Static Analysis
and
Dataflow Analysis

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?

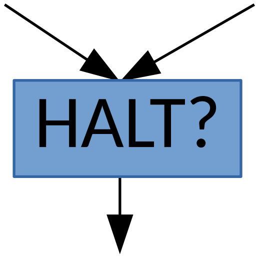
Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?
  - Does my program terminate?

Static Analysis

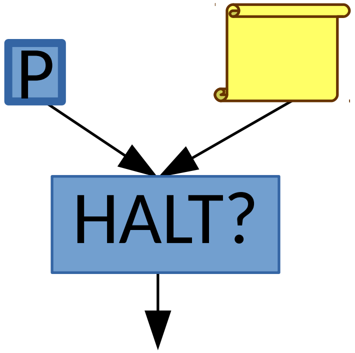
Brief Review of Undecidability



HALT? “Does my program terminate?”

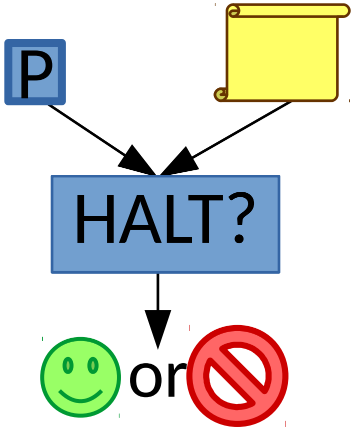
Static Analysis

Brief Review of Undecidability



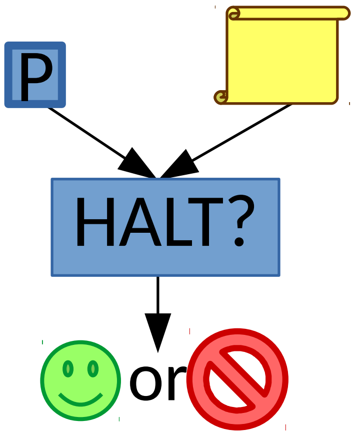
Static Analysis

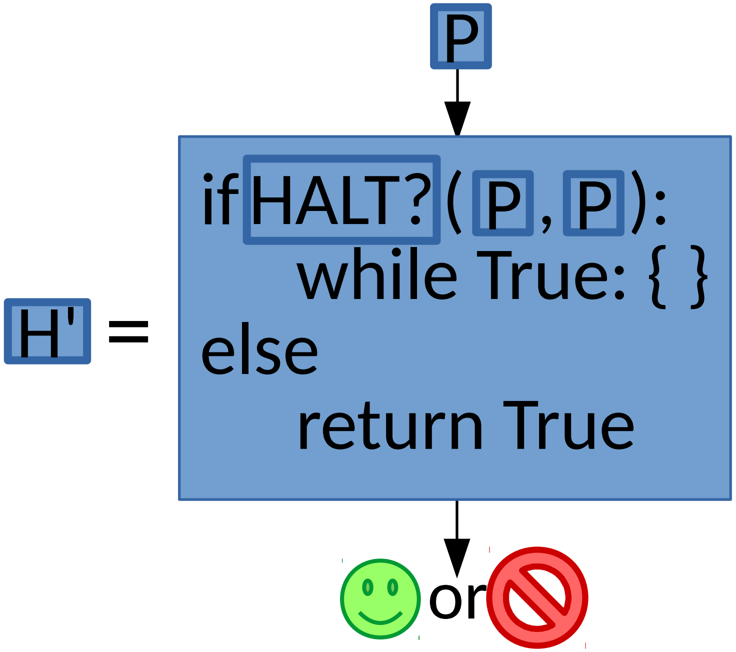
Brief Review of Undecidability



Static Analysis

Brief Review of Undecidability

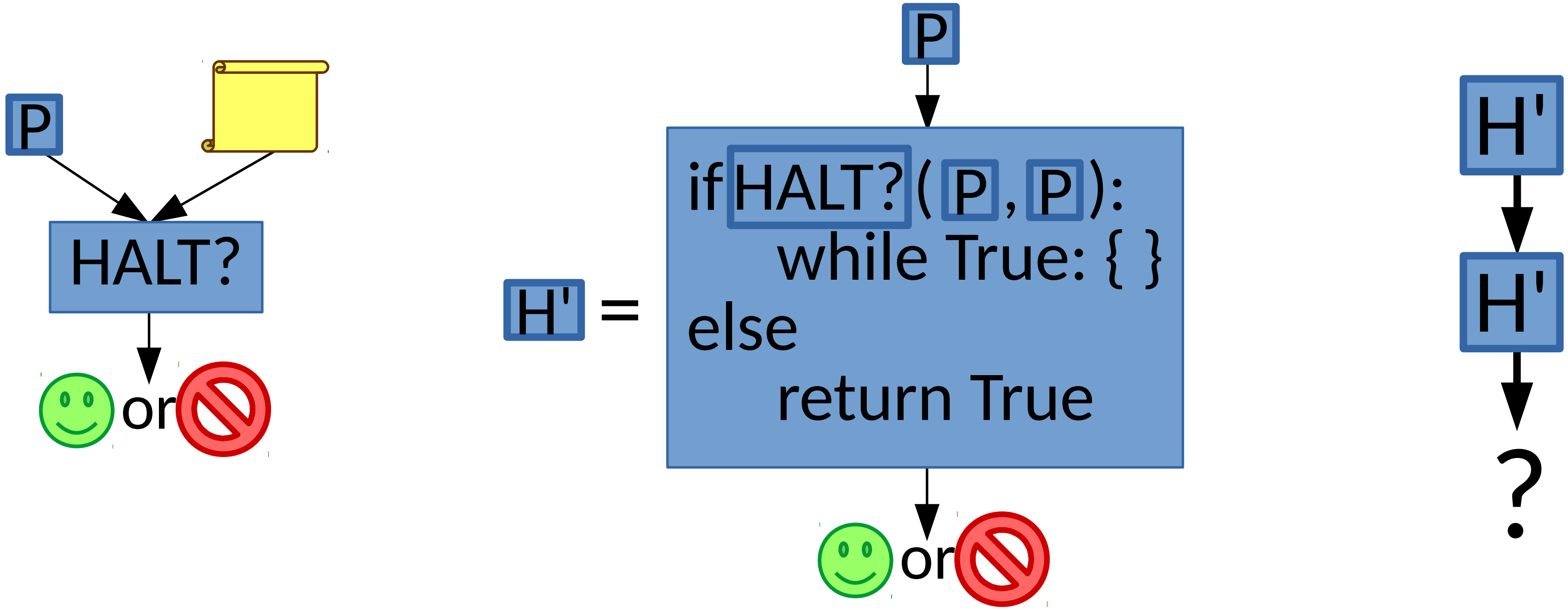




```
if HALT?(P, P):
    while True: { }
else
    return True
```

Static Analysis

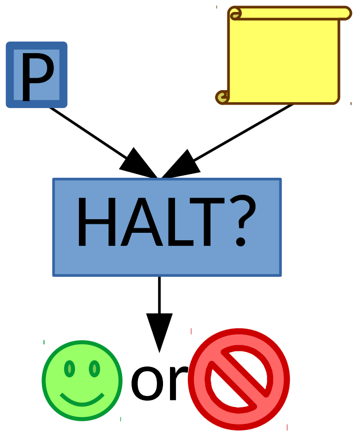
Brief Review of Undecidability

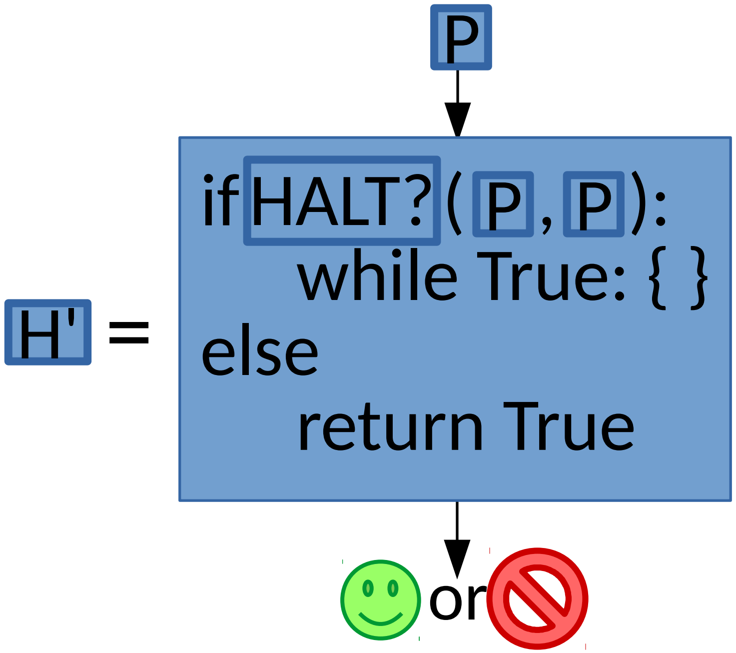


```
if HALT?(P, P):
    while True: { }
else
    return True
```

Static Analysis

Brief Review of Undecidability





```
if HALT?(P, P):
        while True: { }
else
        return True
```

It's a classic paradox!

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?
  - Does my program terminate?

But wait? Isn't that impossible?

Static Analysis

Static analyses consider *all* possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?
  - Does my program terminate?

But wait? Isn't that impossible?

- Only if answers must be perfect.

Static Analysis

Overapproximate or *underapproximate* the problem, and try to solve this simpler version.

Static Analysis

Overapproximate or underapproximate the problem, and try to solve this simpler version.

- Sound analyses
  - Overapproximate
  - Guaranteed to find violations of property
  - May raise false alarms

Static Analysis

Overapproximate or underapproximate the problem, and try to solve this simpler version.

- Sound analyses
  - Overapproximate
  - Guaranteed to find violations of property
  - May raise false alarms
- Complete analyses
  - Underapproximate
  - Reported violations are real
  - May miss violations

Static Analysis

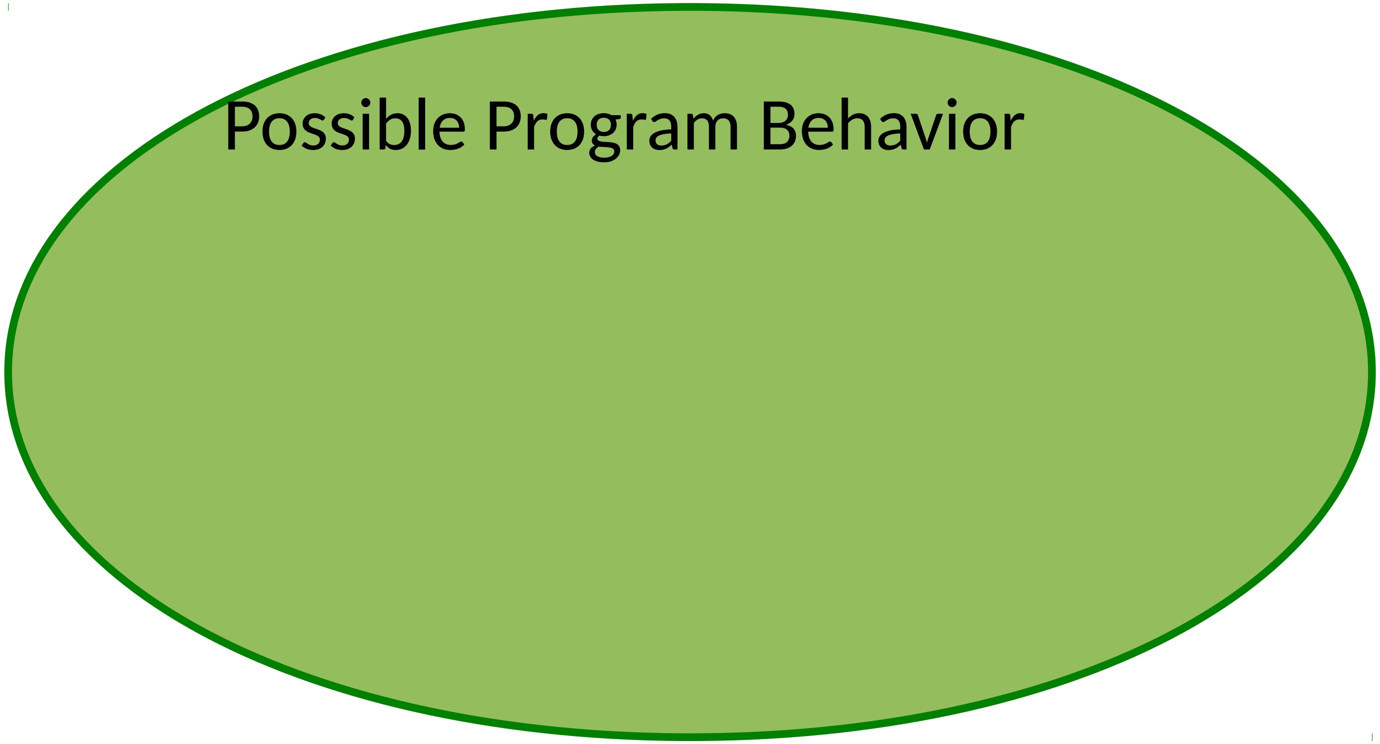
Overapproximate or underapproximate the problem, and try to solve this simpler version.

- Sound analyses
  - Overapproximate
  - Guaranteed to find violations of property
  - May raise false alarms
- Complete analyses
  - Underapproximate
  - Reported violations are real
  - May miss violations

Striking the right balance is key to a useful analysis

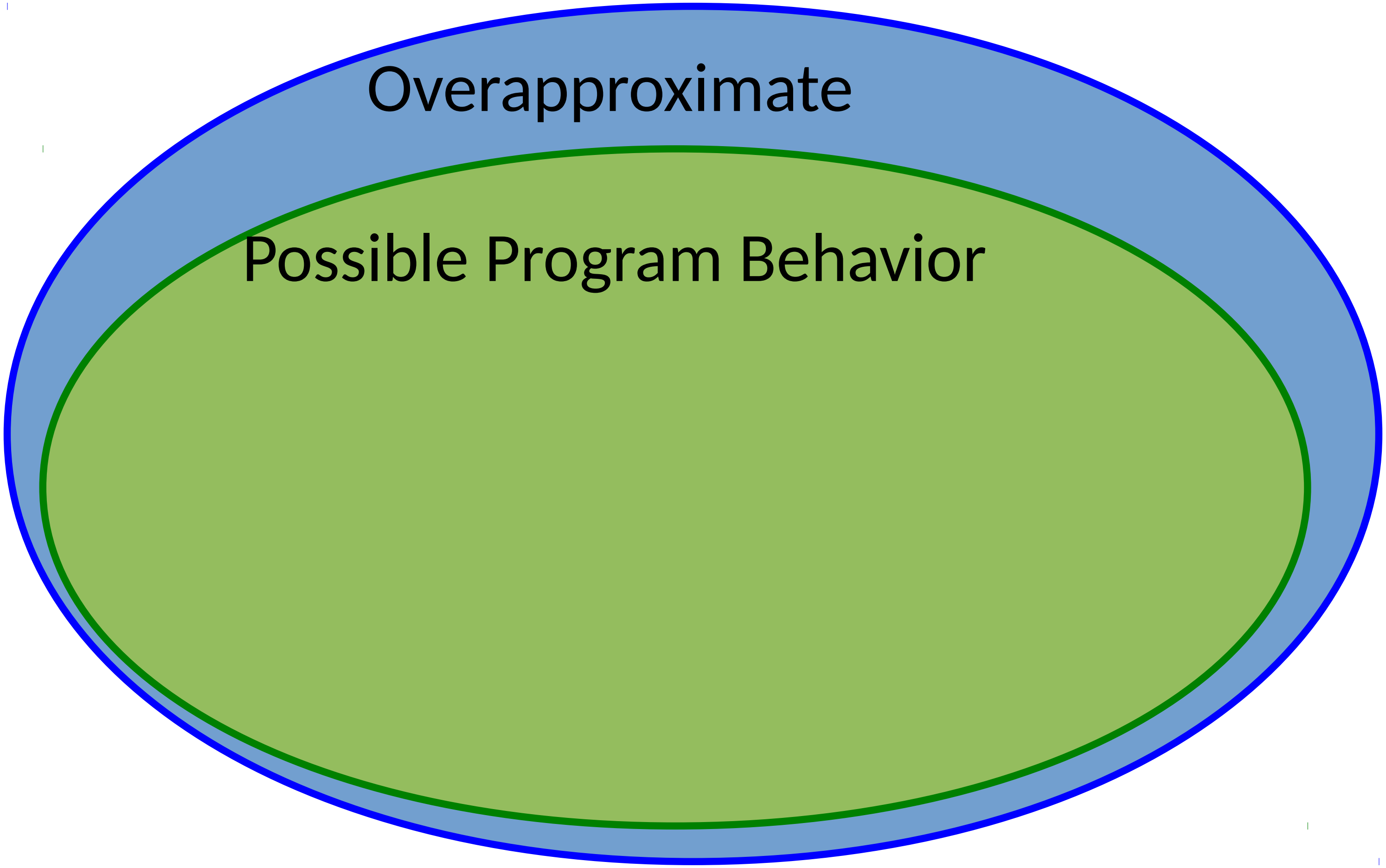
Static Analysis

Modeled program behaviors



Static Analysis

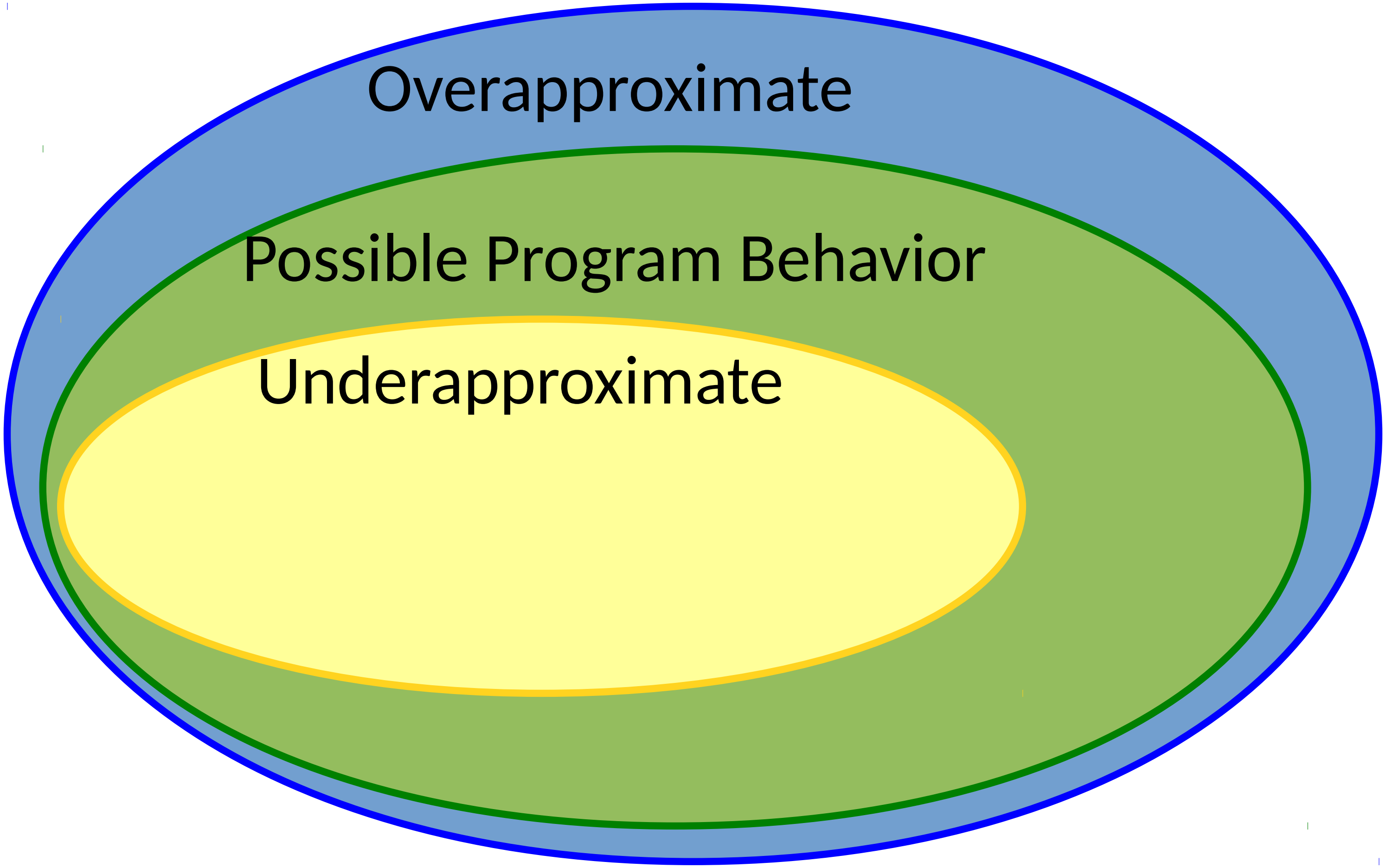
Modeled program behaviors



Consider some behaviors possible when they are not.

