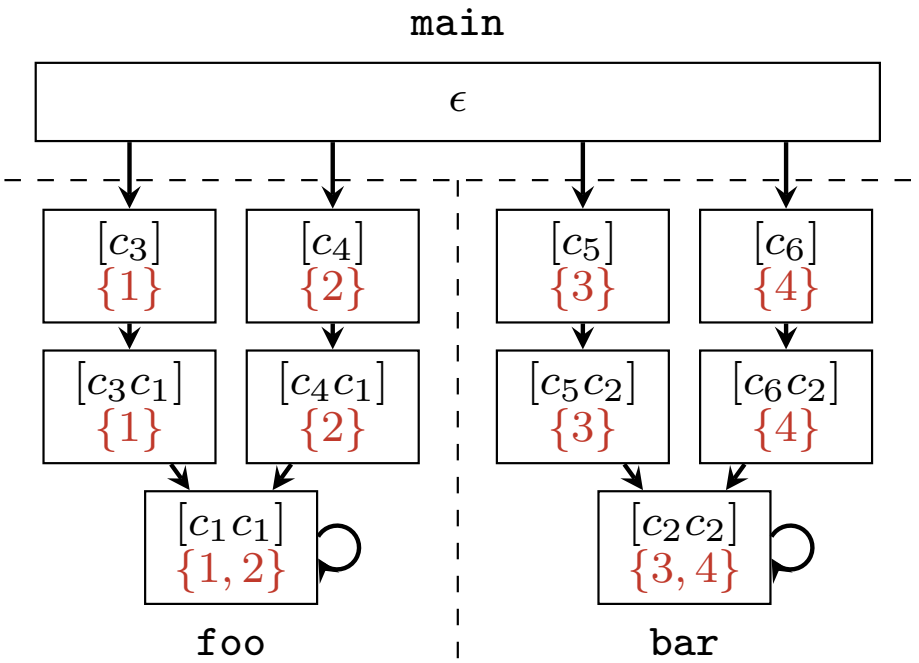
k-call-site sensitivity (k-CFA): keep the last k call sites.

$k = 1$: every recursive call of `foo` is $[c_1]$, so `x` $\mapsto \{1, 2\}$.

2-Call-Site Sensitivity

Real Python:

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```



$k = 2$: one more level is exact. From the third on, x still merges.
A larger k only goes one level deeper: no k is enough.

The Cost

A helper called from 50 places:

k	contexts of the helper
0	1

The Cost

A helper called from 50 places:

k	contexts of the helper
0	1
1	50

The Cost

A helper called from 50 places:

k	contexts of the helper
0	1
1	50
2	$2{,}500$
3	$125{,}000$

- Up to $|\text{Sites}|^k$ copies, each with its own facts, and the calls inside each copy multiply again.

The Cost

A helper called from 50 places:

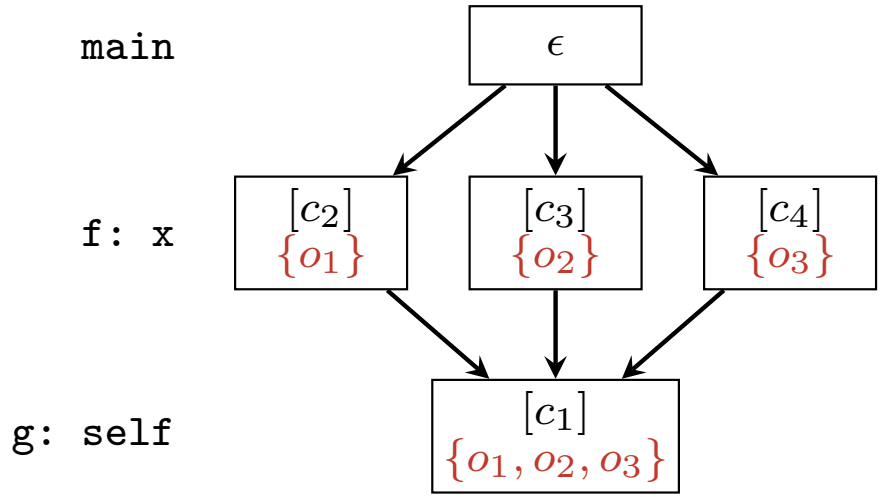
k	contexts of the helper
0	1
1	50
2	$2{,}500$
3	$125{,}000$

- Up to $|\text{Sites}|^k$ copies, each with its own facts, and the calls inside each copy multiply again.
- Most copies are never reached, or equal. Still, tools stop at $k = 1$ or 2.

