AAA616: Program Analysis

Lecture 2 — Static Analysis Examples

Hakjoo Oh
2024 Fall

Principles of Static Analysis

$$30 \times 12 + 11 \times 9 = ?$$

2. "Execute" the program with abstract values

- Dynamic analysis (testing): 459
- Static analysis: a variety of answers
  - "integer" (type system)
  - "odd integer"
  - "positive integer"
  - "integer between 400 and 500"
  - …

1. Choose abstract value (domain)

$$e \,\hat{\times}\, e \,\hat{+}\, o \,\hat{\times}\, o = o$$

$$e \,\hat{\times}\, e = e \qquad e \,\hat{+}\, e = e$$

$$e \,\hat{\times}\, o = e \qquad e \,\hat{+}\, o = o$$

$$o \,\hat{\times}\, e = e \qquad o \,\hat{+}\, e = o$$

$$o \,\hat{\times}\, o = o \qquad o \,\hat{+}\, o = e$$

Strength of Static Analysis

- By contrast to testing, static analysis can prove the absence of bugs

Even

⊤ (don’t know)

```
void f (int x) {
    y = x * 12 + 9 * 11;
    assert (y % 2 == 1);
}
```

Odd

Odd

Strength of Static Analysis

- By contrast to program verification, static analysis can prove the absence of bugs automatically

```
@pre: n >= 0
@post: rv == n
int SimpleWhile (int n) {
   int i = 0;
   while
   @L: 0 <= i <= n
   (i < n) {
      i = i + 1;
   }
}
```

Weakness of Static Analysis

- Instead, static analysis may produce false alarms

⊤ (don’t know)

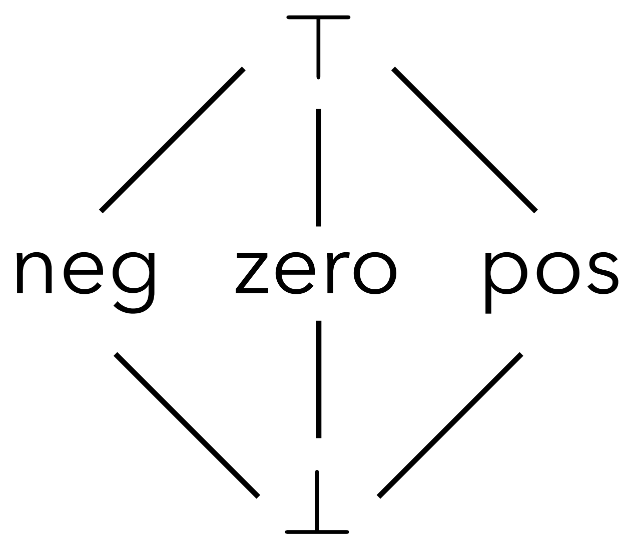
```
void f (int x) {
    y = x + x;
    assert (y % 2 == 0);
}
```

⊤ (don’t know)

false alarm

A Simple Sign Domain

- Abstract values



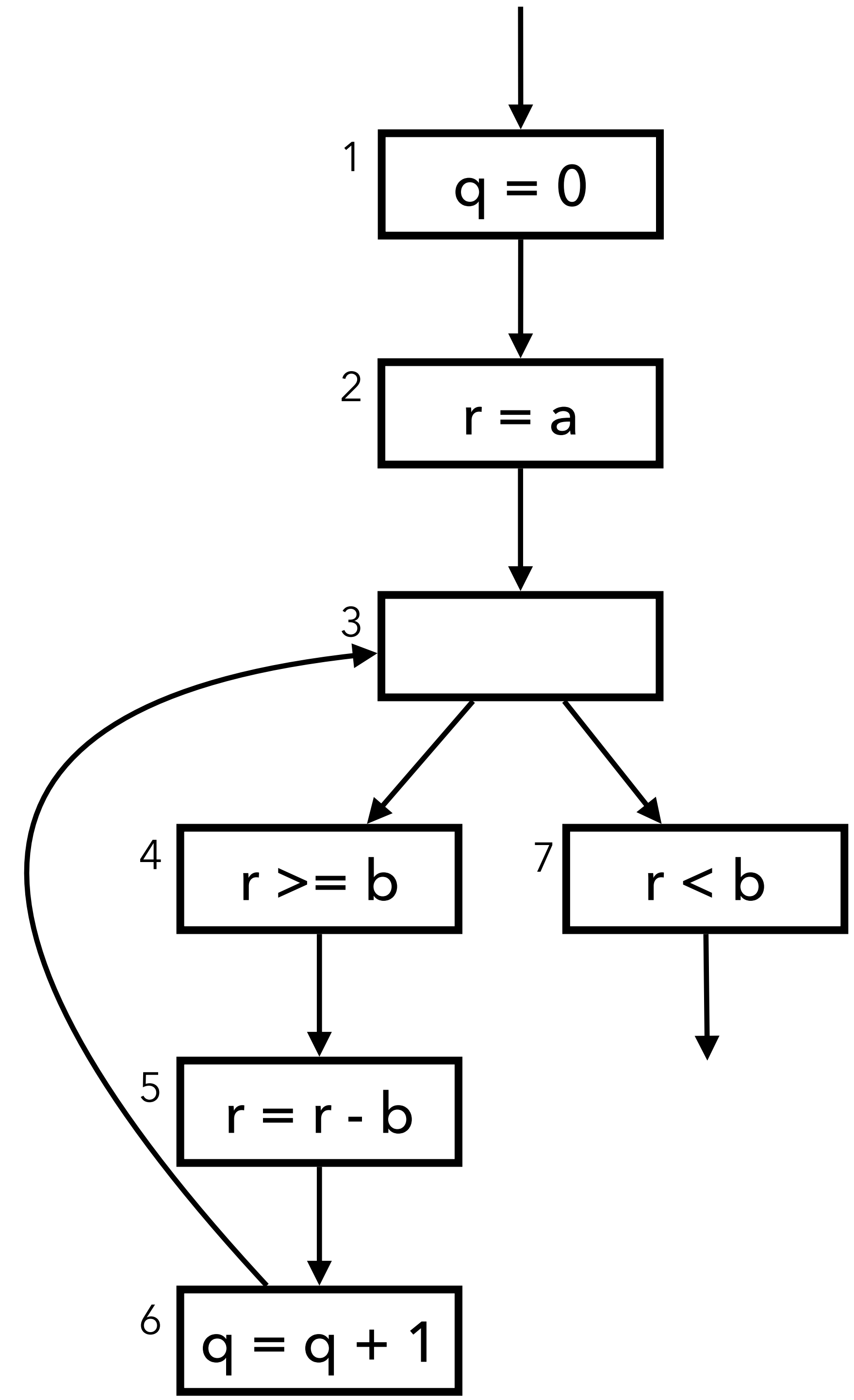
- Abstract operators

+/−	top	neg	zero	pos	bot
top					
neg					
zero					
pos					
bot					

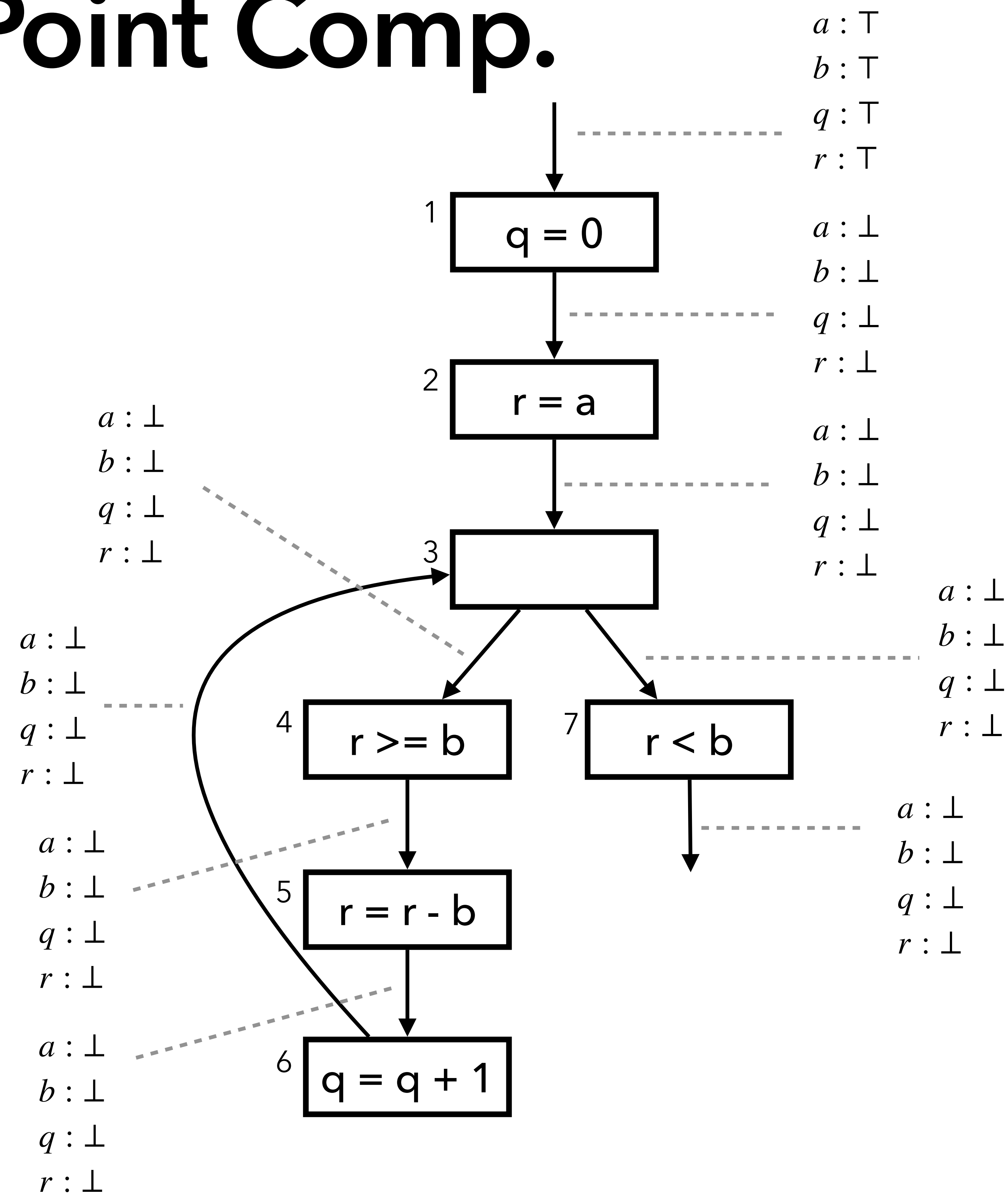
×	top	neg	zero	pos	bot
top					
neg					
zero					
pos					
bot					

Example Program

```
// a >= 0, b >= 0
q = 0;
r = a;
while (r >= b) {
  r = r - b;
  q = q + 1;
}
assert(q >= 0);
assert(r >= 0);
```

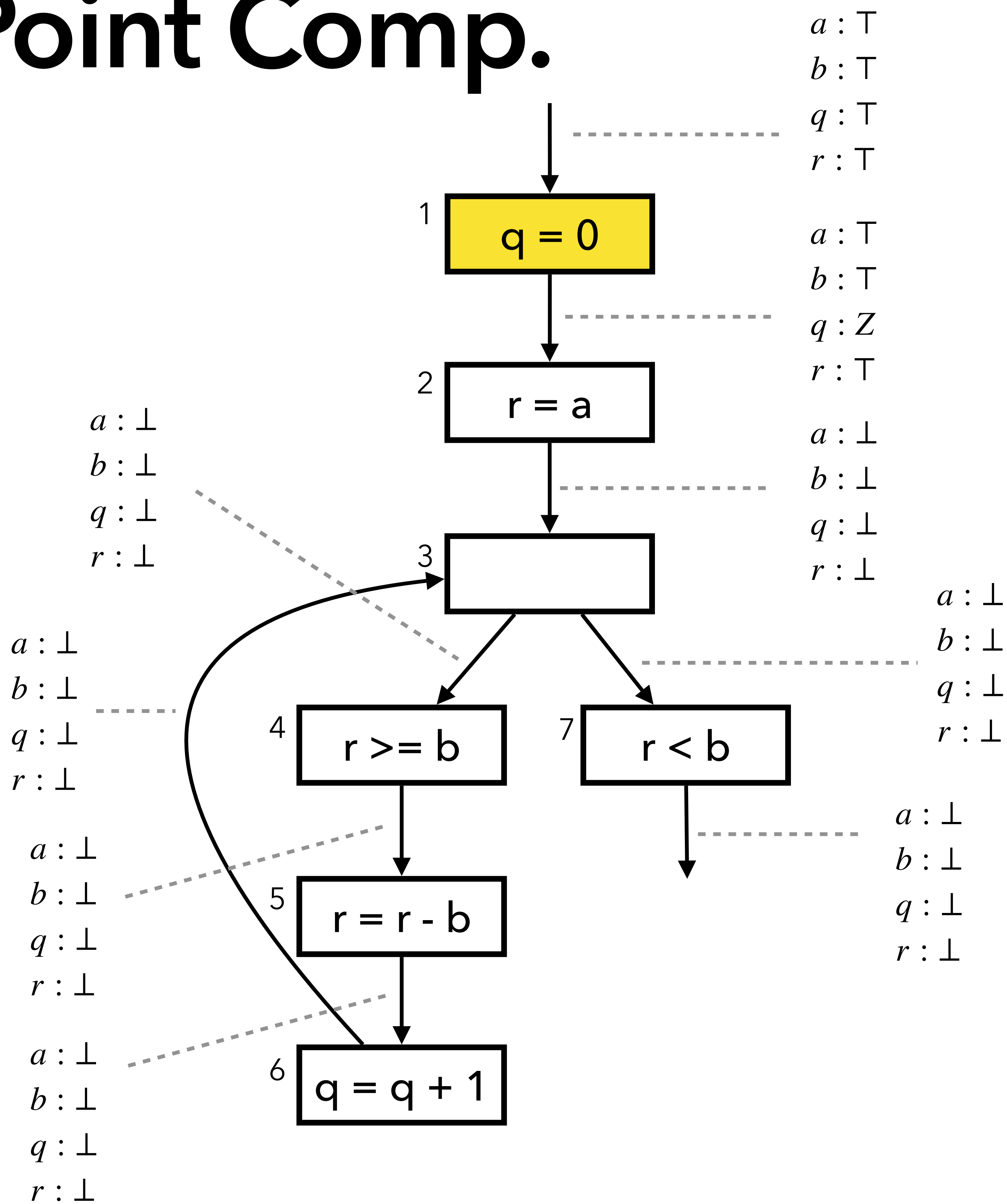


Fixed Point Comp.



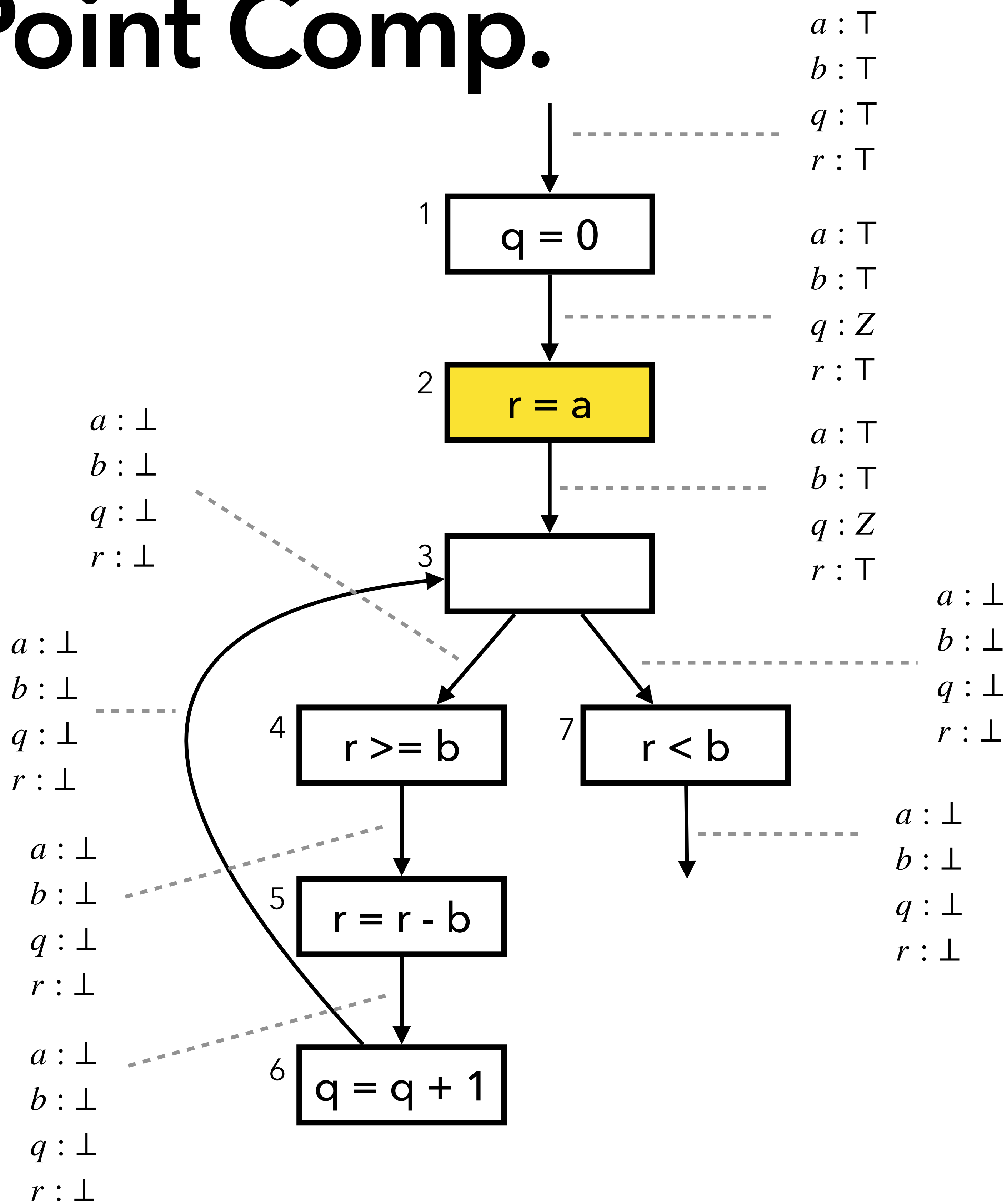
W = { 1, 2, 3, 4, 5, 6, 7 }

Fixed Point Comp.



W = { ~~1~~, 2, 3, 4, 5, 6, 7 }

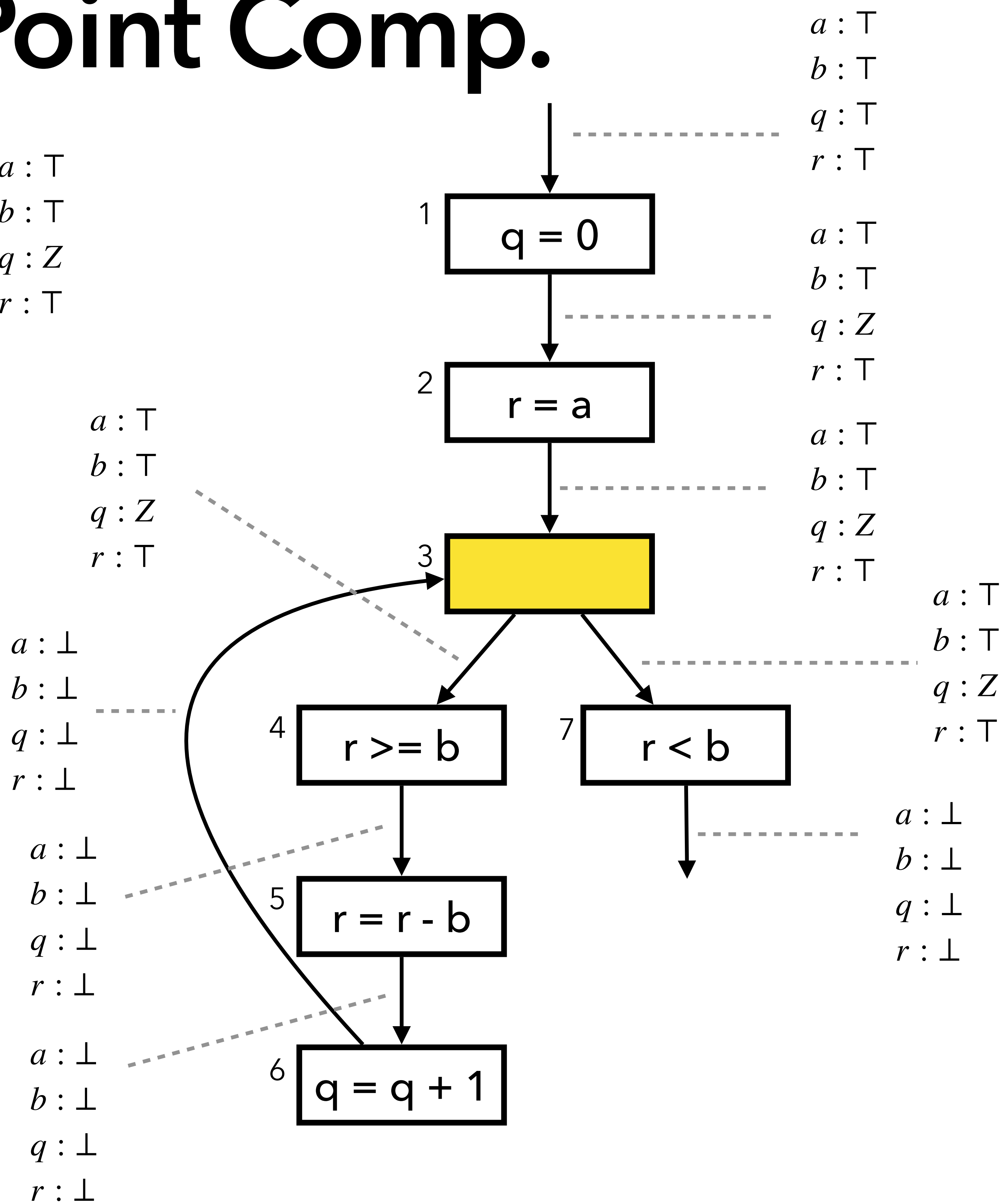
Fixed Point Comp.



W = { ~~1~~, ~~2~~, 3, 4, 5, 6, 7 }

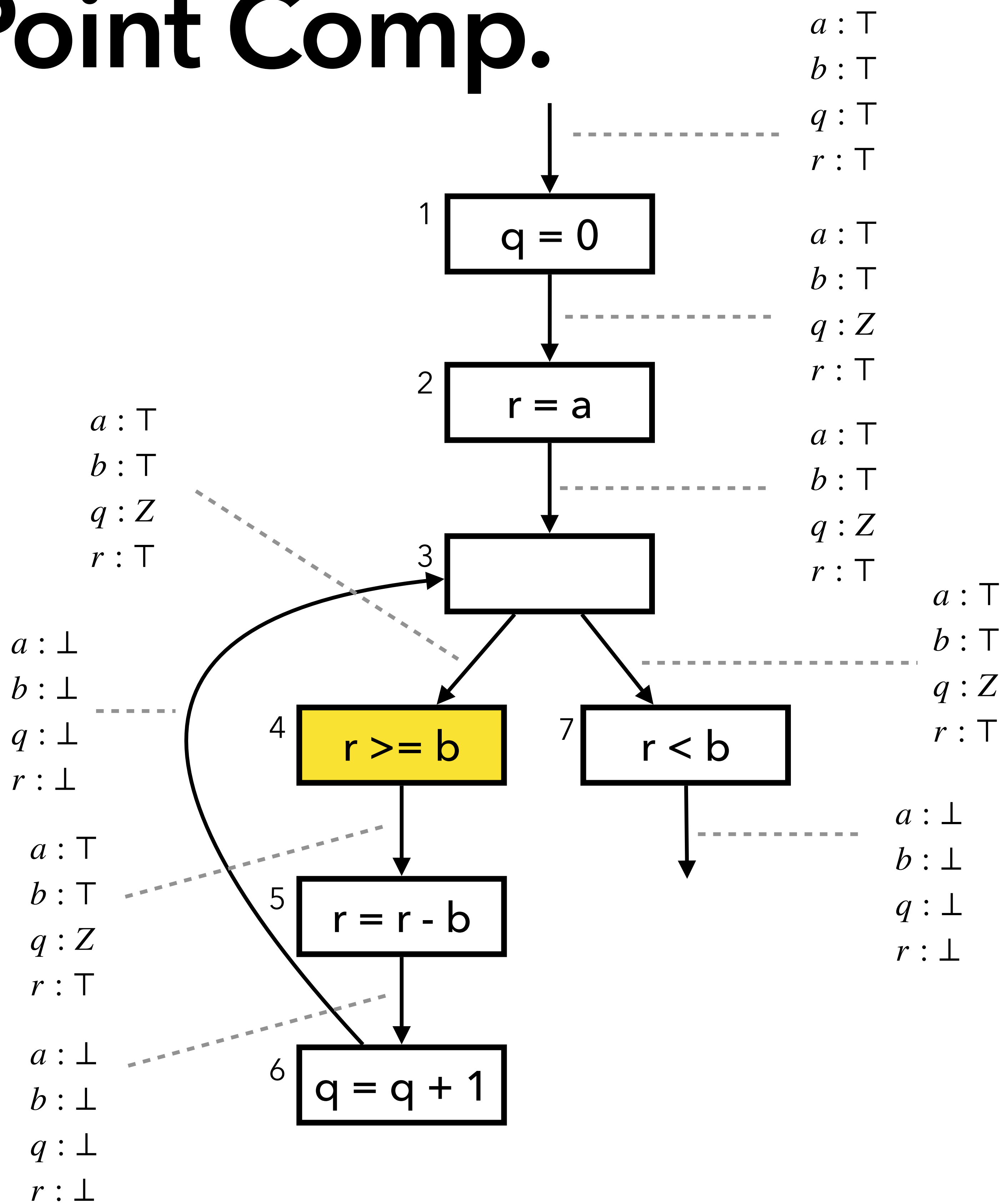
Fixed Point Comp.

$$\begin{matrix} a:\top \\ b:\top \\ q:Z \\ r:\top \end{matrix} \sqcup \begin{matrix} a:\bot \\ b:\bot \\ q:\bot \\ r:\bot \end{matrix} = \begin{matrix} a:\top \\ b:\top \\ q:Z \\ r:\top \end{matrix}$$



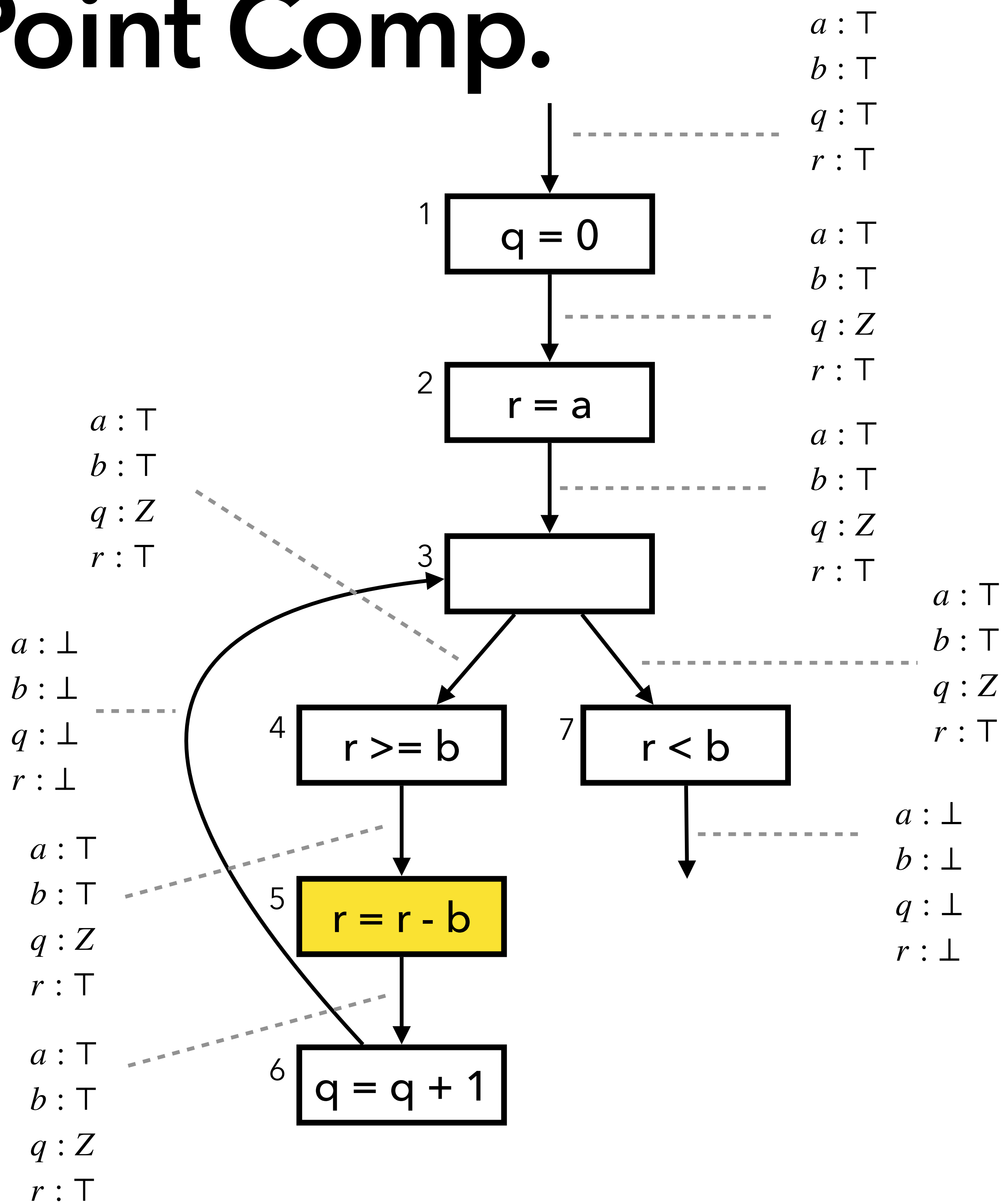
W = { ~~1~~, ~~2~~, ~~3~~, 4, 5, 6, 7 }

Fixed Point Comp.



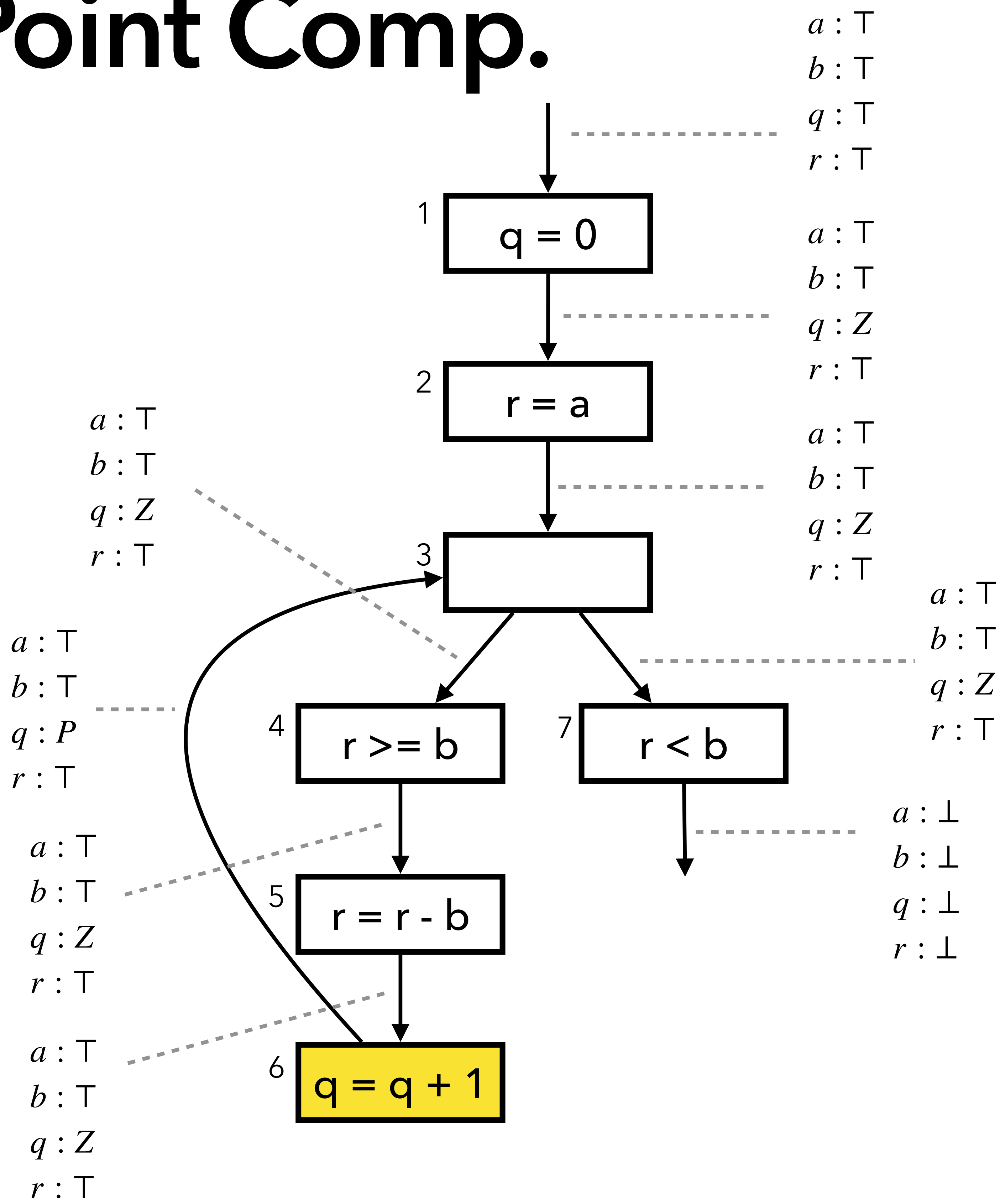
W = { ~~1~~, ~~2~~, ~~3~~, ~~4~~, 5, 6, 7 }

Fixed Point Comp.



W = { ~~1~~, ~~2~~, ~~3~~, 4, ~~5~~, 6, 7 }

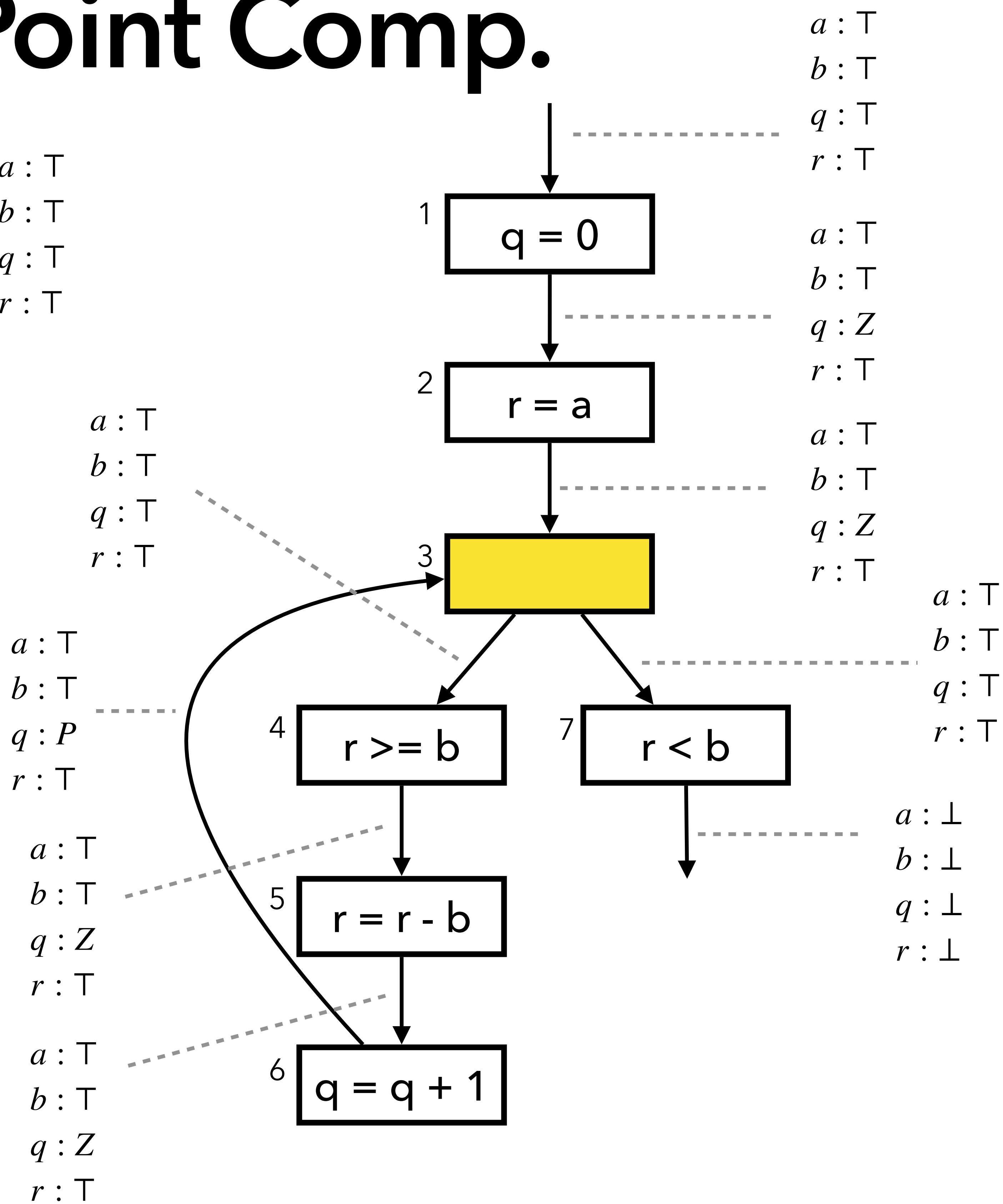
Fixed Point Comp.



W = { ~~1~~, ~~2~~, 3, 4, ~~5~~, ~~6~~, 7 }

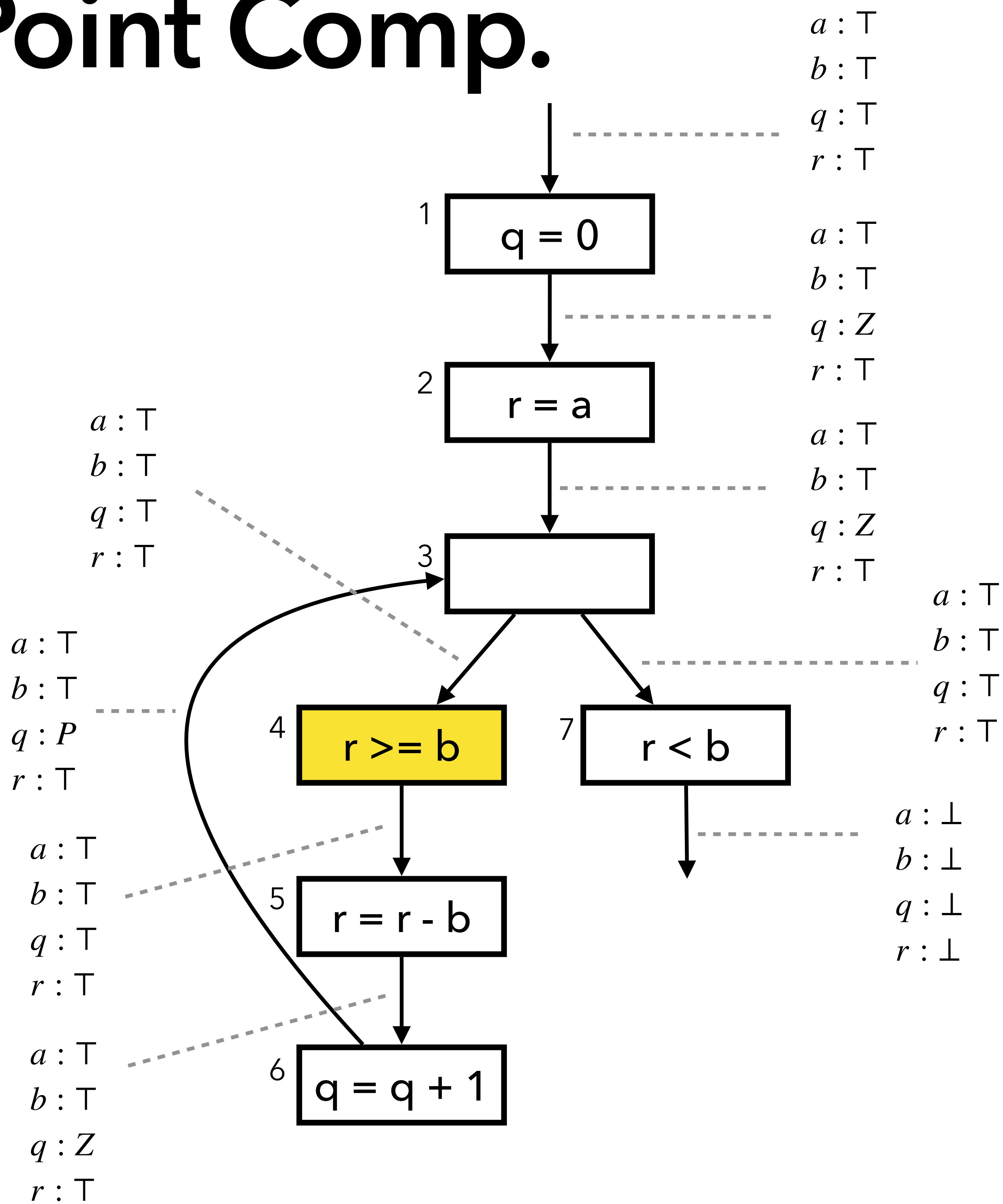
Fixed Point Comp.

$$\begin{matrix} a:\top \\ b:\top \\ q:Z \\ r:\top \end{matrix} \sqcup \begin{matrix} a:\top \\ b:\top \\ q:P \\ r:\top \end{matrix} = \begin{matrix} a:\top \\ b:\top \\ q:\top \\ r:\top \end{matrix}$$



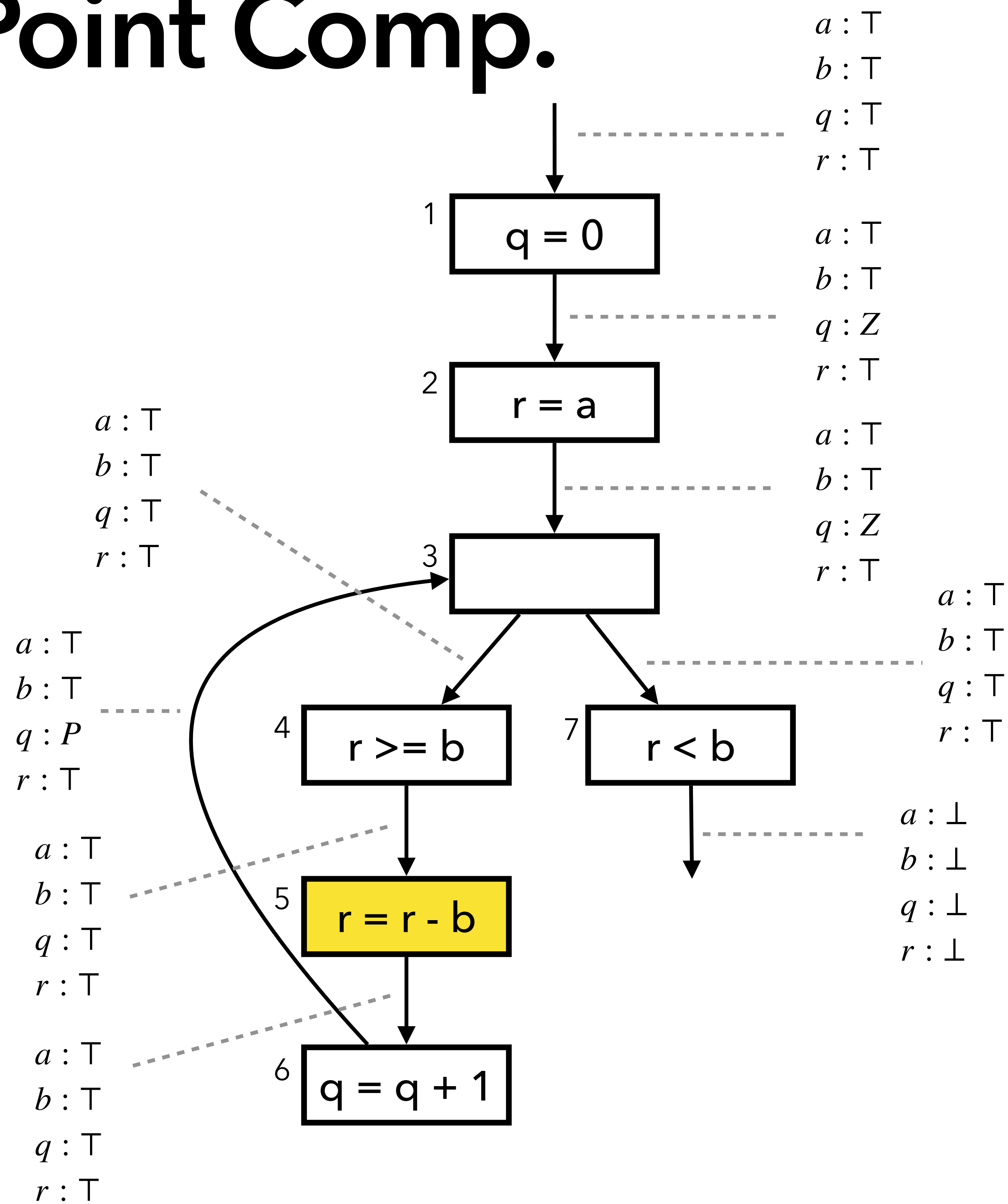
W = { ~~1~~, ~~2~~, ~~3~~, 4, ~~5~~, ~~6~~, 7 }

Fixed Point Comp.



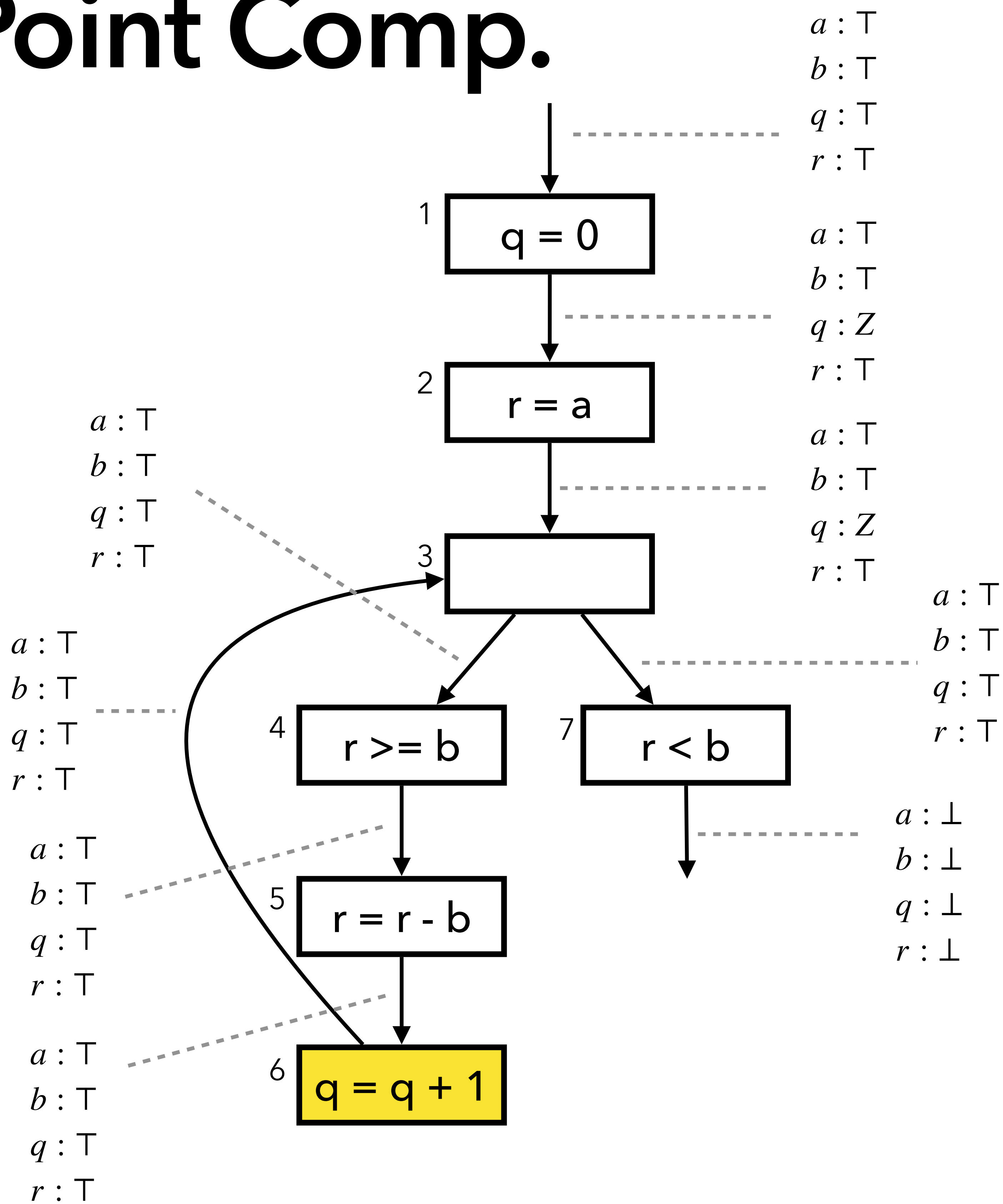
W = { ~~1~~, ~~2~~, ~~3~~, ~~4~~, 5, ~~6~~, 7 }

Fixed Point Comp.



W = { ~~1~~, ~~2~~, ~~3~~, 4, ~~5~~, 6, 7 }

Fixed Point Comp.

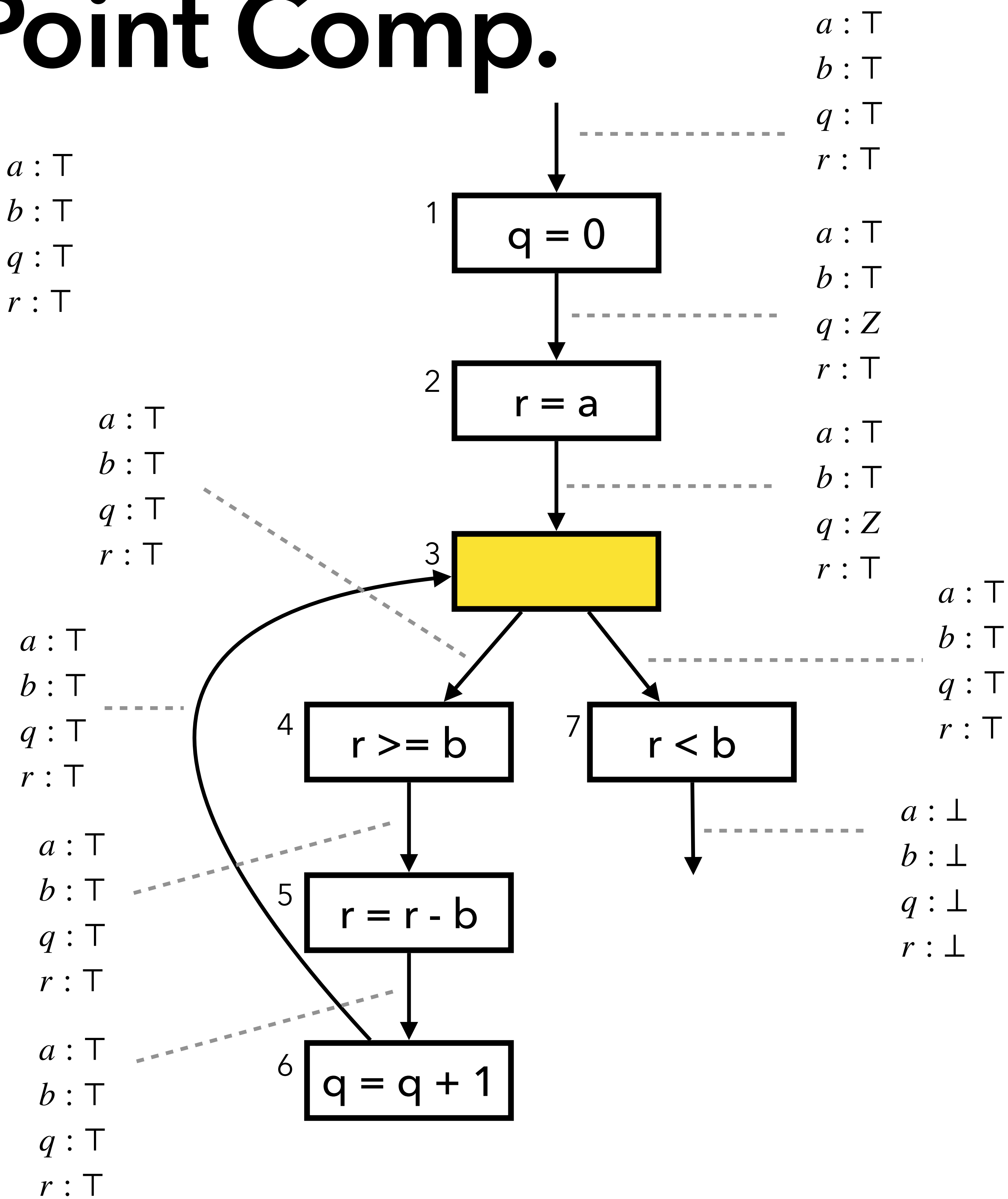


W = { ~~1~~, ~~2~~, 3, 4, ~~5~~, ~~6~~, 7 }

Fixed Point Comp.