Static Analysis

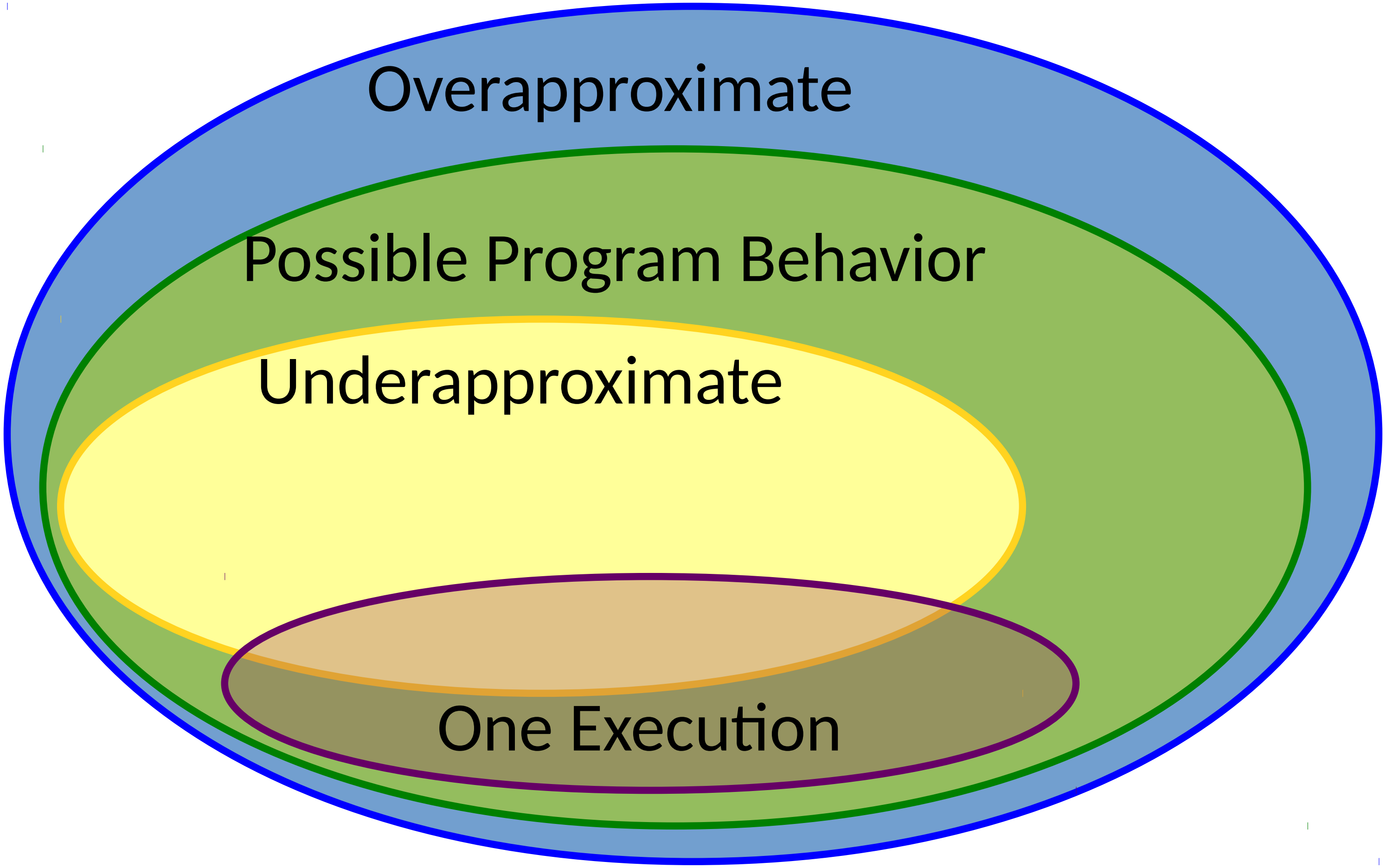
Modeled program behaviors



Ignore some behaviors that *are* possible.

Static Analysis

Modeled program behaviors



A Simple Example – Dataflow Analysis

Q: Is a particular number ever negative?
- Might be an offset into invalid memory!

Approximate the program's behavior

A Simple Example – Dataflow Analysis

Q: Is a particular number ever negative?

- Might be an offset into invalid memory!

Approximate the program's behavior

- Concrete domain: integers
- Abstract domain: {-,0,+} $\bigcup$ {$\top$,$\bot$}

A Simple Example – Dataflow Analysis

Q: Is a particular number ever negative?

- Might be an offset into invalid memory!

Approximate the program's behavior

- Concrete domain: integers
- Abstract domain: {-,0,+} ⋃ {⊤,⊥}

concrete(x) = 5 ↦ abstract(x) = +
concrete(y) = -3 ↦ abstract(y) = -
concrete(z) = 0 ↦ abstract(z) = 0

Combines sets of the concrete domain

A Simple Example – Dataflow Analysis

- Transfer Functions show how to evaluate this approximated program:

A Simple Example – Dataflow Analysis

- Transfer Functions show how to evaluate this approximated program:
  - **+** + **+** → **+**
  - **-** + **-** → **-**
  - **0** + **0** → **0**
  - **0** + **-** → **-**
  - …
  - **+** + **-** → ⊤ (unknown / might vary)
  - **…** / **0** → ⊥ (undefined)

A Simple Example – Dataflow Analysis

- Transfer Functions show how to evaluate this approximated program:
  - **+** + **+** → **+**
  - **-** + **-** → **-**
  - **0** + **0** → **0**
  - **0** + **-** → **-**
  - …
  - **+** + **-** → **⊤** (unknown / might vary)
  - **…** / **0** → **⊥** (undefined)

This type of approximation is called *abstract interpretation*.

A Simple Example – Dataflow Analysis

- Transfer Functions show how to evaluate this approximated program:
  - **+** + **+** → **+**
  - **-** + **-** → **-**
  - **0** + **0** → **0**
  - **0** + **-** → **-**
  - …
  - **+** + **-** → ⊤ (unknown / might vary)
  - **…** / **0** → ⊥ (undefined)
- Meet Operator (⊓) combines results across program paths

A Simple Example – Dataflow Analysis

- Transfer Functions show how to evaluate this approximated program:
  - + + + → +
  - - + - → -
  - 0 + 0 → 0
  - 0 + - → -
  - …
  - + + - → ⊤(unknown / might vary)
  - … / 0 → ⊥(undefined)
- Meet Operator (⊓) combines results across program paths
- Can be subtle.
  - The above is not sound or complete. Why?

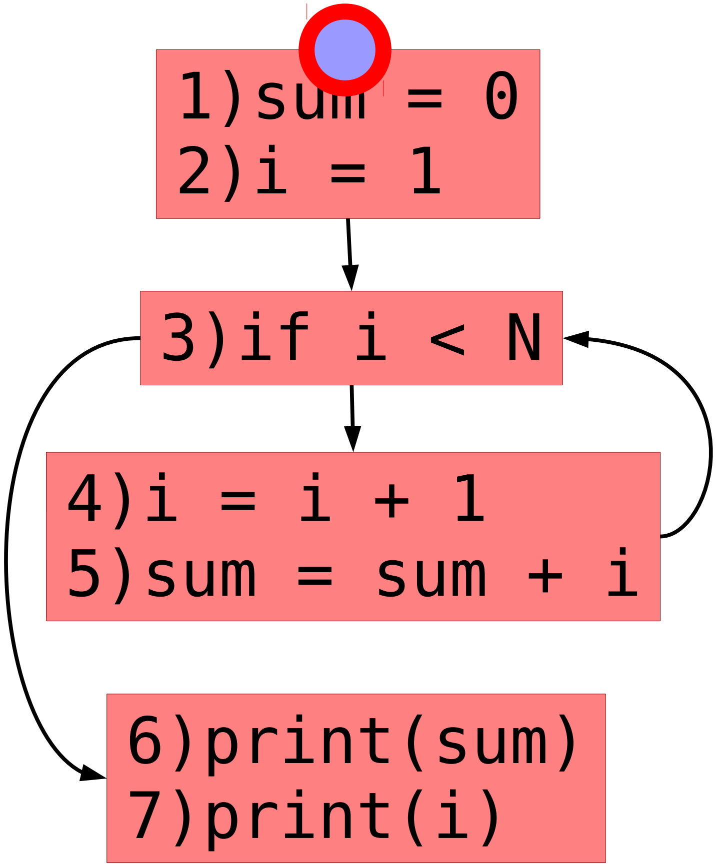
A Simple Example – Dataflow Analysis

- Transfer Functions show how to evaluate this approximated program:
  - + + + → +
  - - + - → -
  - 0 + 0 → 0
  - 0 + - → -
  - …
  - + + - → ⊤ (unknow
  - … / 0 → ⊥ (undefi
- Meet Operator (⊓) gram paths
- Can be subtle.
  - The above is not sound or complete. Why?

Consider a divide by 0 analysis.
What are:
    True Positives
    False Positives
    True Negatives
    False Negatives

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

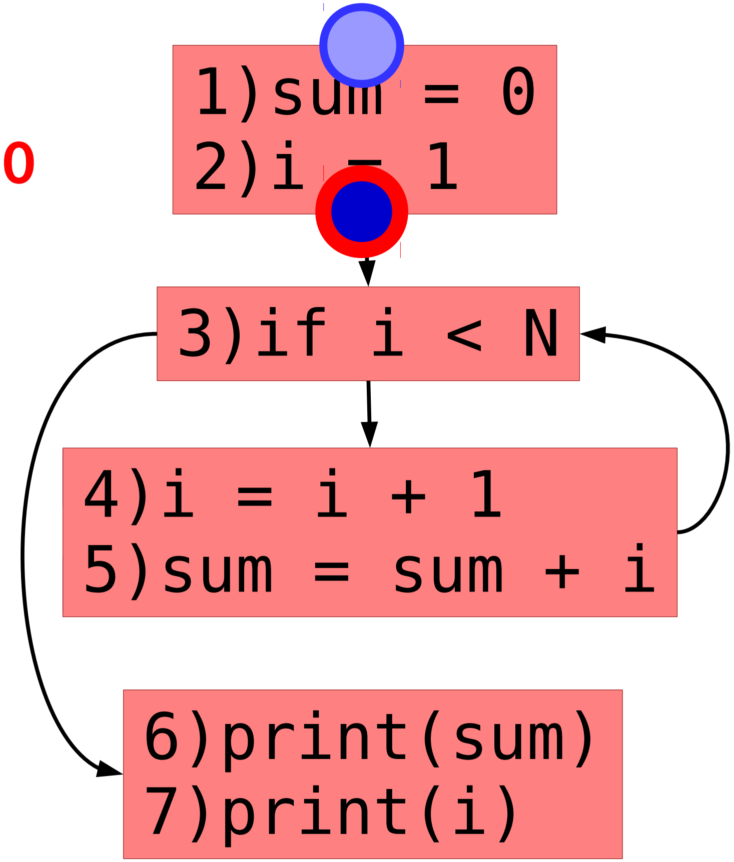
sum → ⊥
i → ⊥

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.

sum → ⊥
i → ⊥

sum → 0
i → +



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

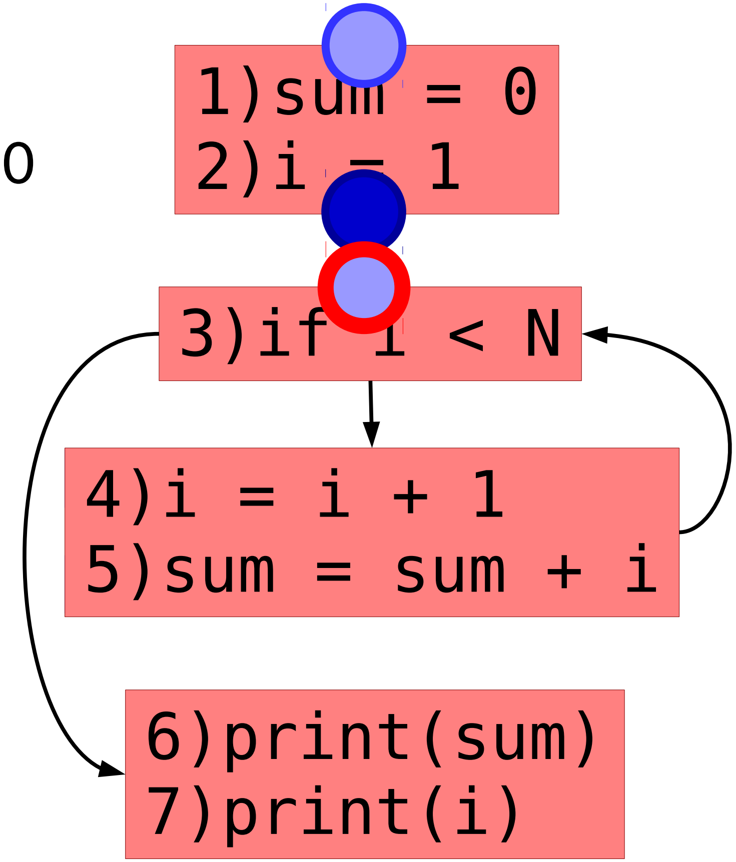
```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.

sum → 0
i → +



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

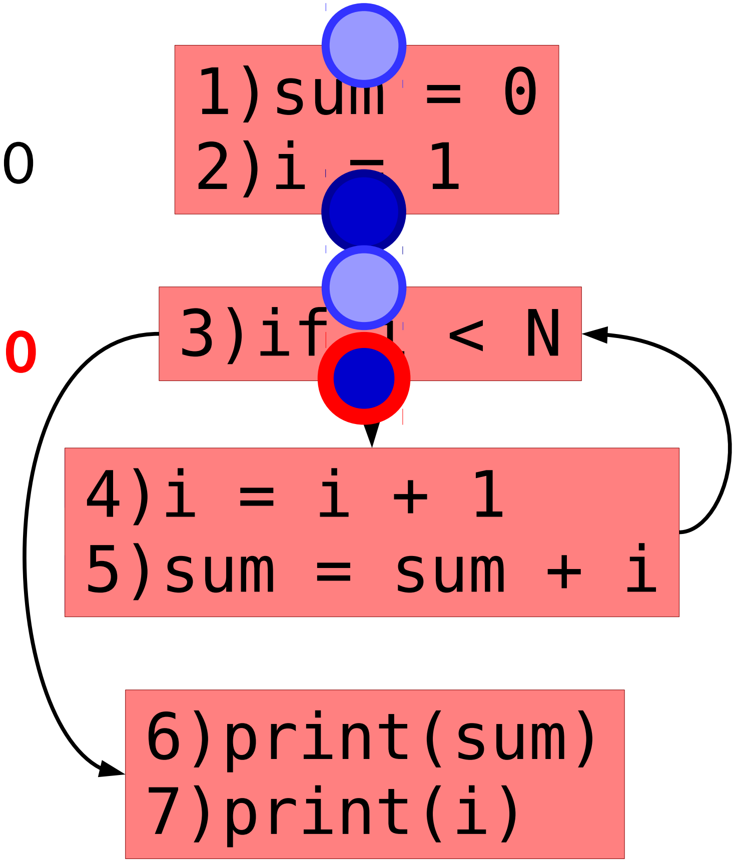
```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

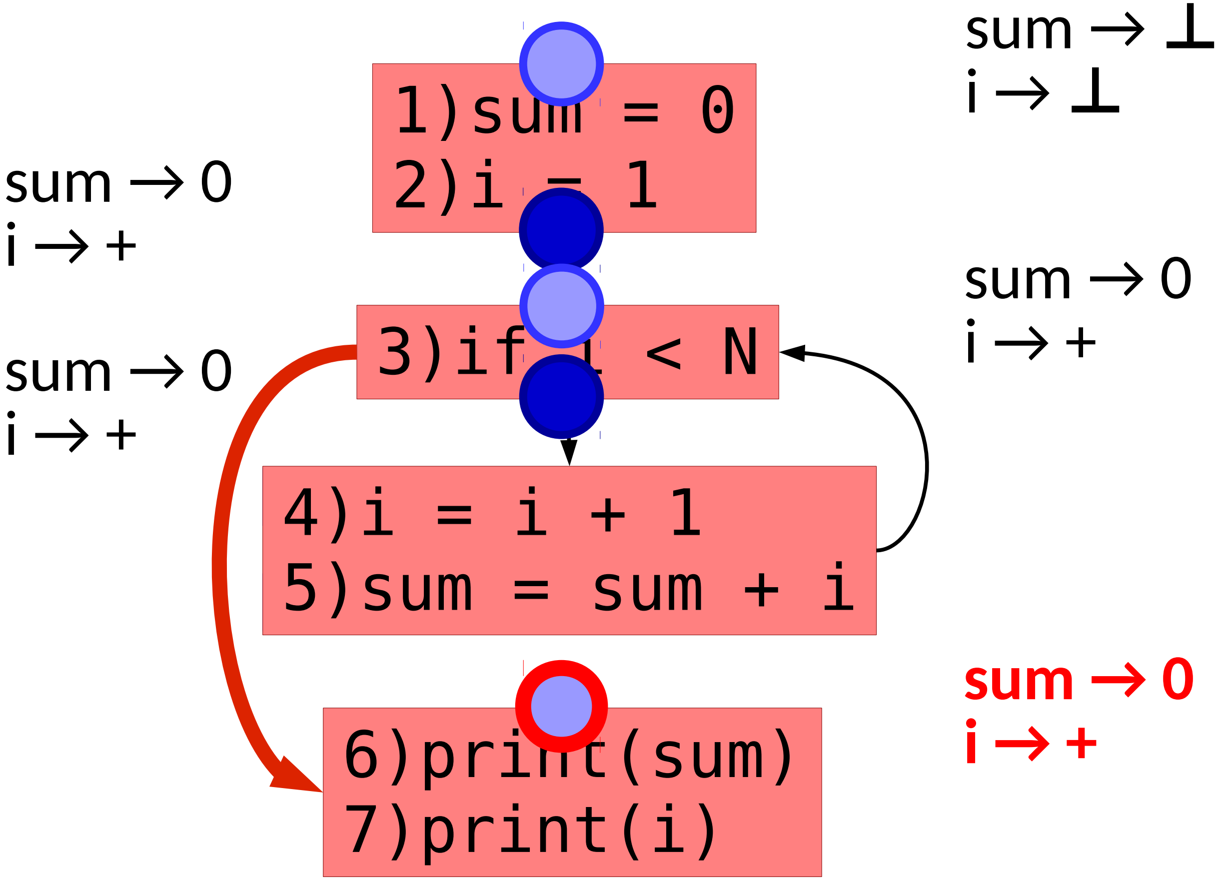
sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