Call Sites for Methods

Real Python:

```
class A:
    def f(self, x):
        x.g()              # c1
    def g(self):
        pass
class B(A):
    pass
def main():
    a = A()                # o1
    a.f(a)                 # c2
    a.f(A())               # c3, o2
    a.f(B())               # c4, o3
```

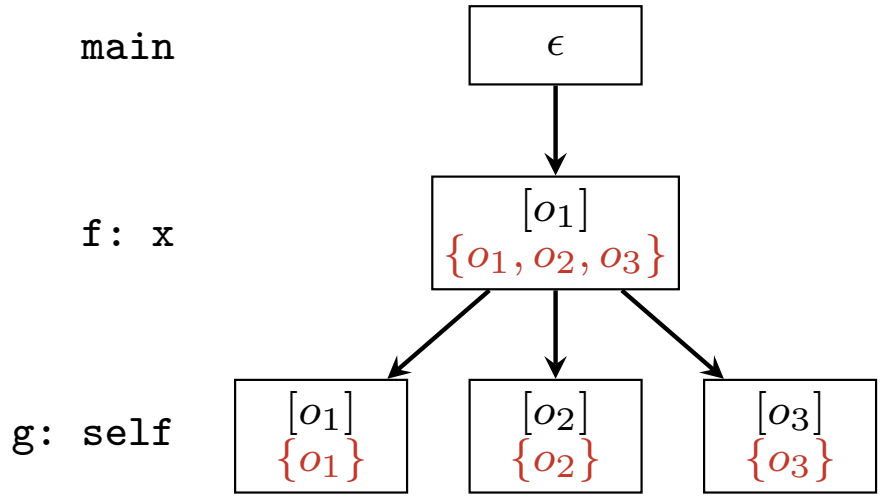


Without contexts, both are $\{o_1, o_2, o_3\}$. **Call sites**: `f` splits three ways, so `x` is exact. `g` has one site c_1: `self` still $\{o_1, o_2, o_3\}$.

Object Sensitivity

Real Python:

```
class A:
    def f(self, x):
        x.g()              # c1
    def g(self):
        pass
class B(A):
    pass
def main():
    a = A()                # o1
    a.f(a)                 # c2
    a.f(A())               # c3, o2
    a.f(B())               # c4, o3
```

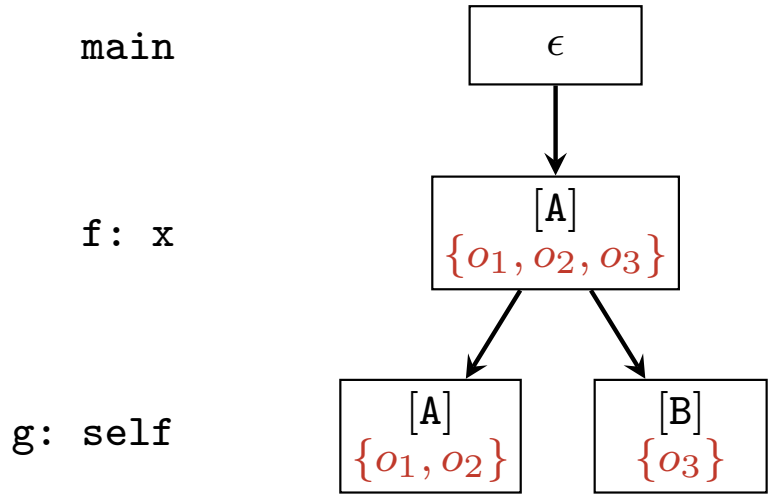


Objects: the receiver. `f` always runs on o_1, so `x` merges. `g` runs on o_1, o_2, o_3: `self` is exact.