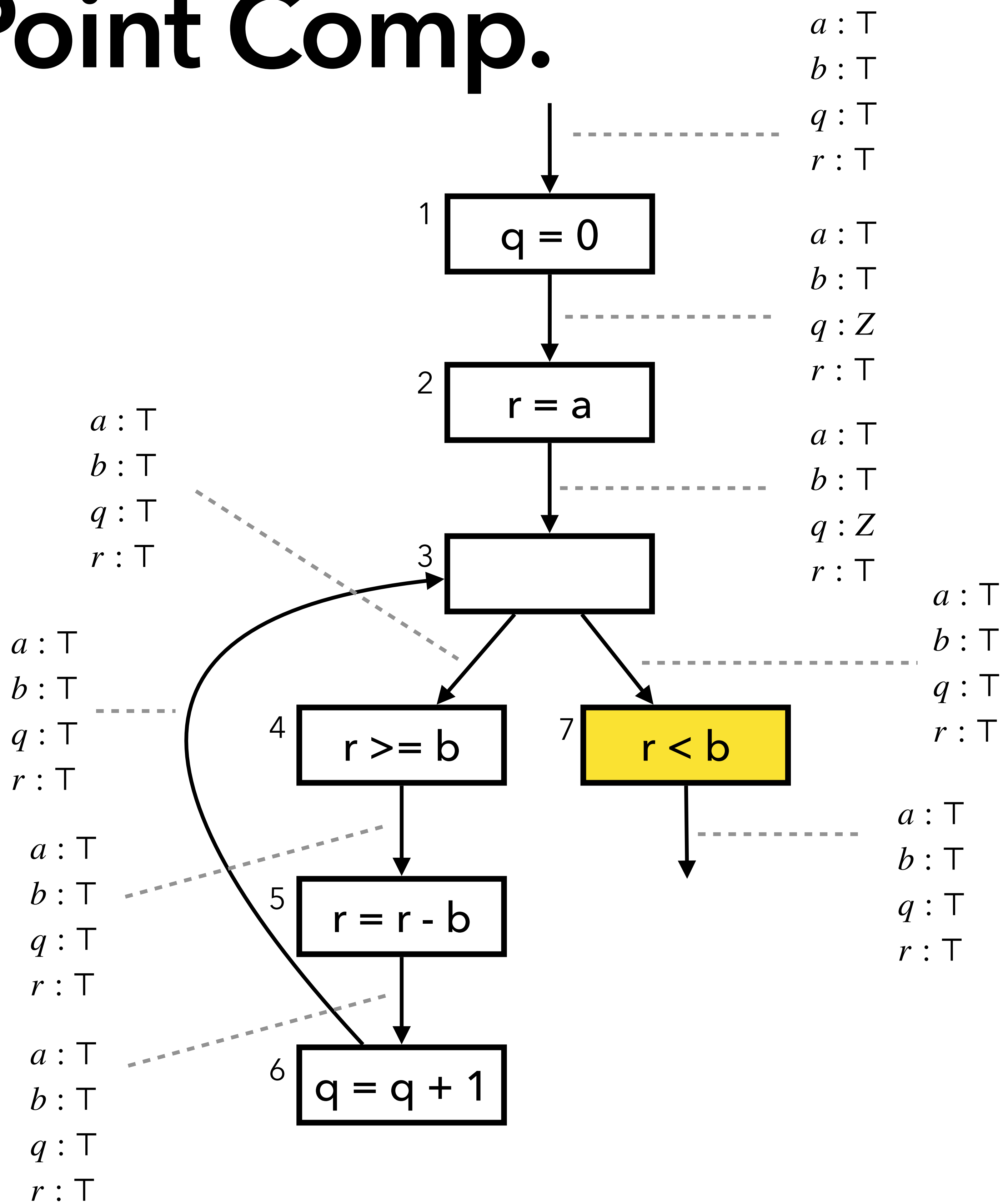
$$\begin{array}{l} a : \top \\ b : \top \\ q : Z \\ r : \top \end{array} \sqcup \begin{array}{l} a : \top \\ b : \top \\ q : \top \\ r : \top \end{array} = \begin{array}{l} a : \top \\ b : \top \\ q : \top \\ r : \top \end{array}$$

(fixed point)



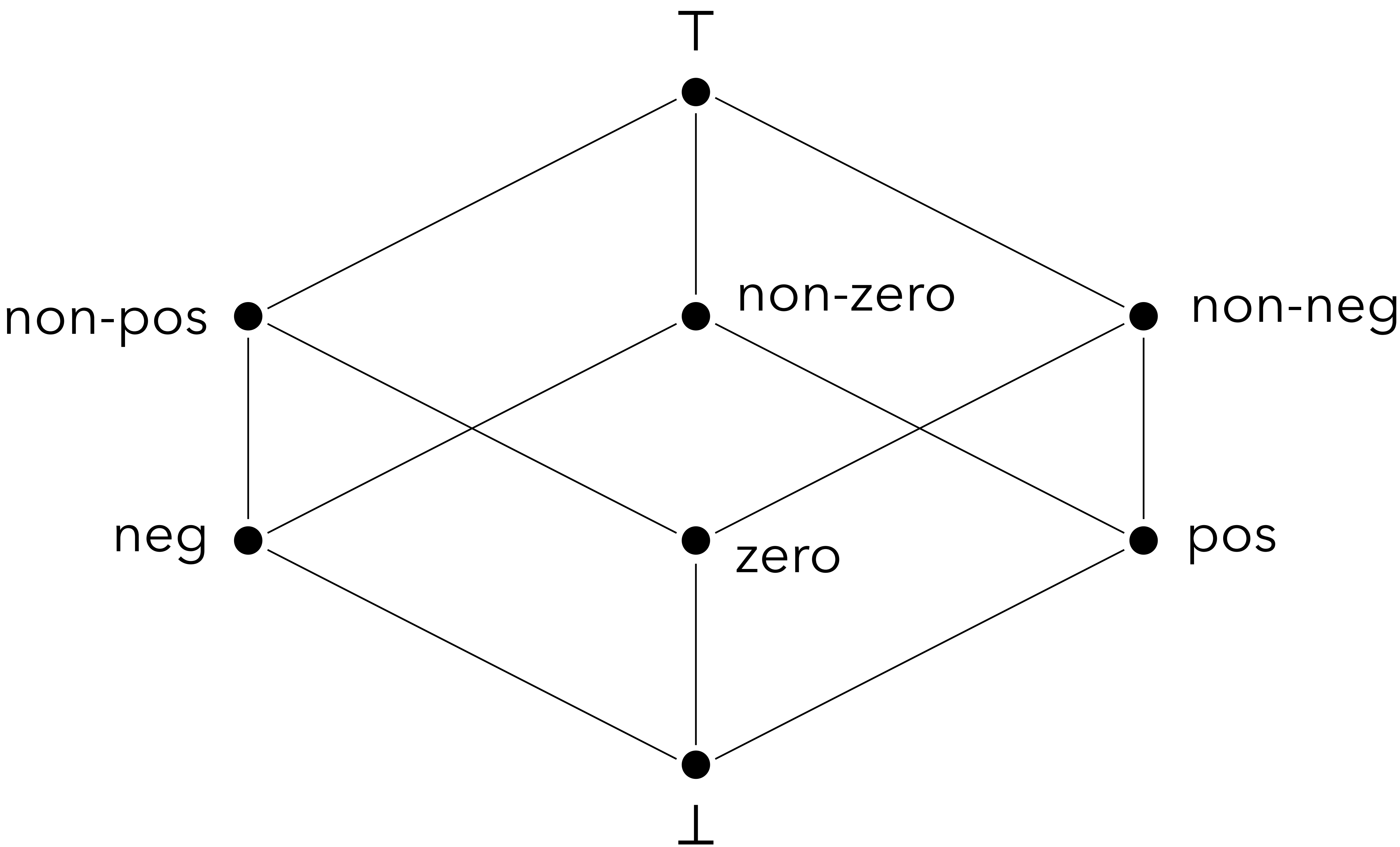
W = { ~~1~~, ~~2~~, ~~3~~, 4, ~~5~~, ~~6~~, 7 }

Fixed Point Comp.



W = { ~~1~~, ~~2~~, ~~3~~, 4, ~~5~~, ~~6~~, ~~7~~ }

An Extended Sign Domain



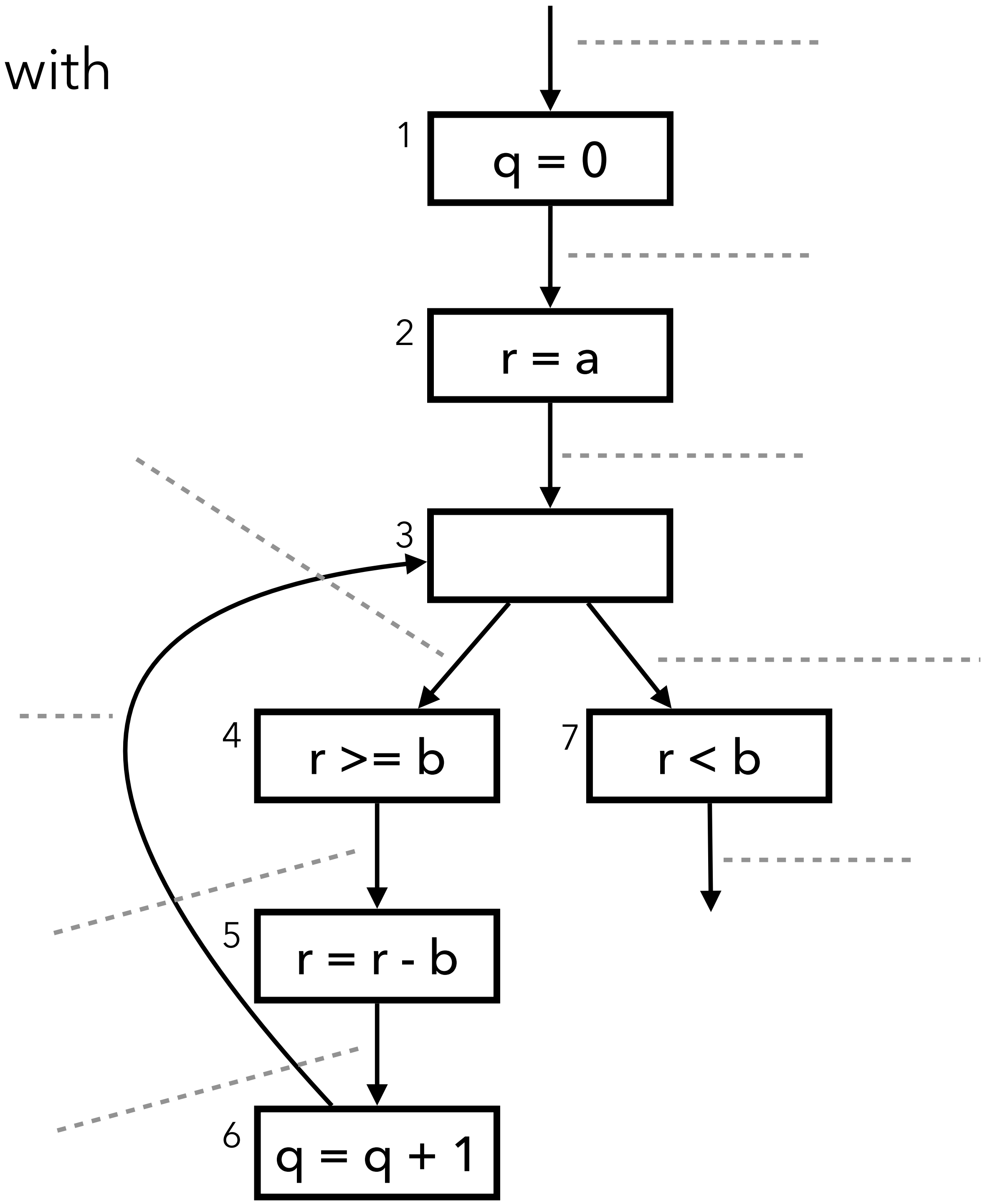
+	top	neg	zero	pos	non-pos	non-zero	non-neg	bot
top								
neg								
zero								
pos								
non-pos								
non-zero								
non-neg								
bot								

−	top	neg	zero	pos	non-pos	non-zero	non-neg	bot
top								
neg								
zero								
pos								
non-pos								
non-zero								
non-neg								
bot								

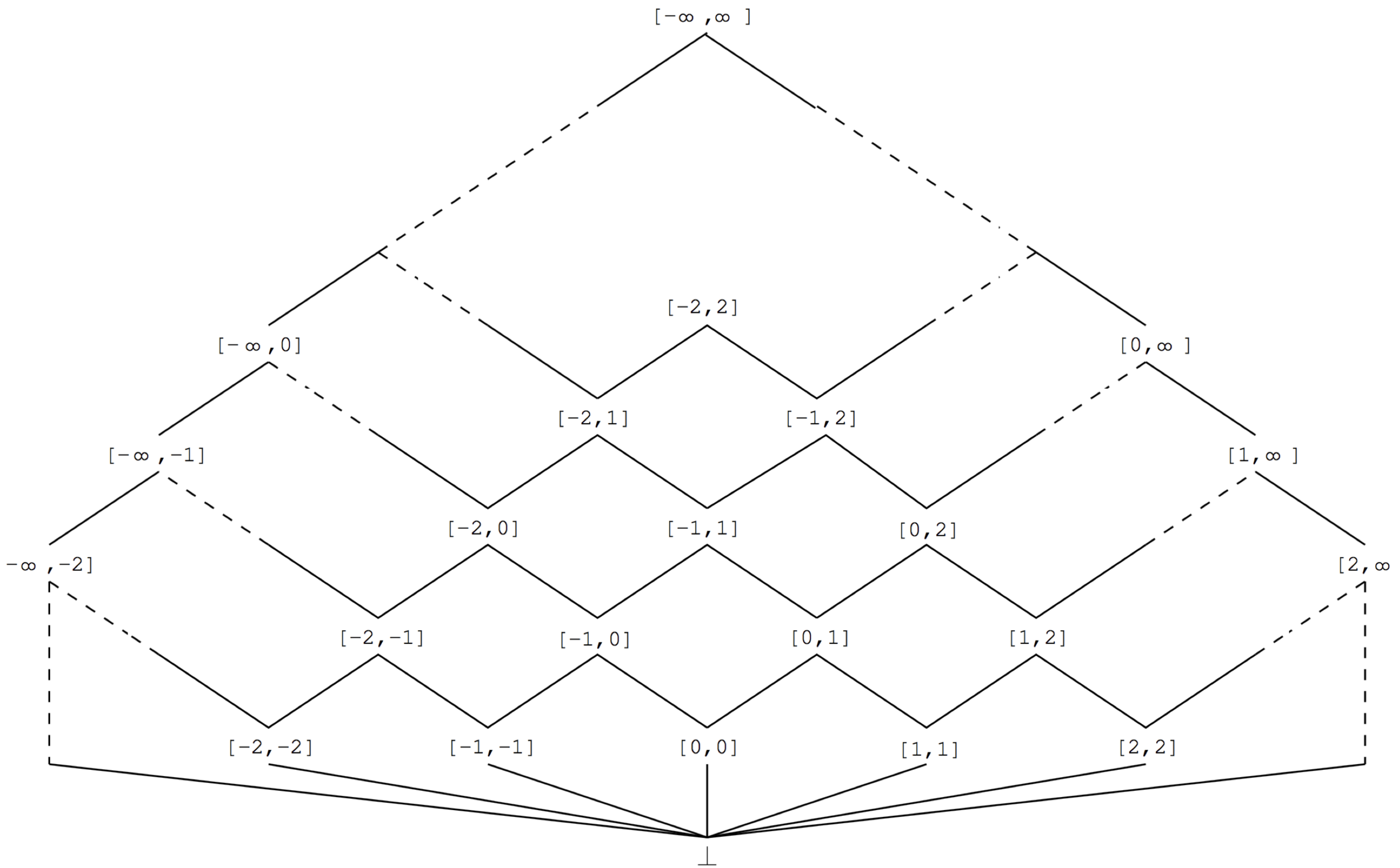
Exercise (1)

Describe the result of the analysis with the extended sign domain

```
// a >= 0, b >= 0
q = 0;
r = a;
while (r >= b) {
  r = r - b;
  q = q + 1;
}
assert(q >= 0);
assert(r >= 0);
```

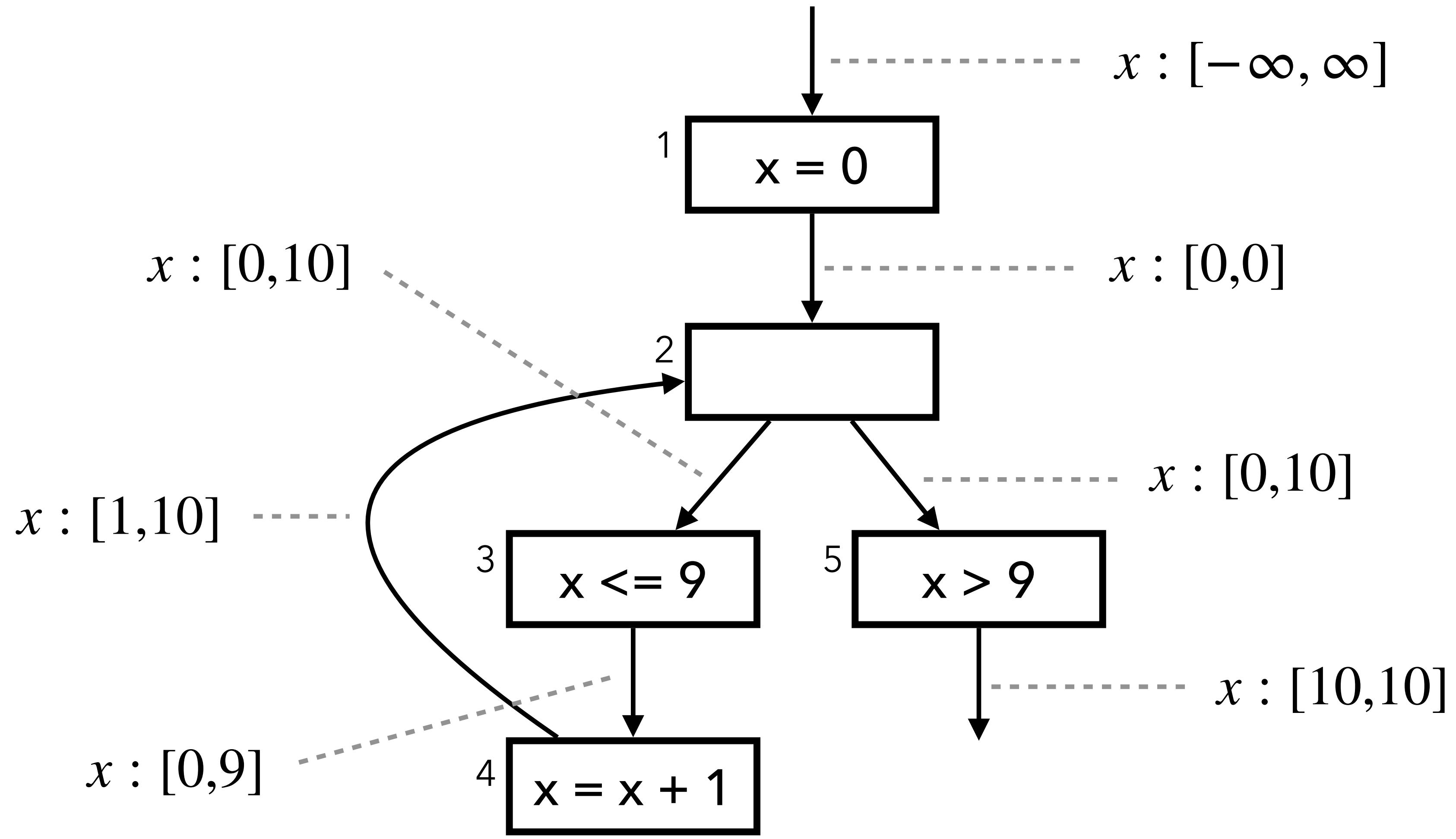


The Interval Domain

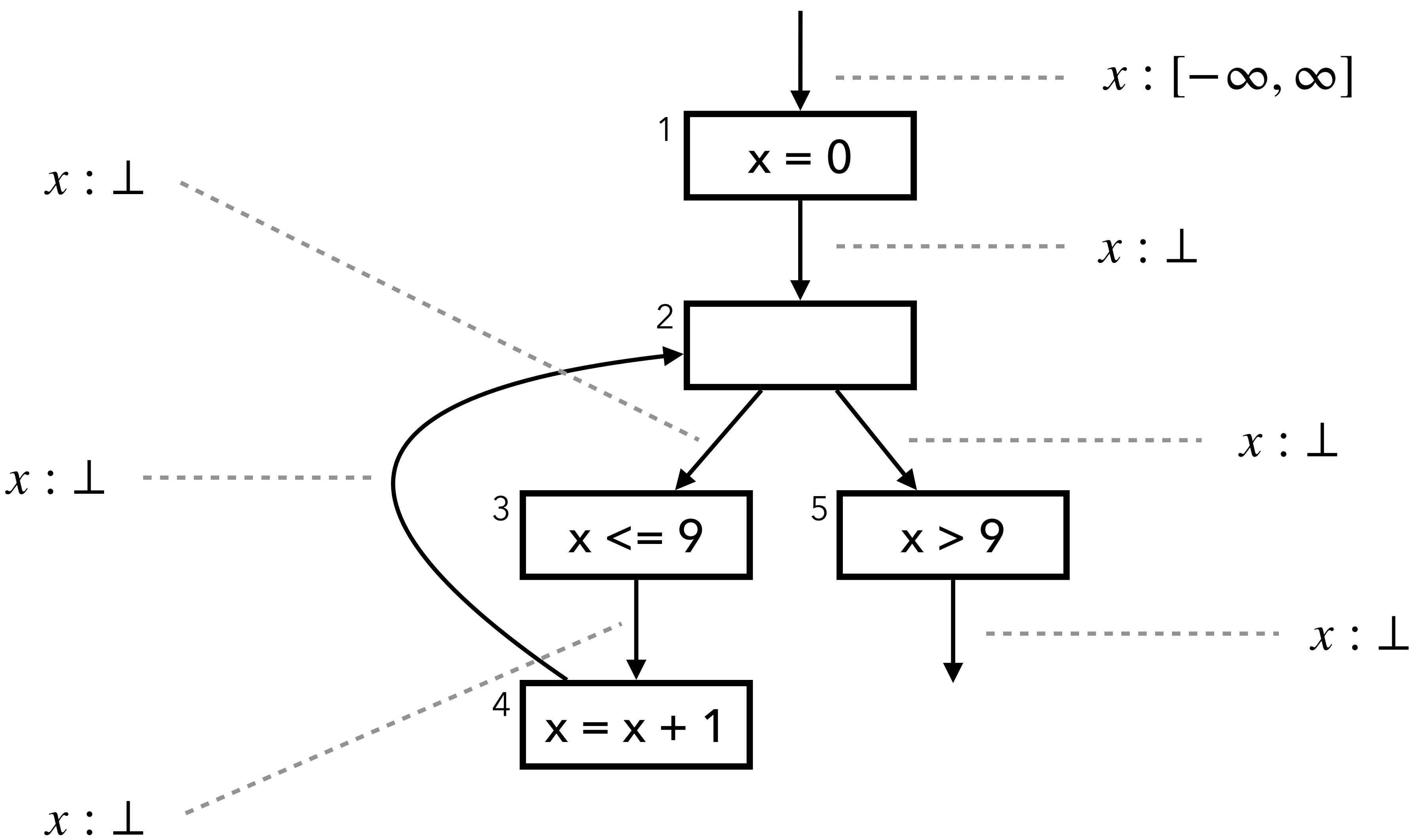


Example Program

```
x = 0;

while (x <= 9)
  x = x + 1;
```

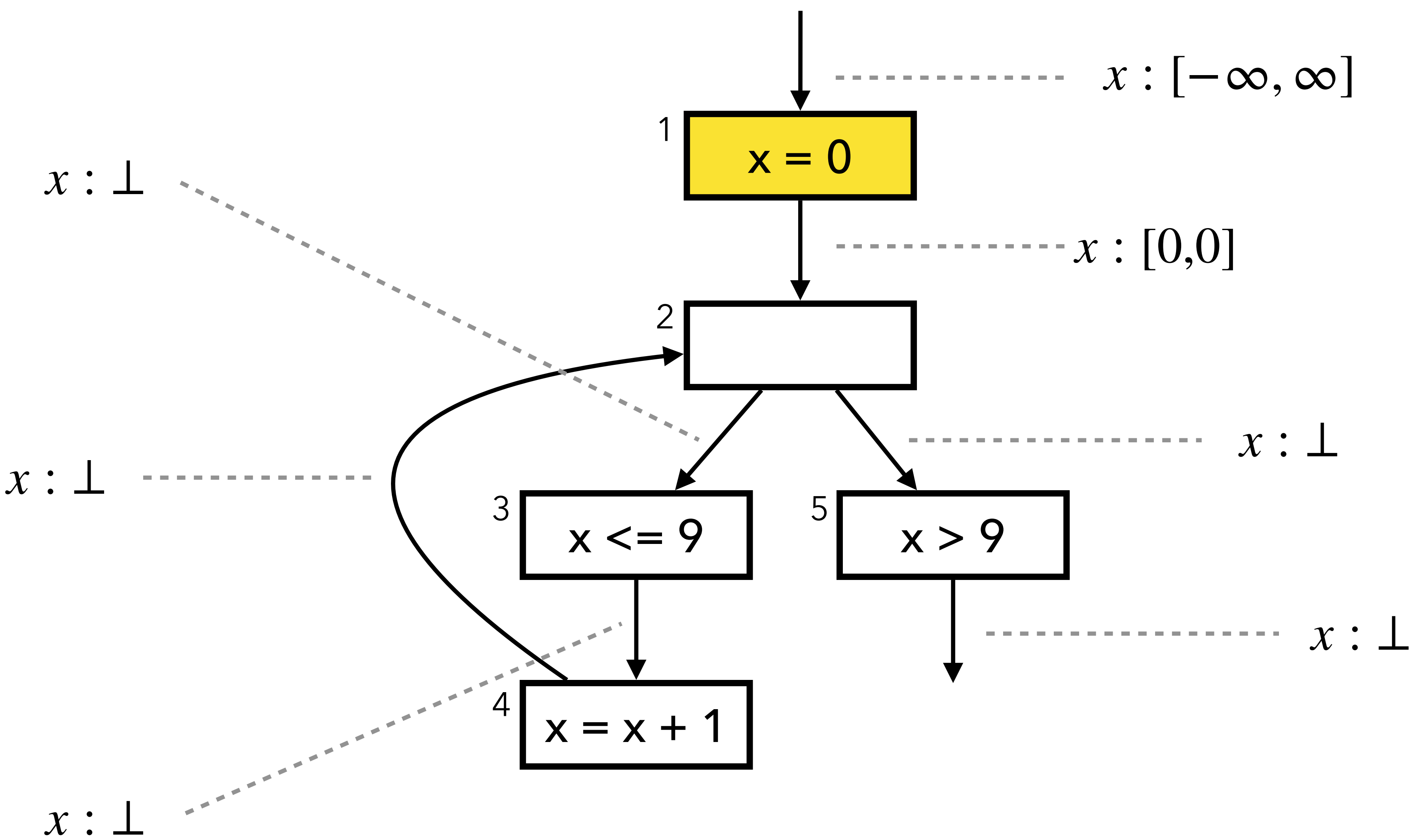


Fixed Point Computation

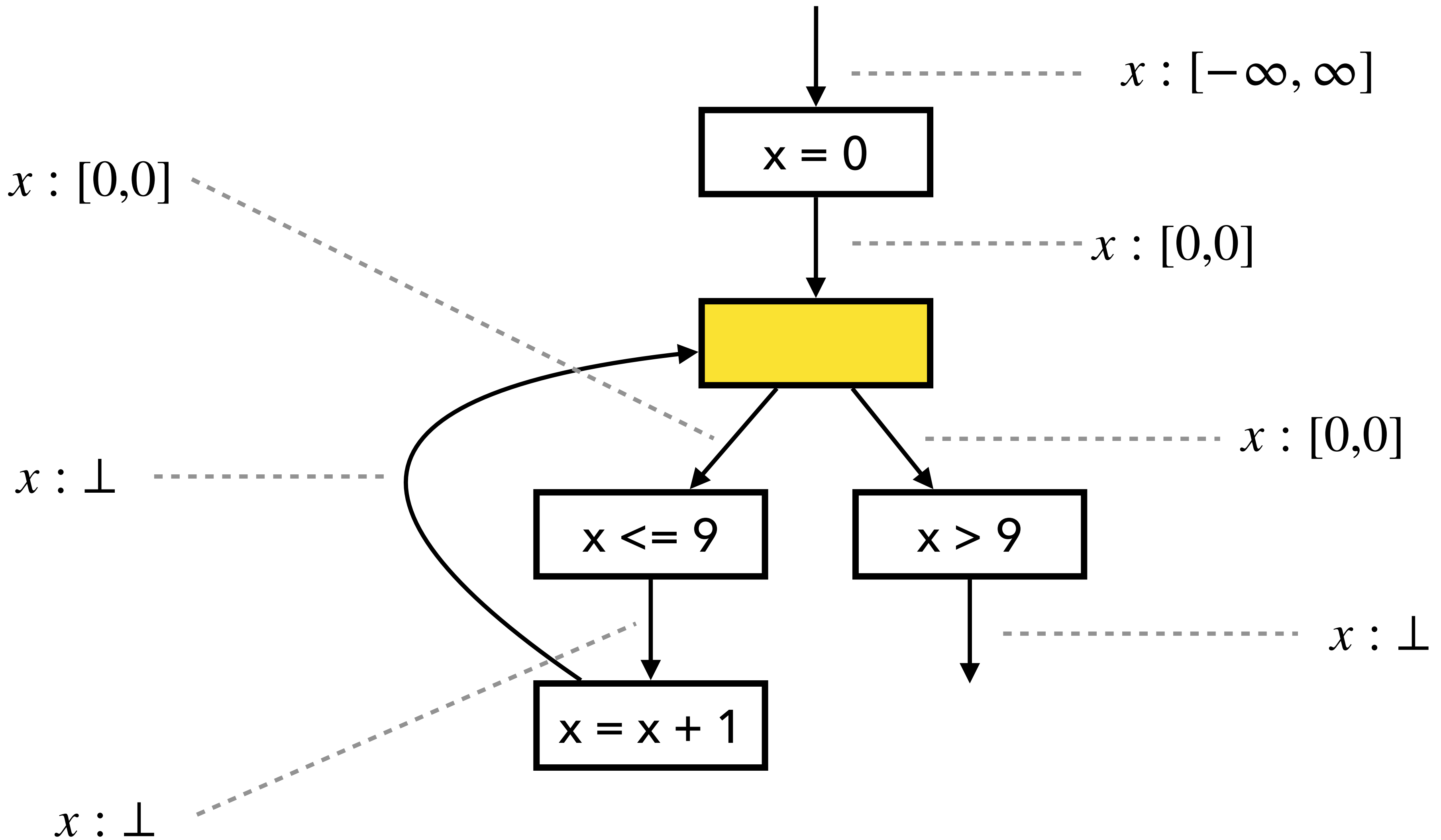


Initial states

Fixed Point Computation

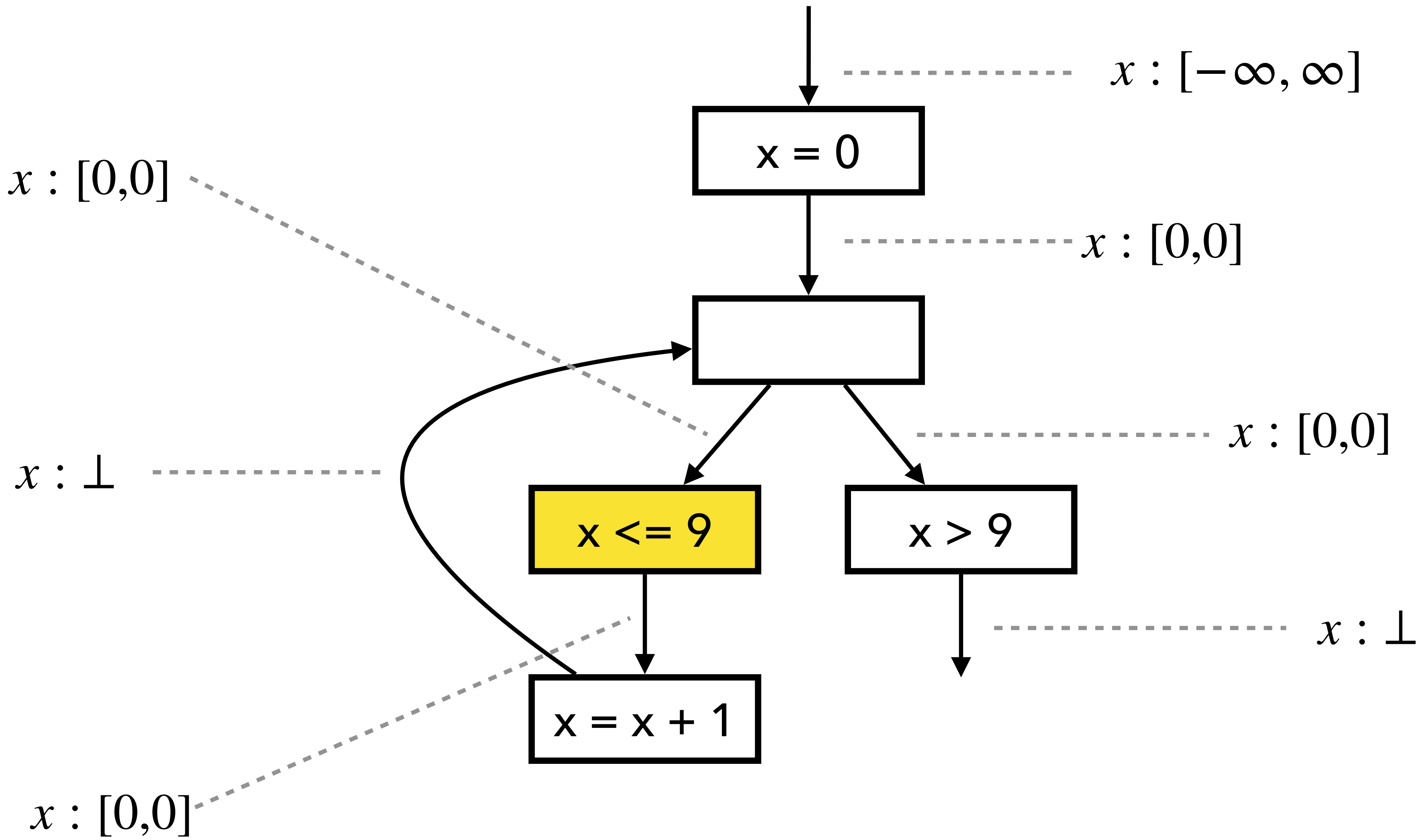


Fixed Point Computation



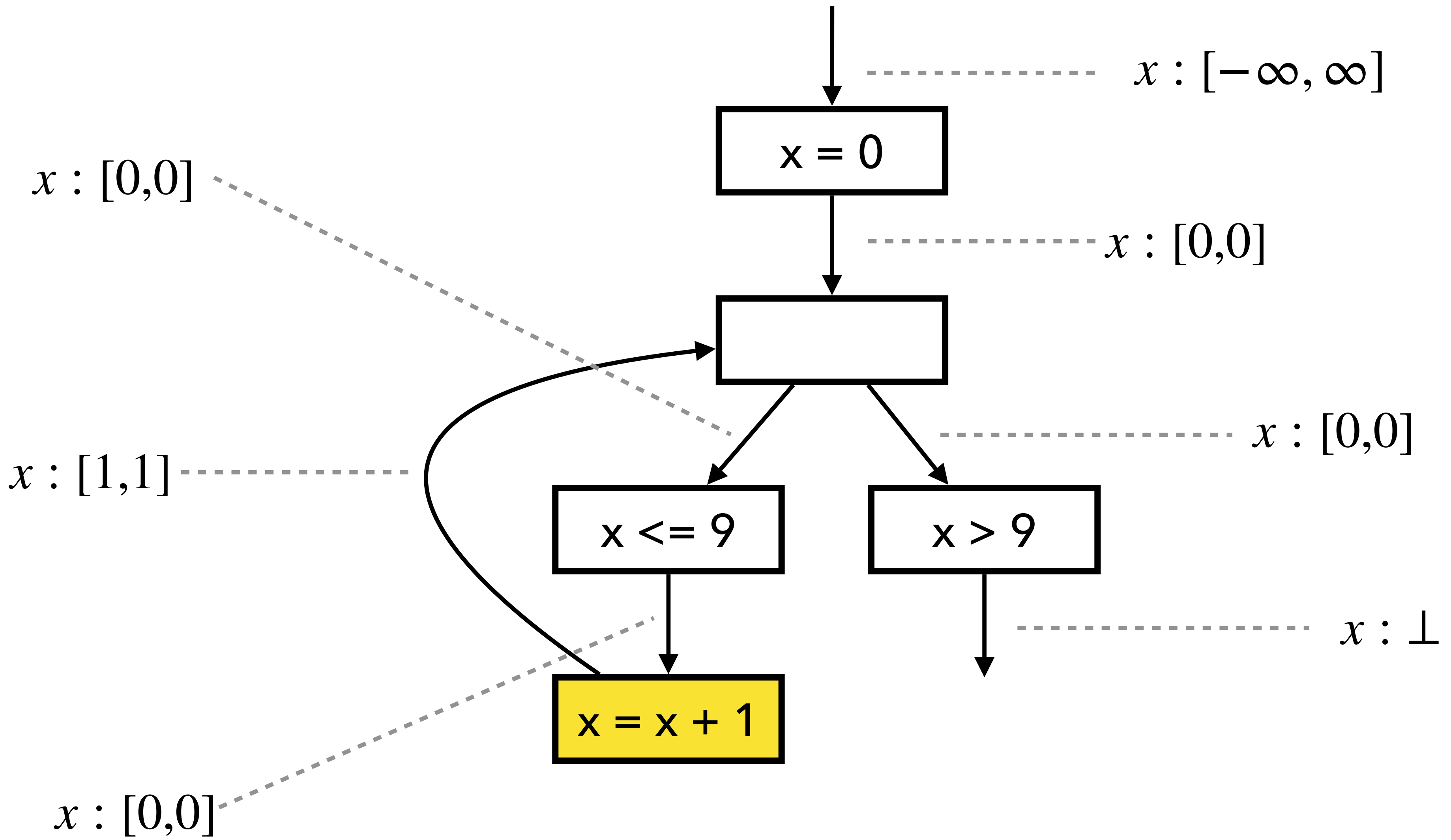
Input state: $[0,0] \sqcup \bot = [0,0]$

Fixed Point Computation

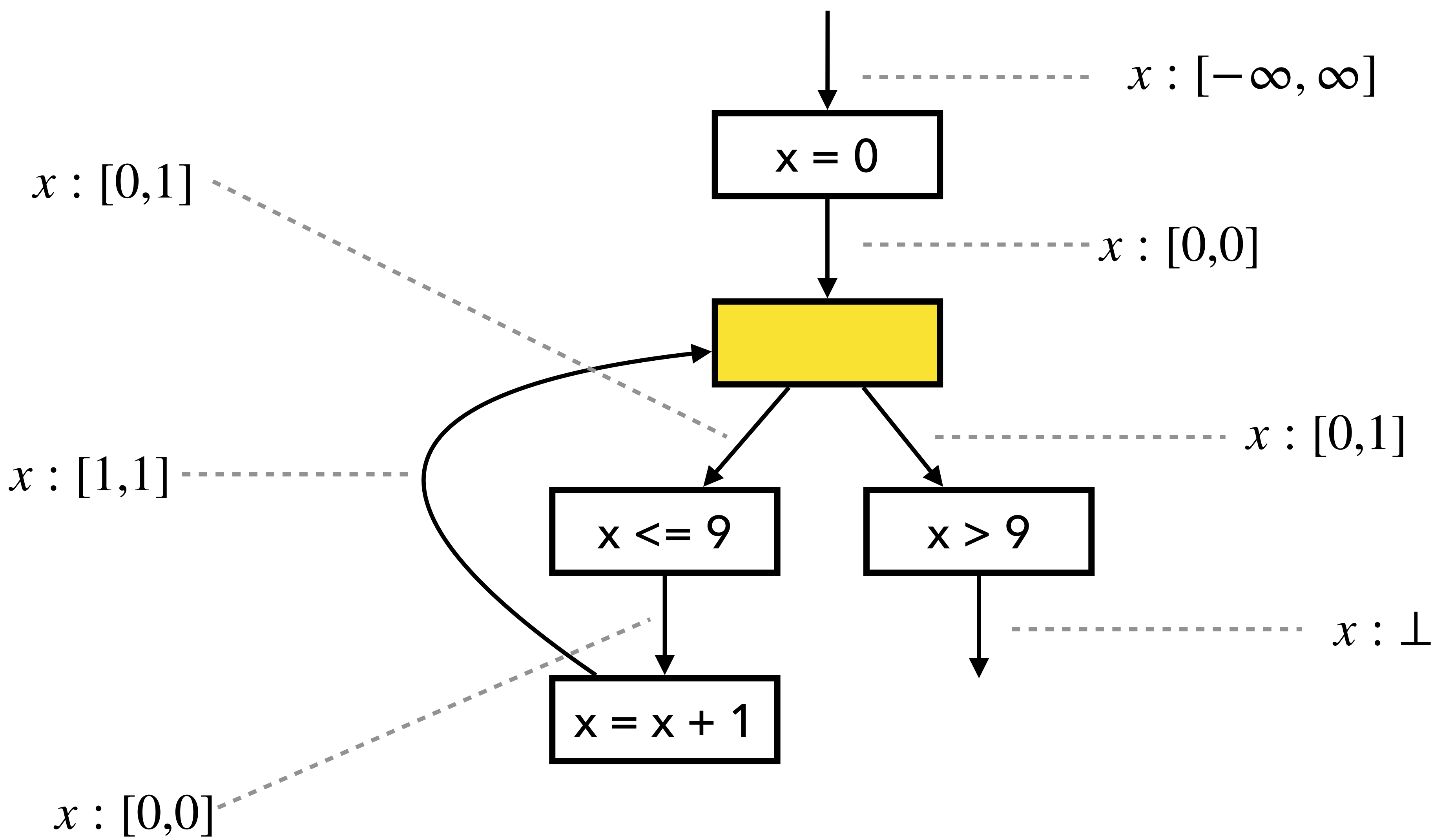


$$[0,0] \sqcap [-\infty,9] = [0,0]$$

Fixed Point Computation



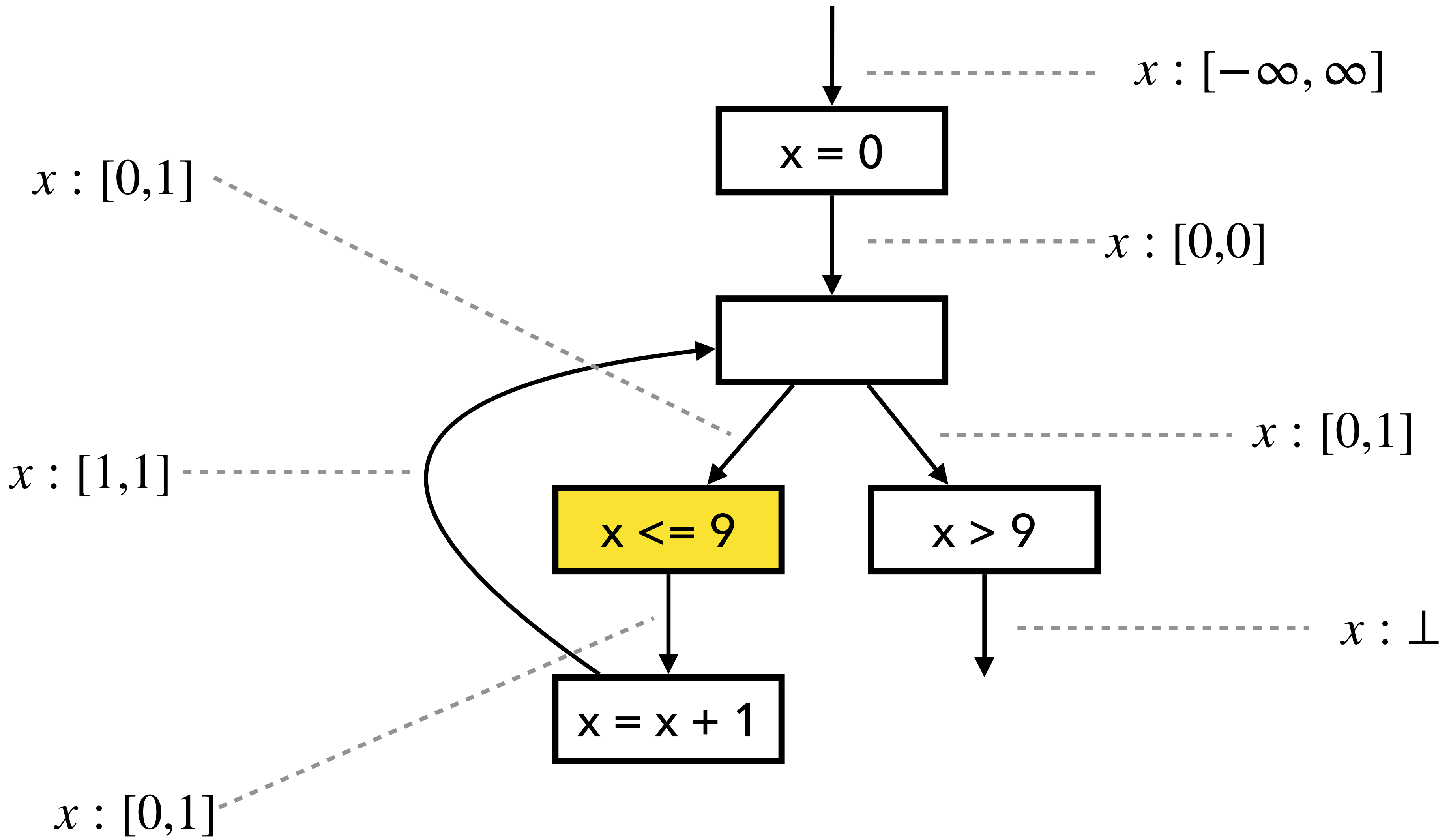
Fixed Point Computation



Input state: $[0,0] \sqcup [1,1] = [0,1]$

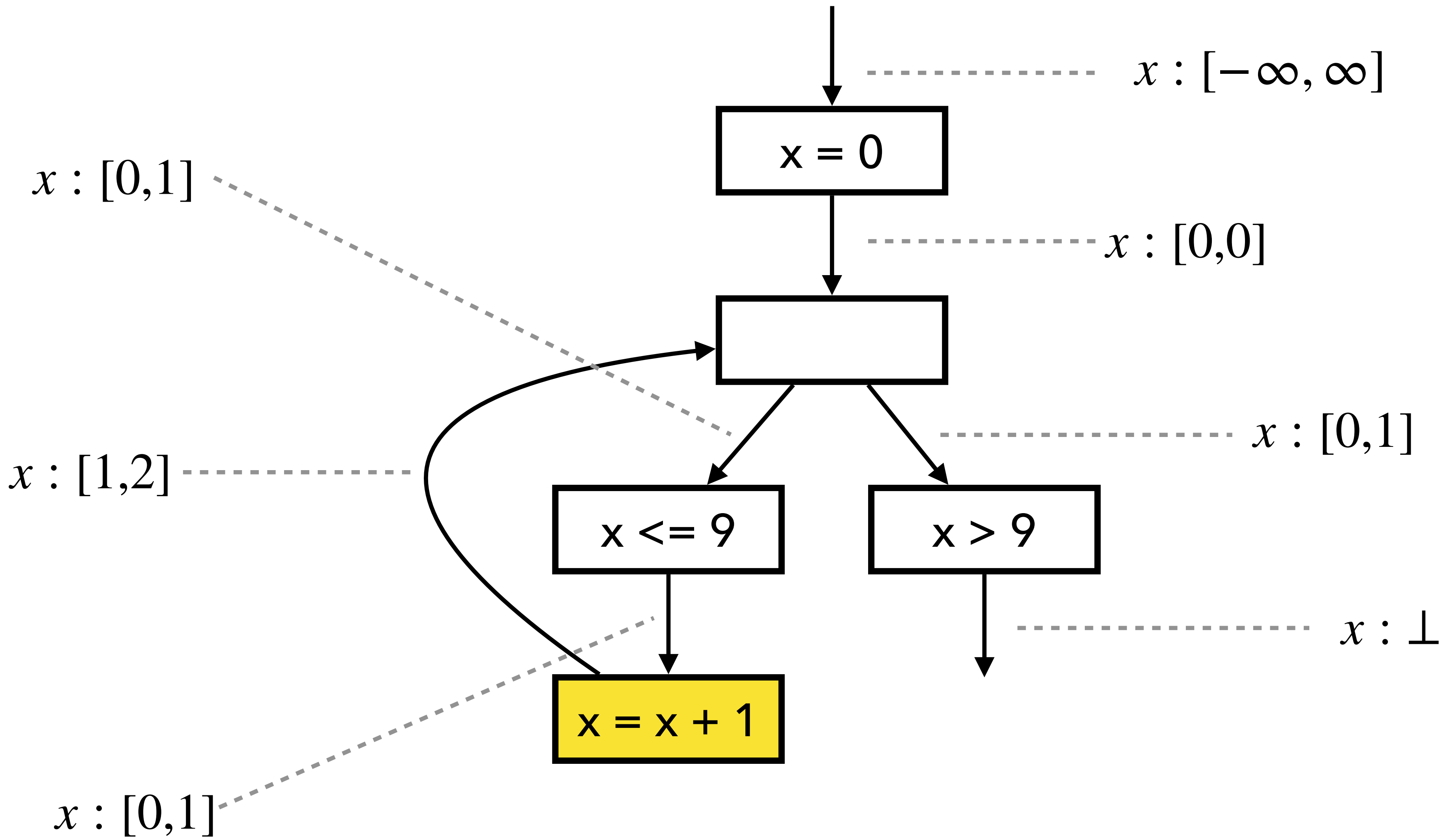
(1st iteration of loop)