sum → ⊥
i → ⊥

```
1)sum = 0
2)i = 1
```

sum → 0
i → +

sum → 0
i → +

```
3)if i < N
```

sum → 0
i → +

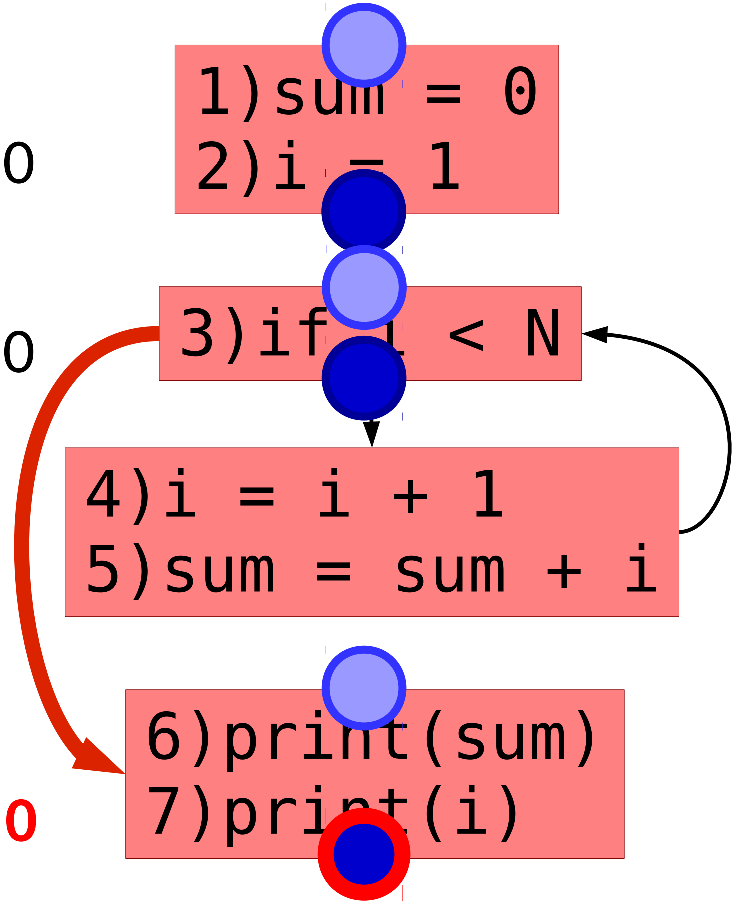
```
4)i = i + 1
5)sum = sum + i
```

sum → 0
i → +

```
6)print(sum)
7)print(i)
```

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → 0
i → +

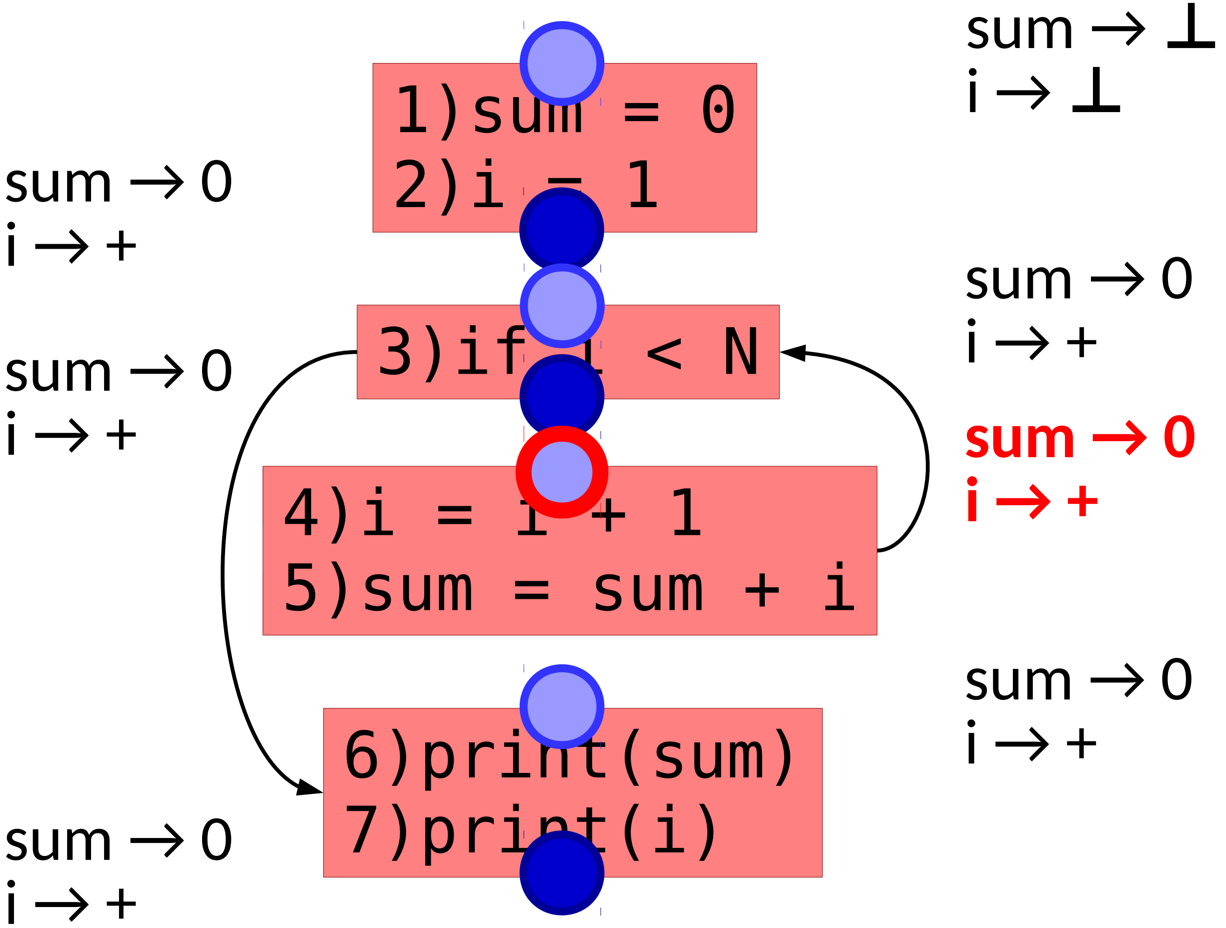
sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

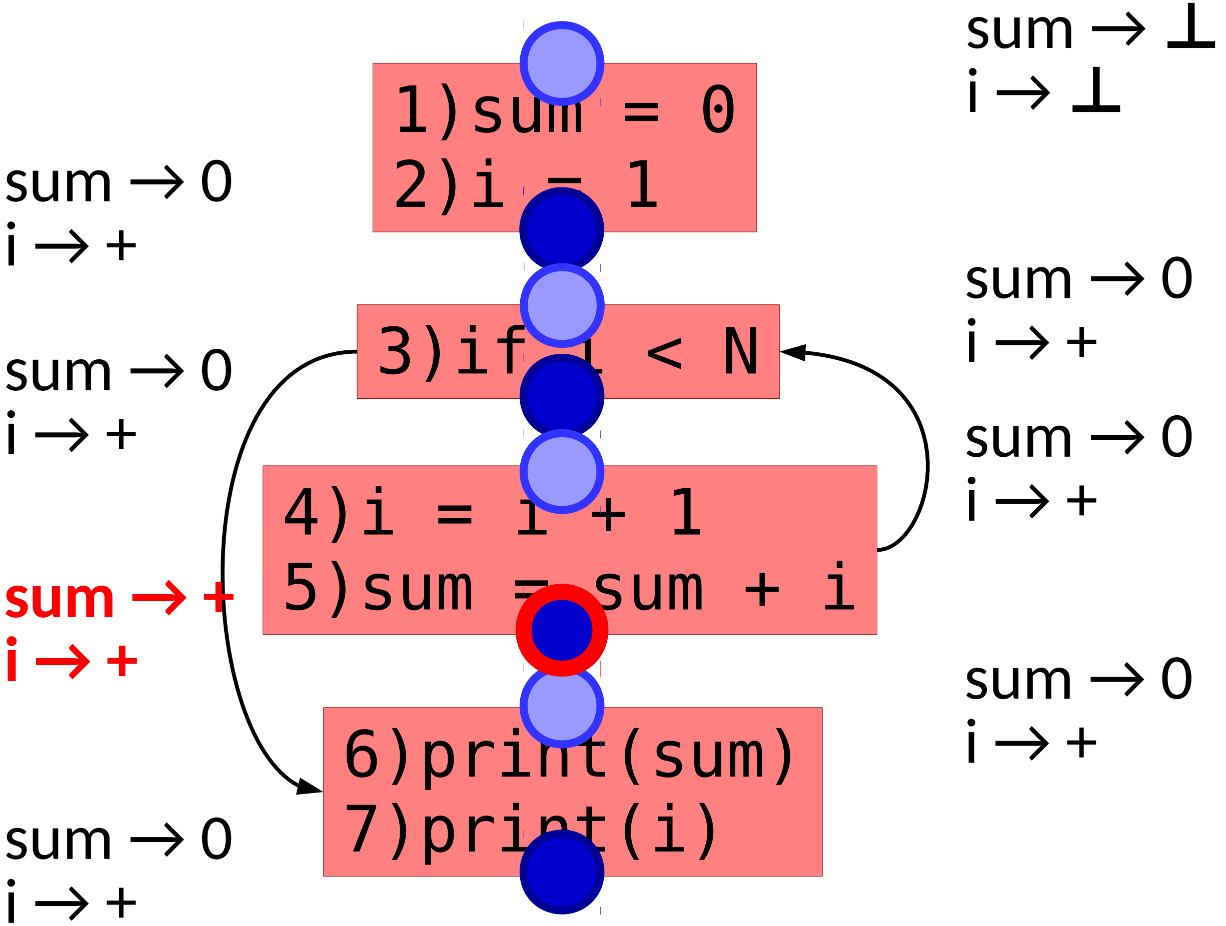
sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