Type Sensitivity

Real Python:

```
class A:
    def f(self, x):
        x.g()              # c1
    def g(self):
        pass
class B(A):
    pass
def main():
    a = A()                # o1
    a.f(a)                 # c2
    a.f(A())               # c3, o2
    a.f(B())               # c4, o3
```



Types: the receiver's class. g runs in A (o_1, o_2) and B (o_3). Fewer contexts, a little less precise. No kind wins everywhere: call sites suit f, objects suit g.

Contents

1. Crossing the Call

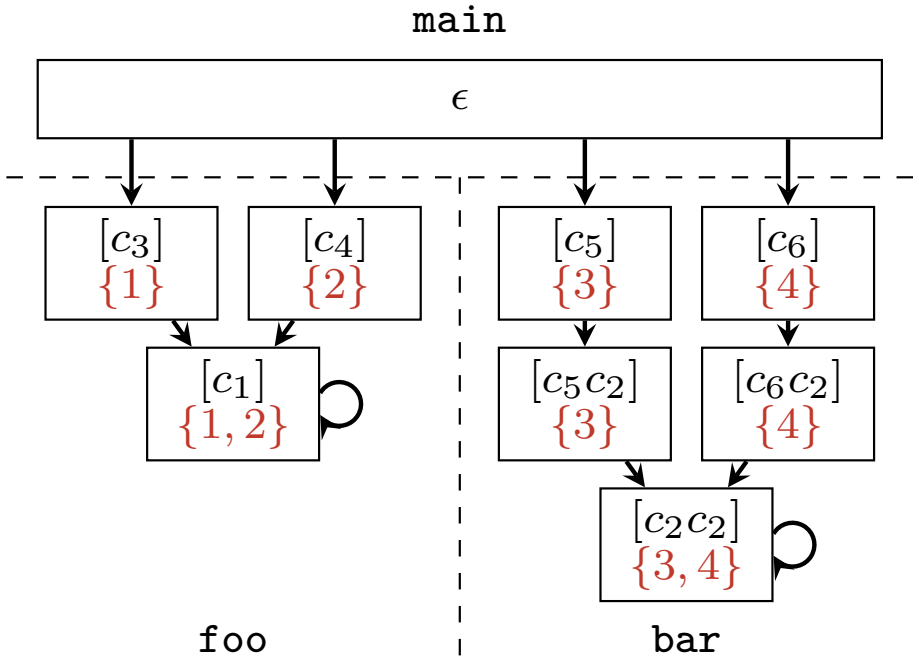
2. Context Sensitivity

3. Selective Context Sensitivity

4. Putting It Together

A k per Function

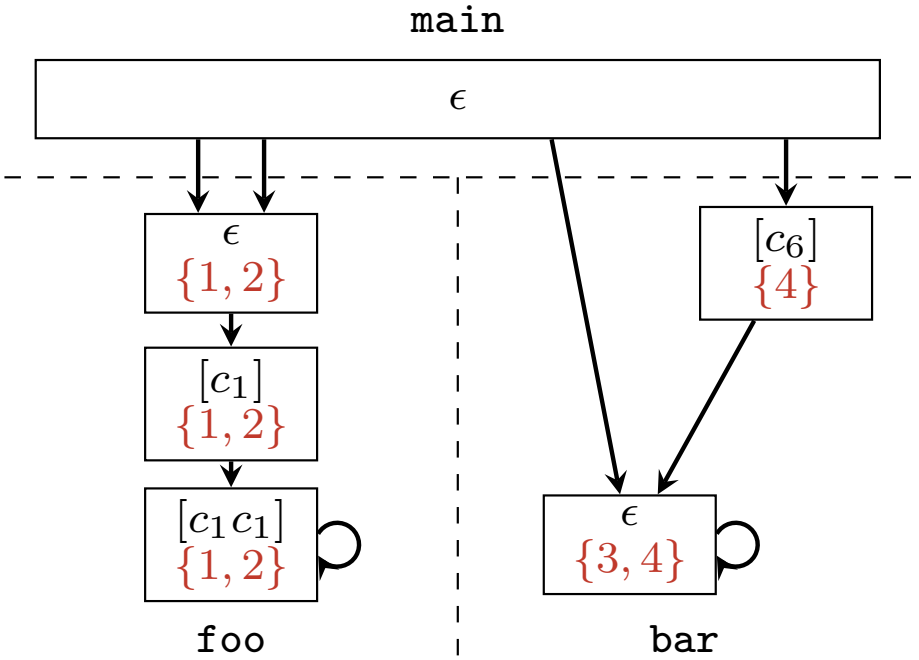