Fixed Point Computation

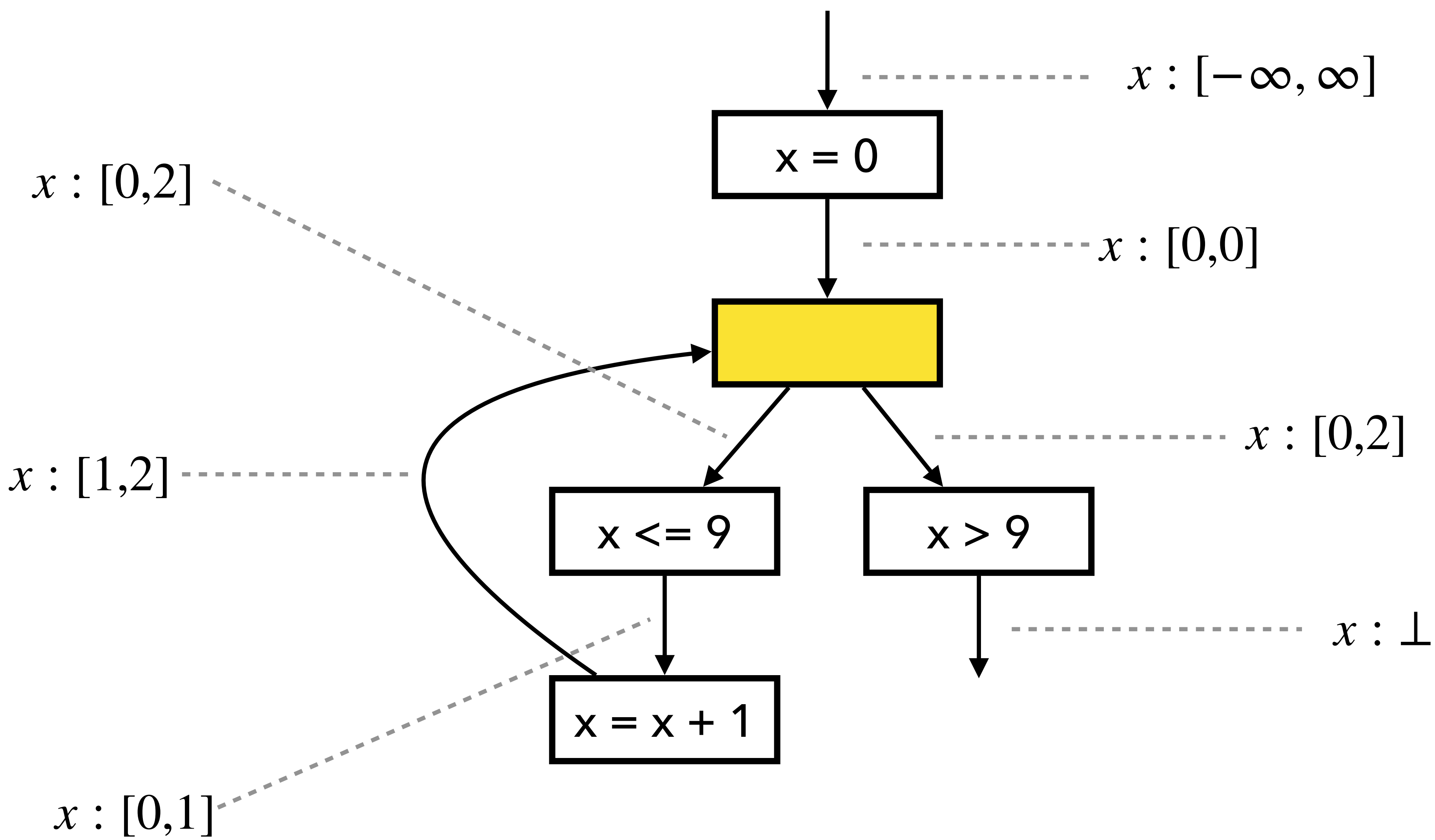


$$[0,1] \sqcap [-\infty,9] = [0,1]$$

Fixed Point Computation



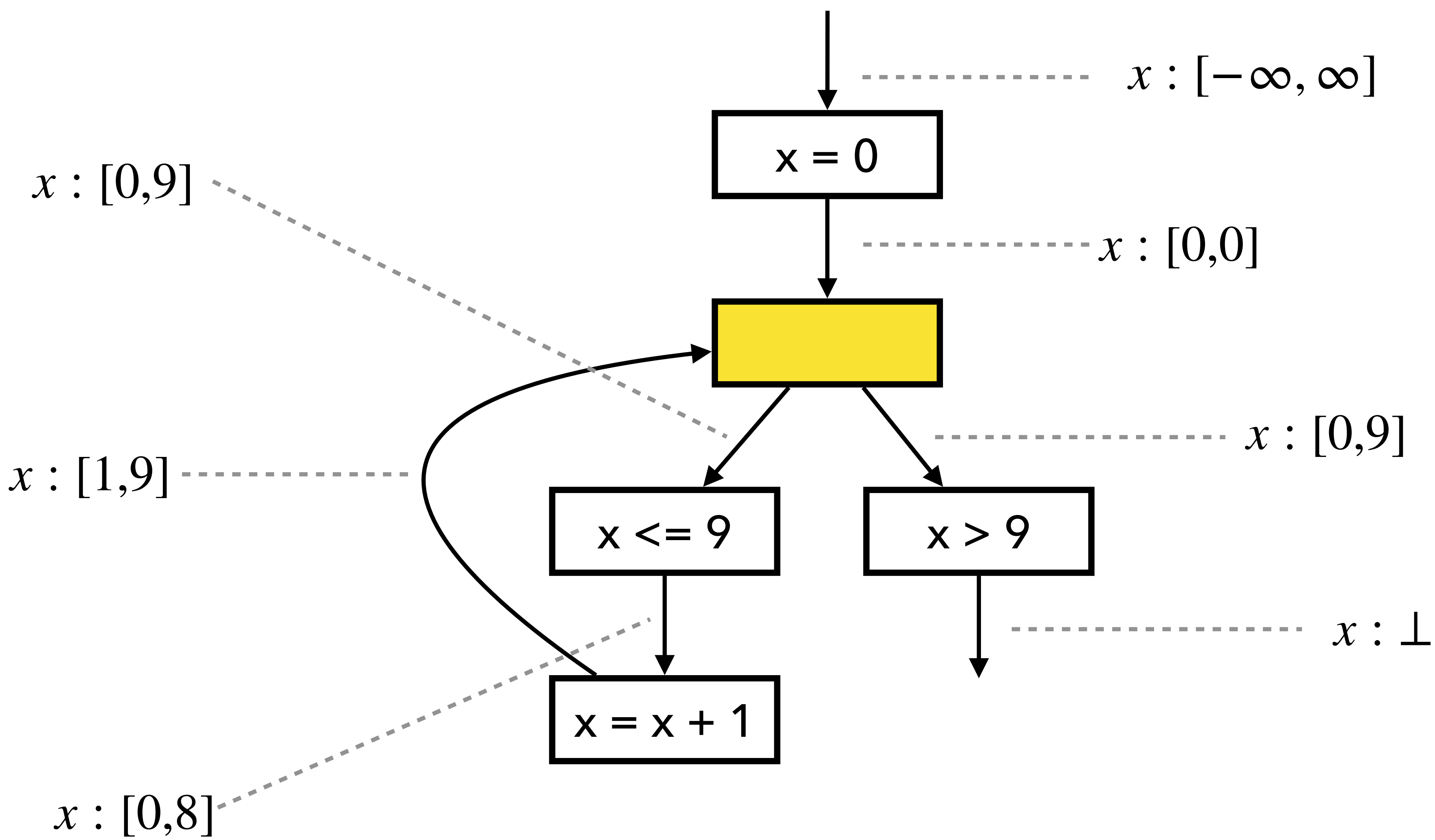
Fixed Point Computation



Input state: $[0,0] \sqcup [1,2] = [0,2]$

(2nd iteration of loop)

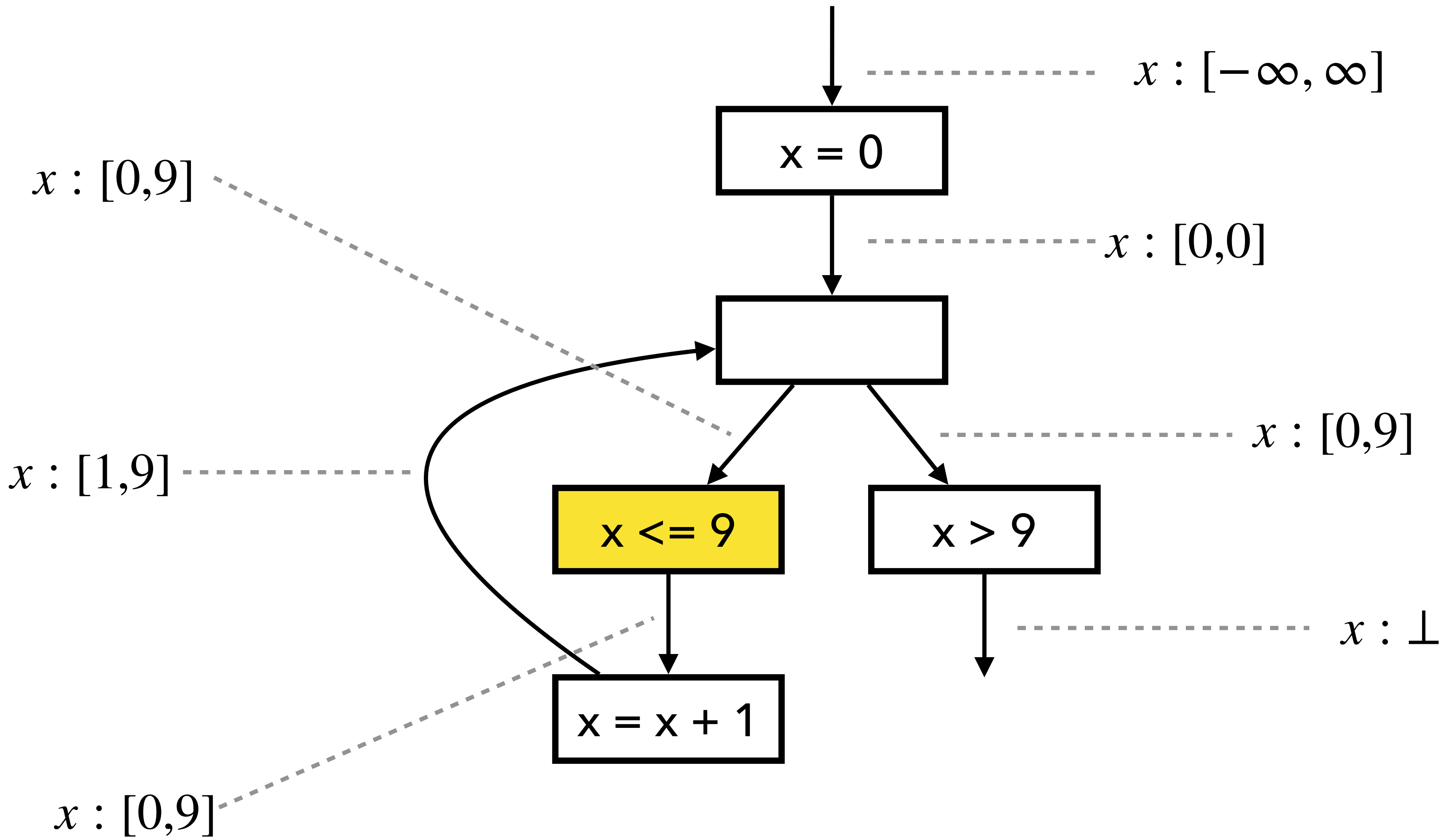
Fixed Point Computation



Input state: $[0,0] \sqcup [1,9] = [0,9]$

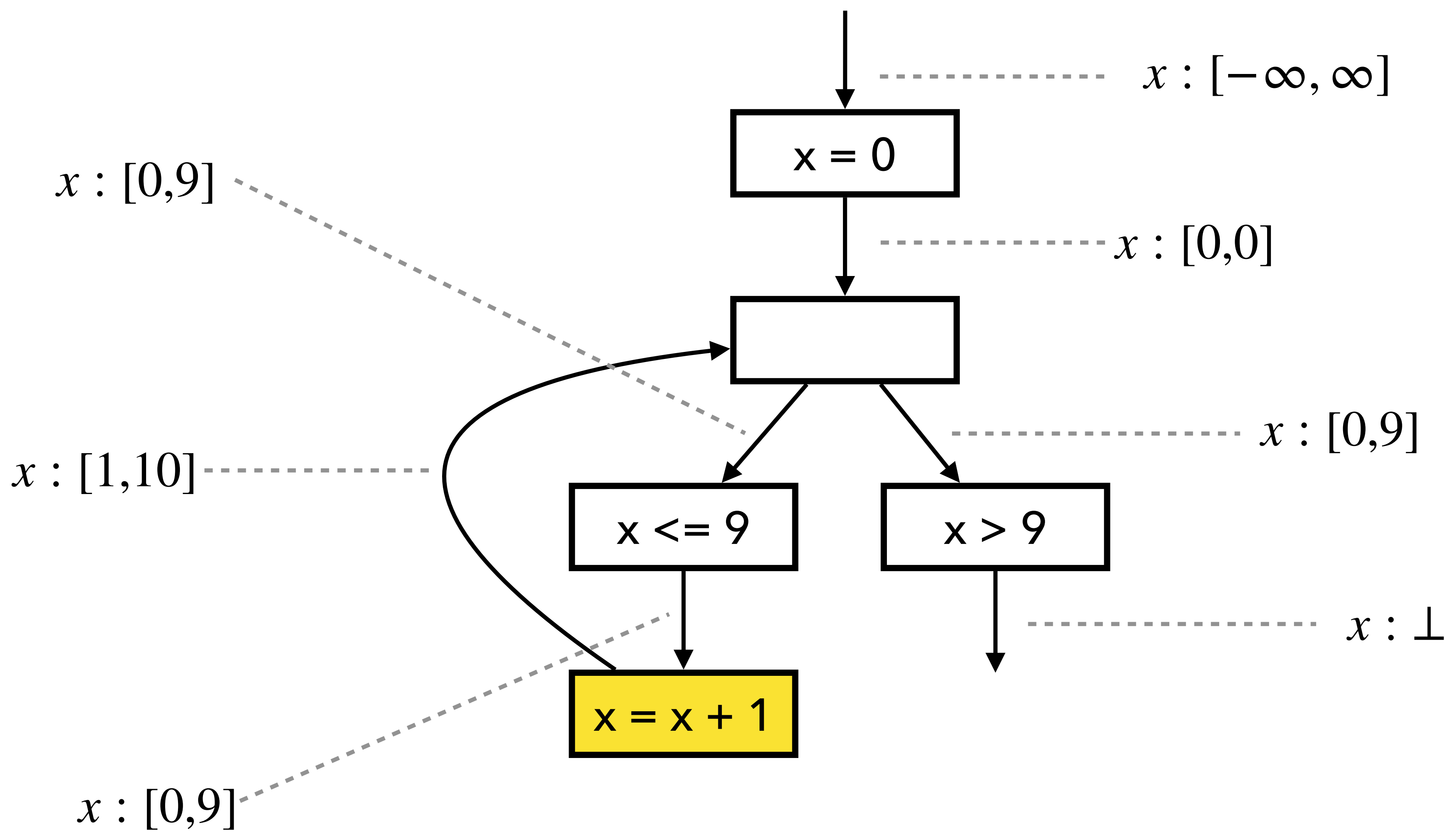
(9th iteration of loop)

Fixed Point Computation

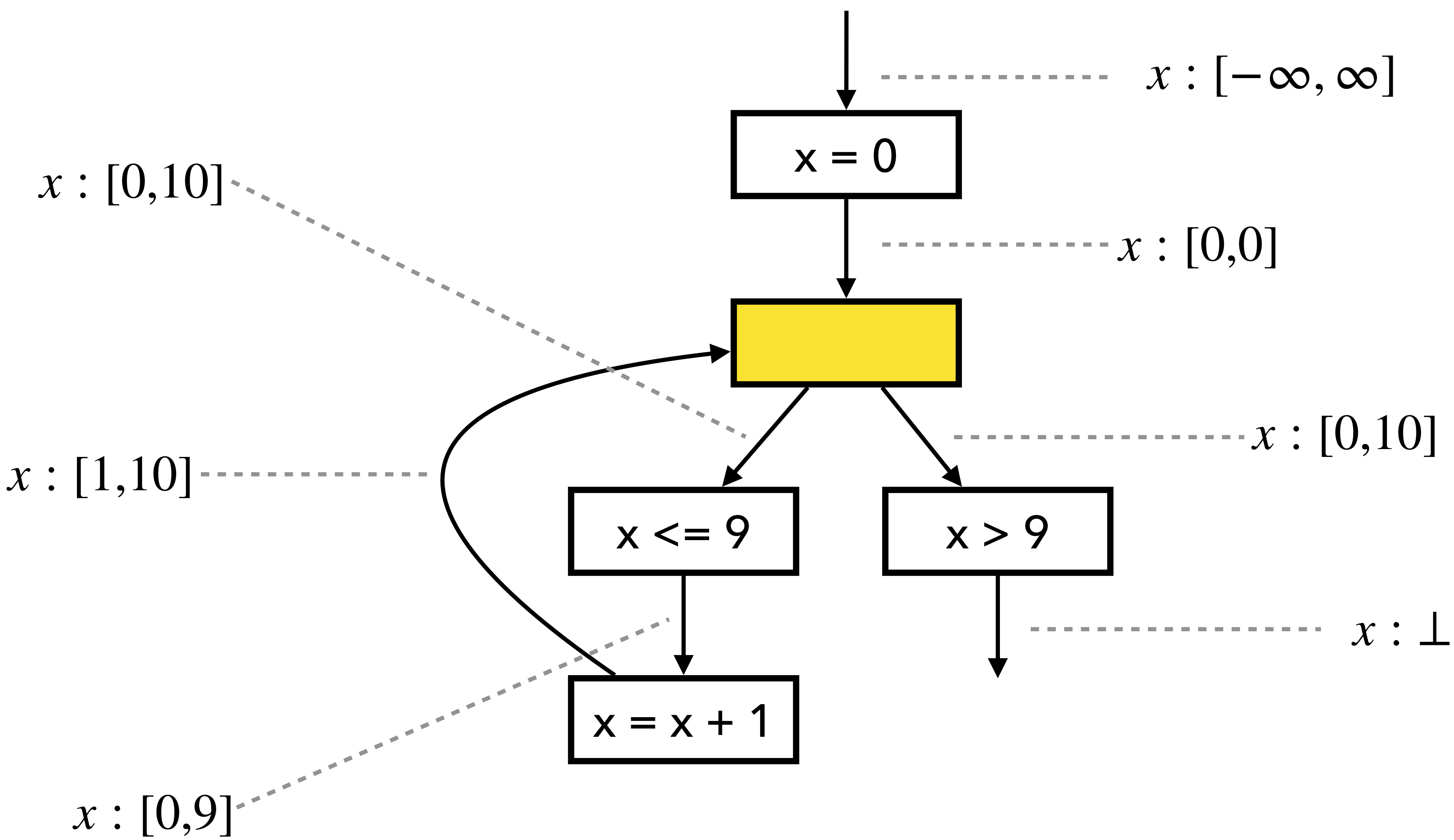


$$[0,9] \sqcap [-\infty,9] = [0,9]$$

Fixed Point Computation

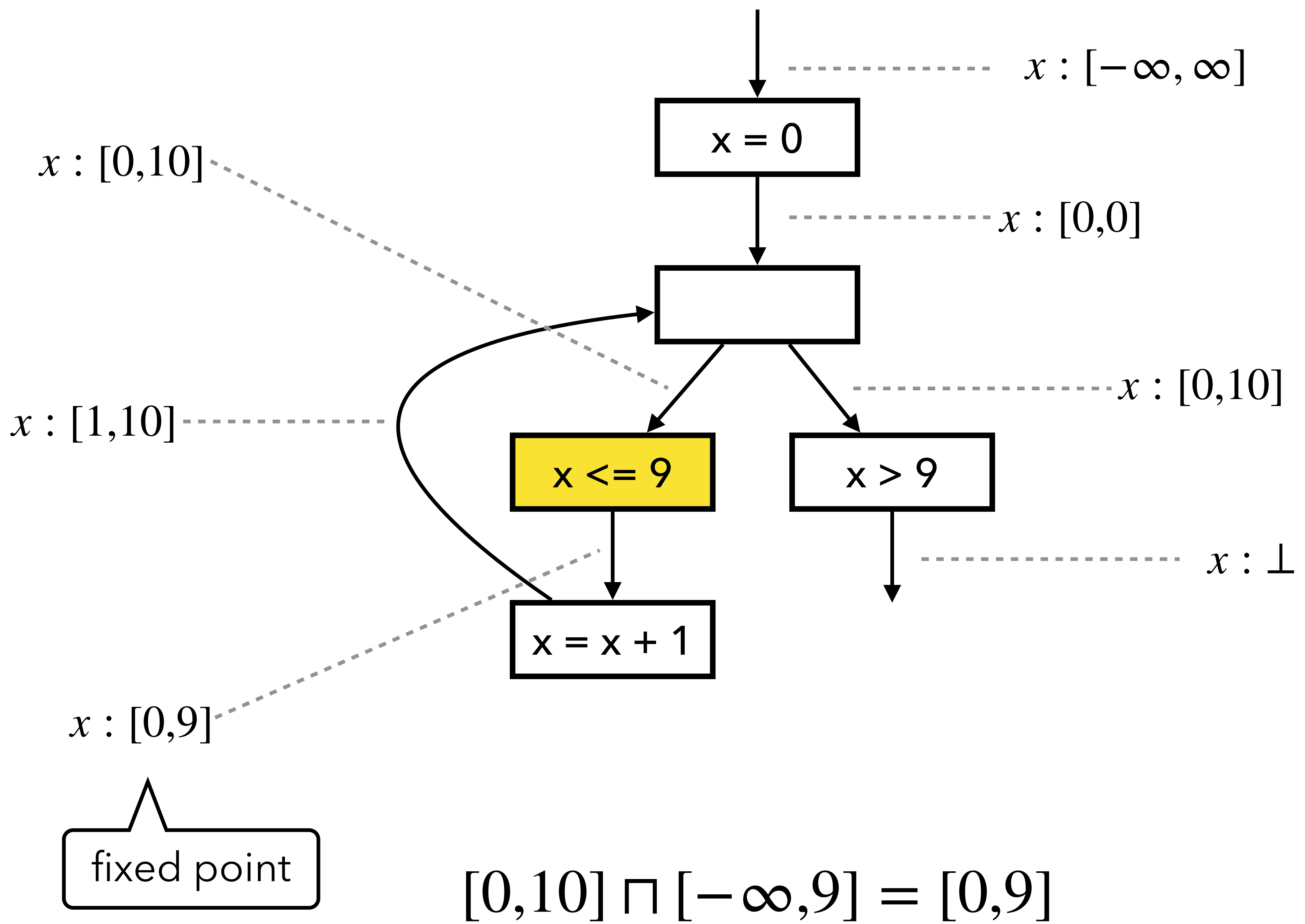


Fixed Point Computation



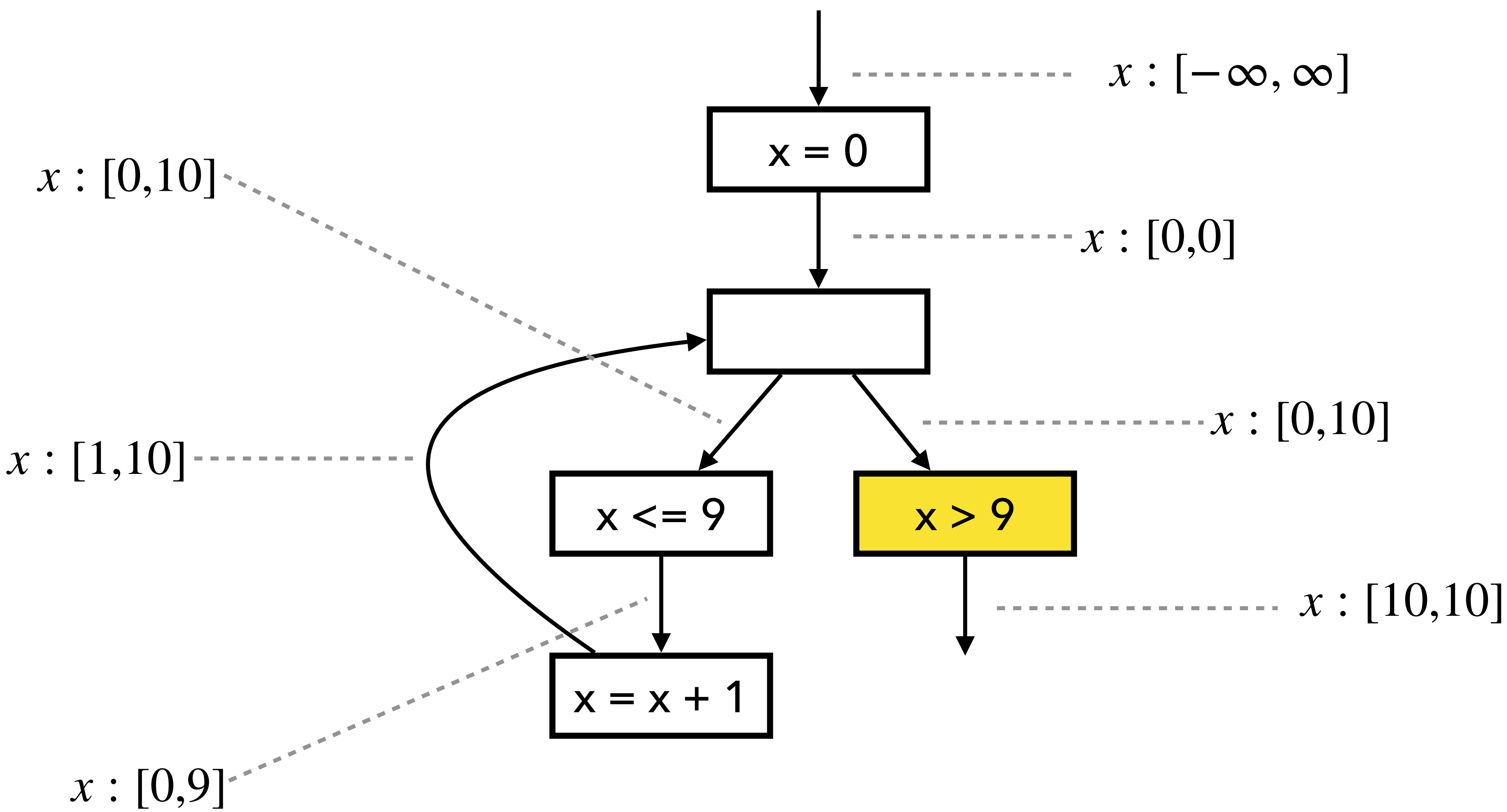
Input state: $[0,0] \sqcup [1,10] = [0,10]$
(10th iteration of loop)

Fixed Point Computation



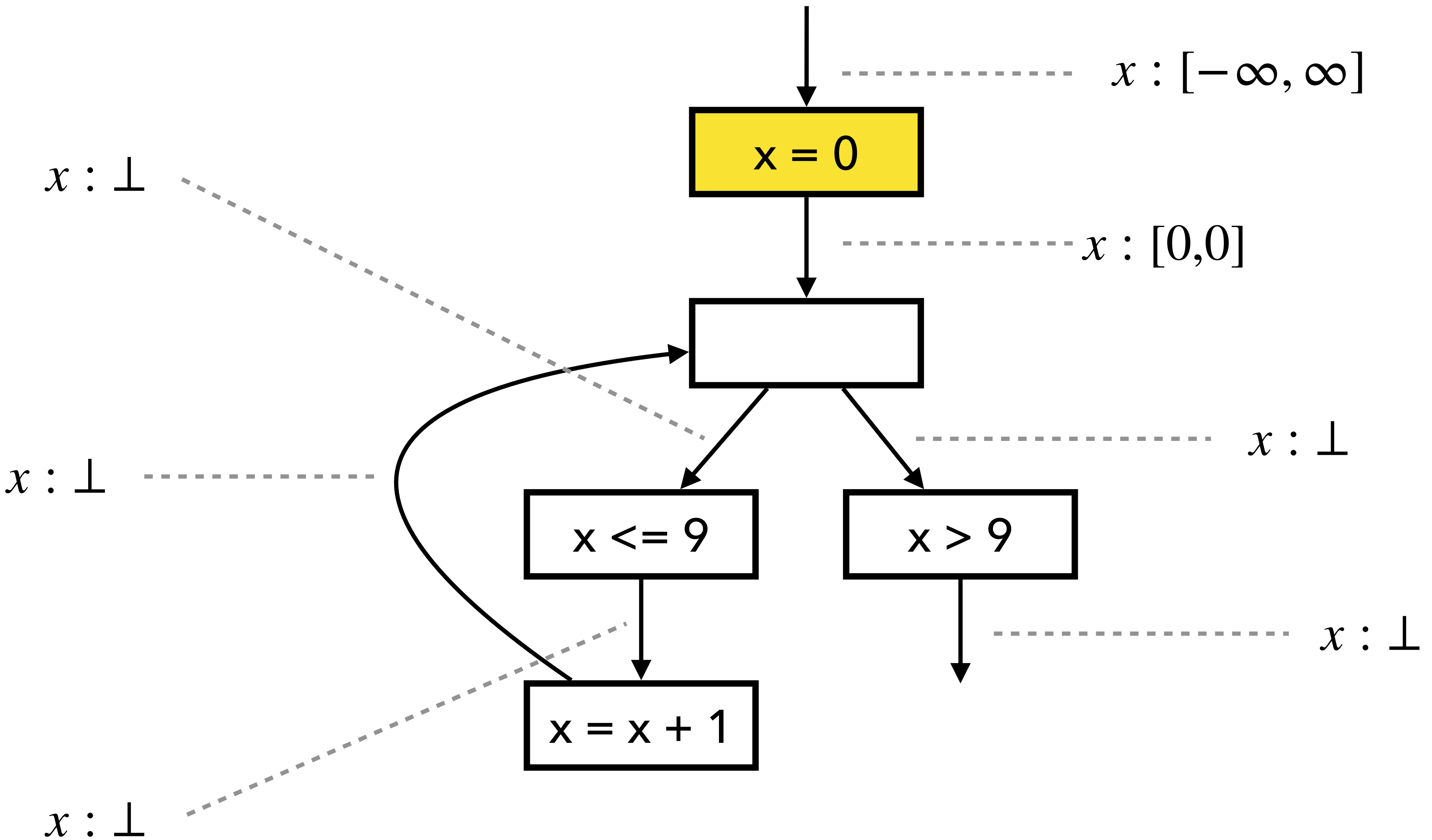
$$[0,10] \sqcap [-\infty,9] = [0,9]$$

Fixed Point Computation

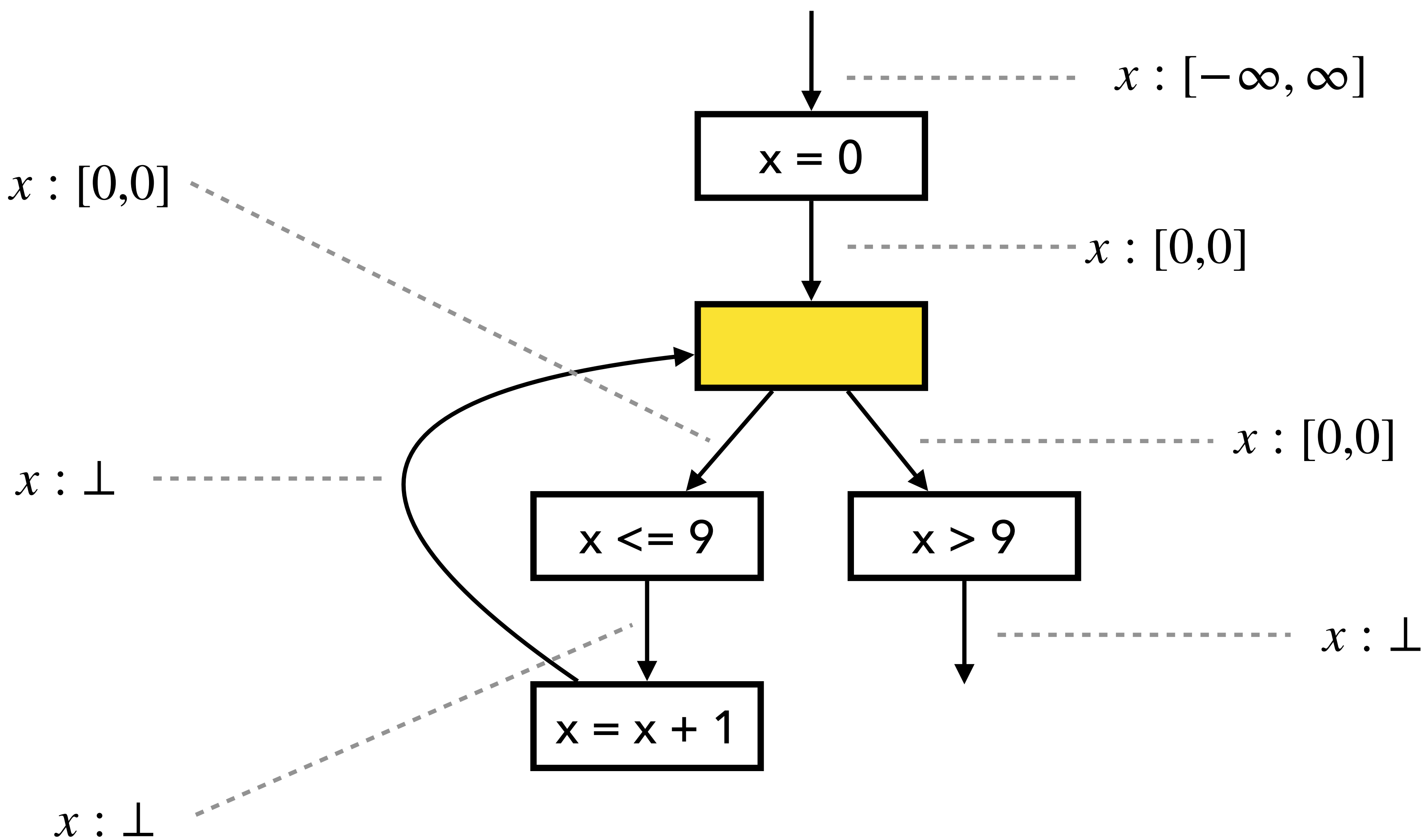


$$[0,10] \sqcap [10,\infty] = [10,10]$$

Fixed Point Comp. with Widening

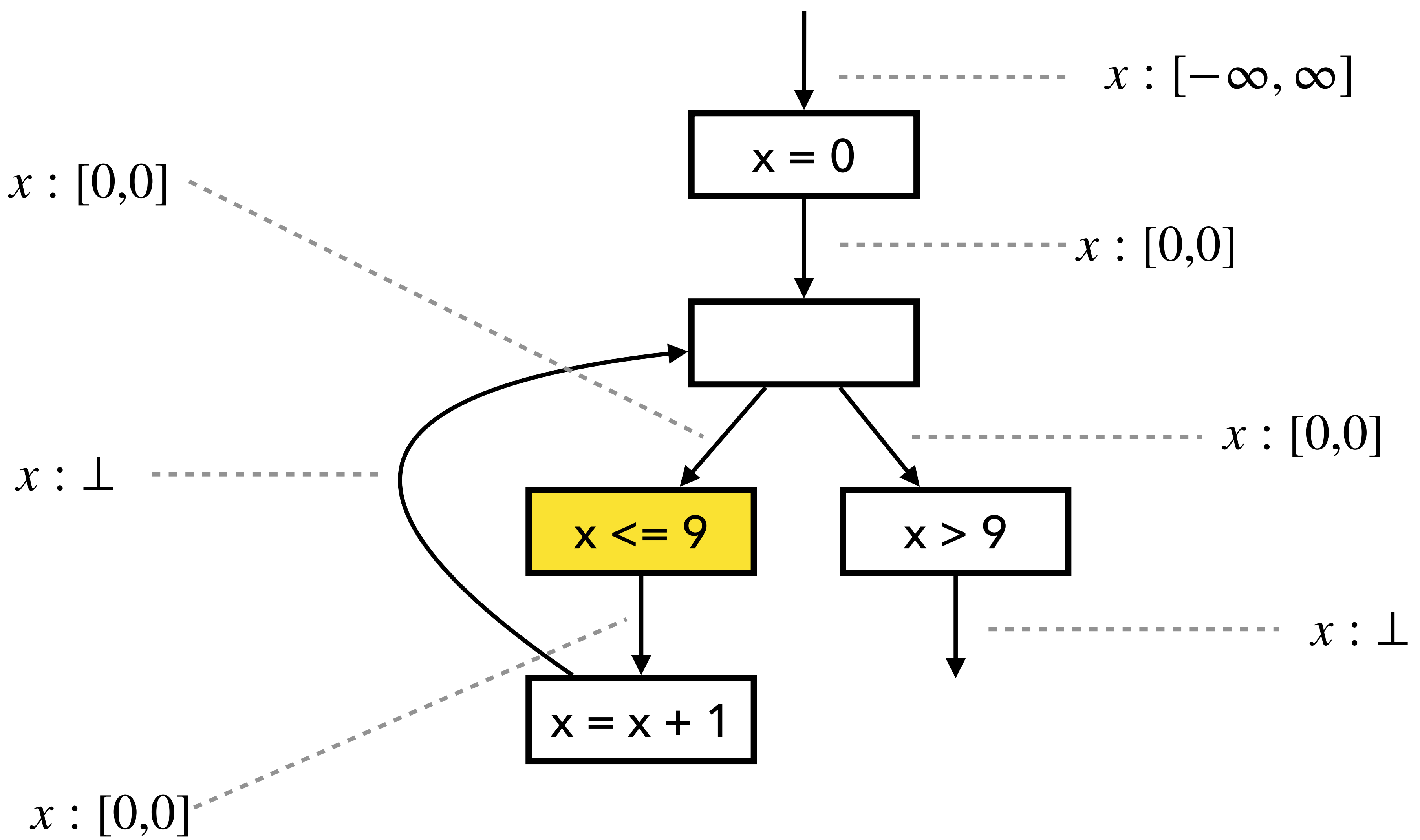


Fixed Point Comp. with Widening



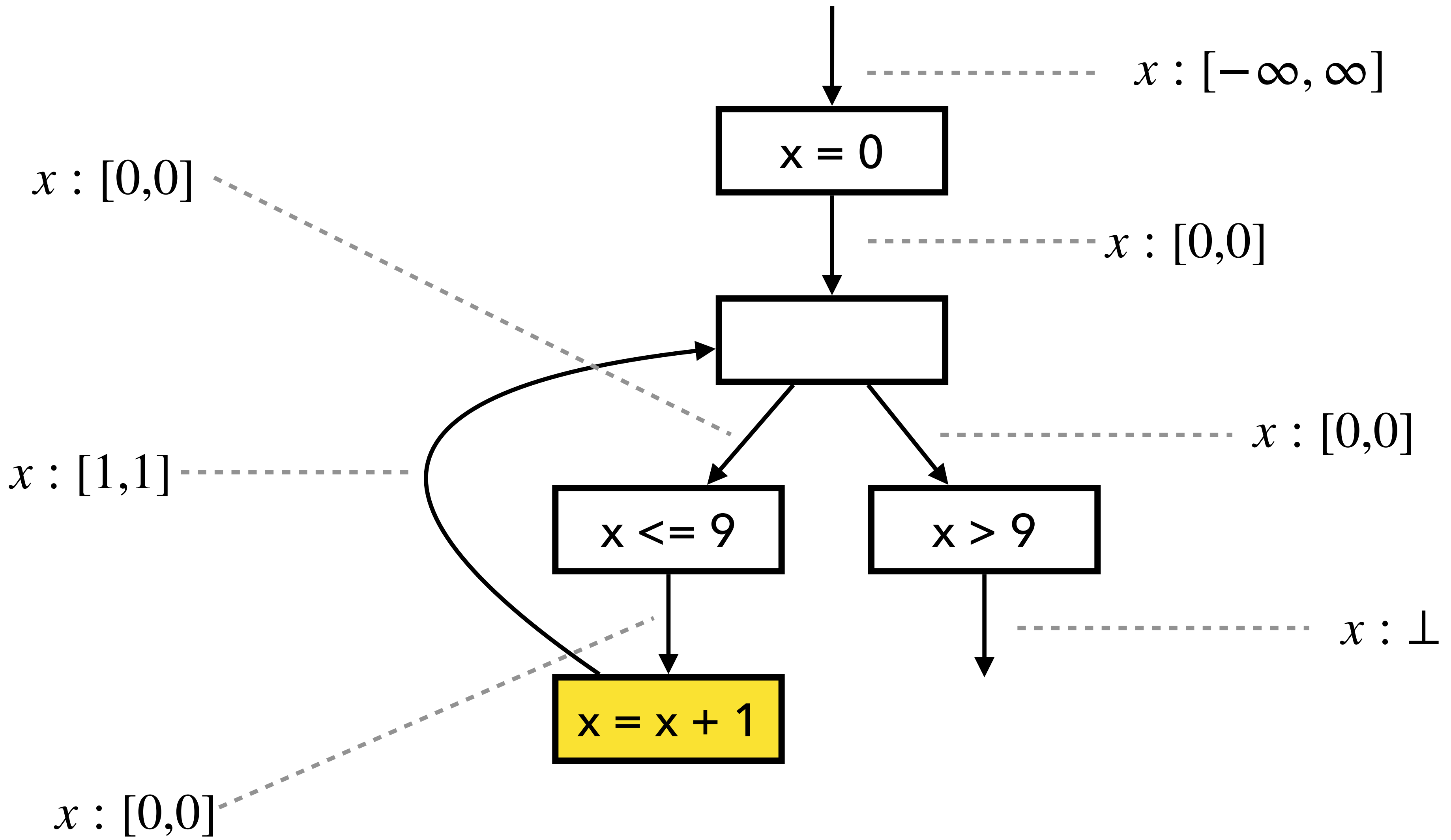
Input state: $[0,0] \sqcup \bot = [0,0]$

Fixed Point Comp. with Widening



$$[0,0] \sqcap [-\infty,9] = [0,0]$$

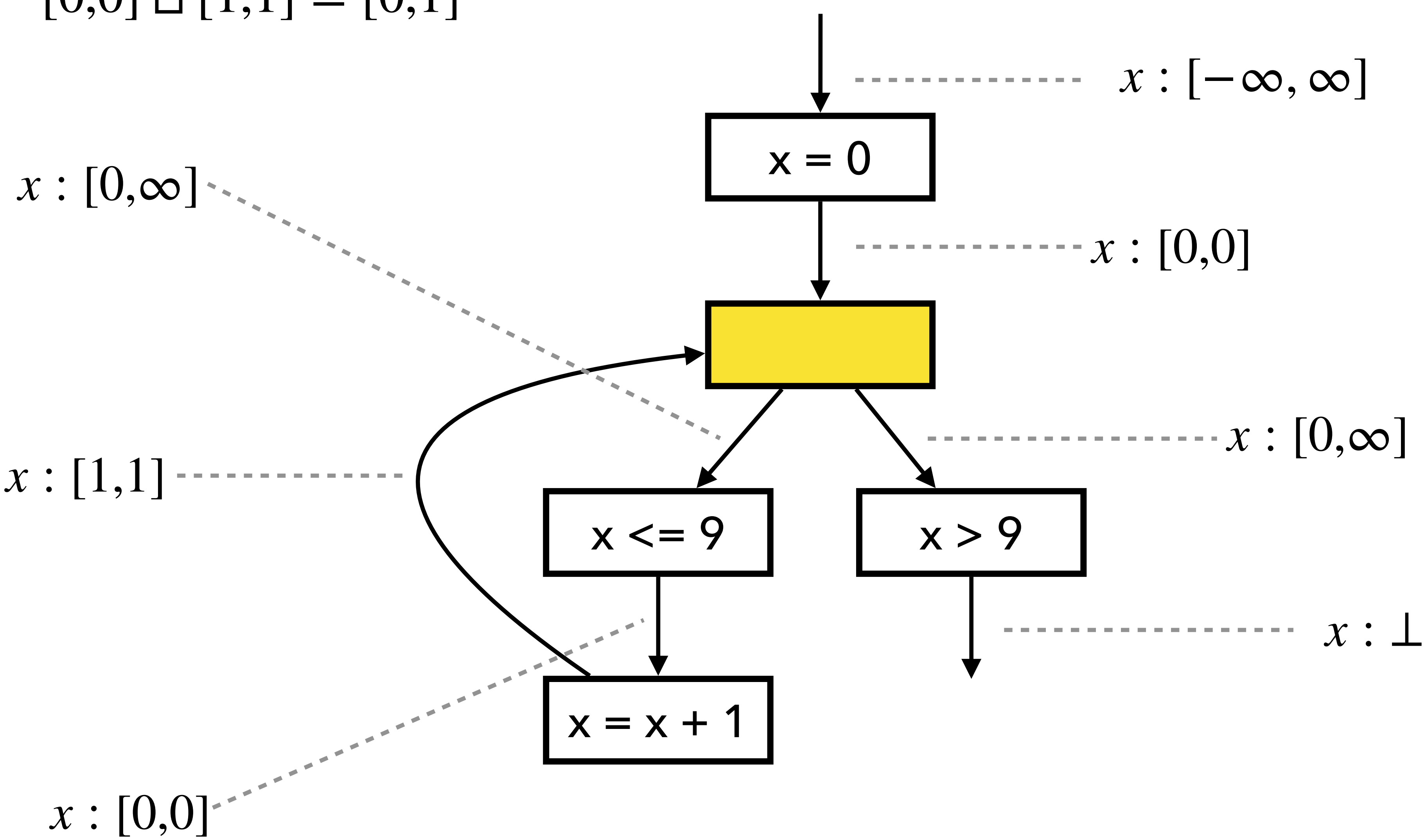
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

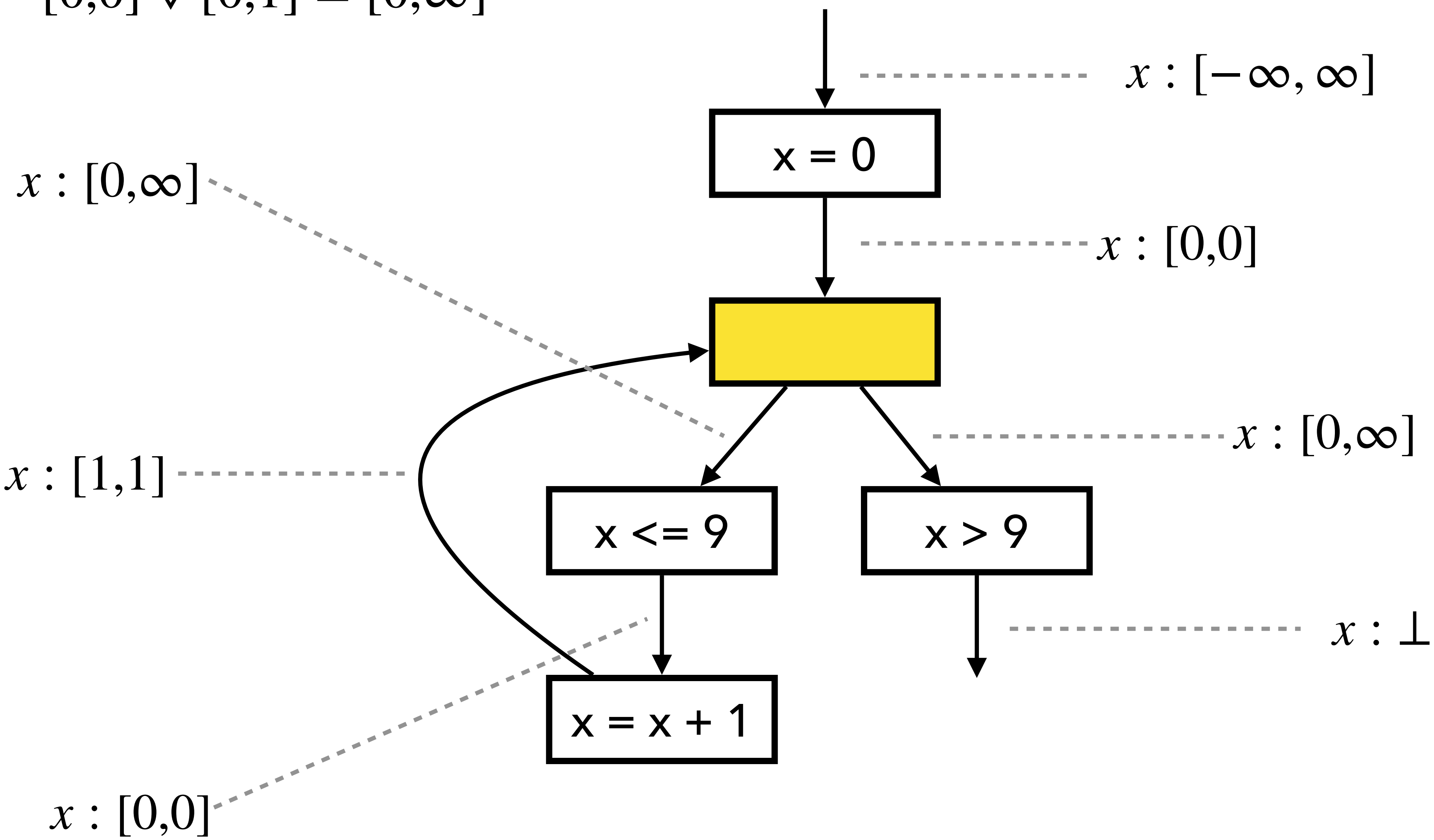
$$[0,0] \sqcup [1,1] = [0,1]$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

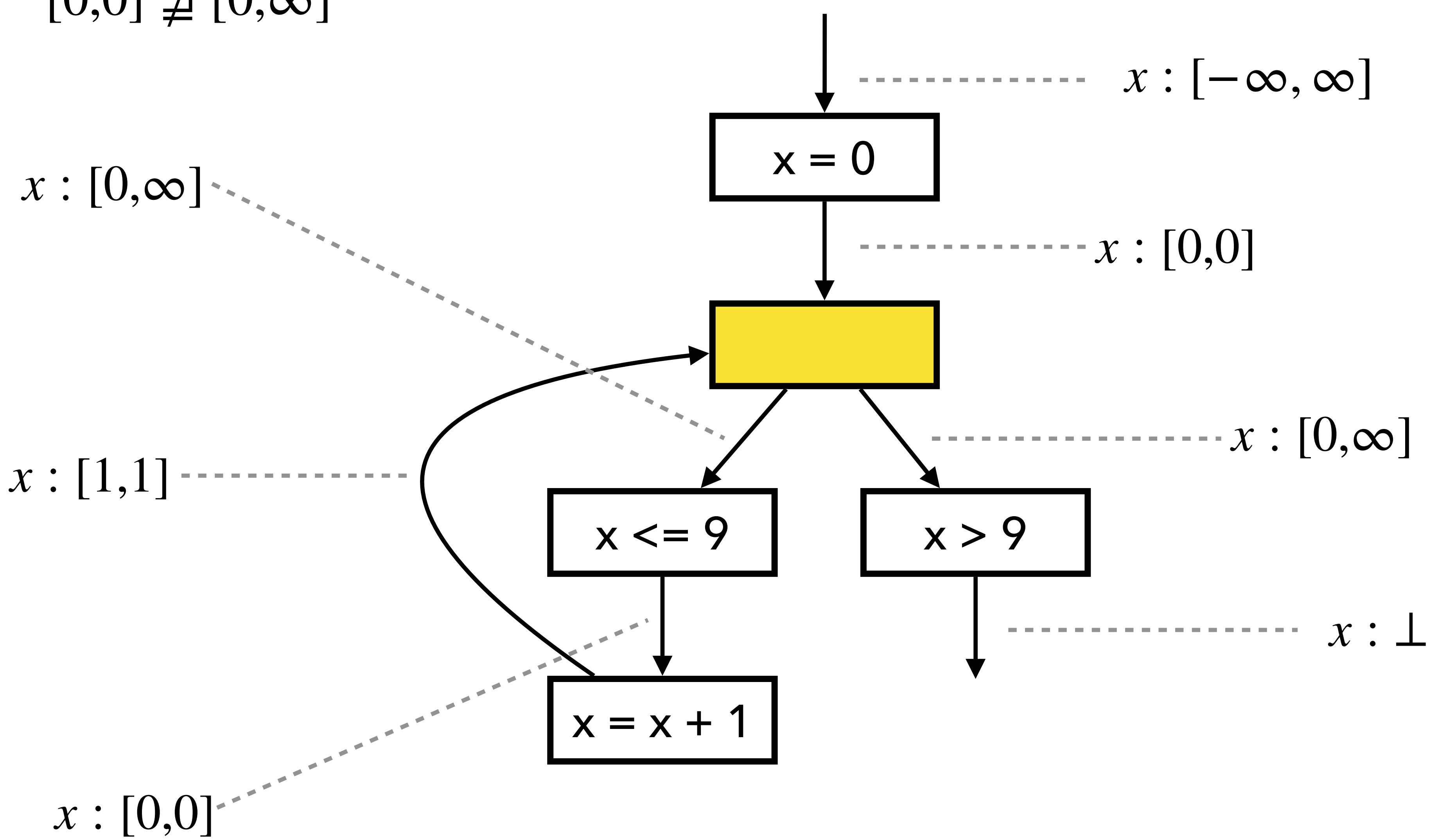
$$[0,0] \triangledown [0,1] = [0,\infty]$$