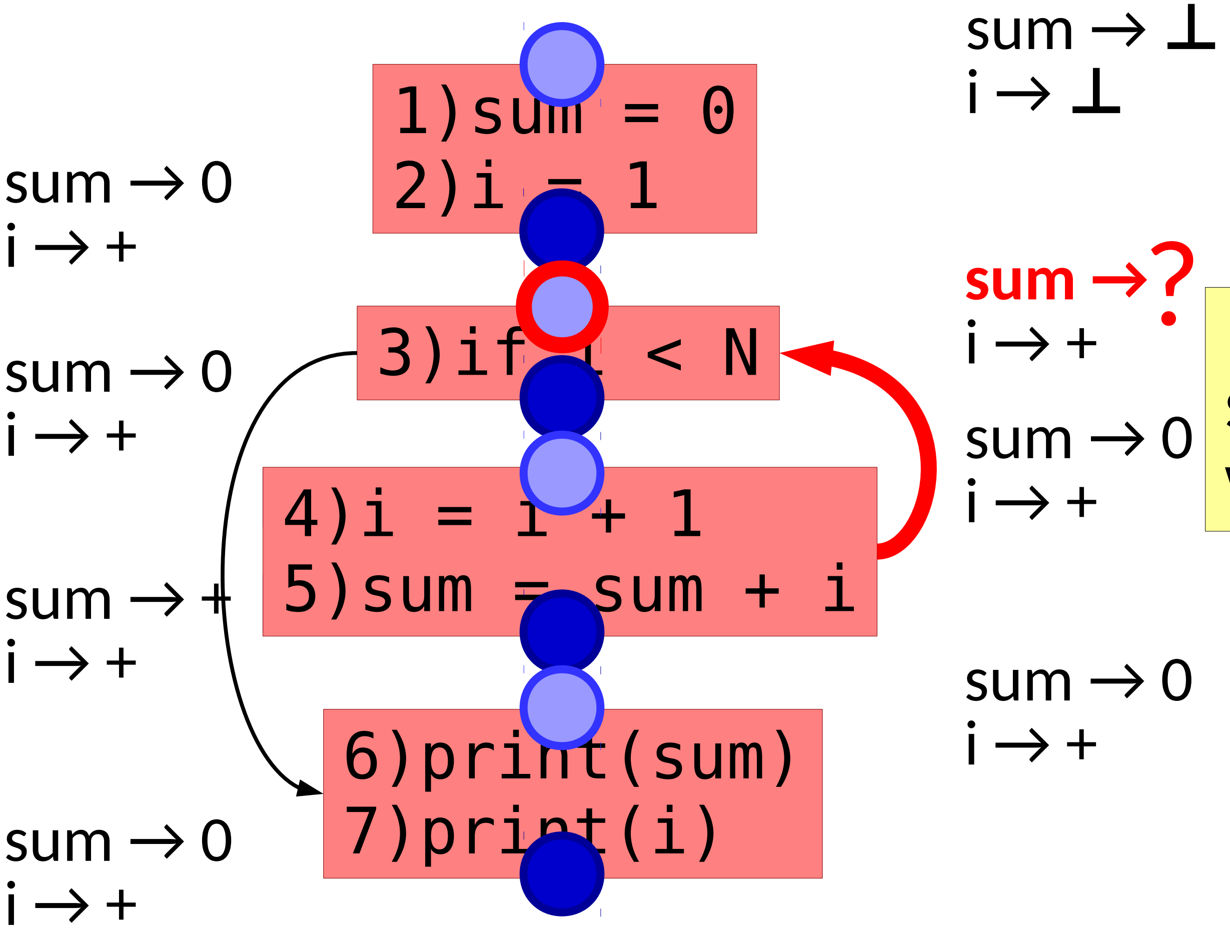
sum → +
i → +

sum → 0
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

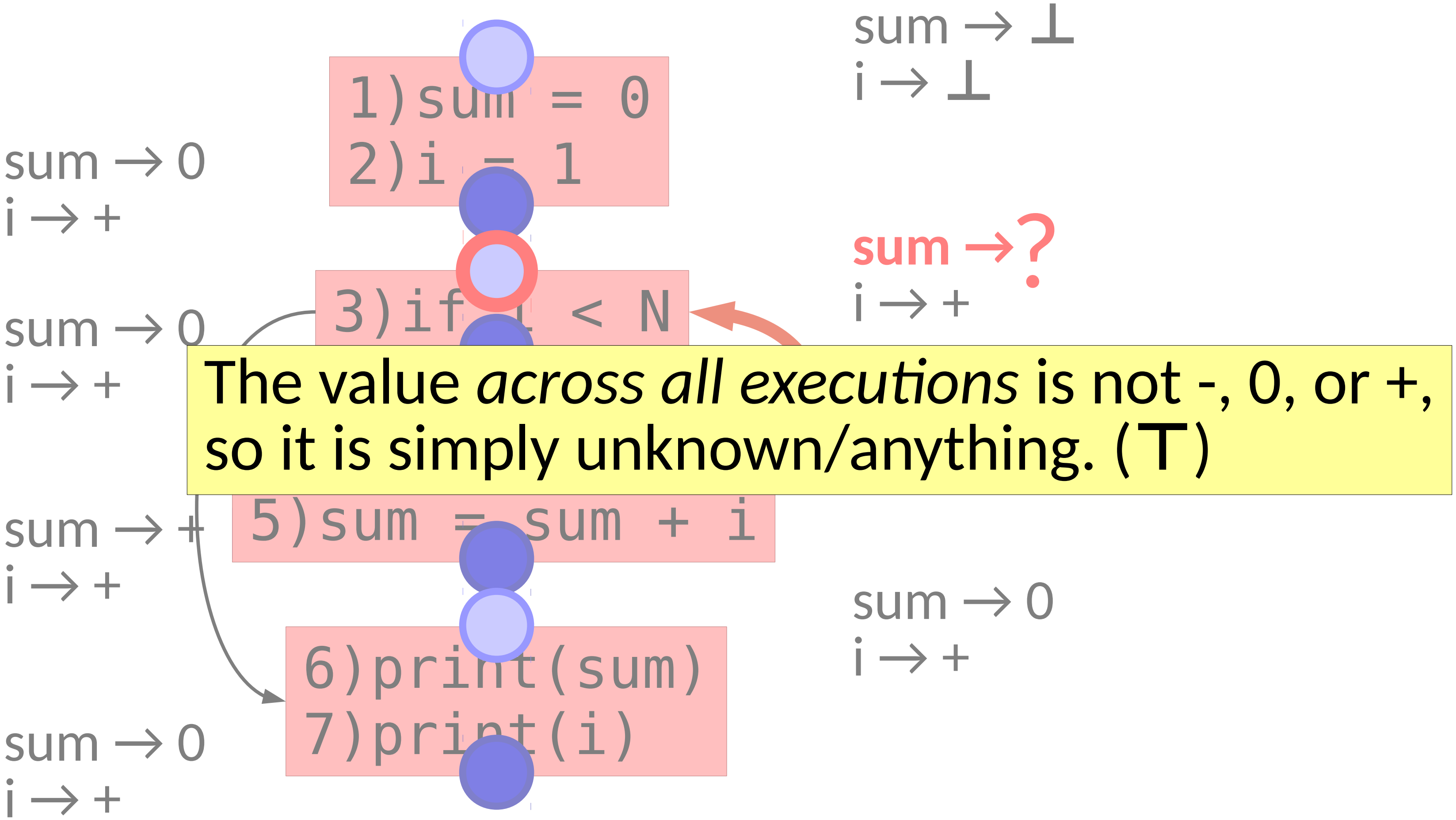
```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

Meet Operator
sum was 0, but
what should it be now?

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



sum → ⊥
i → ⊥

1)sum = 0
2)i = 1

sum → 0
i → +

sum →?
i → +

3)if i < N

sum → 0
i → +

The value *across all executions* is not -, 0, or +, so it is simply unknown/anything. (⊤)

5)sum = sum + i

sum → +
i → +

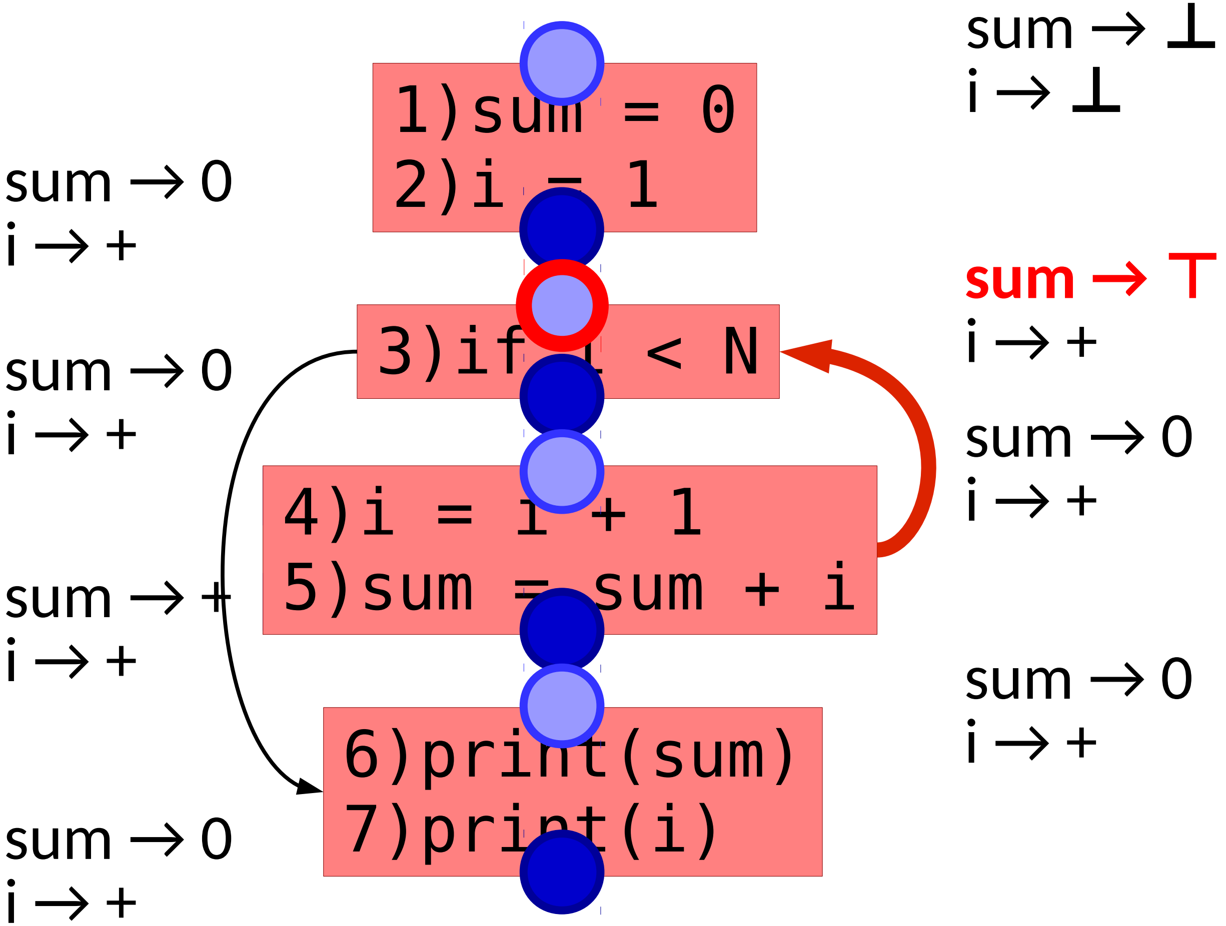
sum → 0
i → +

6)print(sum)
7)print(i)

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → ⊤
i → +

sum → 0
i → +

sum → 0
i → +

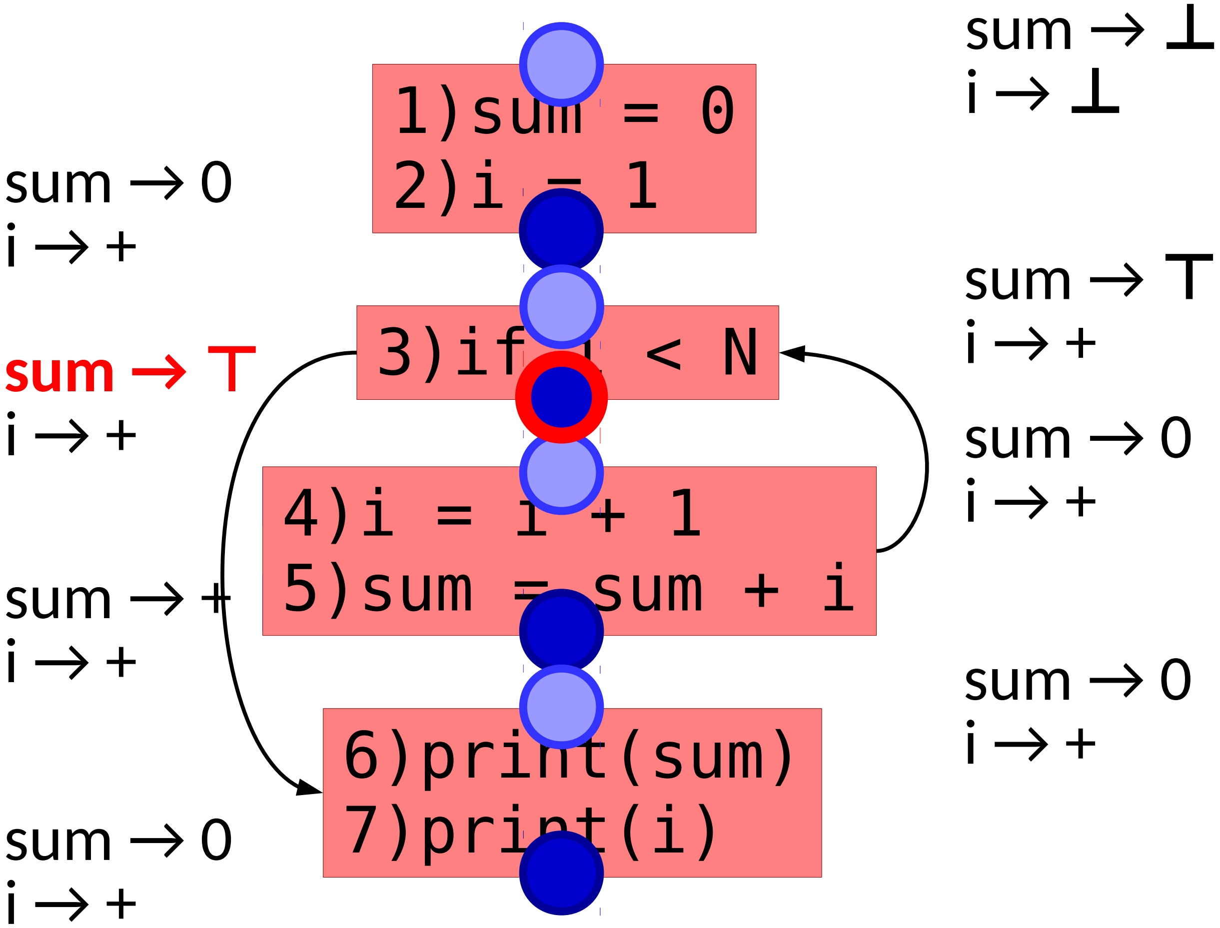
sum → +
i → +

sum → 0
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
3)if i < N
4)i = i + 1
5)sum = sum + i
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → ⊤
i → +

sum → ⊤
i → +

sum → 0
i → +

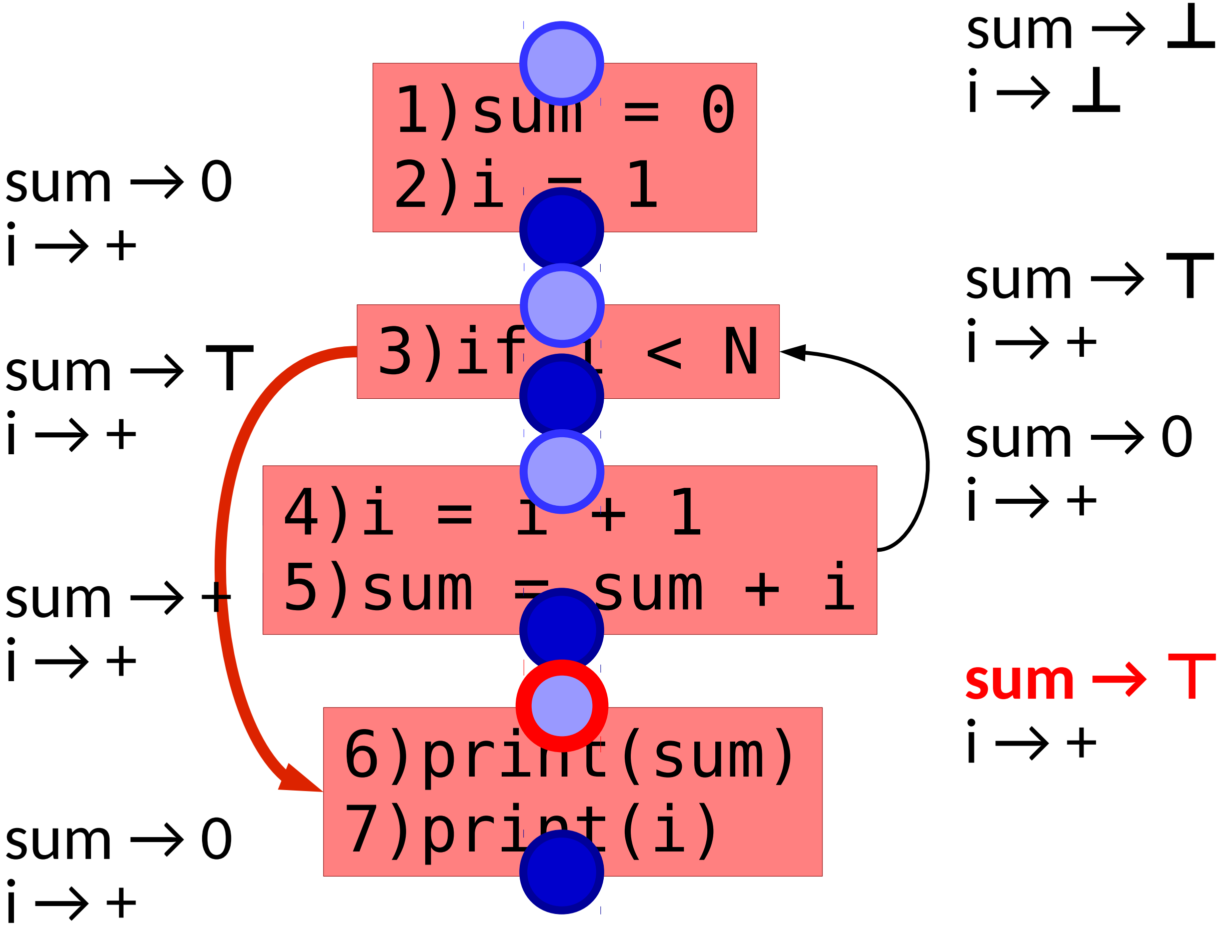
sum → +
i → +

sum → 0
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → ⊤
i → +

sum → ⊤
i → +

sum → 0
i → +

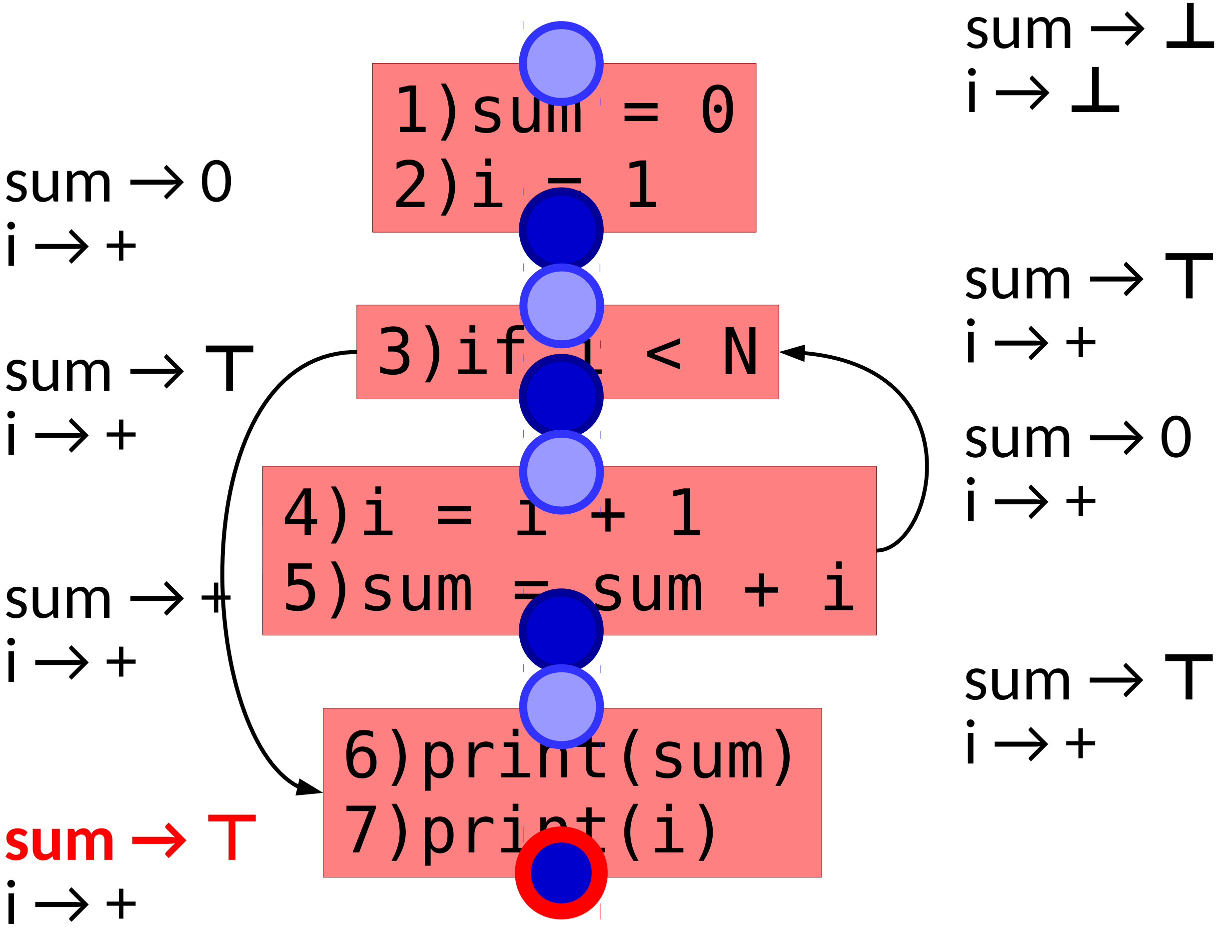
sum → +
i → +

sum → ⊤
i → +

sum → 0
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.



```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

```
4)i = i + 1
5)sum = sum + i
```

```
6)print(sum)
7)print(i)
```

sum → ⊥
i → ⊥

sum → 0
i → +

sum → ⊤
i → +

sum → ⊤
i → +

sum → 0
i → +

sum → +
i → +

sum → ⊤
i → +

sum → ⊤
i → +

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.
  - Continue until we reach a fixed point

    (No more changes)

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.
  - Continue until we reach a fixed point

    (No more changes)
  - Proper ordering can improve the efficiency.

    (Topological Order, Strongly Connected Components)

Dataflow Analysis

- Now model the abstract program state and propagate through the CFG.
  - Continue until we reach a fixed point

    (No more changes)
  - Proper ordering can improve the efficiency.
    - (Topological Order, Strongly Connected Components)

Will it always terminate?

Dataflow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function

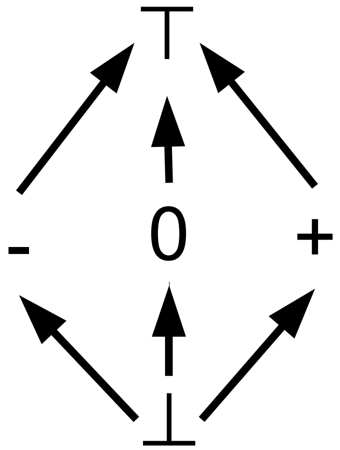
Dataflow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework

  Loosely: <CFG, Transfer Function, Lattice Abstraction>

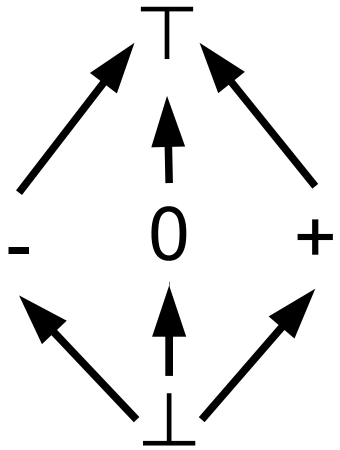
Dataflow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework
  - $\{-,0,+\} \cup \{\top,\bot\}$
  - They define a partial order
  - Abstract state can only move ***up*** lattice at a statement



Dataflow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework
  - {-,0,+} ⋃ {⊤,⊥}
  - They define a partial order
  - Abstract state can only move ***up*** lattice at a statement



Why does this specific example terminate?

Dataflow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework
- But in theory a lattice need not be finite!

Dataflow Analysis

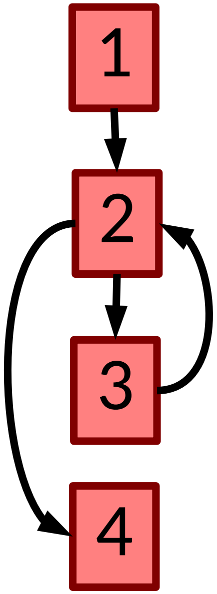
- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework
- But in theory a lattice need not be finite!
  - Widening operators can still make it feasible (e.g., heuristically raise to $\top$)

Dataflow Analysis

- Note: need to model program state before and after each statement
- Proper ordering & a work list algorithm improves the efficiency

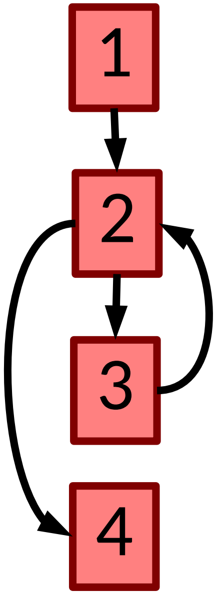
Worklist Algorithms

```
work = nodes()
state(n) = $\bot$ $\forall$ n $\in$ nodes()
while work $\neq \emptyset$:
    unit = take(work)
    old = state(unit)
    before = $\prod$state(p)
                 $\forall$ p $\in$ preds(unit)
    new = transfer(before, unit)
    if old $\neq$ after:
        work = work $\cup$ succs(unit)
        state(unit) = new
```



Worklist Algorithms

```
work = nodes()
state(n) = $\bot$ $\forall$ n $\in$ nodes()
while work $\neq \emptyset$:
    unit = take(work)
    old = state(unit)
    before = $\prod$state(p)
                $\forall$ p $\in$ preds(unit)
    new = transfer(before, unit)
    if old $\neq$ after:
        work = work $\cup$ succs(unit)
        state(unit) = new
```



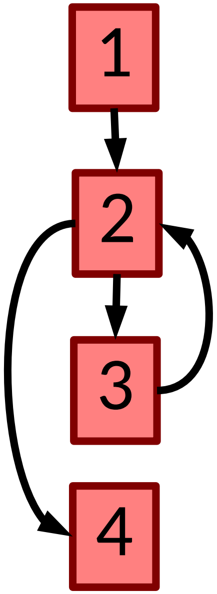
work: 1 2 3 4

state: { (1 ↦ ⊥) (3 ↦ ⊥)
(2 ↦ ⊥) (4 ↦ ⊥) }

Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
             ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 1



work: 2 3 4

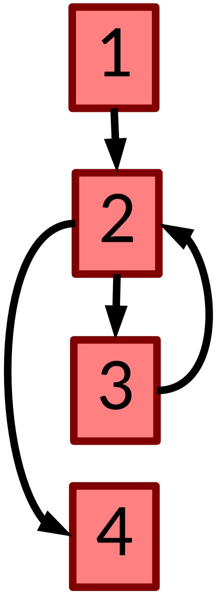
state: $\{ (1 \mapsto \bot) \quad (3 \mapsto \bot)$
$(2 \mapsto \bot) \quad (4 \mapsto \bot) \}$

Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
             ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 1

old = ⊥



work: 2 3 4

state: $\left\{ \begin{matrix} (1 \mapsto \bot) & (3 \mapsto \bot) \\ (2 \mapsto \bot) & (4 \mapsto \bot) \end{matrix} \right\}$

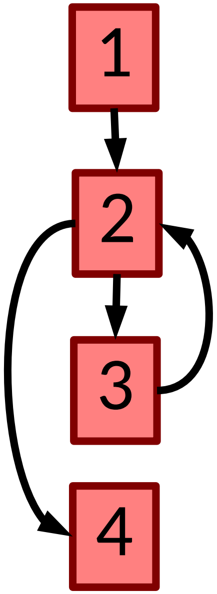
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
            ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 1

old = ⊥

new = sum → 0
i → +



work: 2 3 4

state: { (1 ↦ ⊥) (2 ↦ ⊥) (3 ↦ ⊥) (4 ↦ ⊥) }

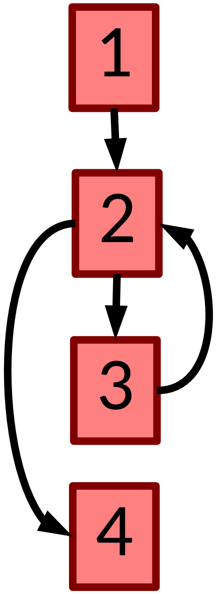
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
                 ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 1

old = ⊥

new = sum → 0
i → +



work: 2 3 4

state: {(1 ↦ sum → 0
i → +) (3 ↦ ⊥)
(2 ↦ ⊥) (4 ↦ ⊥) }

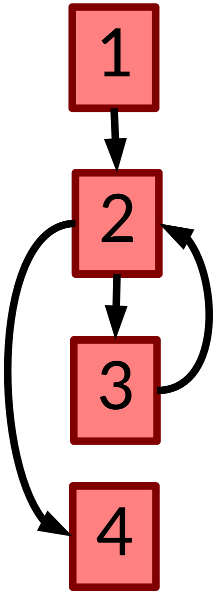
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
             ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 2

old = ⊥

new = sum → 0
i → +



work: 3 4

state: { (1 ↦ sum → 0, i → +) (3 ↦ ⊥)

(2 ↦ sum → 0, i → +) (4 ↦ ⊥) }

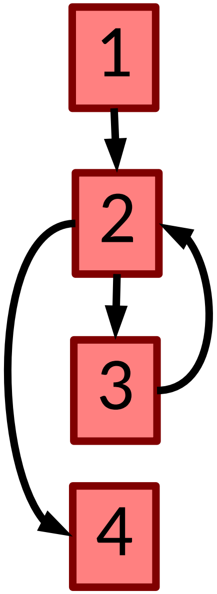
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
             ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 3

old = ⊥

new = sum → +
i → +



work: ⬚ 4 2

2 was added back to the list

state: {(1 ↦ sum → 0, i → +) (2 ↦ sum → 0, i → +) (3 ↦ sum → +, i → +) (4 ↦ ⊥)}

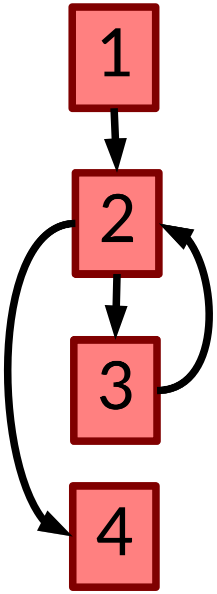
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
              ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 4

old = ⊥

new = sum → 0
i → +



work: 2

state: {(1 ↦ sum → 0, i → +) (3 ↦ sum → +, i → +)
(2 ↦ sum → 0, i → +) (4 ↦ sum → 0, i → +)}

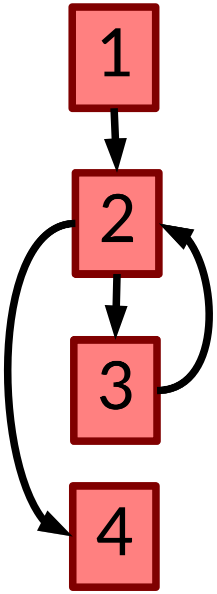
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
            ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 2

old = sum → 0
i → +

new = sum → ⊤
i → +



4,3 were added back to the list

work: 4 3

state: {
(1 ↦ sum → 0, i → +)
(2 ↦ sum → ⊤, i → +)
(3 ↦ sum → +, i → +)
(4 ↦ sum → 0, i → +)
}