```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```



A k per function: $k = 1$ for `foo`, $k = 2$ for `bar`. `foo` loses one exact level, and two contexts.

Selected Calls

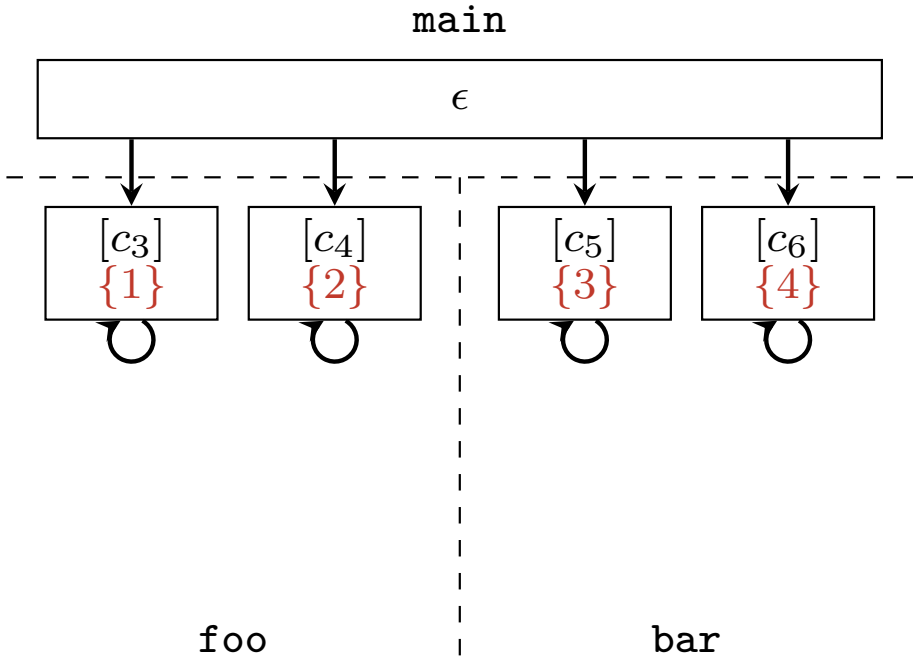
```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```



Selected calls: only $(c_1, \texttt{foo})$ and $(c_6, \texttt{bar})$ add their site; c_2 gives ϵ.
Five contexts, but `foo` is $\{1,2\}$ everywhere: selecting c_1 bought nothing.

Context Tunneling

```
def foo(x):
    if input():
        foo(x)          # c1
def bar(y):
    if input():
        bar(y)          # c2
def main():
    foo(1)              # c3
    foo(2)              # c4
    bar(3)              # c5
    bar(4)              # c6
```