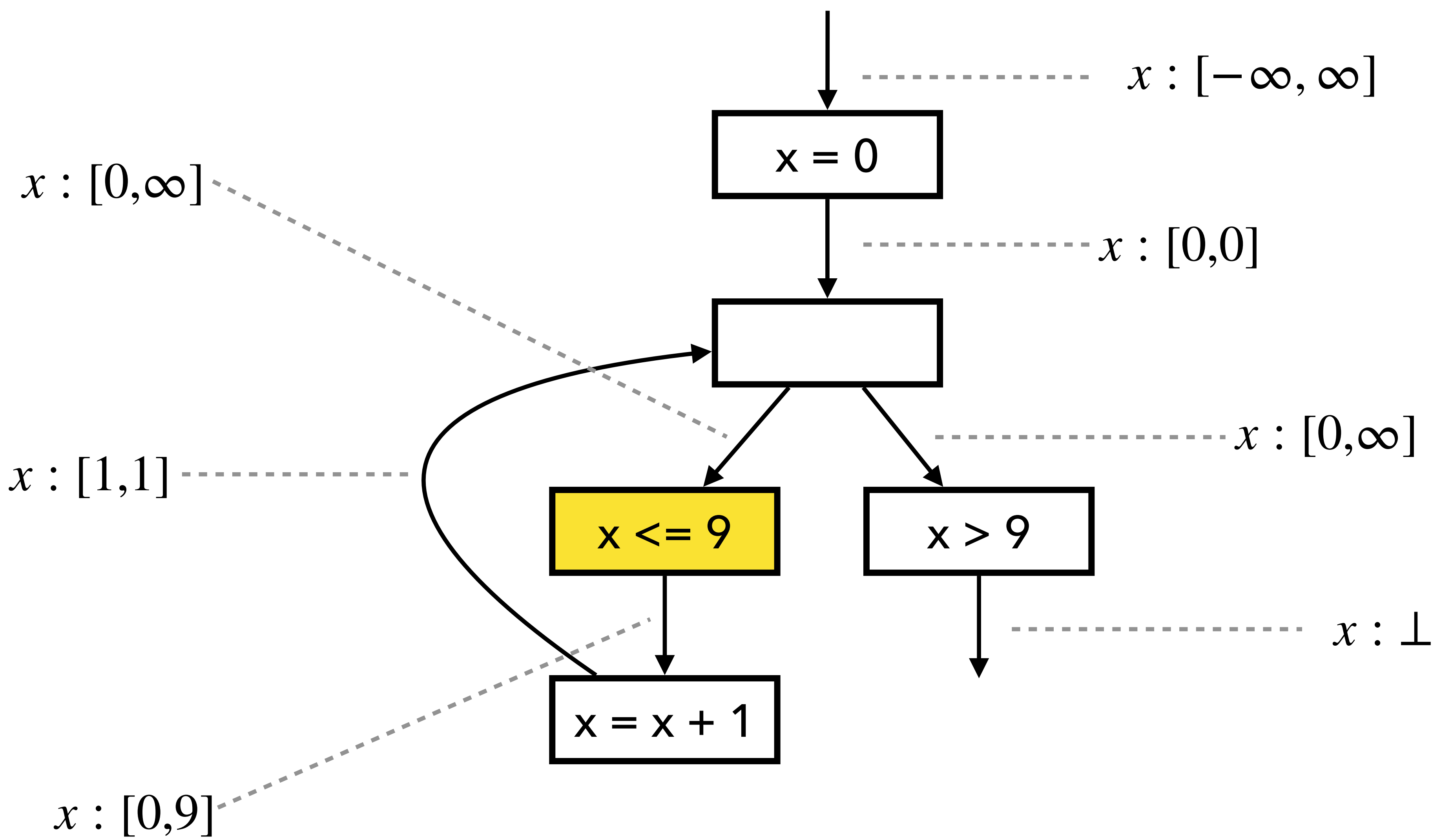
Fixed Point Comp. with Widening

3. Check if fixed point is reached

$[0,0] \not\sqsupseteq [0,\infty]$

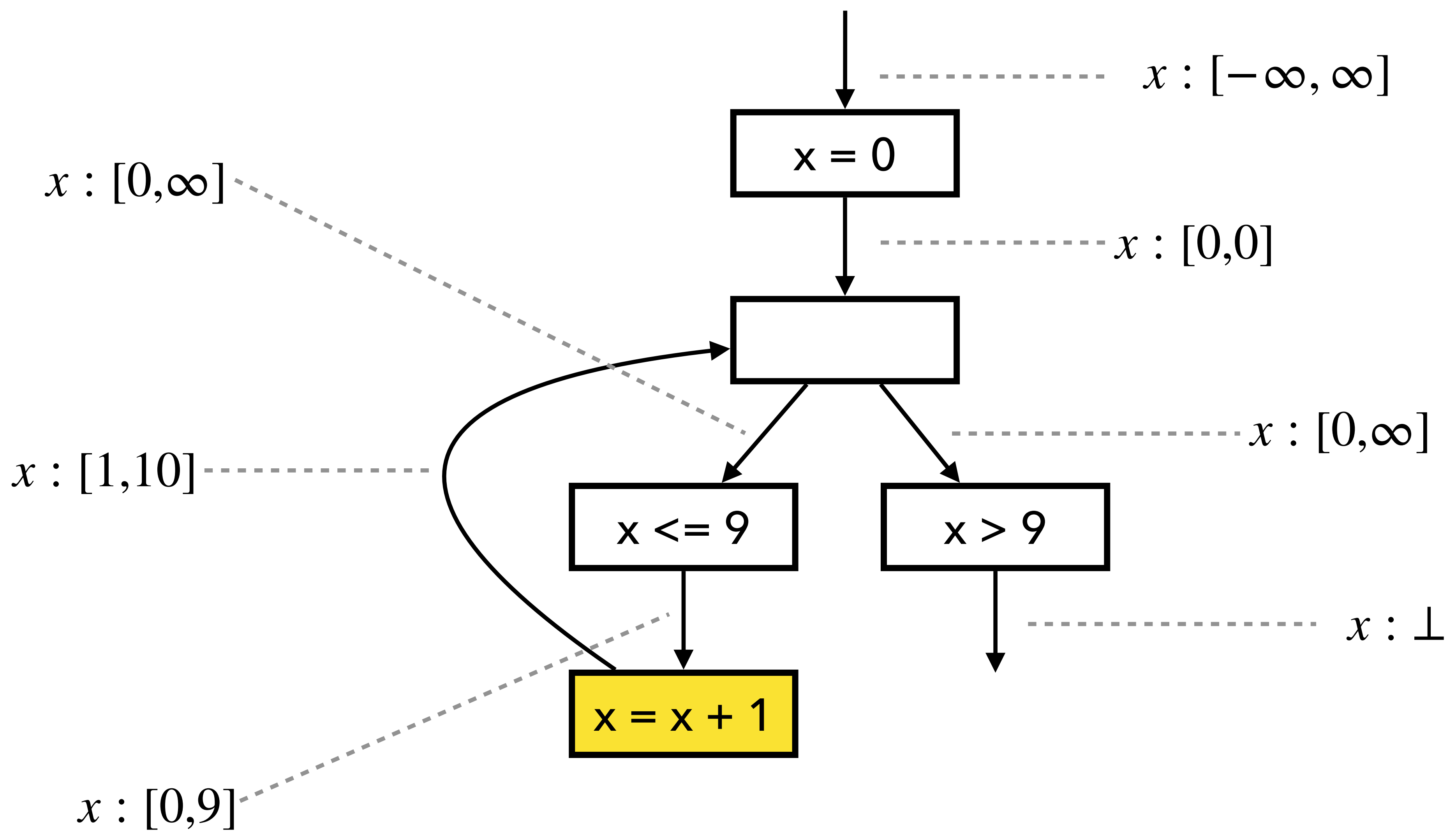


Fixed Point Comp. with Widening



$$[0,\infty] \sqcap [-\infty,9] = [0,9]$$

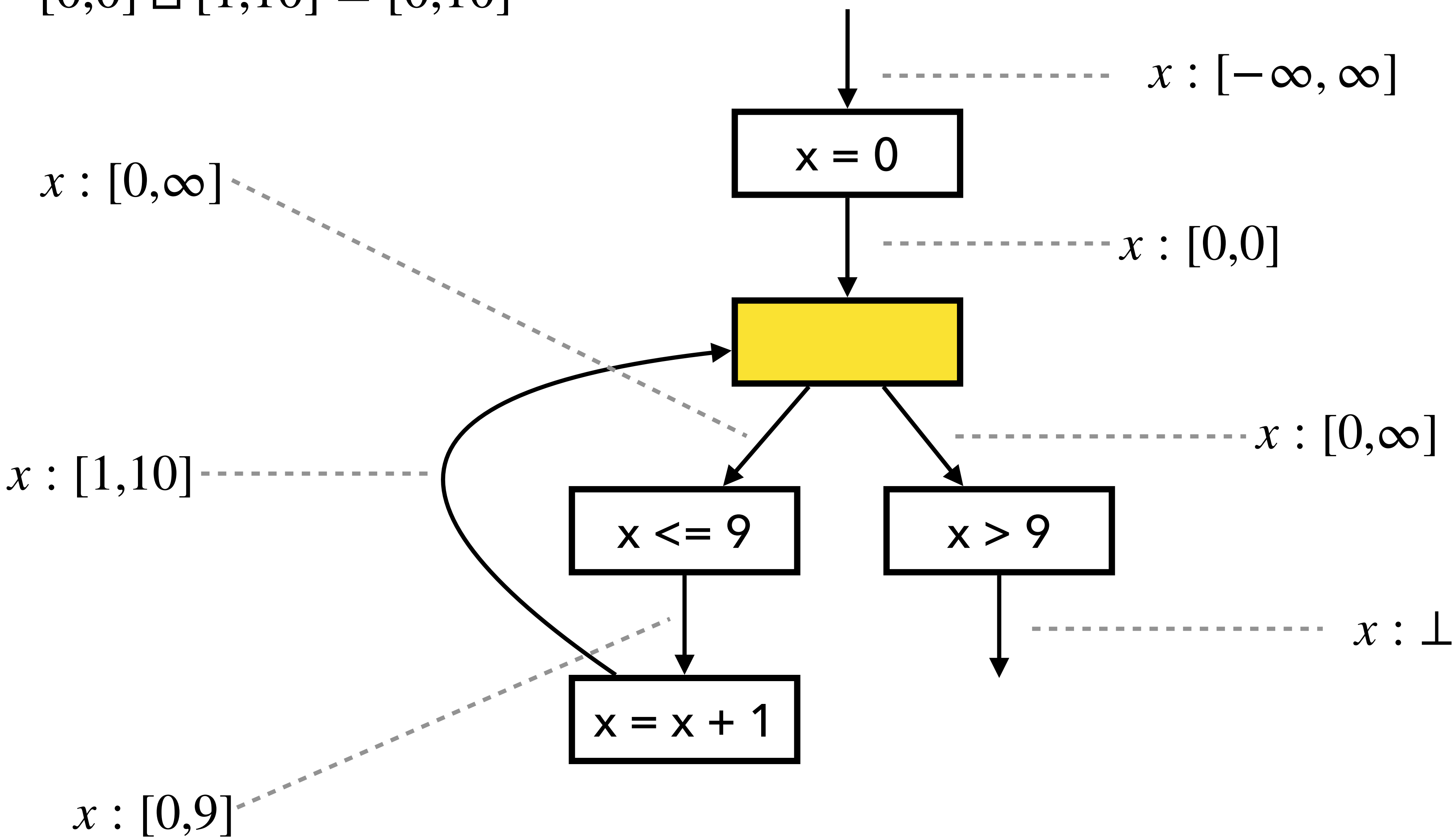
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

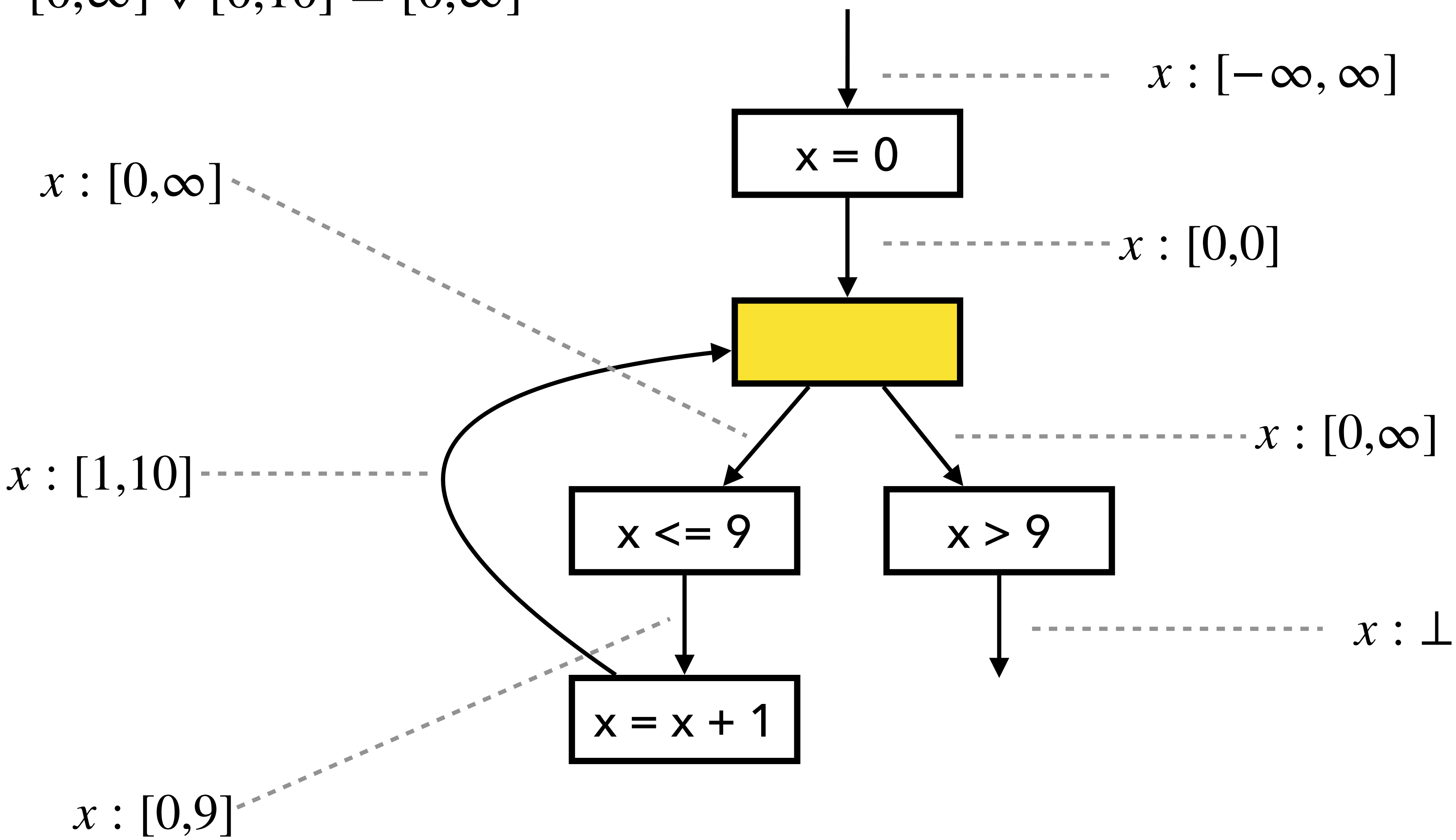
$$[0,0] \sqcup [1,10] = [0,10]$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

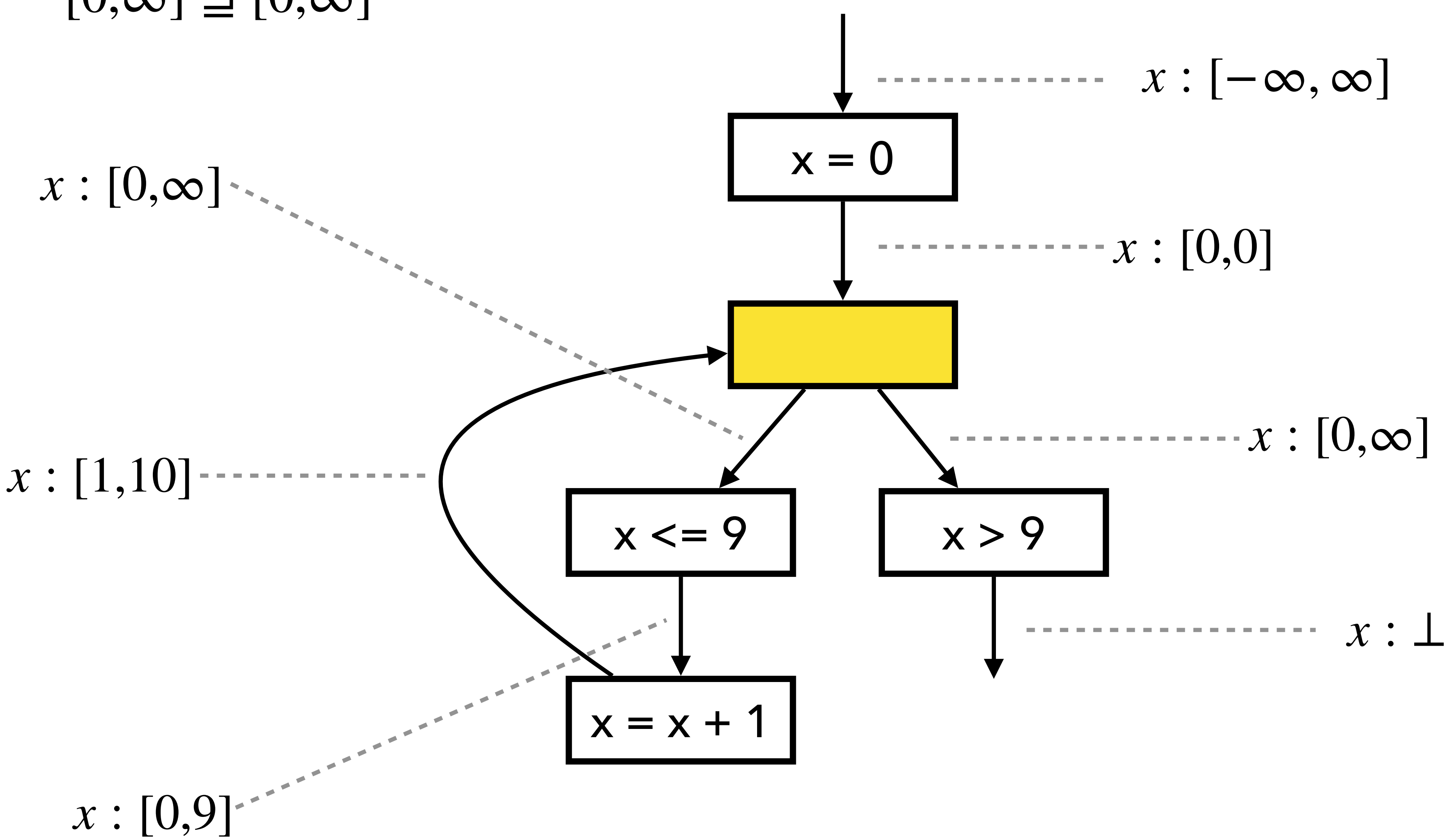
$[0,\infty] \triangledown [0,10] = [0,\infty]$



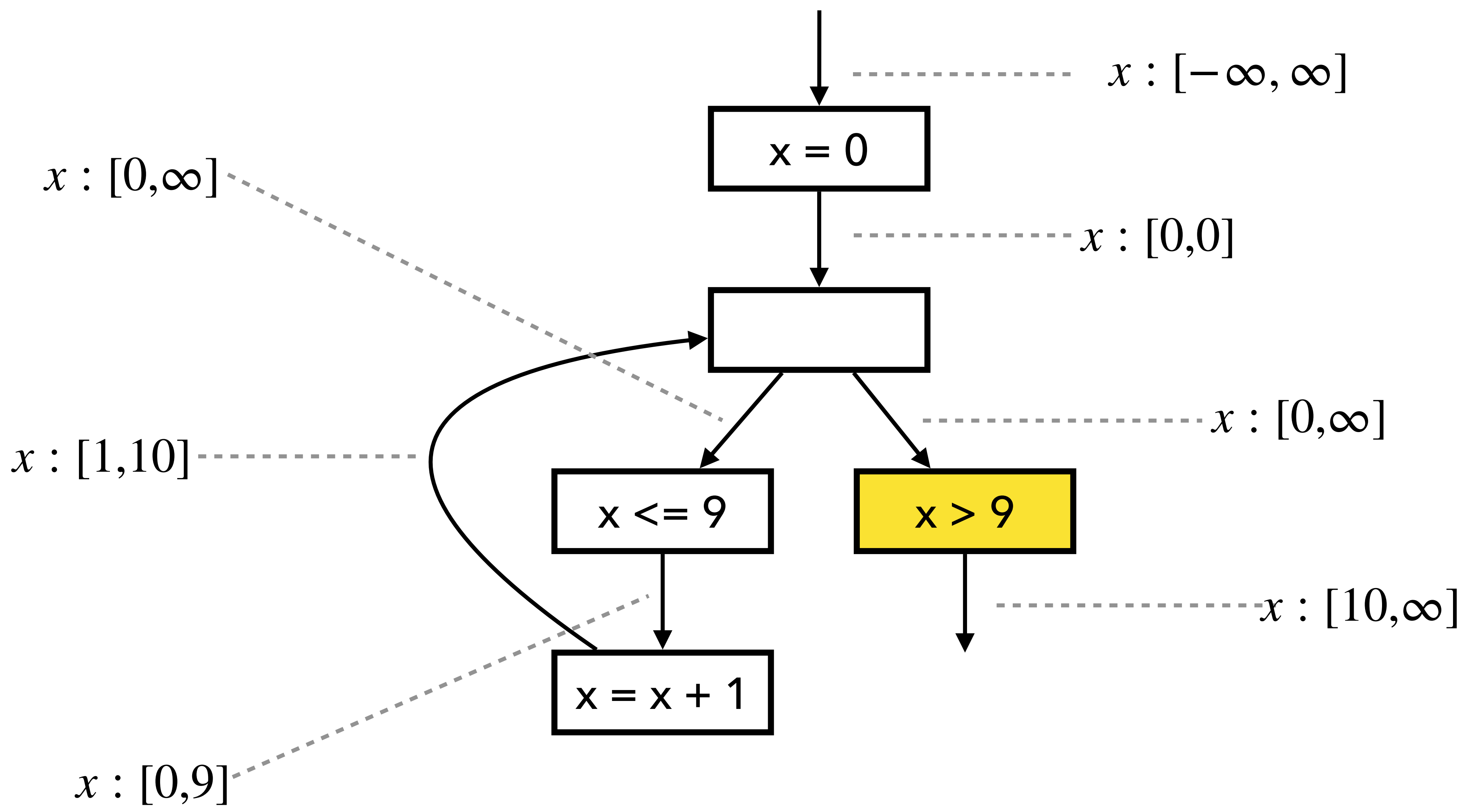
Fixed Point Comp. with Widening

3. Check if fixed point is reached

$[0,\infty] \sqsupseteq [0,\infty]$



Fixed Point Comp. with Widening

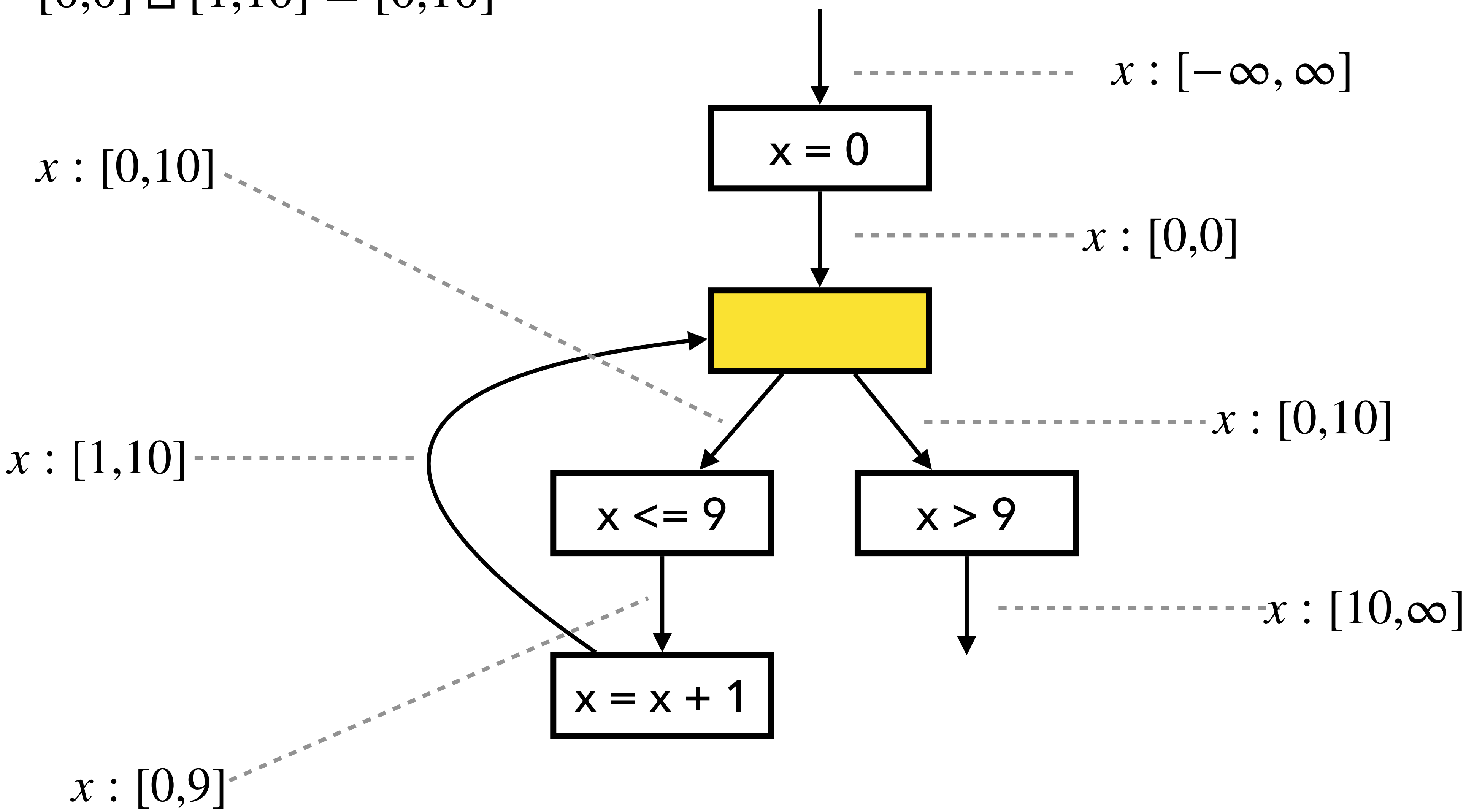


$$[0,\infty] \sqcap [10,\infty] = [10,\infty]$$

Fixed Point Comp. with Narrowing

1. Compute output by joining inputs:

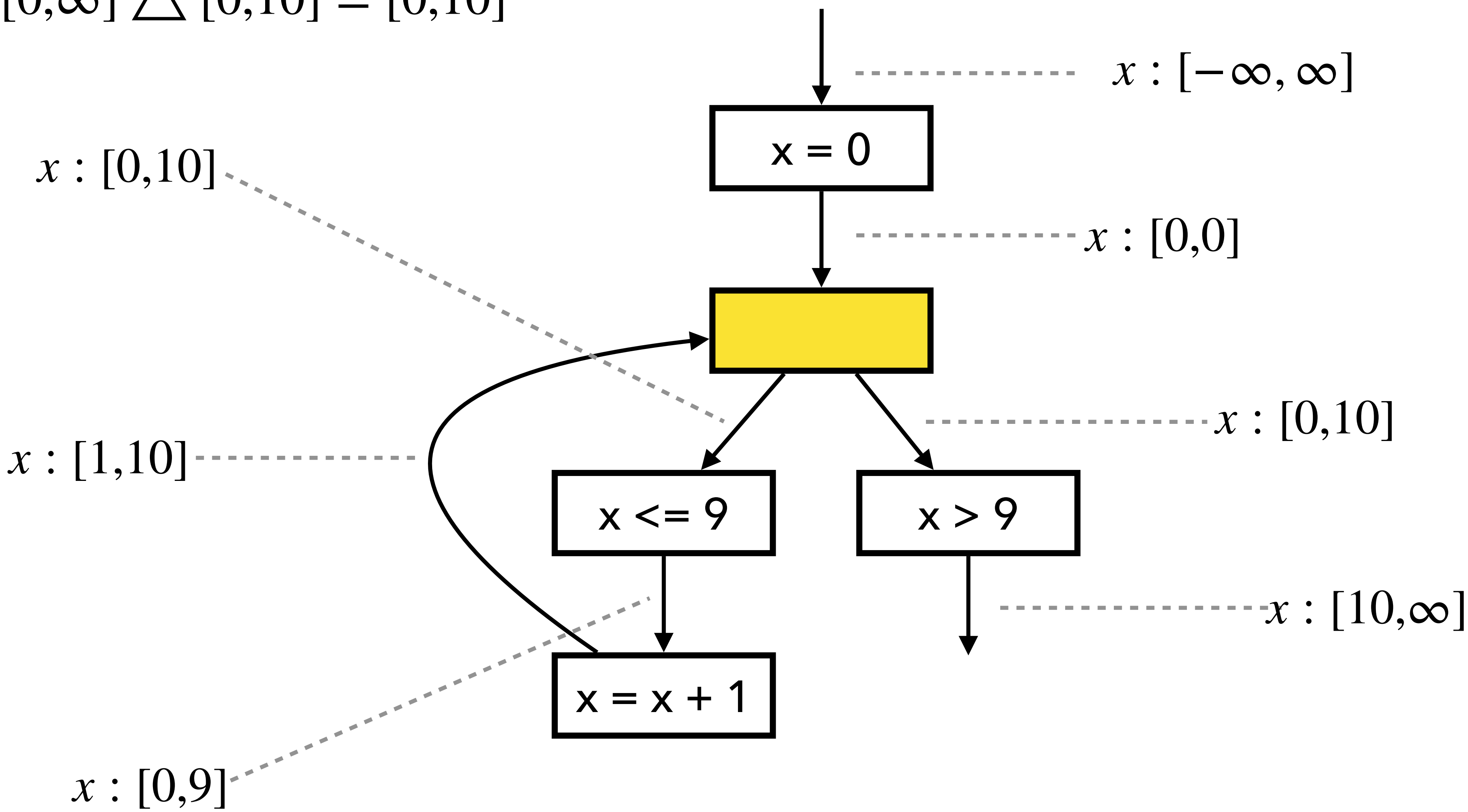
$$[0,0] \sqcup [1,10] = [0,10]$$



Fixed Point Comp. with Narrowing

2. Apply narrowing with old output:

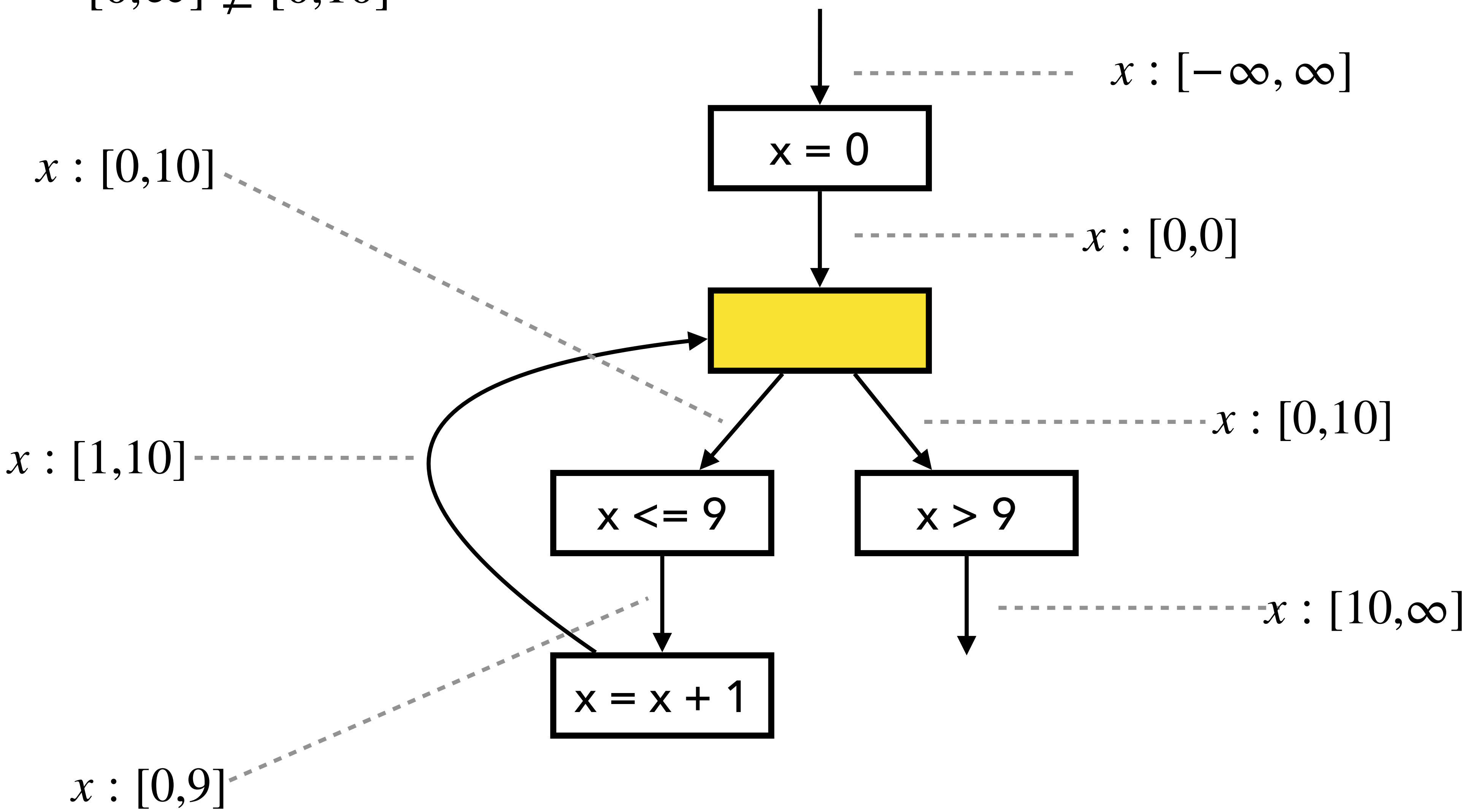
$[0,\infty] \triangle [0,10] = [0,10]$



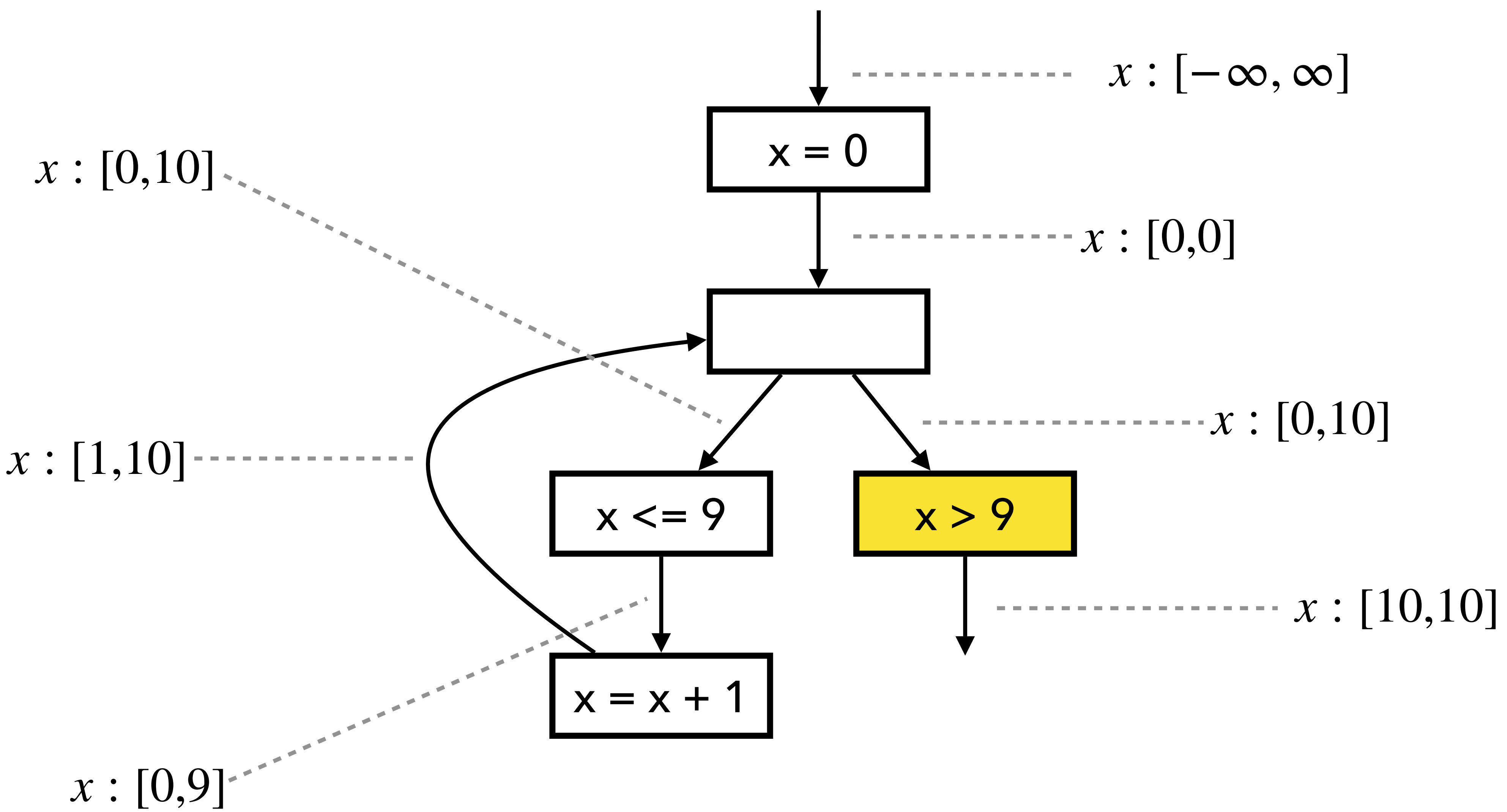
Fixed Point Comp. with Narrowing

3. Check if fixed point is reached:

$$[0,\infty] \not\sqsubseteq [0,10]$$



Fixed Point Comp. with Narrowing



The Interval Domain

- The set of intervals:

$$\hat{\mathbb{Z}} = \{\ \bot\ \} \cup \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, \infty\}, l \leq u\}$$

- Partial order:

$$\bot \sqsubseteq \hat{z} \quad (\text{for any } \hat{z} \in \hat{\mathbb{Z}}) \qquad [l_1, u_1] \sqsubseteq [l_2, u_2] \iff l_2 \leq l_1 \wedge u_1 \leq u_2$$

- Join:

$$\bot \sqcup \hat{z} = \hat{z} \qquad \hat{z} \sqcup \bot = \hat{z} \qquad [l_1, u_1] \sqcup [l_2, u_2] = [\text{min}(l_1, l_2), \text{max}(u_1, u_2)]$$

- Meet:

$$[l_1, u_1] \sqcap [l_2, u_2] = [l_2, u_1] \quad (\text{if } l_1 \leq l_2 \wedge l_2 \leq u_1)$$

$$[l_1, u_1] \sqcap [l_2, u_2] = [l_1, u_2] \quad (\text{if } l_2 \leq l_1 \wedge l_1 \leq u_2)$$

$$\hat{z}_1 \sqcap \hat{z}_2 = \bot \quad (\text{otherwise})$$

The Interval Domain

- Widening:

$$\bot \bigtriangledown \hat{z} = \hat{z}$$
$$\hat{z} \bigtriangledown \bot = \hat{z}$$
$$[l_1, u_1] \bigtriangledown [l_2, u_2] = [l_1 > l_2 ? -\infty : l_1, u_1 < u_2 ? +\infty : u_1]$$

- Narrowing:

$$\bot \bigtriangleup \hat{z} = \bot$$
$$\hat{z} \bigtriangleup \bot = \bot$$
$$[l_1, u_1] \bigtriangleup [l_2, u_2] = [l_1 = -\infty ? l_2 : l_1, u_1 = +\infty ? u_2 : u_1]$$

The Interval Domain

- Addition / Subtraction / Multiplication:

$$[l_1, u_1] \hat{+} [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$$

$$[l_1, u_1] \hat{-} [l_2, u_2] = [l_1 - u_2, u_1 - l_2]$$

$$[l_1, u_1] \hat{\times} [l_2, u_2] = [\text{min}(l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2), \text{max}(l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2)]$$

- Equality (=) produces T except for the cases:

$$[l_1, u_1] \hat{=} [l_2, u_2] = tr\hat{u}e \quad (\text{if } l_1 = u_1 = l_2 = u_2)$$

$$[l_1, u_1] \hat{=} [l_2, u_2] = fa\hat{l}se \quad (\text{no overlap})$$

- ``Less than" (<) produces T except for the cases:

$$[l_1, u_1] \hat{<} [l_2, u_2] = tr\hat{u}e \quad (\text{if } u_1 < l_2)$$

$$[l_1, u_1] \hat{<} [l_2, u_2] = fa\hat{l}se \quad (\text{if } l_1 > u_2)$$

Abstract Memory

$$\hat{\mathbb{M}} = \textbf{Var} \rightarrow \hat{\mathbb{Z}}$$

$$m_1 \sqsubseteq m_2 \iff \forall x \in \textbf{Var} \,.\, m_1(x) \sqsubseteq m_2(x)$$

$$m_1 \sqcup m_2 = \lambda x \,.\, m_1(x) \sqcup m_2(x)$$

$$m_1 \sqcap m_2 = \lambda x \,.\, m_1(x) \sqcap m_2(x)$$

$$m_1 \bigtriangledown m_2 = \lambda x \,.\, m_1(x) \bigtriangledown m_2(x)$$

$$m_1 \bigtriangleup m_2 = \lambda x \,.\, m_1(x) \bigtriangleup m_2(x)$$

Worklist Algorithm

Fixpoint comp. with widening

$W :=$ **Node**
$T := \lambda n\,.\,\bot_{\hat{\mathbb{M}}}$
$\mathit{while}\ W \neq \emptyset$
$\quad n := \mathit{choose}(W)$
$\quad W := W \setminus \{n\}$
$\quad \mathit{in} := \mathit{inputof}(n, T)$
$\quad \mathit{out} := \mathit{analyze}(n, \mathit{in})$
$\quad \mathit{if}\ \mathit{out} \not\sqsubseteq T(n)$
$\qquad \mathit{if\ widening\ is\ needed}$
$\qquad\quad T(n) := T(n) \bigtriangledown \mathit{out}$
$\qquad \mathit{else}$
$\qquad\quad T(n) := T(n) \sqcup \mathit{out}$
$\qquad W := W \cup \mathit{succ}(n)$

Fixpoint comp. with narrowing

$W :=$ **Node**
$\mathit{while}\ W \neq \emptyset$
$\quad n := \mathit{choose}(W)$
$\quad W := W \setminus \{n\}$
$\quad \mathit{in} := \mathit{inputof}(n, T)$
$\quad \mathit{out} := \mathit{analyze}(n, \mathit{in})$
$\quad \mathit{if}\ T(n) \not\sqsubseteq \mathit{out}$
$\qquad T(n) := T(n) \bigtriangleup \mathit{out}$
$\qquad W := W \cup \mathit{succ}(n)$

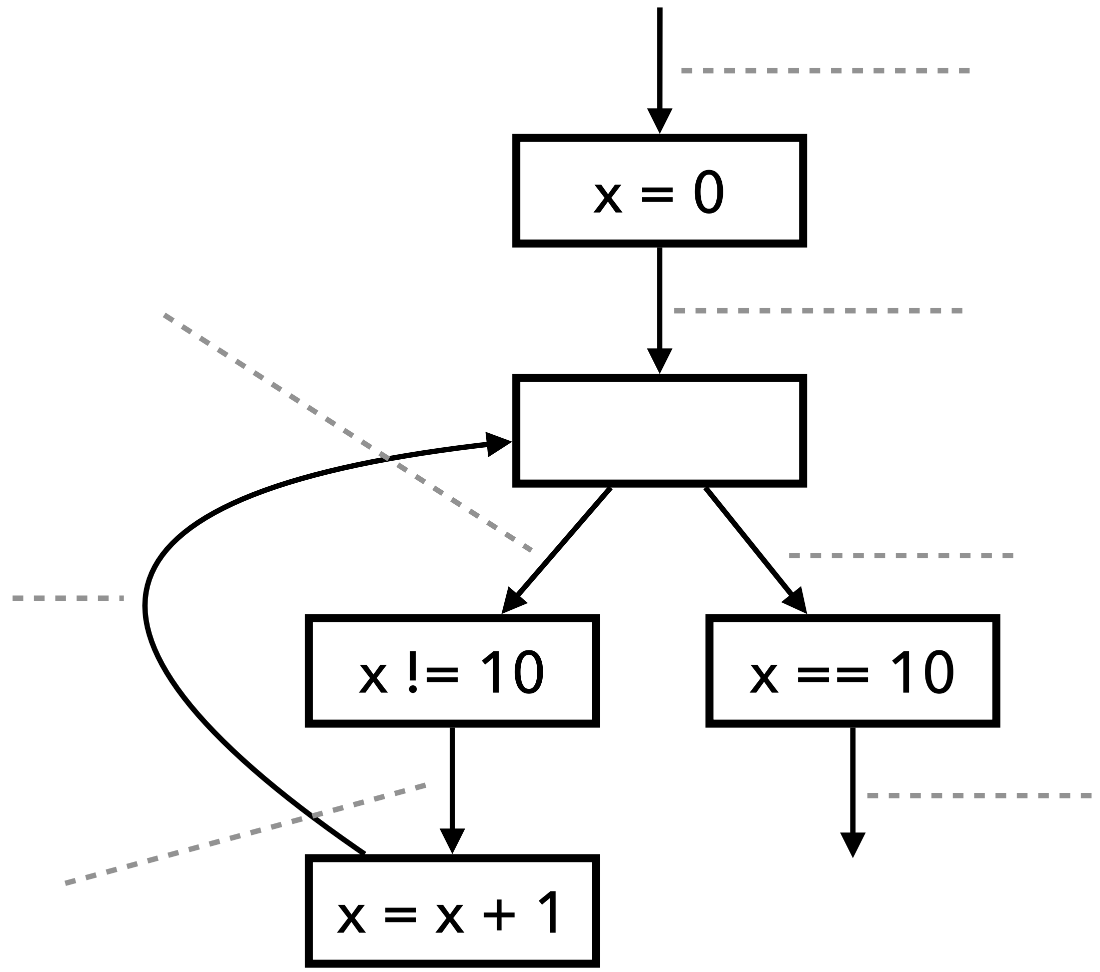
Exercise (2)

Describe the result of the interval analysis:

(1) without widening

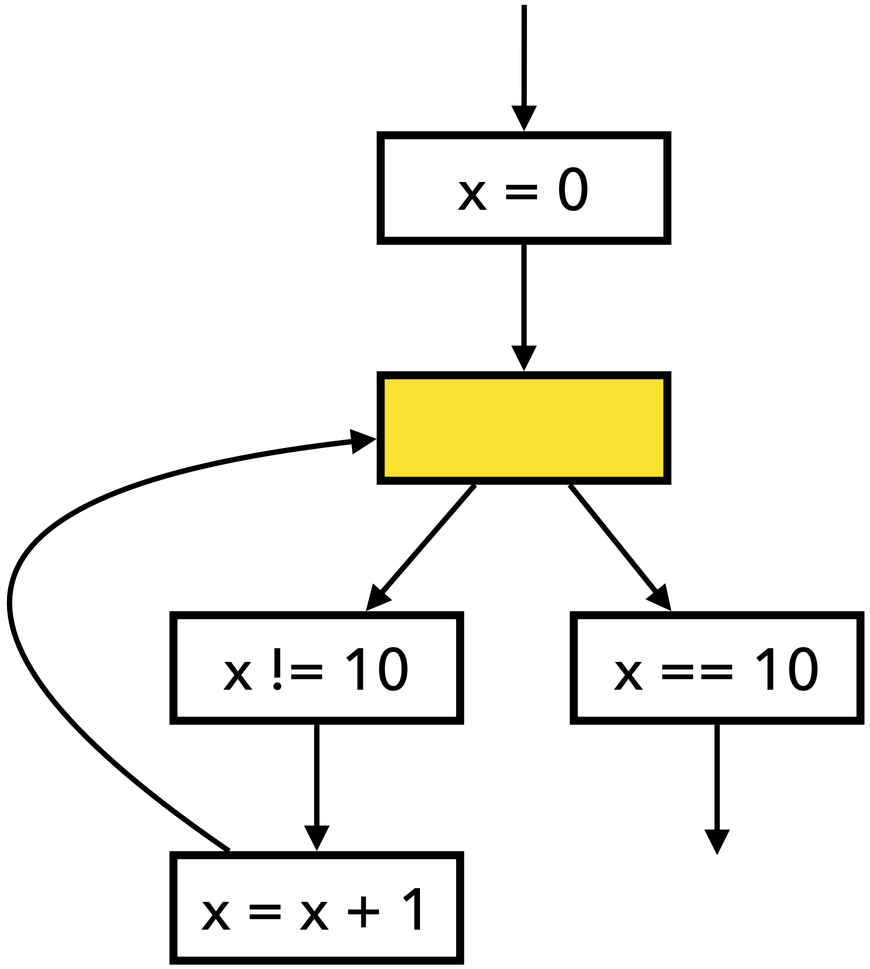
(2) with widening/narrowing

```
x = 0;

while (x != 10)
  x = x + 1;
```



Widening with Thresholds

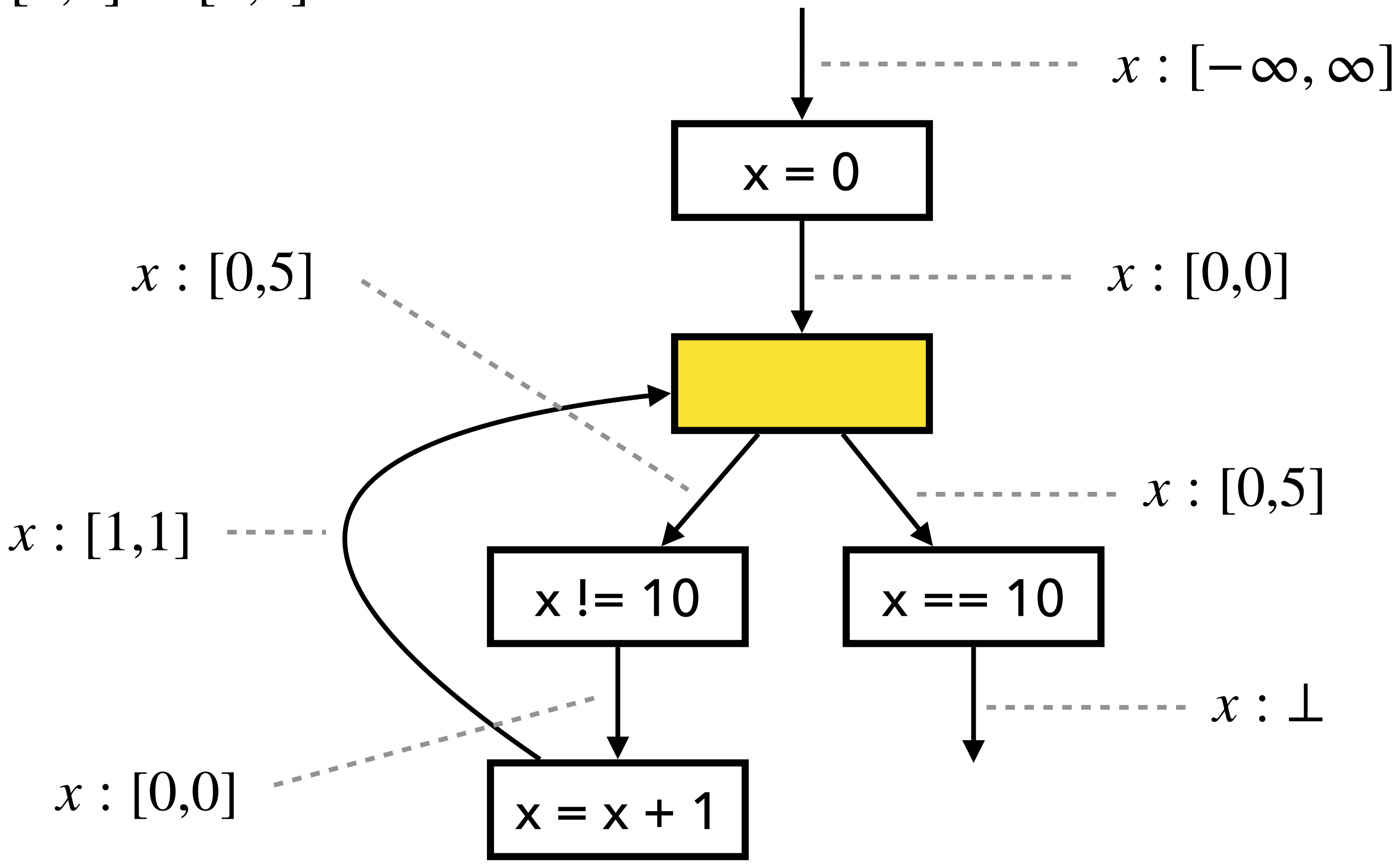
Assume a set T of thresholds is given beforehand: e.g., $T = \{5,10\}$