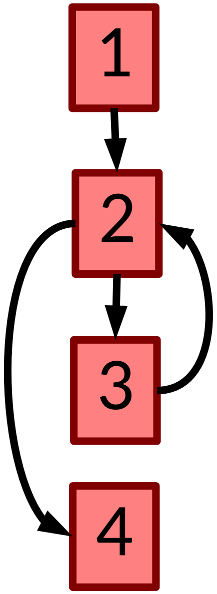
Worklist Algorithms

```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
             ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 4

old = sum → 0, i → +

new = sum → ⊤, i → +



work: 3

state: { (1 ↦ sum → 0, i → +) (2 ↦ sum → ⊤, i → +) (3 ↦ sum → +, i → +) (4 ↦ sum → ⊤, i → +) }

Worklist Algorithms

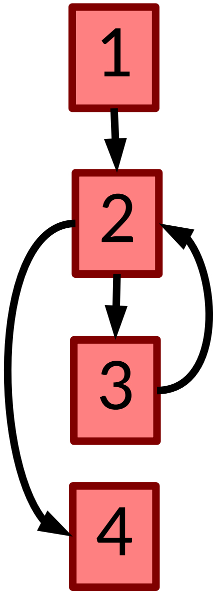
```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
            ∀ p ∈ preds(unit)
    new = transfer(before, unit)
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 3

old = sum → +, i → +

new = sum → +, i → +

No change



work:

state: {(1 ↦ sum → 0, i → +), (2 ↦ sum → ⊤, i → +), (3 ↦ sum → +, i → +), (4 ↦ sum → ⊤, i → +)}

Worklist Algorithms

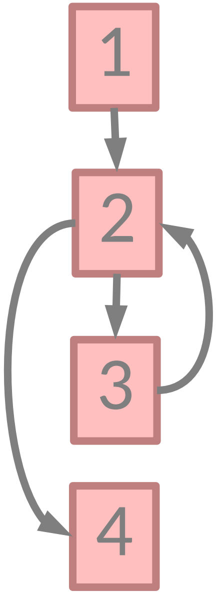
```
work = nodes()
state(n) = ⊥ ∀ n ∈ nodes()
while work ≠ ∅:
    unit = take(work)
    old = state(unit)
    before = ∏state(p)
            ∀ p ∈ preds(un
    new = transfer(before,
    if old ≠ after:
        work = work ∪ succs(unit)
        state(unit) = new
```

unit = 4

old = sum → 0
i → +

new = sum → ⊤

Done!



work:

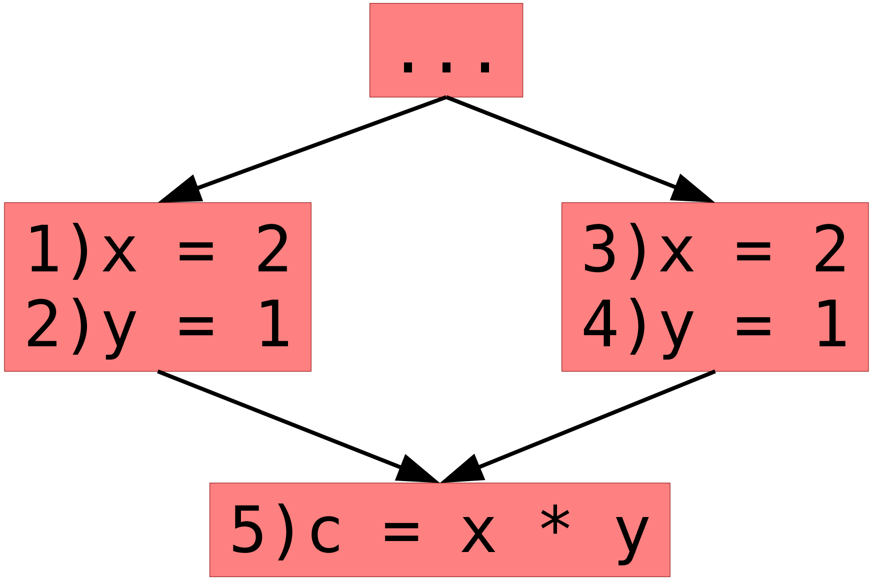
state: {
(1 ↦ sum → 0, i → +)
(2 ↦ sum → ⊤, i → +)
(3 ↦ sum → +, i → +)
(4 ↦ sum → ⊤, i → +)
}

Effect of Approximation

- There are several possible sources of imprecision

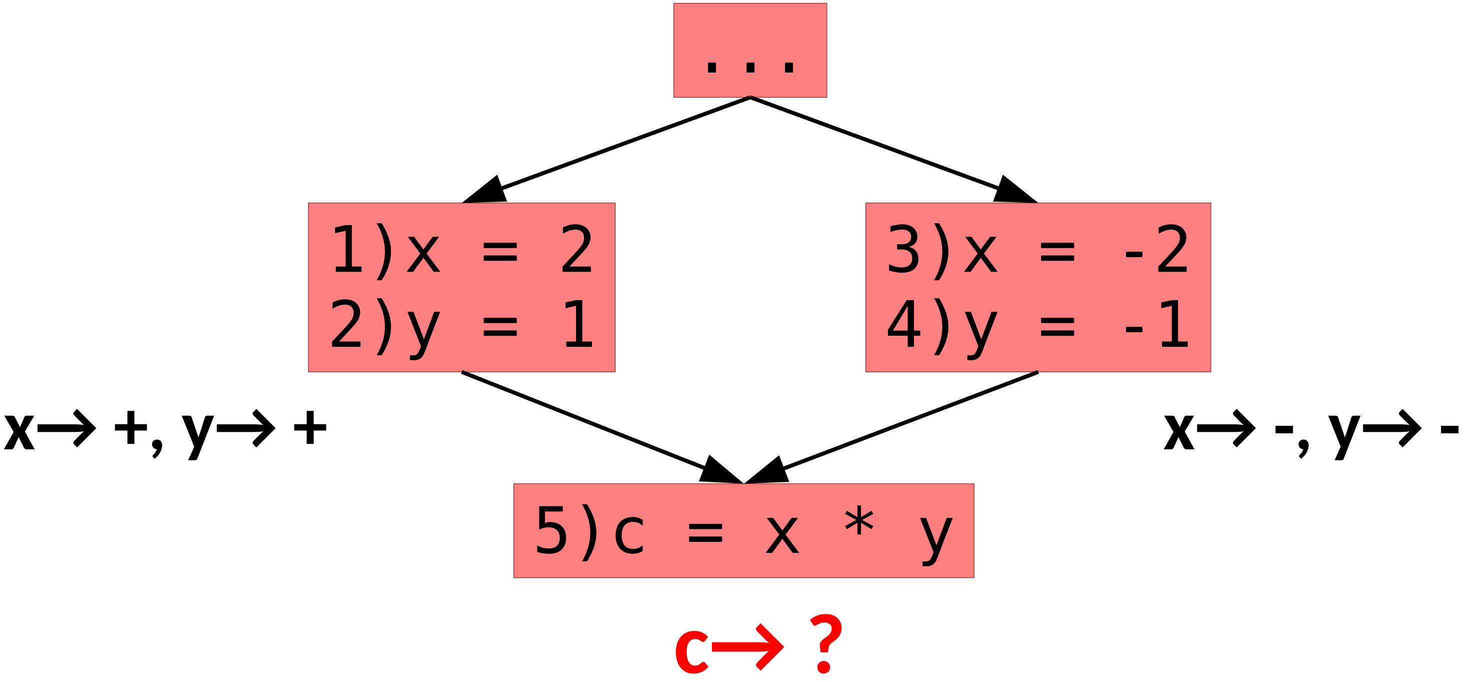
Effect of Approximation

- There are several possible sources of imprecision



Effect of Approximation

- There are several possible sources of imprecision



```
...
```

```
1)x = 2
2)y = 1
```

```
3)x = -2
4)y = -1
```

x→ +, y→ +

x→ -, y→ -

```
5)c = x * y
```

c→ ?

Effect of Approximation

- There are several possible sources of imprecision
- 2 Key sources are
  - Control flow
    - Many different paths are summarized together

Effect of Approximation

- There are several possible sources of imprecision
- 2 Key sources are
  - Control flow
    - Many different paths are summarized together
  - Abstraction
    - Deliberately throwing away information
    - Granularity of program state affects correlations across variables

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)

For one path p: $f_p(\perp) = f_n(f_{n-1}(\ldots f_1(f_0(\perp))))$

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)

For one path p: $f_p(\perp) = f_n(f_{n-1}(\ldots f_1(f_0(\perp))))$
For all paths p: $\prod_p f_p(\perp)$

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)
- Are they different?

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)
- Are they different?
  - Sometimes. But sometime solutions are perfect.

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)
- Are they different?
  - Sometimes. But sometime solutions are perfect.
  - When f() is distributive, MFP=MOP

    $f(x \prod y \prod z) = f(x) \prod f(y) \prod f(z)$

Effect of Approximation

- We compute results with maximal fixed points (MFP) in the lattice
- Ideal solution is a Meet Over all Paths (MOP)
- Are they different?
  - Sometimes. But sometime solutions are perfect.
  - When f() is distributive, MFP=MOP

    $f(x \prod y \prod z) = f(x) \prod f(y) \prod f(z)$
  - This applies to an important class of problems called bitvector frameworks.

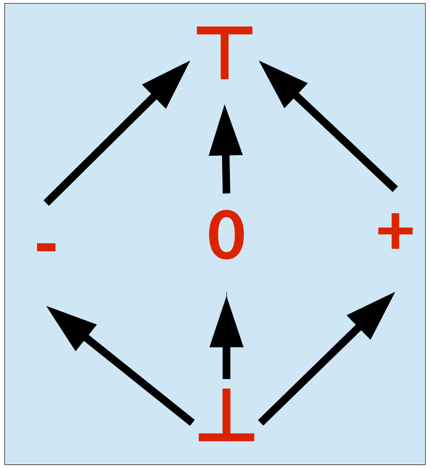
Effect of Approximation

- If approximation yields imprecise results, why do we do it?

Recap: Dataflow Analysis

Analyze complex behavior with approximation:

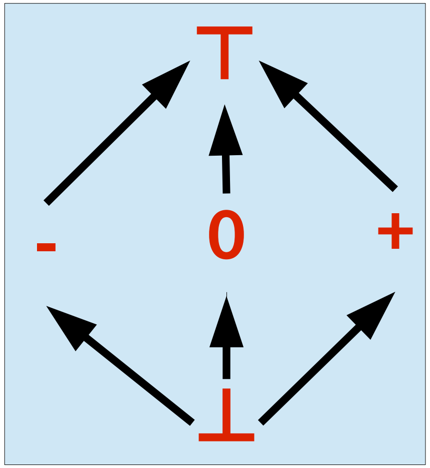
- Abstract domain: e.g. {**-**,**0**,**+**} ⋃ {⊤,⊥}
- Transfer functions: **-** + **+** → ⊤
- Bounded domain lattice height:
- Concern for false + & -



Recap: Dataflow Analysis

Analyze complex behavior with approximation:

- Abstract domain: e.g. {**-**,**0**,**+**} ⋃ {⊤,⊥}
- Transfer functions: **-** + **+** → ⊤
- Bounded domain lattice height:
- Concern for false + & -



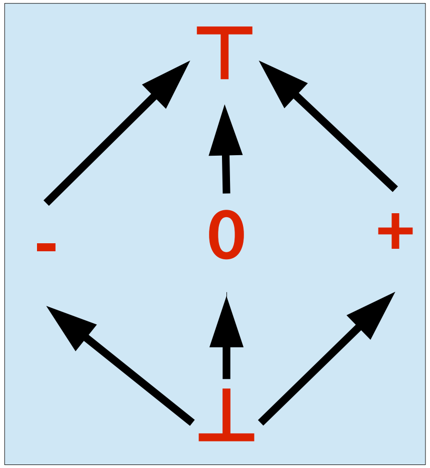
Implementation:

- Computing using work lists
- Speeding up by sorting CFG nodes

Recap: Dataflow Analysis

Analyze complex behavior with approximation:

- Abstract domain: e.g. {**-**,**0**,**+**} ⋃ {⊤,⊥}
- Transfer functions: **-** + **+** → ⊤
- Bounded domain lattice height:
- Concern for false + & -



Implementation:

- Computing using work lists
- Speeding up by sorting CFG nodes

Let's see an example

File Policy Analysis

Goal: Identify potential misuses of open/closed files

File Policy Analysis

Goal: Identify potential misuses of open/closed files

- Files may be open or closed

File Policy Analysis

Goal: Identify potential misuses of open/closed files

- Files may be open or closed
- Many operations may only occur on open files

e.g. read, write, print, flush, close, …

File Policy Analysis

Goal: Identify potential misuses of open/closed files

- Files may be open or closed
- Many operations may only occur on open files

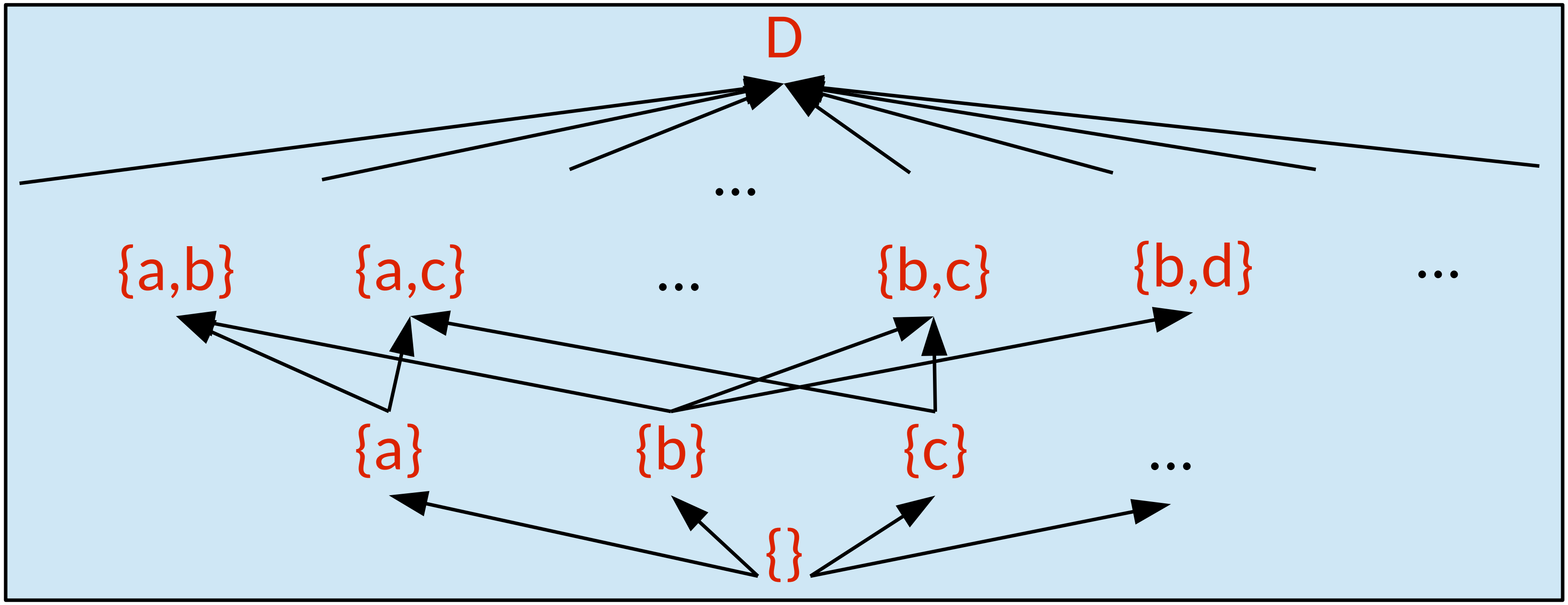
e.g. read, write, print, flush, close, …

What should our design actually be?

- Abstract domain?
- Transfer functions?
- Lattice?

Bitvector Frameworks

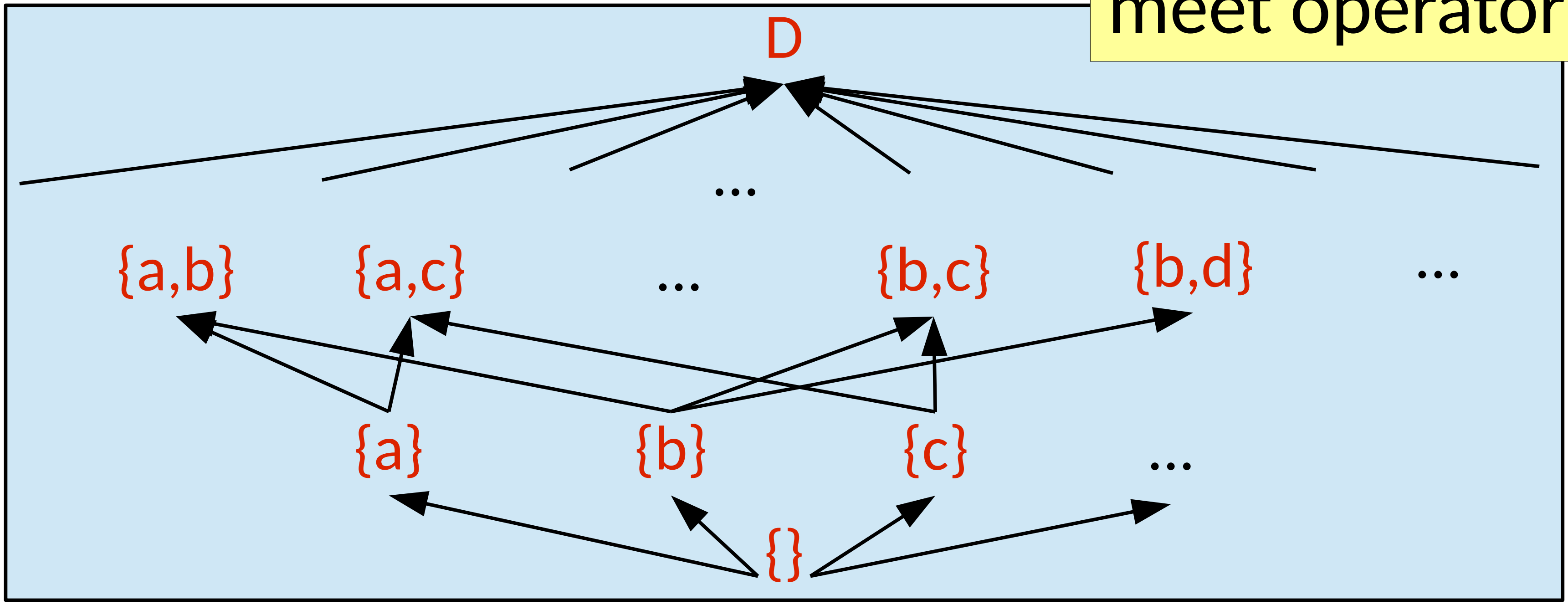
- When the property concerns subsets of a finite set, the abstract domain & lattice are easy:
  - Concrete: D = {a, b, c, d, … }
  - Abstract: $\wp$(D) = { {}, {a}, {b}, …, {a, b}, {a, c}, …}
  - Lattice: Defined by subset relation:



Bitvector Frameworks

- When the property concerns subsets of a finite set, the abstract domain & lattice are easy:
  - Concrete: D = {a, b, c, d, … }
  - Abstract: $\wp$(D) = { {}, {a}, {b}, …, {a, b}, {a, c}, …}
  - Lattice: Defined by subset relation:

What would the meet operator be?



Bitvector Frameworks

- Why is this convenient?
  - Hint: ***bitvector*** frameworks

Bitvector Frameworks

- Why is this convenient?
  - Hint: ***bitvector*** frameworks
  - X={a,b}, Y={c,d} → $X \sqcup Y$ = {a,b}∪{c,d} = {a,b,c,d}
  - We can implement the abstract state using efficient bitvectors!

Bitvector Frameworks

- Why is this convenient?
  - Hint: ***bitvector*** frameworks
  - X={a,b}, Y={c,d} → X⊔Y = {a,b}∪{c,d} = {a,b,c,d}
  - We can implement the abstract state using efficient bitvectors!

Let's see how we might implement the file policy framework in LLVM…

[DEMO]

Flow Insensitive Analysis

- Saw *flow sensitive* analysis
  - Modeling state at each statement is expensive
  - Scales to functions and small components
  - Usually not beyond 1000s of lines without care

Flow Insensitive Analysis

- Saw *flow sensitive* analysis
  - Modeling state at each statement is expensive
  - Scales to functions and small components
  - Usually not beyond 1000s of lines without care
- *Flow insensitive* analyses aggregate into a global state
  - Better scalability
  - Less precision
  - “Does this function modify global variable X?”

Context Sensitive Analyses

- Program behavior may be dependent on the call stack / calling context.
  - “If bar() is called by foo(), then it is exception free.”
  - Can enable more precise *interprocedural* analyses

Context Sensitive Analyses

- Program behavior may be dependent on the call stack / calling context.
  - “If bar() is called by foo(), then it is exception free.”
  - Can enable more precise *interprocedural* analyses

Can you imagine how to solve this?
What problems might arise?