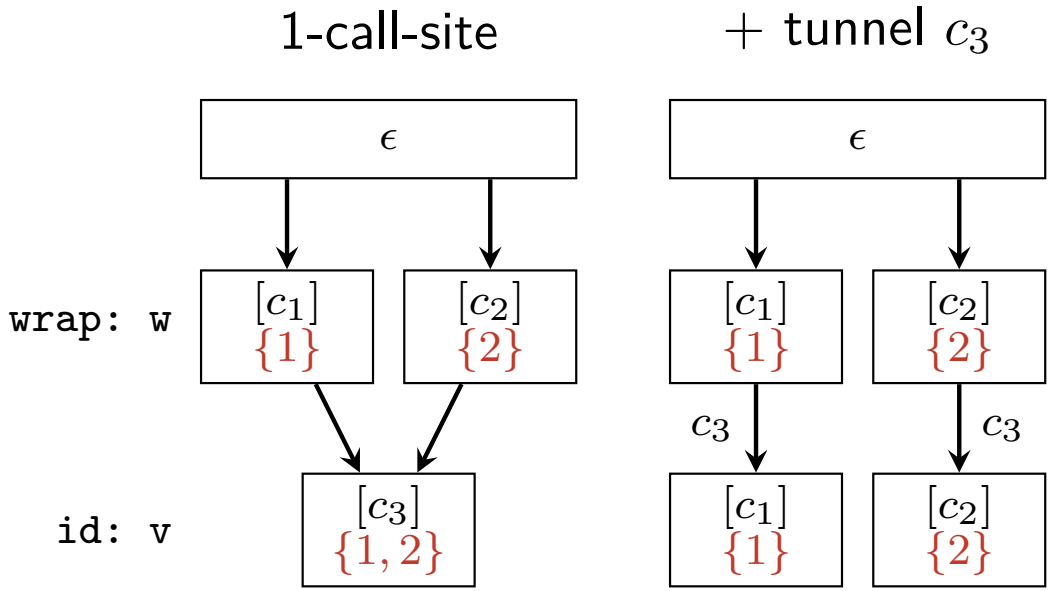
Keep $c_3, \ldots, c_6$ as context sites; make the recursive calls c_1, c_2 **tunnels**, which pass the caller's context through. With $k = 1$: four contexts, every value exact.

A context is the last k **selected** call sites. The others neither add nor erase.

Tunneling Through Wrappers

Real Python:

```
def id(v):
    return v
def wrap(w):
    return id(w)    # c3
def main():
    a = wrap(1)     # c1
    b = wrap(2)     # c2
```



A helper called through a wrapper: with $k = 1$, the wrapper's own call c_3 uses up the one site, and $\mathtt{v} \mapsto \{1, 2\}$. Make c_3 a tunnel: `id` runs in $[c_1]$ and $[c_2]$, exact, with no larger k.

Selective Heap Cloning

```
def f(x):
    t = [x]        # o2
    return t
def g(y):
    u = [y]        # o3
    return u
def main():
    a = [1]        # o1
    b = f(2)       # c1
    c = f(3)       # c2
    d = g(4)       # c3
    e = g(5)       # c4
```

object	$o[0]$
o_1	$\{1\}$
o_2	$\{2, 3\}$
(o_3, c_3)	$\{4\}$
(o_3, c_4)	$\{5\}$

- Clone only the sites that need it. Here only o_3: `d` and `e` are apart, `b` and `c` share o_2.
- The same choice as for calls: which sites, and how deep.

How to Choose

The analysis stays the same. Only its **parameters** change, per program: which calls get contexts, of which kind, how deep, which sites are cloned.

How to Choose

The analysis stays the same. Only its **parameters** change, per program: which calls get contexts, of which kind, how deep, which sites are cloned.

- **By syntax**: call sites for plain functions, objects for methods.

How to Choose

The analysis stays the same. Only its **parameters** change, per program: which calls get contexts, of which kind, how deep, which sites are cloned.

- **By syntax**: call sites for plain functions, objects for methods.
- **By a pre-analysis**: a cheap analysis first finds the calls where precision would pay. The same idea as octagon packs, earlier.

How to Choose

The analysis stays the same. Only its **parameters** change, per program: which calls get contexts, of which kind, how deep, which sites are cloned.

- **By syntax**: call sites for plain functions, objects for methods.
- **By a pre-analysis**: a cheap analysis first finds the calls where precision would pay. The same idea as octagon packs, earlier.
- **By learning**: train on many programs which calls need a context, which k, where to tunnel.

How to Choose

The analysis stays the same. Only its **parameters** change, per program: which calls get contexts, of which kind, how deep, which sites are cloned.

- **By syntax**: call sites for plain functions, objects for methods.
- **By a pre-analysis**: a cheap analysis first finds the calls where precision would pay. The same idea as octagon packs, earlier.
- **By learning**: train on many programs which calls need a context, which k, where to tunnel.
- **By the alarms**: run cheap, then refine only where a false alarm came from.