Widening with Thresholds

1. Compute output by joining inputs:

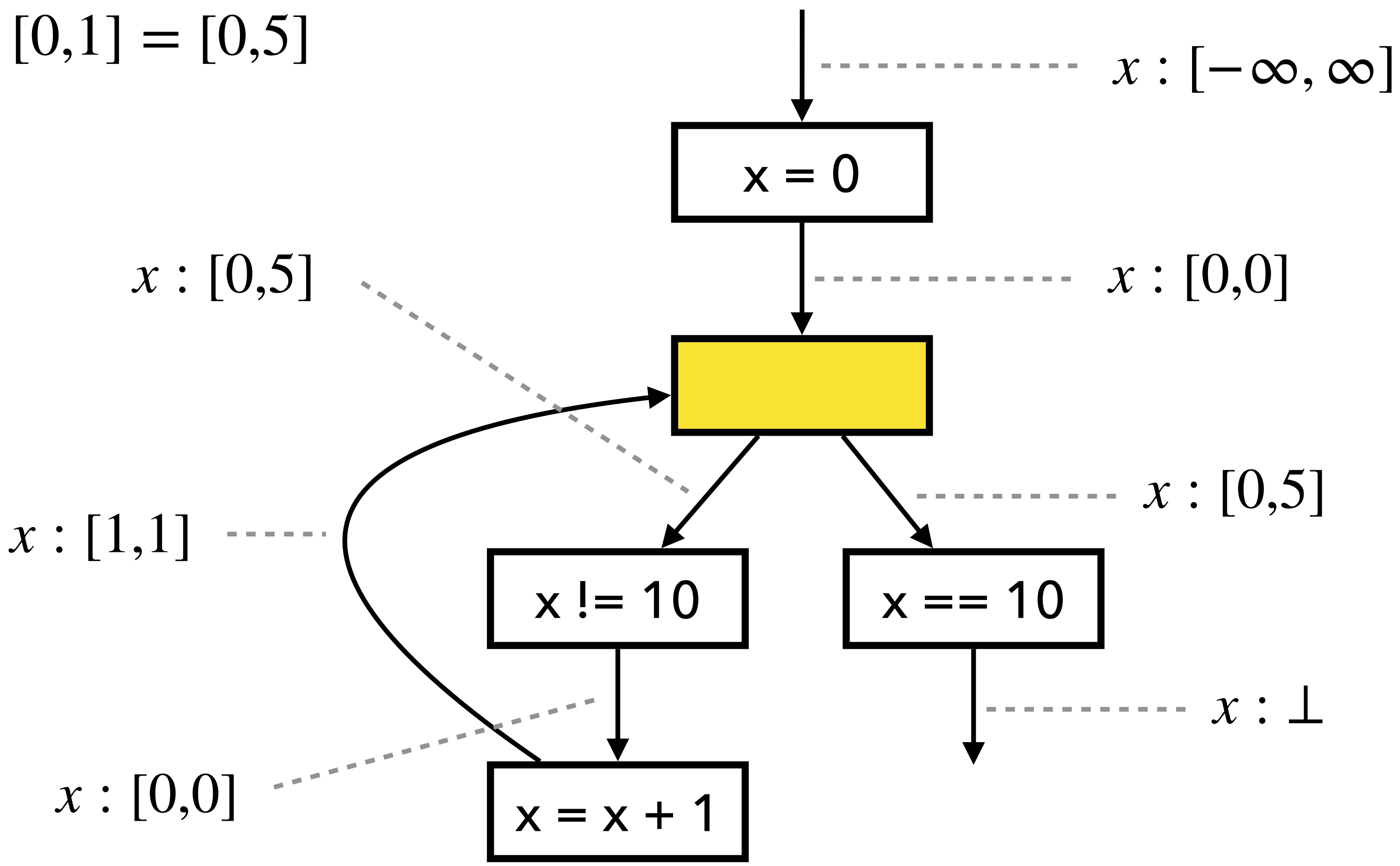
$$[0,0] \sqcup [1,1] = [0,1]$$



Widening with Thresholds

2. Given $T = \{5,10\}$, use 5 as threshold when applying widening:

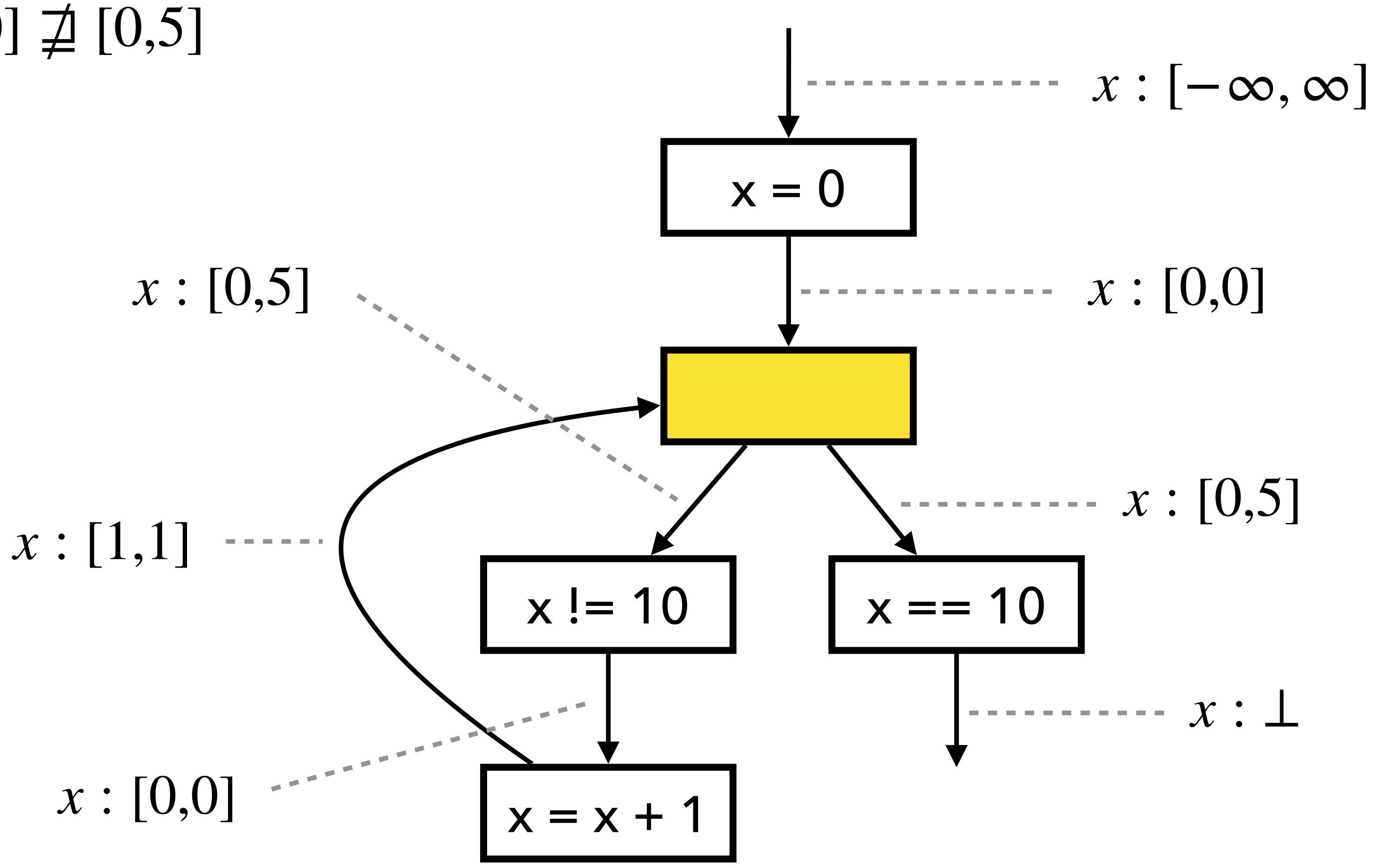
$$[0,0] \triangledown [0,1] = [0,5]$$



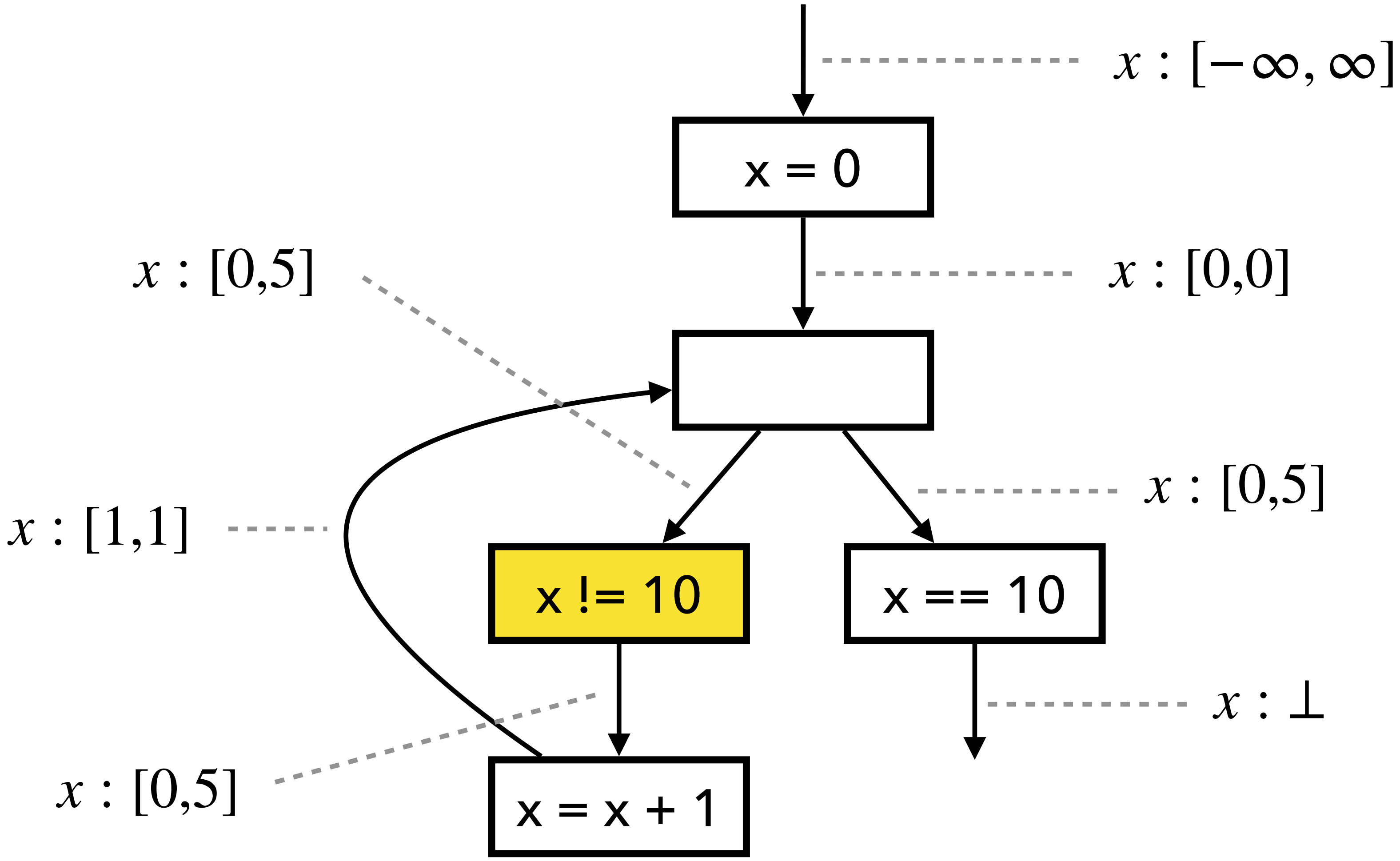
Widening with Thresholds

3. Check if fixed point is reached:

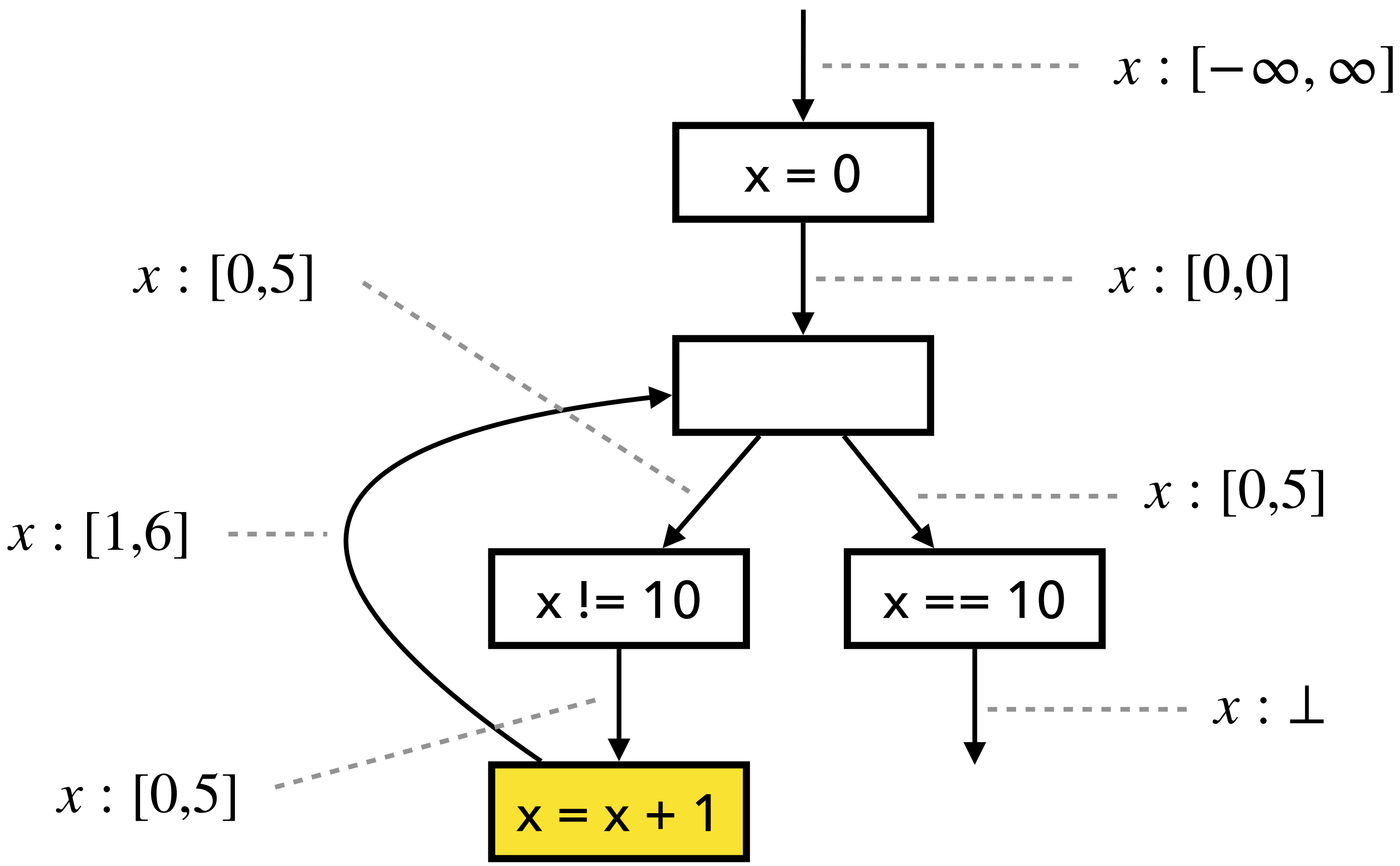
$$[0,0] \not\sqsupseteq [0,5]$$



Widening with Thresholds



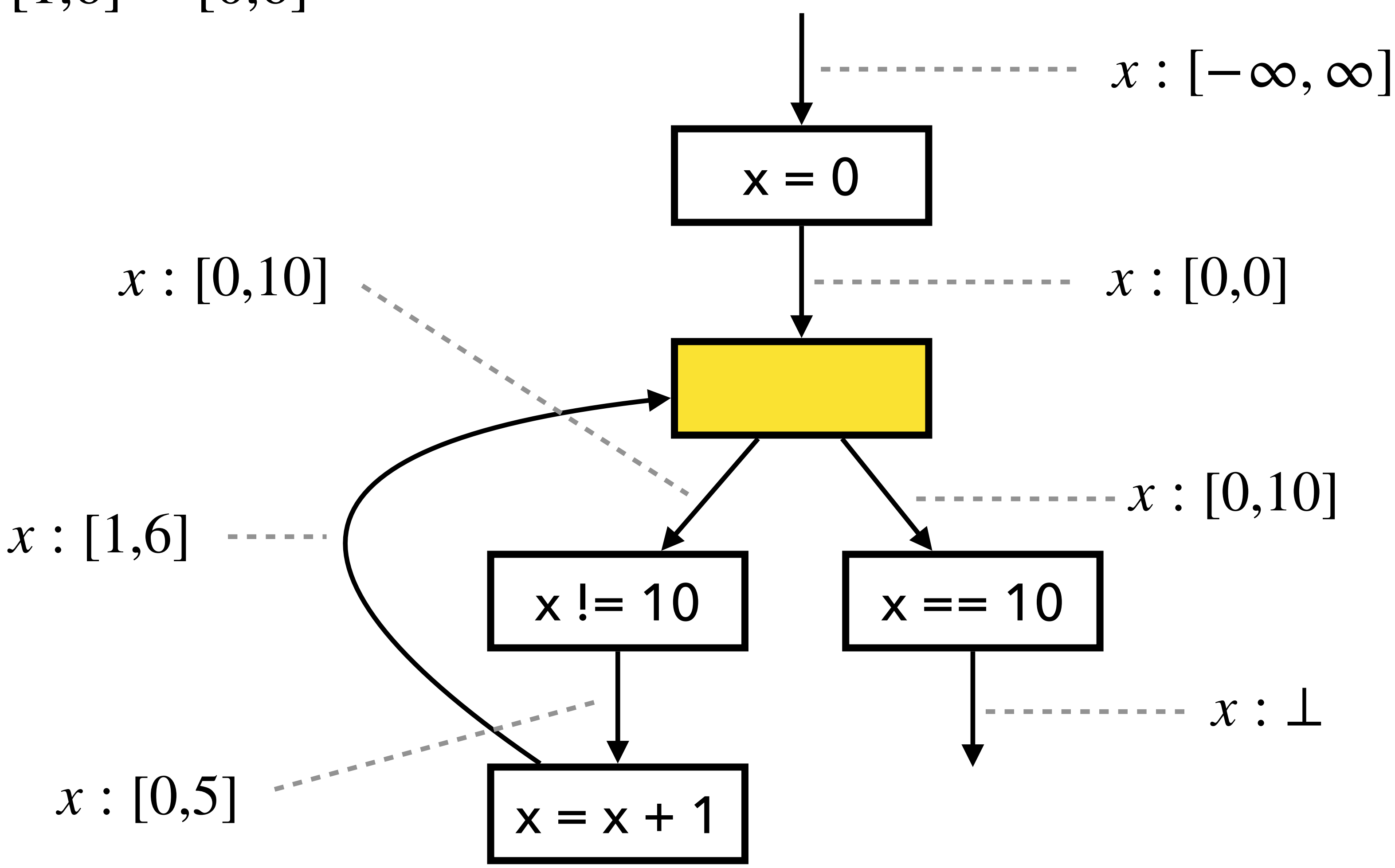
Widening with Thresholds



Widening with Thresholds

1. Compute output by joining inputs:

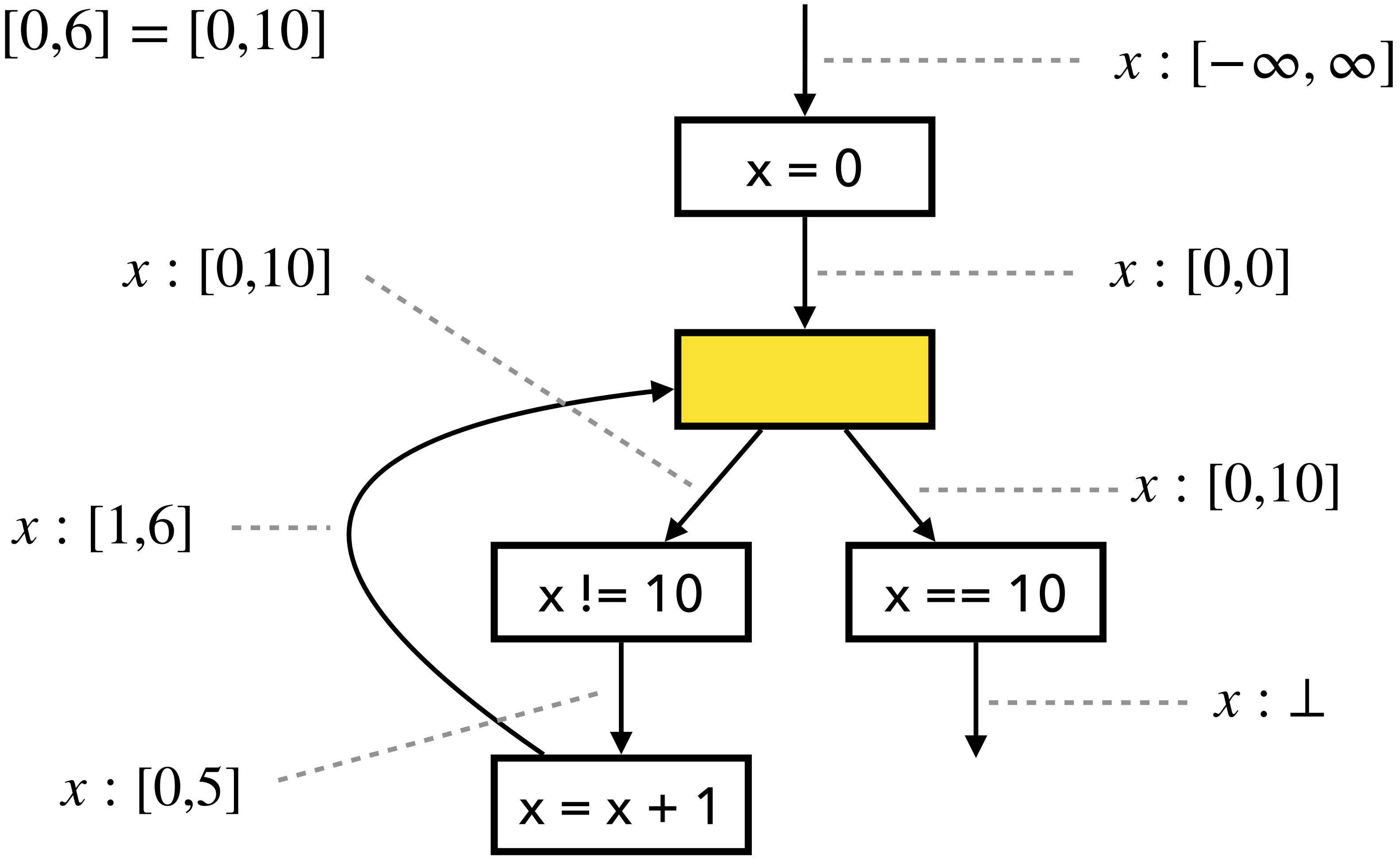
$$[0,0] \sqcup [1,6] = [0,6]$$



Widening with Thresholds

2. Given $T = \{5,10\}$, use 10 as threshold when applying widening:

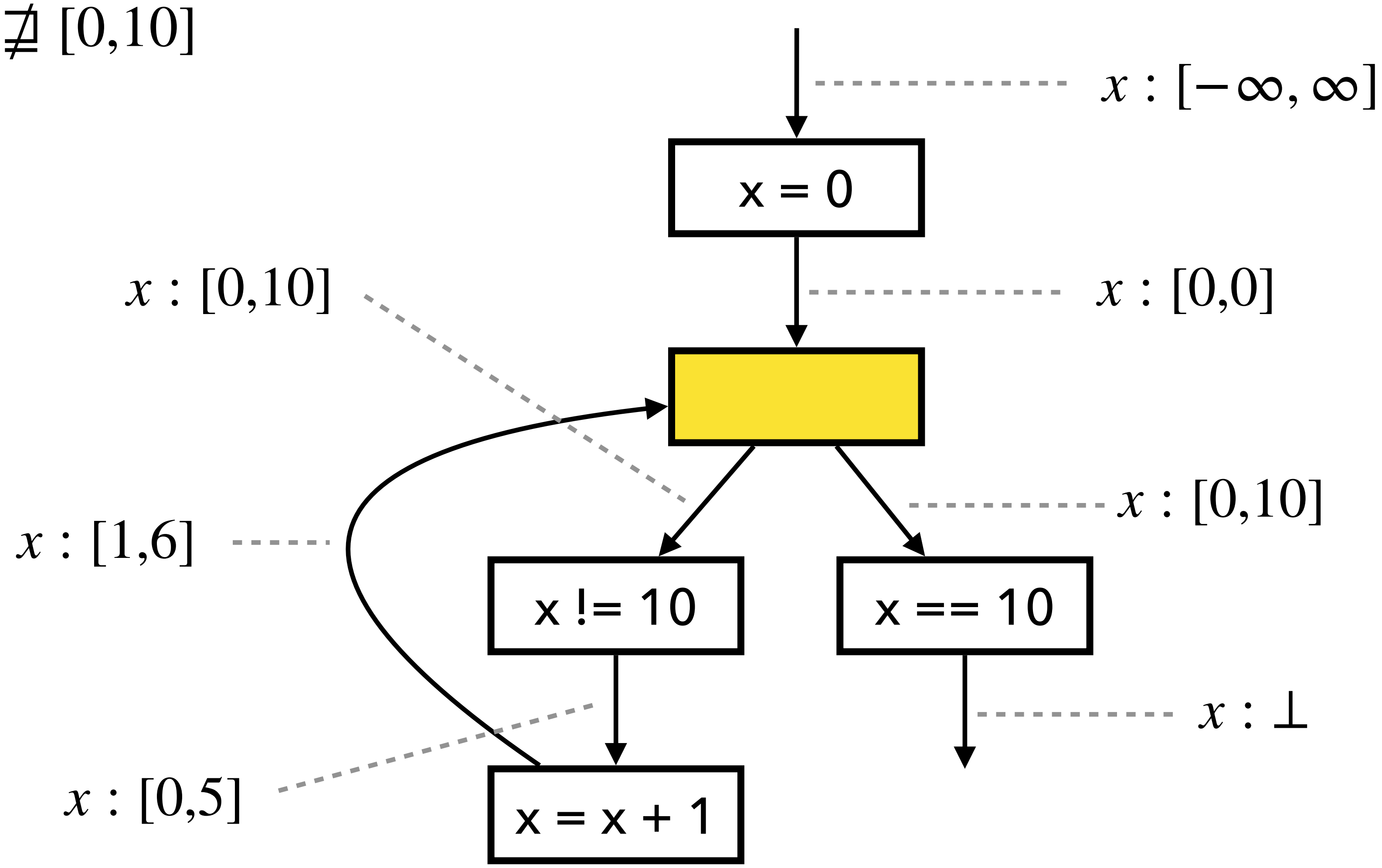
$$[0,5] \triangledown [0,6] = [0,10]$$



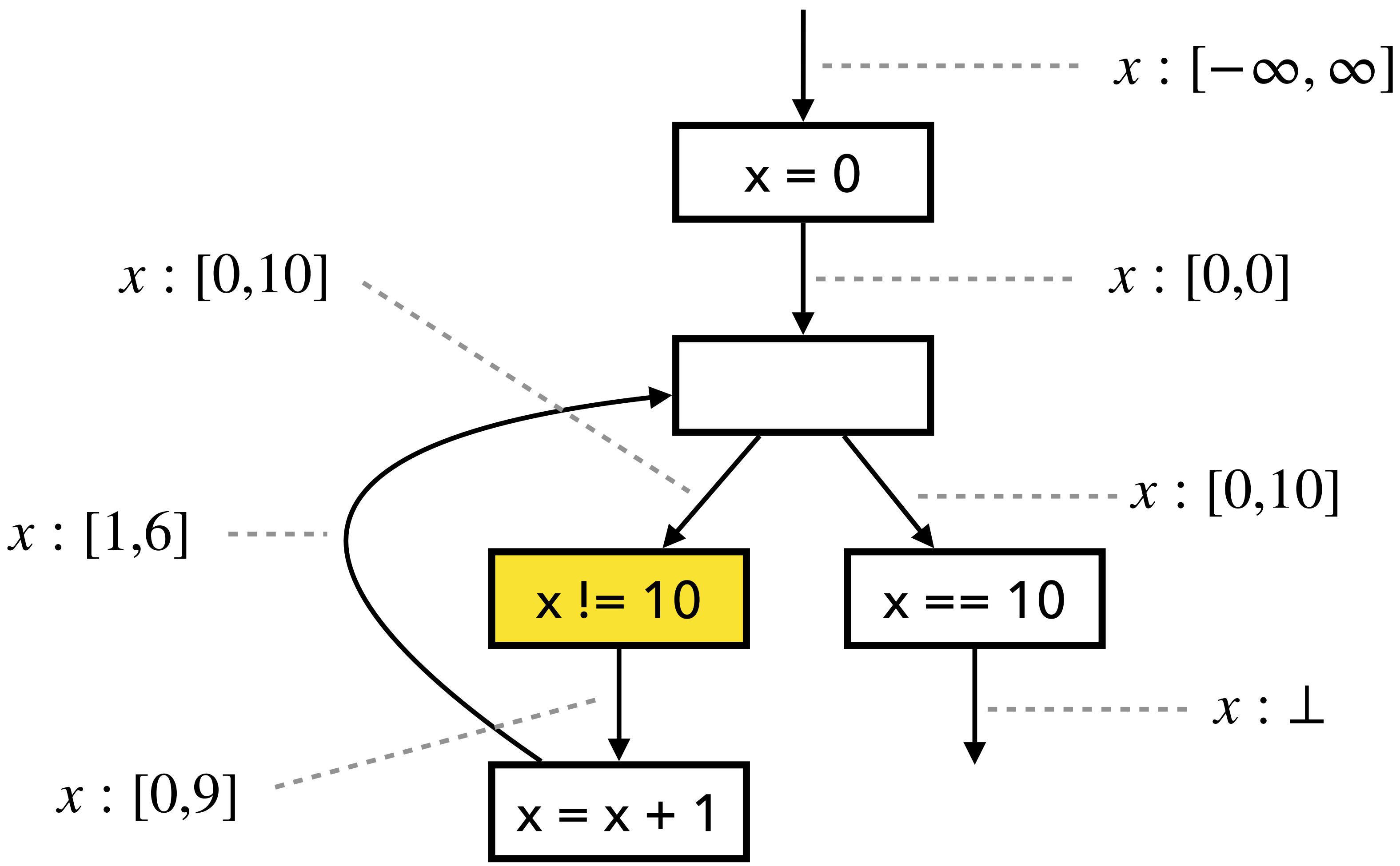
Widening with Thresholds

3. Check if fixed point is reached:

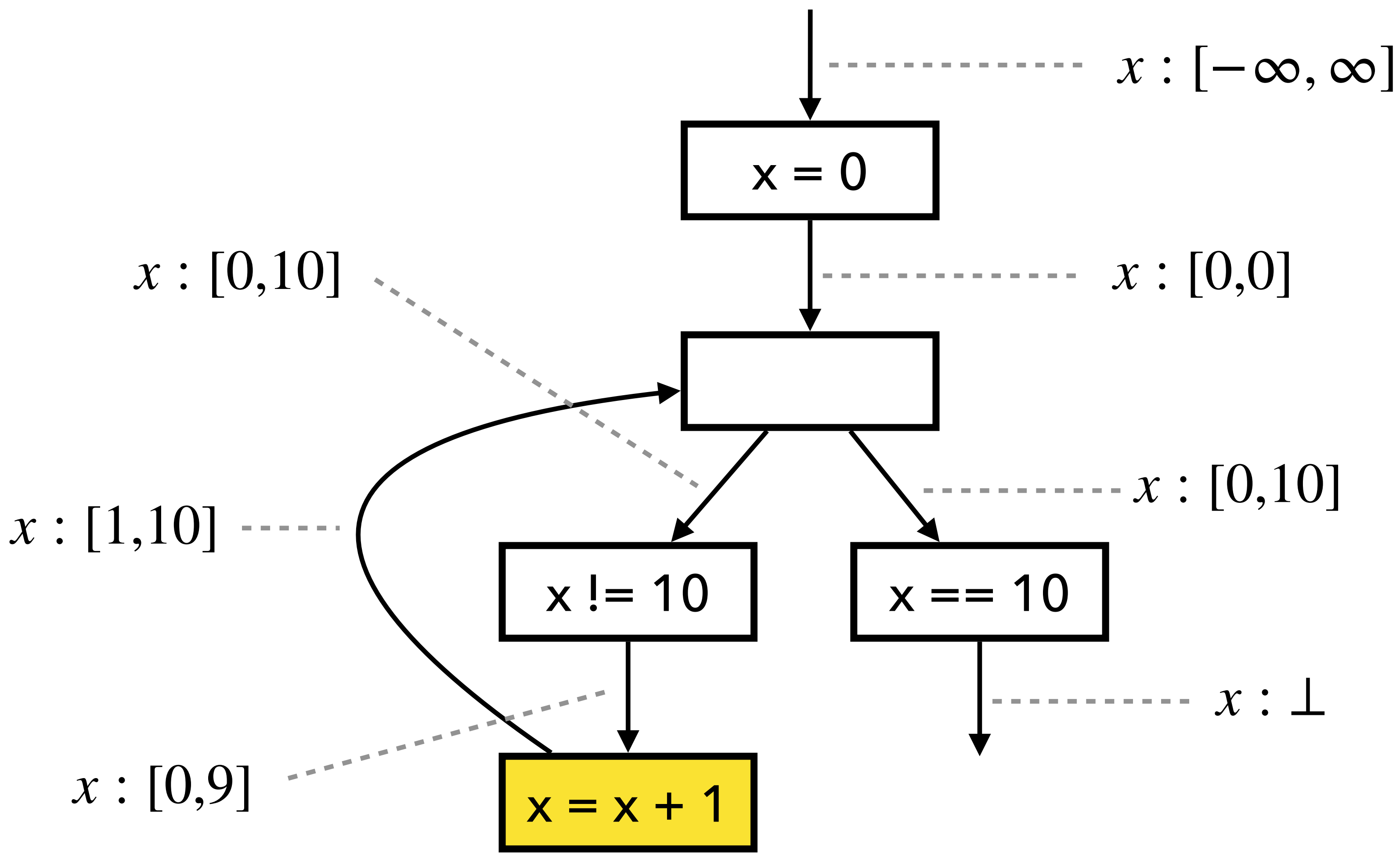
$$[0,5] \not\sqsupseteq [0,10]$$



Widening with Thresholds



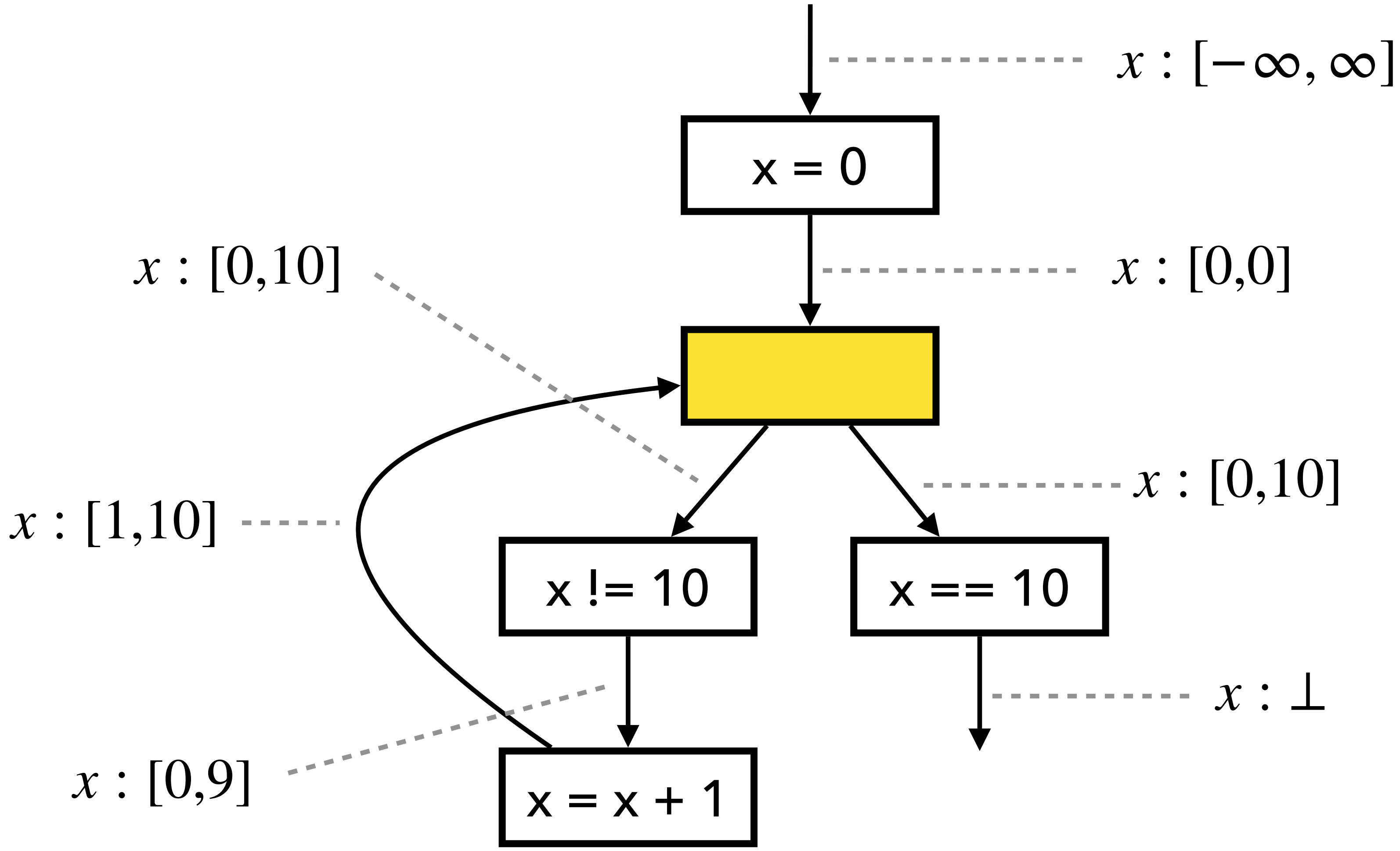
Widening with Thresholds



Widening with Thresholds

1. Compute output by joining inputs:

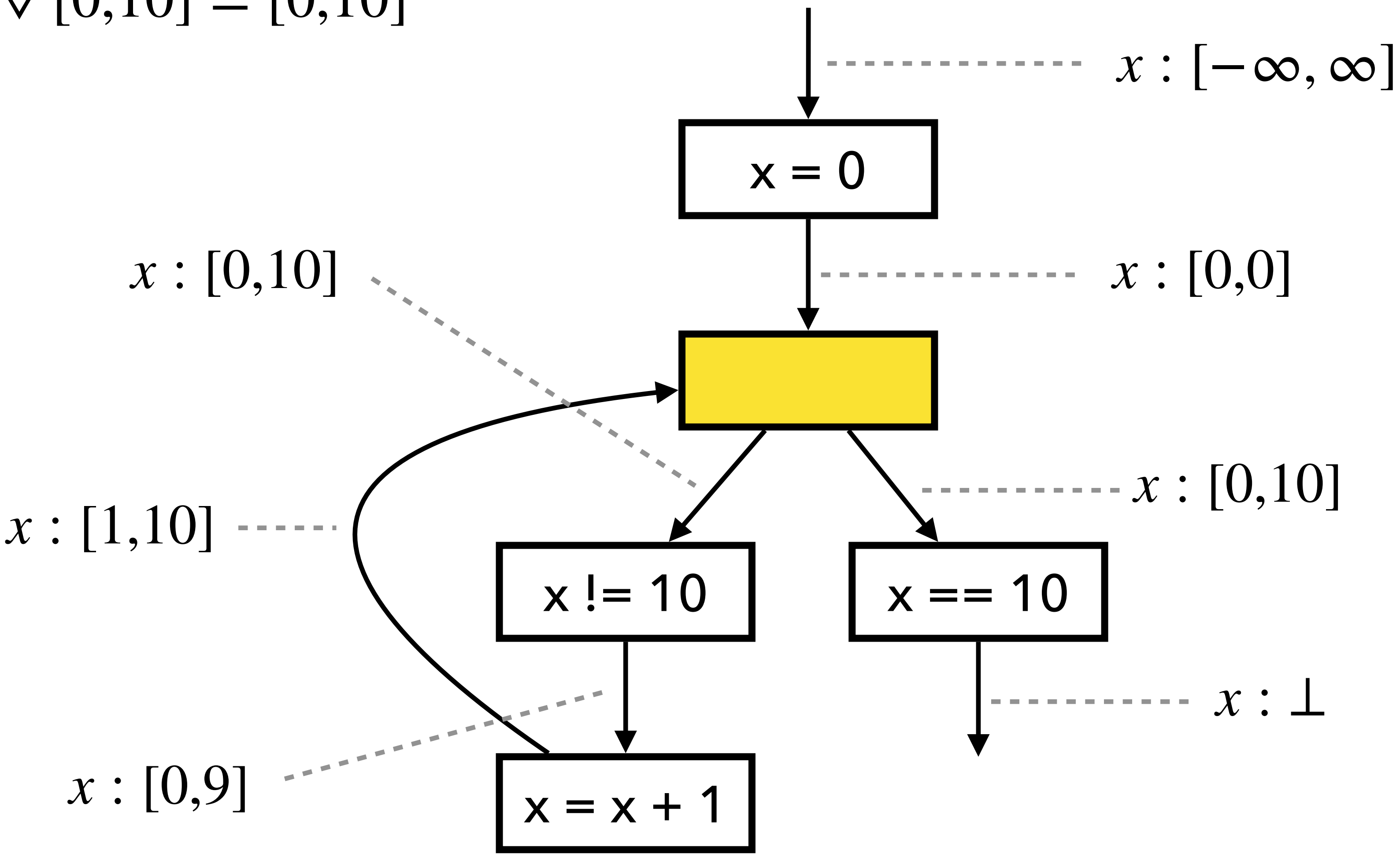
$$[0,0] \sqcup [1,10] = [0,10]$$



Widening with Thresholds

2. Apply widening:

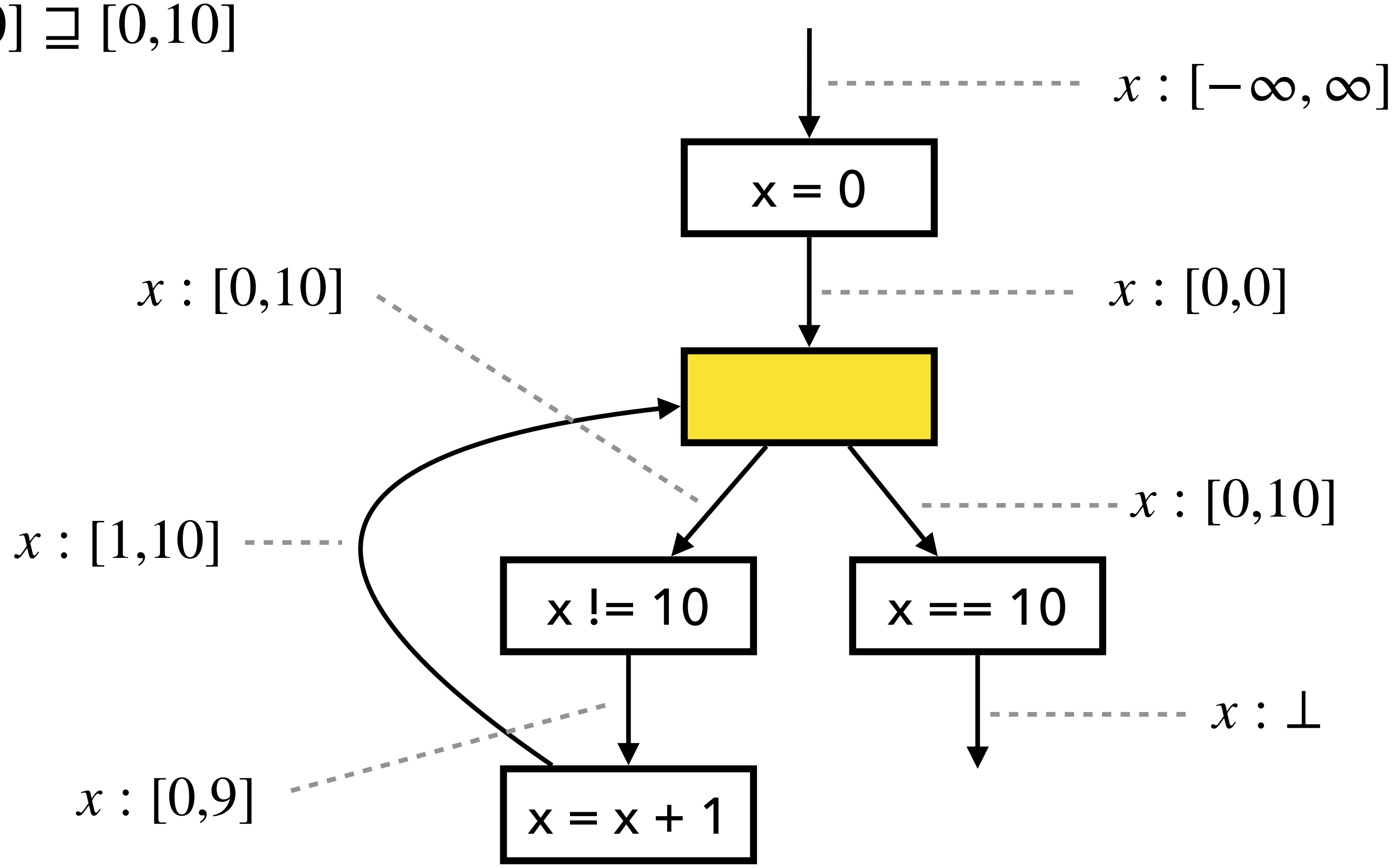
$$[0,10] \triangledown [0,10] = [0,10]$$



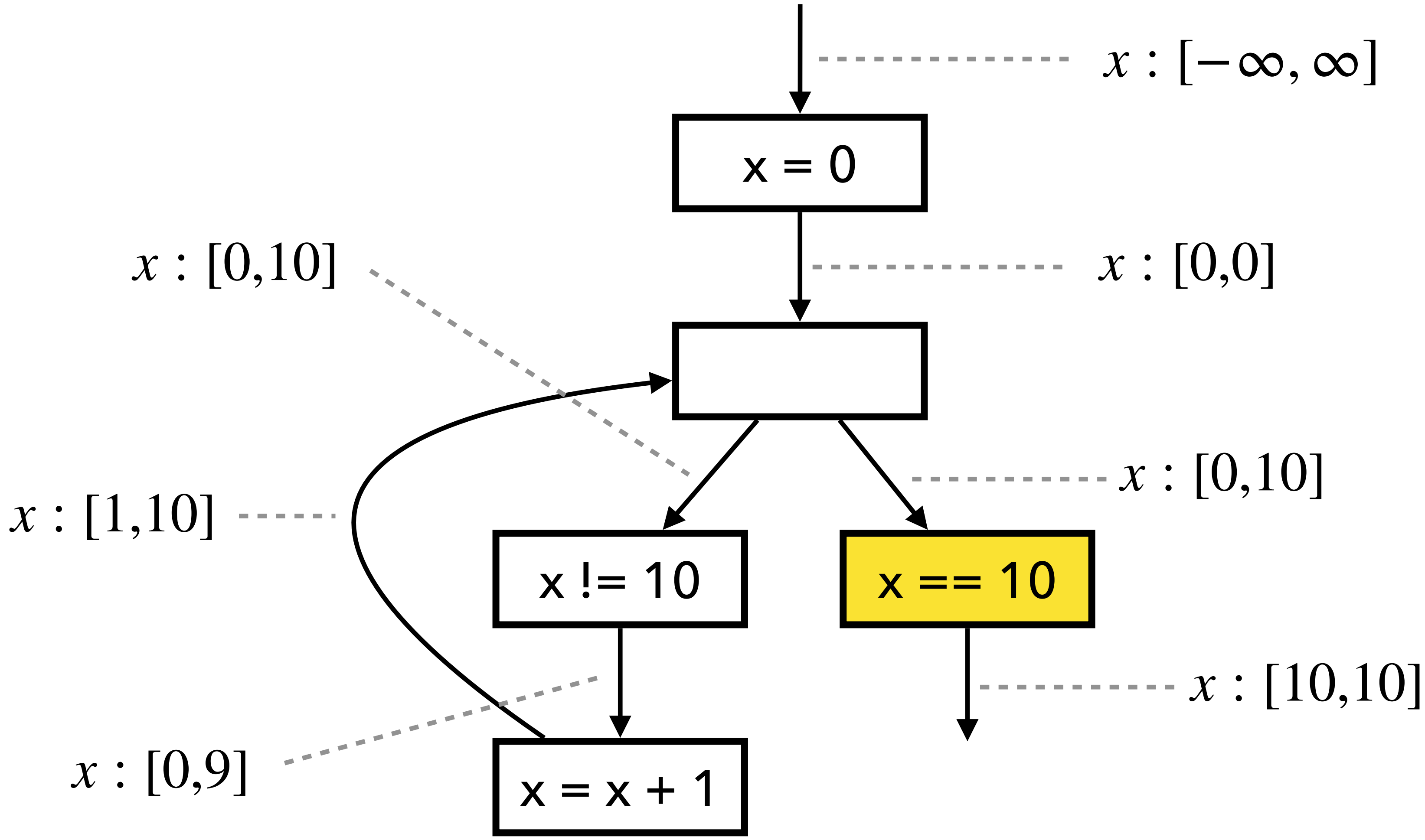
Widening with Thresholds

3. Check if fixed point is reached:

$$[0,10] \sqsupseteq [0,10]$$



Widening with Thresholds



Widening with Thresholds

- A threshold set $T \subseteq \mathbb{Z}$ is given.

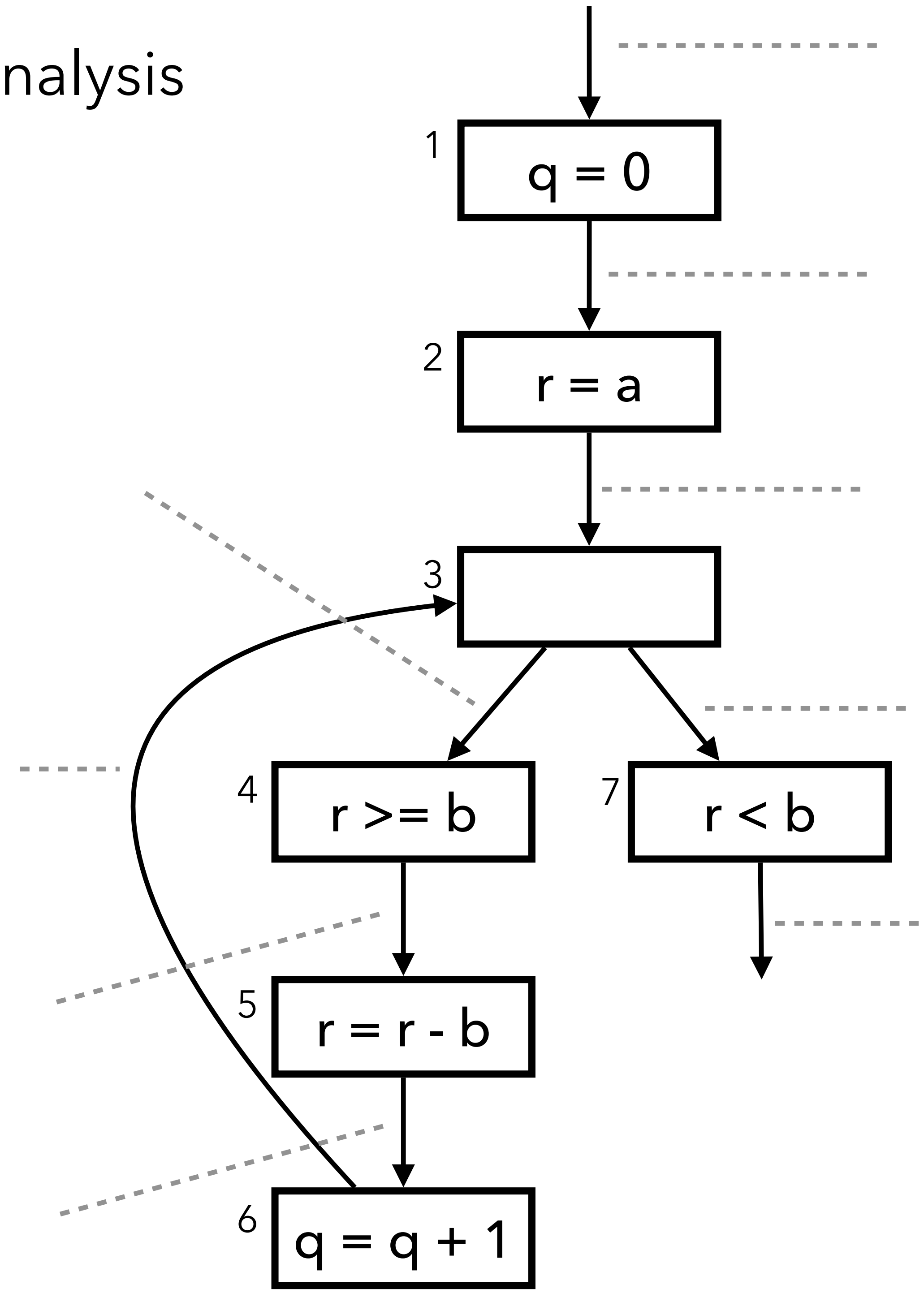
$$\bot \bigtriangledown_T \hat{z} = \hat{z}$$
$$\hat{z} \bigtriangledown_T \bot = \hat{z}$$
$$[l_1, u_1] \bigtriangledown_T [l_2, u_2] = [l_1 > l_2 ? glb(T, l_2) : l_1, u_1 < u_2 ? lub(T, u_2) : u_1]$$

$$glb(T, n) = max\{t \in T \mid t \leq n\}$$
$$lub(T, n) = min\{t \in T \mid t \geq n\}$$

Exercise (3)

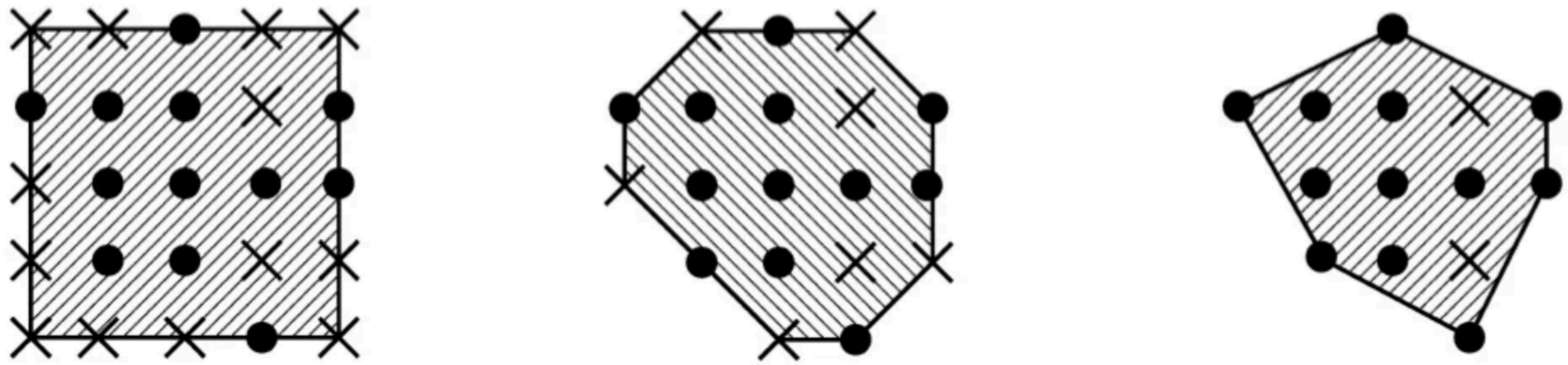
Describe the result of the interval analysis with widening and narrowing

```
// a >= 0, b >= 0
q = 0;
r = a;
while (r >= b) {
  r = r - b;
  q = q + 1;
}
assert(q >= 0);
assert(r >= 0);
```



Relational Abstract Domains

- Intervals vs. Octagons vs. Polyhedra



- Focus: Core idea of the Octagon domain*

```
int a[10];
x = 0; y = 0;

while (x < 9) {
    x++; y++;
}

a[y] = 0;
```

Octagon analysis

$x : [9,9]$
$y : [9,9]$
$x - y : [0,0]$
$x + y : [18,18]$

Interval analysis

$x : [9,9]$
$y : [0,\infty]$

*Antoine Miné. The Octagon Abstract Domain. https://arxiv.org/abs/cs/0703084

Difference Bound Matrix (DBM)

- $(N+1) \times (N+1)$ matrix (N: the number of variables): e.g.,

$$\begin{array}{cc} & \begin{array}{ccc} 0 & x & y \end{array} \\ \begin{array}{c} 0 \\ x \\ y \end{array} & \begin{bmatrix} 0-0 & x-0 & y-0 \\ 0-x & x-x & y-x \\ 0-y & x-y & y-y \end{bmatrix} \end{array}$$

- Example

$$\begin{bmatrix} 0 & 10 & 10 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \iff \begin{array}{c} 0 \le x \le 10 \\ 0 \le y \le 10 \\ y - x \le 0 \\ x - y \le 0 \end{array} \qquad \begin{bmatrix} 0 & 10 & +\infty \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \iff \begin{array}{c} 1 \le x \le 10 \\ 0 \le y \\ y - x \le -1 \\ x - y \le 1 \end{array}$$

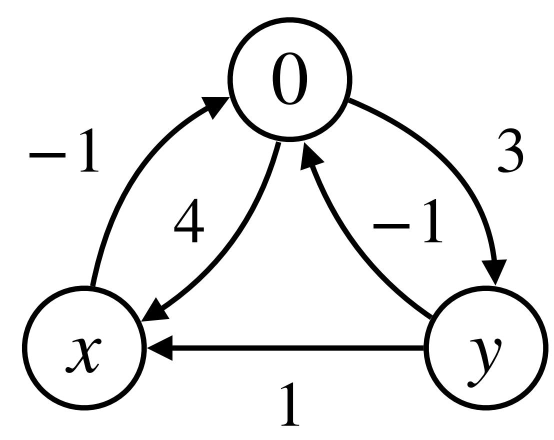
Difference Bound Matrix (DBM)

- A DBM represents a set of program states (N-dim points)

$$\gamma\left(\begin{bmatrix} 0 & 10 & +\infty \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}\right) = \{(x, y) \mid 1 \leq x \leq 10, 0 \leq y, y - x \leq -1, x - y \leq 1\}$$

- A DBM can also be represented by a directed graph

$$\begin{array}{c} \\ 0 \\ x \\ y \end{array} \begin{array}{c} \begin{array}{ccc} 0 & x & y \end{array} \\ \begin{bmatrix} +\infty & 4 & 3 \\ -1 & +\infty & +\infty \\ -1 & 1 & +\infty \end{bmatrix} \end{array} \iff$$



Difference Bound Matrix (DBM)

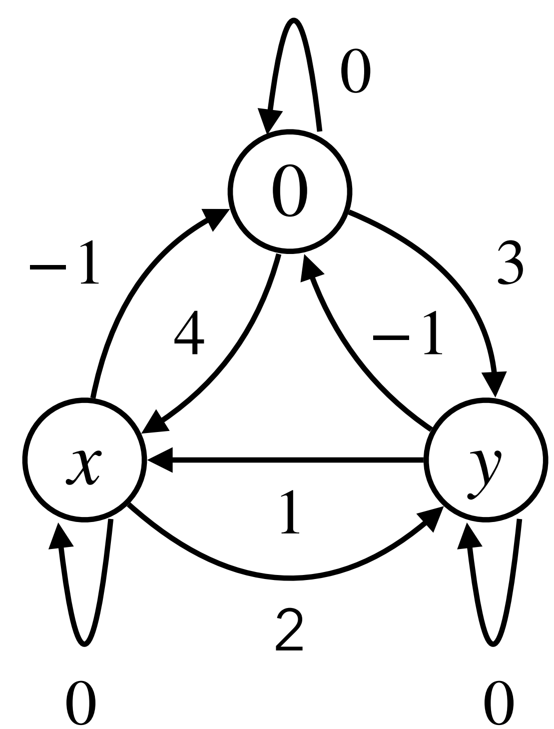
- Two different DBMs can represent the same set of points

$$\gamma\left(\begin{bmatrix} +\infty & 4 & 3 \\ -1 & +\infty & +\infty \\ -1 & 1 & +\infty \end{bmatrix}\right) = \gamma\left(\begin{bmatrix} 0 & 5 & 3 \\ -1 & +\infty & +\infty \\ -1 & 1 & +\infty \end{bmatrix}\right)$$

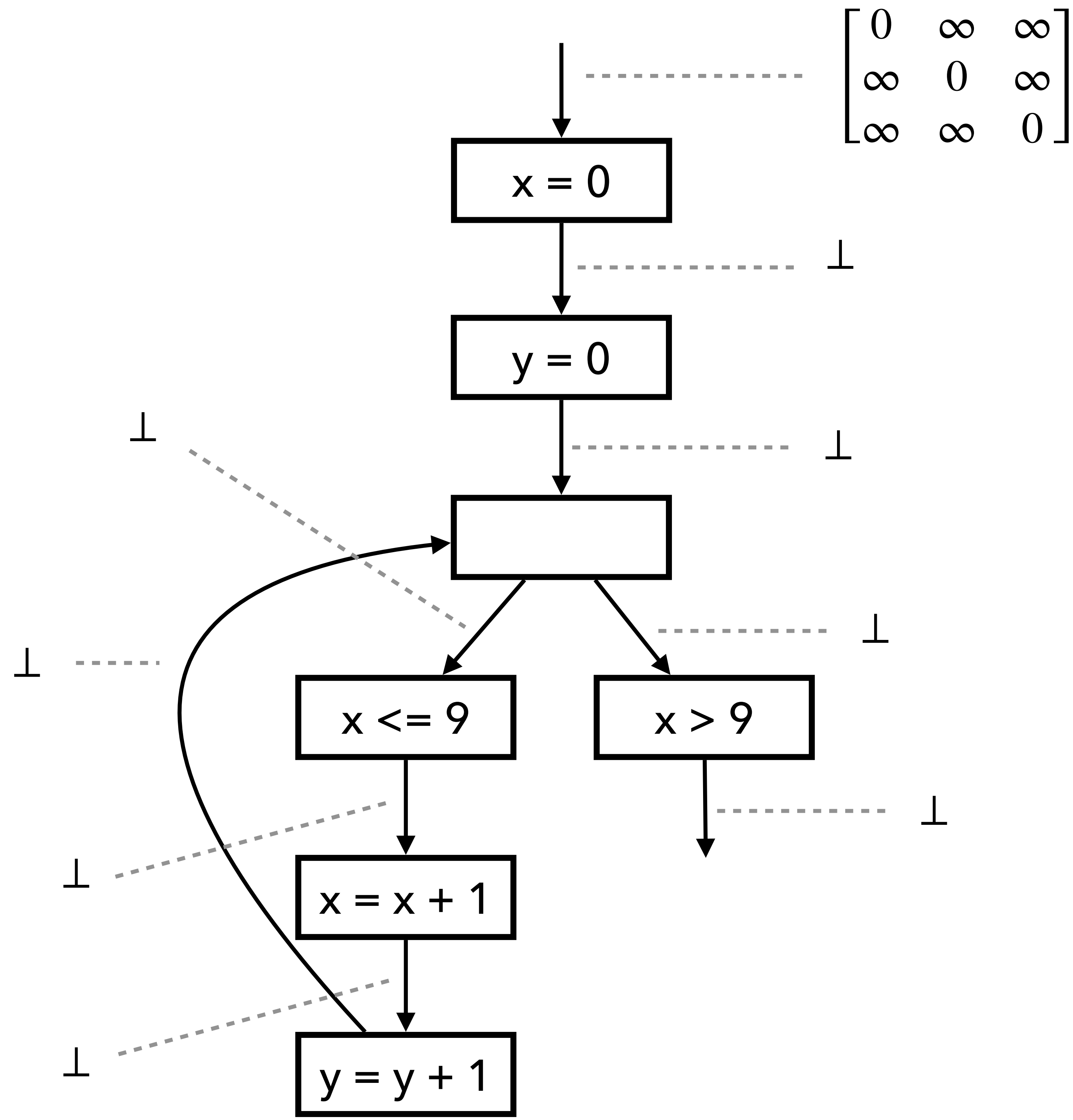
- Closure (normalization) via the Floyd-Warshall algorithm

$$\begin{bmatrix} +\infty & 4 & 3 \\ -1 & +\infty & +\infty \\ -1 & 1 & +\infty \end{bmatrix}^* = \begin{bmatrix} 0 & 4 & 3 \\ -1 & 0 & 2 \\ -1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 5 & 3 \\ -1 & +\infty & +\infty \\ -1 & 1 & +\infty \end{bmatrix}^* = \begin{bmatrix} 0 & 4 & 3 \\ -1 & 0 & 2 \\ -1 & 1 & 0 \end{bmatrix}$$



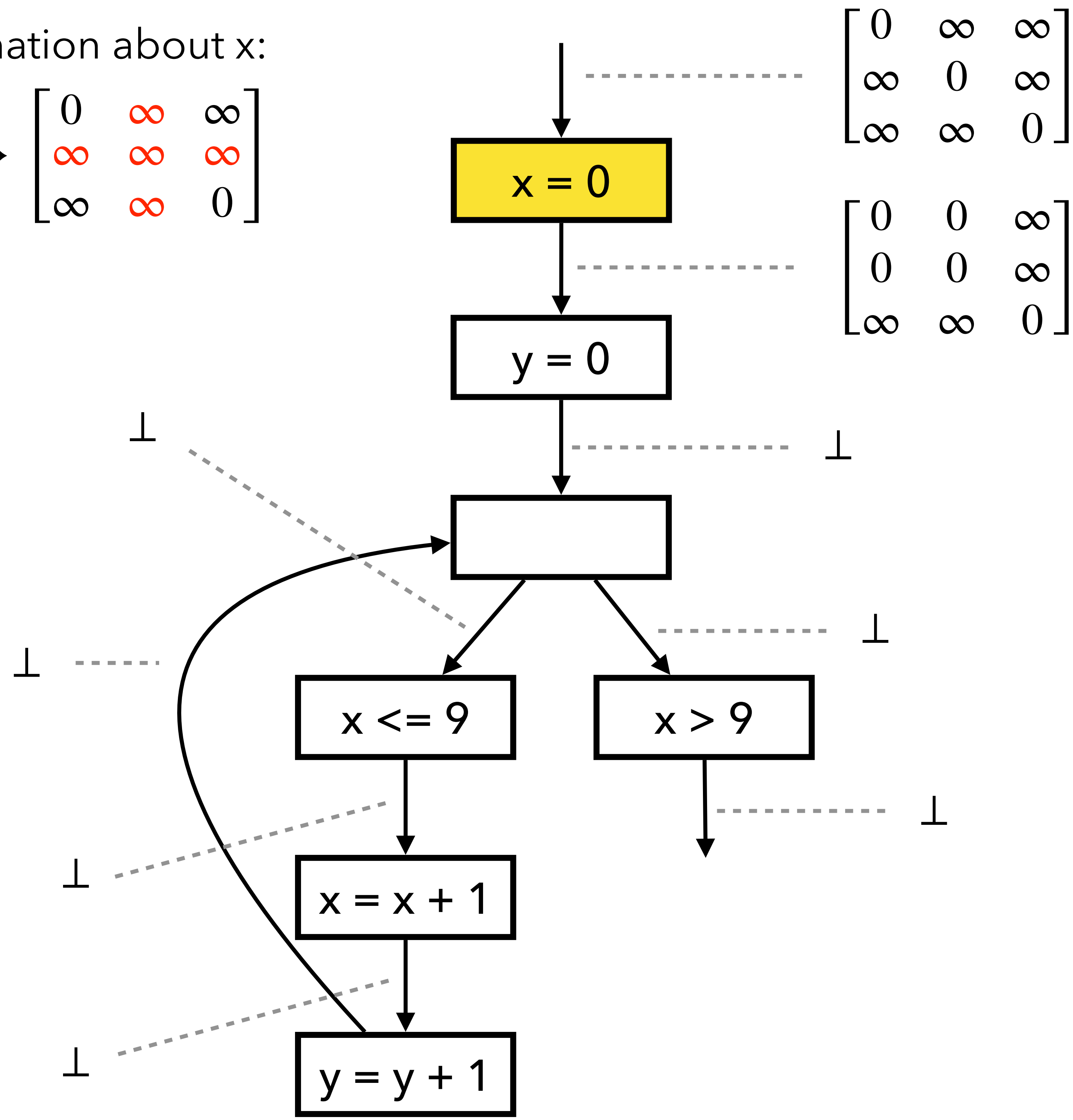
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Remove information about x:

$$\begin{bmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix}$$

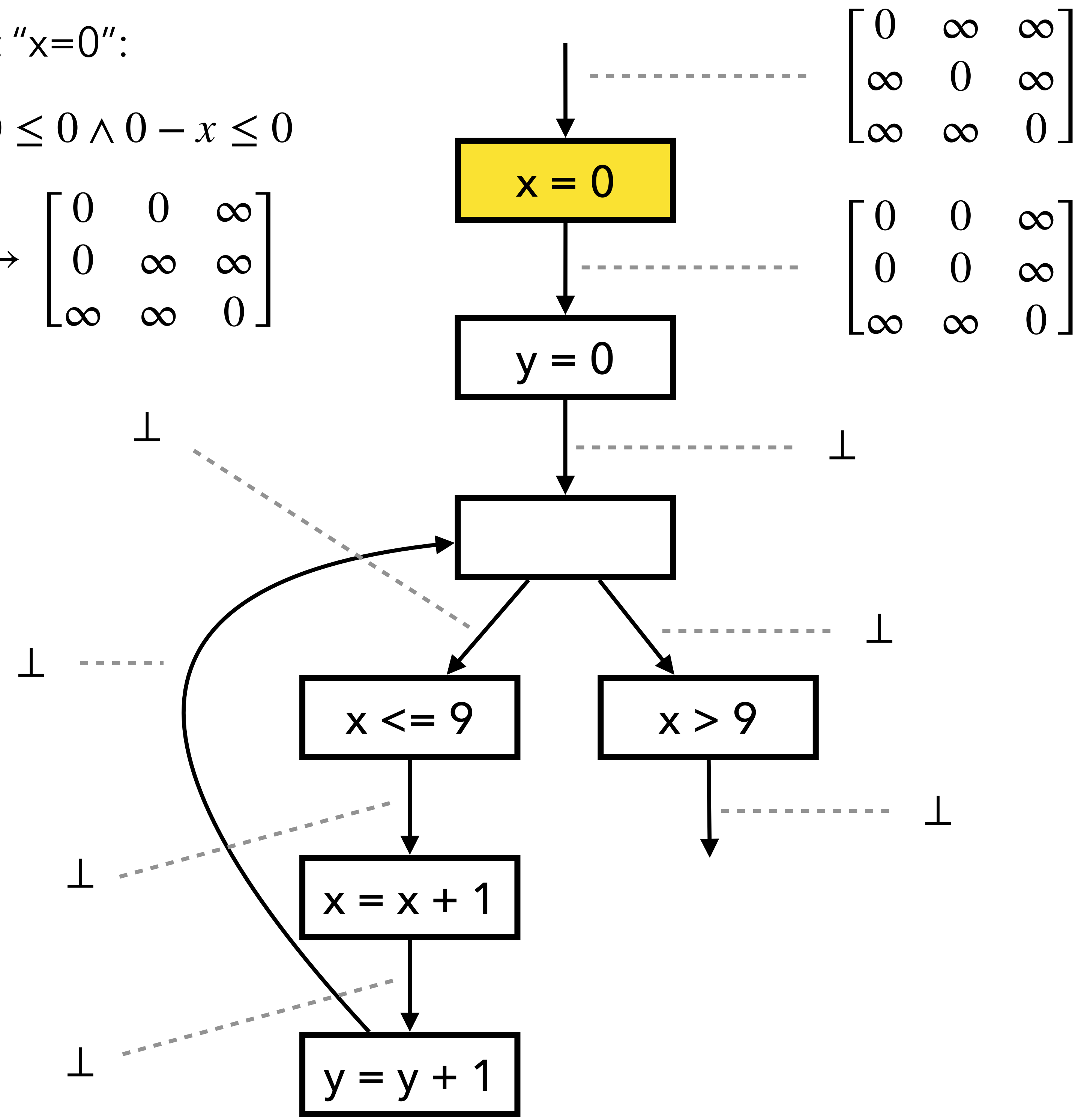


Fixed Point Comp. with Widening

2. Add constraint "x=0":

$$x = 0 \iff x - 0 \le 0 \land 0 - x \le 0$$

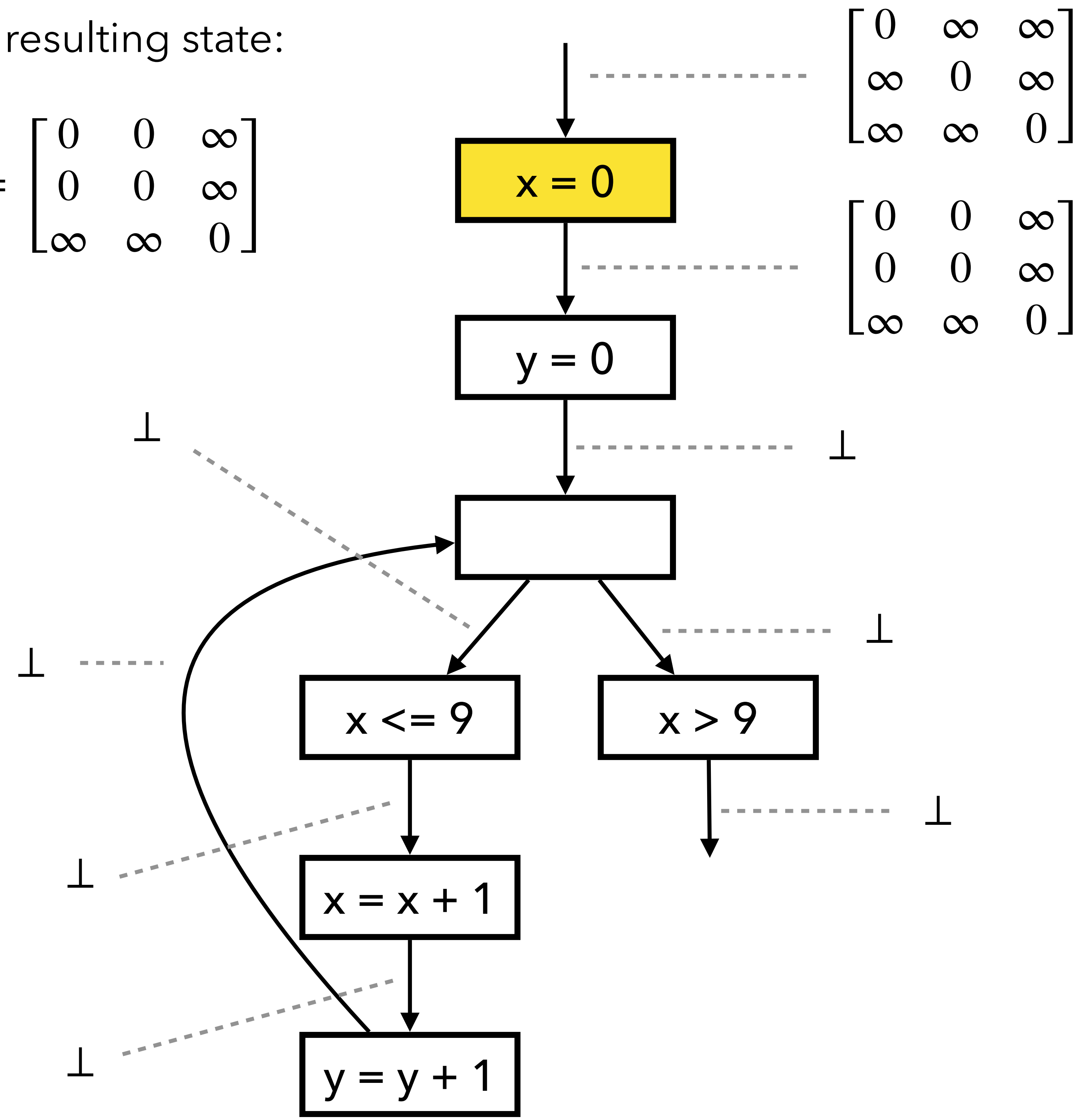
$$\begin{bmatrix} 0 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & \infty \\ 0 & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

3. Normalize the resulting state:

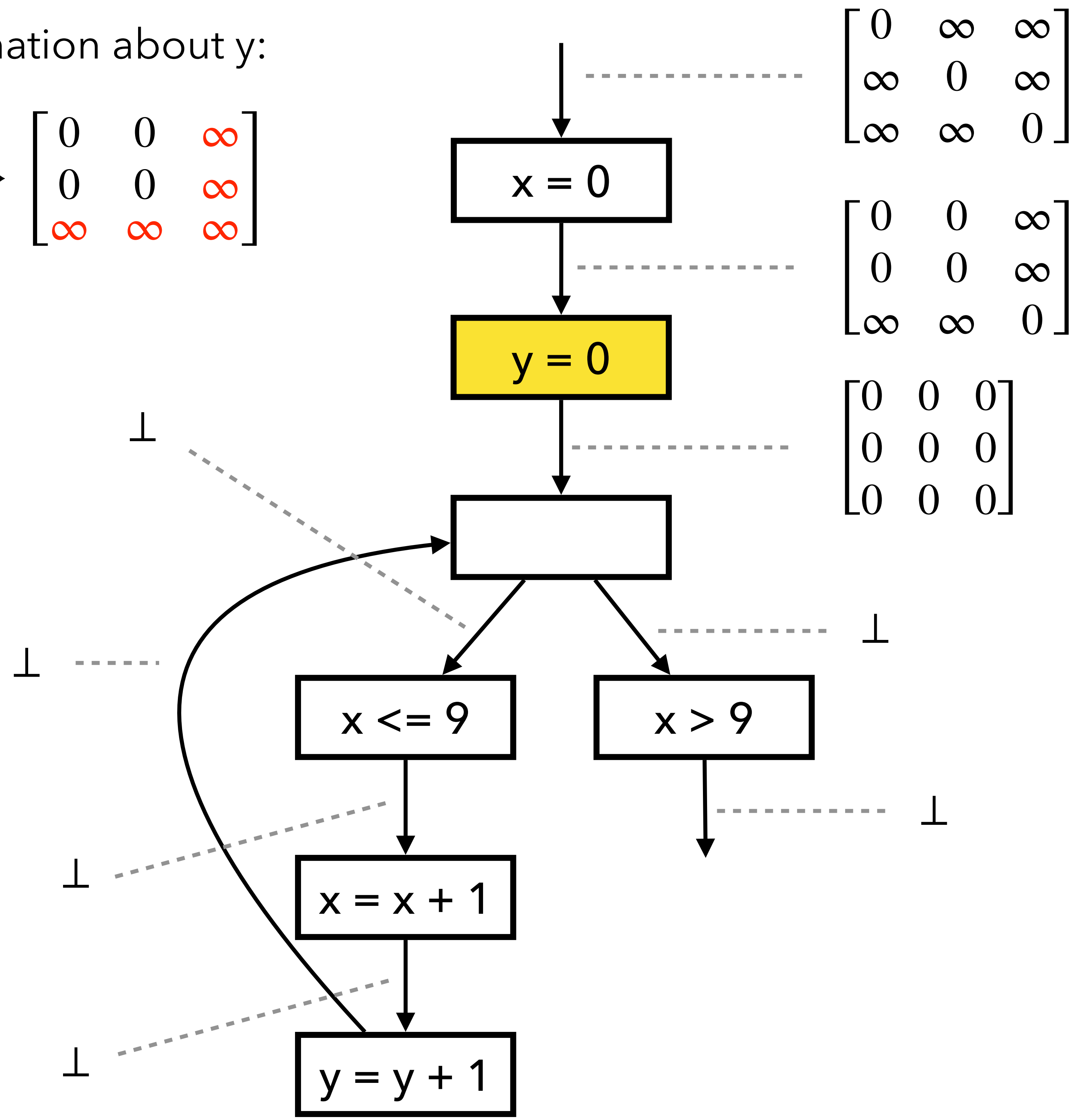
$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix}^* = \begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

1. Remove information about y:

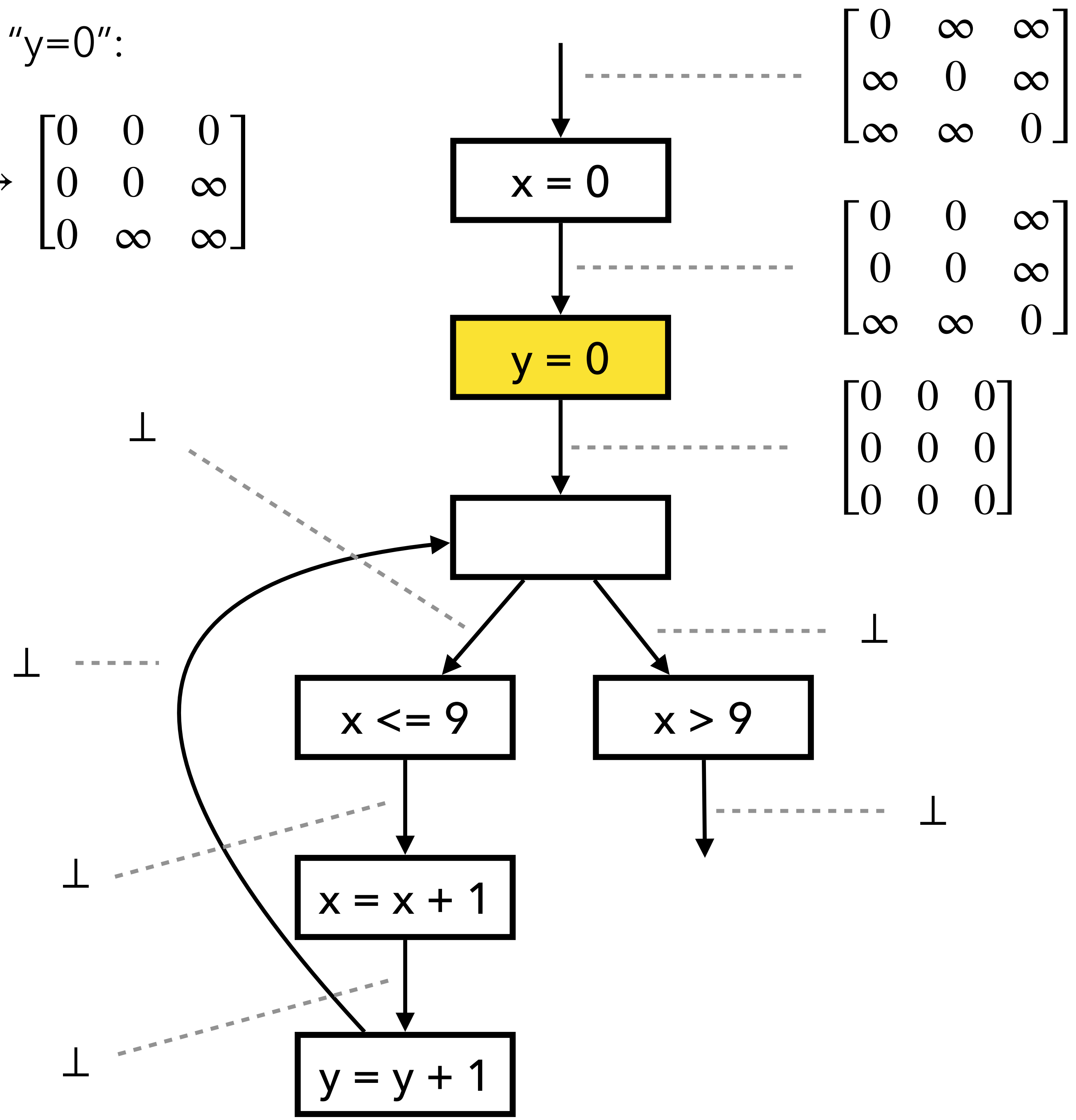
$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & \infty \end{bmatrix}$$



Fixed Point Comp. with Widening

2. Add constraint "y=0":

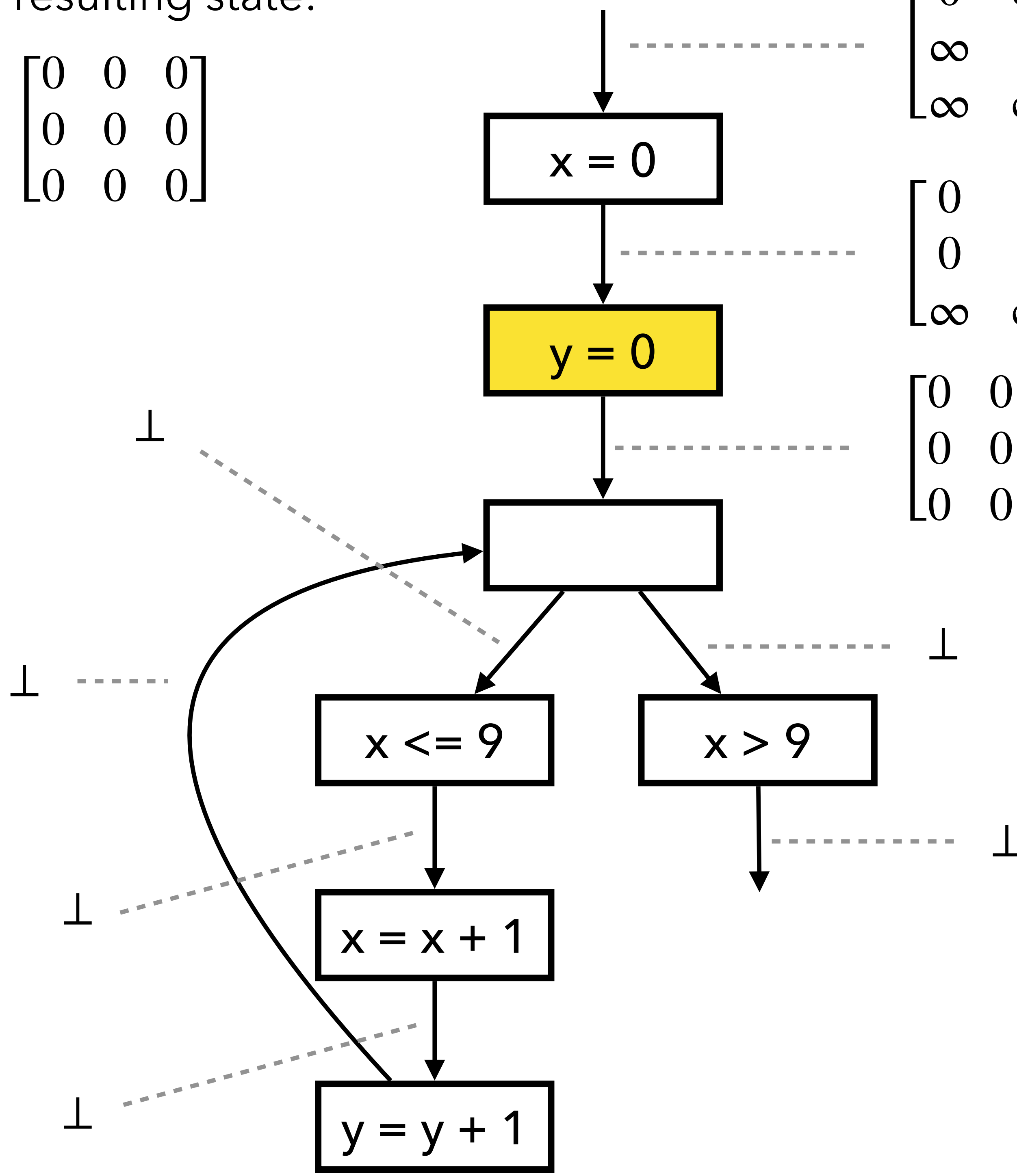
$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & \infty \\ 0 & \infty & \infty \end{bmatrix}$$



Fixed Point Comp. with Widening

3. Normalize the resulting state:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & \infty \\ 0 & \infty & \infty \end{bmatrix}^* = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

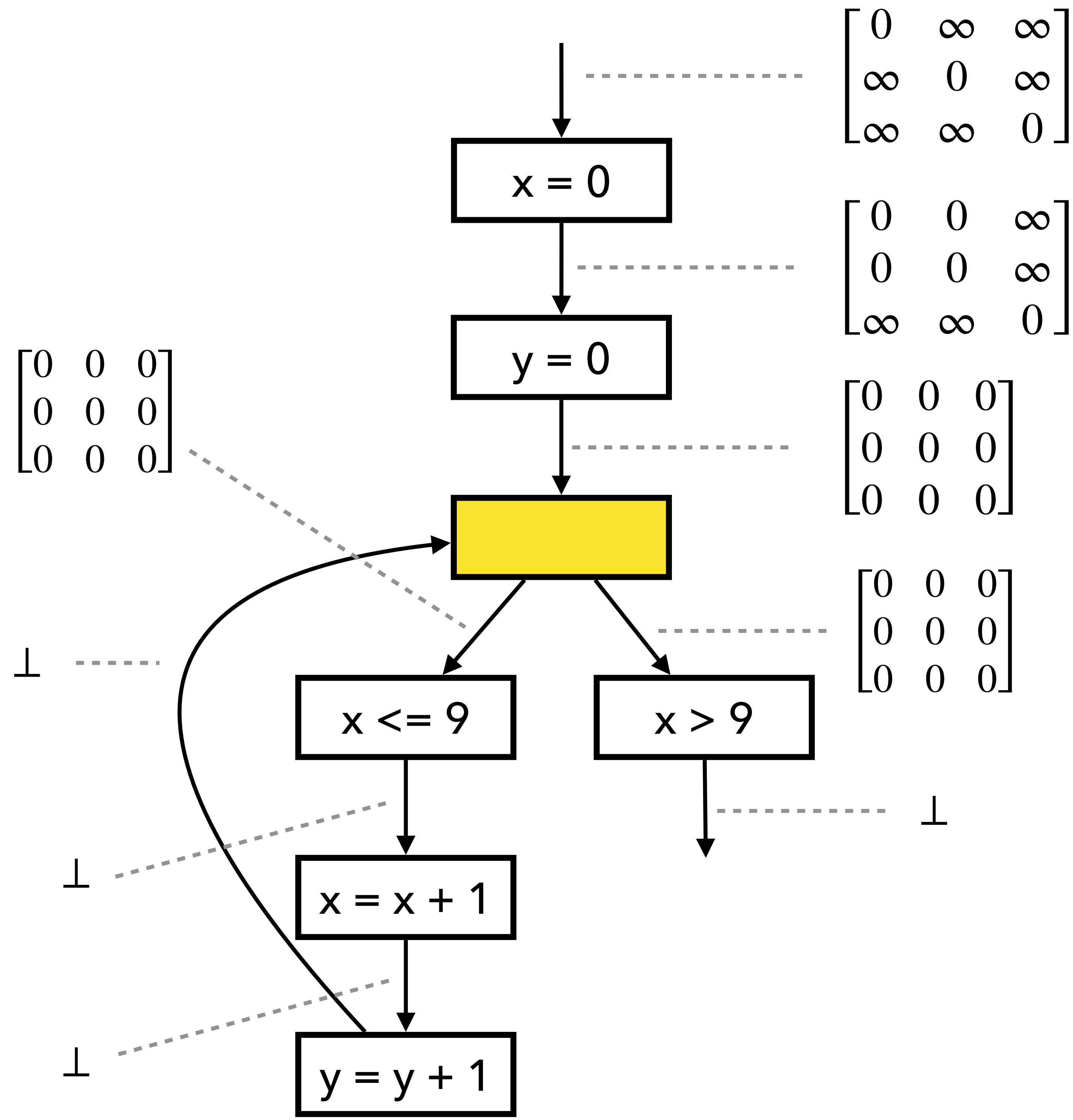


$$\begin{bmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix}$$

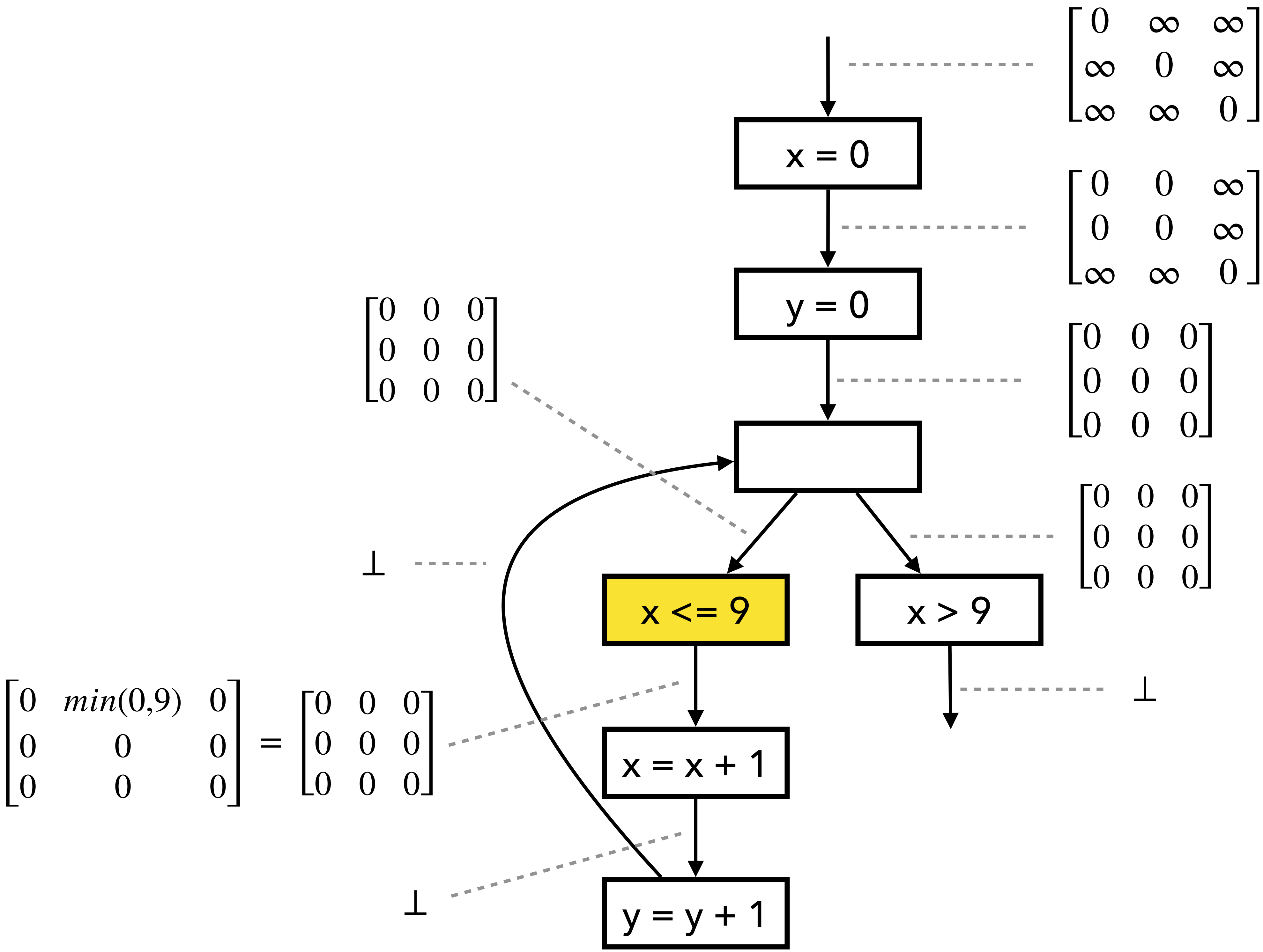
$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

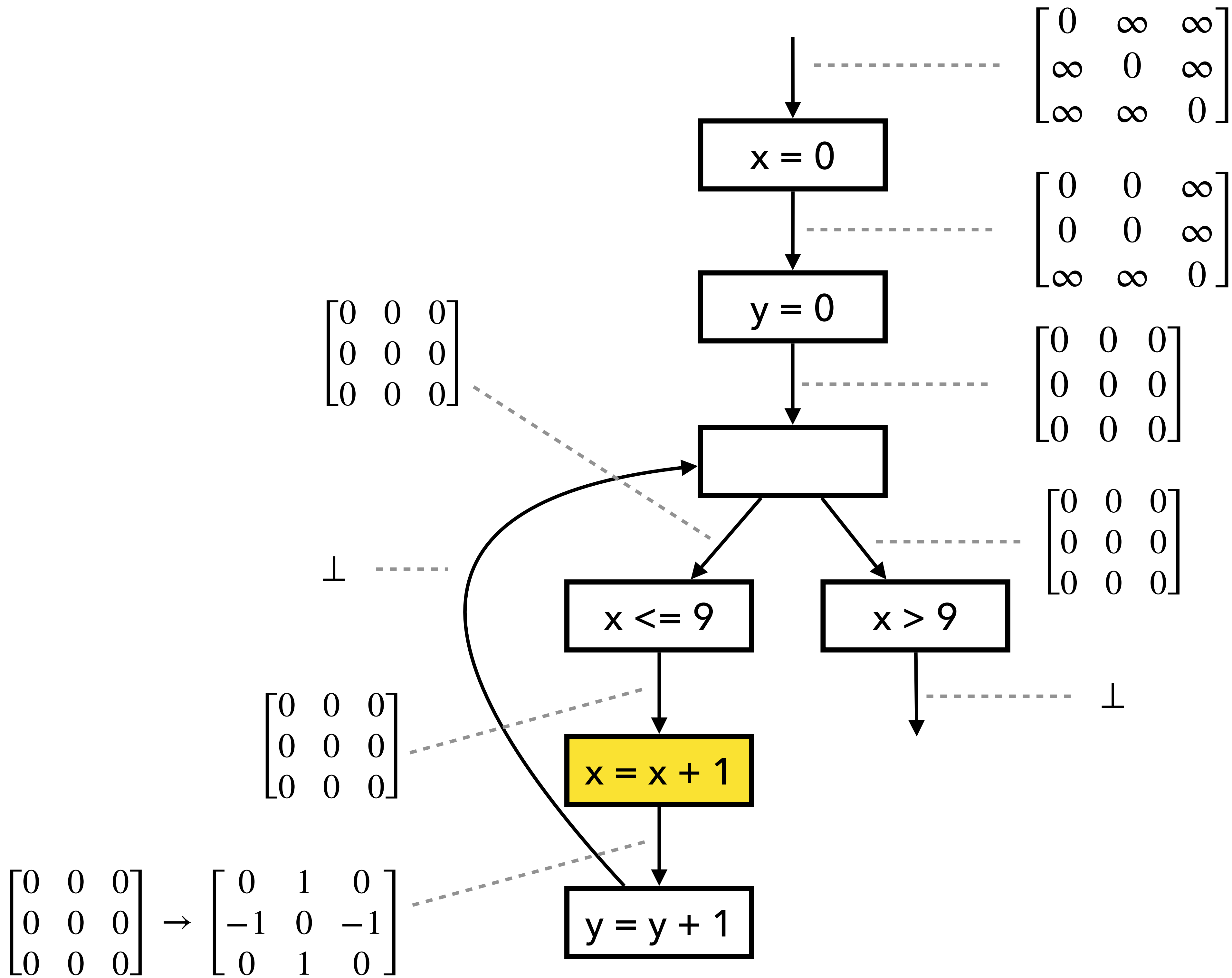
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening



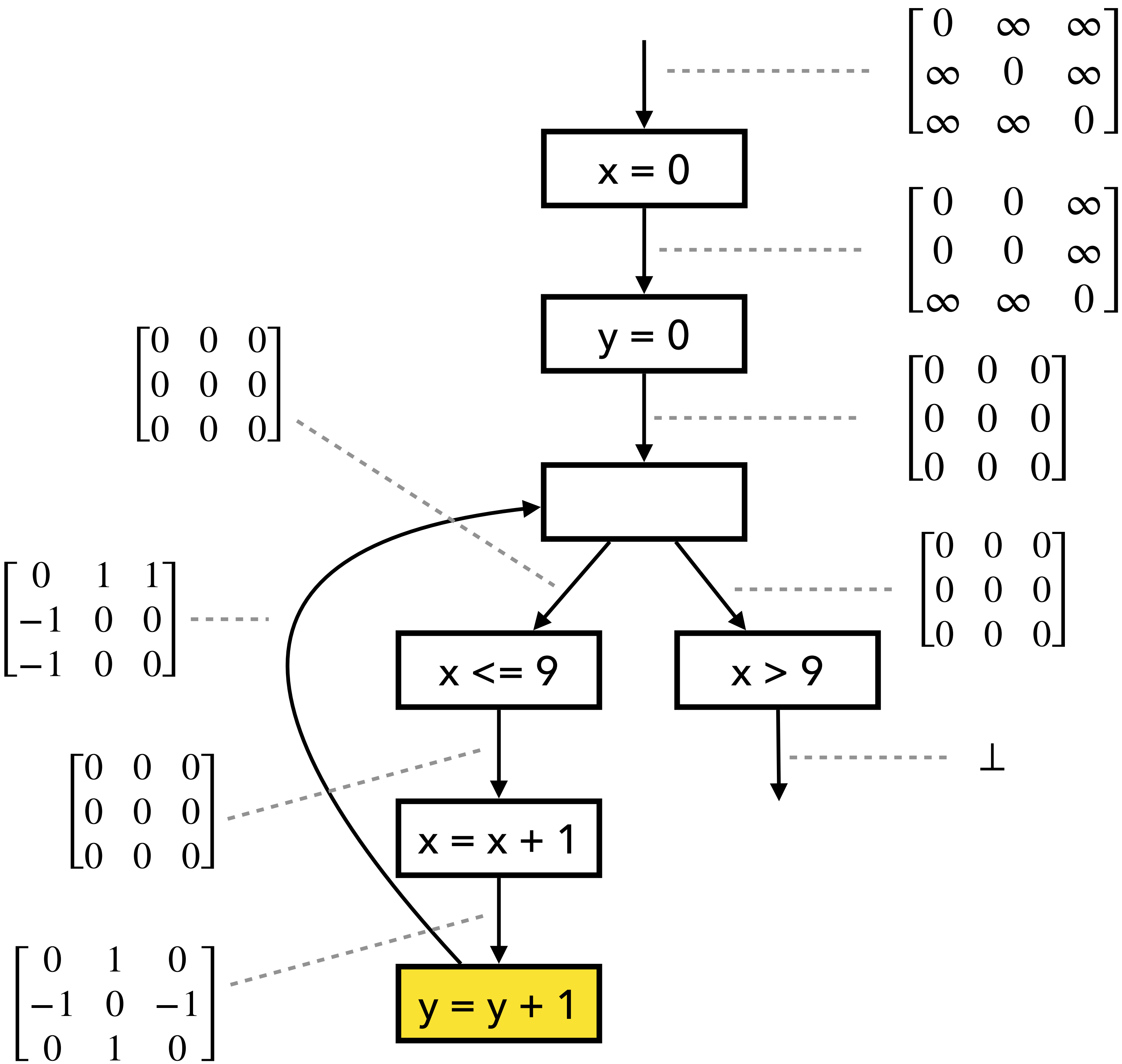
Fixed Point Comp. with Widening



$$x - x' \le c \rightarrow x - x' \le c + 1$$

$$x' - x \le c \rightarrow x' - x \le c - 1$$

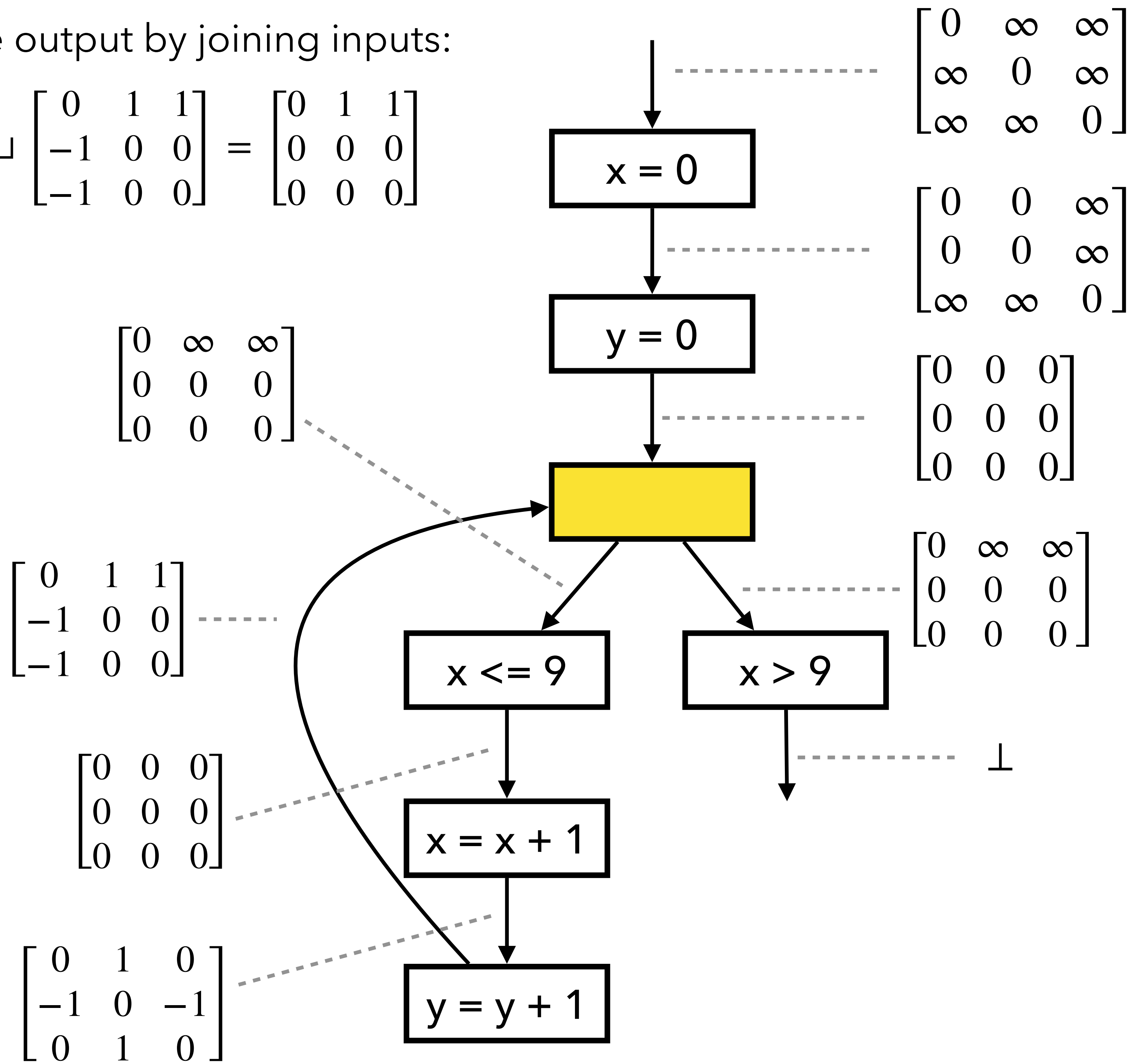
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

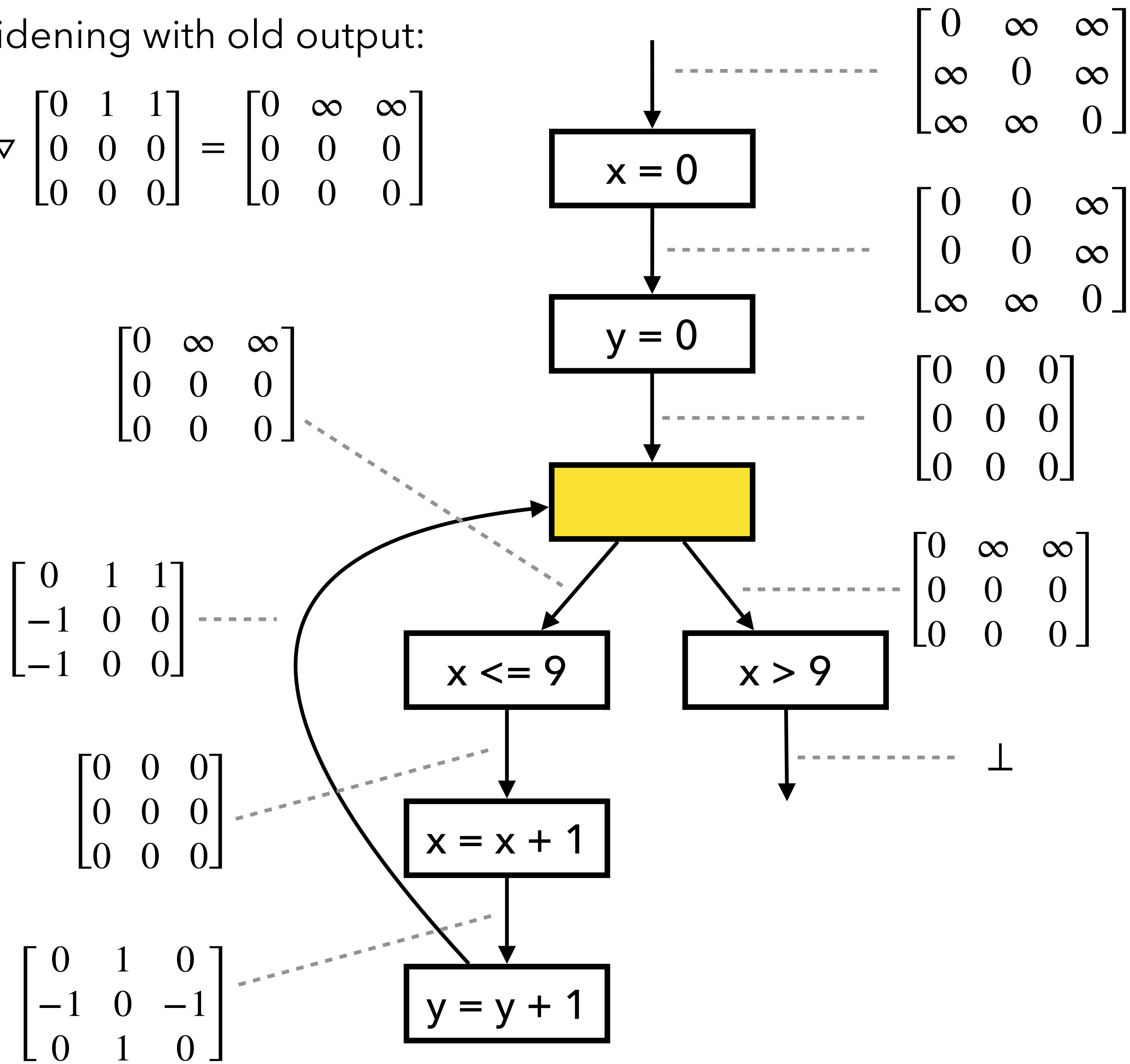
$$\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} \sqcup \begin{bmatrix}0 & 1 & 1\\-1 & 0 & 0\\-1 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & 1 & 1\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

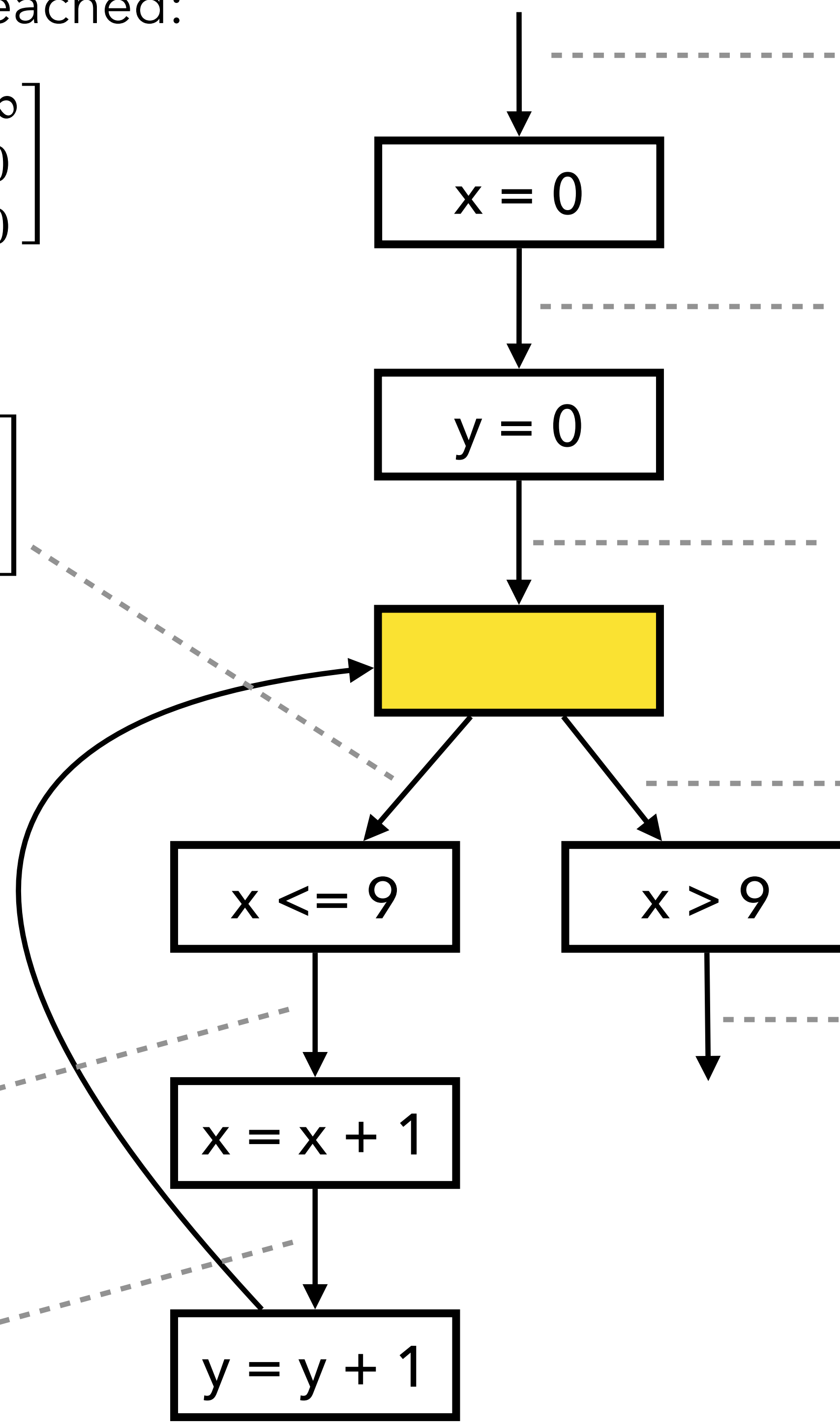
$$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix} \triangledown \begin{bmatrix}0&1&1\\0&0&0\\0&0&0\end{bmatrix} = \begin{bmatrix}0&\infty&\infty\\0&0&0\\0&0&0\end{bmatrix}$$



Fixed Point Comp. with Widening

3. Check if fixed point is reached:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \not\sqsubseteq \begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

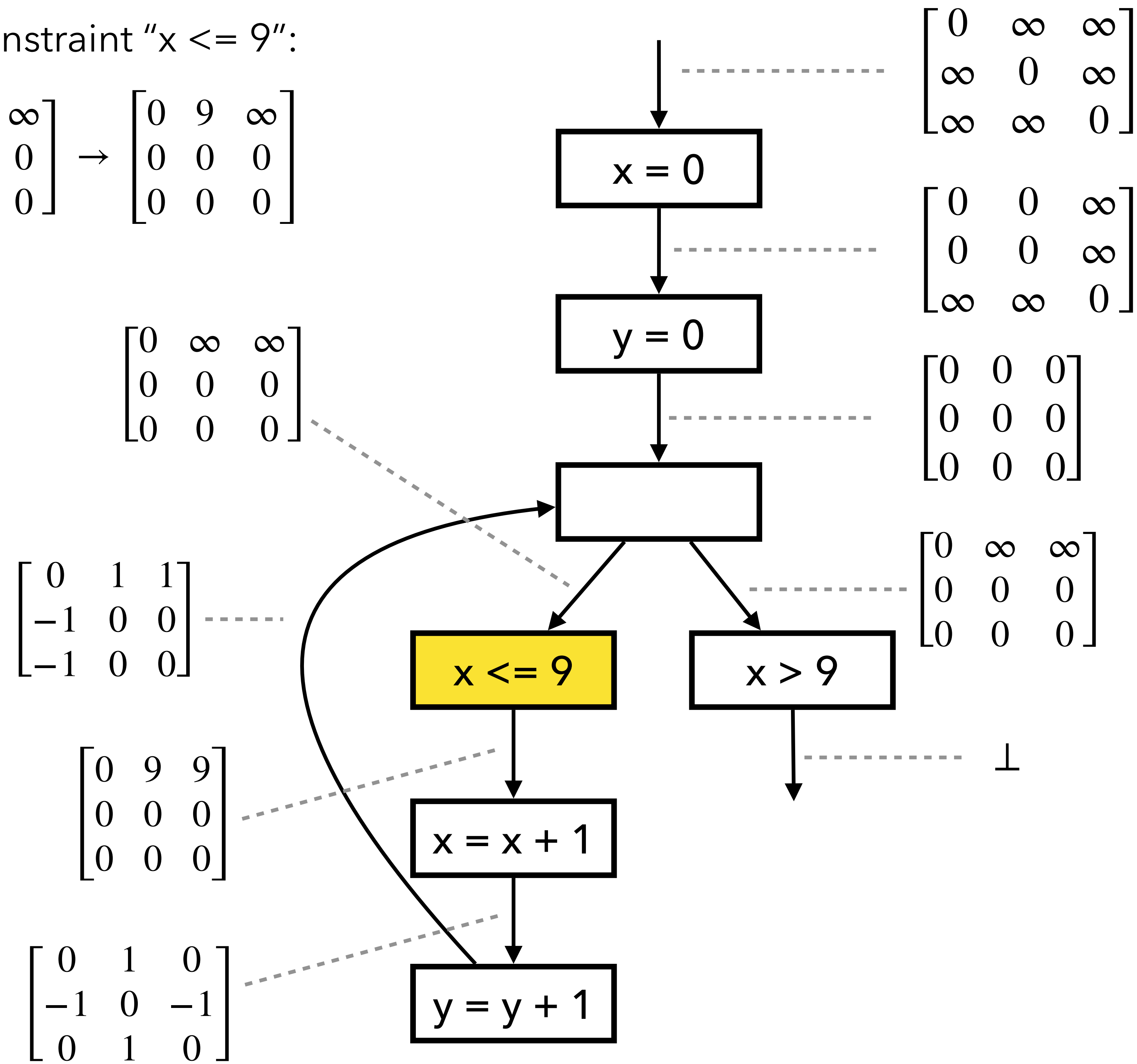
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

Fixed Point Comp. with Widening

1. Add constraint "x <= 9":

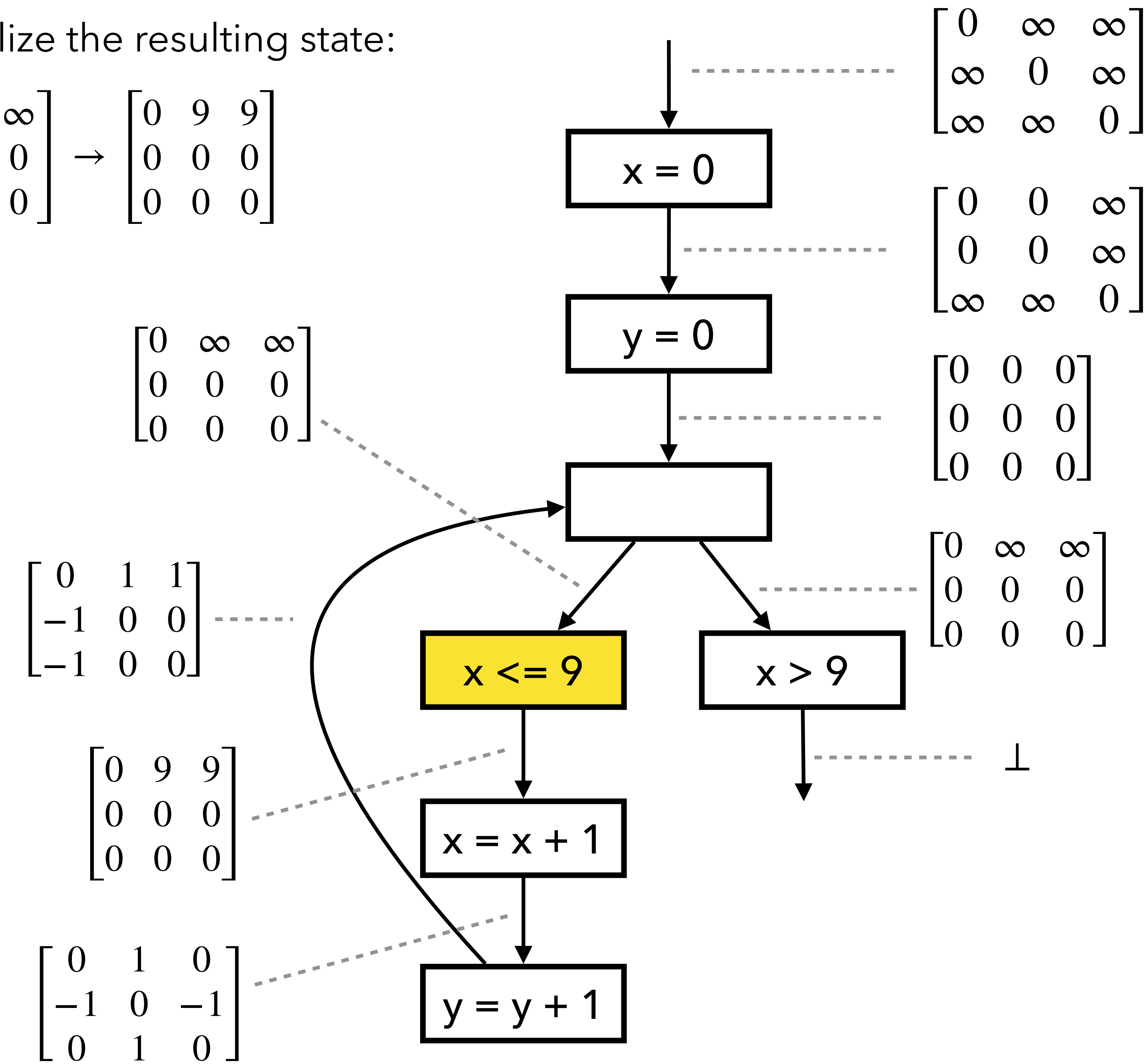
$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 9 & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



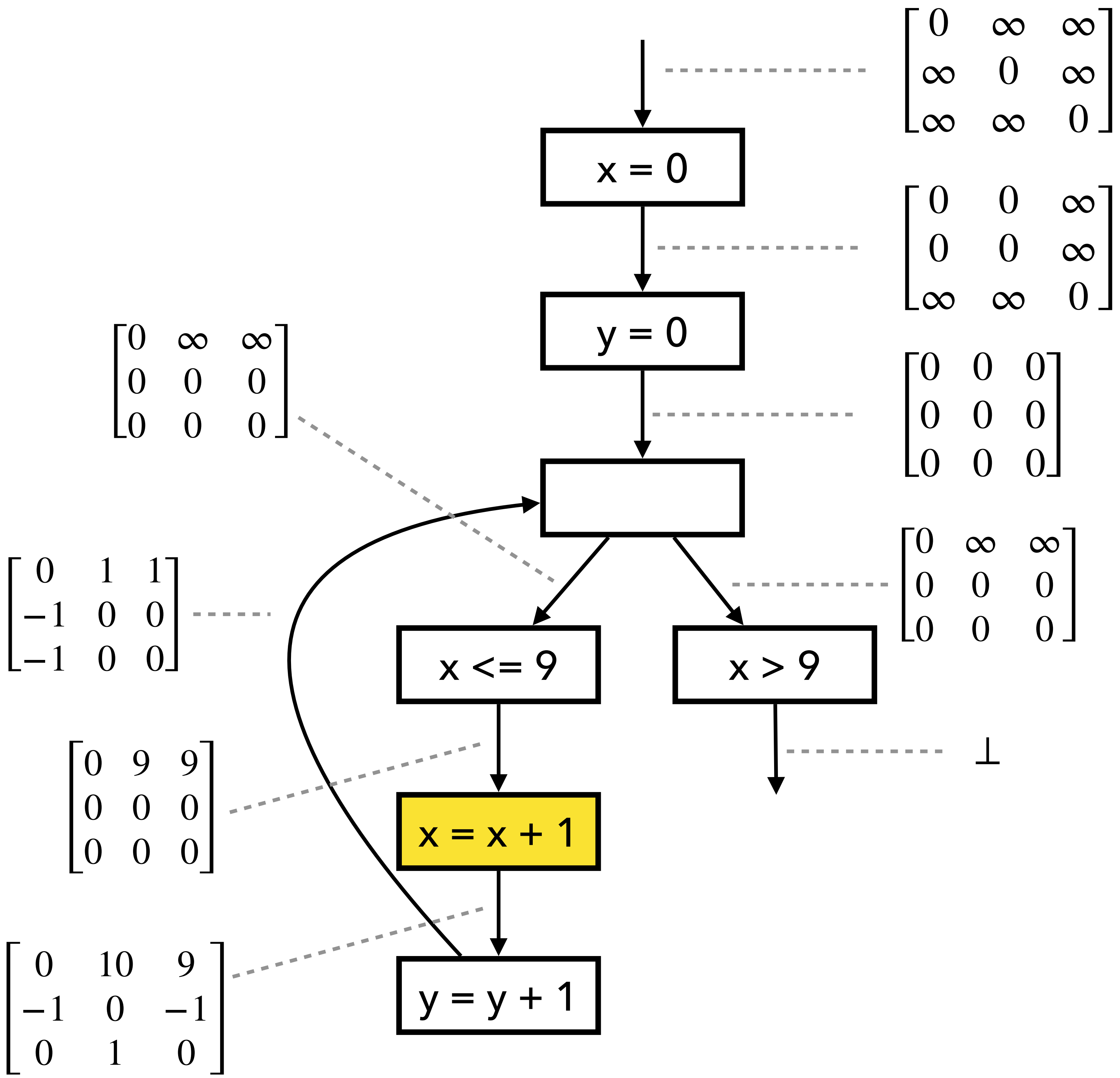
Fixed Point Comp. with Widening

2. Normalize the resulting state:

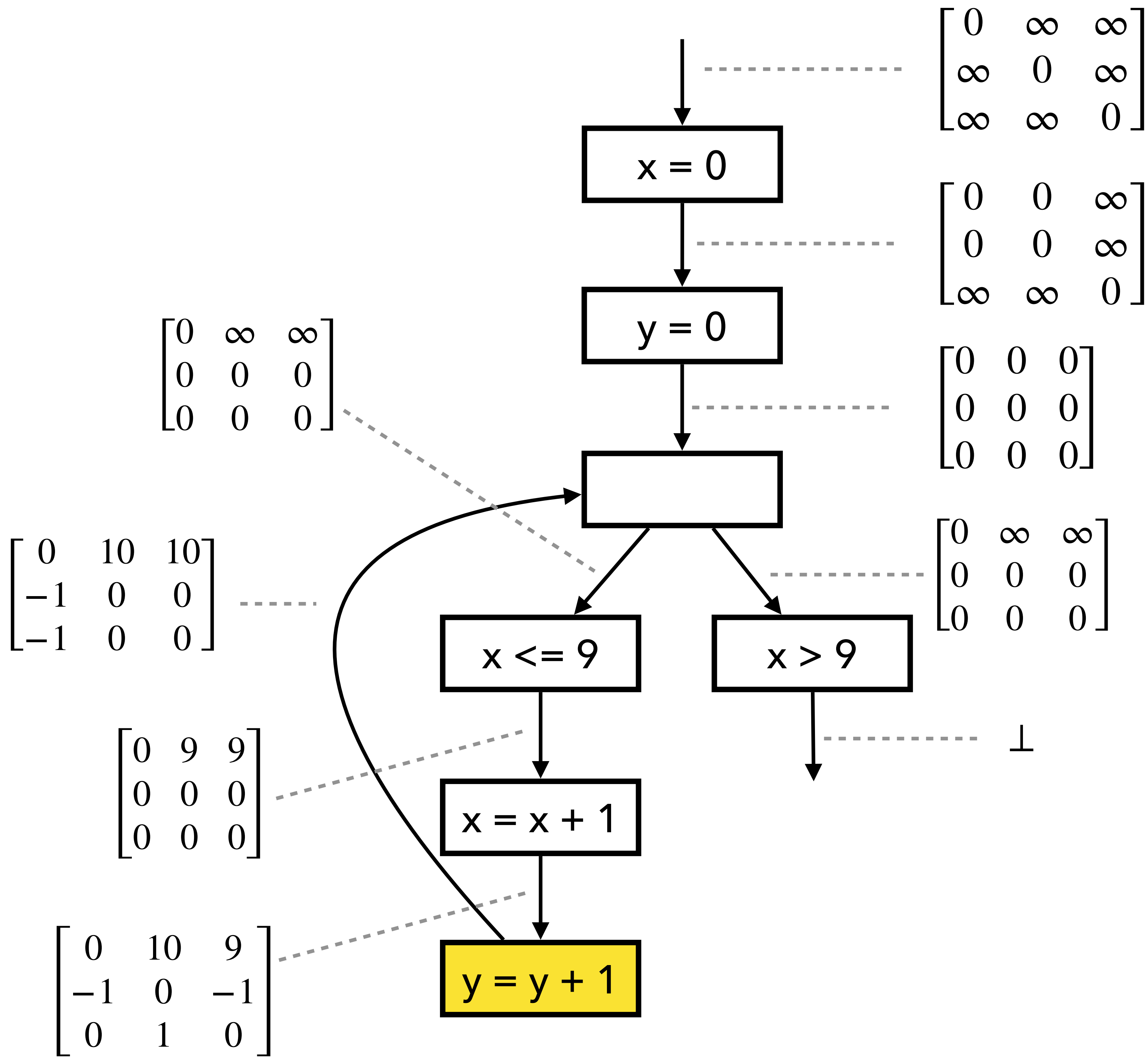
$$\begin{bmatrix} 0 & 9 & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 9 & 9 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening



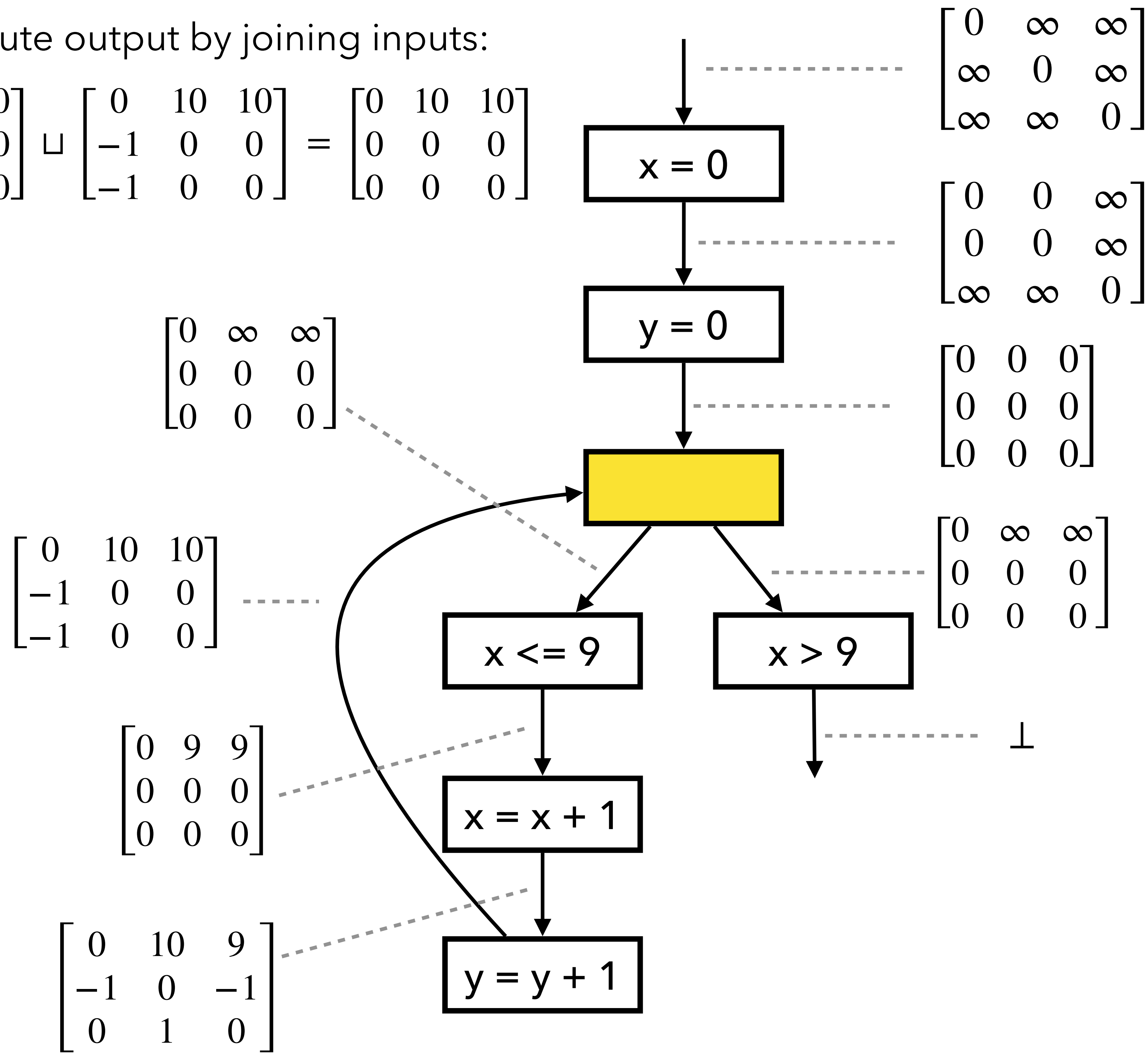
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

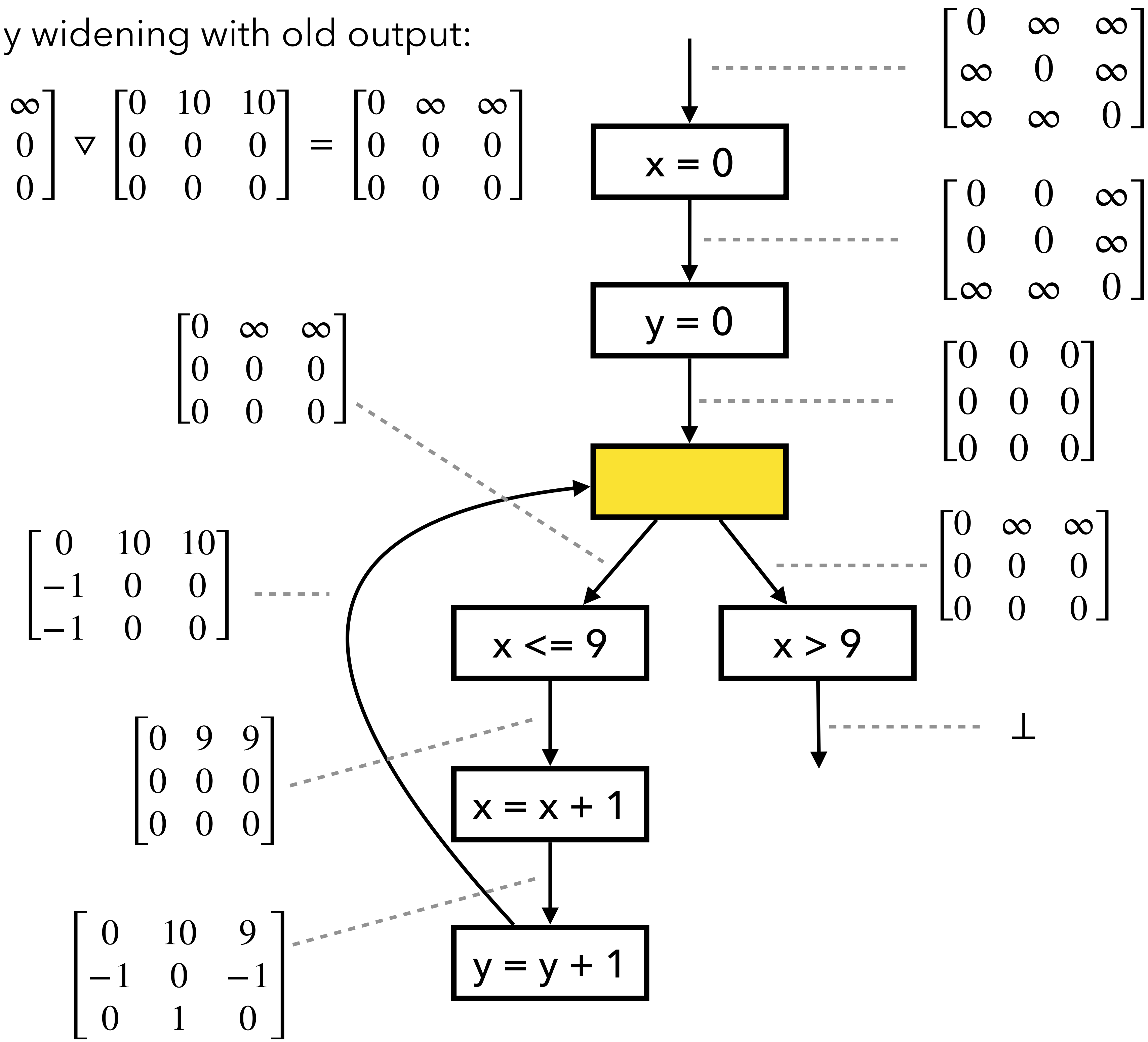
$$\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} \sqcup \begin{bmatrix}0 & 10 & 10\\-1 & 0 & 0\\-1 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & 10 & 10\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

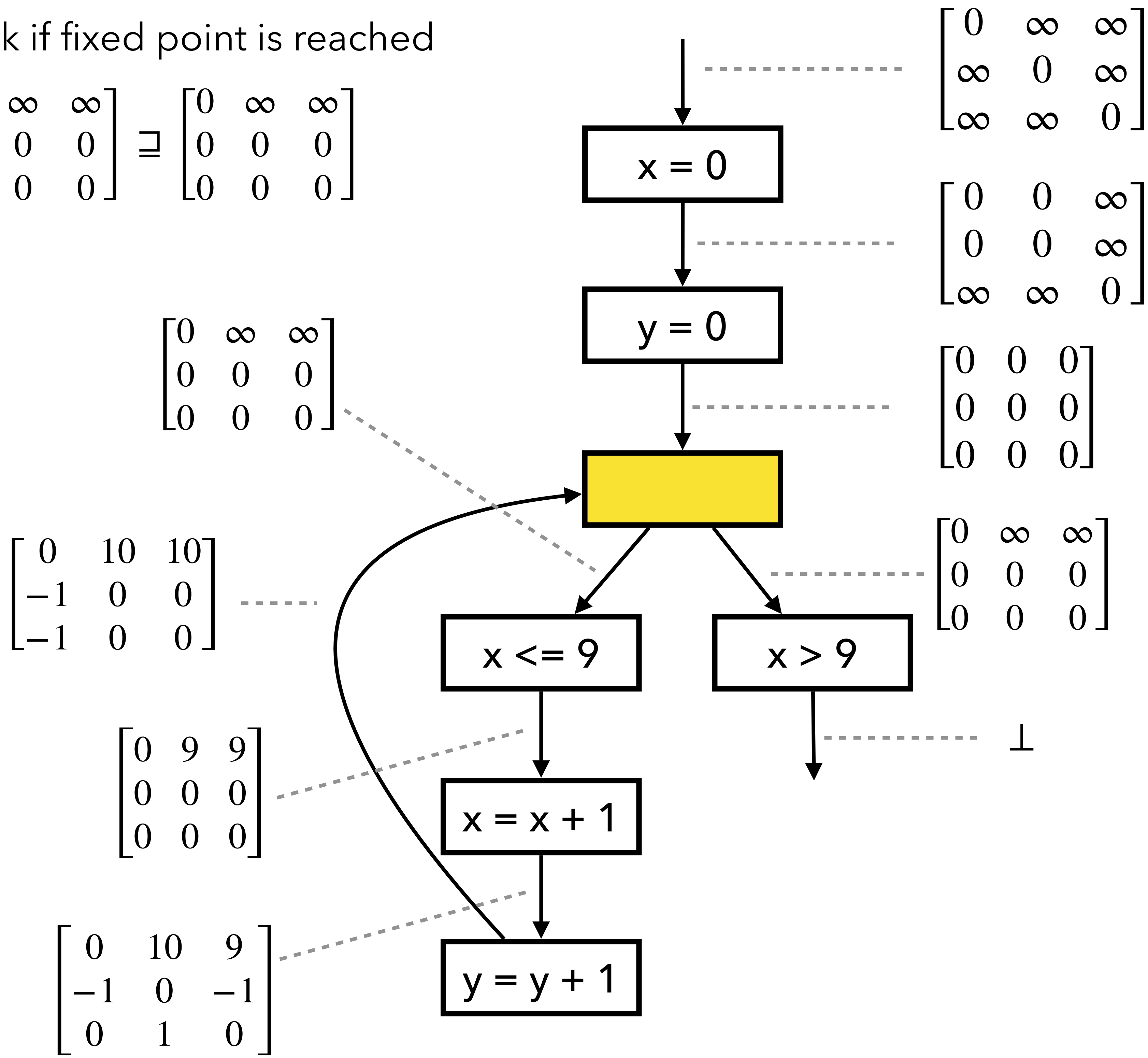
$$\begin{bmatrix}0 & \infty & \infty\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} \triangledown \begin{bmatrix}0 & 10 & 10\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & \infty & \infty\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$



Fixed Point Comp. with Widening

3. Check if fixed point is reached

$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \sqsupseteq \begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

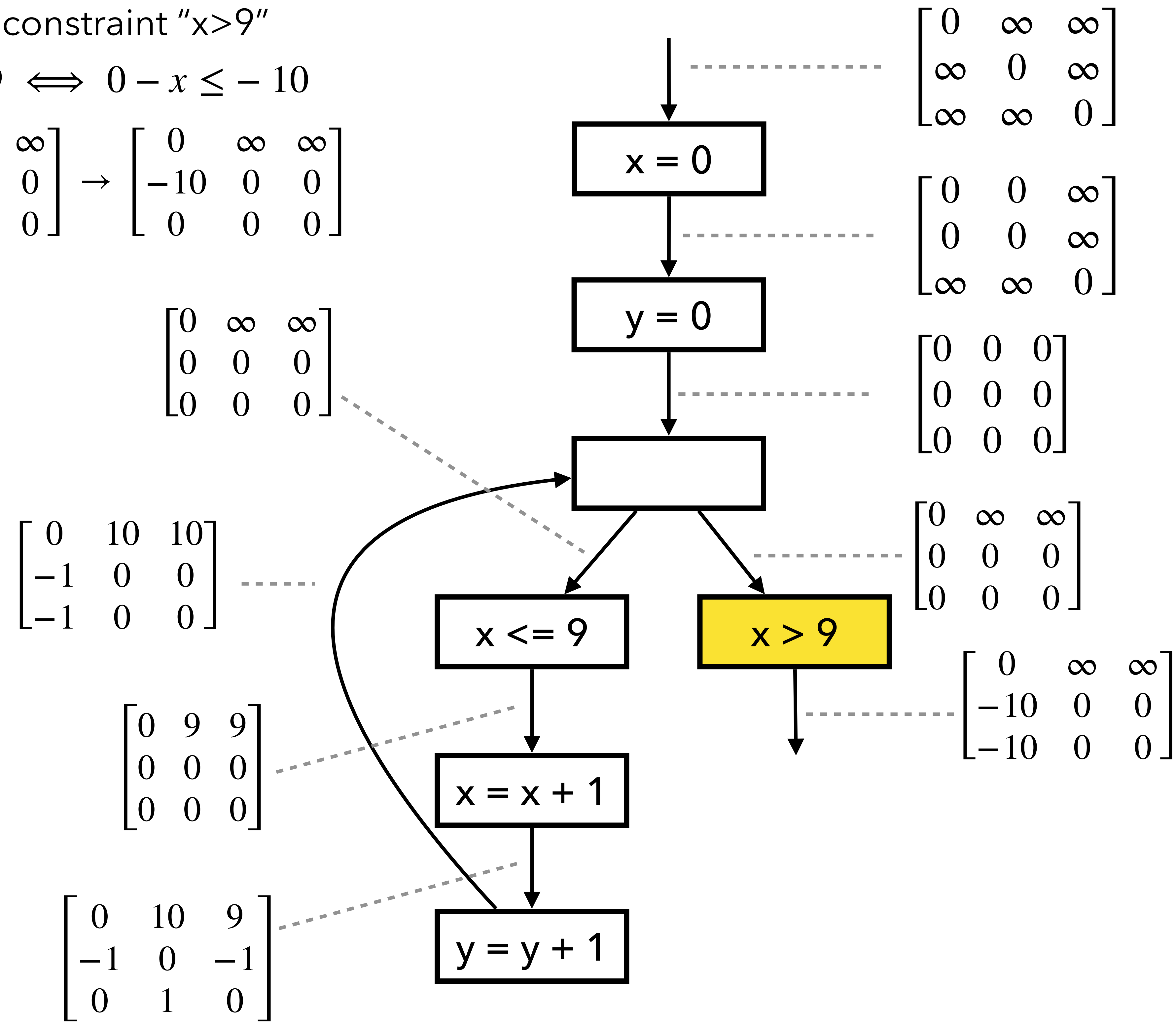


Fixed Point Comp. with Widening

1. Add constraint "x>9"

$$x > 9 \iff 0 - x \le -10$$

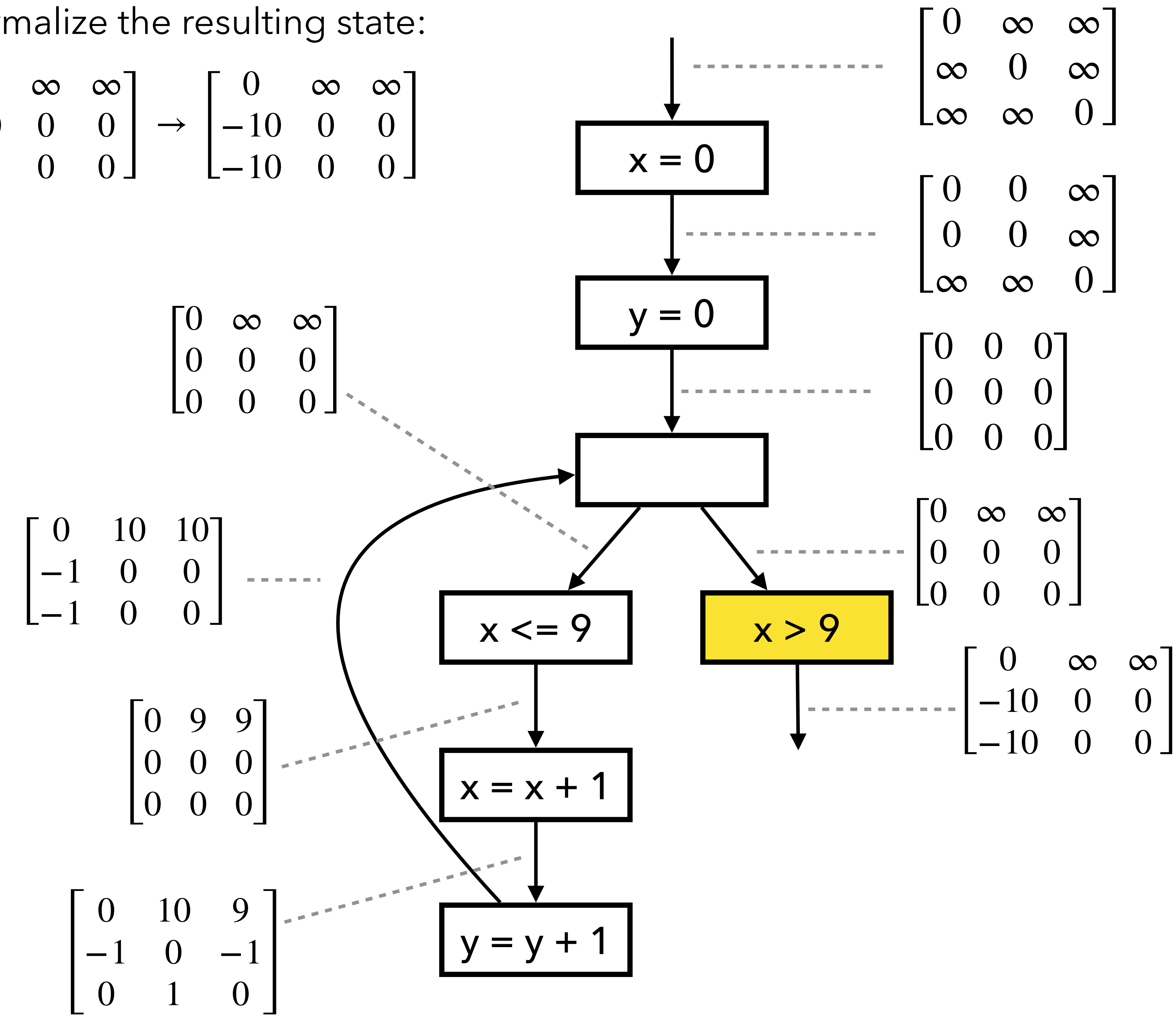
$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

2. Normalize the resulting state:

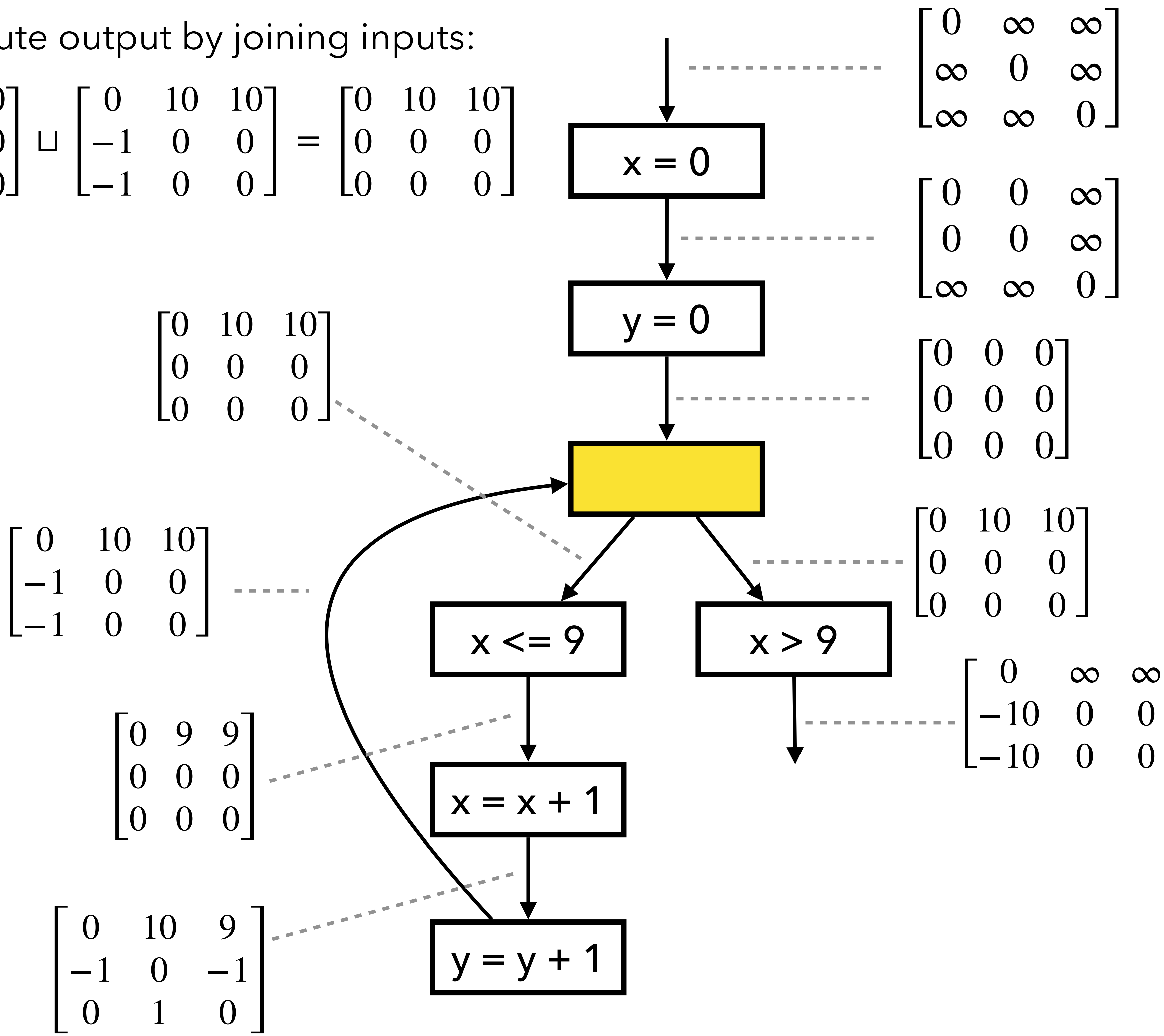
$$\begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ -10 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Narrowing

1. Compute output by joining inputs:

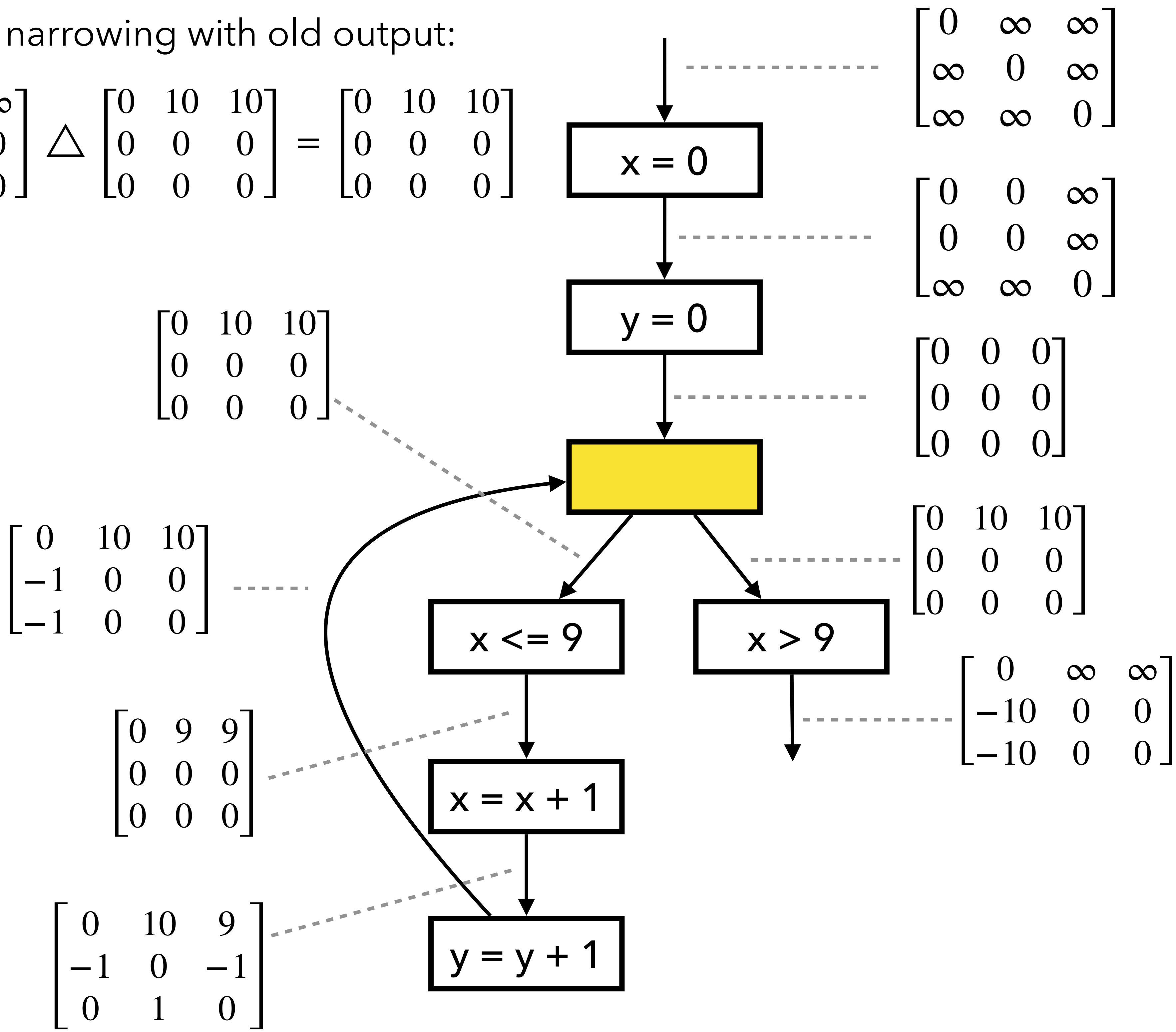
$$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix} \sqcup \begin{bmatrix}0&10&10\\-1&0&0\\-1&0&0\end{bmatrix} = \begin{bmatrix}0&10&10\\0&0&0\\0&0&0\end{bmatrix}$$



Fixed Point Comp. with Narrowing

2. Apply narrowing with old output:

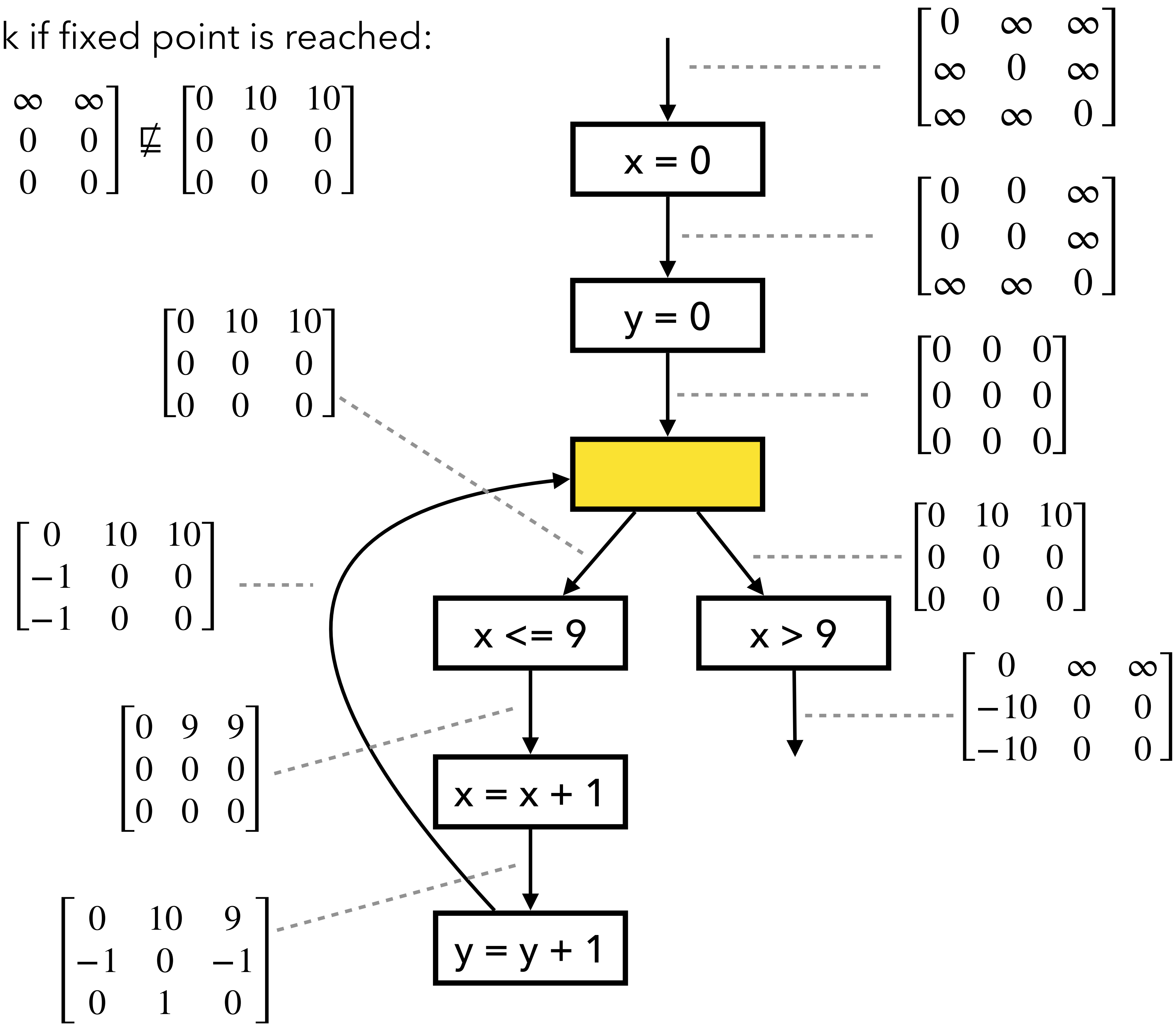
$$\begin{bmatrix}0 & \infty & \infty\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} \triangle \begin{bmatrix}0 & 10 & 10\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & 10 & 10\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$



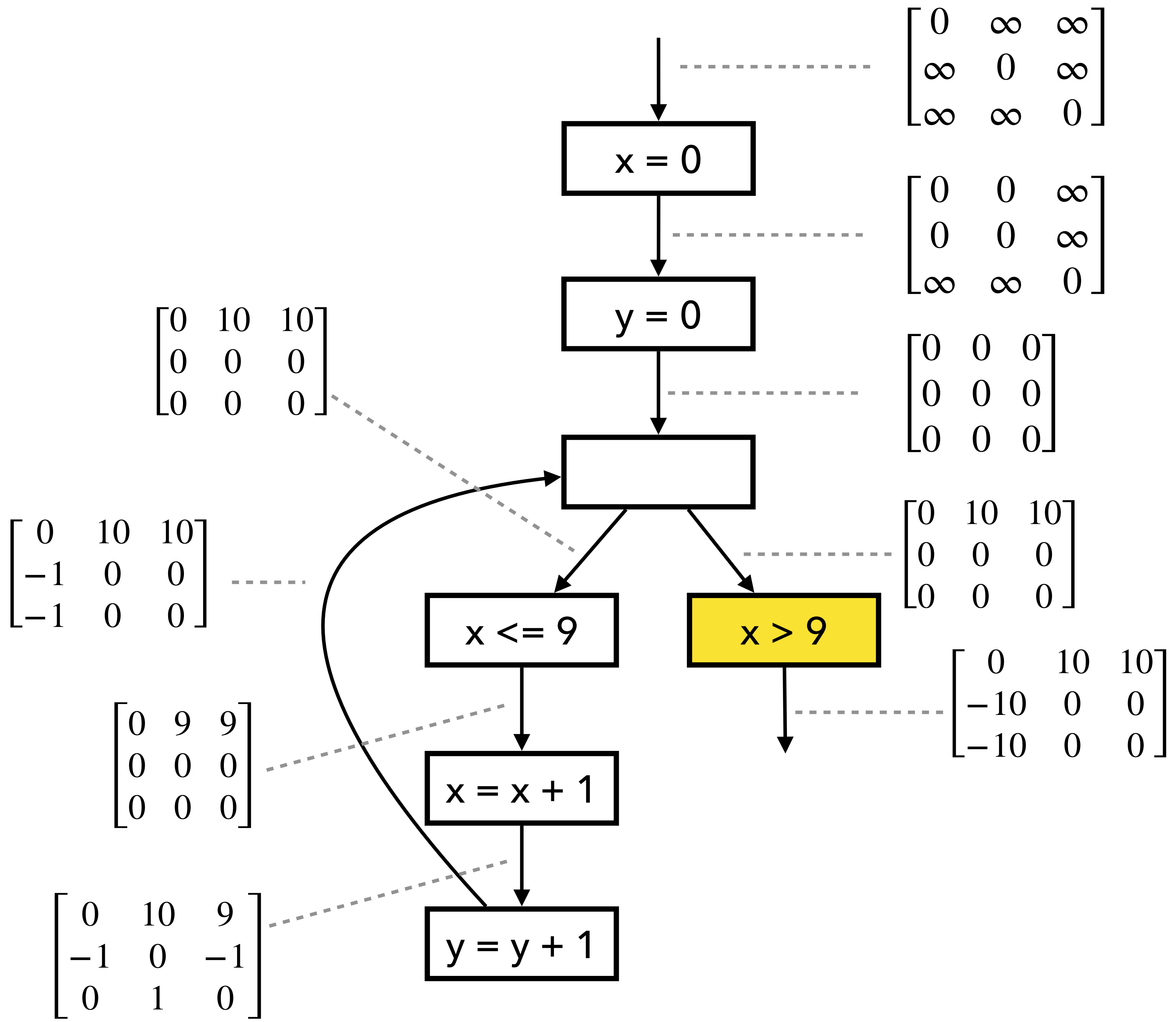
Fixed Point Comp. with Narrowing

3. Check if fixed point is reached:

$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \not\sqsubseteq \begin{bmatrix} 0 & 10 & 10 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



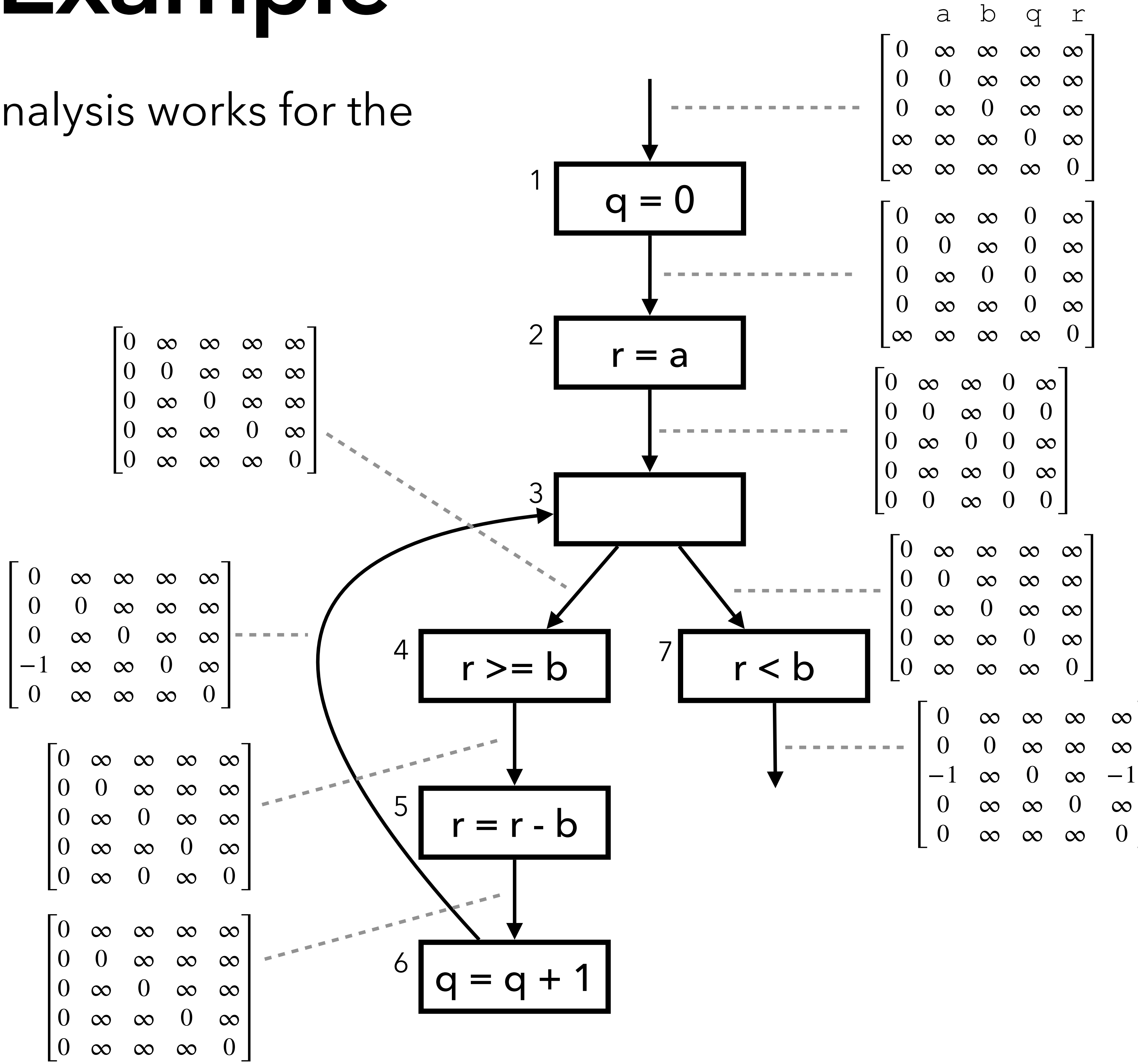
Fixed Point Comp. with Narrowing



Motivating Example

Describe how the zone analysis works for the following example.

```
// a >= 0, b >= 0
q = 0;
r = a;
while (r >= b) {
  r = r - b;
  q = q + 1;
}
assert(q >= 0);
assert(r >= 0);
```



Static Analysis Use Cases: Infer

- Install (https://github.com/facebook/infer/)

```
# Checkout Infer
git clone https://github.com/facebook/infer.git
cd infer
# Compile Infer
./build-infer.sh java
# install Infer system-wide...
sudo make install
# ...or, alternatively, install Infer into your PATH
export PATH=`pwd`/infer/bin:$PATH
```

- Running Infer: e.g.,
  - `infer capture -- make`
  - `infer analyze`

Infer's Intermediate Language

https://github.com/facebook/infer/blob/main/infer/src/IR/Sil.mli

```
type instr =
  | Load of {id: Ident.t; e: Exp.t; typ: Typ.t; loc: Location.t}
      (** Load a value from the heap into an identifier.

          [id = *e:typ] where

          - [e] is an expression denoting a heap address
          - [typ] is the type of [*e] and [id]. *)
  | Store of {e1: Exp.t; typ: Typ.t; e2: Exp.t; loc: Location.t}
      (** Store the value of an expression into the heap.

          [*e1:typ = e2] where

          - [e1] is an expression denoting a heap address
          - [typ] is the type of [*e1] and [e2]. *)
  | Prune of Exp.t * Location.t * bool * if_kind
      (** The semantics of [Prune (exp, loc, is_then_branch, if_kind)] is that it prunes the state
          (blocks, or diverges) if [exp] evaluates to [1]; the boolean [is_then_branch] is [true] if
          this is the [then] branch of an [if] condition, [false] otherwise (it is meaningless if
          [if_kind] is not [Ik_if], [Ik_bexp], or other [if]-like cases

          This instruction, together with the CFG structure, is used to encode control-flow with
          tests in the source program such as [if] branches and [while] loops. *)
  | Call of (Ident.t * Typ.t) * Exp.t * (Exp.t * Typ.t) list * Location.t * CallFlags.t
      (** [Call ((ret_id, ret_typ), e_fun, arg_ts, loc, call_flags)] represents an instruction
          [ret_id = e_fun(arg_ts)] *)
```

Example: Buffer Overflow Detection

```
static char *curfinal = "HDACB  FE";

keysym = read_from_input ();

if ((((KeySym)(keysym) >= 0xFF91) && ((KeySym)(keysym) <= 0xFF94)))
{
   unparseputc((char)(keysym-0xFF91 +'P'), pty);
   key = 1;
}
else if (keysym >= 0)
{
    if (keysym < 16)
    {
        if (read_from_input())
        {
           if (keysym >= 10) return;
           curfinal[keysym] = 1;
        }
        else
        {
            curfinal[keysym] = 2;
        }
    }
    if (keysym < 10)
    {
        unparseputc(curfinal[keysym], pty);
    }
 }
```

curfinal:[10,10]
keysym: [10,15]

⊢ Infer

Example: Memory Leak Detection

```
1   int swTableColumn_add(swTable *table, ...) {
2     col = sw_malloc(sizeof(swTableColumn));
3     if (type == SW_TABLE_INT)
4         col->size = 1;
5     col->index = table->size;
6     return swHashMap_add(table->columns, ..., col);
7   }
8
9   int swHashMap_add(swHashMap *hmap, ..., void *data) {
10    node = sw_malloc(sizeof(swHashMap_node));
11    if (node == NULL)
12        return SW_ERR;
13    node->data = data;
14    swHashMap_node_add(hmap, ... node);
15    return SW_OK;
16  }
```

Memory leak

Infer

```
Memory Leak:
An object allocated at line 2
becomes unreachable after line 7
```

Example: Double Free Detection

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {

  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);

  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

메모리 할당

메모리 해제

메모리 중복 해제
(double-free)

⊢ Infer

USB: fix double frees in error code paths of ipaq driver

the error code paths can be enter with buffers to freed buffers.
Serial core would do a kfree() on memory already freed.

Signed-off-by: Oliver Neukum <oneukum@suse.de>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

master **v4.15-rc1** ... v2.6.24-rc1

Oliver Neukum committed with **gregkh** on 18 Sep 2007 1 par

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

memory leak

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```


USB: fix double kfree in ipaq in error case

in the error case the ipaq driver leaves a dangling pointer to already freed memory that will be freed again.

Signed-off-by: Oliver Neukum <oneukum@suse.de>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

master v4.15-rc1 ... v2.6.27-rc1

Oliver Neukum committed with **gregkh** on 30 Jun 2008 1 parent 35

fix for a memory leak in an error case introduced by fix for double free

The fix NULLed a pointer without freeing it.

Signed-off-by: Oliver Neukum <oneukum@suse.de>
Reported-by: Juha Motorsportcom <juha_motorsportcom@luukku.com>
Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>

master v4.15-rc1 ... v2.6.27-rc1

Oliver Neukum committed with **torvalds** on 27 Jul 2008 1 parent 9ee08c2

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);
out = NULL;
in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

Static Analysis-based SW Repair

```c
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {

  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);

  goto err;
}
... // use in, out
err:
  free(in); // double-free
  free(out);// double-free
  return;
```

SAVER

✓Productivity↑

✓Quality↑

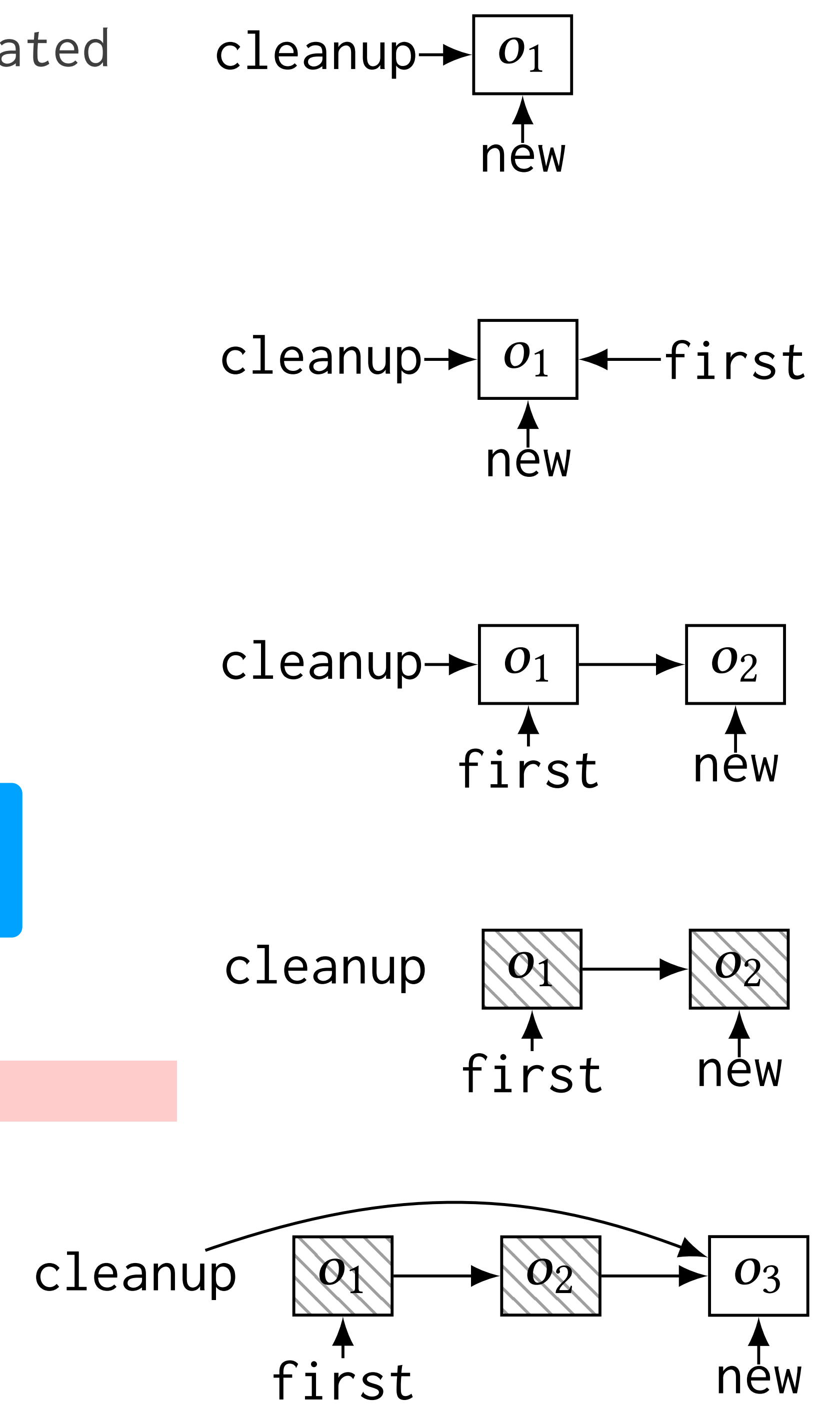
✓Safety guarantee

```c
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {

  goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
  free(in);

  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

Example: Use-After-Free Detection

```
struct node *cleanup; // list of objects to be deallocated
struct node *first = NULL;
for (...) {
    struct node *new = xmalloc(sizeof(*new));
    make_cleanup(new); // add new to the cleanup list
    new->name = ...;
    ...
    if (...) {
        first = new;

        continue;
    }
  /* potential use-after-free: `first->name` */
(-)  if (first == NULL || new->name != first->name)

        continue;
  do_cleanups(); // deallocate all objects in cleanup
}
```

use-after-free



Example: Use-After-Free Detection

```
struct node *cleanup; // list of objects to be deallocated
struct node *first = NULL;
for (...) {
    struct node *new = xmalloc(sizeof(*new));
    make_cleanup(new); // add new to the cleanup list
    new->name = ...;
    ...
    if (...) {
        first = new;
(+)     tmp = first->name;
        continue;
    }
  /* potential use-after-free: `first->name` */
(-)   if (first == NULL || new->name != first->name)
(+)   if (first == NULL || new->name != tmp)
        continue;
  do_cleanups(); // deallocate all objects in cleanup
}
```

Pointer Analysis

- Pointer analysis computes the set of memory locations (objects) that a pointer variable may point to at runtime.
- One of the most important static analyses: all interesting questions about program properties need pointer analysis.
  - E.g., control-flows, data-flows, types, numeric values, etc

Need for Pointer Analysis

- Example 1: Detecting memory errors in C programs
- Example 2: Callgraph construction

Abstraction of Memory Objects

- Memory locations are unbounded:

```
def id (p): return p

def f():
  x = A()     // l1
  y = id(x)

def g():
  a = B()     // l2
  b = id(a)

while True: {f(); g()}
```

- In a typical pointer analysis, objects are abstracted into their **allocation-sites**. Pointer analysis result:

$$x \mapsto \{l_1\}, y \mapsto \{l_1\}, a \mapsto \{l_2\}, b \mapsto \{l_2\}, p \mapsto \{l_1, l_2\}$$

cf) Flow Sensitivity

- A flow-sensitive analysis maintains abstract states separately for each program point: e.g.,

```
x = A()
y = id(x)
x = B()
y = id(x)
```

- Pointer analysis is often defined flow-insensitively

Constraint-based Analysis

- Pointer analysis is expressed as subset constraints. The analysis is to compute the smallest solution of the constraints. E.g.,

```
x = A() // l1
y = x
```

$$\Longrightarrow \quad \begin{array}{c} \{l_1\} \subseteq pts(x) \\ pts(x) \subseteq pts(y) \end{array}$$

- We use the Datalog language to express such constraints

Input and Output Relations

- A program is represented by a set of "facts" (relations):

$\text{Alloc}(var : V, heap : H)$
$\text{Move}(to : V, from : V)$
$\text{Load}(to : V, base : V, fld : F)$
$\text{Store}(base : V, fld : F, from : V)$

V: the set of program variables
H: the set of allocation sites
F: the set of field names

- Output relations: $\text{VarPointsTo}(var : V, heap : H)$
$\text{FldPointsTo}(baseH : H, fld : F, heap : H)$

```
a = A() // l1
b = B() // l2
c = a
a.f = b
d = c.f
```

➡

$\text{Alloc}(a, l_1)$
$\text{Alloc}(b, l_2)$
$\text{Move}(c, a)$
$\text{Store}(a, f, b)$
$\text{Load}(d, c, f)$

➡

$\text{VarPointsTo}(a, l_1)$
$\text{VarPointsTo}(b, l_2)$
$\text{VarPointsTo}(c, l_1)$
$\text{FldPointsTo}(l_1, f, l_2)$
$\text{VarPointsTo}(d, l_2)$

Fixed Point Computation

Alloc(a, l_1)
Alloc(b, l_2)
Move(c, a)
Store(a, f, b)
Load(d, c, f)

$\xrightarrow{(1)}$

Alloc(a, l_1)
Alloc(b, l_2)
Move(c, a)
Store(a, f, b)
Load(d, c, f)
VarPointsTo(a, l_1)
VarPointsTo(b, l_2)

$\xrightarrow{(2), (3)}$

Alloc(a, l_1)
Alloc(b, l_2)
Move(c, a)
Store(a, f, b)
Load(d, c, f)
VarPointsTo(a, l_1)
VarPointsTo(b, l_2)
VarPointsTo(c, l_1)
FldPointsTo(l_1, f, l_2)

$\xrightarrow{(4)}$

Alloc(a, l_1)
Alloc(b, l_2)
Move(c, a)
Store(a, f, b)
Load(d, c, f)
VarPointsTo(a, l_1)
VarPointsTo(b, l_2)
VarPointsTo(c, l_1)
FldPointsTo(l_1, f, l_2)
VarPointsTo(d, l_2)

Pointer Analysis Rules

(1) $\text{VarPointsTo}(var, heap) \leftarrow \text{Alloc}(var, heap)$

(2) $\text{VarPointsTo}(to, heap) \leftarrow$
$\quad \text{Move}(to, from), \text{VarPointsTo}(from, heap)$

(3) $\text{FldPointsTo}(baseH, fld, heap) \leftarrow$
$\quad \text{Store}(base, fld, from), \text{VarPointsTo}(from, heap),$
$\quad \text{VarPointsTo}(base, baseH)$

(4) $\text{VarPointsTo}(to, heap) \leftarrow$
$\quad \text{Load}(to, base, fld), \text{VarPointsTo}(base, baseH),$
$\quad \text{FldPointsTo}(baseH, fld, heap)$

Interprocedural Analysis (First-Order)

```
def f(p):   // m1
  return p
a = A()     // l1
b = f(a)    // l2
```

→

$\text{FormalArg}(m_1, 0, p)$

$\text{FormalReturn}(m_1, p)$

$\text{Alloc}(a, l_1, global)$

$\text{CallGraph}(l_2, m_1)$

$\text{Reachable}(global)$

$\text{Reachable}(m_1)$

$\text{ActualArg}(l_2, 0, a)$

$\text{ActualReturn}(l_2, b)$

→

$\text{InterProcAssign}(p, a)$

$\text{InterProcAssign}(b, p)$

$\text{VarPointsTo}(a, l_1)$

$\text{VarPointsTo}(p, l_1)$

$\text{VarPointsTo}(b, l_1)$

Input and Output Relations

- Input relations (program representation)

$\text{Alloc}(var : V, heap : H, inMeth : M)$

$\text{Move}(to : V, from : V)$

$\text{Load}(to : V, base : V, fld : F)$

$\text{Store}(base : V, fld : F, from : V)$

$\text{CallGraph}(invo : I, meth : M)$

$\text{Reachable}(meth : M)$

$\text{FormalArg}(meth : M, i : \mathbb{N}, arg : V)$

$\text{ActualArg}(invo : I, i : \mathbb{N}, arg : V)$

$\text{FormalReturn}(meth : M, ret : V)$

$\text{ActualReturn}(invo : I, var : V)$

V: the set of program variables
H: the set of allocation sites
F: the set of field names
M: the set of method identifiers
S: the set of method signatures
I: the set of instructions
T: the set of class types
$\mathbb{N}$: the set of natural numbers

- Output relations

$\text{VarPointsTo}(var : V, heap : H)$

$\text{FldPointsTo}(baseH : H, fld : F, heap : H)$

$\text{InterProcAssign}(to : V, from : V)$

Fixed Point Computation

FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc($a, l_1, global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)

(1), (5), (6) $\longrightarrow$

FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc($a, l_1, global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)
VarPointsTo(a, l_1)
InterProcAssign(p, a)
InterProcAssign(b, p)

(7) $\longrightarrow^*$

FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc($a, l_1, global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)
VarPointsTo(a, l_1)
InterProcAssign(p, a)
InterProcAssign(b, p)
VarPointsTo(p, l_1)
VarPointsTo(b, l_1)

Analysis Rules

(1) VarPointsTo($var, heap$) ← Reachable($meth$), Alloc($var, heap, meth$)

(2) VarPointsTo($to, heap$) ←
Move($to, from$), VarPointsTo($from, heap$)

(3) FldPointsTo($baseH, fld, heap$) ←
Store($base, fld, from$), VarPointsTo($from, heap$), VarPointsTo($base, baseH$)

(4) VarPointsTo($to, heap$) ←
Load($to, base, fld$), VarPointsTo($base, baseH$), FldPointsTo($baseH, fld, heap$)

(5) InterProcAssign($to, from$) ←
CallGraph($invo, meth$), FormalArg($meth, n, to$), ActualArg($invo, n, from$)

(6) InterProcAssign($to, from$) ←
CallGraph($invo, meth$), FormalReturn($meth, from$), ActualReturn($invo, to$)

(7) VarPointsTo($to, heap$) ←
InterProcAssign($to, from$), VarPointsTo($from, heap$)

Interprocedural Analysis (Higher-Order)

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

$\Rightarrow$

FormalArg(m_1,0,v)
FormalReturn(m_1, v)
ThisVar(m_1, $self$)
LookUp(C, id, m_1)
ThisVar(m_2, $self$)
LookUp(B, g, m_2)
Alloc(c, l_1, m_2)
Alloc(s, l_2, m_2)
Alloc(t, l_3, m_2)
HeapType(l_1, C)
HeapType(l_2, D)
HeapType(l_3, E)

VCall(c, id, l_4, m_2)
VCall(c, id, l_5, m_2)
ActualArg(l_4,0,s)
ActualArg(l_5,0,t)
ActualReturn(l_4, d)
ActualReturn(l_5, e)
ThisVar(m_3, $self$)
LookUp(A, f, m_3)
Alloc(b, l_6, m_3)
HeapType(l_6, B)
VCall(b, g, l_7, m_3)
VCall(b, g, l_8, m_3)
Reachable(m_3)

$\Rightarrow$

VarPointsTo(b, l_6)
Reachable(m_2)
VarPointsTo($self$, l_6)
CallGraph(l_7, m_2)
CallGraph(l_8, m_2)
VarPointsTo(c, l_1)
VarPointsTo(s, l_2)
VarPointsTo(t, l_3)
Reachable(m_1)
VarPointsTo($self$, l_1)
CallGraph(l_4, m_1)
CallGraph(l_5, m_1)

InterProcAssign(v, s)
InterProcAssign(v, t)
InterProcAssign(d, v)
InterProcAssign(e, v)
VarPointsTo(v, l_2)
VarPointsTo(v, l_3)
VarPointsTo(d, l_2)
VarPointsTo(d, l_3)
VarPointsTo(e, l_2)
VarPointsTo(e, l_3)

Input and Output Relations

- Input relations

Alloc$(var : V, heap : H, inMeth : M)$

Move$(to : V, from : V)$

Load$(to : V, base : V, fld : F)$

Store$(base : V, fld : F, from : V)$

VCall$(base : V, sig : S, invo : I, inMeth : M)$

FormalArg$(meth : M, i : \mathbb{N}, arg : V)$

ActualArg$(invo : I, i : \mathbb{N}, arg : V)$

FormalReturn$(meth : M, ret : V)$

ActualReturn$(invo : I, var : V)$

ThisVar$(meth : M, this : V)$

HeapType$(heap : H, type : T)$

LookUp$(type : T, sig : S, meth : M)$

- Output relations

VarPointsTo$(var : V, heap : H)$

FldPointsTo$(baseH : H, fld : F, heap : H)$

InterProcAssign$(to : V, from : V)$

CallGraph$(invo : I, meth : M)$

Reachable$(meth : M)$

Analysis Rules

(1) VarPointsTo($var, heap$) $\leftarrow$ Reachable($meth$), Alloc($var, heap, meth$)

(2) VarPointsTo($to, heap$) $\leftarrow$
Move($to, from$), VarPointsTo($from, heap$)

(3) FldPointsTo($baseH, fld, heap$) $\leftarrow$
Store($base, fld, from$), VarPointsTo($from, heap$), VarPointsTo($base, baseH$)

(4) VarPointsTo($to, heap$) $\leftarrow$
Load($to, base, fld$), VarPointsTo($base, baseH$), FldPointsTo($baseH, fld, heap$)

(5) InterProcAssign($to, from$) $\leftarrow$
CallGraph($invo, meth$), FormalArg($meth, n, to$), ActualArg($invo, n, from$)

(6) InterProcAssign($to, from$) $\leftarrow$
CallGraph($invo, meth$), FormalReturn($meth, from$), ActualReturn($invo, to$)

(7) VarPointsTo($to, heap$) $\leftarrow$
InterProcAssign($to, from$), VarPointsTo($from, heap$)

Analysis Rules

(8) Reachable($toMeth$),
VarPointsTo($this, heap$),
CallGraph($invo, toMeth$) $\leftarrow$
VCall($base, sig, invo, inMeth$), Reachable($inMeth$),
VarPointsTo($base, heap$),
HeapType($heap, heapT$), LookUp($heapT, sig, toMeth$),
ThisVar($toMeth, this$)

- This analysis performs **on-the-fly call-graph construction.** Pointer analysis and call-graph construction are closely inter-connected in object-oriented and higher-order languages. For example, to resolve call `obj.fun()`, we need pointer analysis. To compute points-to set of `a` in `f(Object a){...}`, we need call-graph.

$\mathsf{FormalArg}(m_1, 0, v)$
$\mathsf{FormalReturn}(m_1, v)$
$\mathsf{ThisVar}(m_1, \mathit{self})$
$\mathsf{LookUp}(C, id, m_1)$
$\mathsf{ThisVar}(m_2, \mathit{self})$
$\mathsf{LookUp}(B, g, m_2)$
$\mathsf{Alloc}(c, l_1, m_2)$
$\mathsf{Alloc}(s, l_2, m_2)$
$\mathsf{Alloc}(t, l_3, m_2)$
$\mathsf{HeapType}(l_1, C)$
$\mathsf{HeapType}(l_2, D)$
$\mathsf{HeapType}(l_3, E)$
$\mathsf{VCall}(c, id, l_4, m_2)$
$\mathsf{VCall}(c, id, l_5, m_2)$
$\mathsf{ActualArg}(l_4, 0, s)$
$\mathsf{ActualArg}(l_5, 0, t)$
$\mathsf{ActualReturn}(l_4, d)$
$\mathsf{ActualReturn}(l_5, e)$
$\mathsf{ThisVar}(m_3, \mathit{self})$
$\mathsf{LookUp}(A, f, m_3)$
$\mathsf{Alloc}(b, l_6, m_3)$
$\mathsf{HeapType}(l_6, B)$
$\mathsf{VCall}(b, g, l_7, m_3)$
$\mathsf{VCall}(b, g, l_8, m_3)$
$\mathsf{Reachable}(m_3)$

$$\xrightarrow{(1)} \mathsf{VarPointsTo}(b, l_6) \xrightarrow{(8)} \begin{array}{l}\mathsf{Reachable}(m_2)\\ \mathsf{VarPointsTo}(\mathit{self}, l_6)\\ \mathsf{CallGraph}(l_7, m_2)\\ \mathsf{CallGraph}(l_8, m_2)\end{array} \xrightarrow{(1)} \begin{array}{l}\mathsf{VarPointsTo}(c, l_1)\\ \mathsf{VarPointsTo}(s, l_2)\\ \mathsf{VarPointsTo}(t, l_3)\end{array}$$

$$\xrightarrow{(8)} \begin{array}{l}\mathsf{Reachable}(m_1)\\ \mathsf{VarPointsTo}(\mathit{self}, l_1)\\ \mathsf{CallGraph}(l_4, m_1)\\ \mathsf{CallGraph}(l_5, m_1)\end{array} \xrightarrow{(5),(6)} \begin{array}{l}\mathsf{InterProcAssign}(v, s)\\ \mathsf{InterProcAssign}(v, t)\\ \mathsf{InterProcAssign}(d, v)\\ \mathsf{InterProcAssign}(e, v)\end{array} \xrightarrow{(7)} \begin{array}{l}\mathsf{VarPointsTo}(v, l_2)\\ \mathsf{VarPointsTo}(v, l_3)\end{array}$$

$$\xrightarrow{(7)} \begin{array}{l}\mathsf{VarPointsTo}(d, l_2)\\ \mathsf{VarPointsTo}(d, l_3)\\ \mathsf{VarPointsTo}(e, l_2)\\ \mathsf{VarPointsTo}(e, l_3)\end{array}$$

```
class C:
  def id(self, v): // m1
     return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

Context Sensitivity

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

context-insensitive

VarPointsTo(b, l_6)
VarPointsTo$(self, l_6)$
VarPointsTo(c, l_1)
VarPointsTo(s, l_2)
VarPointsTo(t, l_3)
VarPointsTo$(self, l_1)$
VarPointsTo(v, l_2)
VarPointsTo(v, l_3)
VarPointsTo(d, l_2)
VarPointsTo(d, l_3)
VarPointsTo(e, l_2)
VarPointsTo(e, l_3)

context-sensitive

VarPointsTo$(b, \star, l_6, \star)$
VarPointsTo$(self, l_7, l_6, \star)$
VarPointsTo$(self, l_8, l_6, \star)$
VarPointsTo$(c, l_7, l_1, \star)$
VarPointsTo$(s, l_7, l_2, \star)$
VarPointsTo$(t, l_7, l_3, \star)$
VarPointsTo$(c, l_8, l_1, \star)$
VarPointsTo$(s, l_8, l_2, \star)$
VarPointsTo$(t, l_8, l_3, \star)$
VarPointsTo$(self, l_4, l_1, \star)$
VarPointsTo$(self, l_5, l_1, \star)$
VarPointsTo$(v, l_4, l_2, \star)$
VarPointsTo$(v, l_5, l_3, \star)$
VarPointsTo$(d, l_7, l_2, \star)$
VarPointsTo$(d, l_8, l_2, \star)$
VarPointsTo$(e, l_7, l_3, \star)$
VarPointsTo$(e, l_8, l_3, \star)$

Domains

V: the set of program variables
H: the set of allocation sites
F: the set of field names
M: the set of method identifiers
S: the set of method signatures
I: the set of instructions
T: the set of class types
$\mathbb{N}$: the set of natural numbers
C: a set of calling contexts
HC: a set of heap contexts

Output Relations

- The output relations are modified to add contexts:

$$\text{VarPointsTo}(var : V, heap : H)$$
$$\text{FldPointsTo}(baseH : H, fld : F, heap : H)$$
$$\text{InterProcAssign}(to : V, from : V)$$
$$\text{CallGraph}(invo : I, meth : M)$$
$$\text{Reachable}(meth : M)$$

$$\Downarrow$$

$$\text{VarPointsTo}(var : V, ctx : C, heap : H, hctx : HC)$$
$$\text{FldPointsTo}(baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC)$$
$$\text{InterProcAssign}(to : V, toCtx : C, from : V, fromCtx : C)$$
$$\text{CallGraph}(invo : I, callerCtx : C, meth : M, calleeCtx : C)$$
$$\text{Reachable}(meth : M, ctx : C)$$

Context Constructors

- Different choices of constructors yield different context-sensitivity flavors

$$\textbf{Record}(heap : H, ctx : C) = newHCtx : HC$$
$$\textbf{Merge}(heap : H, hctx : HC, invo : I, ctx : C) = newCtx : C$$

- **Record** generates heap contexts
- **Merge** generates calling contexts

Analysis Rules

$\textbf{Record}(heap, ctx) = hctx,$
$\text{VarPointsTo}(var, ctx, heap, hctx) \leftarrow$
$\quad \text{Reachable}(meth, ctx), \text{Alloc}(var, heap, meth)$

$\text{VarPointsTo}(to, ctx, heap, hctx) \leftarrow$
$\quad \text{Move}(to, from), \text{VarPointsTo}(from, ctx, heap, hctx)$

$\text{FldPointsTo}(baseH, baseHCtx, fld, heap, hctx) \leftarrow$
$\quad \text{Store}(base, fld, from), \text{VarPointsTo}(from, ctx, heap, hctx),$
$\quad \text{VarPointsTo}(base, ctx, baseH, baseHCtx)$

$\text{VarPointsTo}(to, ctx, heap, hctx) \leftarrow$
$\quad \text{Load}(to, base, fld), \text{VarPointsTo}(base, ctx, baseH, baseHCtx),$
$\quad \text{FldPointsTo}(baseH, baseHCtx, fld, heap, hctx)$

Analysis Rules

$$\begin{aligned}
&\textbf{Merge}(heap, hctx, invo, callerCtx) = calleeCtx, \\
&\text{Reachable}(toMeth, calleeCtx), \\
&\text{VarPointsTo}(this, calleeCtx, heap, hctx), \\
&\text{CallGraph}(invo, callerCtx, toMeth, calleeCtx) \leftarrow \\
&\quad \text{VCall}(base, sig, invo, inMeth), \text{Reachable}(inMeth, callerCtx), \\
&\quad \text{VarPointsTo}(base, callerCtx, heap, hctx), \\
&\quad \text{HeapType}(heap, heapT), \text{LookUp}(heapT, sig, toMeth), \\
&\quad \text{ThisVar}(toMeth, this)
\end{aligned}$$

Analysis Rules

$$\text{InterProcAssign}(to, calleeCtx, from, callerCtx) \leftarrow$$
$$\quad \text{CallGraph}(invo, callerCtx, meth, calleeCtx),$$
$$\quad \text{FormalArg}(meth, n, to), \text{ActualArg}(invo, n, from)$$

$$\text{InterProcAssign}(to, callerCtx, from, calleeCtx) \leftarrow$$
$$\quad \text{CallGraph}(invo, callerCtx, meth, calleeCtx),$$
$$\quad \text{FormalReturn}(meth, from), \text{ActualReturn}(invo, to)$$

$$\text{VarPointsTo}(to, toCtx, heap, hctx) \leftarrow$$
$$\quad \text{InterProcAssign}(to, toCtx, from, fromCtx),$$
$$\quad \text{VarPointsTo}(from, fromCtx, heap, hctx)$$

Call-Site Sensitivity

- The best-known flavor of context sensitivity, which uses call-sites as contexts.
- A method is analyzed under the context that is a sequence of the last *k* call-sites
  - The current call-site of the method, the call-site of the caller method, the call-site of the caller method's caller, …, up to a pre-defined depth (*k*)

Call-Site Sensitivity

- 1-call-site sensitivity with context-insensitive heap:

$$C = I, \qquad HC = \{ \star \}$$
$$\textbf{Record}(heap, ctx) = \star$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = invo$$

- 1-call-site sensitivity with context-sensitive heap:

$$C = I, \qquad HC = I$$
$$\textbf{Record}(heap, ctx) = ctx$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = invo$$

- 2-call-site sensitivity with 1-call-site senstive heap:

$$C = I \times I, \qquad HC = I$$
$$\textbf{Record}(heap, ctx) = first(ctx)$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = pair(invo, first(ctx))$$

Object Sensitivity

- The dominant flavor of context sensitivity for object-oriented languages
- Object abstractions (i.e., allocation sites) are used as contexts, qualifying a method's local variables with the allocation site of the receiver object of the method call.

```
class A:
  def m(self):
    return

a = A()  // l1
a.m()    // l2
```

Object Sensitivity

- 1-object sensitivity with context-insensitive heap:

$$C = H, \qquad HC = \{ \star \}$$
$$\textbf{Record}(heap, ctx) = \star$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = heap$$

- 2-object sensitivity with 1-call-site senstive heap:

$$C = H \times H, \qquad HC = H$$
$$\textbf{Record}(heap, ctx) = first(ctx)$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = pair(heap, hctx)$$

Example

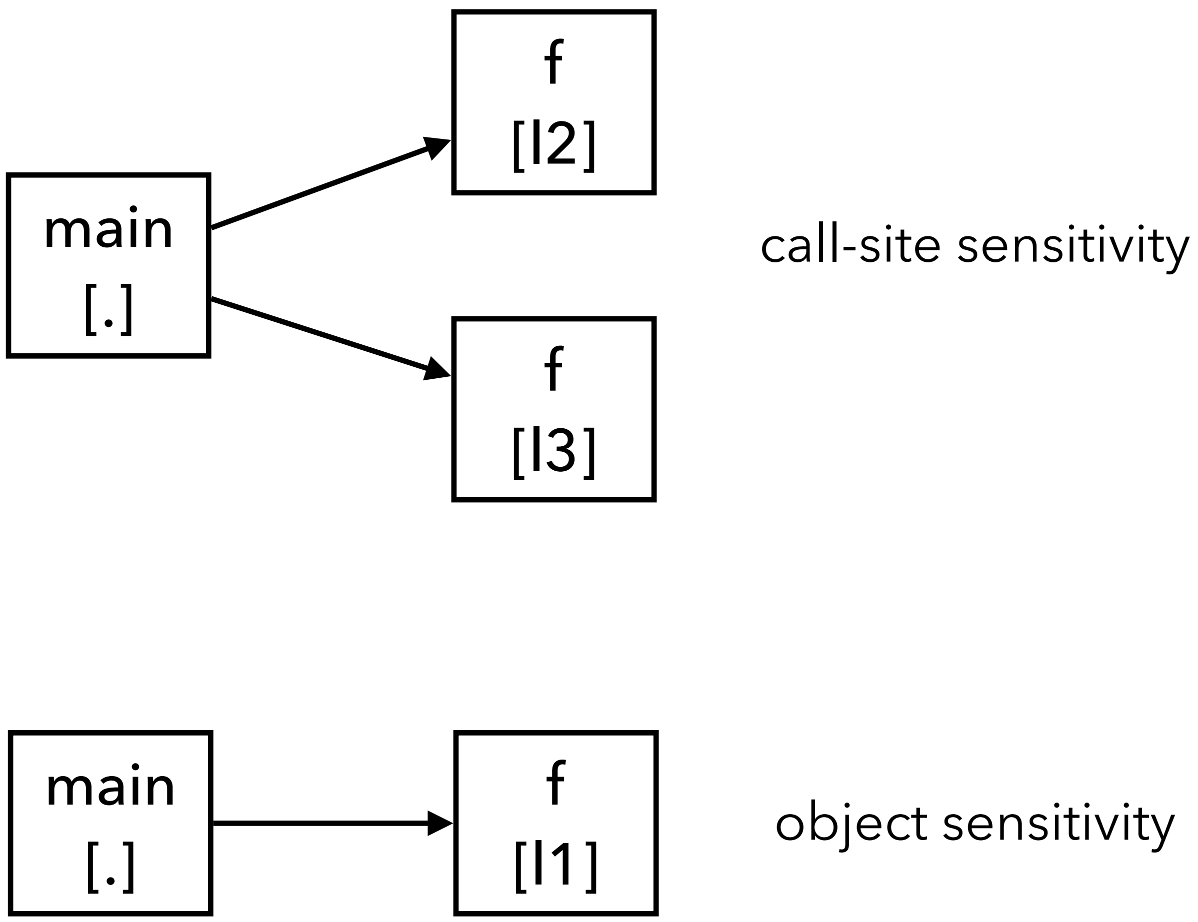
- 2-object sensitivity with 1-call-site senstive heap:

```
class C:
  def h(self):
     return

class B:
  def g(self):
    c = C()        // l3, heap objects: (l3, [l1]), (l3, [l2])
    c.h()          // contexts: (l3, l1), (l3, l2)

class A:
  def f(self):
    b1 = B()       // l1
    b2 = B()       // l2
    b1.g()         // context: l1
    b2.g()         // context: l2
```

Call-site vs. Object Sensitivity

- Typical example that benefits from call-site sensitivity:

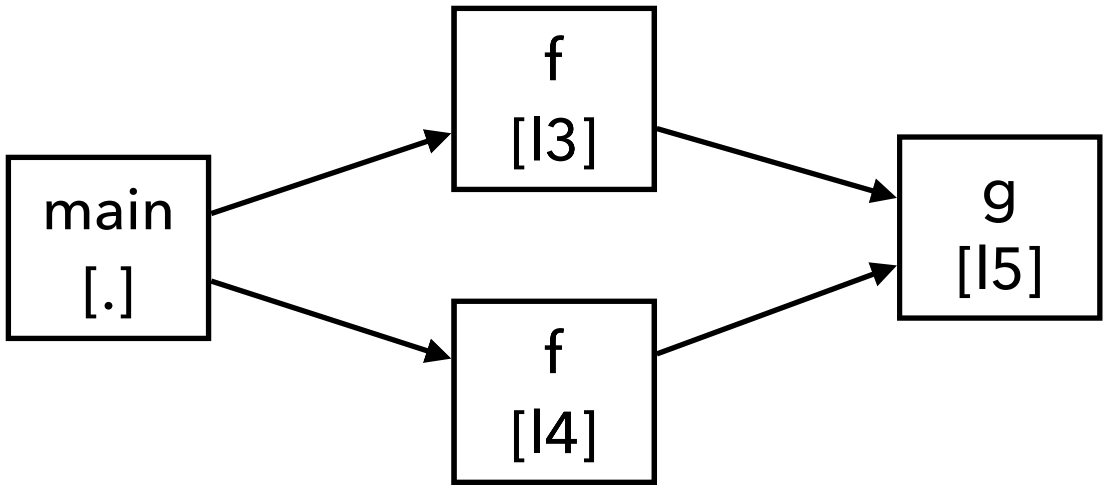
```
class A:
  def f(self): return

def main():
   a = A()   // l1
   a.f()     // l2
   a.f()     // l3
```



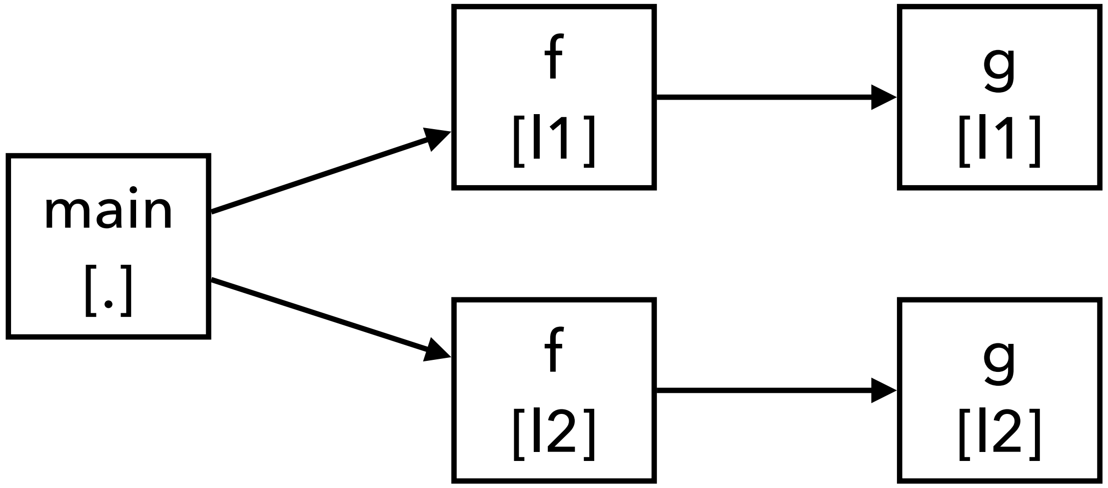
Call-site vs. Object Sensitivity

- Typical example that benefits from object sensitivity:

```
class A:
  def g(self):
     return
  def f(self):
     return self.g() // l5

def main():
   a = A()   // l1
   b = A()   // l2
   a.f()     // l3
   b.f()     // l4
```



1-call-site sensitivity



1-object sensitivity

Summary

- Static analysis examples
  - Numerical analysis: Sign, Interval, Octagon domains
  - Pointer analysis: First/Higher-order, Context sensitive
- Concepts covered
  - Abstract domain and semantics
  - Fixed point computation, acceleration, refinement
  - Analysis sensitivities: flow sensitivity, context sensitivity