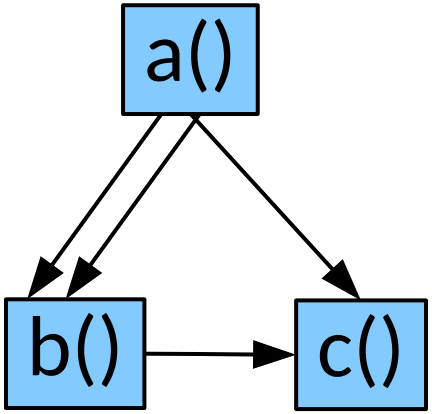
Context Sensitivity

- Recall that we can extract a call graph
  - Just as you are doing in your first project!

```
def a():
   b()
   …
   b()

def b():
   …
   c()

def c():
   …
```



The behavior of c() could be affected by each "…"

Modeling them can make analysis more precise.

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```

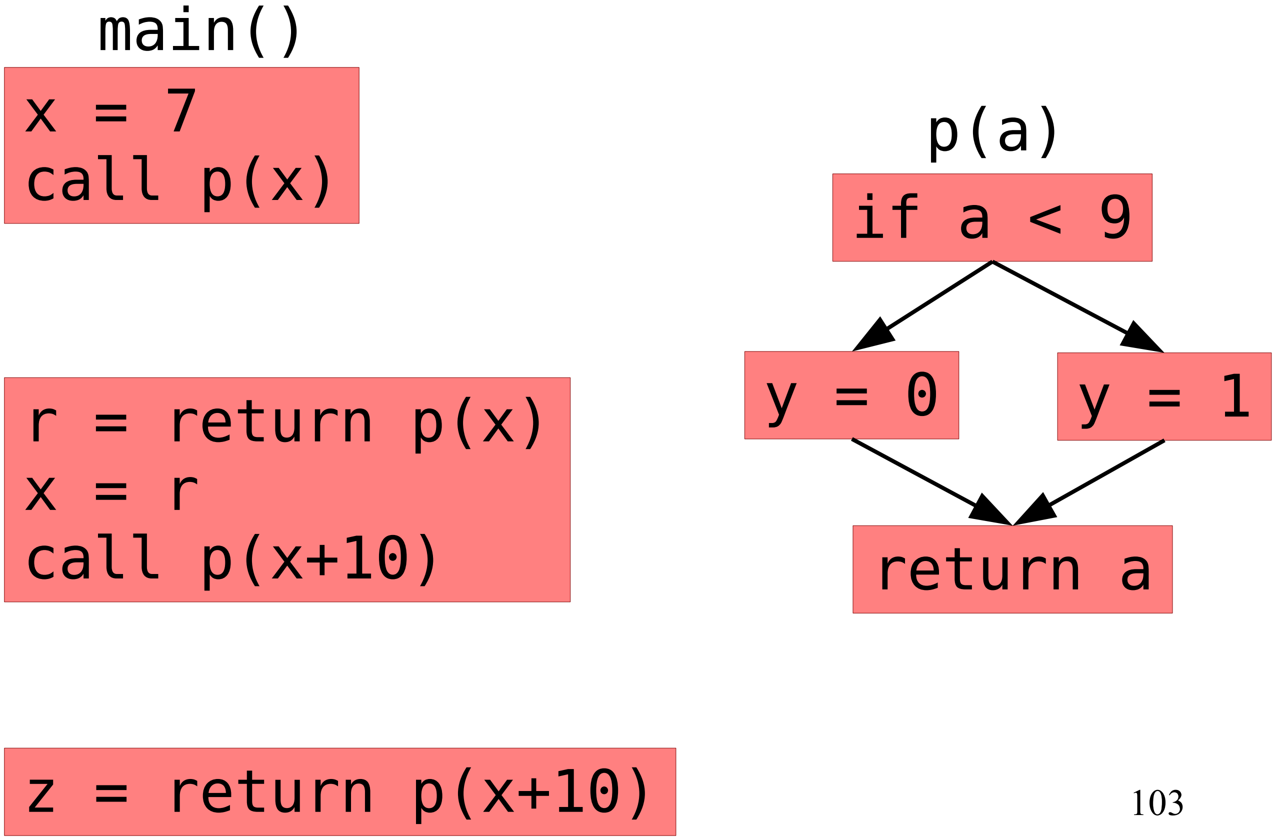
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



main()

```
x = 7
call p(x)
```

```
r = return p(x)
x = r
call p(x+10)
```

```
z = return p(x+10)
```

p(a)

```
if a < 9
```

```
y = 0
```

```
y = 1
```

```
return a
```

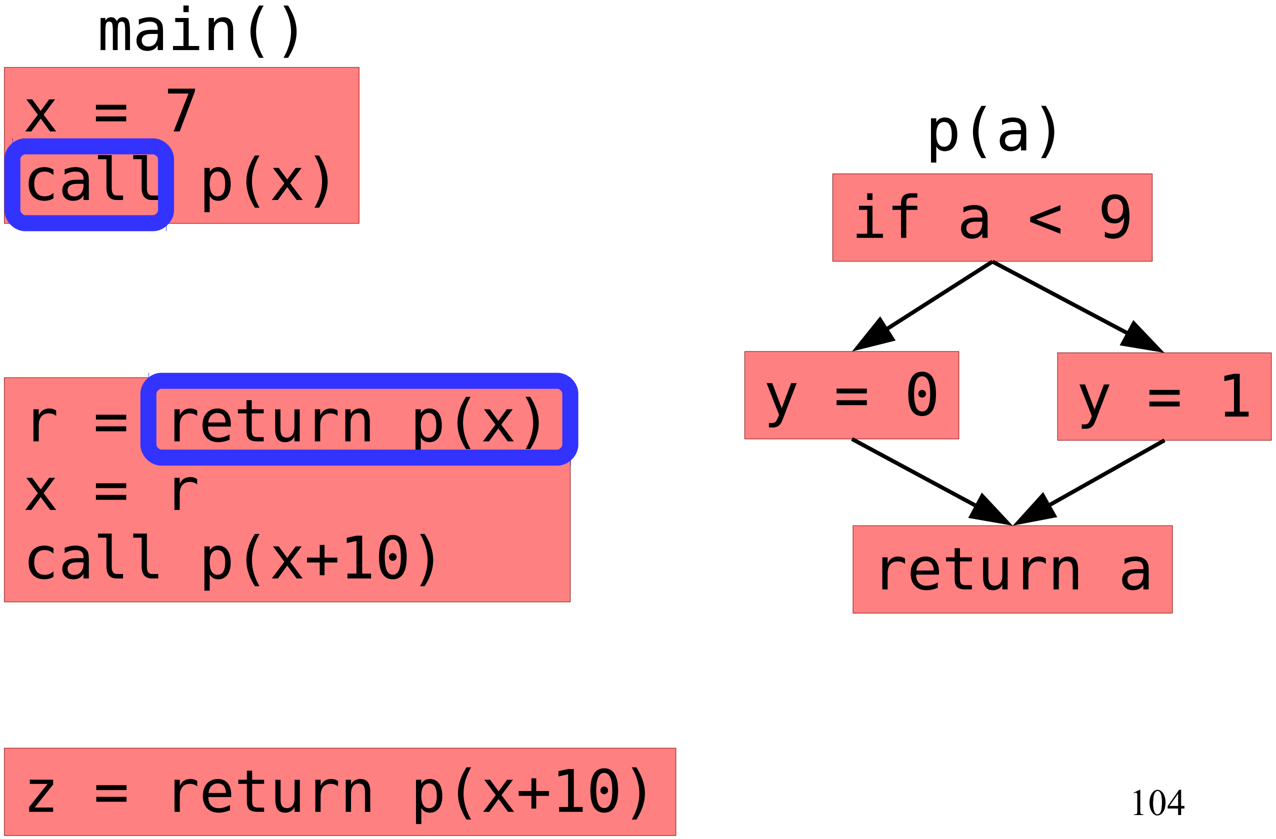
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



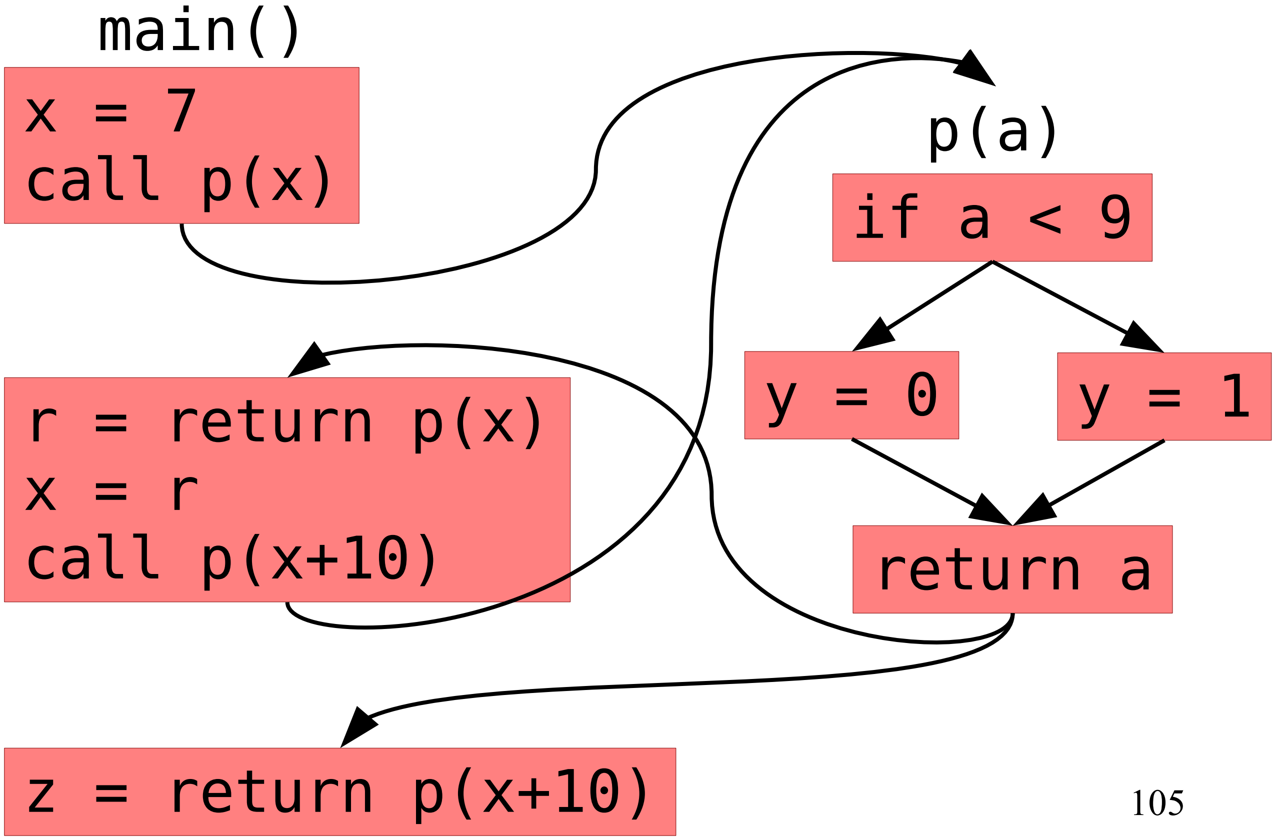
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



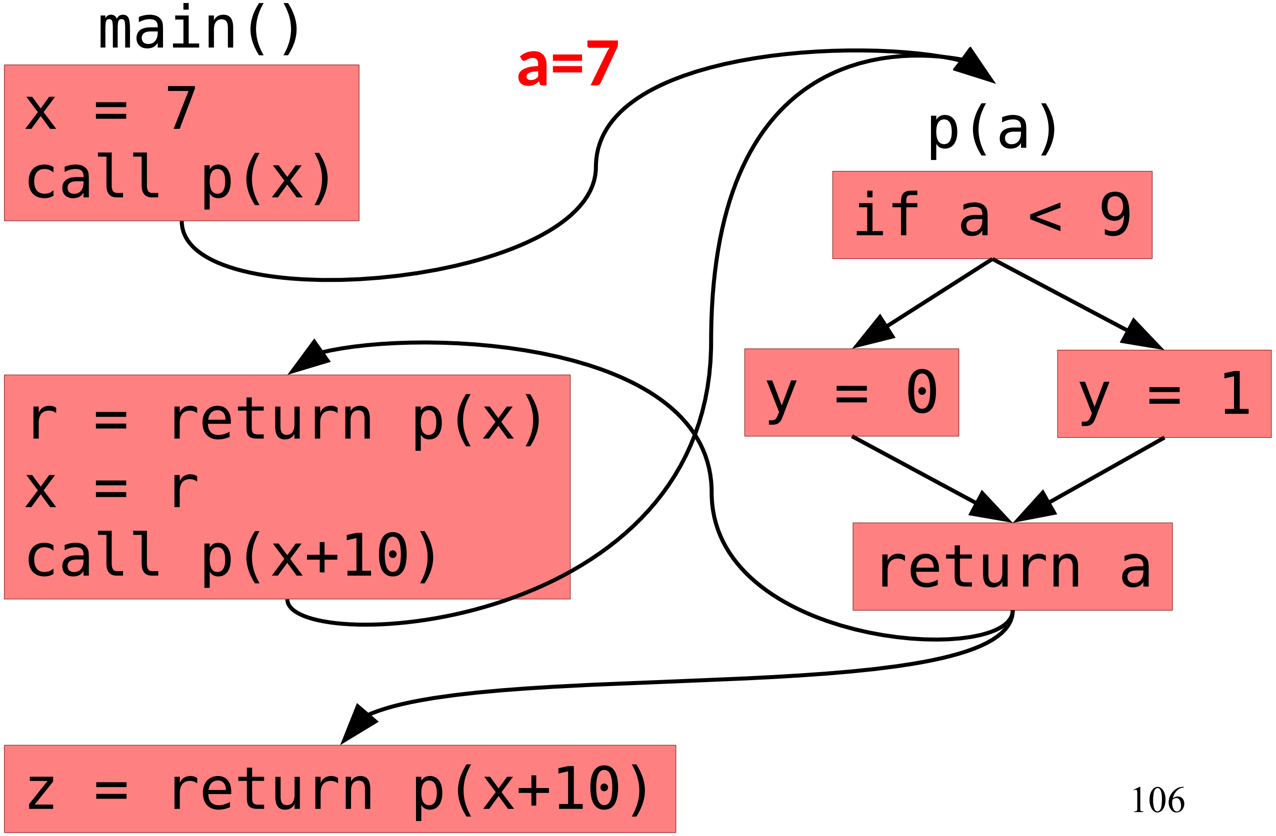
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



main()

```
x = 7
call p(x)
```

a=7

p(a)

```
if a < 9
```

```
y = 0
```

```
y = 1
```

```
return a
```

```
r = return p(x)
x = r
call p(x+10)
```

```
z = return p(x+10)
```

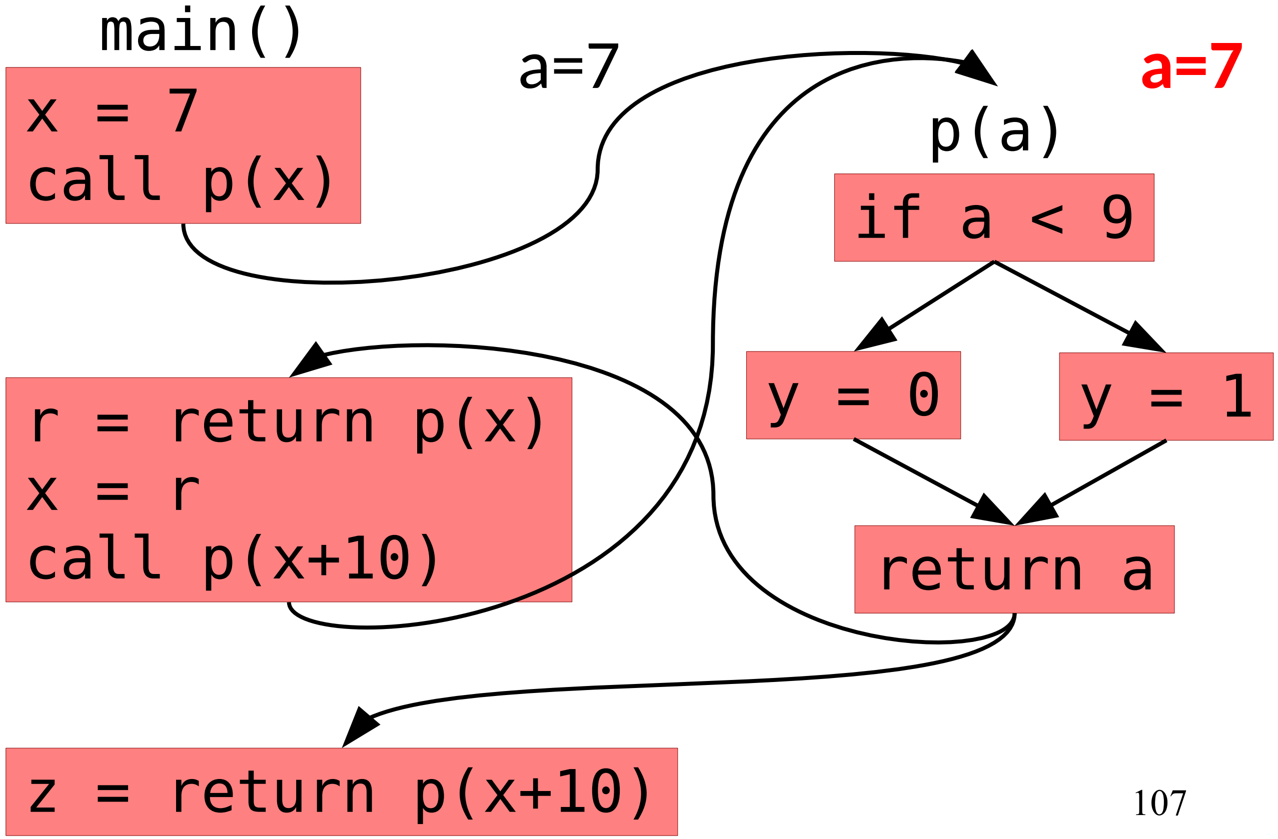
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



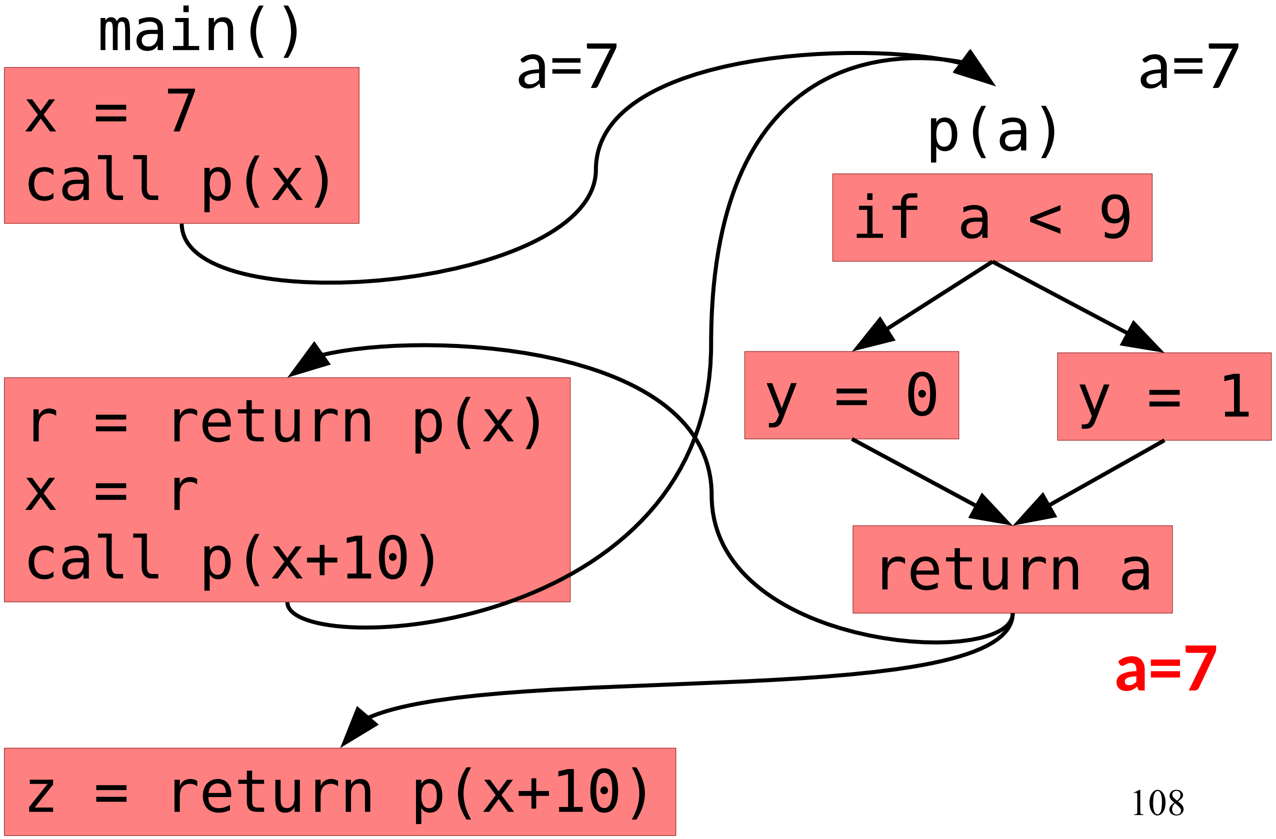
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



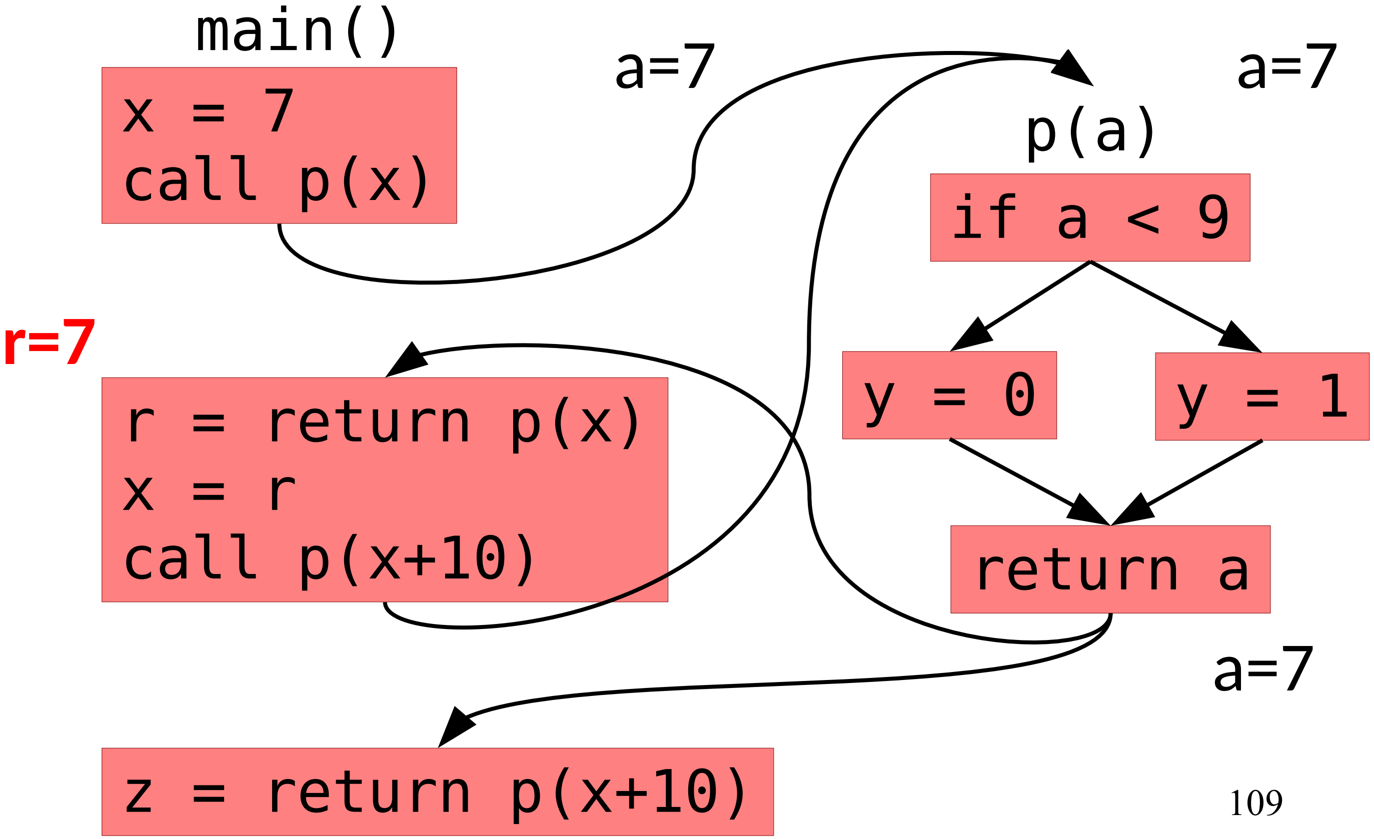
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



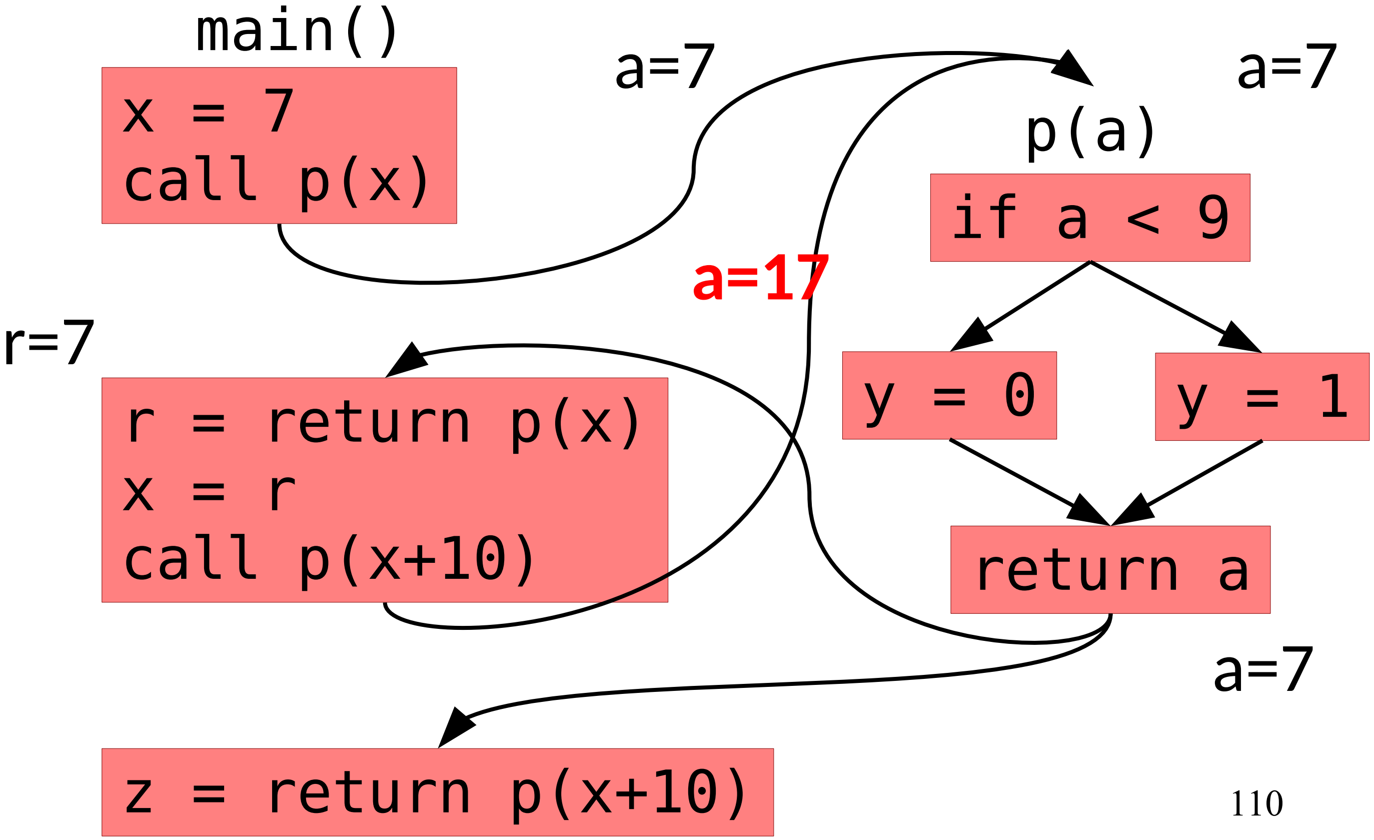
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



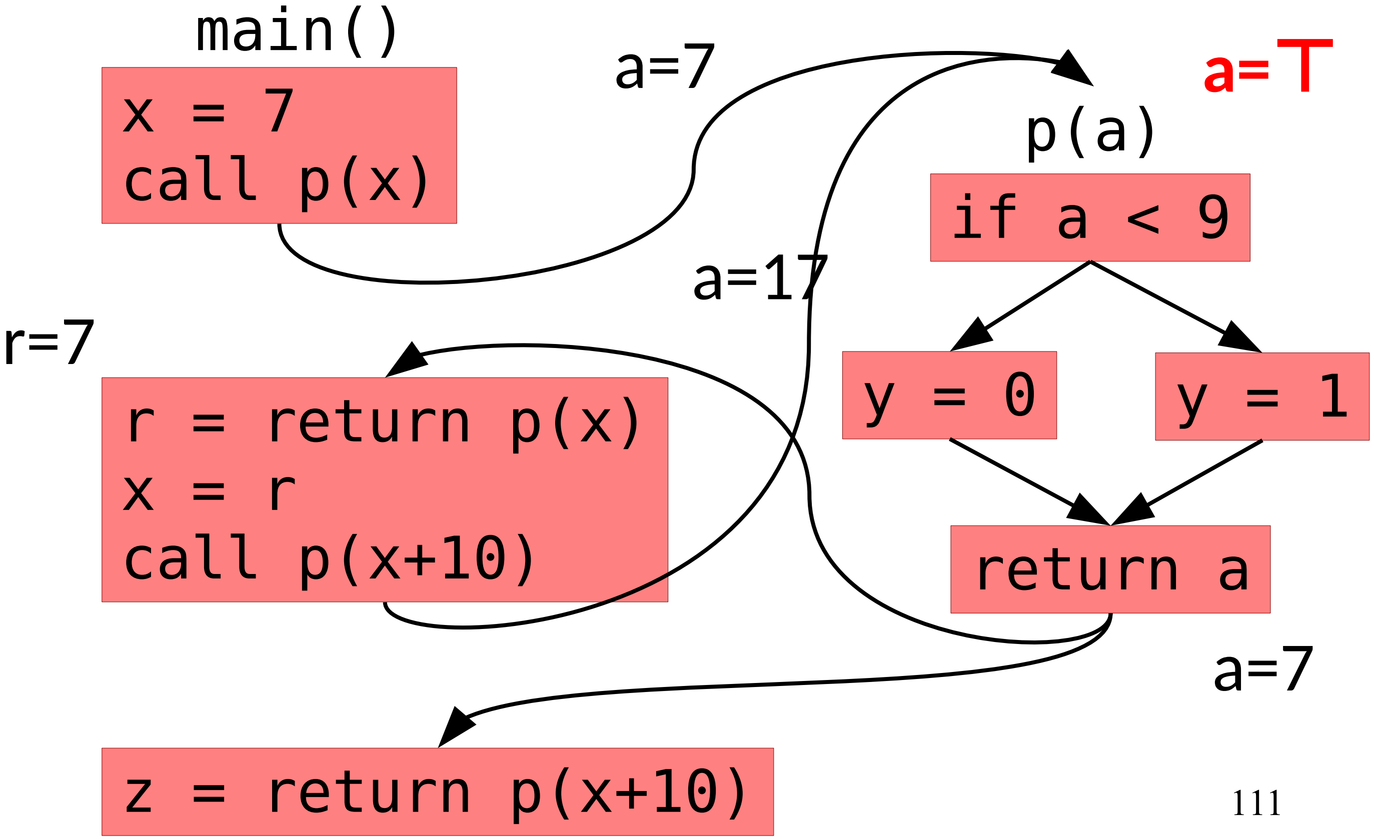
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



main()

```
x = 7
call p(x)
```

a=7

p(a)

a=⊤

```
if a < 9
```

a=17

r=7

```
y = 0
```

```
y = 1
```

```
r = return p(x)
x = r
call p(x+10)
```

```
return a
```

a=7

```
z = return p(x+10)
```

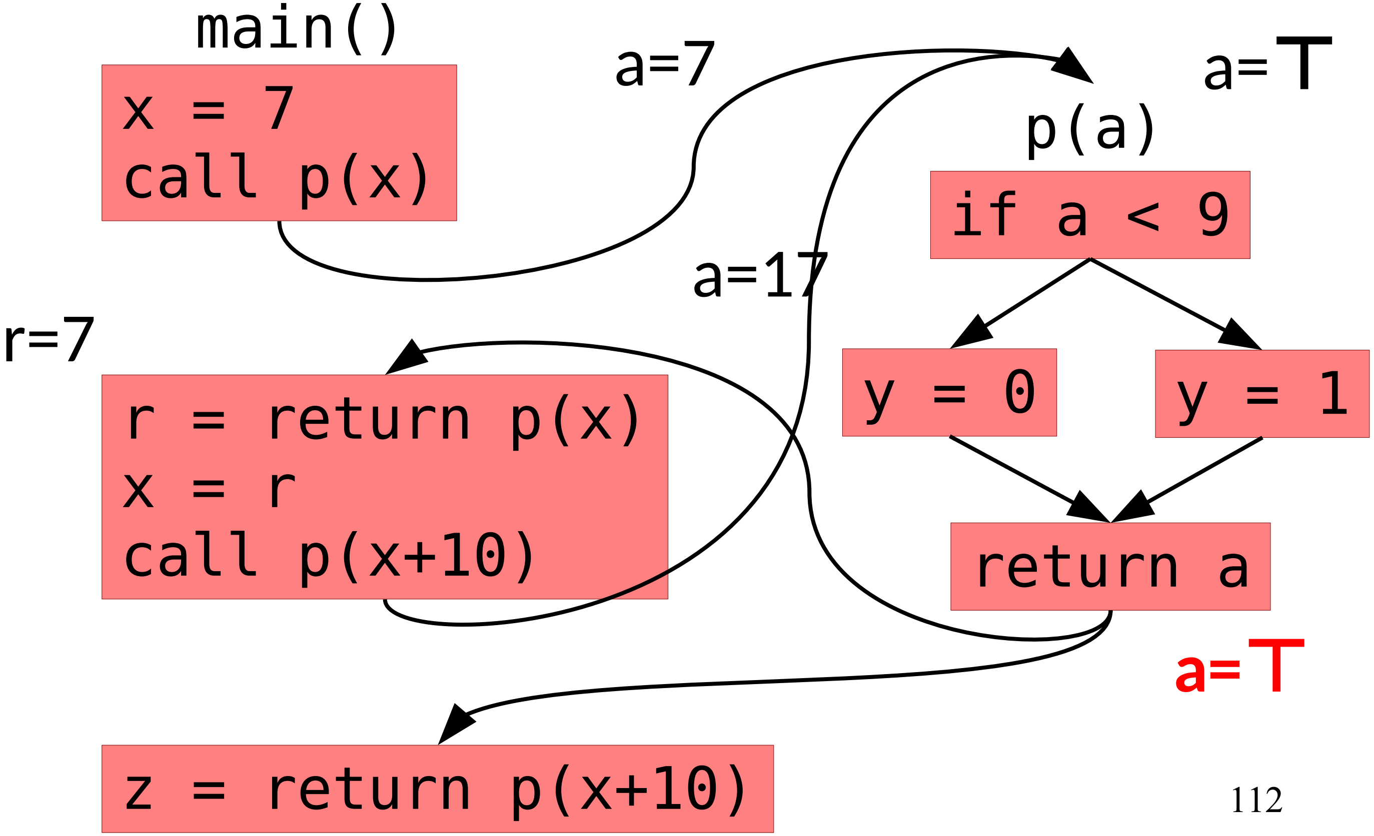
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



main()

a=7

a=⊤

p(a)

```
x = 7
call p(x)
```

```
if a < 9
```

a=17

r=7

```
r = return p(x)
x = r
call p(x+10)
```

```
y = 0
```

```
y = 1
```

```
return a
```

a=⊤

```
z = return p(x+10)
```

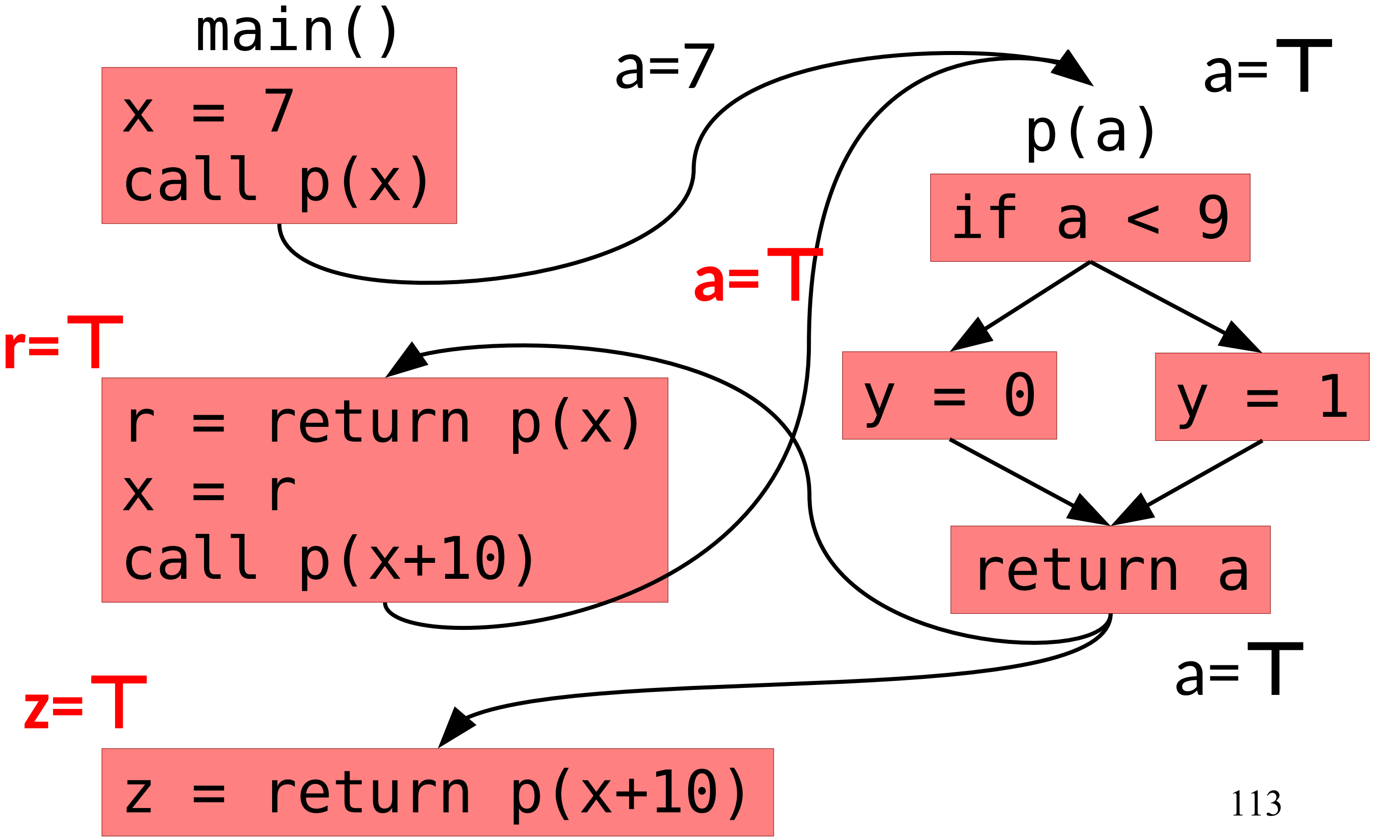
Example from Stephen Chong

Context Sensitivity

- Simplest Approach
  - Add edges between call sites & targets
  - Perform data flow on this larger graph

```
def main():
  x = 7
  r = p(x)
  x = r
  z = p(x+10)
```

```
def p(a):
  if a < 9:
    y = 0
  else:
    y = 1
```



main()

x = 7
call p(x)

a=7

a=⊤

p(a)

if a < 9

y = 0

y = 1

return a

a=⊤

a=⊤

r=⊤

r = return p(x)
x = r
call p(x+10)

z=⊤

z = return p(x+10)

a=⊤

Example from Stephen Chong

Context Sensitivity

- Information from one call site can flow to a mismatched return site!

Context Sensitivity

- Information from one call site can flow to a mismatched return site!
- How could we address it?

Context Sensitivity

- Solution 2: Inlining
  - Make a copy of the function at each call site

Context Sensitivity

- Solution 2: Inlining
  - Make a copy of the function at each call site
- What problems arise?

Context Sensitivity

- Solution 2: Inlining
  - Make a copy of the function at each call site
- What problems arise?
- What other strategies can we use?

Context Sensitivity

- Solution 3: Make a Copy
  - Make one copy of each function per call site

Context Sensitivity

- Solution 3: Make a Copy
  - Make one copy of each function per call site

```
def main():
    a()
    a()
```

```
def a():
    b()
```

```
def b():
    pass
```

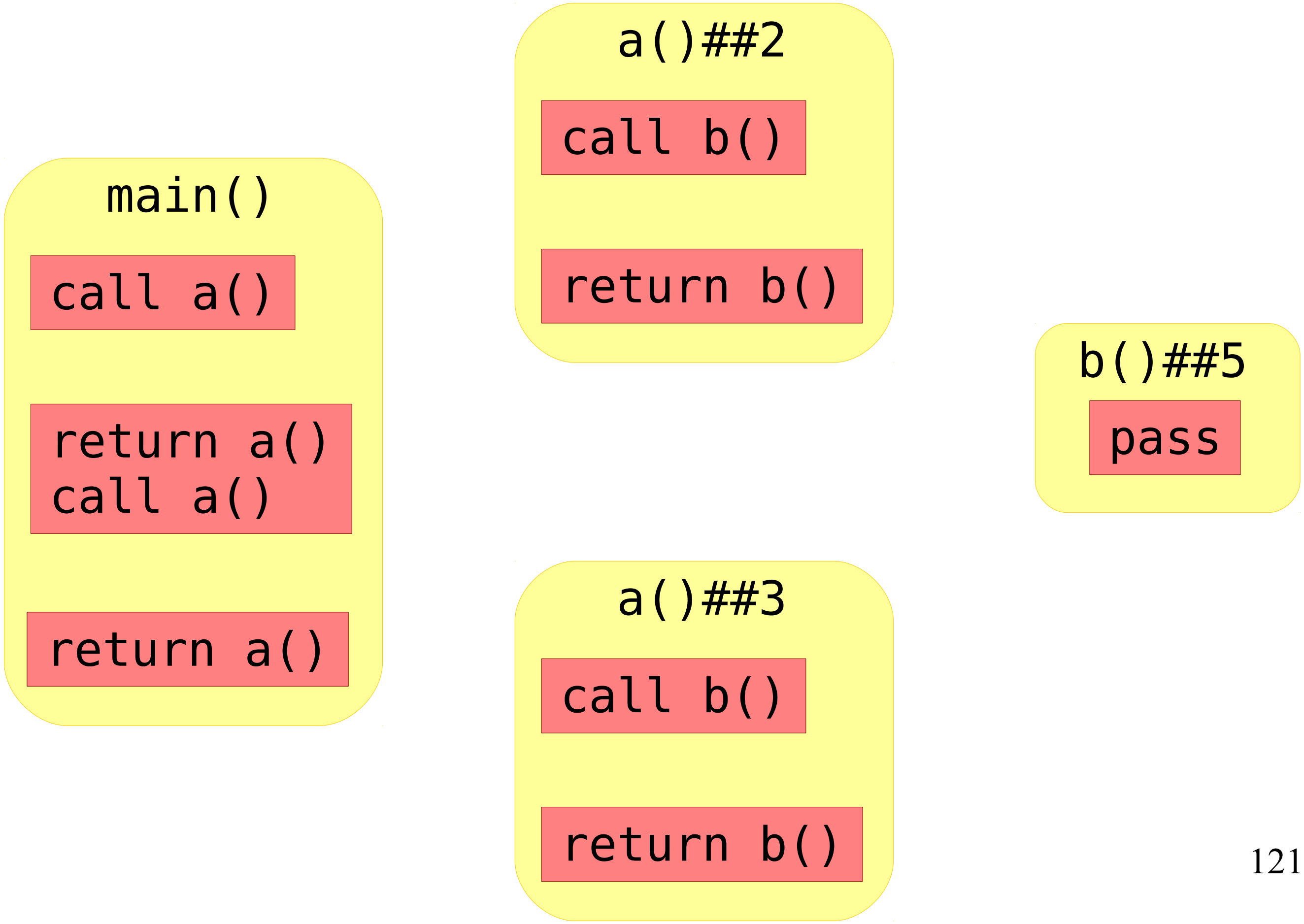
Context Sensitivity

- Solution 3: Make a Copy
  - Make one copy of each function per call site

```
1) def main():
2)     a()
3)     a()
```

```
4) def a():
5)     b()
```

```
6) def b():
7)     pass
```



main()

call a()

return a()
call a()

return a()

a()##2

call b()

return b()

a()##3

call b()

return b()

b()##5

pass

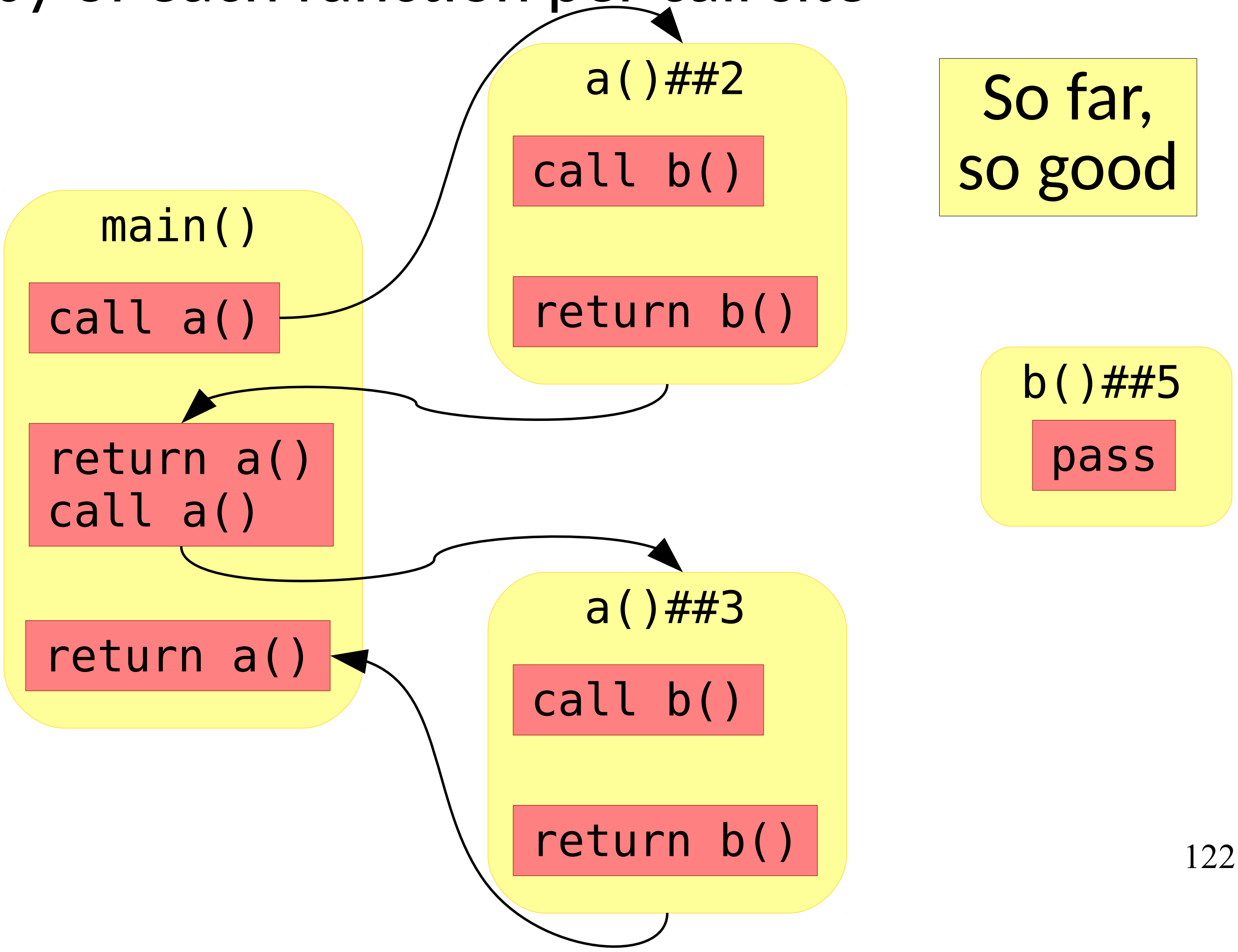
Context Sensitivity

- Solution 3: Make a Copy
  - Make one copy of each function per call site

```
1) def main():
2)     a()
3)     a()
```

```
4) def a():
5)     b()
```

```
6) def b():
7)     pass
```



main()

call a()

return a()
call a()

return a()

a()##2

call b()

return b()

a()##3

call b()

return b()

b()##5

pass

So far, so good

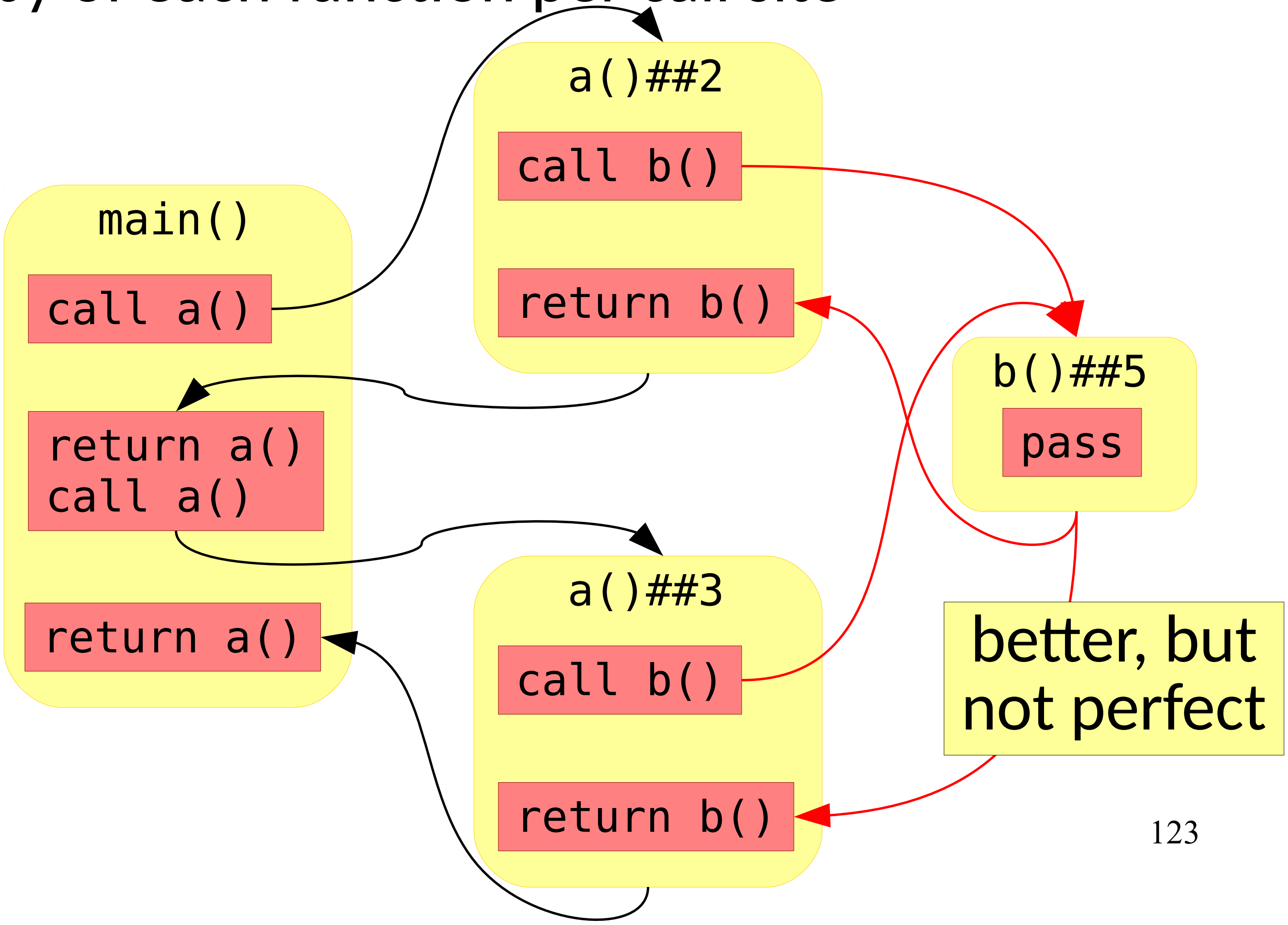
Context Sensitivity

- Solution 3: Make a Copy
  - Make one copy of each function per call site

```
1) def main():
2)    a()
3)    a()
```

```
4) def a():
5)    b()
```

```
6) def b():
7)    pass
```



better, but not perfect

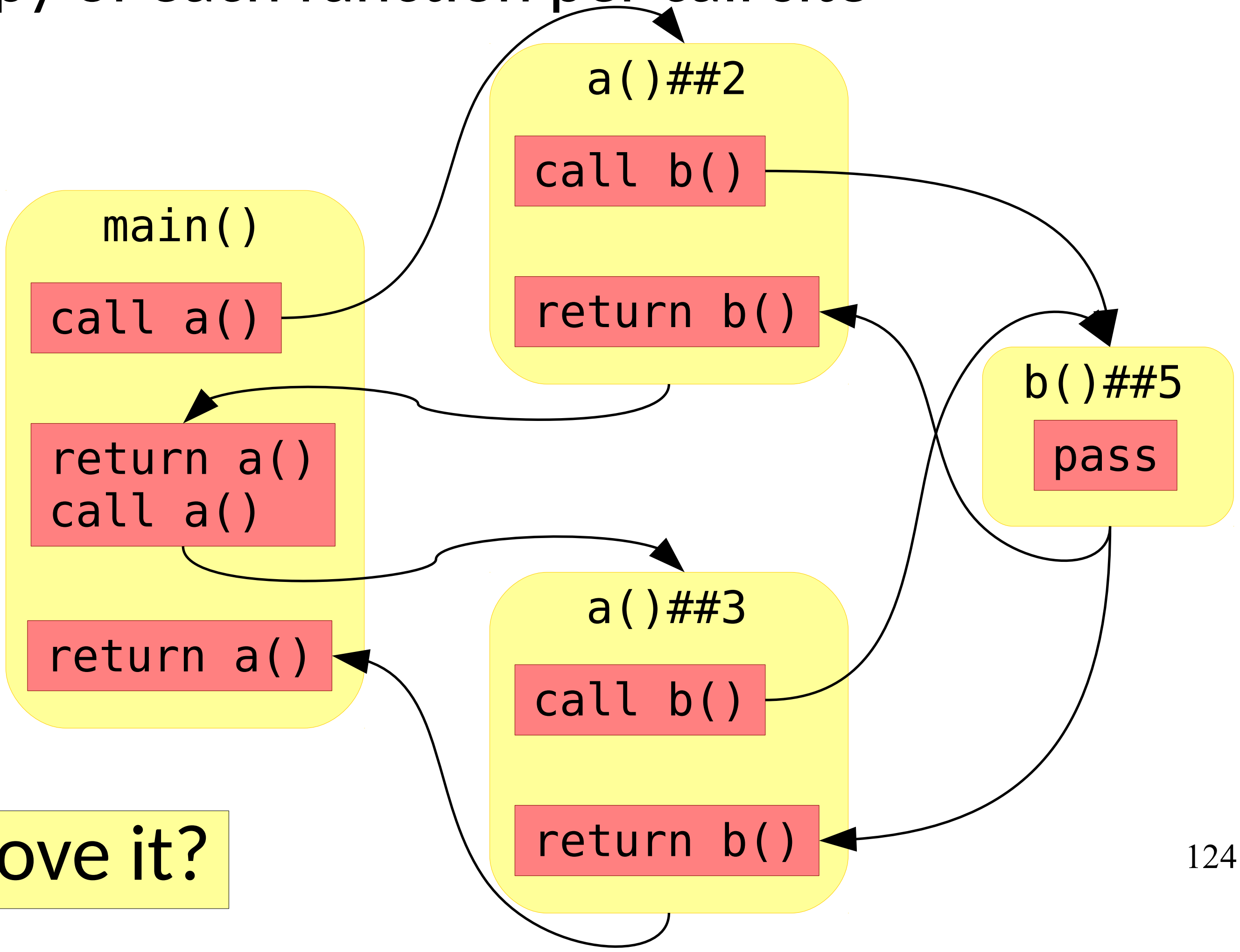
Context Sensitivity

- Solution 3: Make a Copy
  - Make one copy of each function per call site

```
1) def main():
2)   a()
3)   a()
```

```
4) def a():
5)   b()
```

```
6) def b():
7)   pass
```



How can we improve it?

Context Sensitivity

Generalized:

- Make a bounded number of copies

Context Sensitivity

Generalized:

- Make a bounded number of copies
- Choose a key/feature that determines which copy to use
  - Bounded calling context/call stack (*call site sensitivity*)
  - Allocation sites of objects (*object sensitivity*)

Context Sensitivity

- Solution 4: Make a *logical* copy

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis
  - Compute results (called *procedure summaries*) for each logical copy of a function.

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis
  - Compute results (called *procedure summaries*) for each logical copy of a function.
  - Modify the treatment of calls slightly:

    On `foo(in)` with context C:

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis
  - Compute results (called *procedure summaries*) for each logical copy of a function.
  - Modify the treatment of calls slightly:

    On `foo(in)` with context C:

      If (foo,C) doesn't have a summary, process foo(in) in C and save the result to S.

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis
  - Compute results (called *procedure summaries*) for each logical copy of a function.
  - Modify the treatment of calls slightly:

    On `foo(in)` with context C:

      If (foo,C) doesn't have a summary, process foo(in) in C and save the result to S.

      If the summary S already approximates foo(in), use S

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis
  - Compute results (called *procedure summaries*) for each logical copy of a function.
  - Modify the treatment of calls slightly:

    On `foo(in)` with context C:

      If (foo,C) doesn't have a summary, process foo(in) in C and save the result to S.

      If the summary S already approximates foo(in), use S

      Otherwise, process foo(in) in C and update S with (in$\sqcap$S.in).

Context Sensitivity

- Solution 4: Make a *logical* copy
  - Instead of actually making a copy, just keep track of the context information (the key) during analysis
  - Compute results (called *procedure summaries*) for each logical copy of a function.
  - Modify the treatment of calls slightly:

On `foo(in)` with context C:

If (foo,C) doesn't have a summary, process foo(in) in C and save the result to S.

If the summary S already approximates foo(in), use S

Otherwise, process foo(in) in C and update S with (in$\sqcap$S.in).

If the result changes, reprocess all callers of (foo,C)

Context Sensitivity

- In some cases, context sensitive analysis can be reduced to special forms of graph reachability.

Dataflow Configurations

Can be configured in many ways:

Dataflow Configurations

Can be configured in many ways:

- Forward / Backward (e.g. reaching vs liveness)

Dataflow Configurations

Can be configured in many ways:

- Forward / Backward (e.g. reaching vs liveness)
- May / Must ($\cup$ vs $\cap$ in lattice when paths $\prod$)

Dataflow Configurations

Can be configured in many ways:

- Forward / Backward (e.g. reaching vs liveness)
- May / Must (∪ vs ∩ in lattice when paths ∏)
- Sensitivity {Path? Flow? Context?}

Dataflow Configurations

Can be configured in many ways:

- Forward / Backward (e.g. reaching vs liveness)
- May / Must (∪ vs ∩ in lattice when paths ∏)
- Sensitivity {Path? Flow? Context?}

The configuration is ultimately driven by the property/problem of interest

Static Analysis

- We've already seen a few static analyses:
  - Call graph construction
  - Points-to graph construction (What are MAY/MUST?)
  - Static slicing

Static Analysis

- We've already seen a few static analyses:
  - Call graph construction
  - Points-to graph construction (What are MAY/MUST?)
  - Static slicing
- The choices for approximation are why these analyses are imprecise.

Other (Traditionally) Static Approaches

- Type based analyses
- Bounded state exploration
- Symbolic execution
- Model checking

Many of these have been integrated into *dynamic* analyses, as we shall see over the semester.

Static Analysis Summary

- Considers all possible executions

Static Analysis Summary

- Considers all possible executions
- Approximates program behavior to fight undecidability

Static Analysis Summary

- Considers all possible executions
- Approximates program behavior to fight undecidability
- Can answer queries like:
    - Must my program always …?
    - May my program ever …?

Static Analysis Summary

- Considers all possible executions
- Approximates program behavior to fight undecidability
- Can answer queries like:
  - Must my program always …?
  - May my program ever …?
- Dataflow analysis is one common form of static analysis