How to Choose

The analysis stays the same. Only its **parameters** change, per program: which calls get contexts, of which kind, how deep, which sites are cloned.

- **By syntax**: call sites for plain functions, objects for methods.
- **By a pre-analysis**: a cheap analysis first finds the calls where precision would pay. The same idea as octagon packs, earlier.
- **By learning**: train on many programs which calls need a context, which k, where to tunnel.
- **By the alarms**: run cheap, then refine only where a false alarm came from.

Parametric static analysis. Survey: Park, Lee, and Ryu, ACM CSUR 2021.

Contents

1. Crossing the Call

2. Context Sensitivity

3. Selective Context Sensitivity

4. Putting It Together

Which Function?

Real Python:

```
HANDLERS = {"show": show, "delete": delete}
def handle(name, req):
    h = HANDLERS[name]
    return h(req)              # show? delete?
```

Which Function?

Real Python:

```
HANDLERS = {"show": show, "delete": delete}
def handle(name, req):
    h = HANDLERS[name]
    return h(req)              # show? delete?
```

- Everything today assumed we know the callee. Here it is whatever `h` **points to**.

Which Function?

Real Python:

```
HANDLERS = {"show": show, "delete": delete}
def handle(name, req):
    h = HANDLERS[name]
    return h(req)              # show? delete?
```

- Everything today assumed we know the callee. Here it is whatever `h` **points to**.
- The **call graph** needs pointer analysis, and pointer analysis across calls needs the call graph.

Which Function?

Real Python:

```
HANDLERS = {"show": show, "delete": delete}
def handle(name, req):
    h = HANDLERS[name]
    return h(req)              # show? delete?
```

- Everything today assumed we know the callee. Here it is whatever `h` **points to**.
- The **call graph** needs pointer analysis, and pointer analysis across calls needs the call graph.
- Real tools compute both **together**, in one fixpoint: the most expensive part of a whole-program analysis.

Four Kinds of Sensitivity

Each keeps apart what a cheaper analysis would merge:

sensitivity	**keeps apart**	**example**
flow	program points	dataflow yes, Andersen no

Four Kinds of Sensitivity

Each keeps apart what a cheaper analysis would merge:

sensitivity	**keeps apart**	**example**
flow	program points	dataflow yes, Andersen no
field	parts of one object	`xs[0]` and `xs[1]`

Four Kinds of Sensitivity

Each keeps apart what a cheaper analysis would merge:

sensitivity	**keeps apart**	**example**
flow	program points	dataflow yes, Andersen no
field	parts of one object	`xs[0]` and `xs[1]`
context	calls of one function	c_1 and c_2, o_1 and o_2

Four Kinds of Sensitivity

Each keeps apart what a cheaper analysis would merge:

sensitivity	**keeps apart**	**example**
flow	program points	dataflow yes, Andersen no
field	parts of one object	`xs[0]` and `xs[1]`
context	calls of one function	c_1 and c_2, o_1 and o_2
path	branches of one `if`	symbolic execution, later

Four Kinds of Sensitivity

Each keeps apart what a cheaper analysis would merge:

sensitivity	**keeps apart**	**example**
flow	program points	dataflow yes, Andersen no
field	parts of one object	`xs[0]` and `xs[1]`
context	calls of one function	c_1 and c_2, o_1 and o_2
path	branches of one `if`	symbolic execution, later

- Each buys precision with cost, and the costs **multiply**.

Four Kinds of Sensitivity

Each keeps apart what a cheaper analysis would merge:

sensitivity	**keeps apart**	**example**
flow	program points	dataflow yes, Andersen no
field	parts of one object	`xs[0]` and `xs[1]`
context	calls of one function	c_1 and c_2, o_1 and o_2
path	branches of one `if`	symbolic execution, later

- Each buys precision with cost, and the costs **multiply**.
- Doop's `-a 2-object-sensitive+heap`: objects as contexts, $k = 2$, with heap cloning.

Summary

1. Crossing the Call
2. Context Sensitivity
3. Selective Context Sensitivity
4. Putting It Together

Next Lecture

- Constraint-based Analysis

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr