AAA616: Program Analysis

Lecture 2 — Static Analysis Examples

Hakjoo Oh
2024 Fall

Principles of Static Analysis

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

Principles of Static Analysis

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

- Dynamic analysis (testing): 459

Principles of Static Analysis

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

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

Principles of Static Analysis

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

- Dynamic analysis (testing): 459
- Static analysis: a variety of answers
  - “integer” (type system)
  - “odd integer”
    1. Choose abstract value (domain)
  - “positive integer”
  - “integer between 400 and 500”
  - …

Principles of Static Analysis

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

- 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)

2. "Execute" the program with abstract values

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

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

Strength of Static Analysis

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

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

Strength of Static Analysis

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

⊤ (don't know)

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

Strength of Static Analysis

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

Even

$\top$ (don't know)

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

Strength of Static Analysis

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

$\top$ (don't know)

Even

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

Odd

Strength of Static Analysis

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

Even

$\top$ (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

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

Weakness of Static Analysis

- Instead, static analysis may produce false alarms

$\top$ (don't know)

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

Weakness of Static Analysis

- Instead, static analysis may produce false alarms

$\top$ (don't know)

$\top$ (don't know)

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

Weakness of Static Analysis

- Instead, static analysis may produce false alarms

⊤ (don’t know)

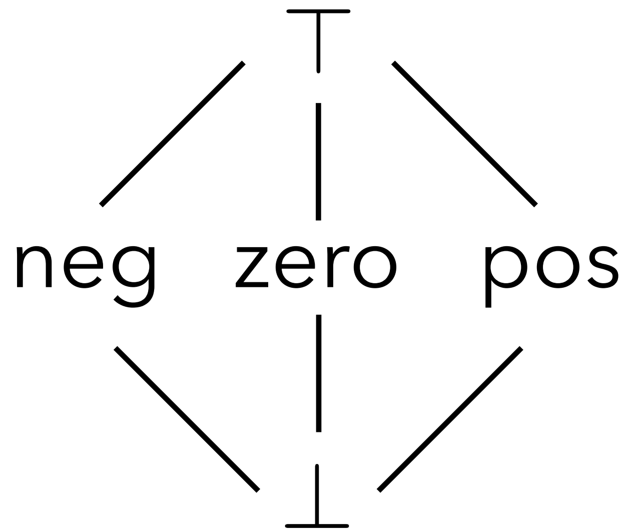
```
void f (int x) {
    y = x + x;
    assert (y % 2 == 0);
}
```

⊤ (don’t know)

false alarm

A Simple Sign Domain

- Abstract values



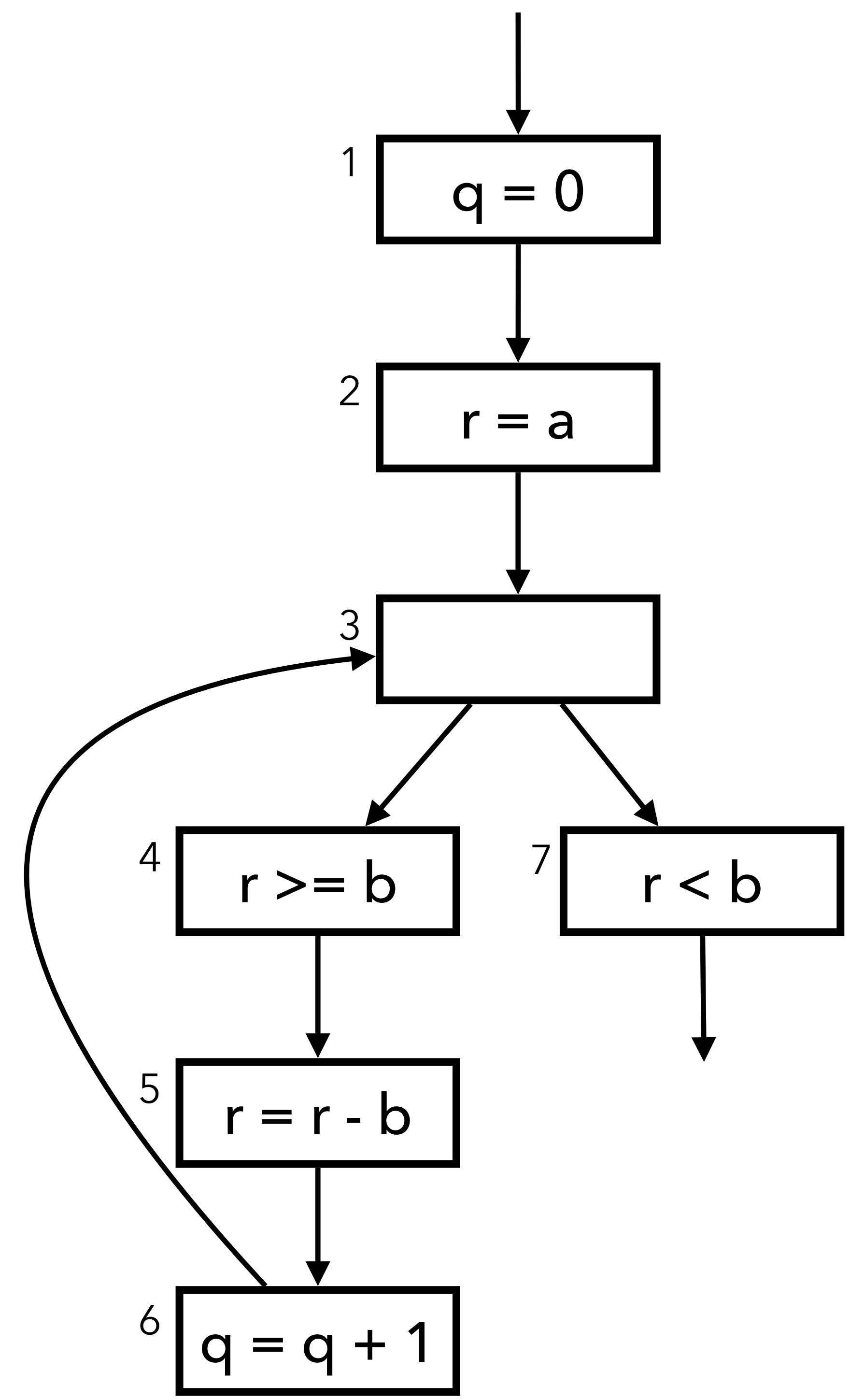
- Abstract operators

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

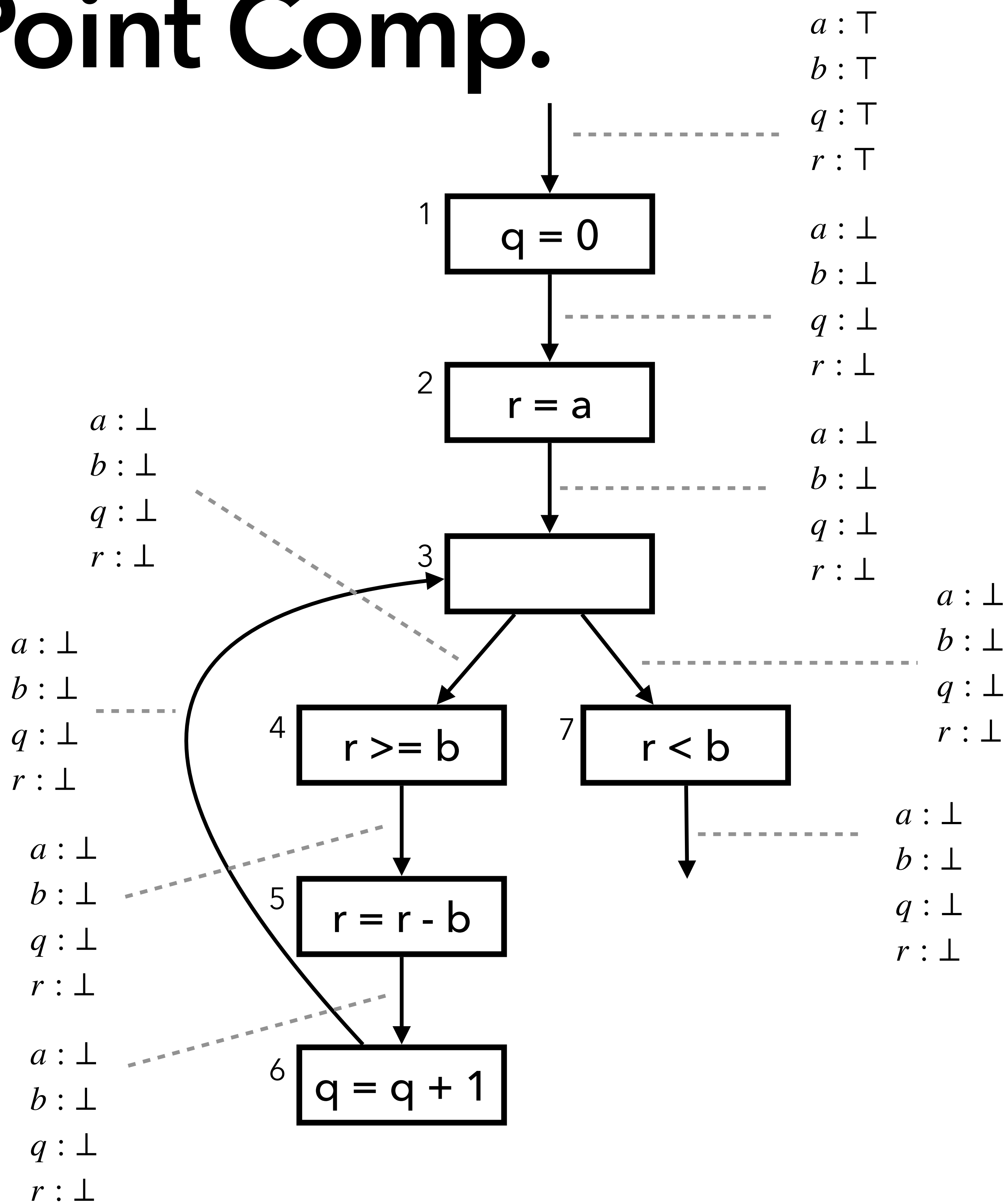
×	top	neg	zero	pos	bot
top					
neg					
zero					
pos					
bot					

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);
```

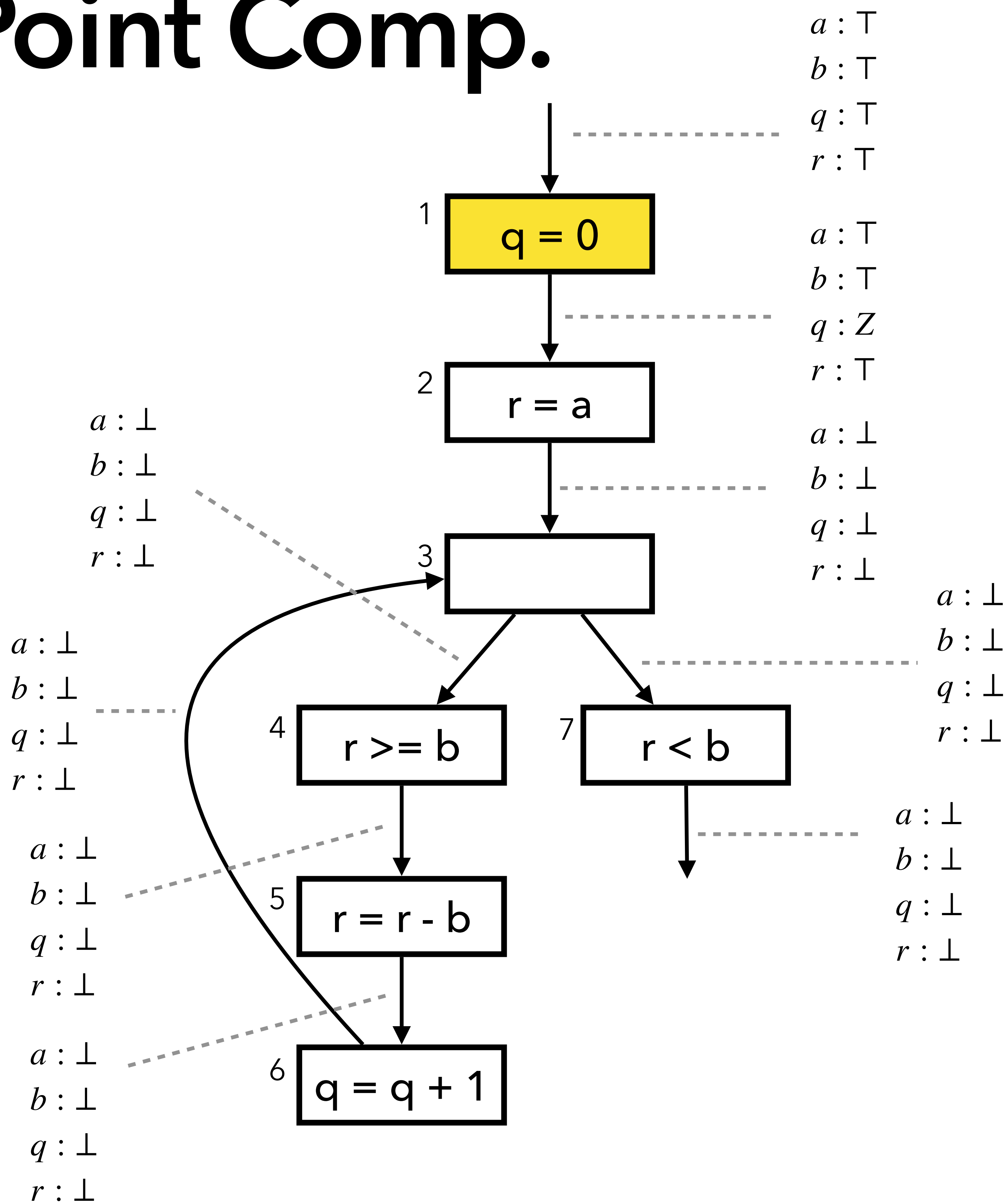


Fixed Point Comp.



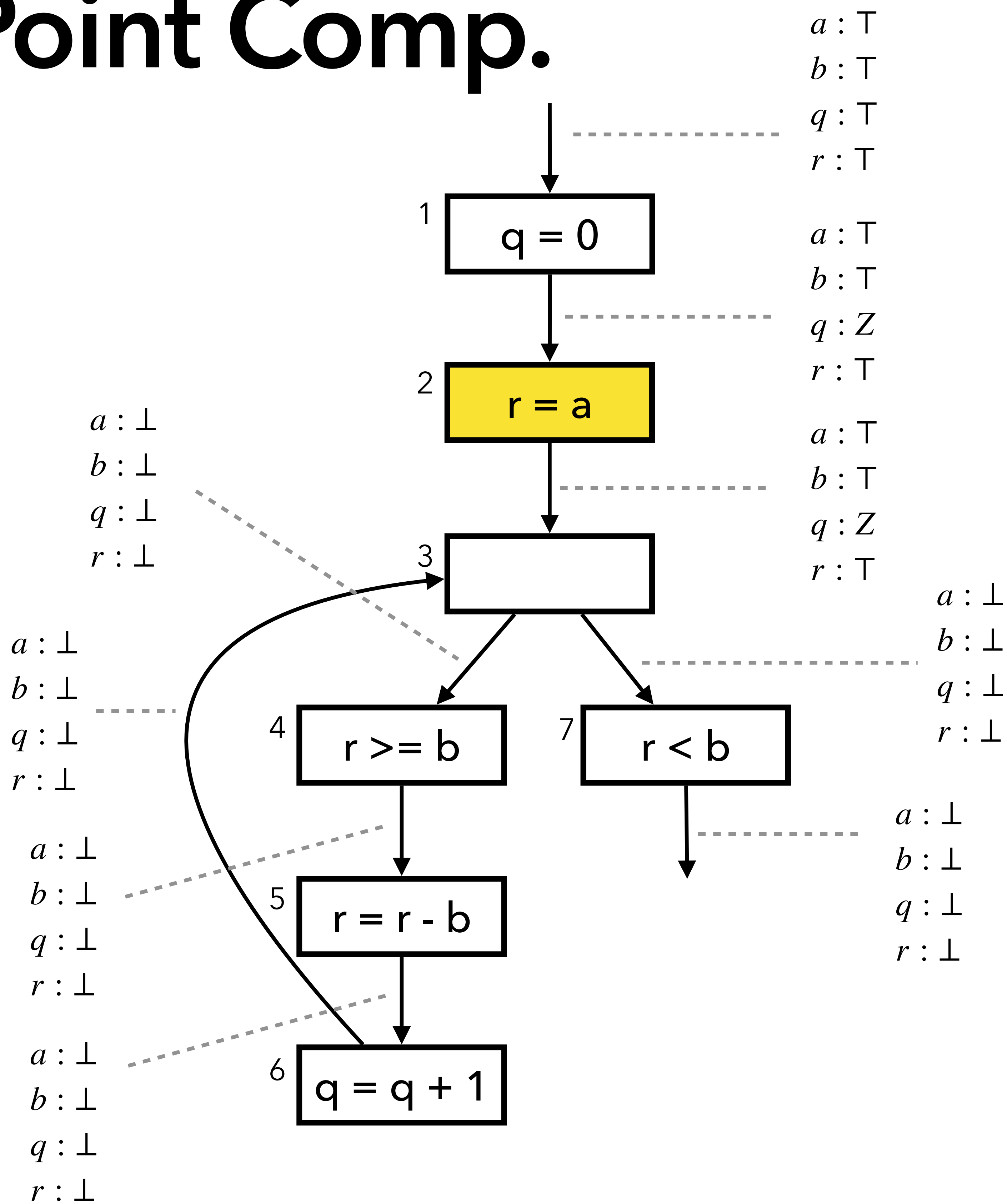
W = { 1, 2, 3, 4, 5, 6, 7 }

Fixed Point Comp.



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

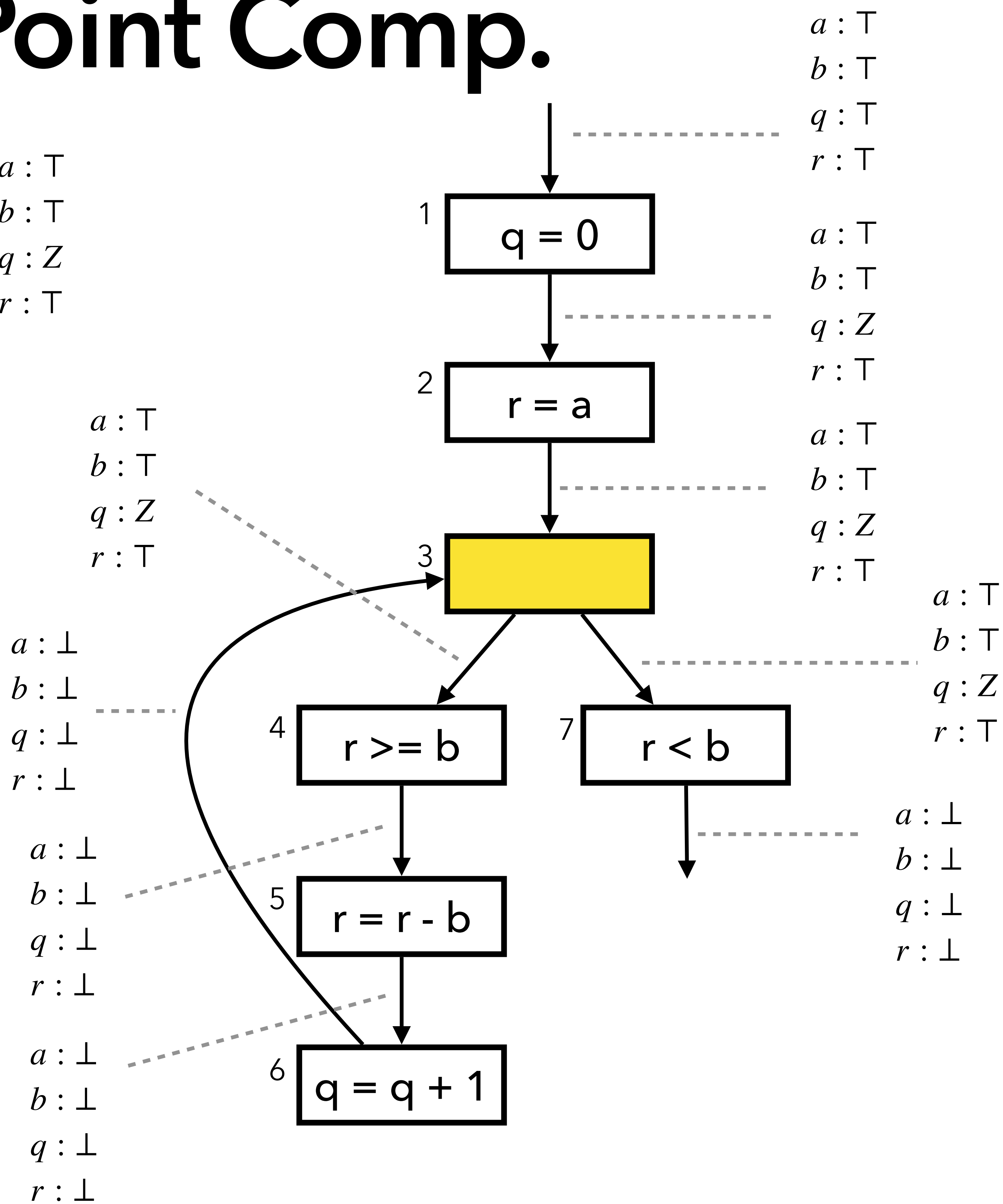
Fixed Point Comp.



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

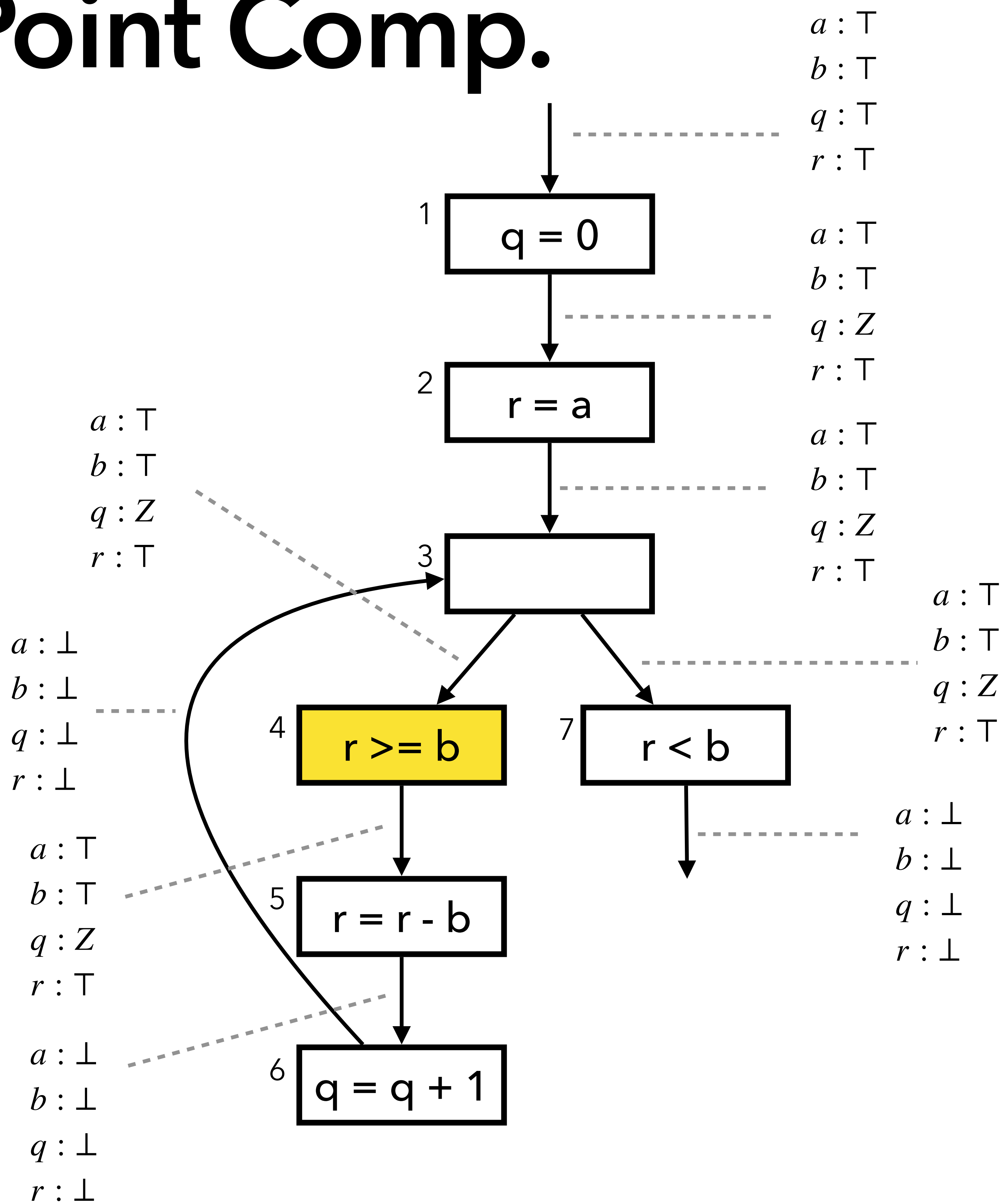
Fixed Point Comp.

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



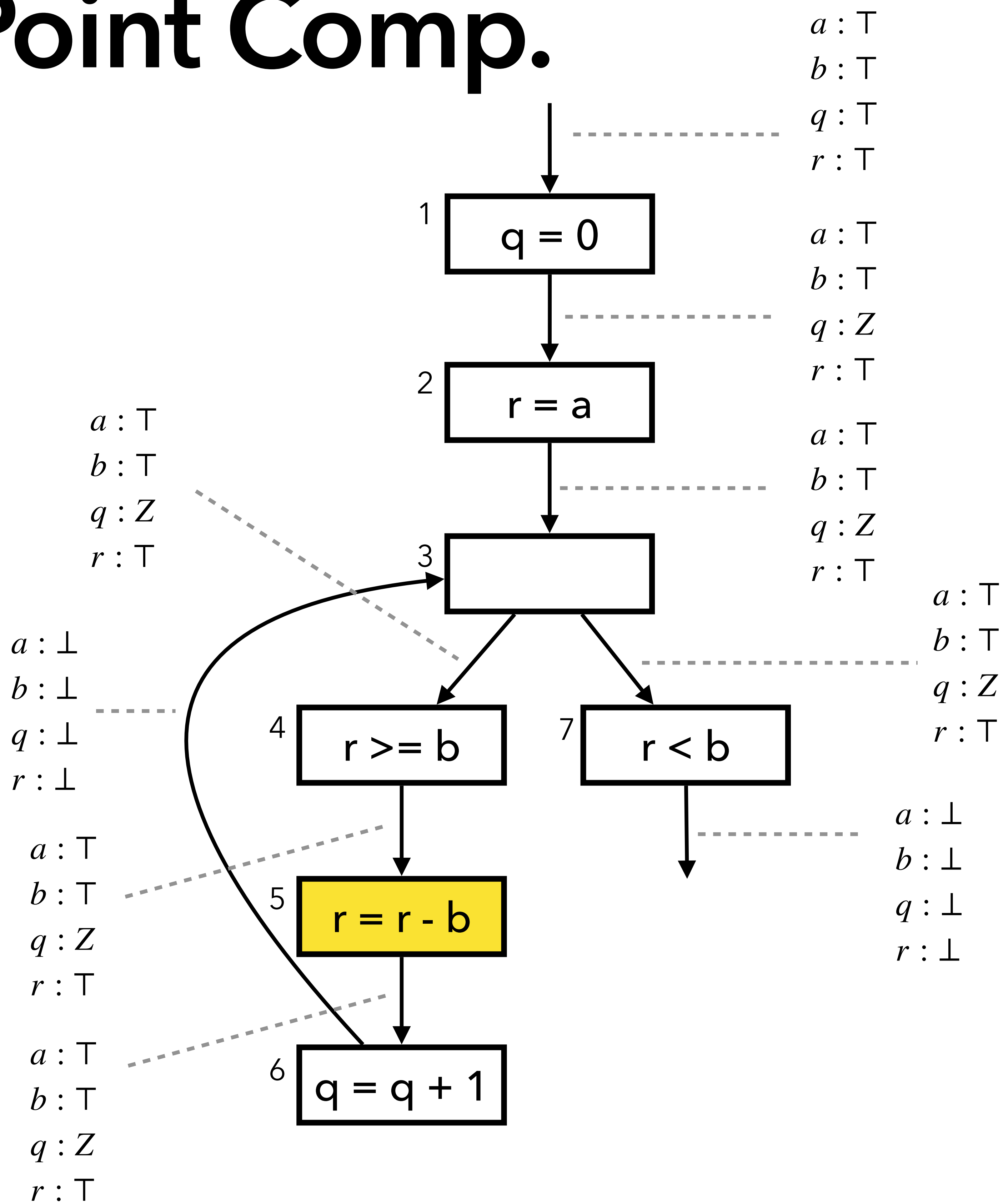
W = { ~~1~~, ~~2~~, ~~3~~, 4, 5, 6, 7 }

Fixed Point Comp.



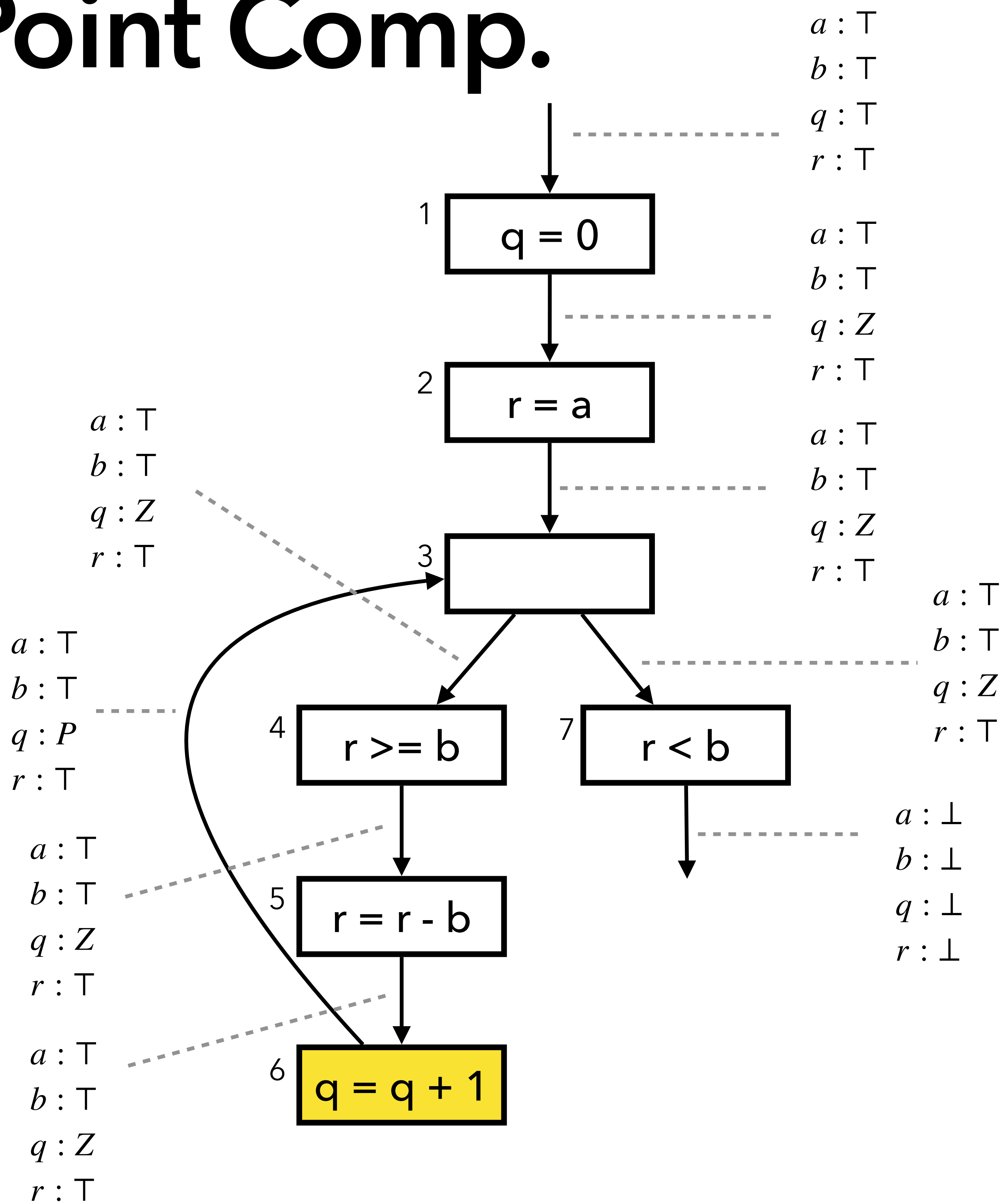
W = { ~~1~~, ~~2~~, ~~3~~, ~~4~~, 5, 6, 7 }

Fixed Point Comp.



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

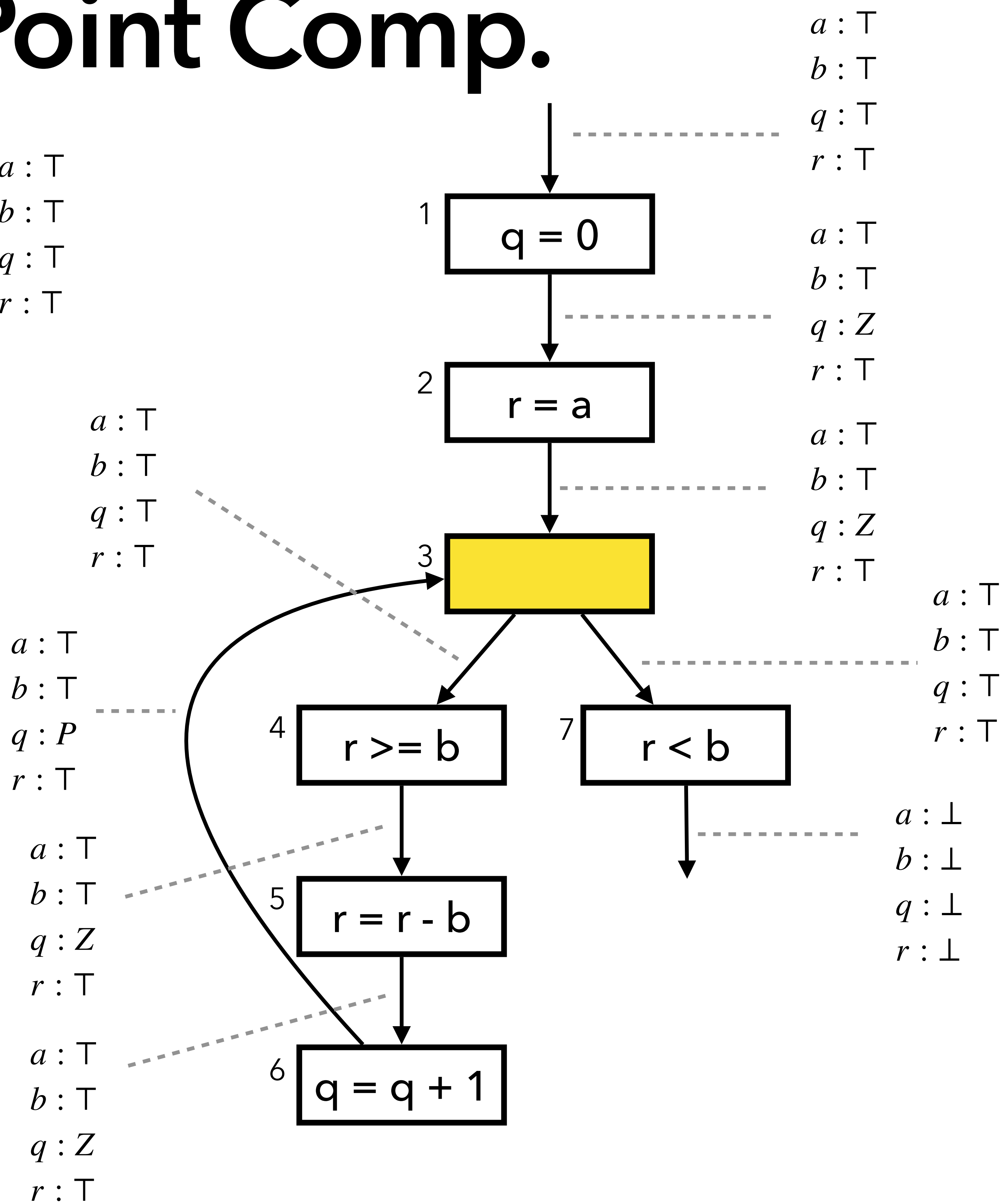
Fixed Point Comp.



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

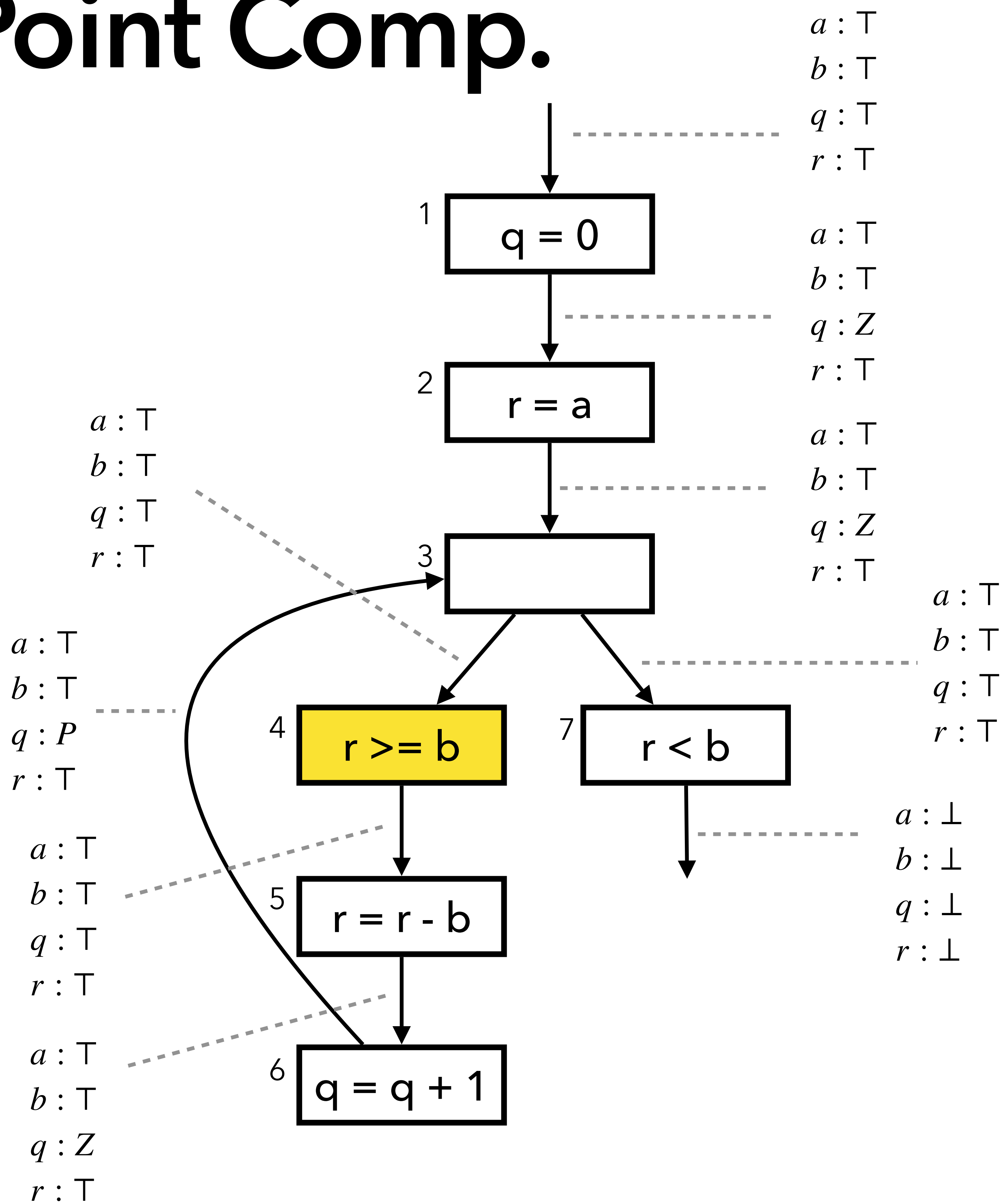
Fixed Point Comp.

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



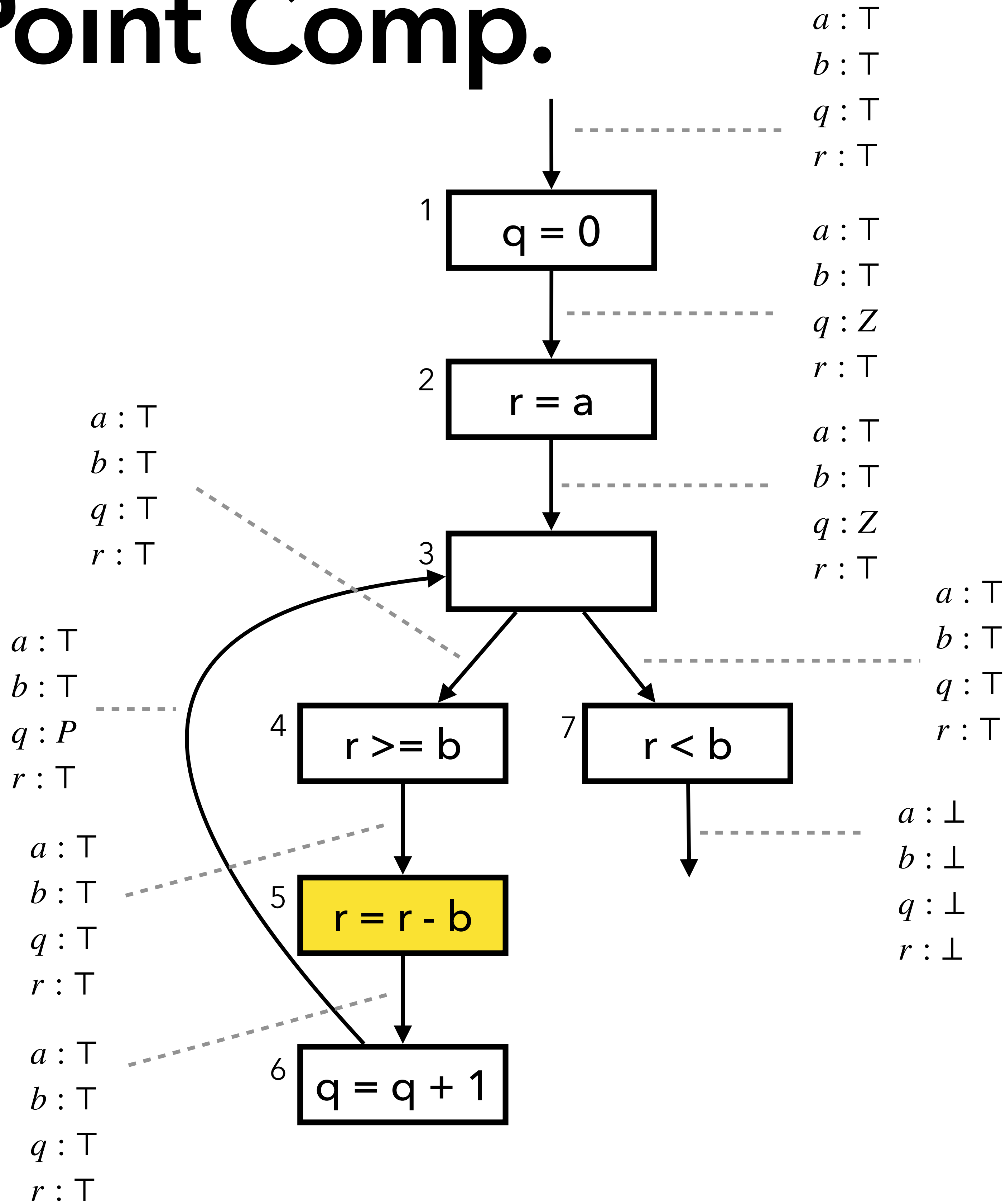
W = { ~~1~~, ~~2~~, ~~3~~, 4, ~~5~~, ~~6~~, 7 }

Fixed Point Comp.



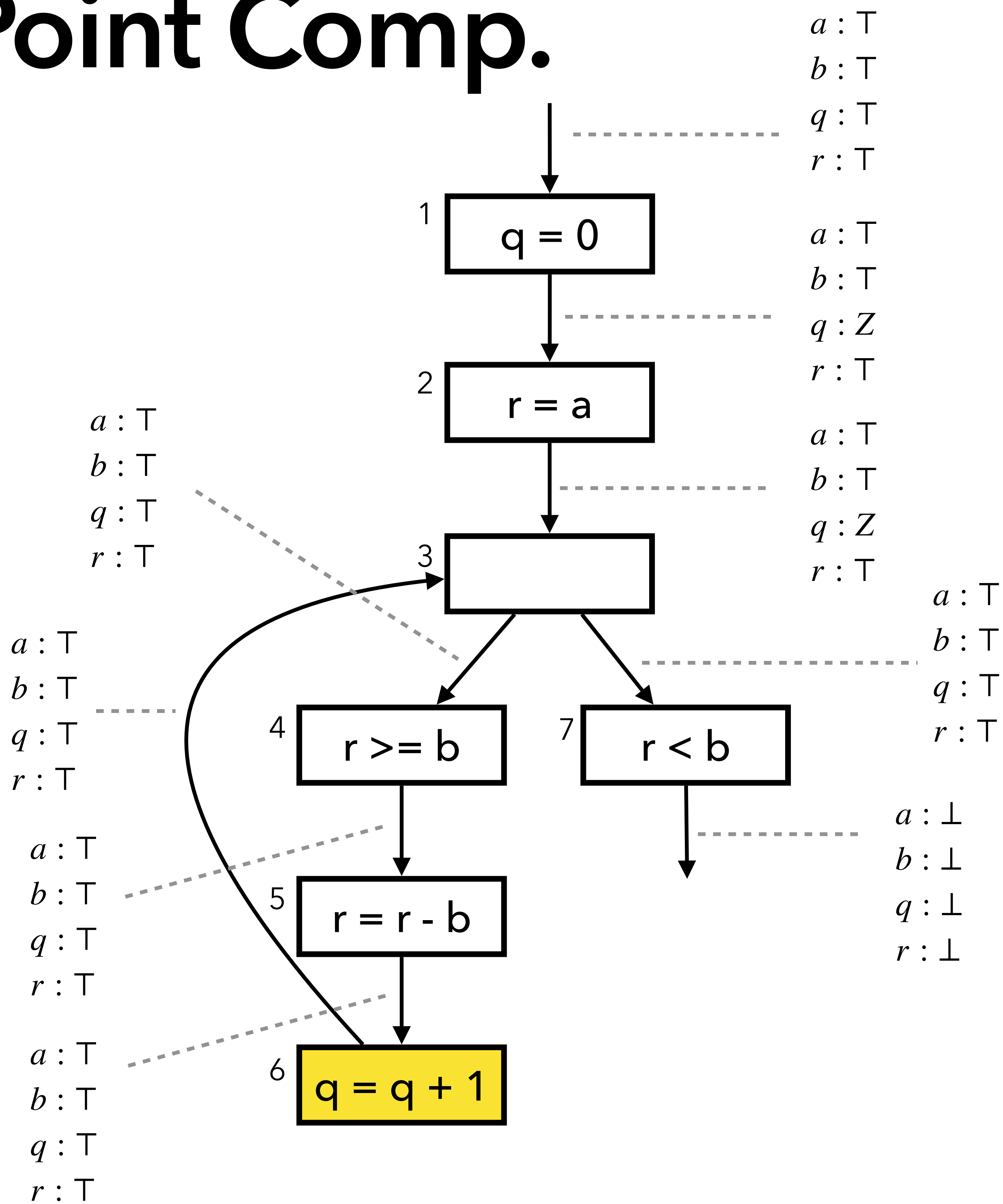
W = { $\cancel{1}$, $\cancel{2}$, $\cancel{3}$, $\cancel{4}$, 5, $\cancel{6}$, 7 }

Fixed Point Comp.



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

Fixed Point Comp.

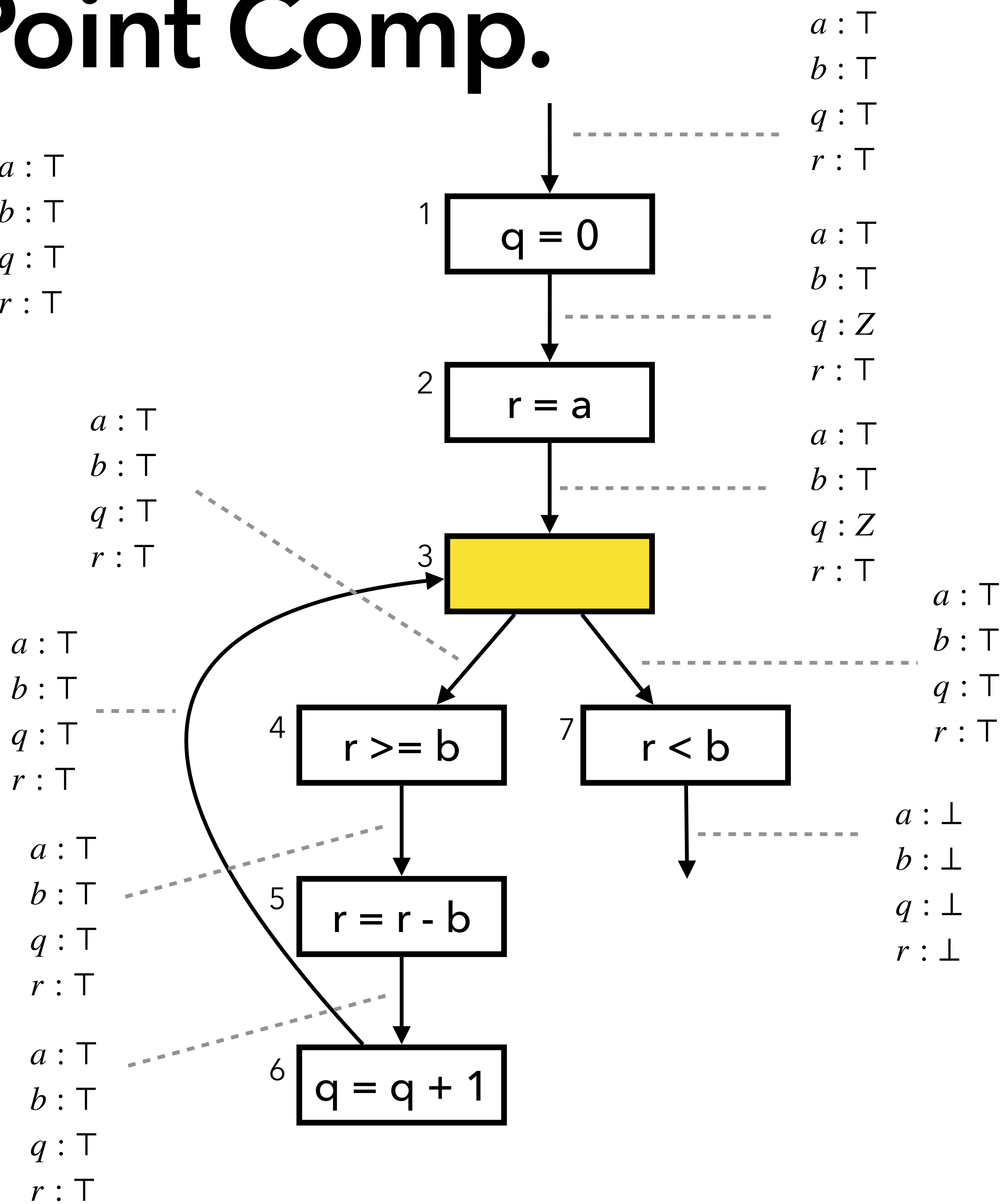


W = { ~~1~~, ~~2~~, 3, ~~4~~, ~~5~~, ~~6~~, 7 }

Fixed Point Comp.

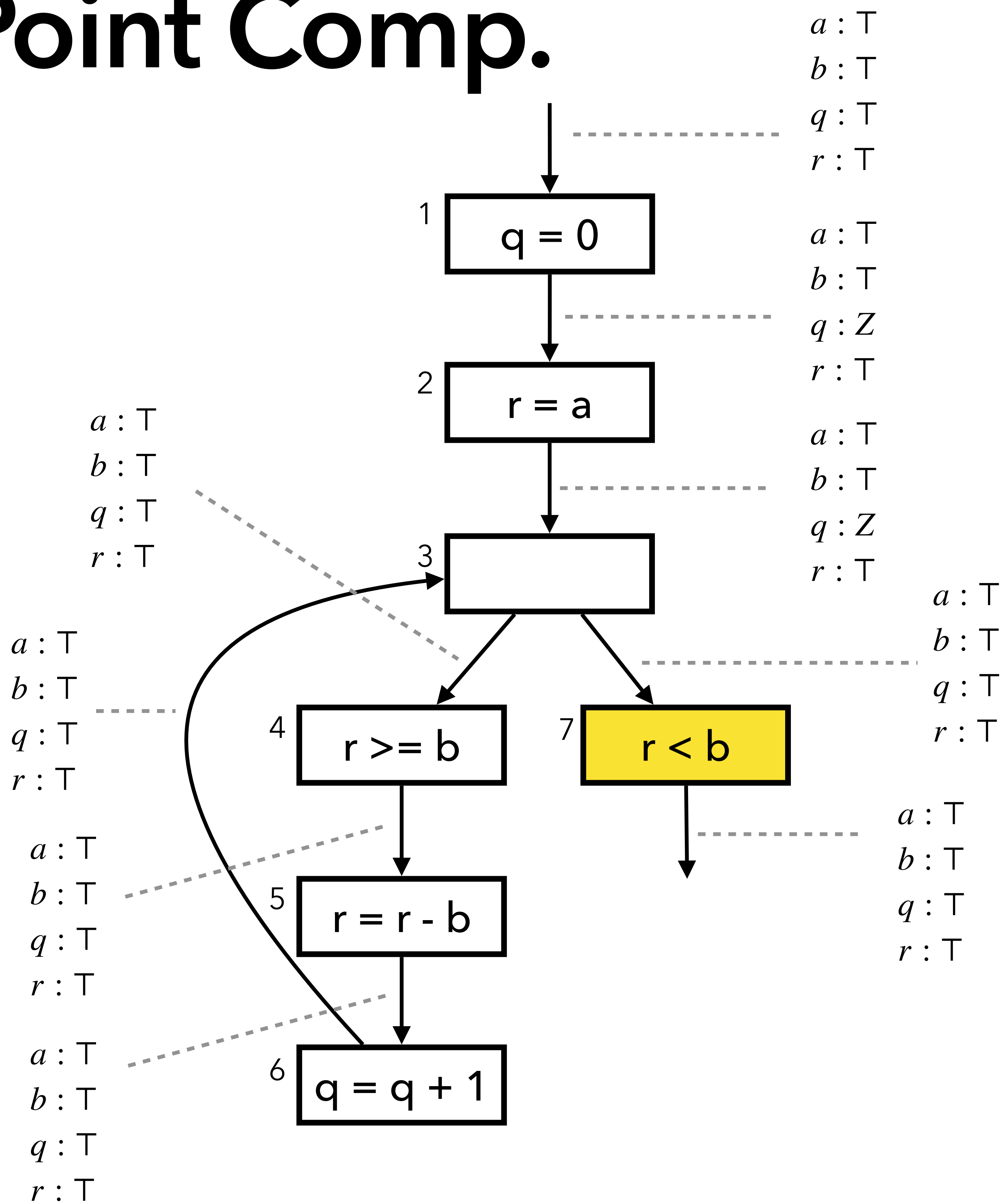
$$\begin{array}{l} a : \top \\ b : \top \\ q : Z \\ r : \top \end{array} \sqcup \begin{array}{l} a : \top \\ b : \top \\ q : \top \\ r : \top \end{array} = \begin{array}{l} a : \top \\ b : \top \\ q : \top \\ r : \top \end{array}$$

(fixed point)



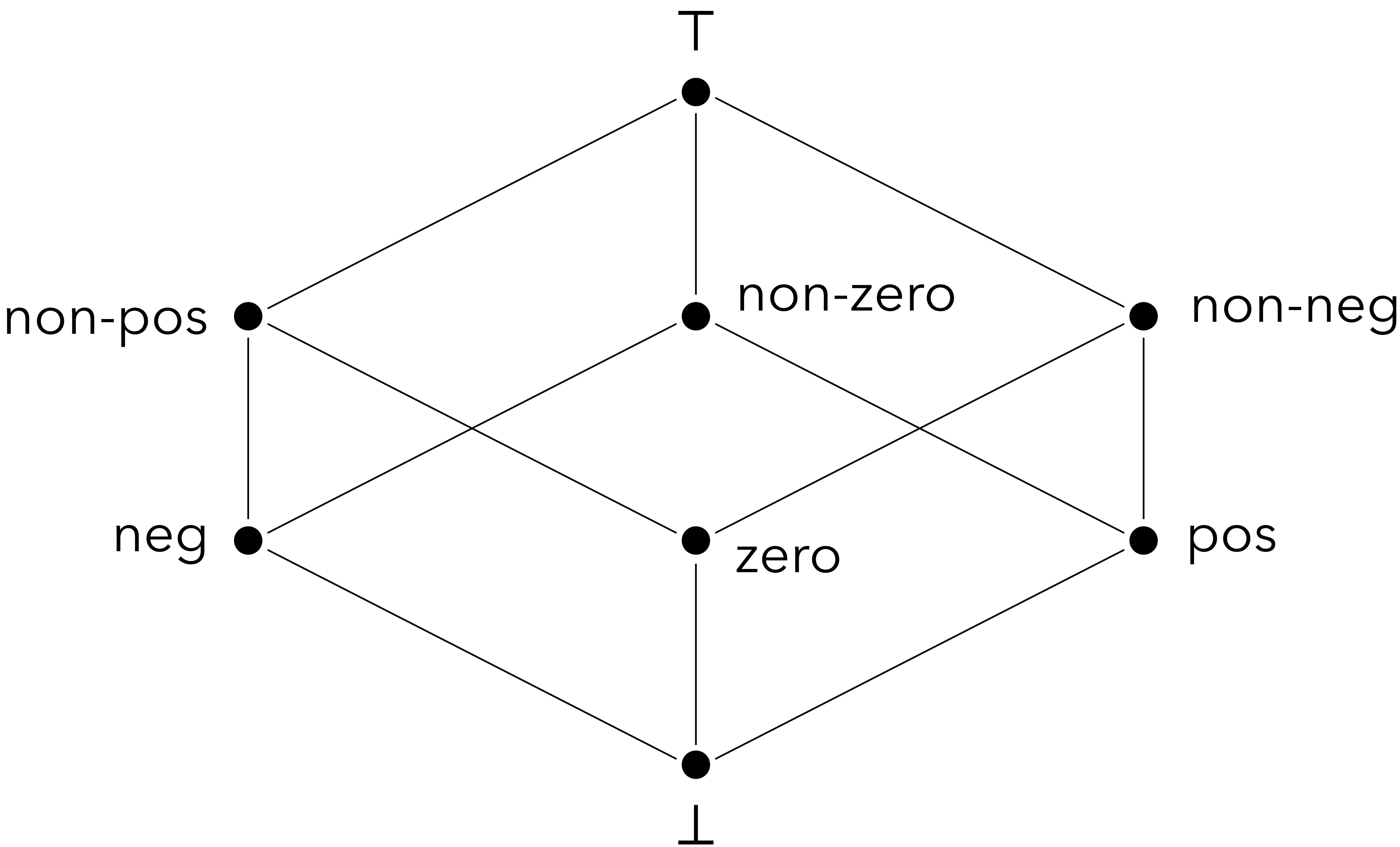
$W = \{ \cancel{1}, \cancel{2}, \cancel{3}, 4, \cancel{5}, \cancel{6}, 7 \}$

Fixed Point Comp.



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

An Extended Sign Domain



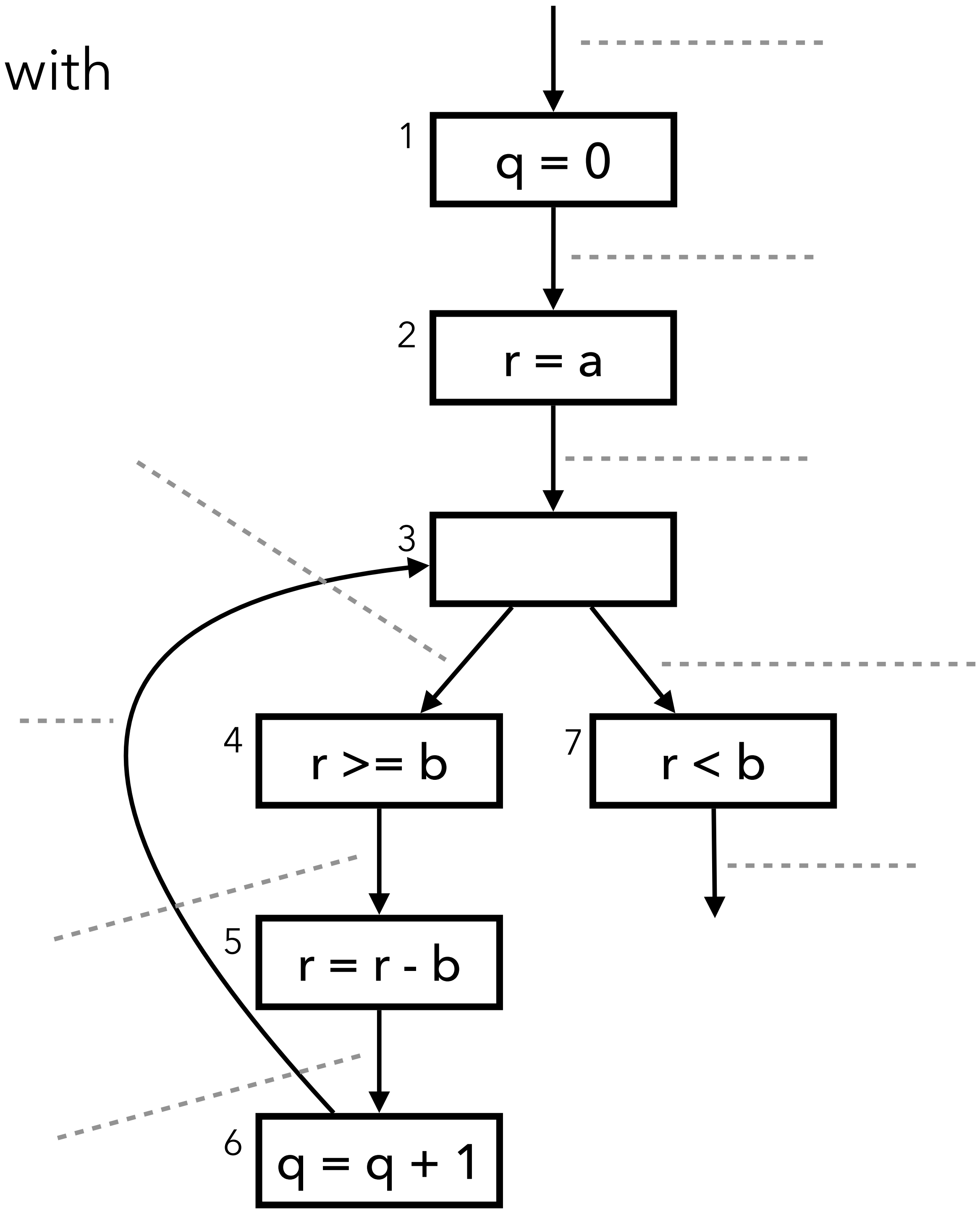
+	top	neg	zero	pos	non-pos	non-zero	non-neg	bot
top								
neg								
zero								
pos								
non-pos								
non-zero								
non-neg								
bot								

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

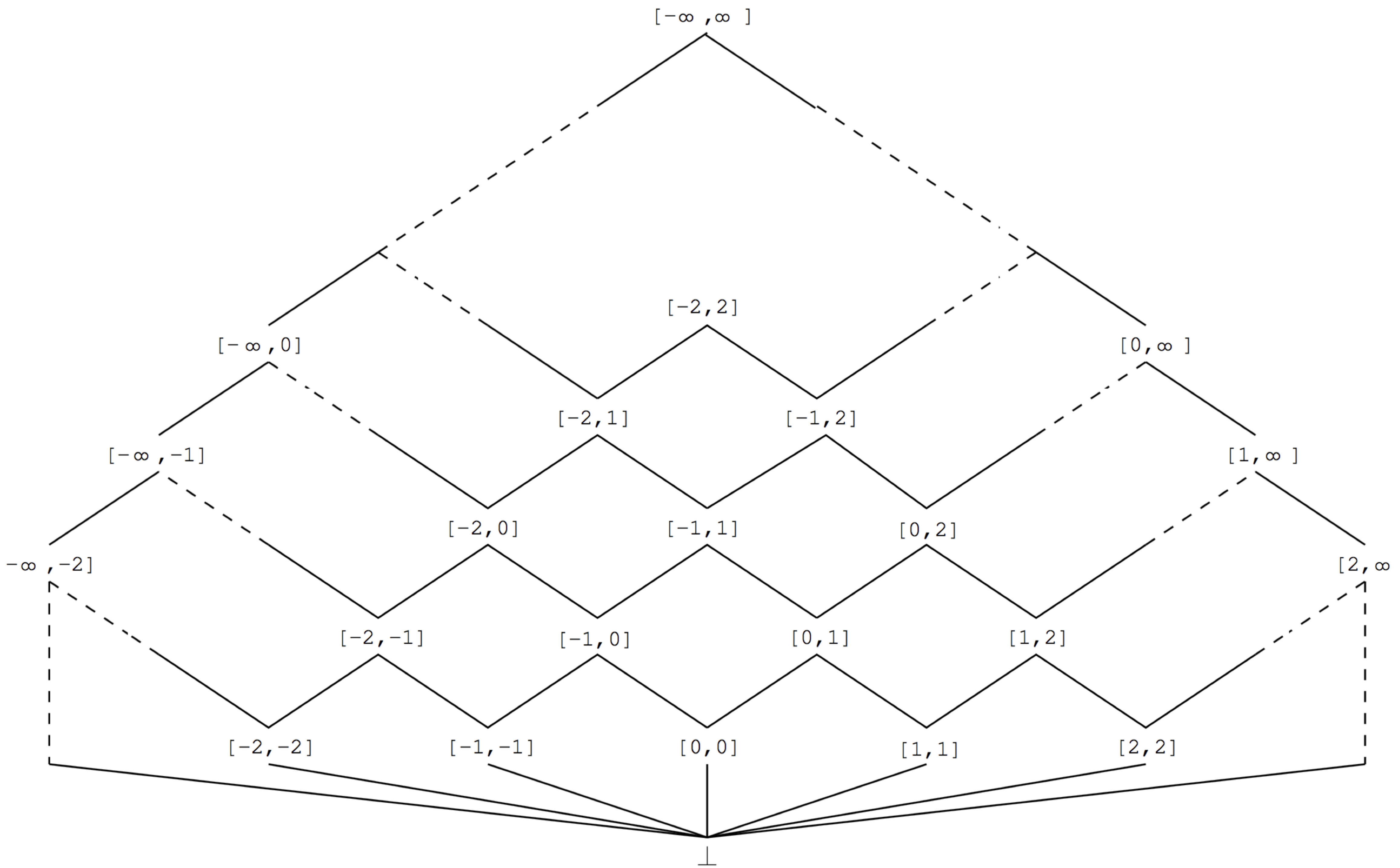
Exercise (1)

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);
```

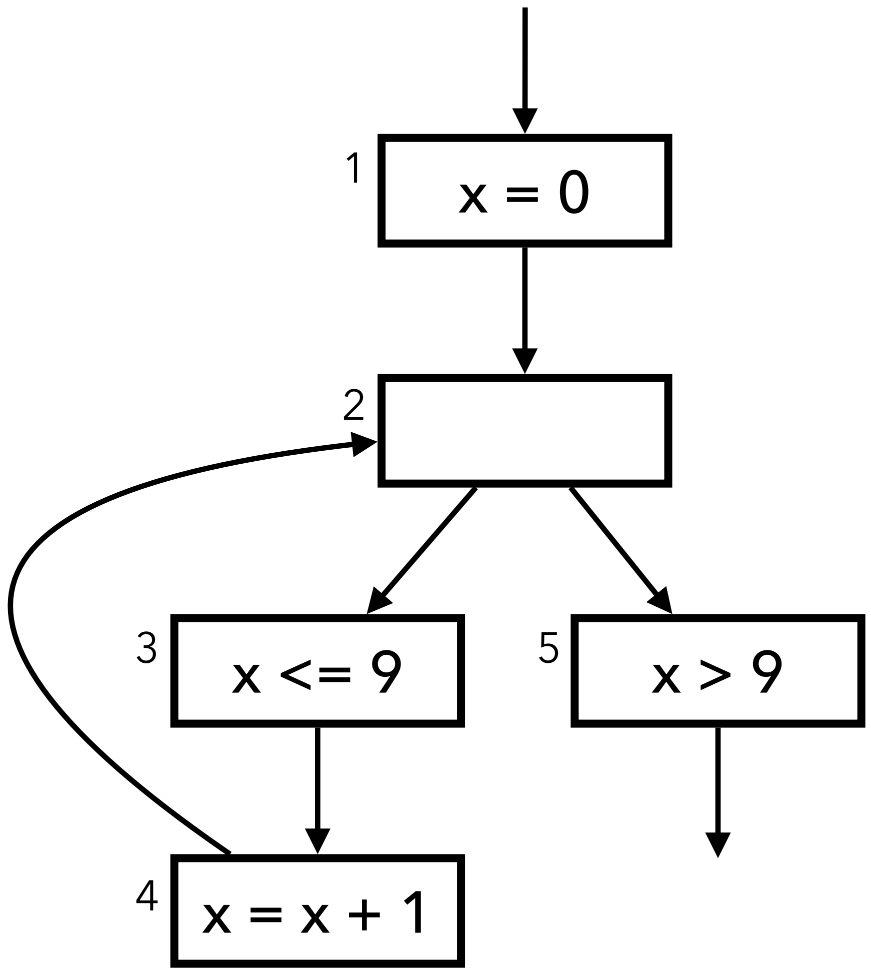


The Interval Domain



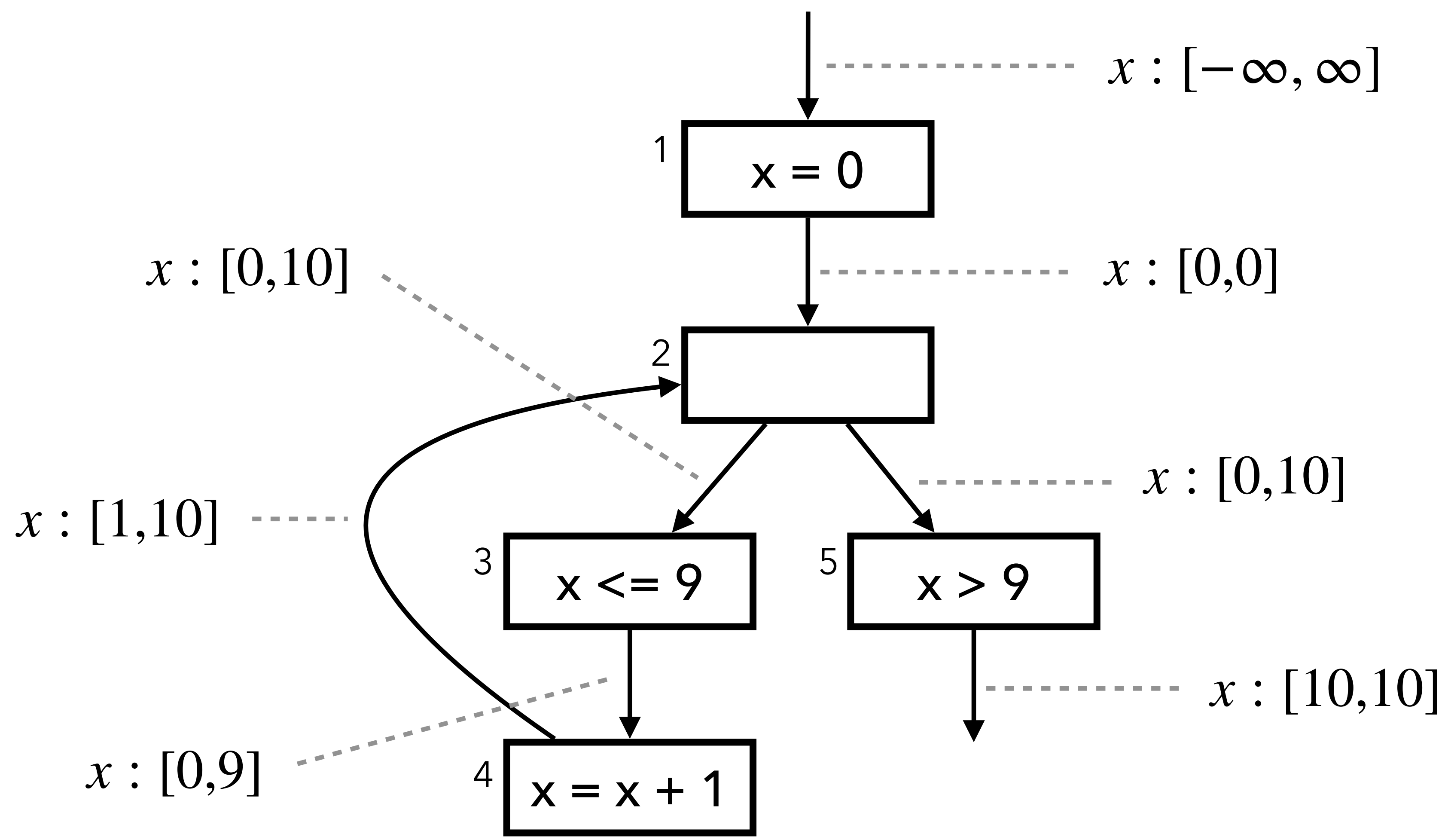
Example Program

```
x = 0;

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

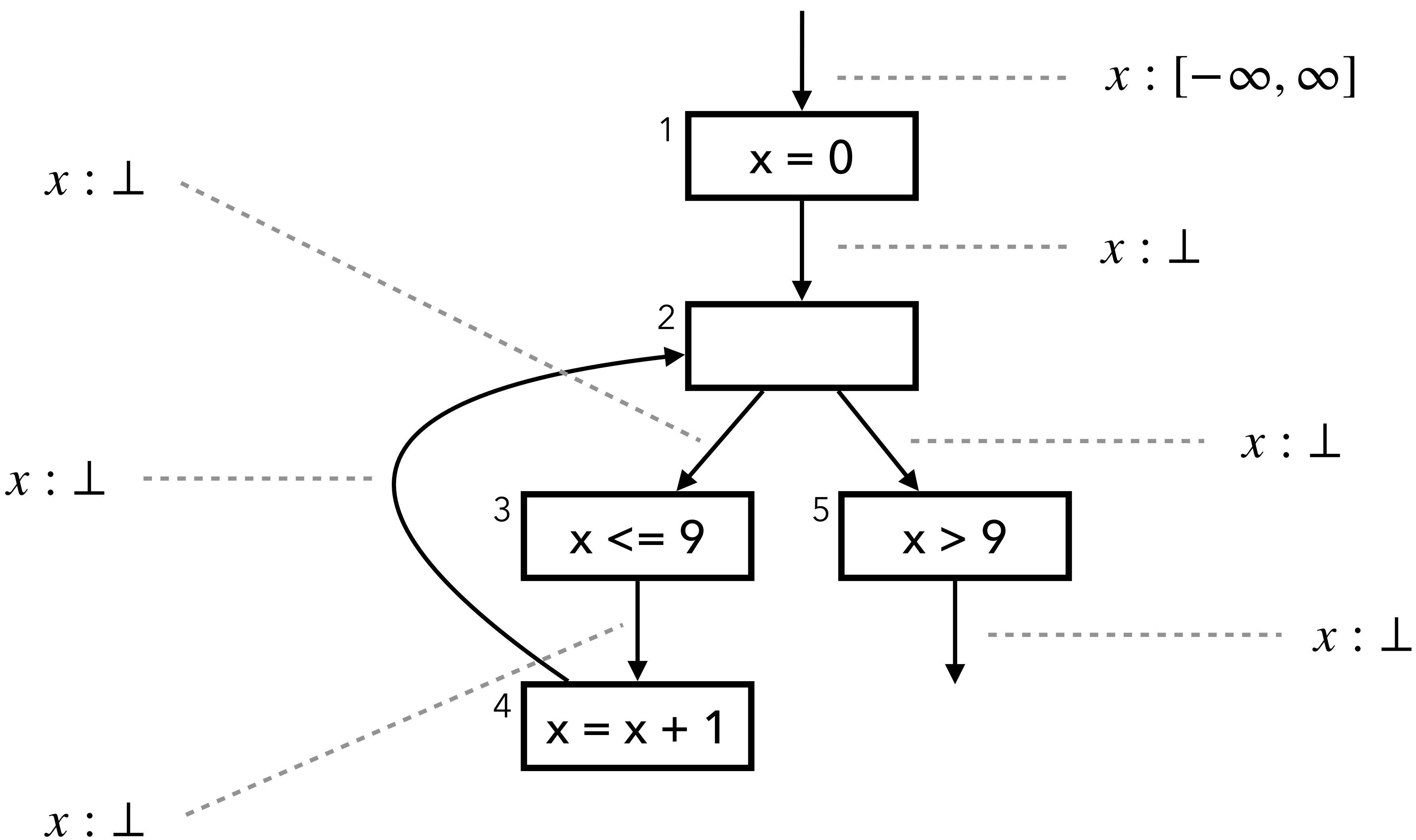


Example Program

```
x = 0;

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

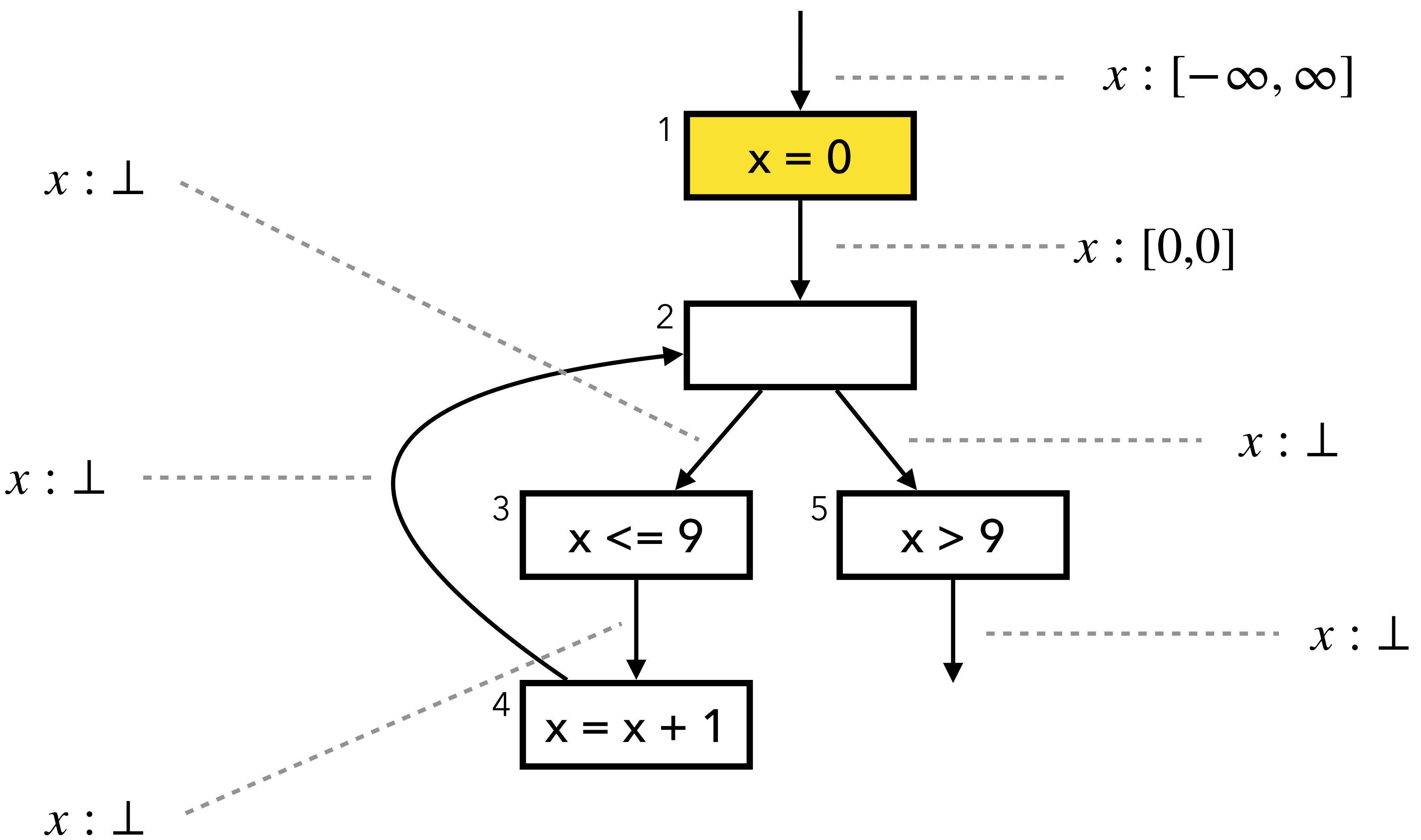


Fixed Point Computation

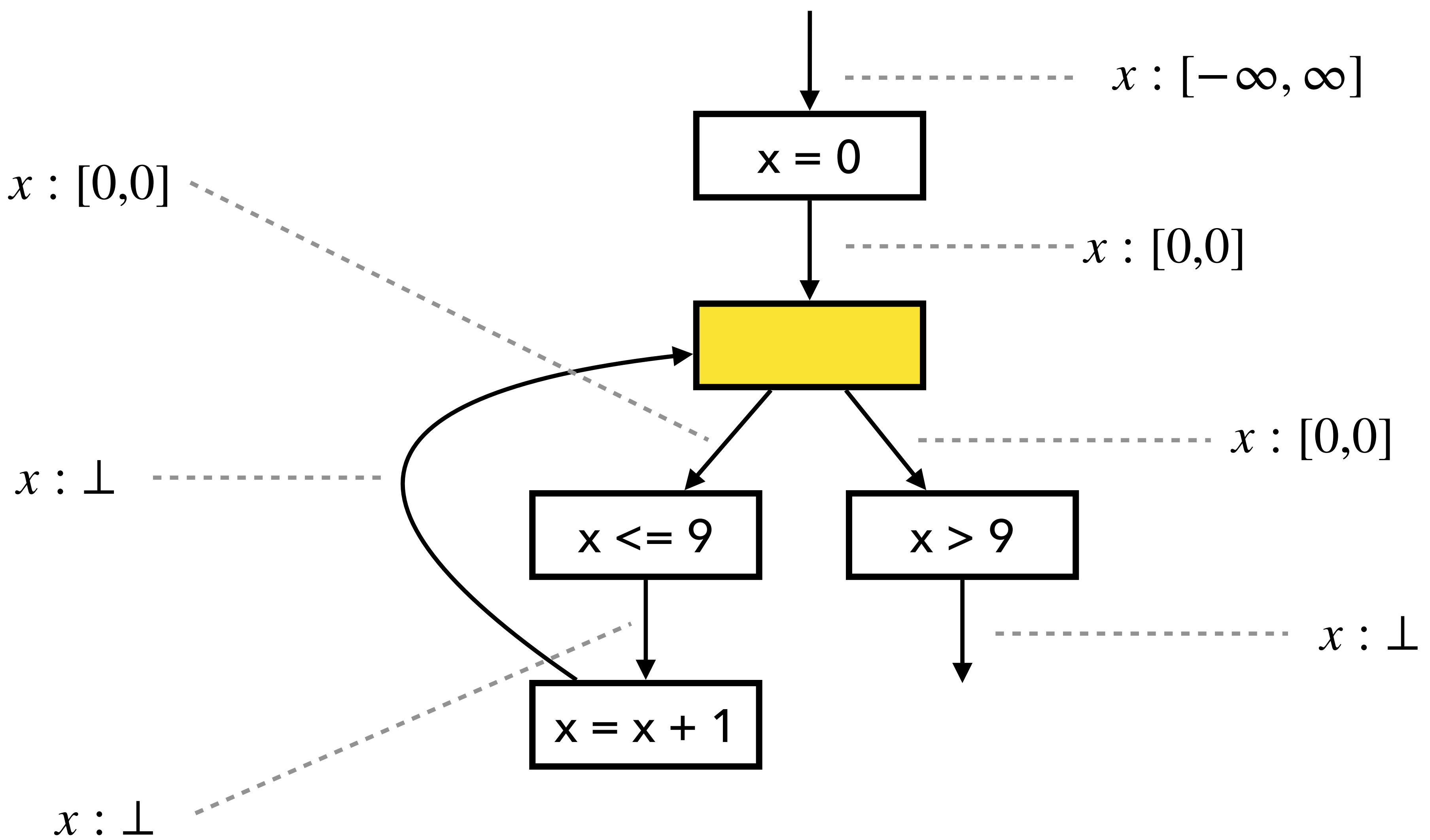


Initial states

Fixed Point Computation

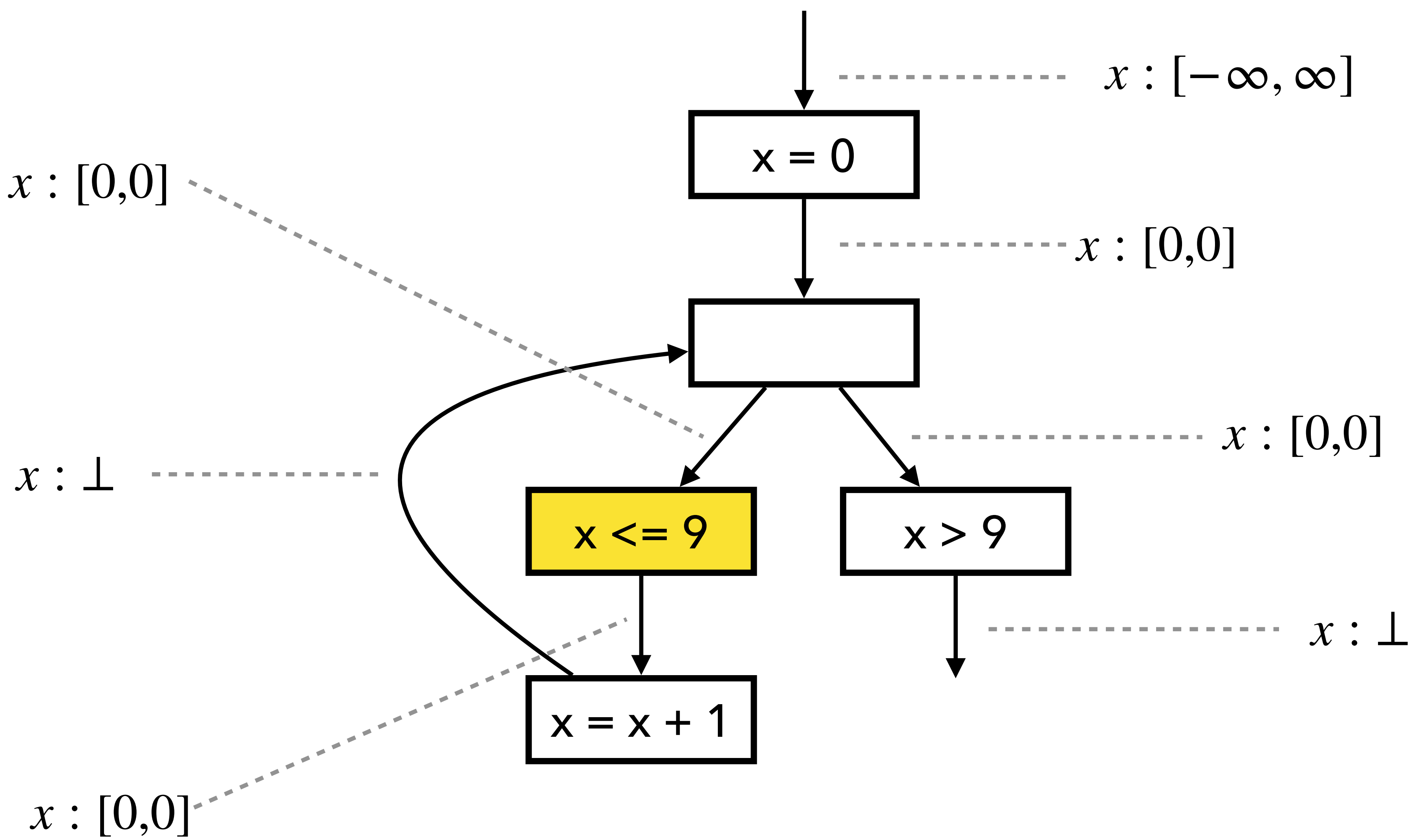


Fixed Point Computation



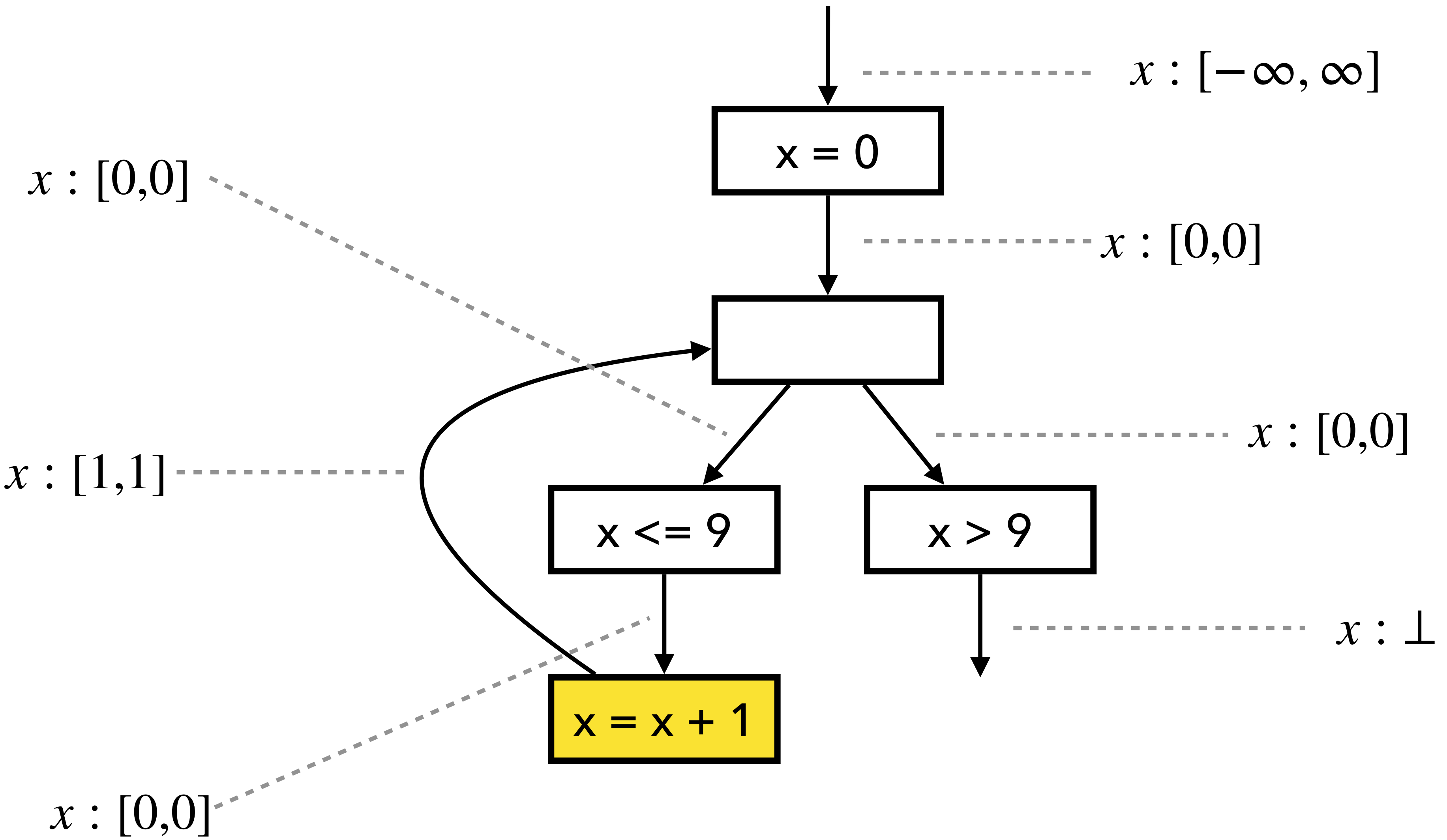
Input state: $[0,0] \sqcup \perp = [0,0]$

Fixed Point Computation

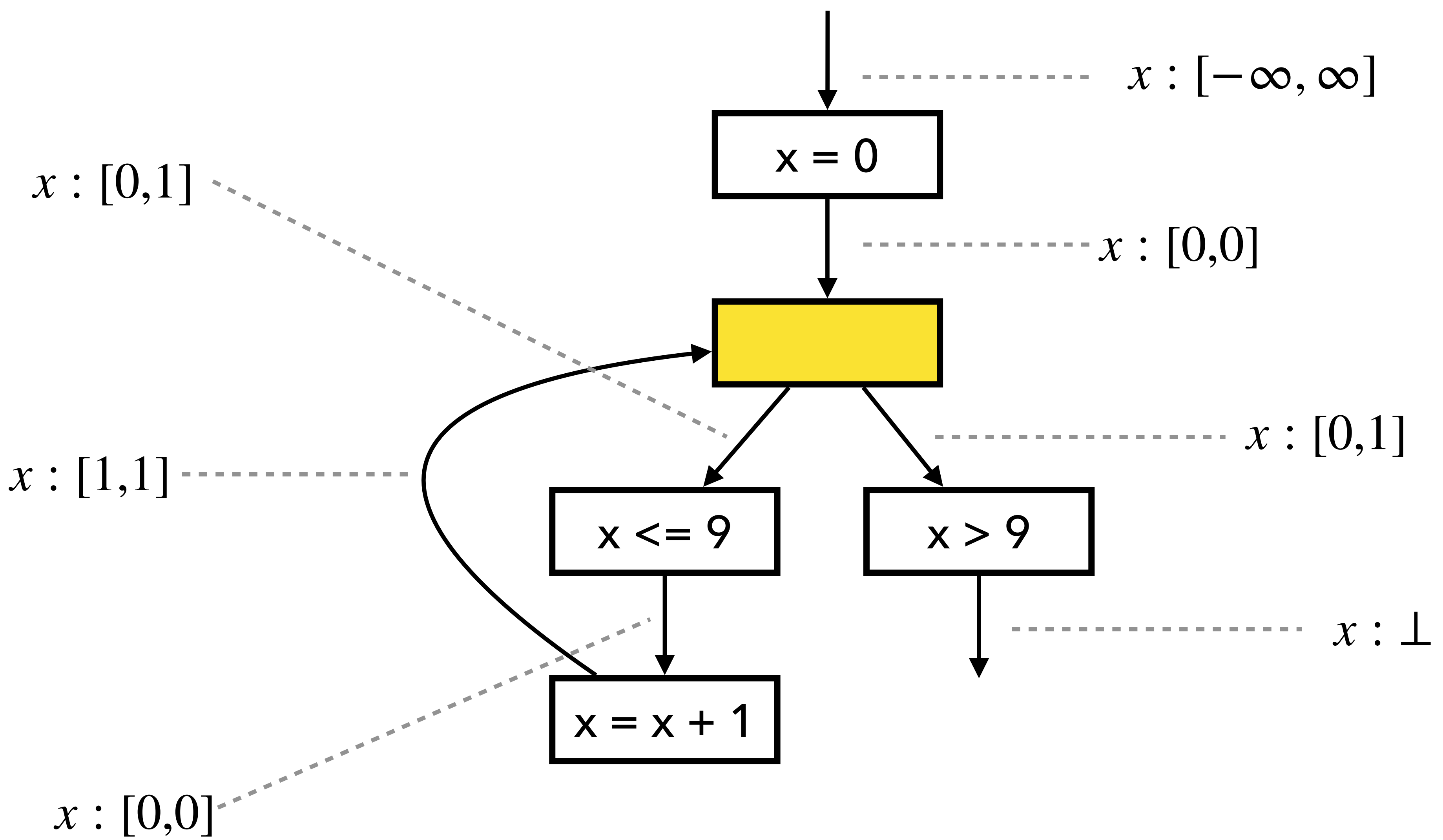


$$[0,0] \sqcap [-\infty,9] = [0,0]$$

Fixed Point Computation

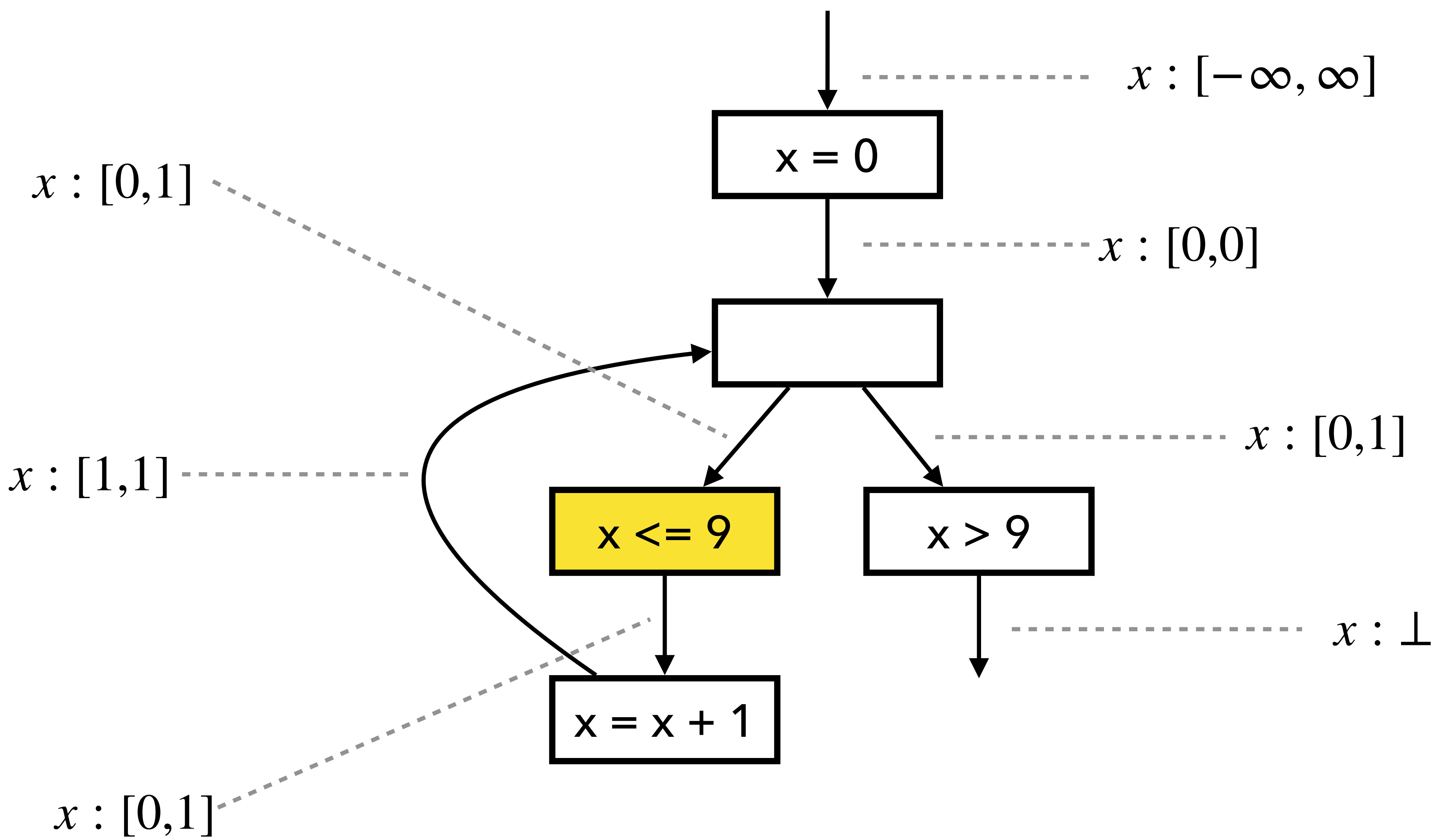


Fixed Point Computation



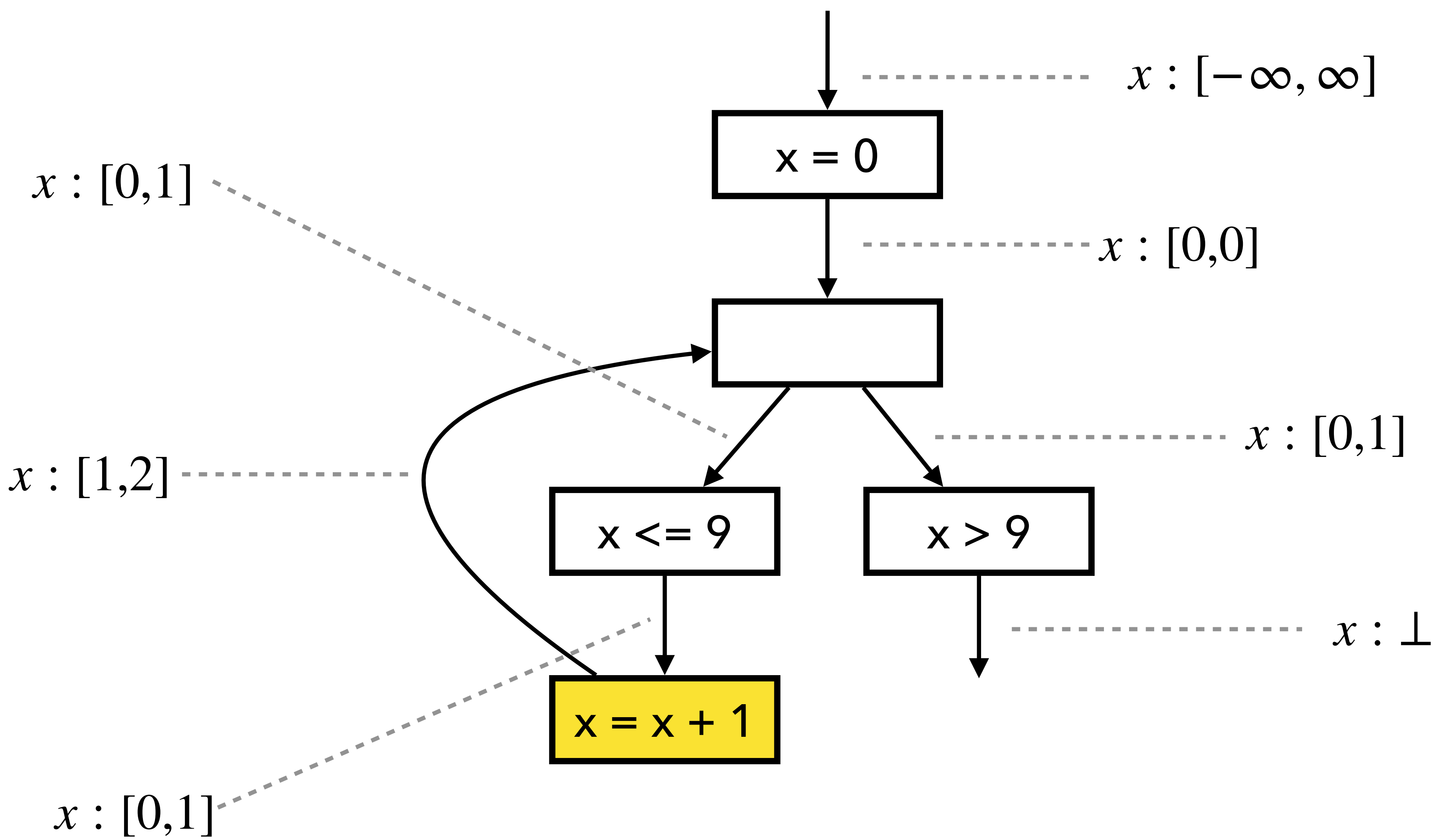
Input state: $[0,0] \sqcup [1,1] = [0,1]$
(1st iteration of loop)

Fixed Point Computation

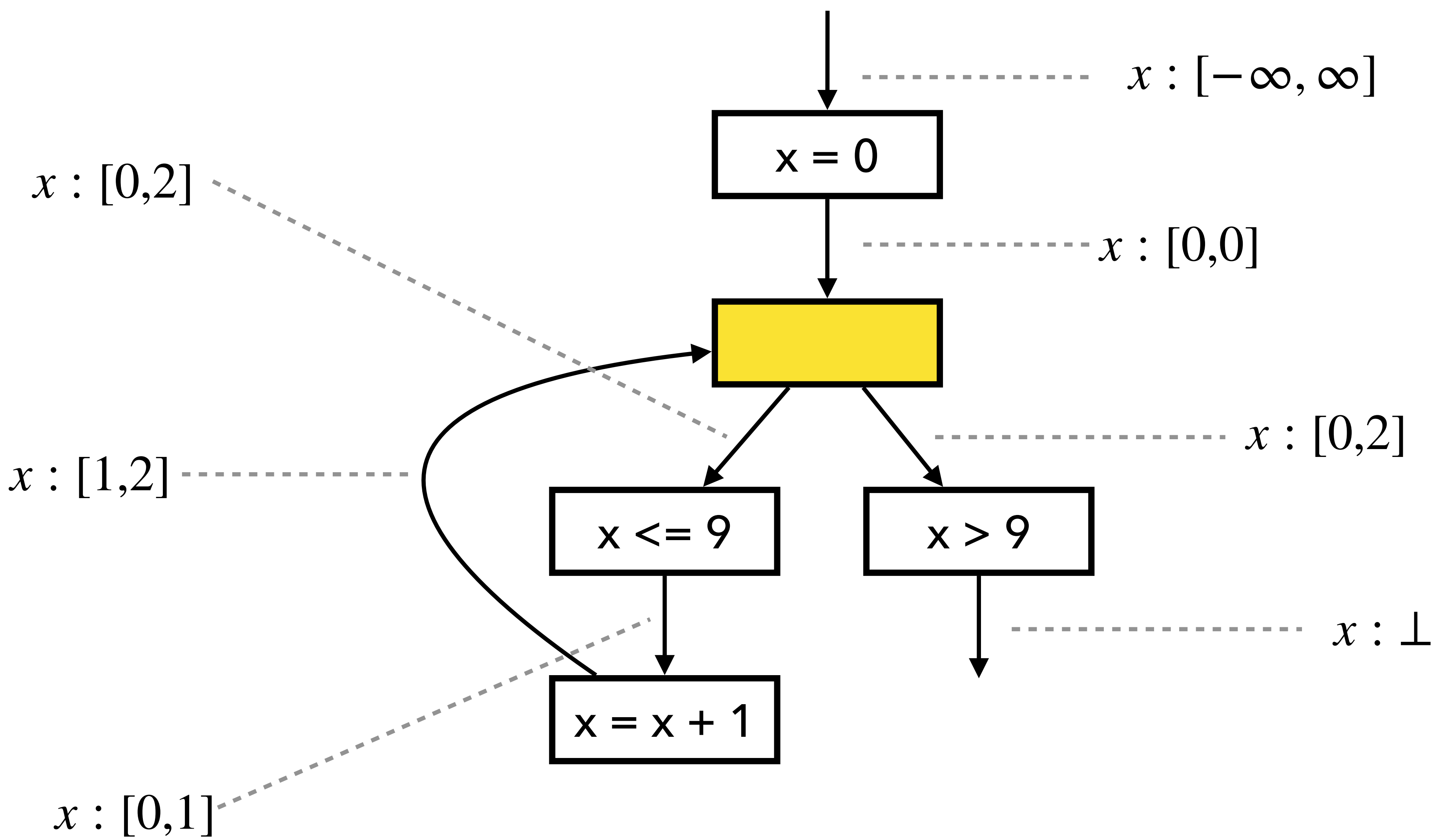


$$[0,1] \sqcap [-\infty,9] = [0,1]$$

Fixed Point Computation

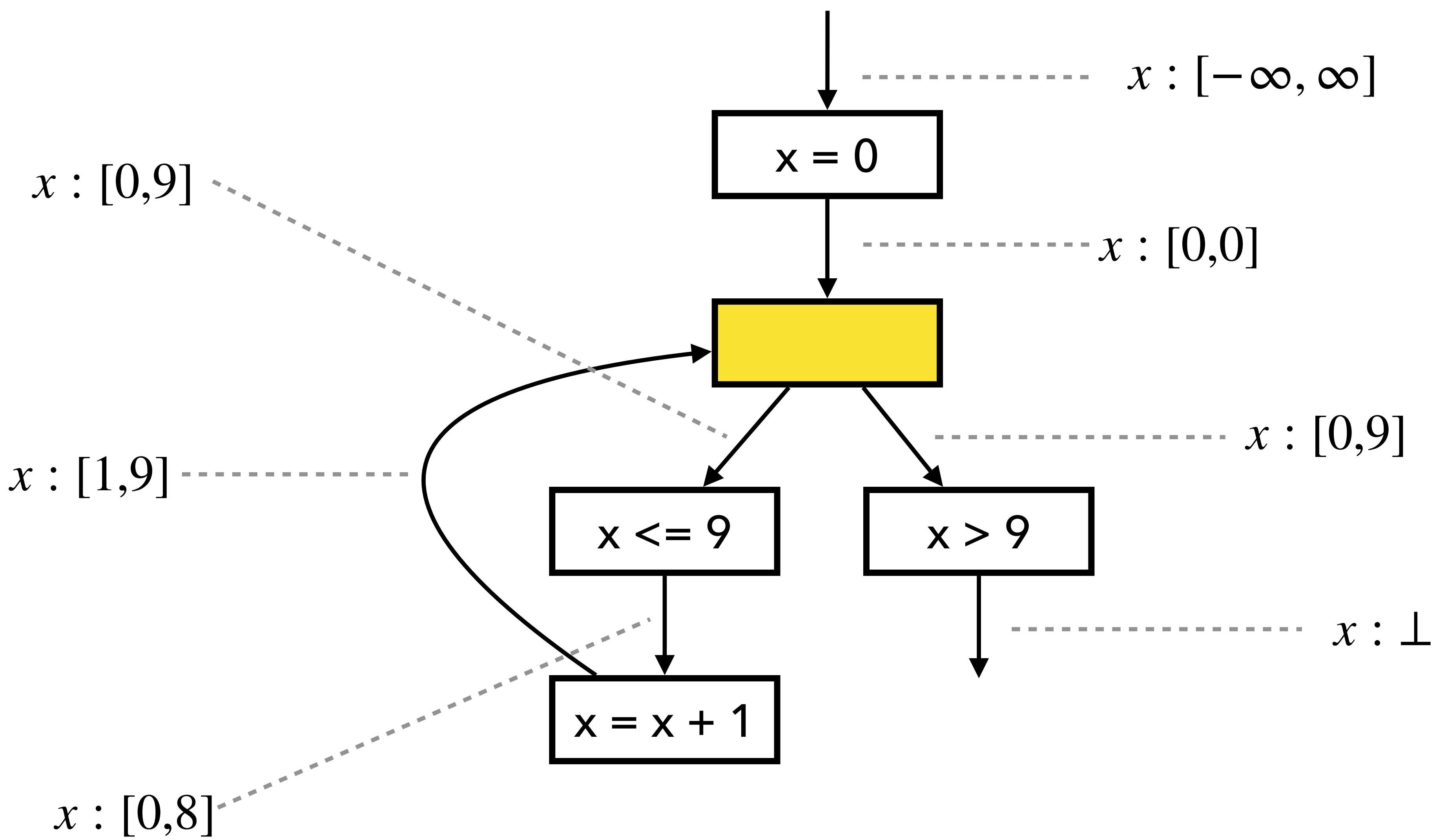


Fixed Point Computation



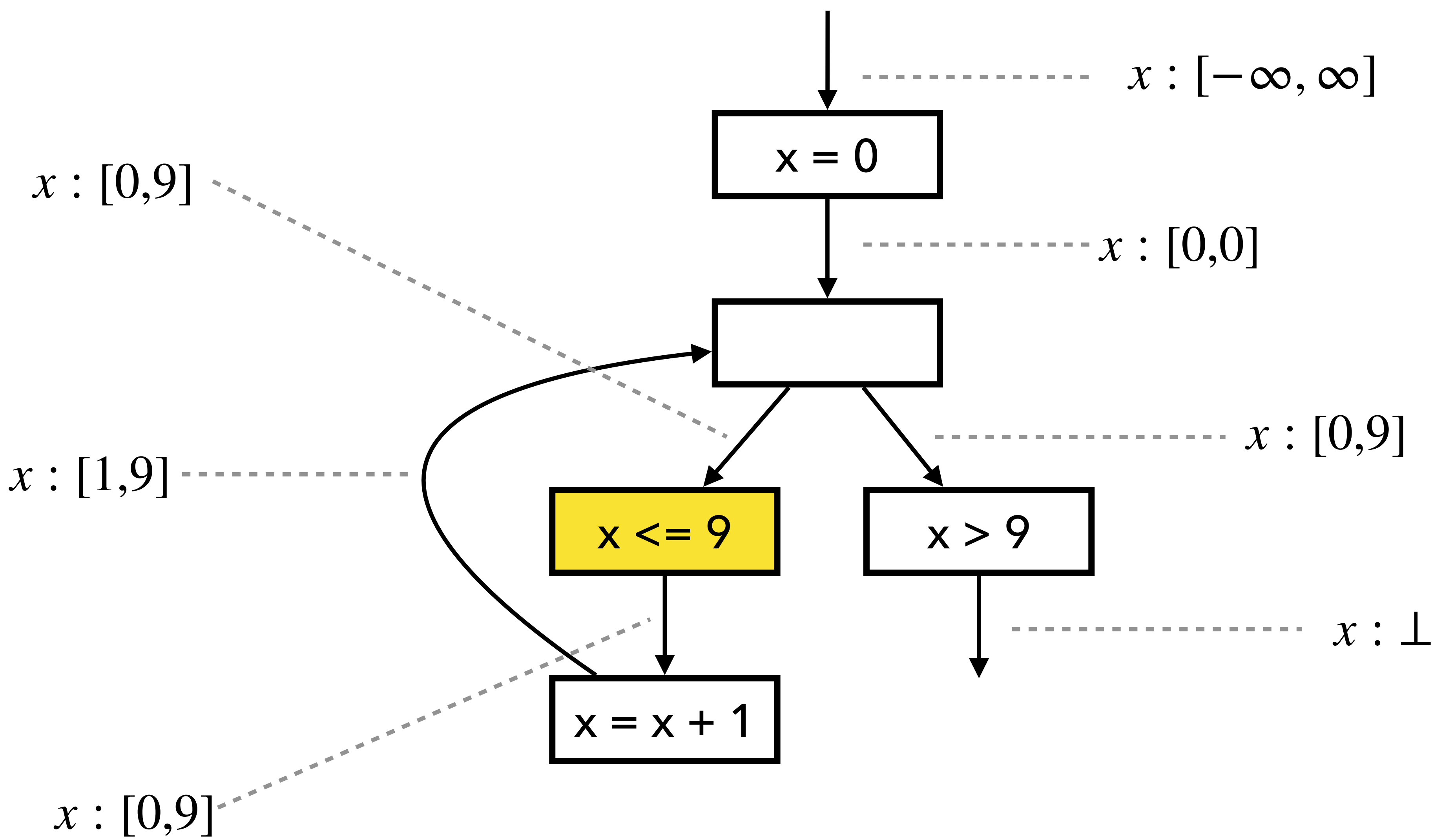
Input state: $[0,0] \sqcup [1,2] = [0,2]$
(2nd iteration of loop)

Fixed Point Computation



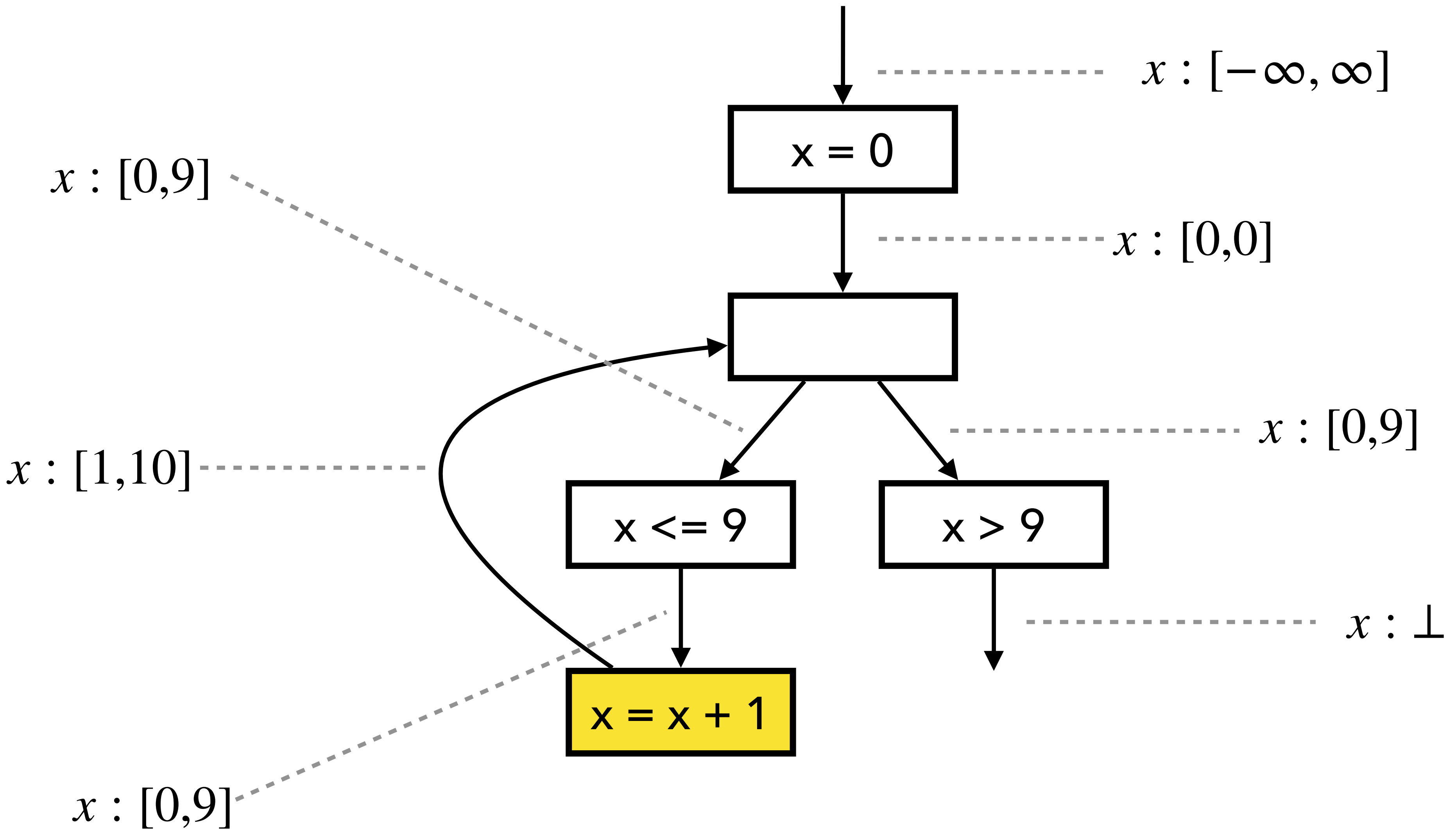
Input state: $[0,0] \sqcup [1,9] = [0,9]$
(9th iteration of loop)

Fixed Point Computation

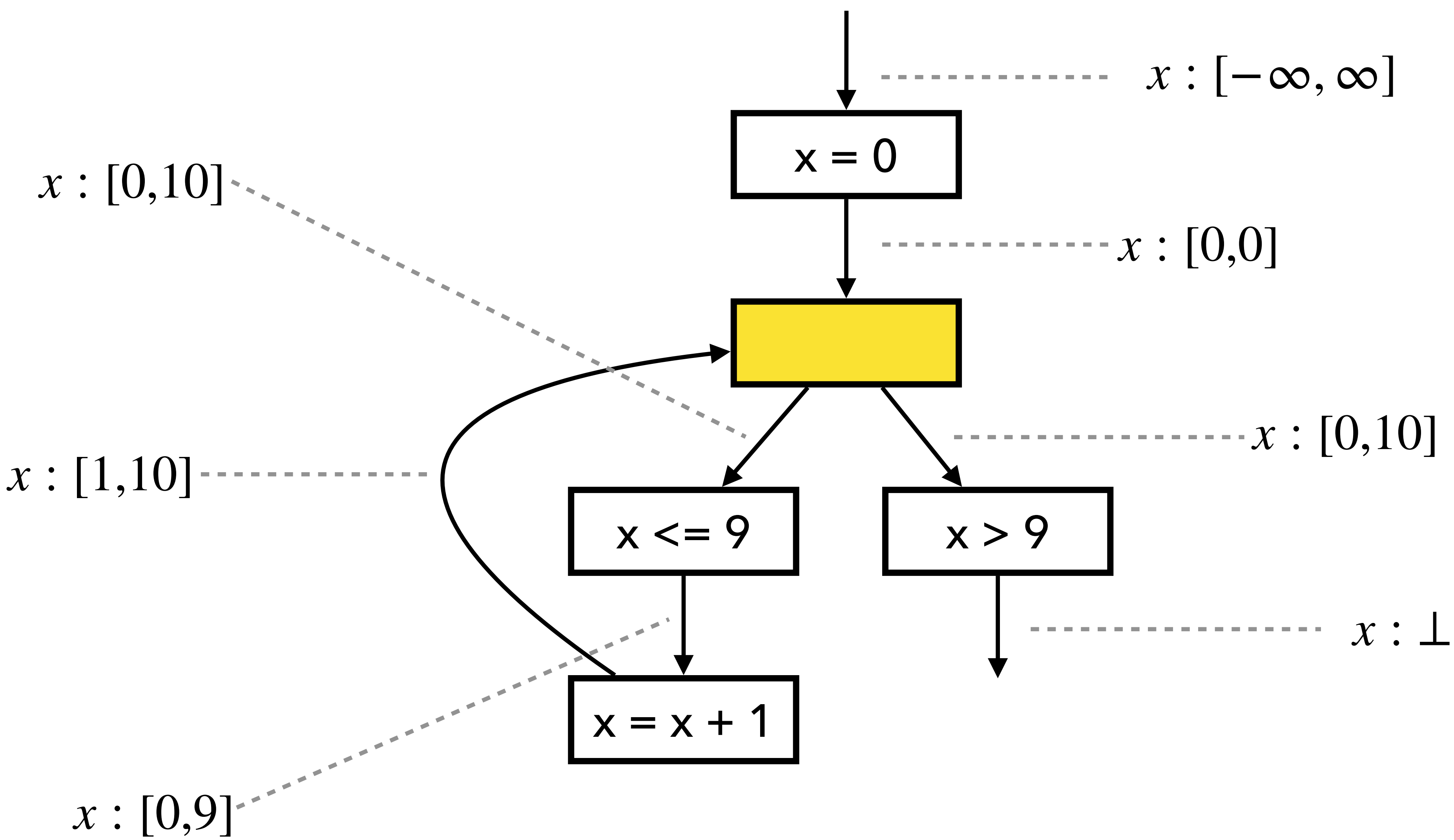


$$[0,9] \sqcap [-\infty,9] = [0,9]$$

Fixed Point Computation

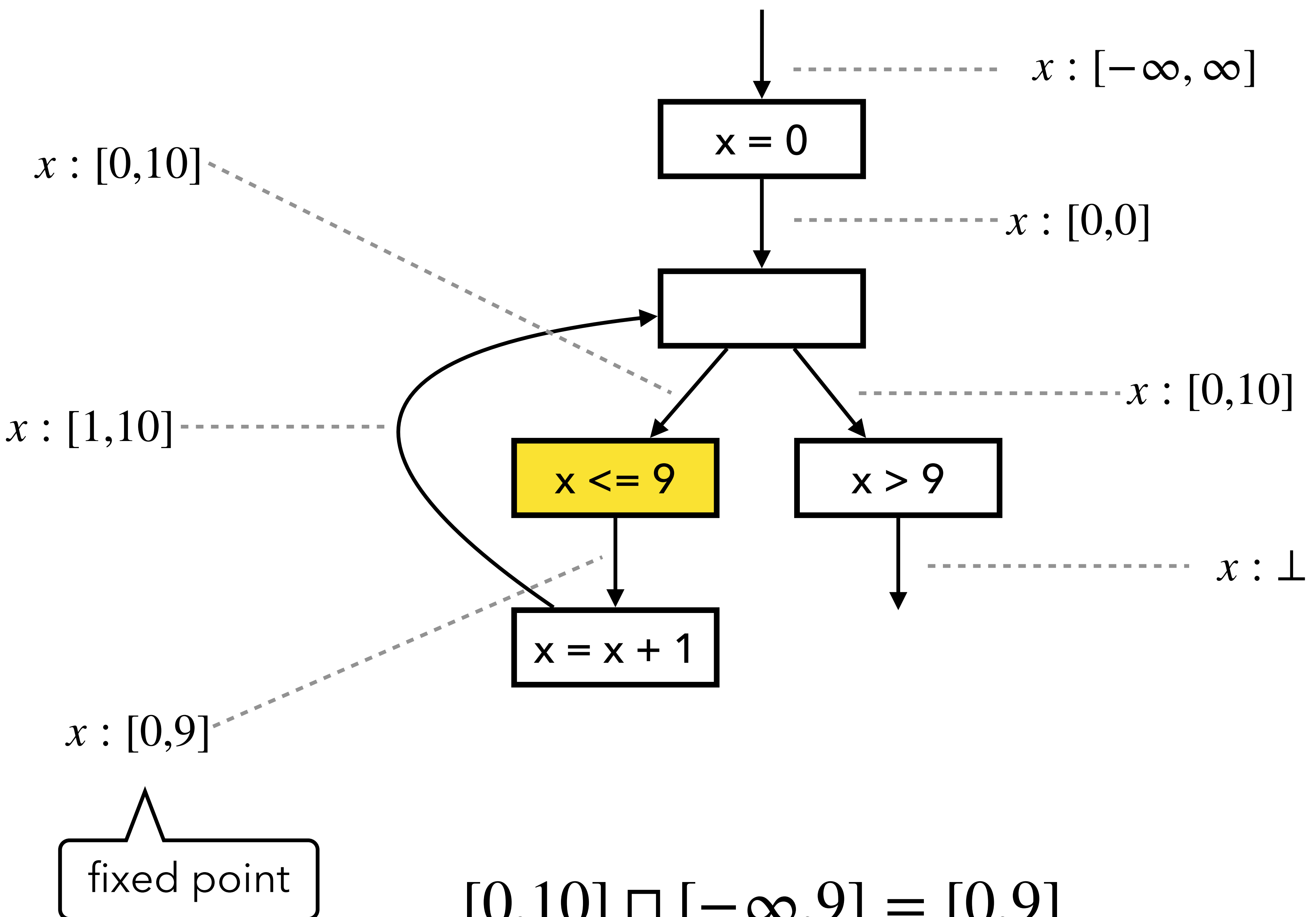


Fixed Point Computation



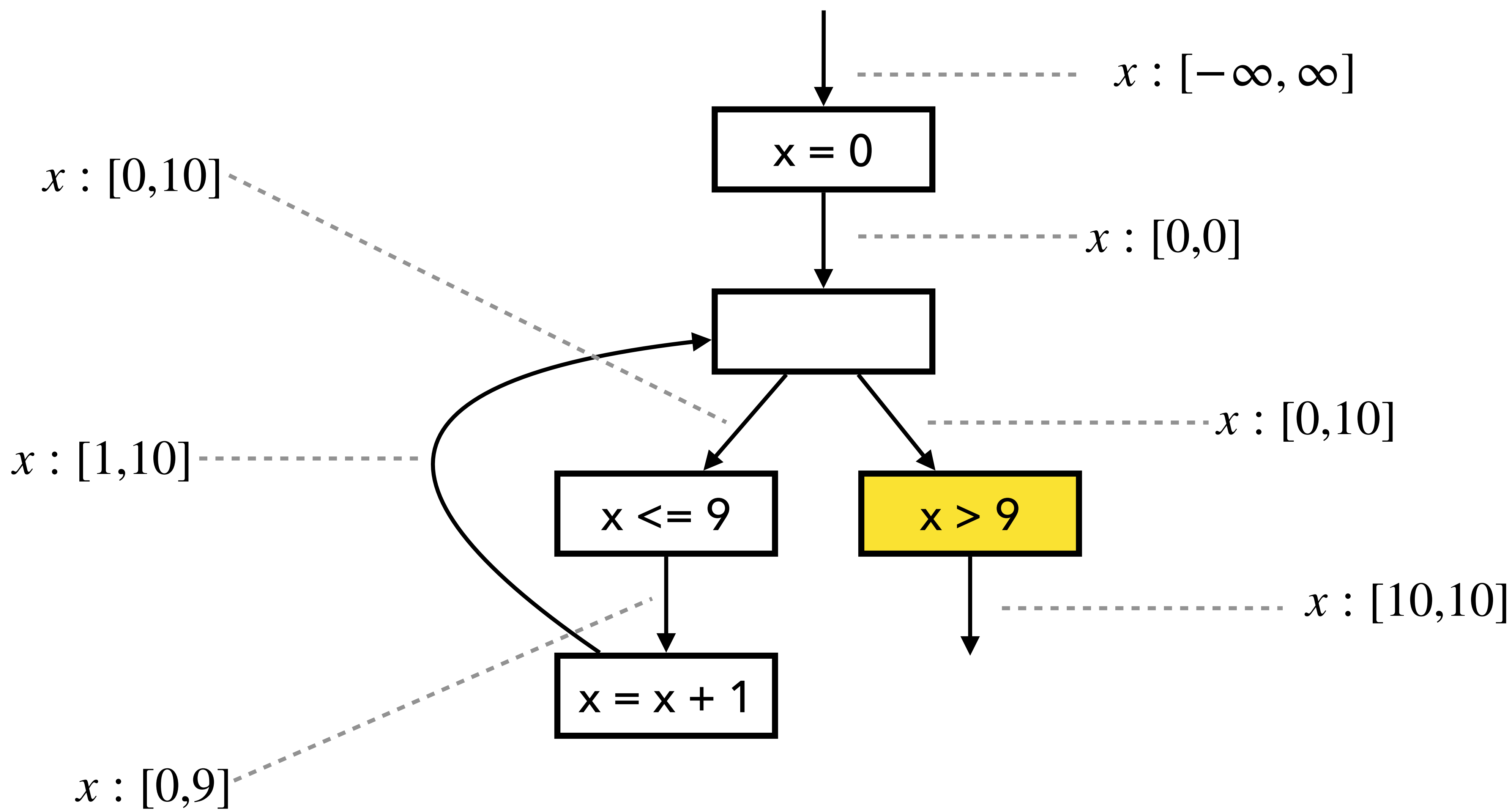
Input state: $[0,0] \sqcup [1,10] = [0,10]$
(10th iteration of loop)

Fixed Point Computation



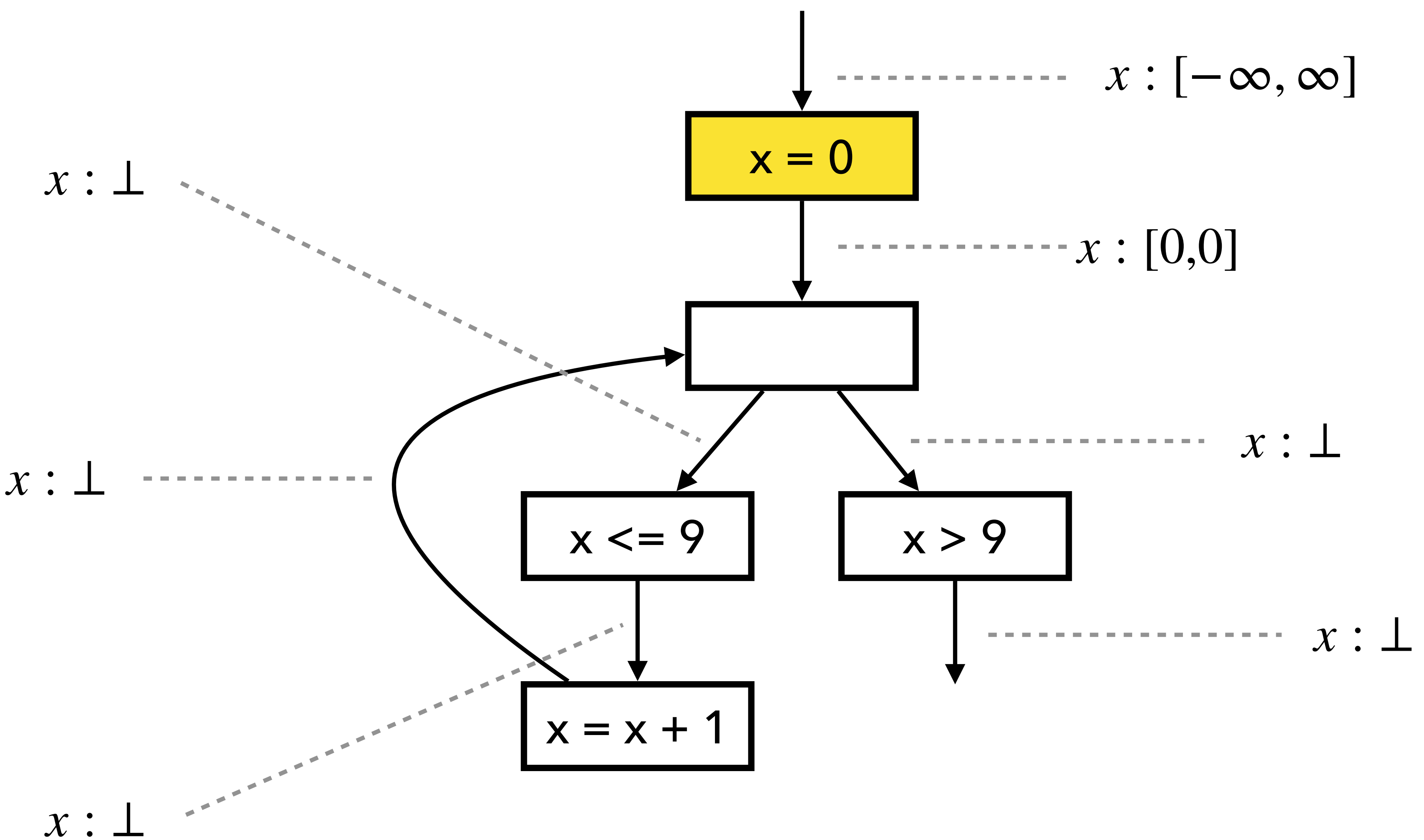
$$[0,10] \sqcap [-\infty,9] = [0,9]$$

Fixed Point Computation

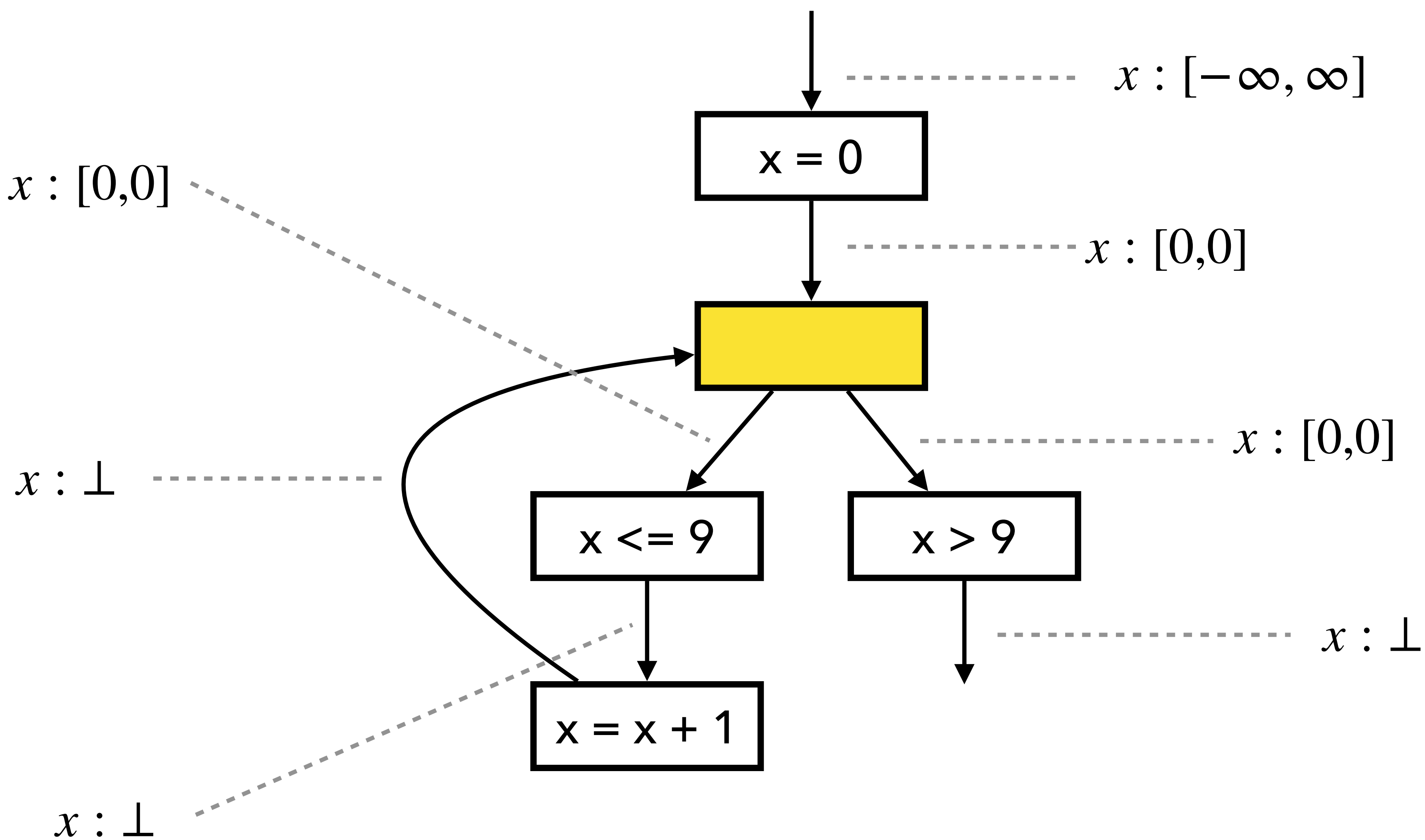


$$[0,10] \sqcap [10,\infty] = [10,10]$$

Fixed Point Comp. with Widening

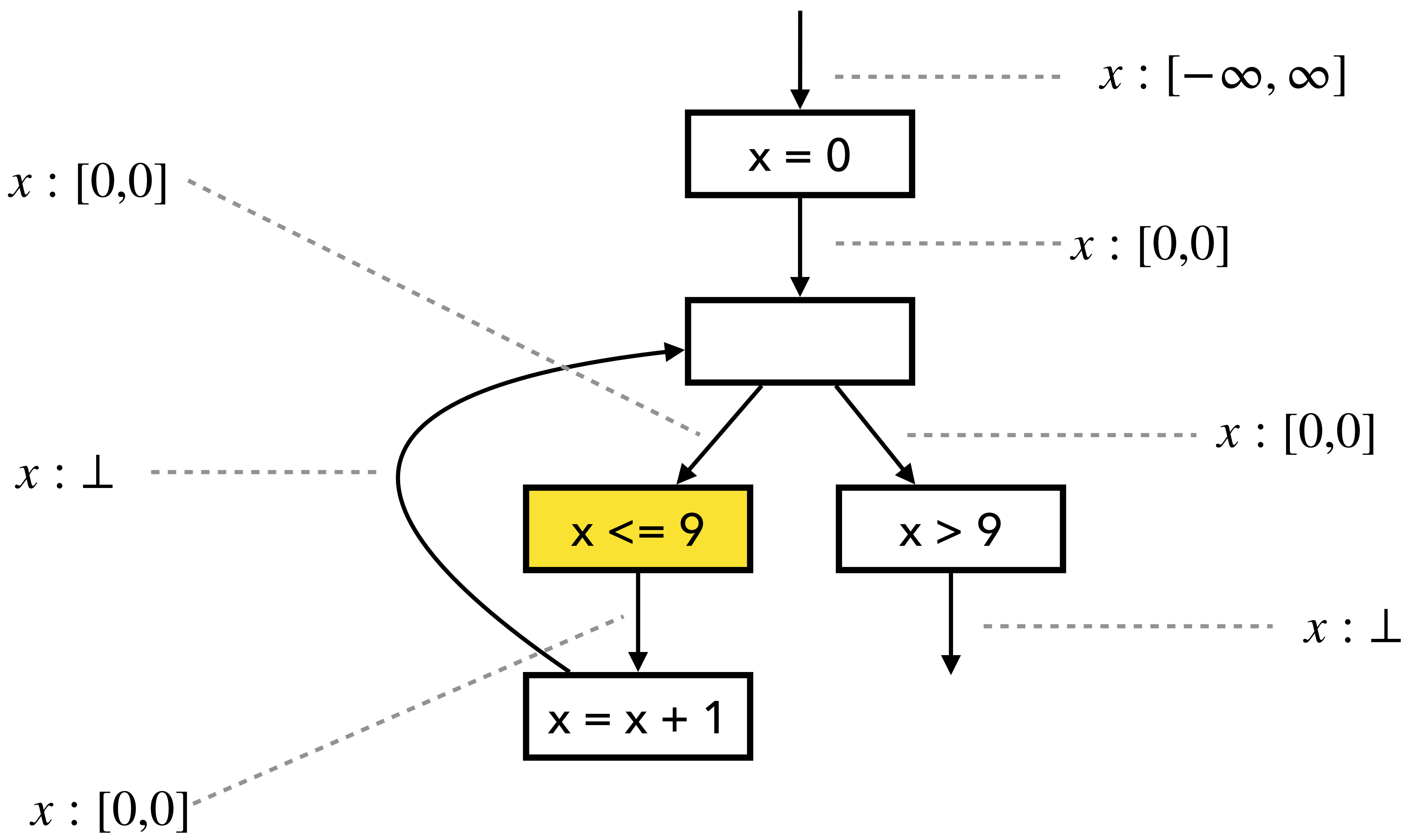


Fixed Point Comp. with Widening



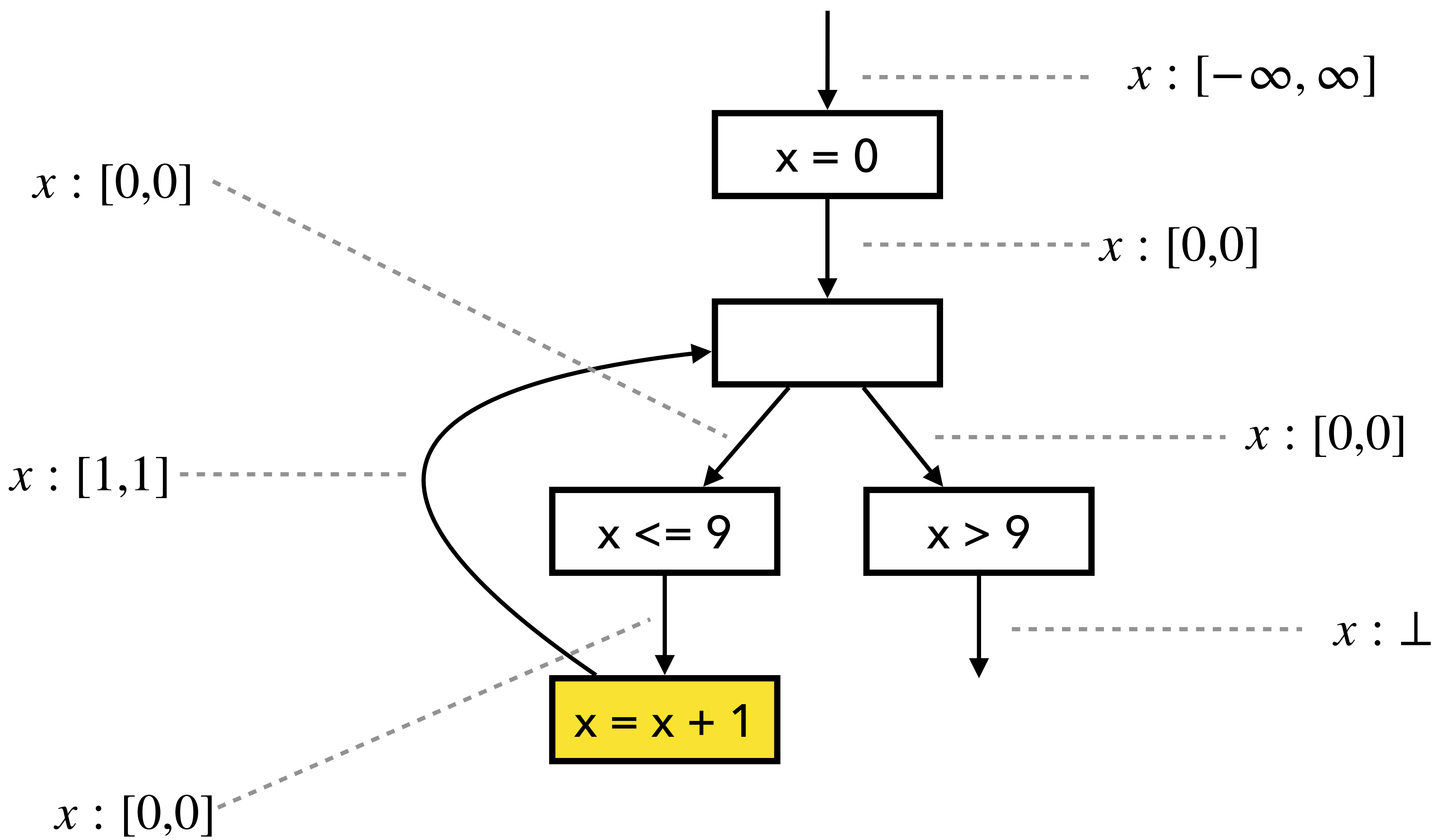
Input state: $[0,0] \sqcup \perp = [0,0]$

Fixed Point Comp. with Widening



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

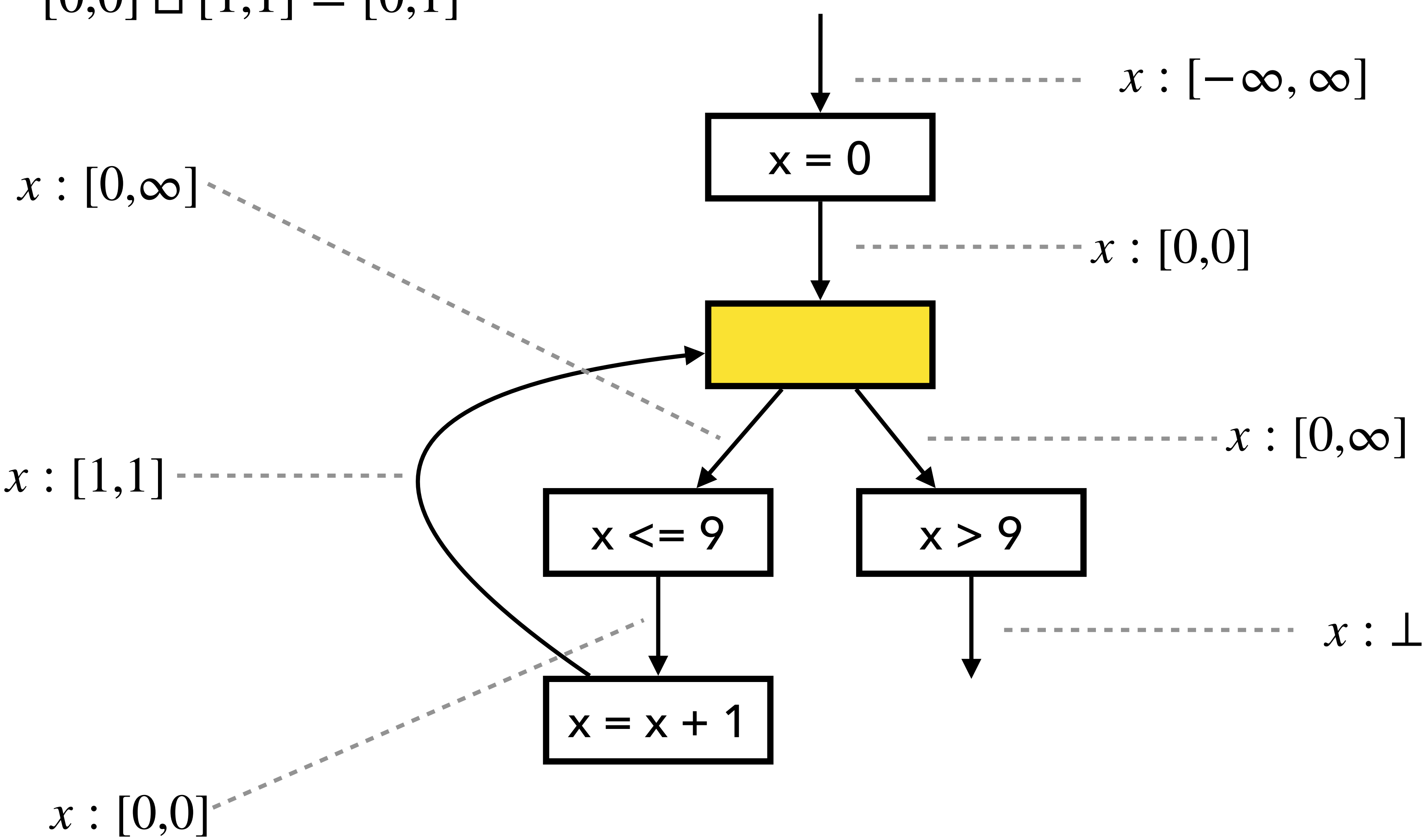
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

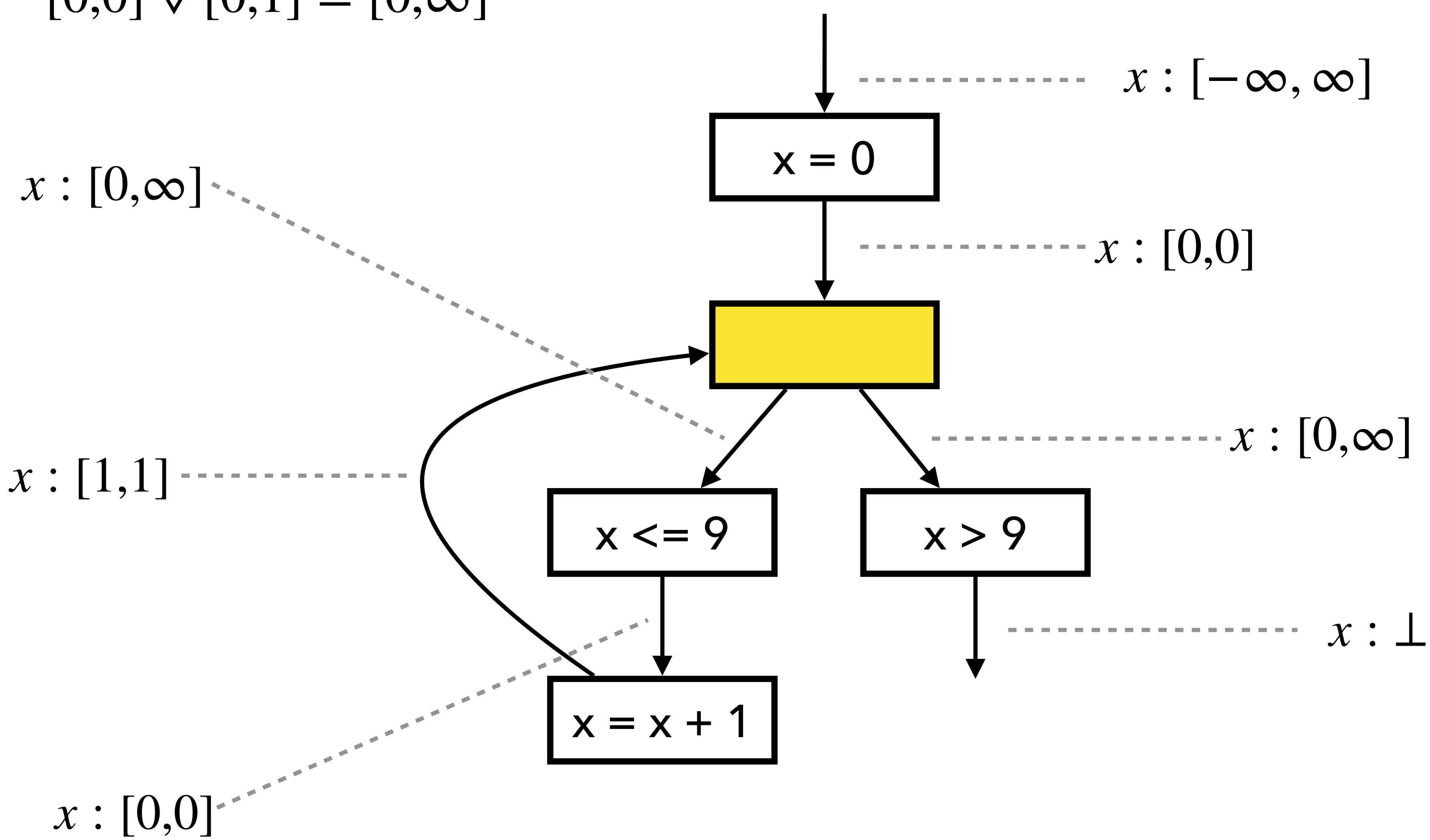
$$[0,0] \sqcup [1,1] = [0,1]$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

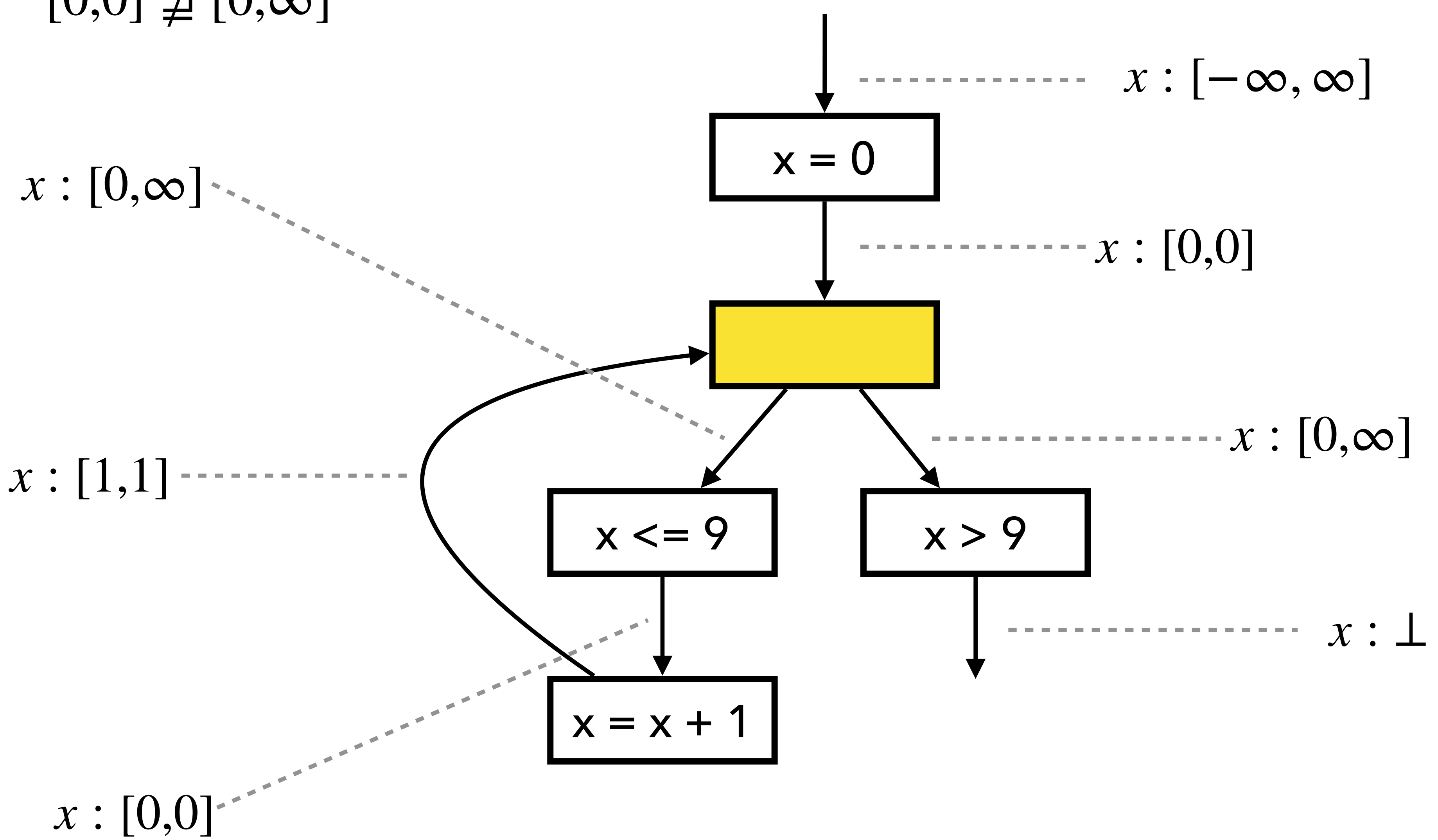
$$[0,0] \triangledown [0,1] = [0,\infty]$$



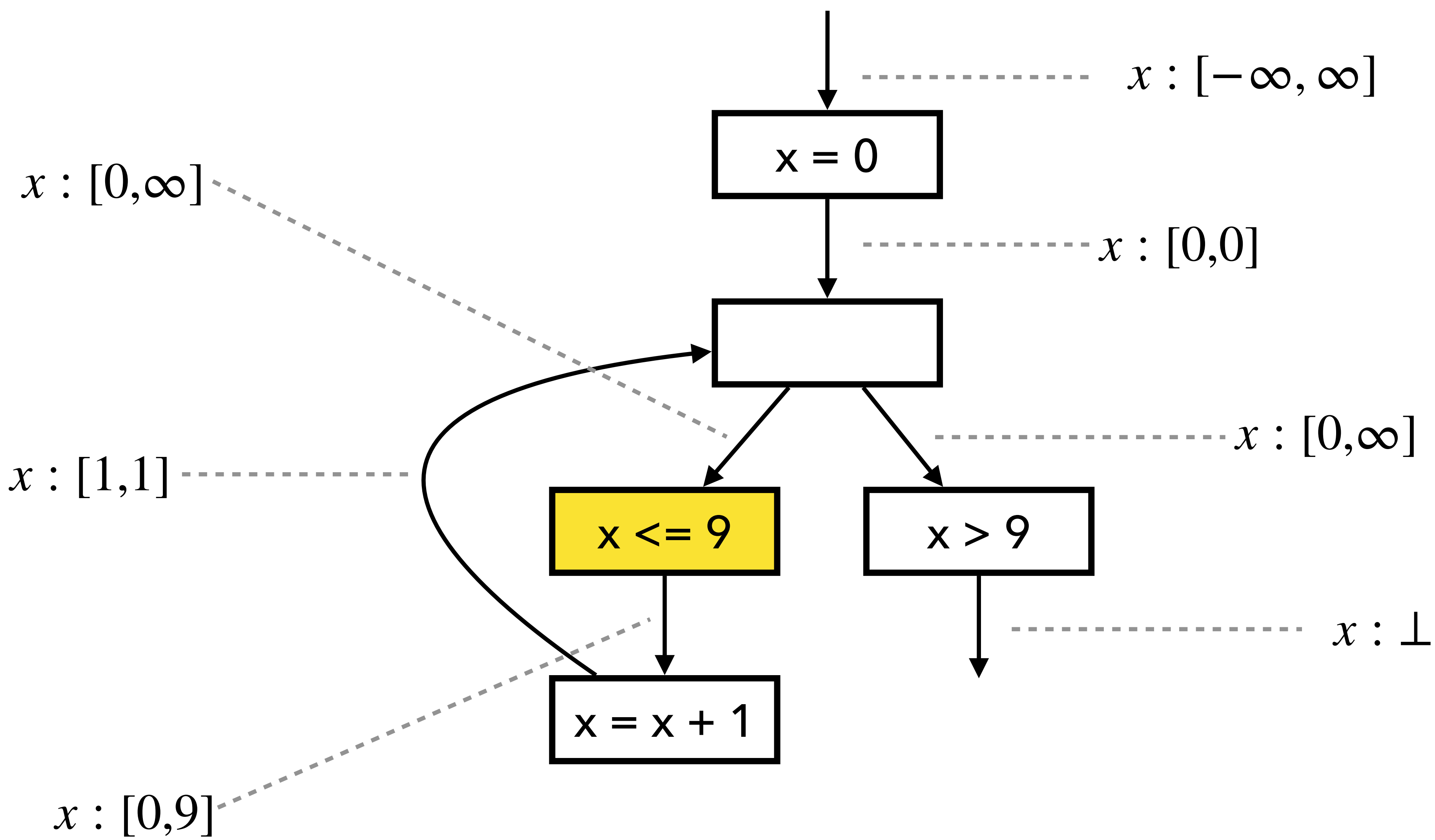
Fixed Point Comp. with Widening

3. Check if fixed point is reached

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

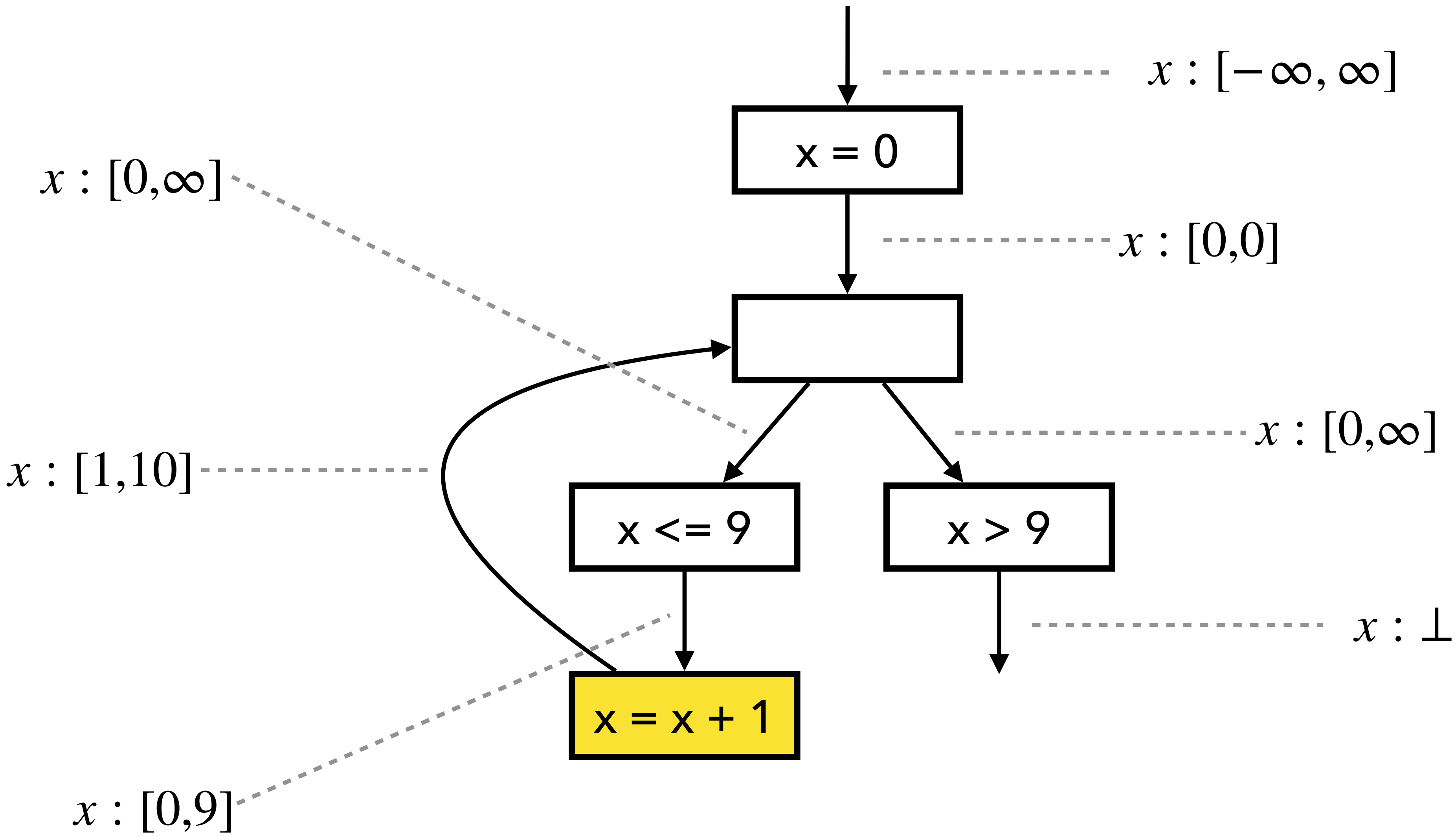


Fixed Point Comp. with Widening



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

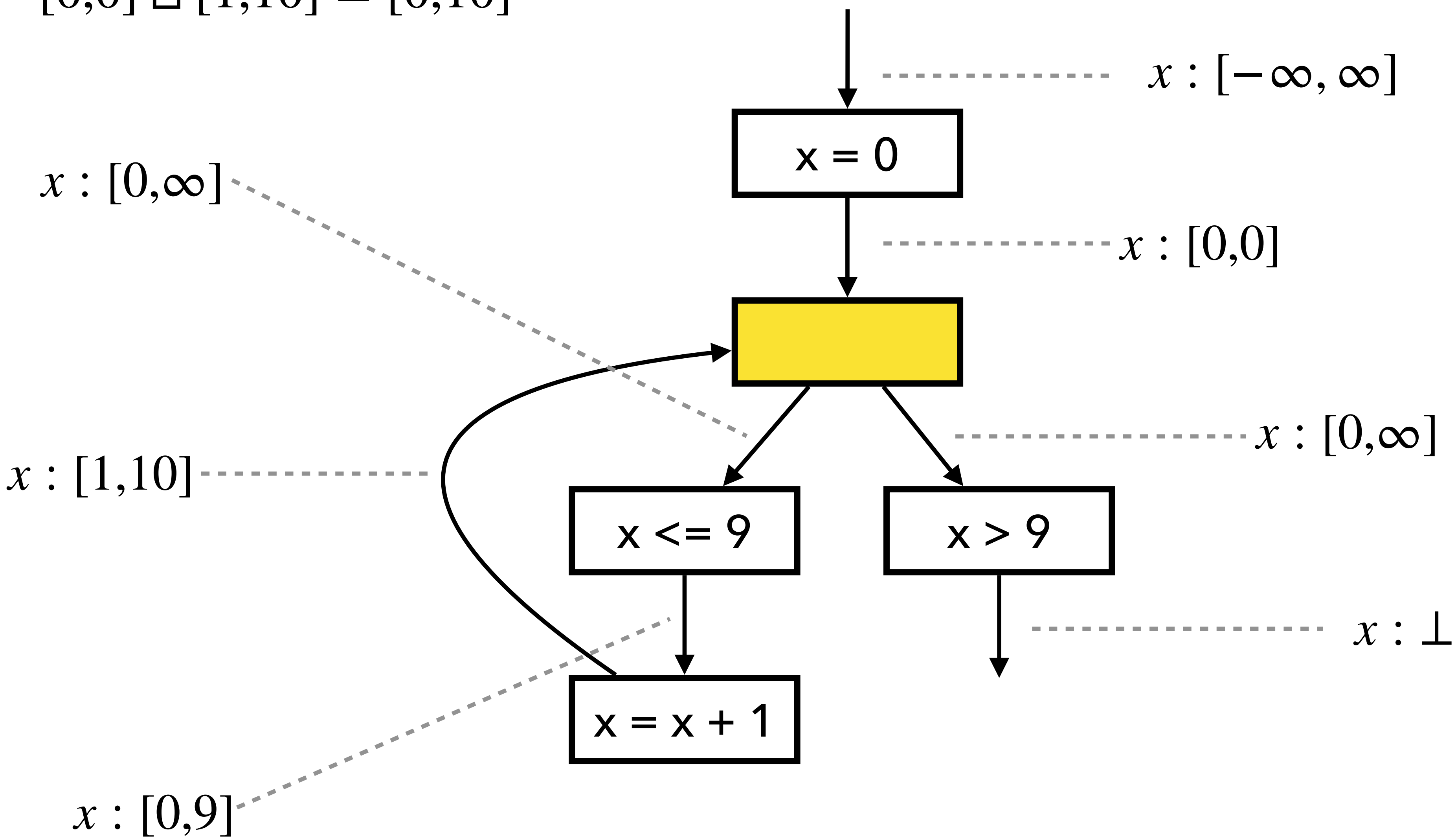
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

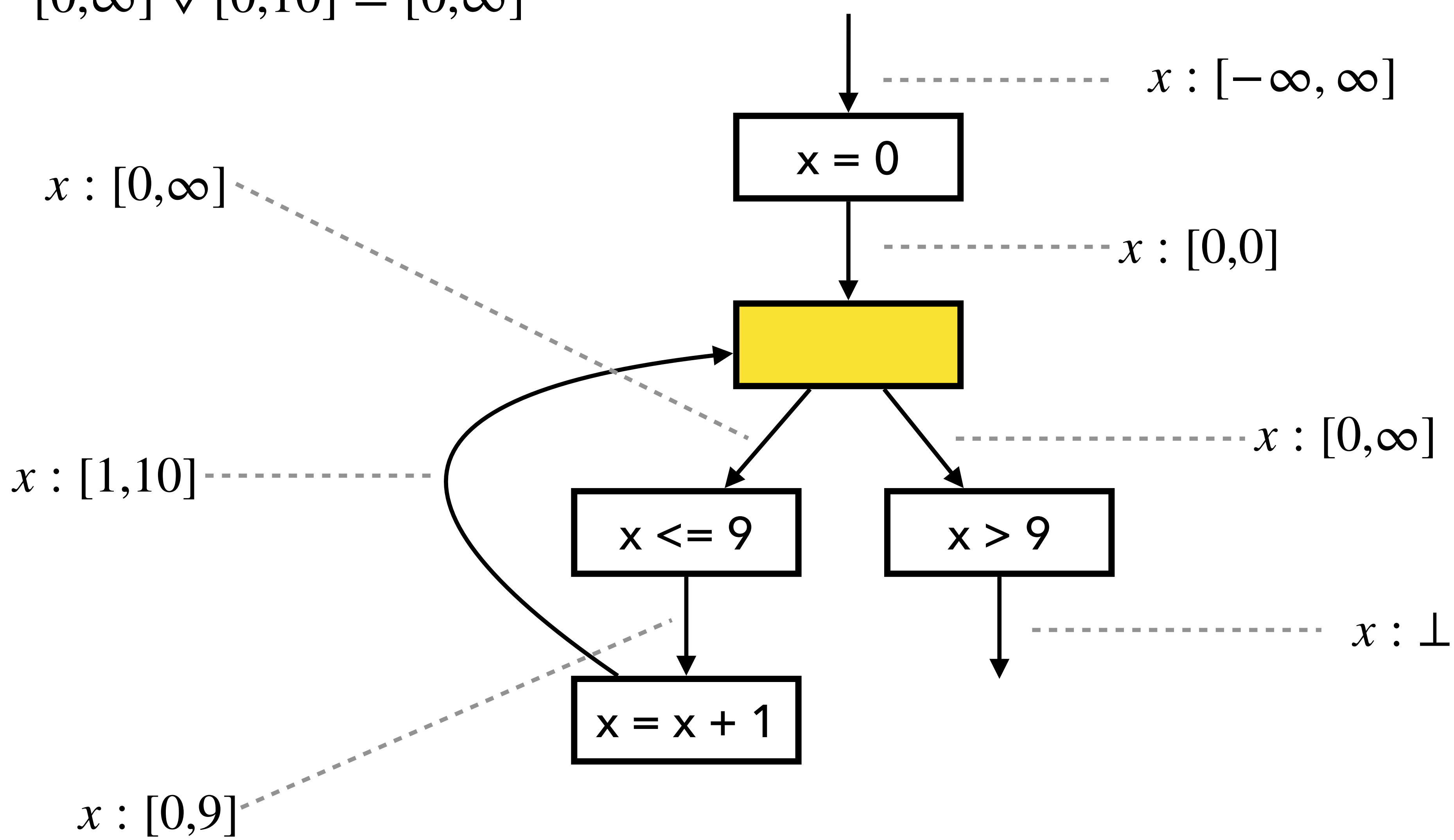
$$[0,0] \sqcup [1,10] = [0,10]$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

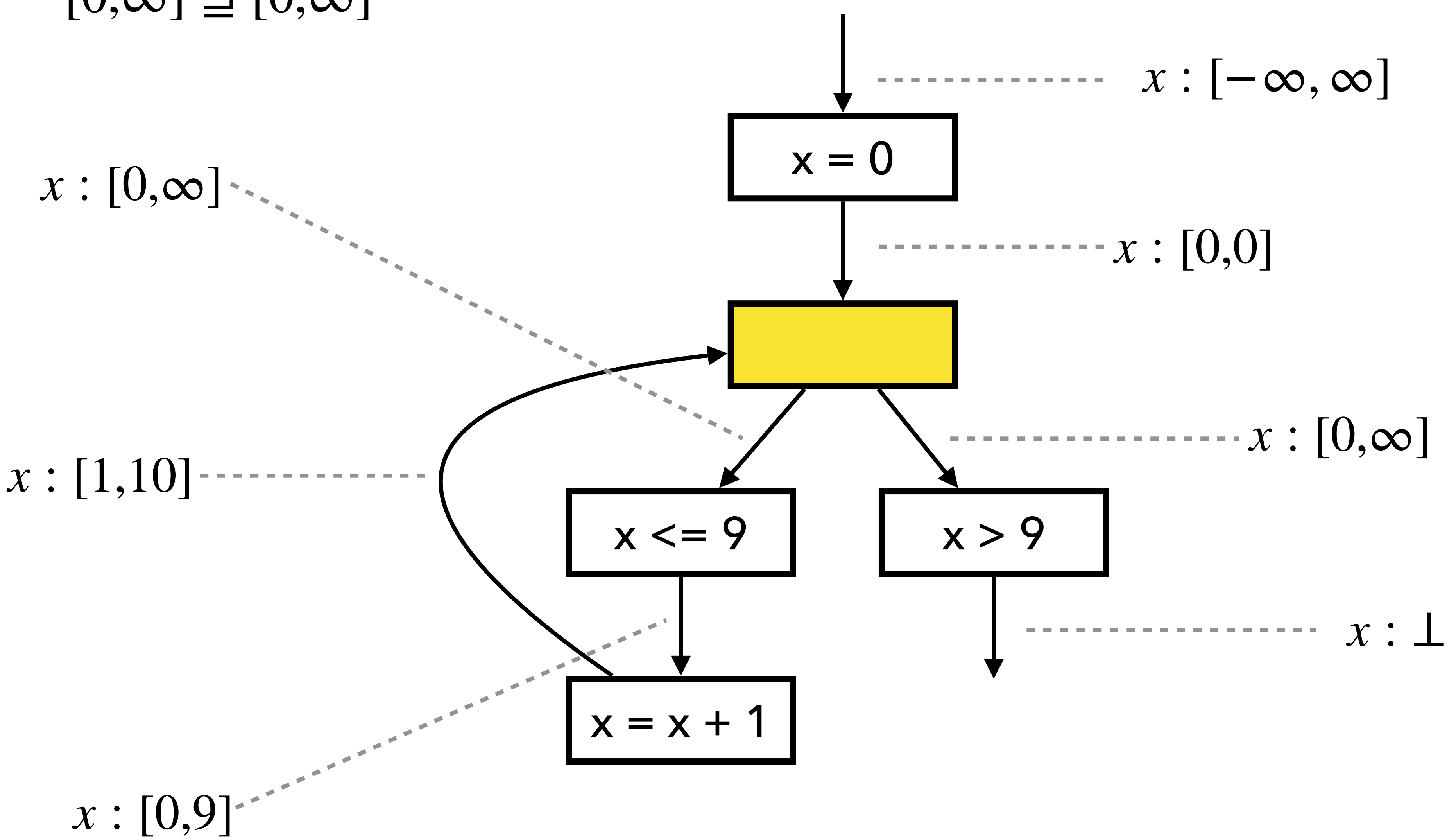
$[0,\infty] \triangledown [0,10] = [0,\infty]$



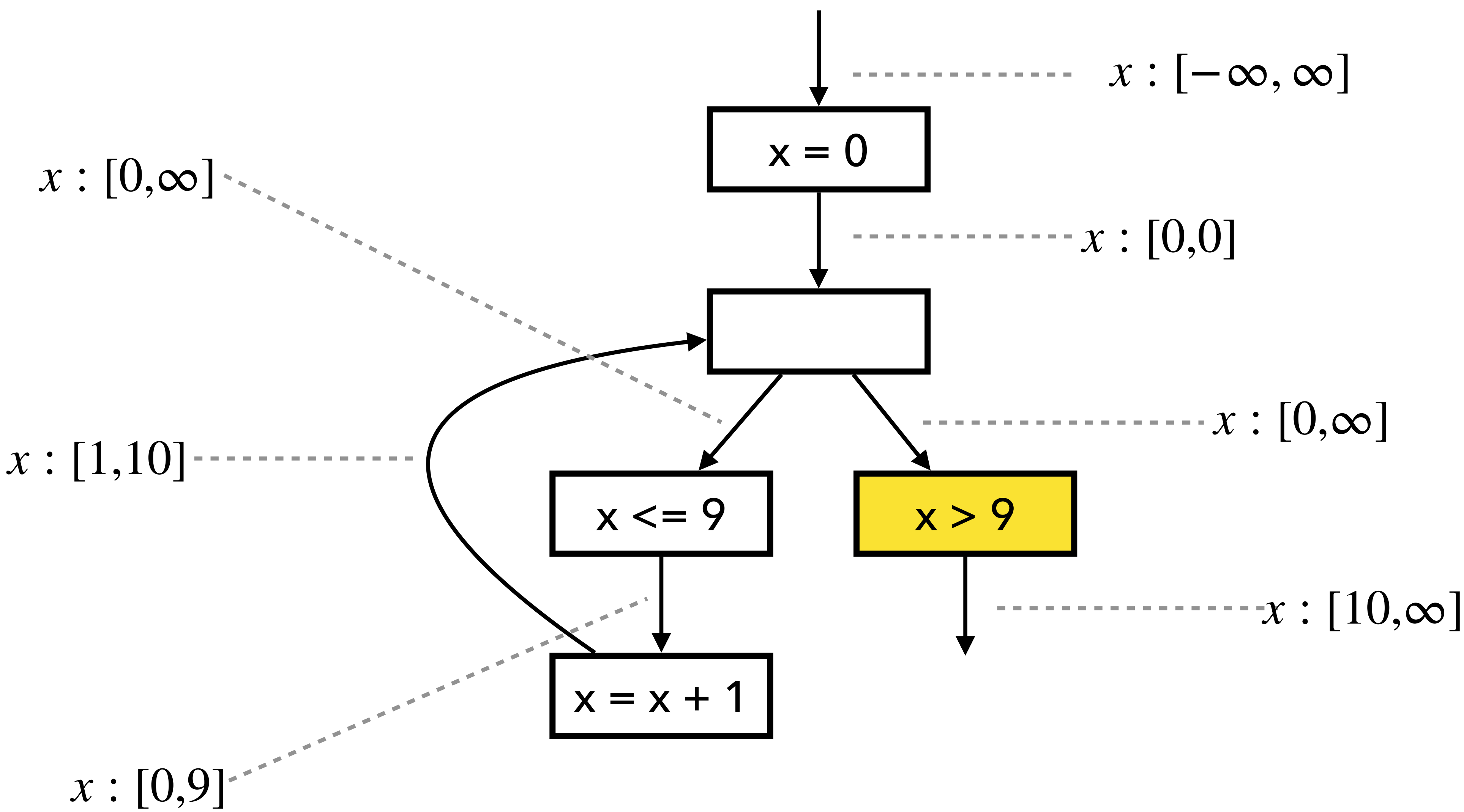
Fixed Point Comp. with Widening

3. Check if fixed point is reached

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



Fixed Point Comp. with Widening

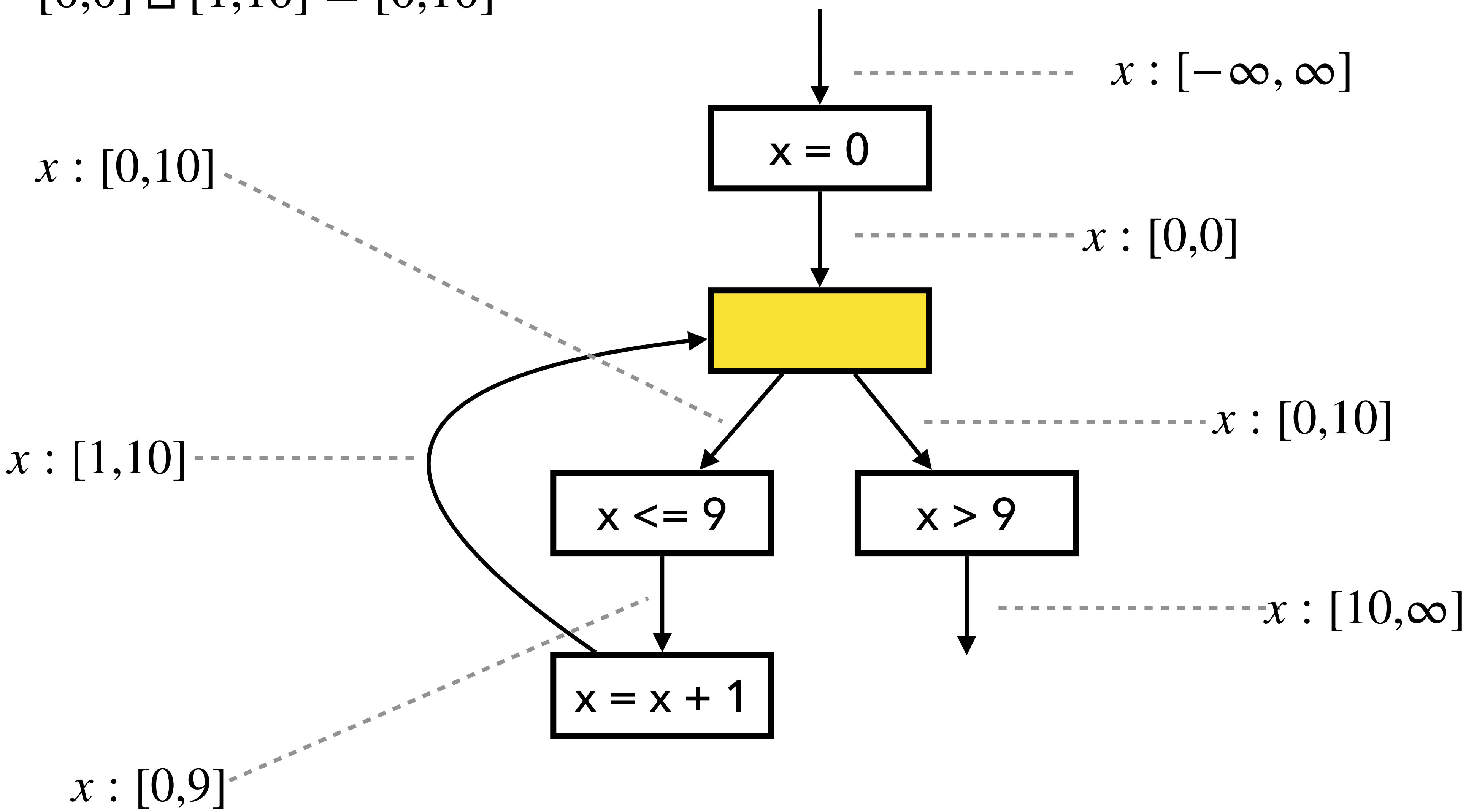


$$[0,\infty] \sqcap [10,\infty] = [10,\infty]$$

Fixed Point Comp. with Narrowing

1. Compute output by joining inputs:

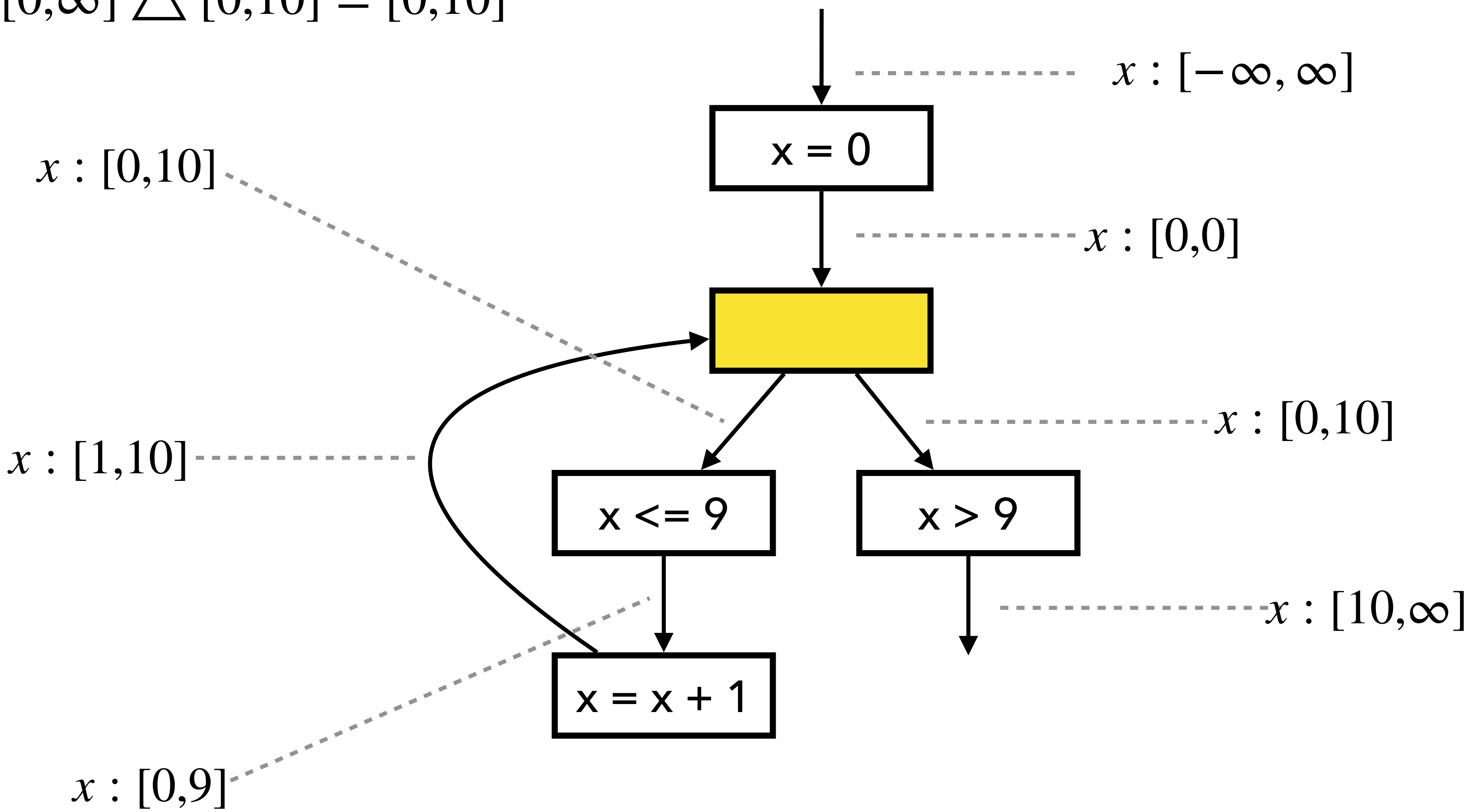
$$[0,0] \sqcup [1,10] = [0,10]$$



Fixed Point Comp. with Narrowing

2. Apply narrowing with old output:

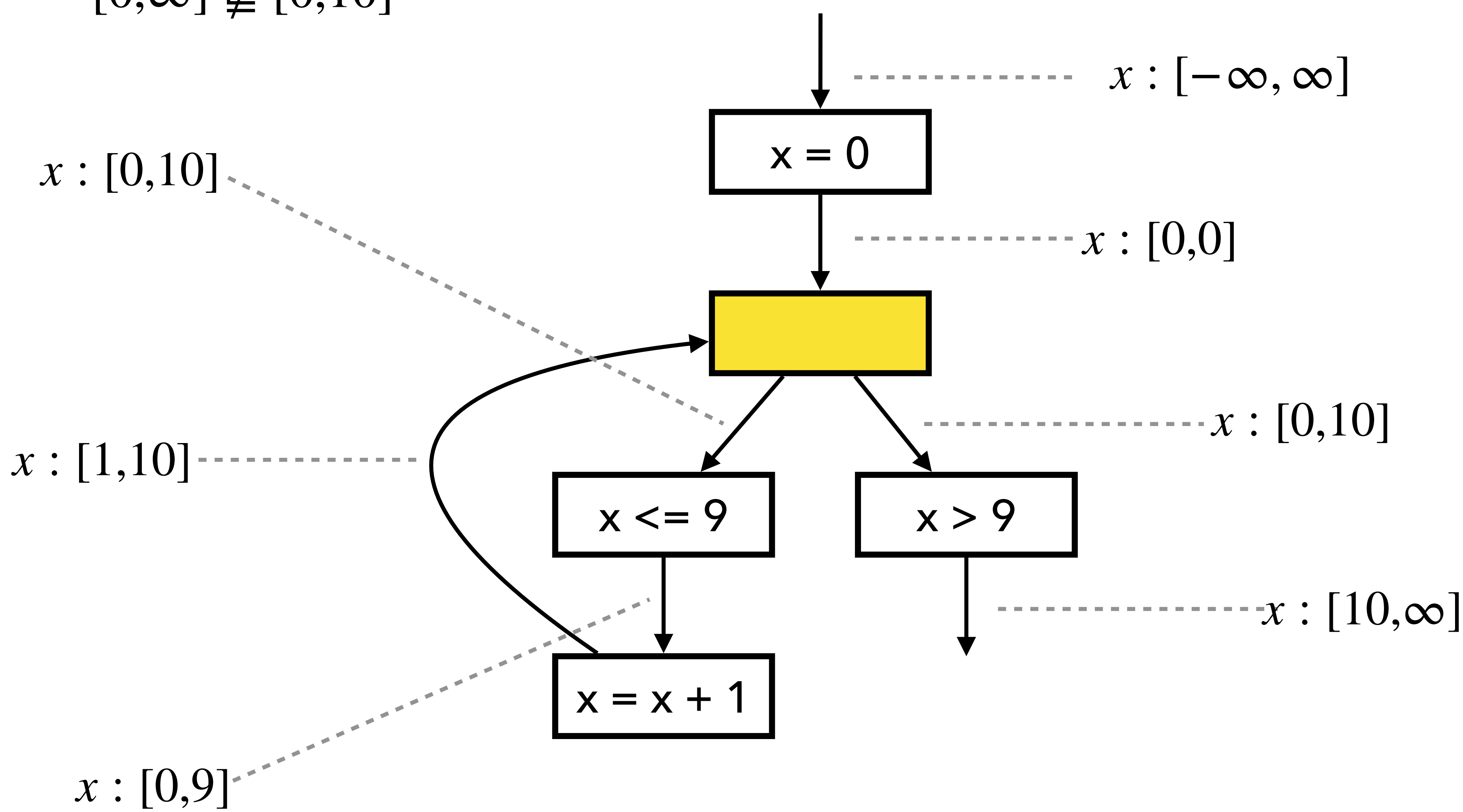
$$[0,\infty] \triangle [0,10] = [0,10]$$



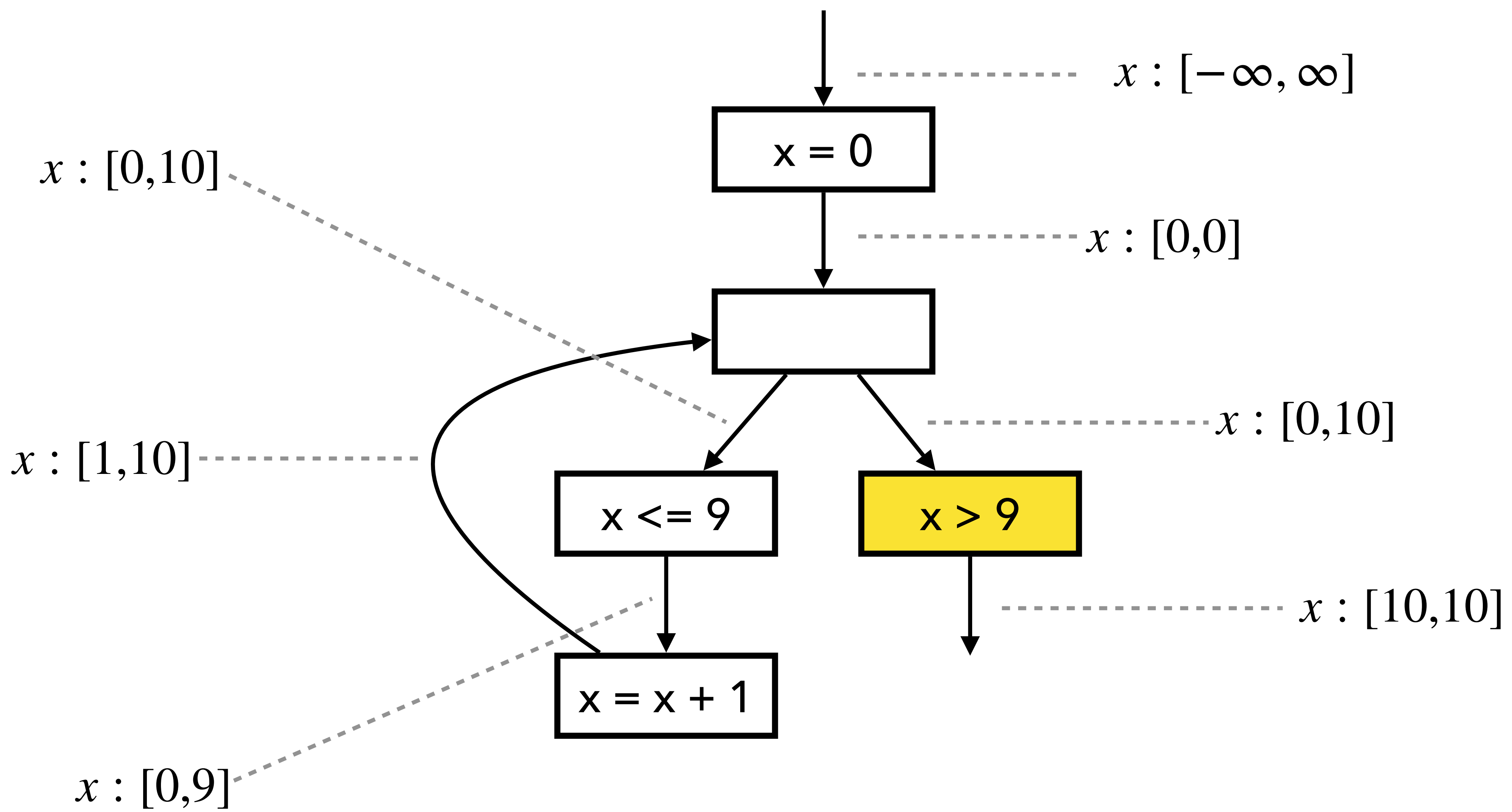
Fixed Point Comp. with Narrowing

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:

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

- 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:

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

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

$$[l_1, u_1] \mathbin{\hat{\times}} [l_2, u_2] = [\mathsf{min}(l_1l_2, l_1u_2, u_1l_2, u_1u_2), \mathsf{max}(l_1l_2, l_1u_2, u_1l_2, u_1u_2)]$$

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

$$[l_1, u_1] \mathbin{\hat{=}} [l_2, u_2] = \widehat{true} \quad (\text{if } l_1 = u_1 = l_2 = u_2)$$

$$[l_1, u_1] \mathbin{\hat{=}} [l_2, u_2] = \widehat{false} \quad (\text{no overlap})$$

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

$$[l_1, u_1] \mathbin{\hat{<}} [l_2, u_2] = \widehat{true} \quad (\text{if } u_1 < l_2)$$

$$[l_1, u_1] \mathbin{\hat{<}} [l_2, u_2] = \widehat{false} \quad (\text{if } l_1 > u_2)$$

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

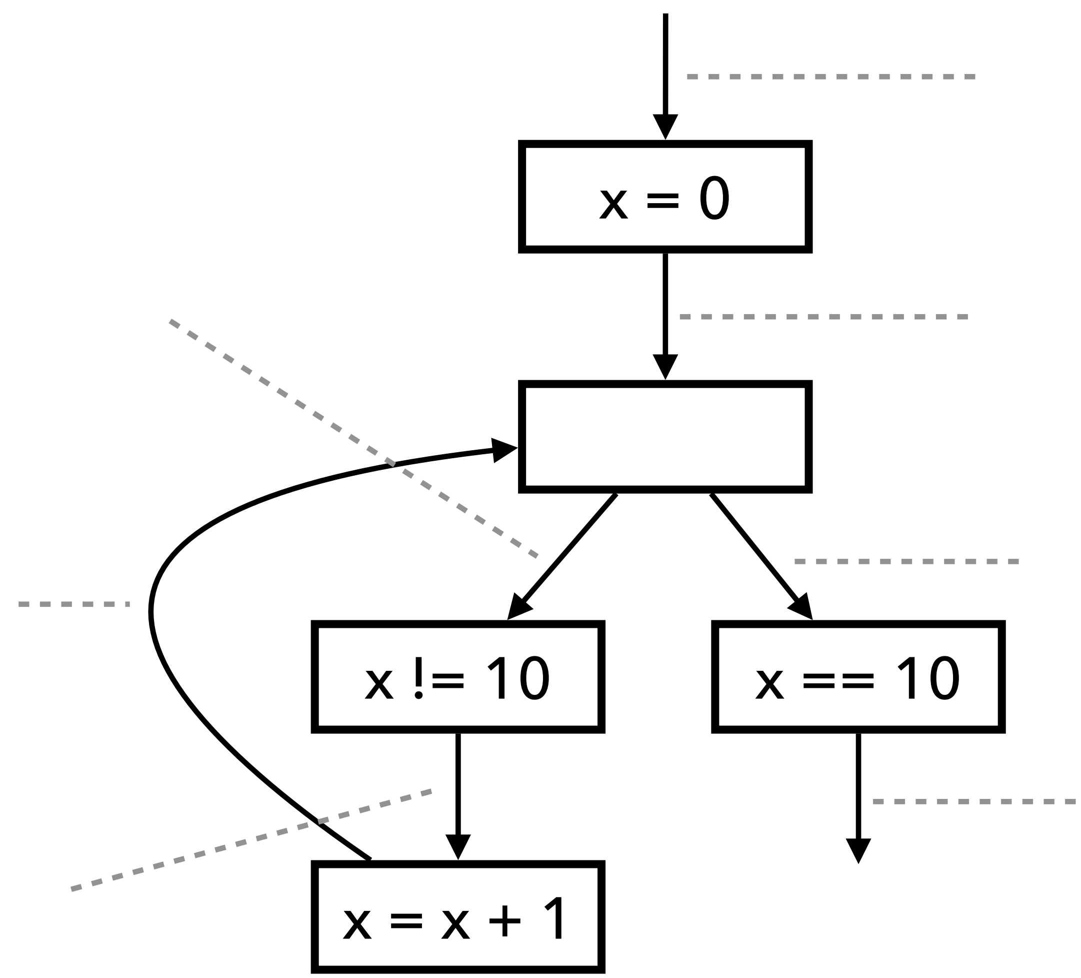
Fixpoint comp. with narrowing

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

Exercise (2)

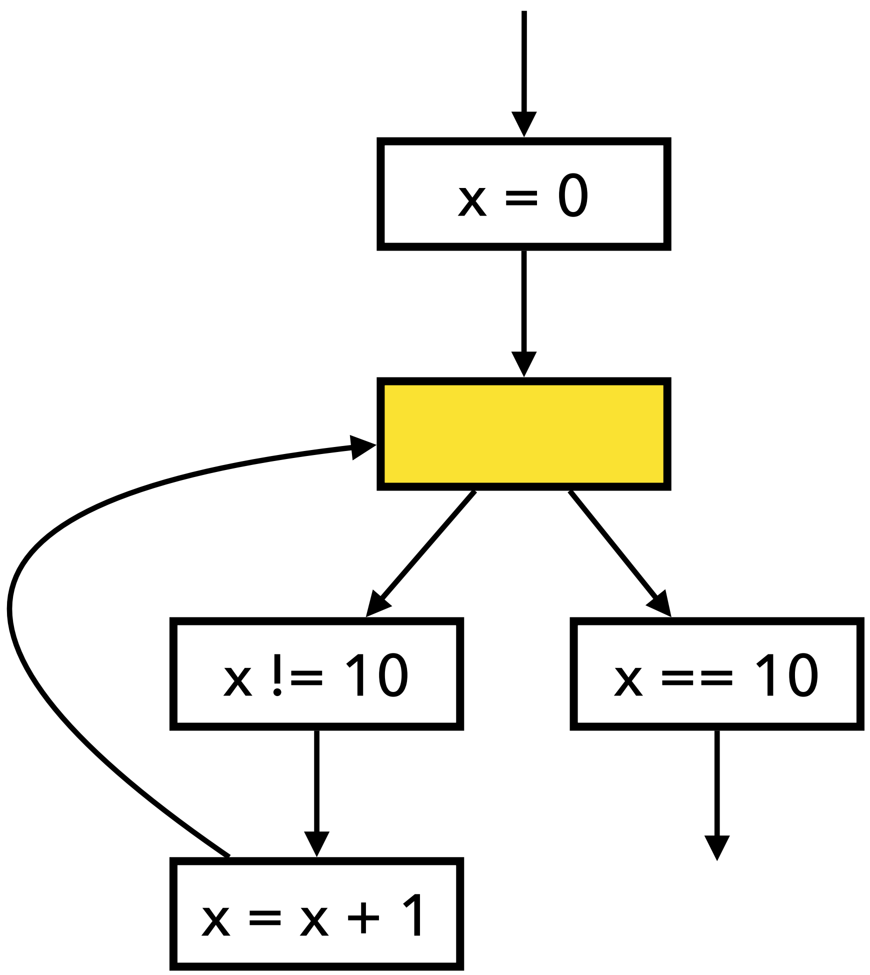
Describe the result of the interval analysis:
(1) without widening
(2) with widening/narrowing

```
x = 0;

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



Widening with Thresholds

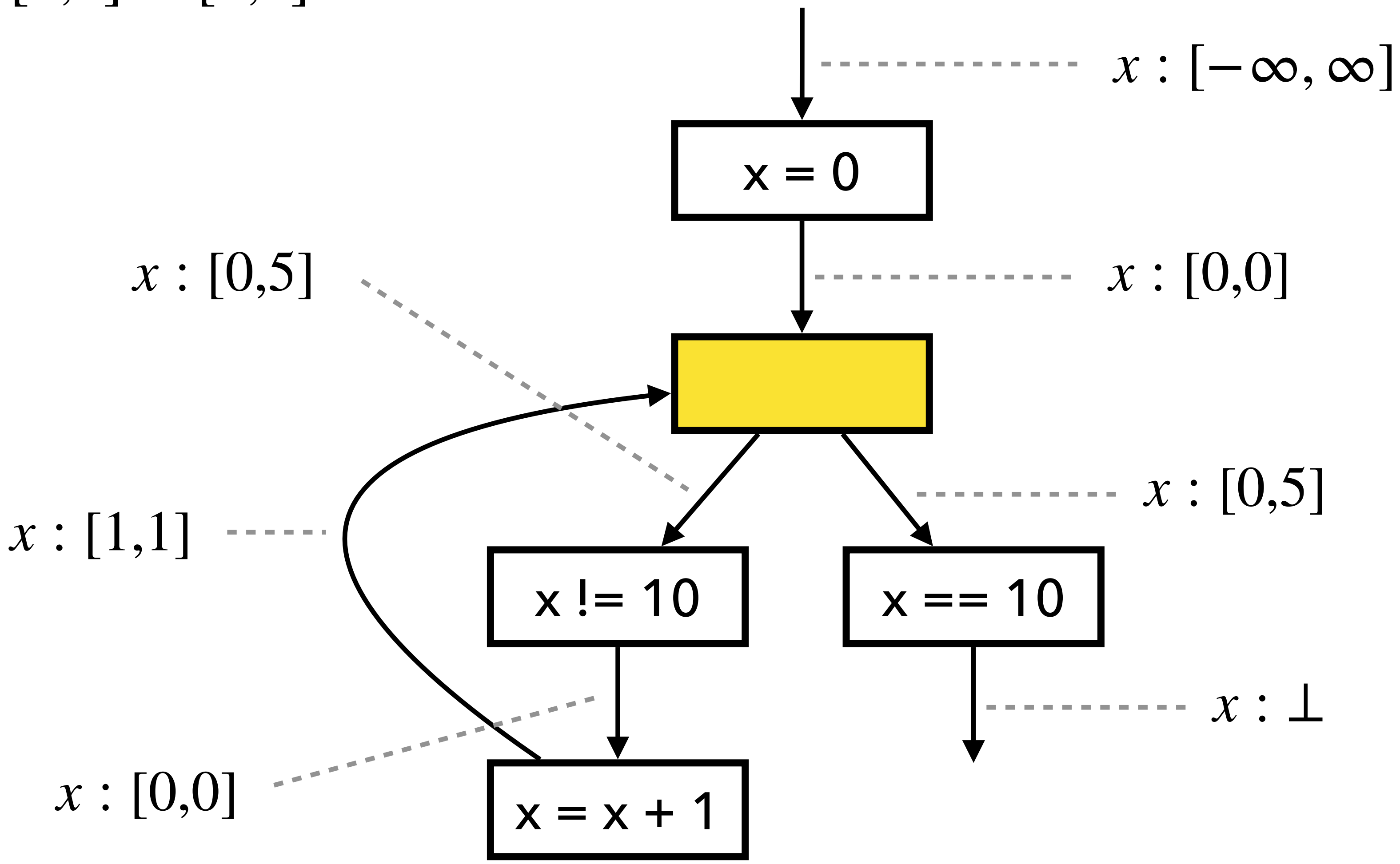
Assume a set T of thresholds is given beforehand: e.g., $T = \{5, 10\}$



Widening with Thresholds

1. Compute output by joining inputs:

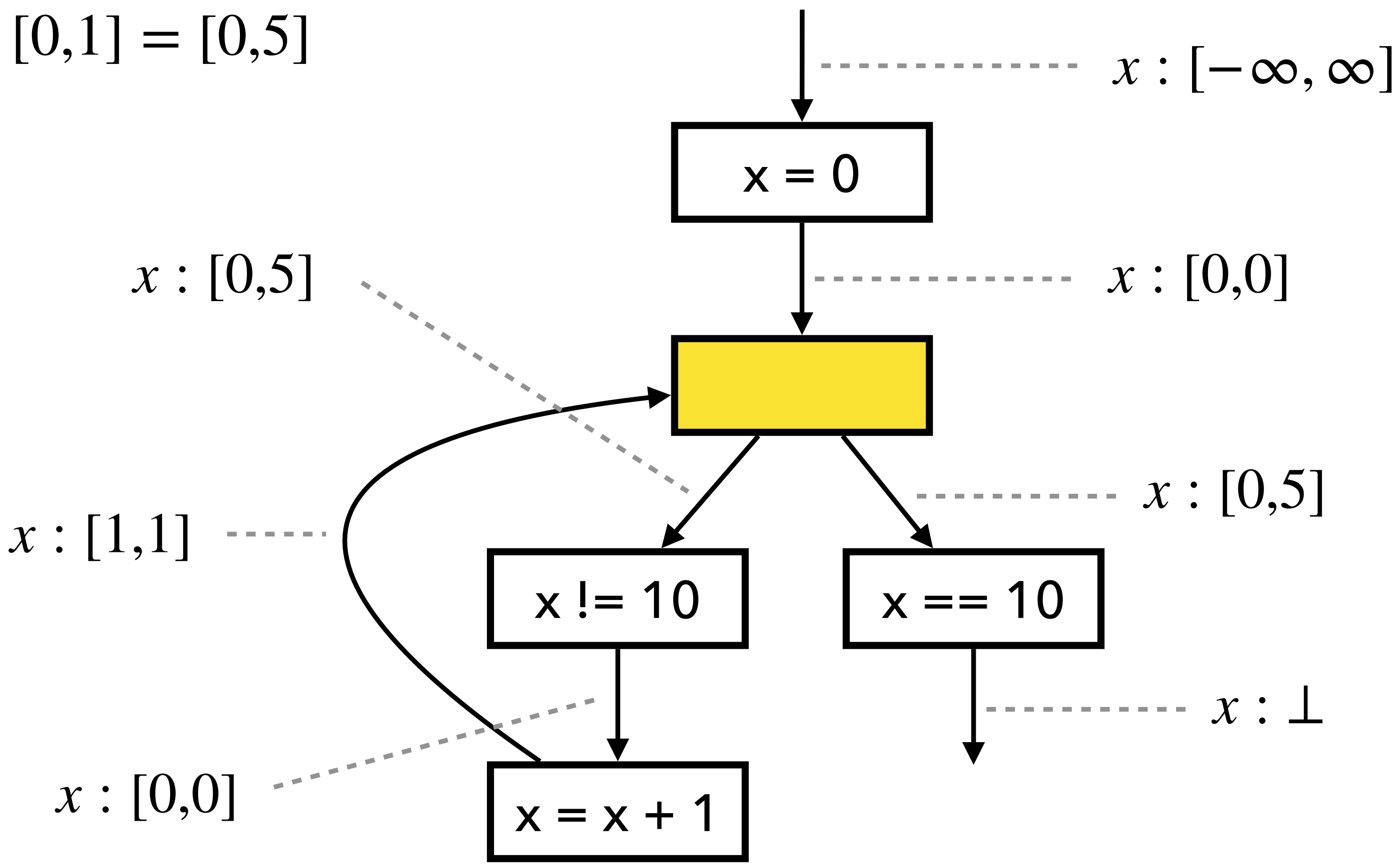
$$[0,0] \sqcup [1,1] = [0,1]$$



Widening with Thresholds

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

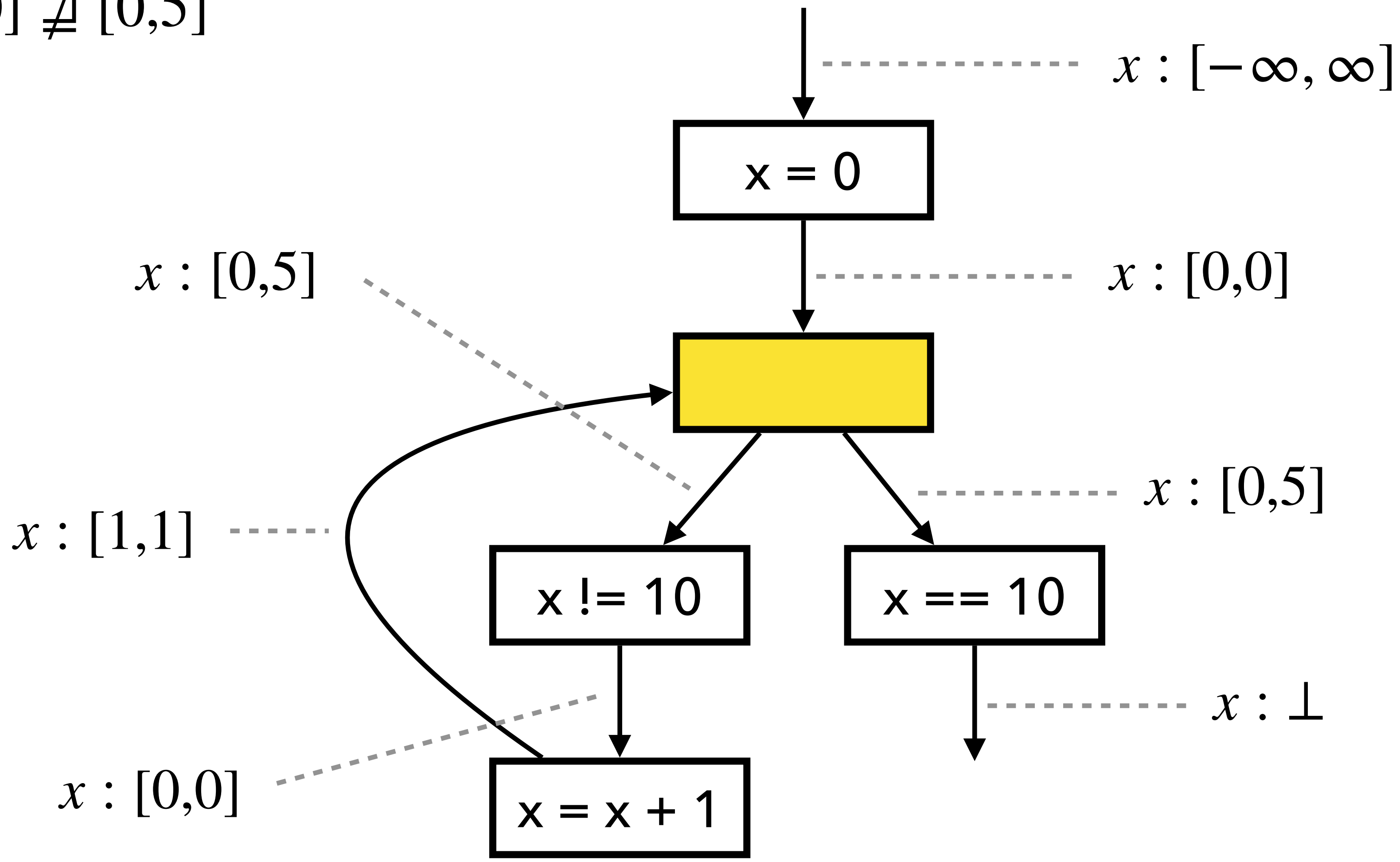
$$[0,0] \triangledown [0,1] = [0,5]$$



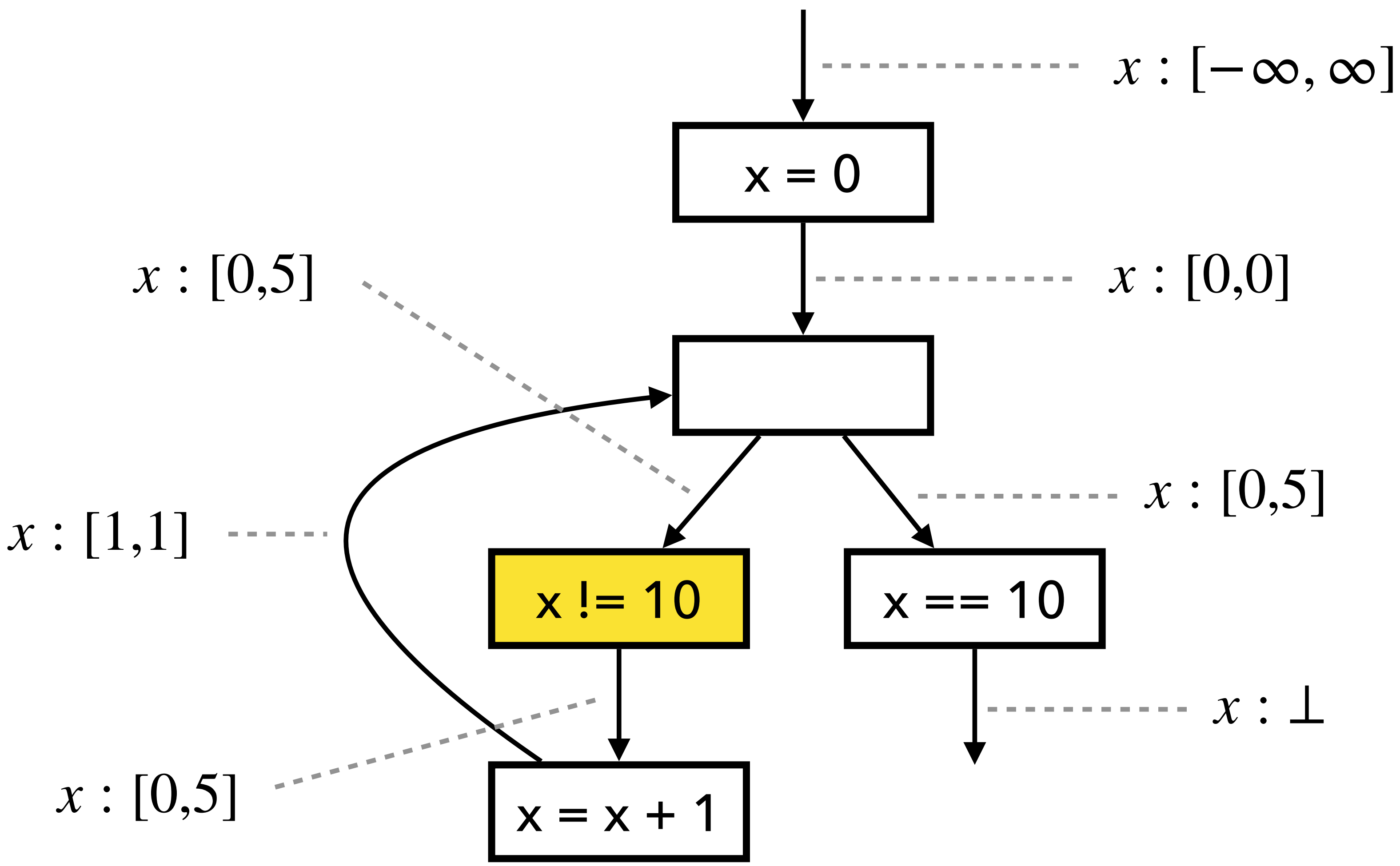
Widening with Thresholds

3. Check if fixed point is reached:

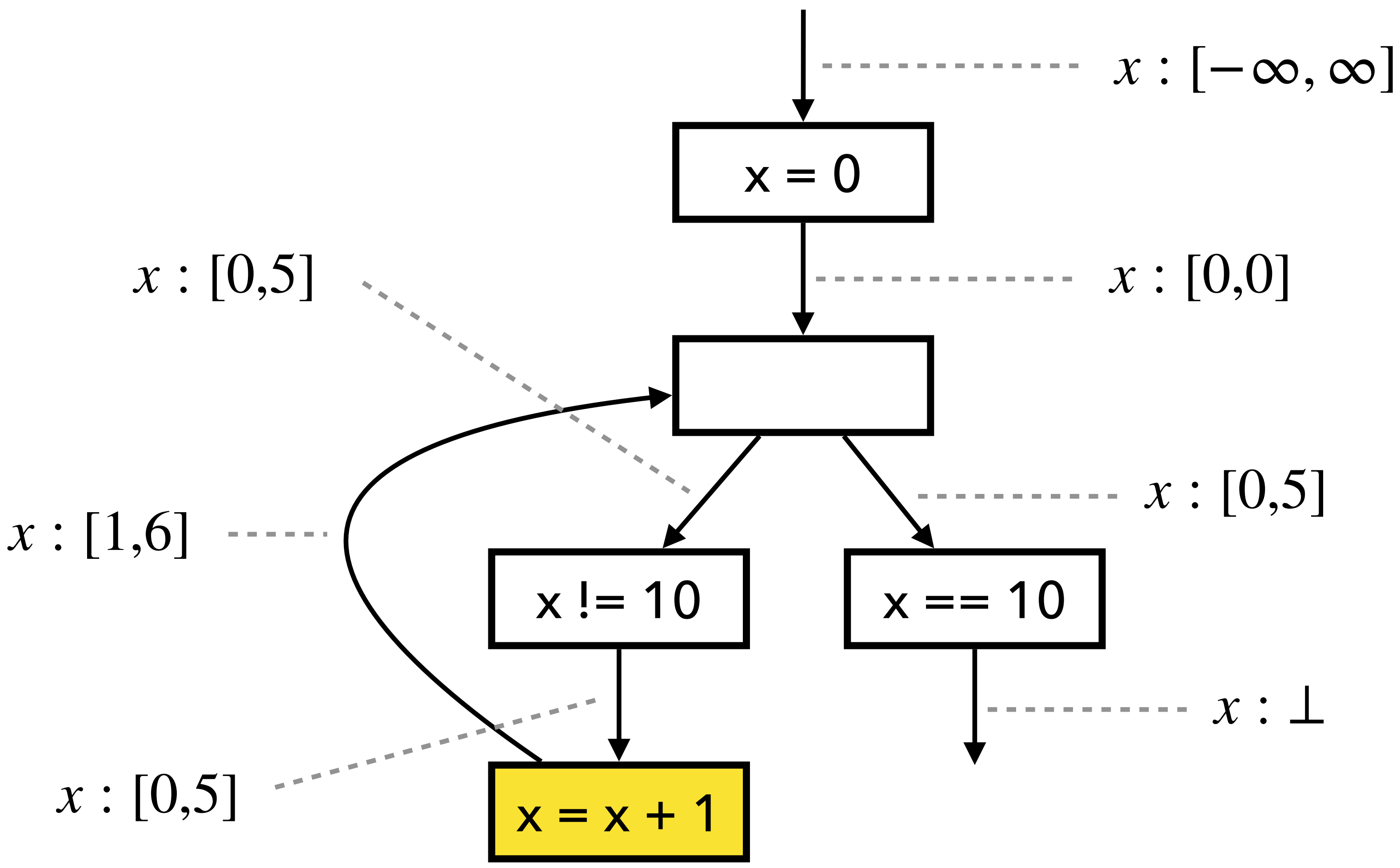
$$[0,0] \not\sqsupseteq [0,5]$$



Widening with Thresholds



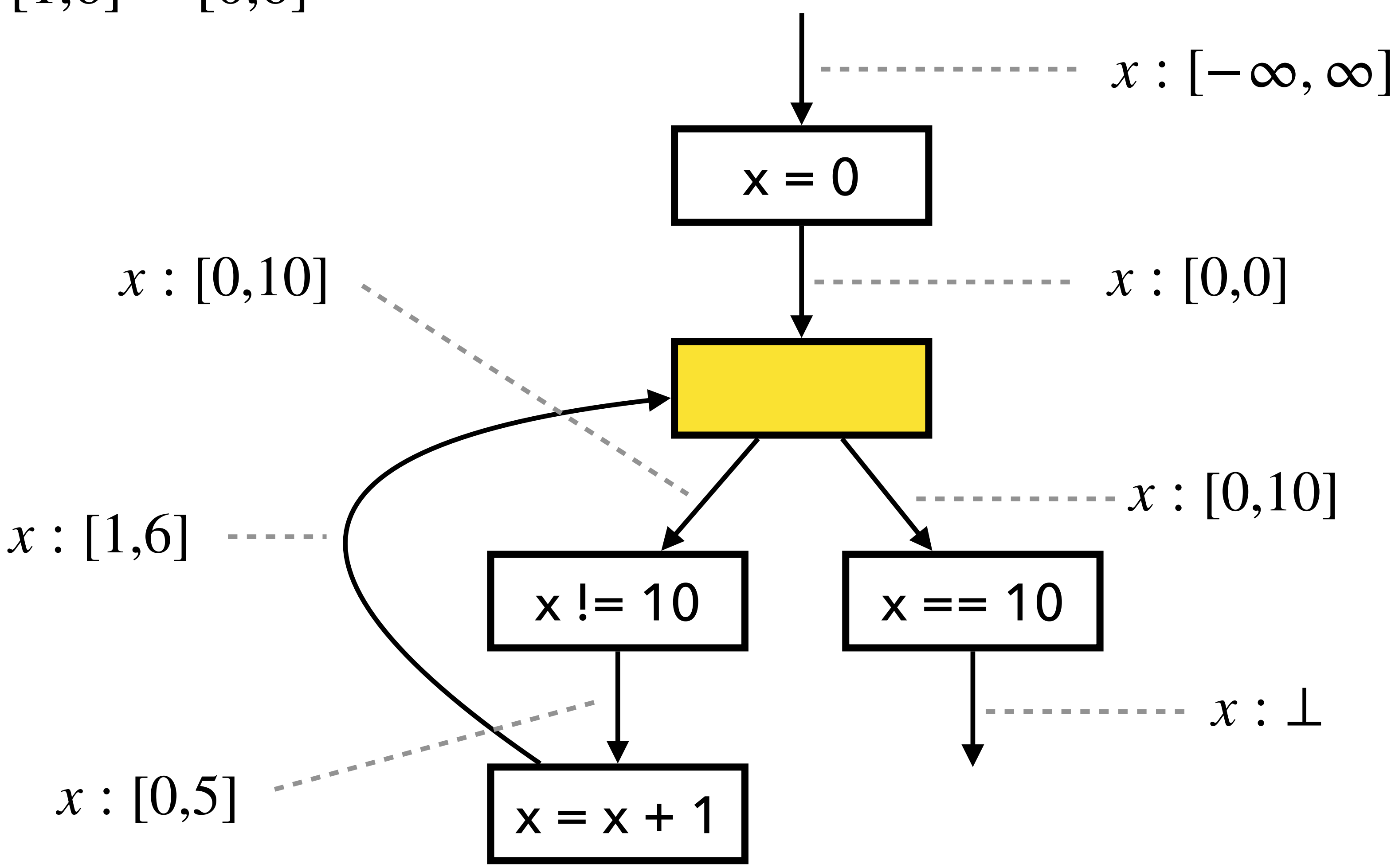
Widening with Thresholds



Widening with Thresholds

1. Compute output by joining inputs:

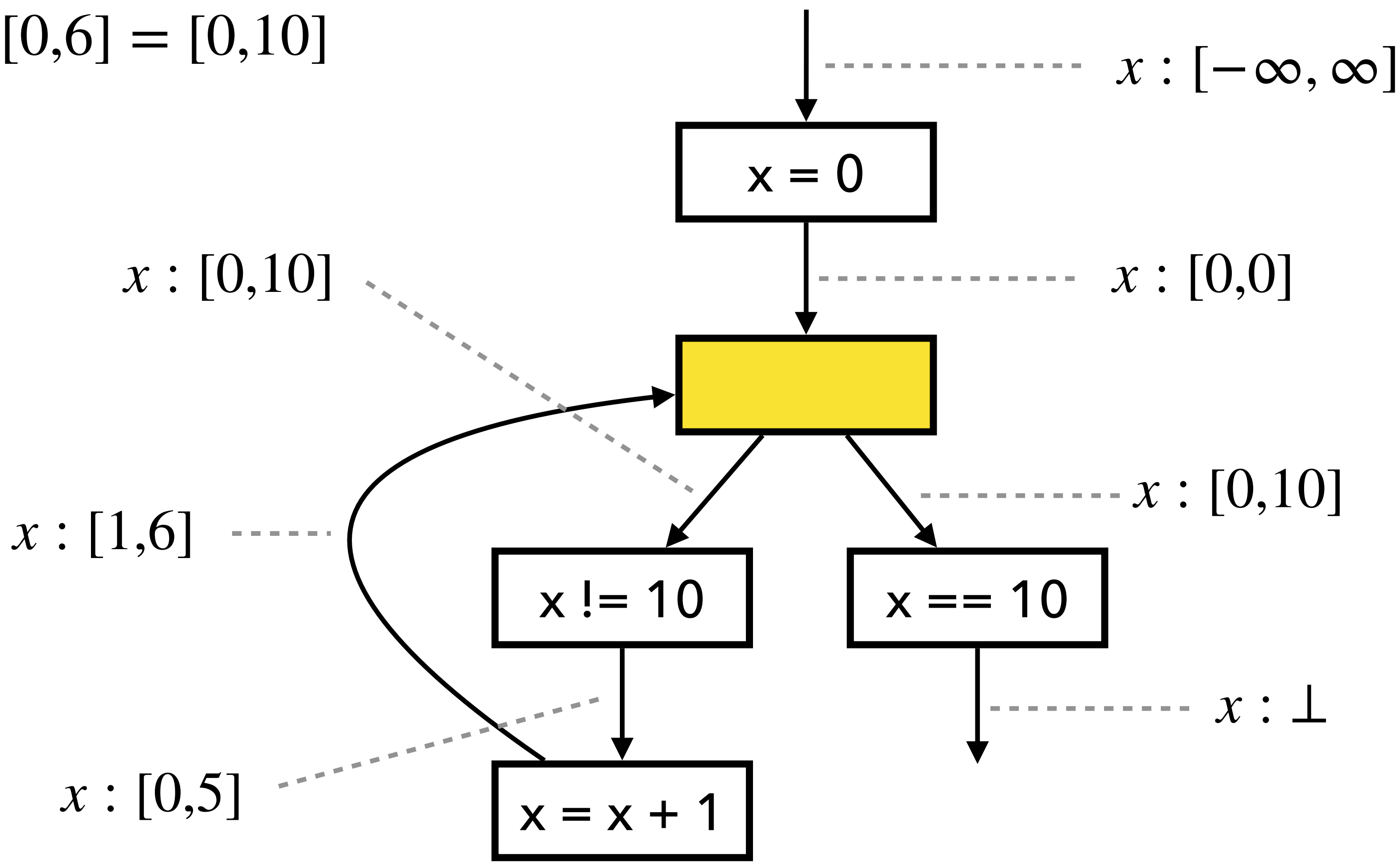
$$[0,0] \sqcup [1,6] = [0,6]$$



Widening with Thresholds

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

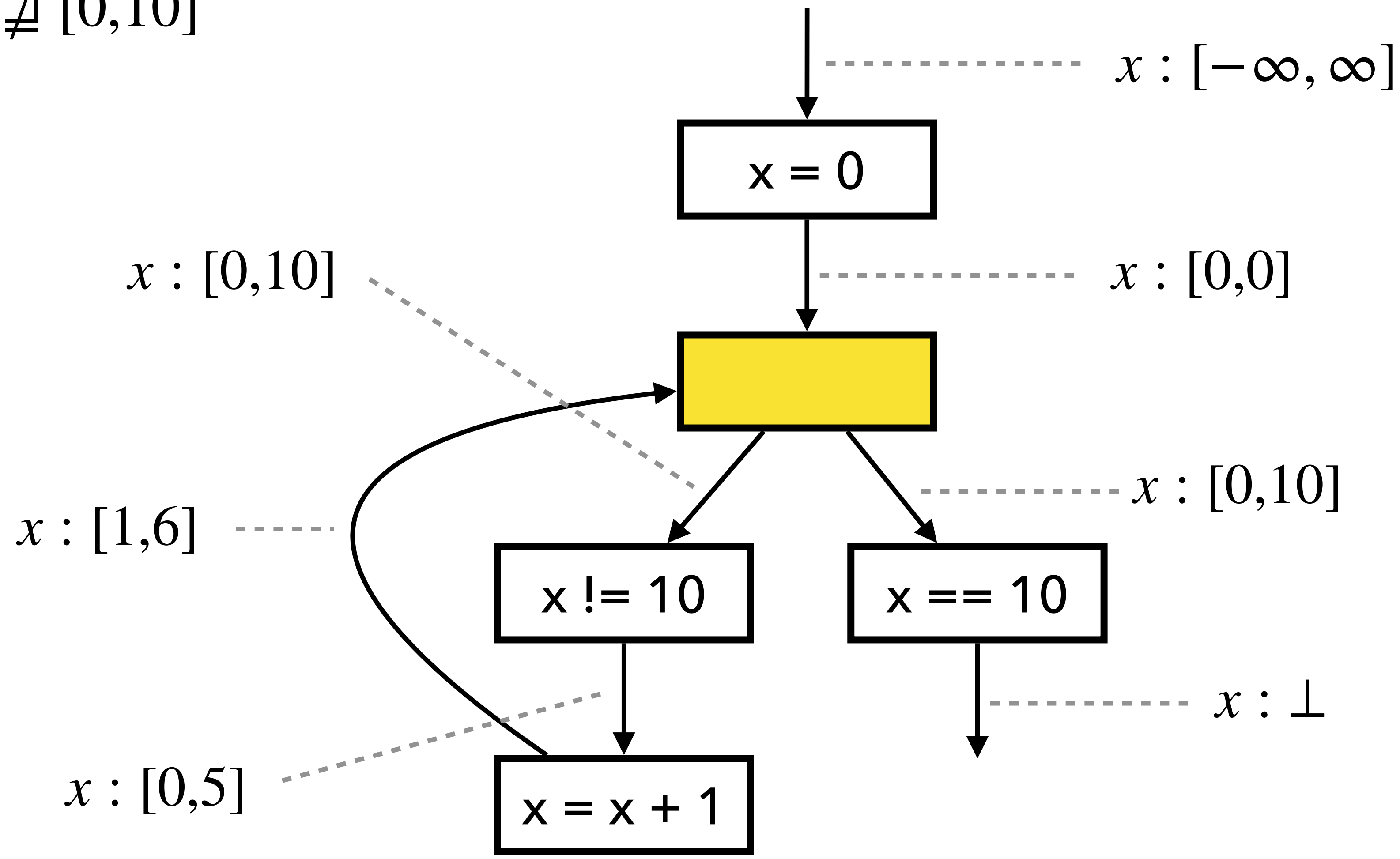
$$[0,5] \triangledown [0,6] = [0,10]$$



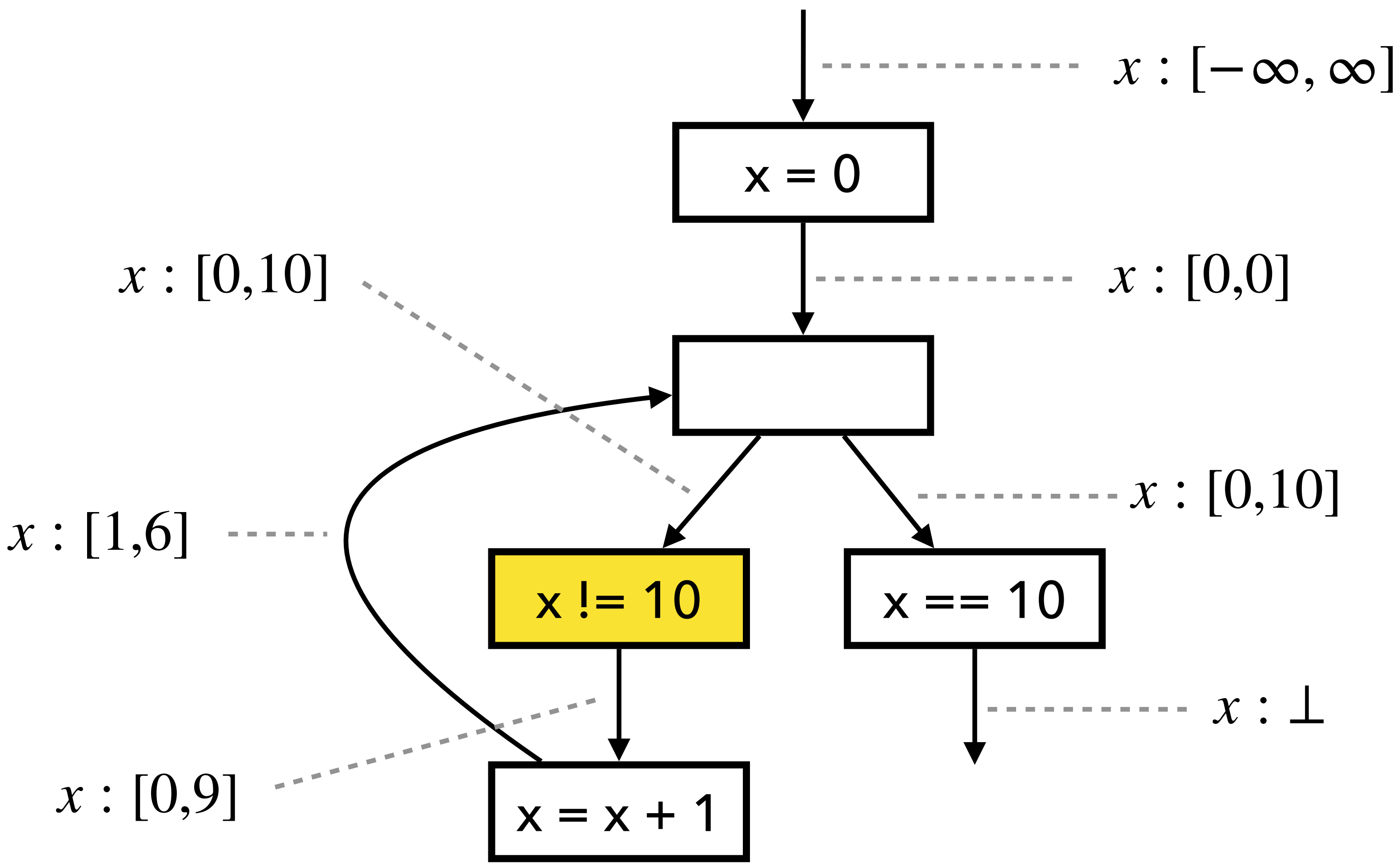
Widening with Thresholds

3. Check if fixed point is reached:

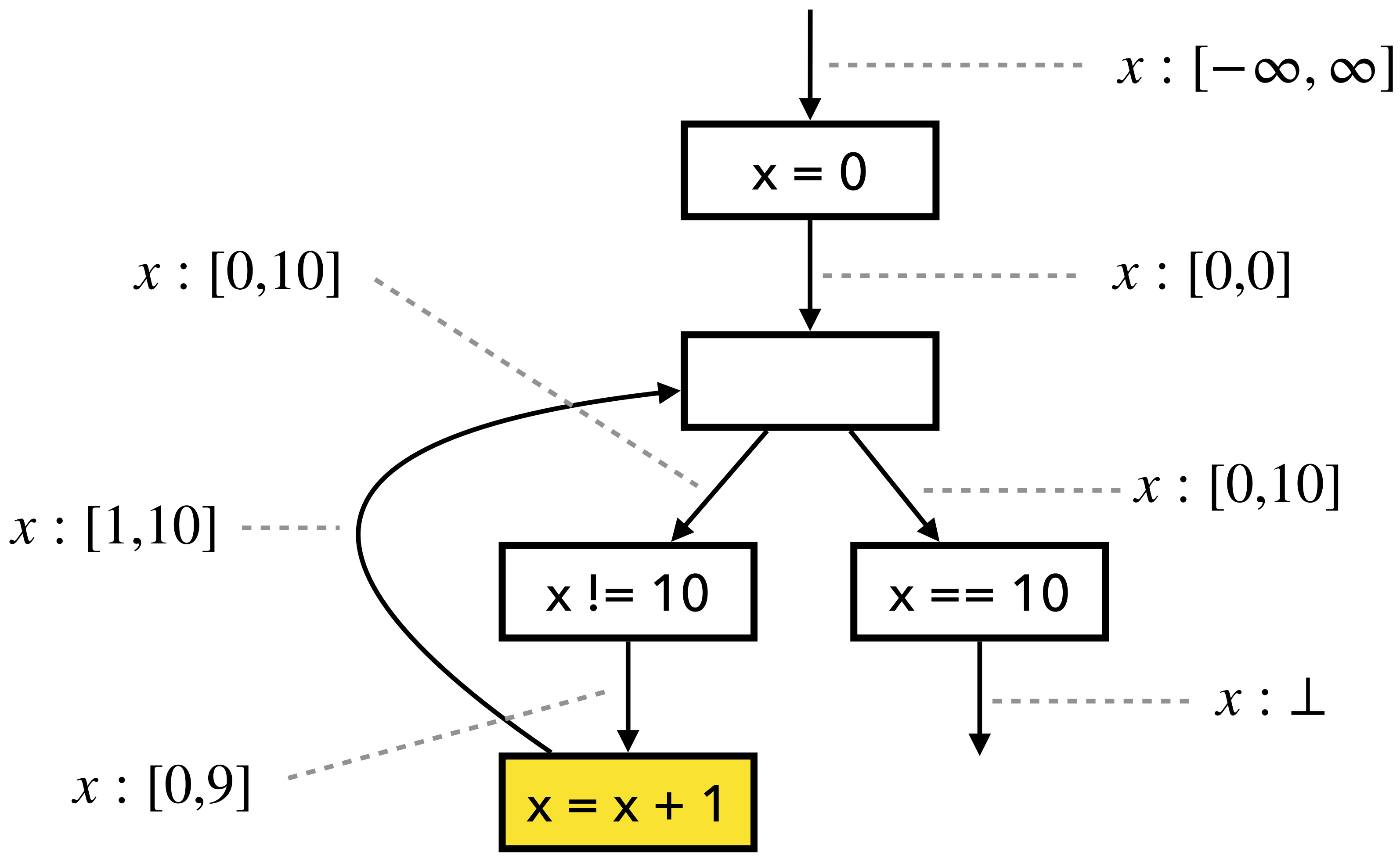
$$[0,5] \not\sqsupseteq [0,10]$$



Widening with Thresholds



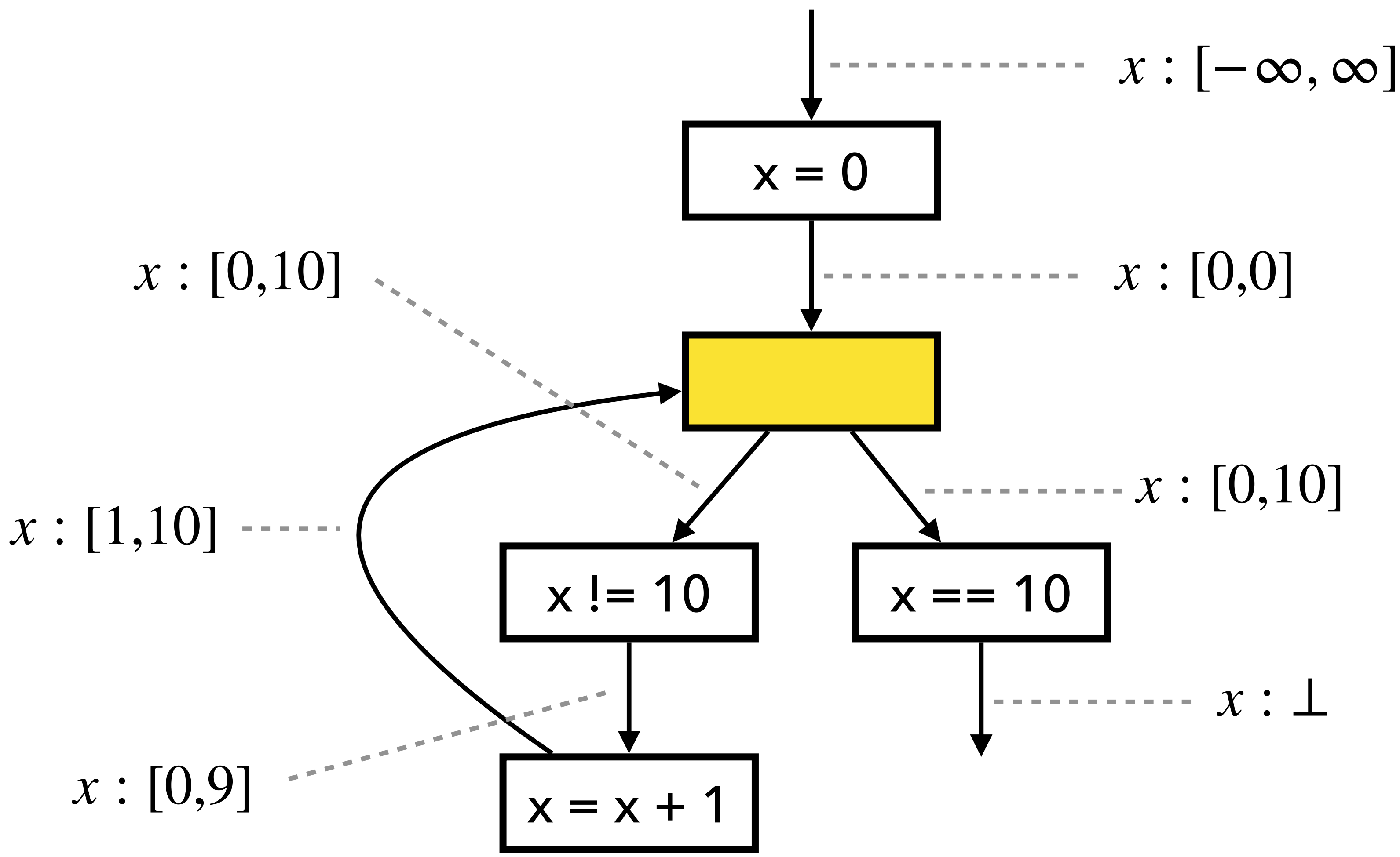
Widening with Thresholds



Widening with Thresholds

1. Compute output by joining inputs:

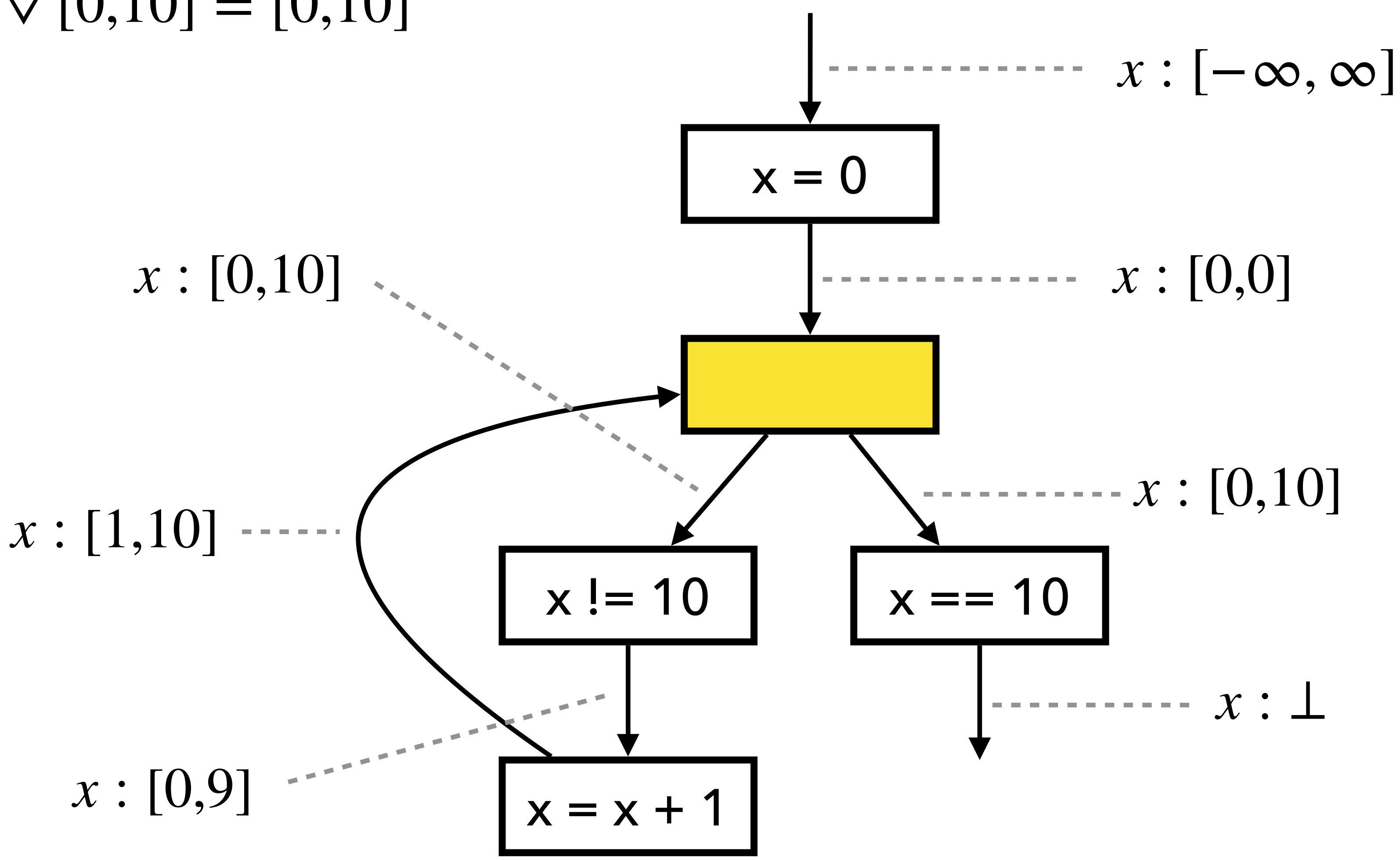
$$[0,0] \sqcup [1,10] = [0,10]$$



Widening with Thresholds

2. Apply widening:

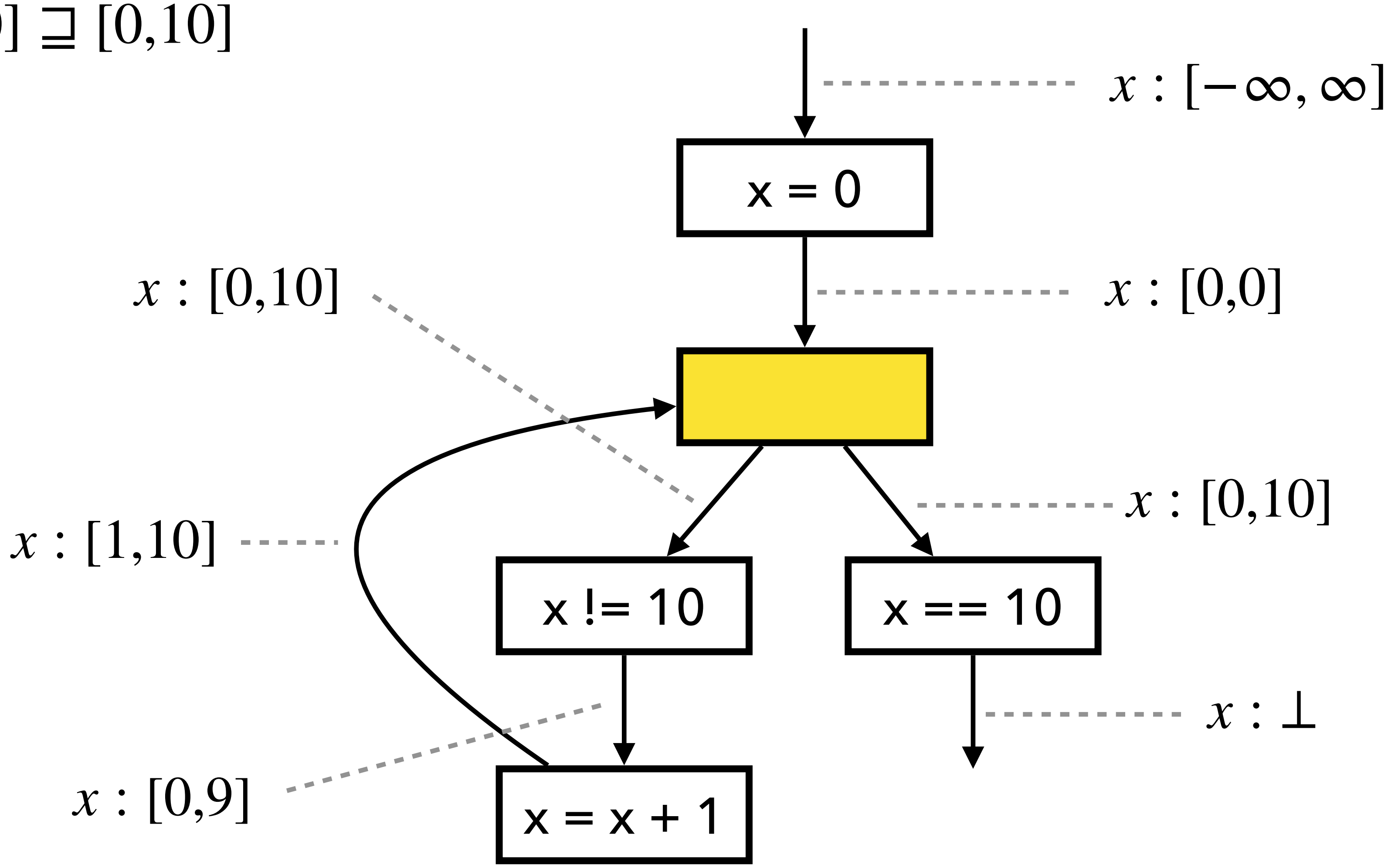
$$[0,10] \triangledown [0,10] = [0,10]$$



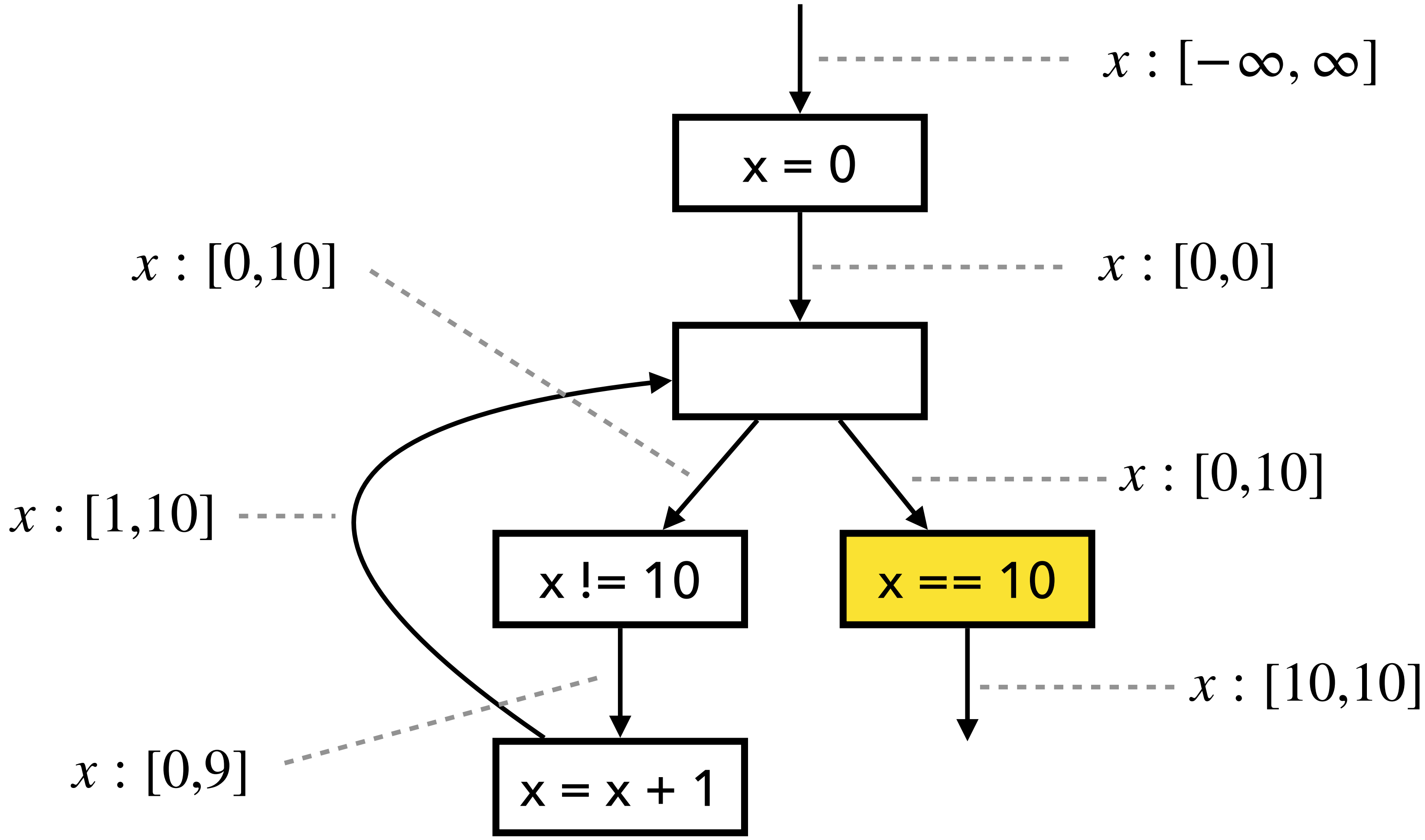
Widening with Thresholds

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.

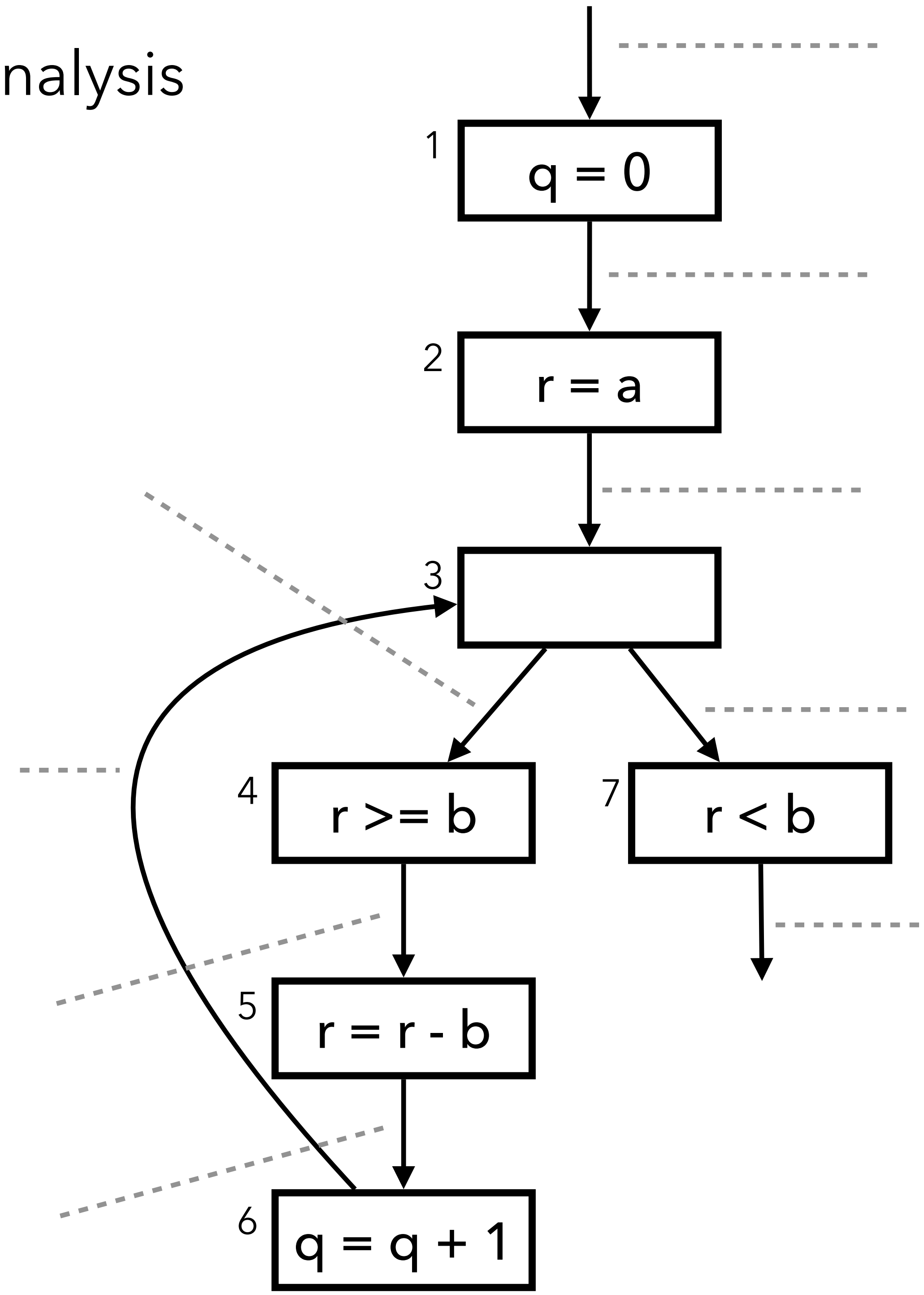
$$\bot \bigtriangledown_T \hat{z} = \hat{z}$$
$$\hat{z} \bigtriangledown_T \bot = \hat{z}$$
$$[l_1, u_1] \bigtriangledown_T [l_2, u_2] = [l_1 > l_2 ? glb(T, l_2) : l_1, u_1 < u_2 ? lub(T, u_2) : u_1]$$

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

Exercise (3)

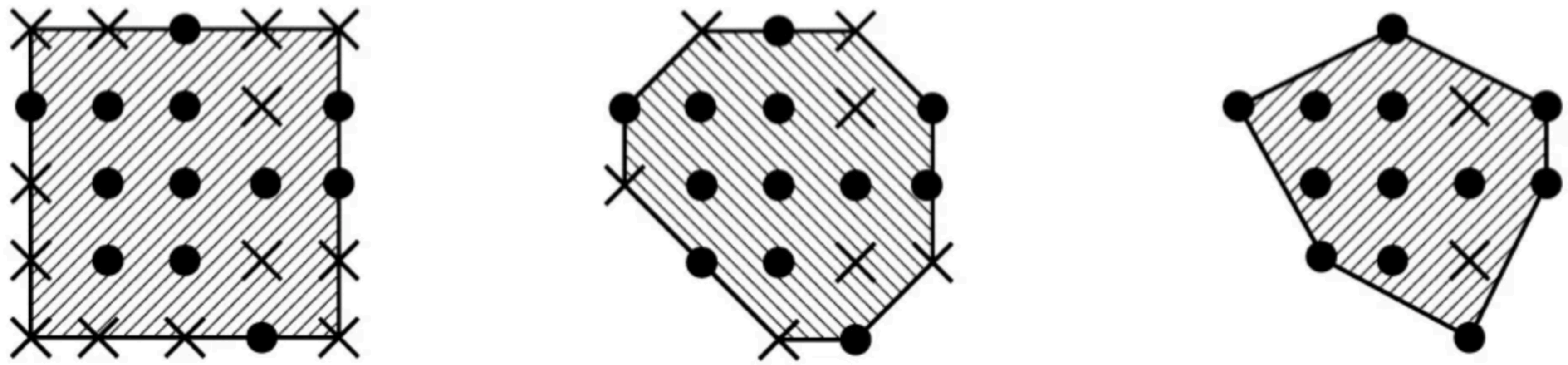
Describe the result of the interval analysis with widening and narrowing

```
// 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

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

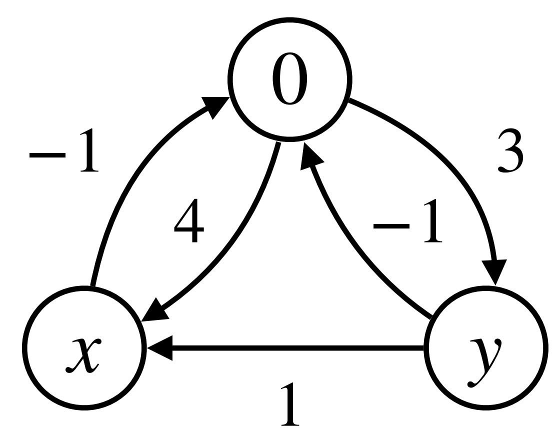
Difference Bound Matrix (DBM)

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

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

- A DBM can also be represented by a directed graph

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



Difference Bound Matrix (DBM)

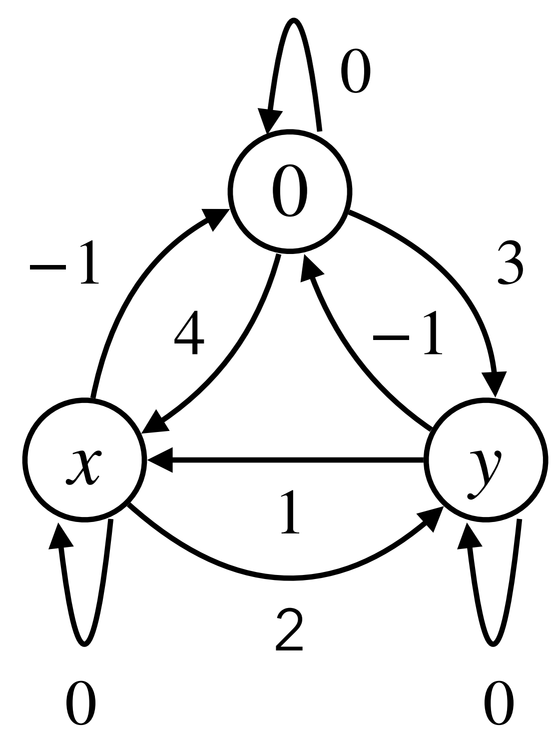
- Two different DBMs can represent the same set of points

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

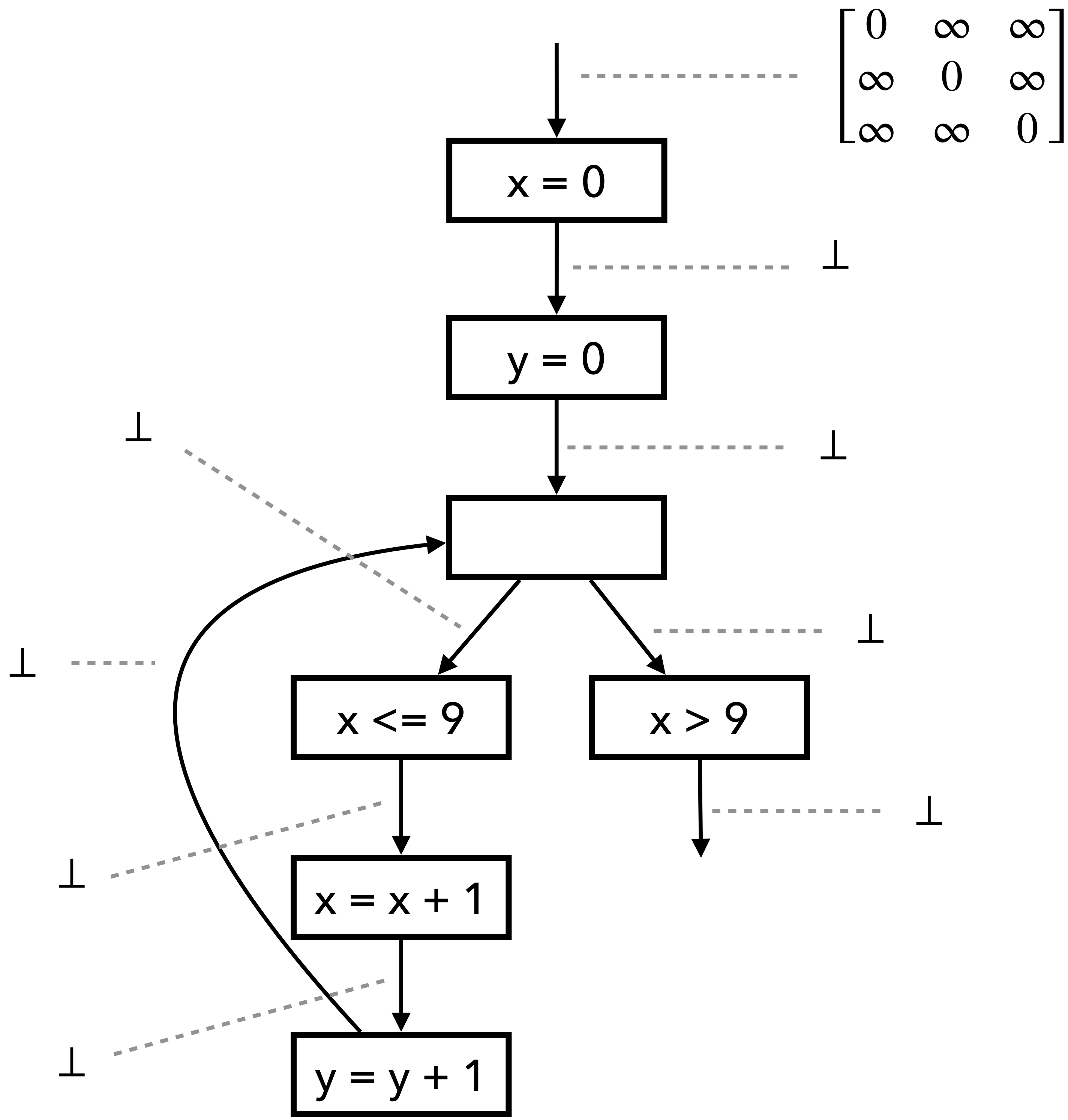
- Closure (normalization) via the Floyd-Warshall algorithm

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

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



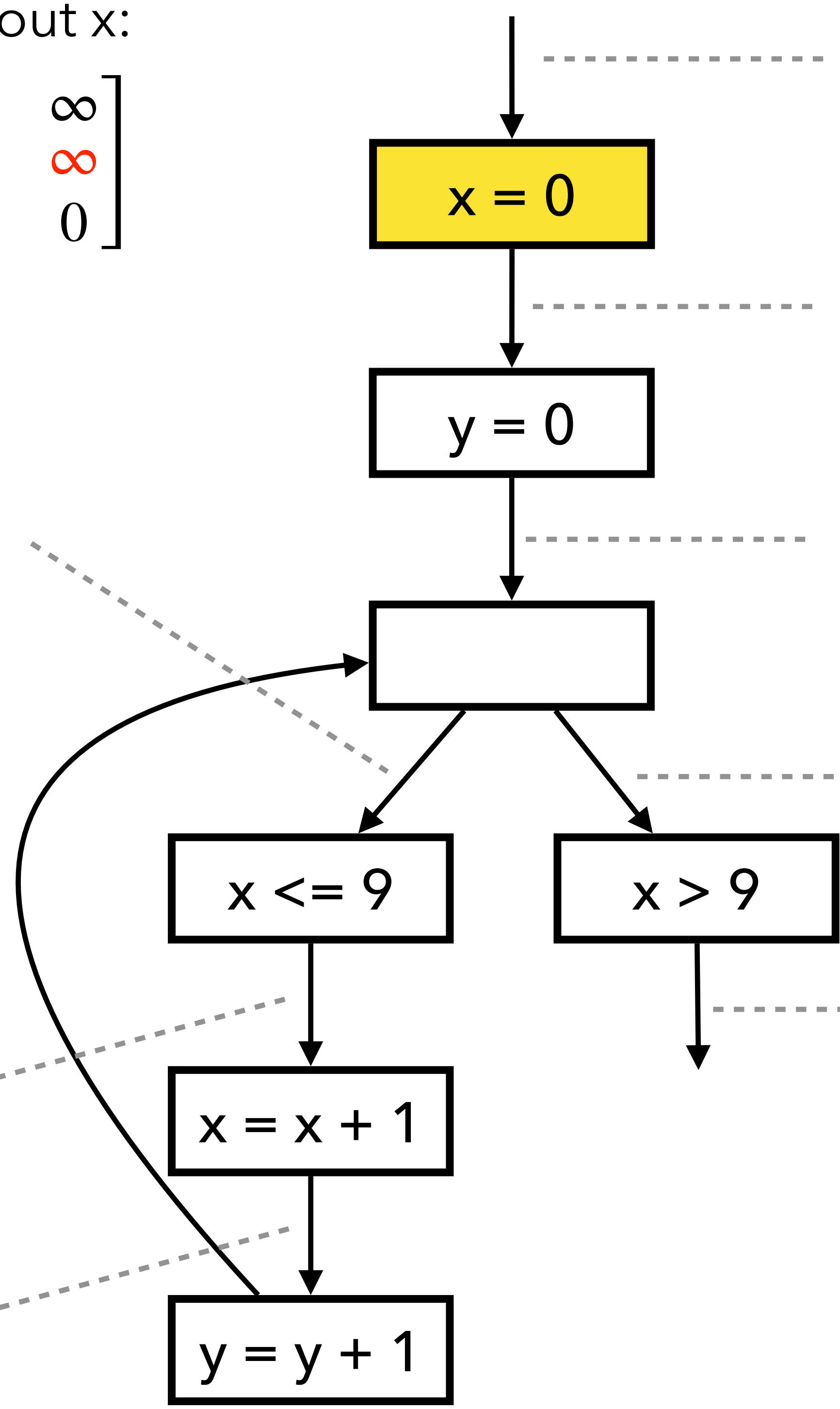
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Remove information about x:

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



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

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

$\perp$

$\perp$

$\perp$

$\perp$

$\perp$

$\perp$

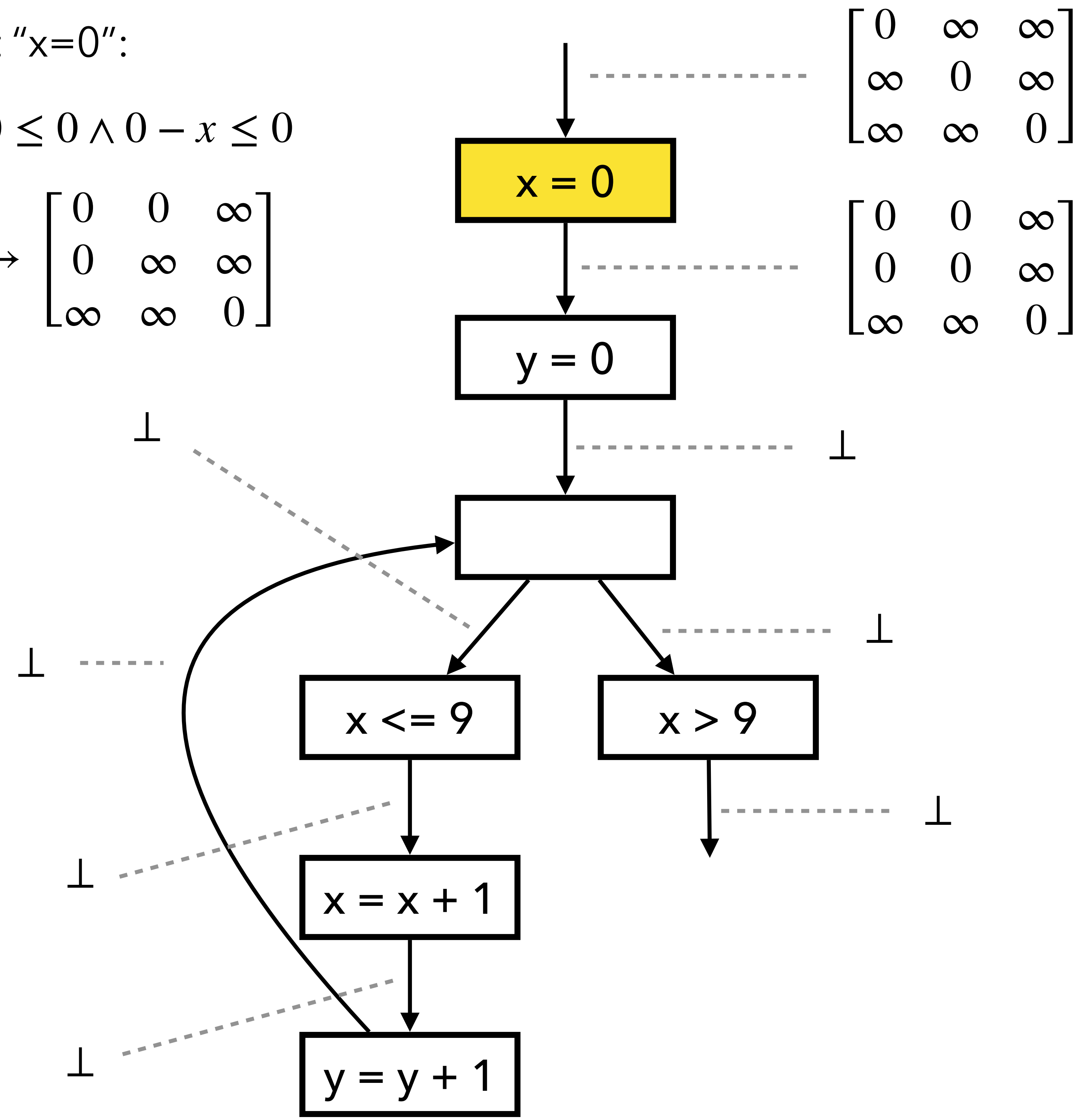
$\perp$

Fixed Point Comp. with Widening

2. Add constraint “x=0”:

$$x = 0 \iff x - 0 \leq 0 \wedge 0 - x \leq 0$$

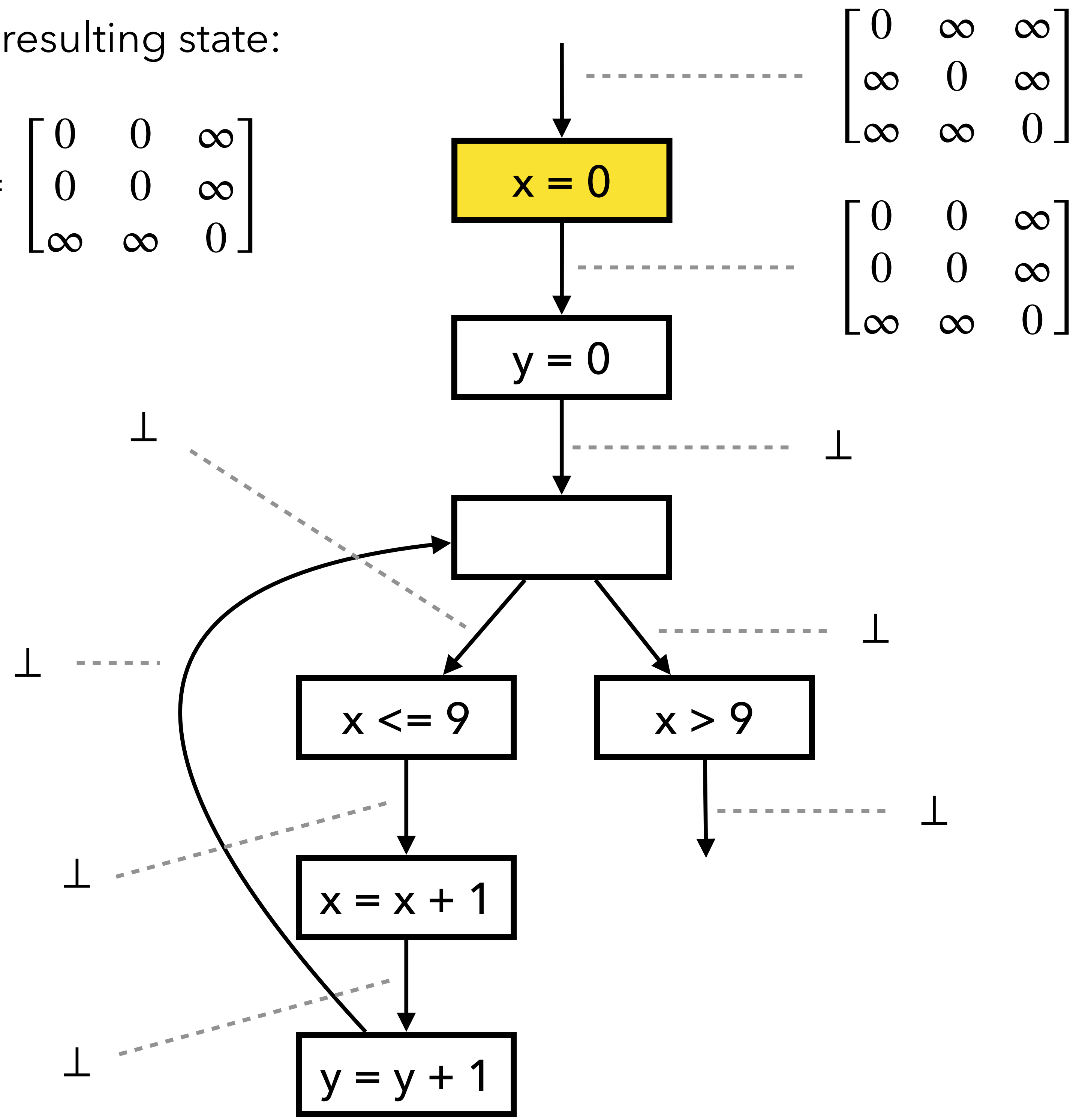
$$\begin{bmatrix} 0 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & \infty \\ 0 & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

3. Normalize the resulting state:

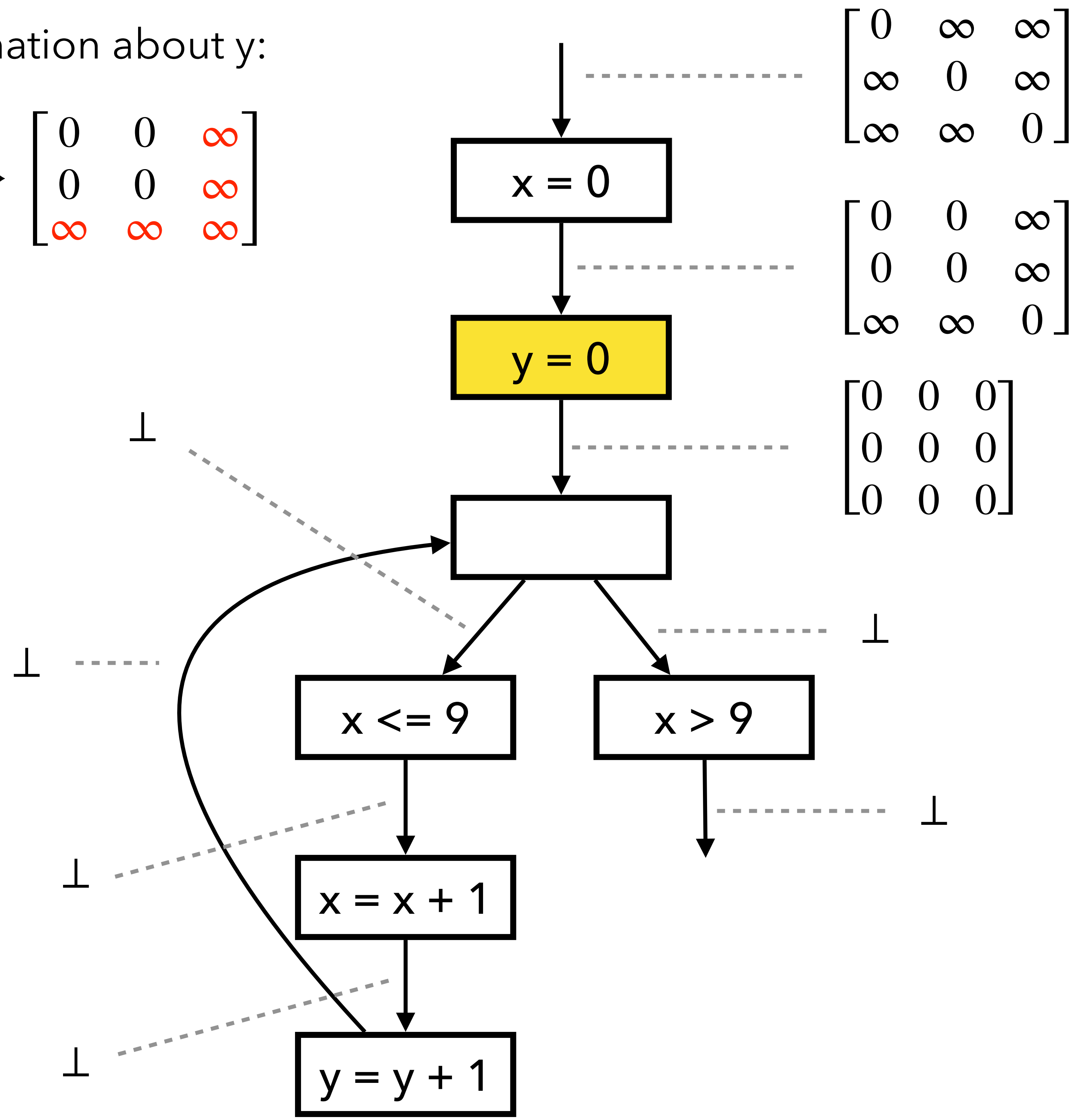
$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & \infty & \infty \\ \infty & \infty & 0 \end{bmatrix}^* = \begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

1. Remove information about y:

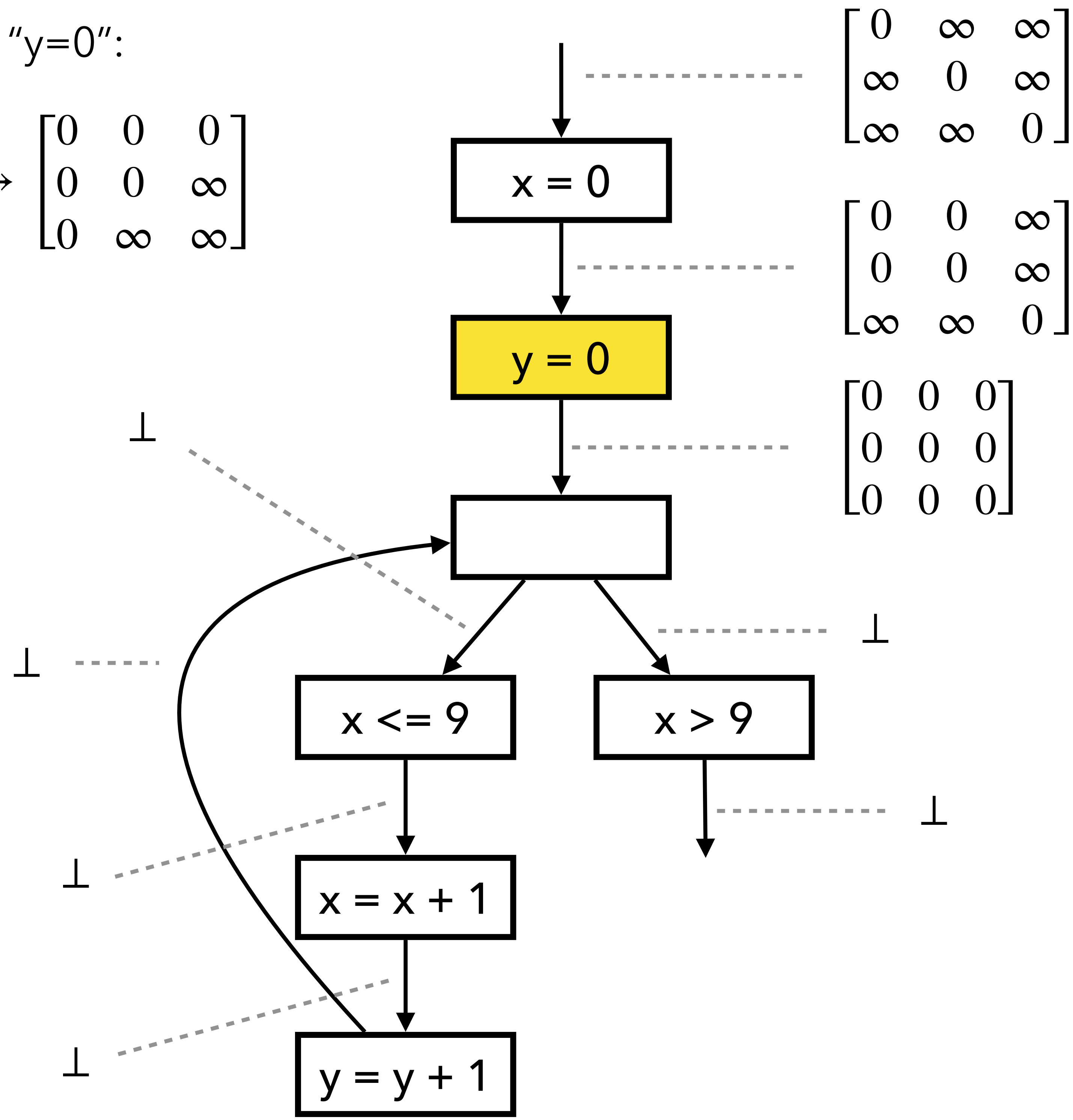
$$\begin{bmatrix}0&0&\infty\\0&0&\infty\\\infty&\infty&0\end{bmatrix} \to \begin{bmatrix}0&0&\infty\\0&0&\infty\\\infty&\infty&\infty\end{bmatrix}$$



Fixed Point Comp. with Widening

2. Add constraint “y=0”:

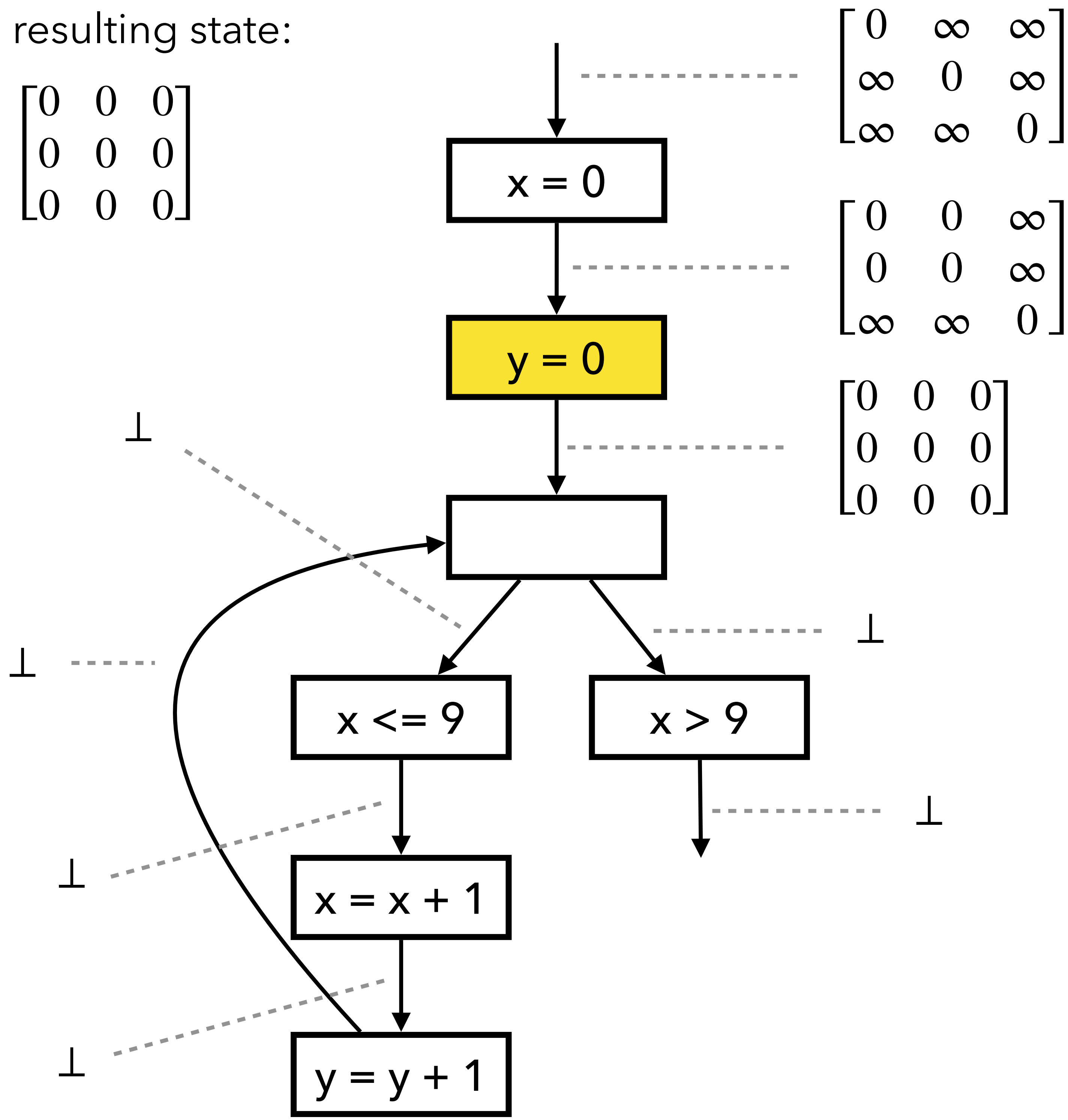
$$\begin{bmatrix} 0 & 0 & \infty \\ 0 & 0 & \infty \\ \infty & \infty & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & \infty \\ 0 & \infty & \infty \end{bmatrix}$$



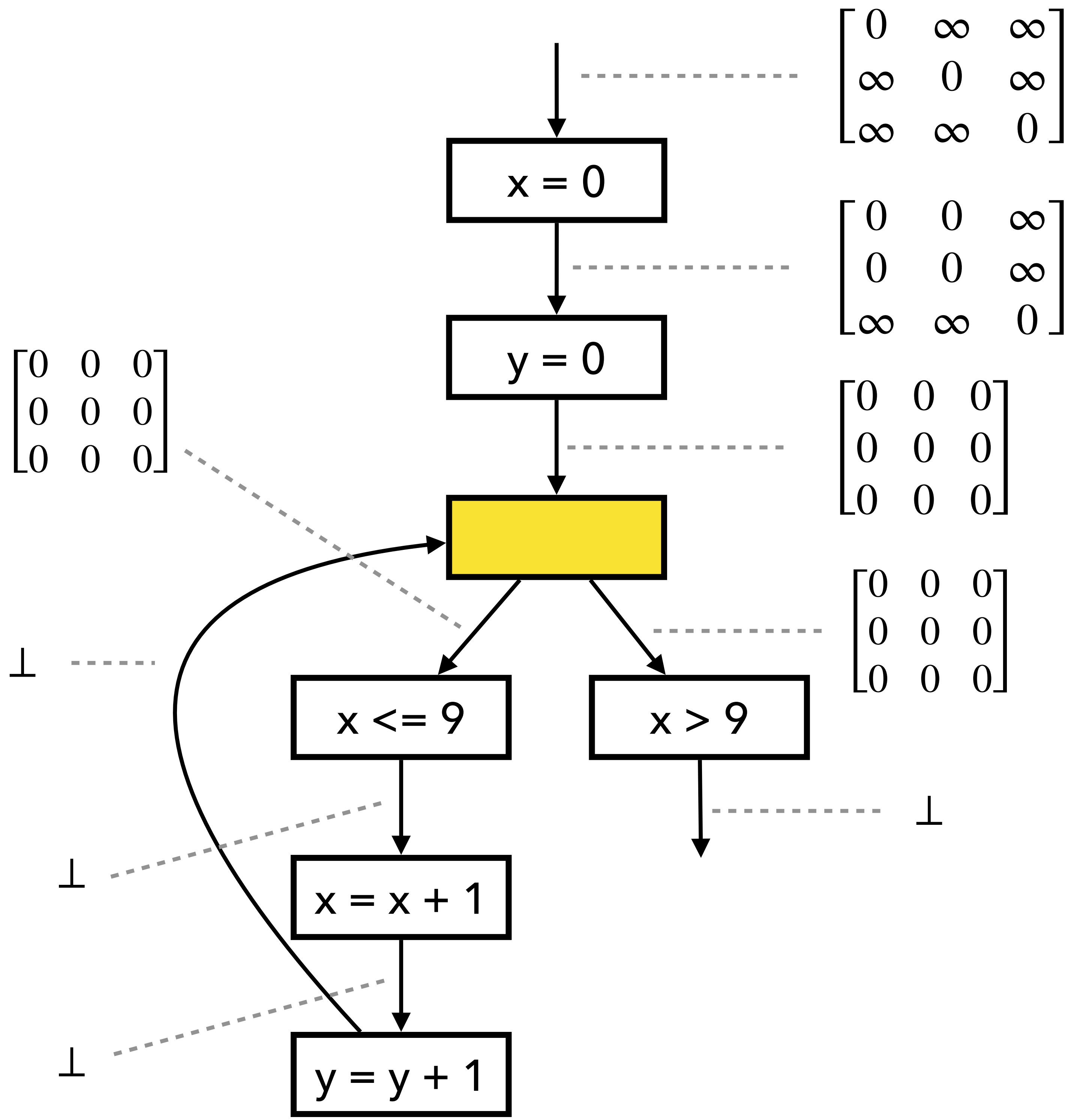
Fixed Point Comp. with Widening

3. Normalize the resulting state:

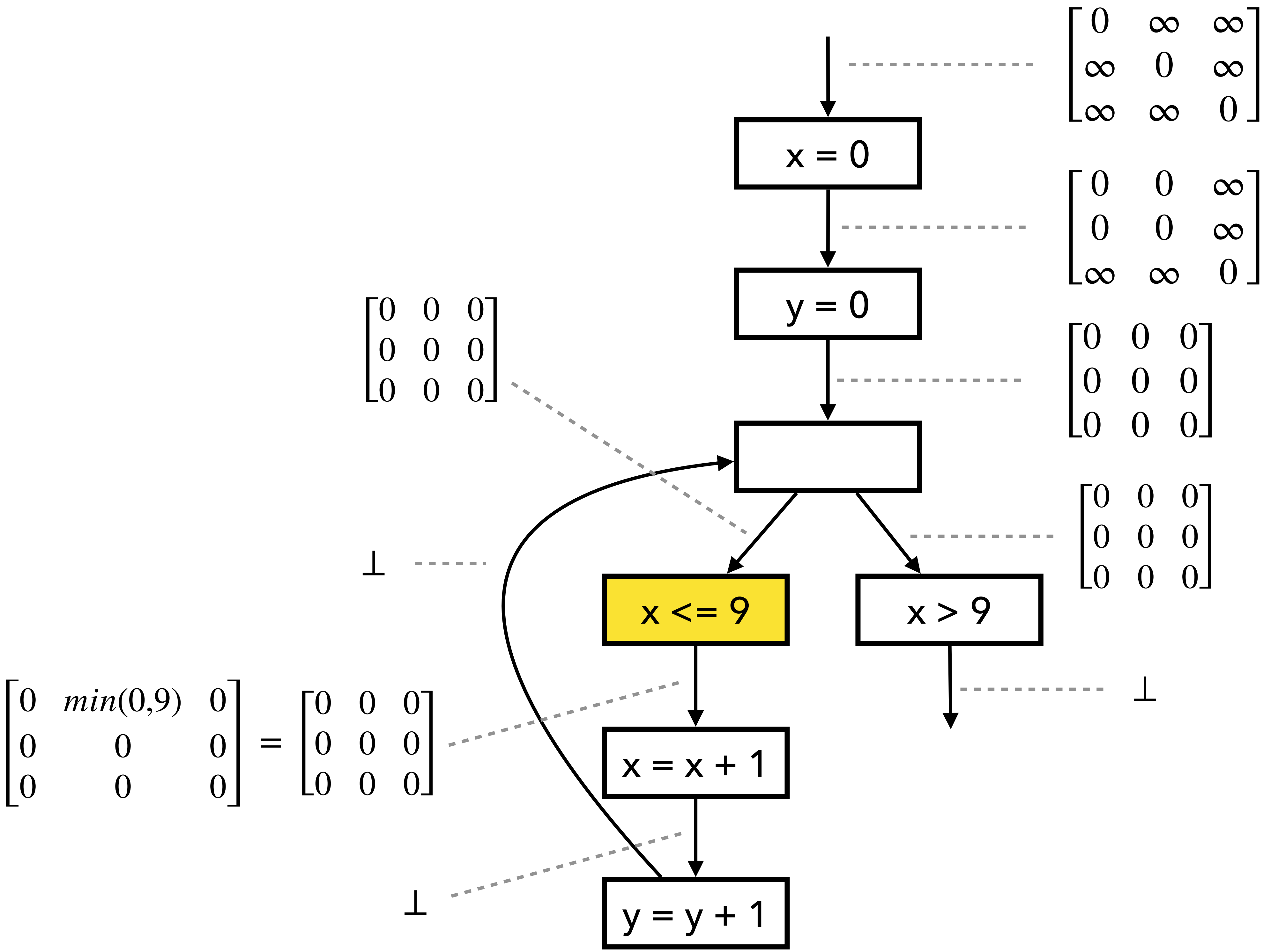
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & \infty \\ 0 & \infty & \infty \end{bmatrix}^* = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



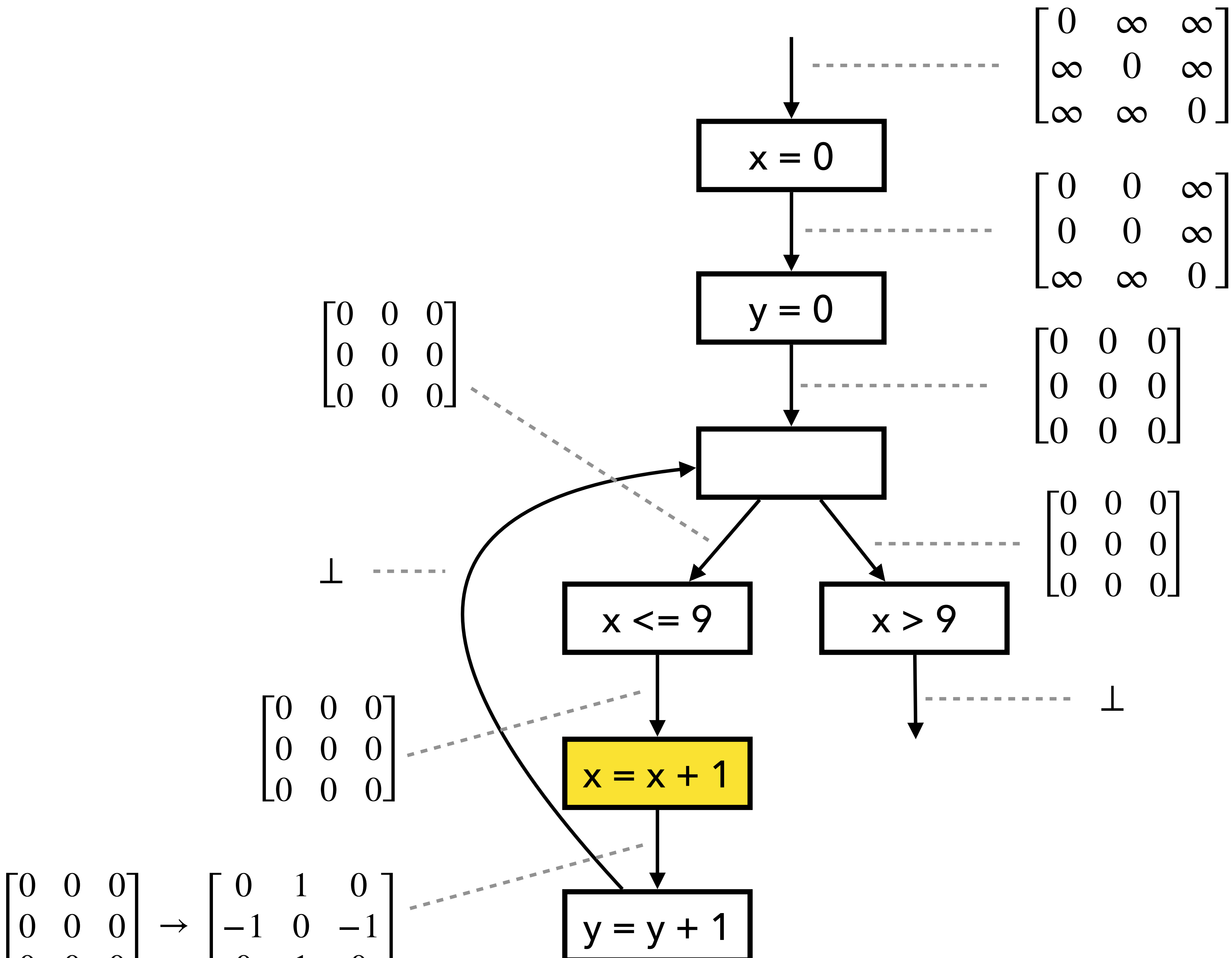
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening



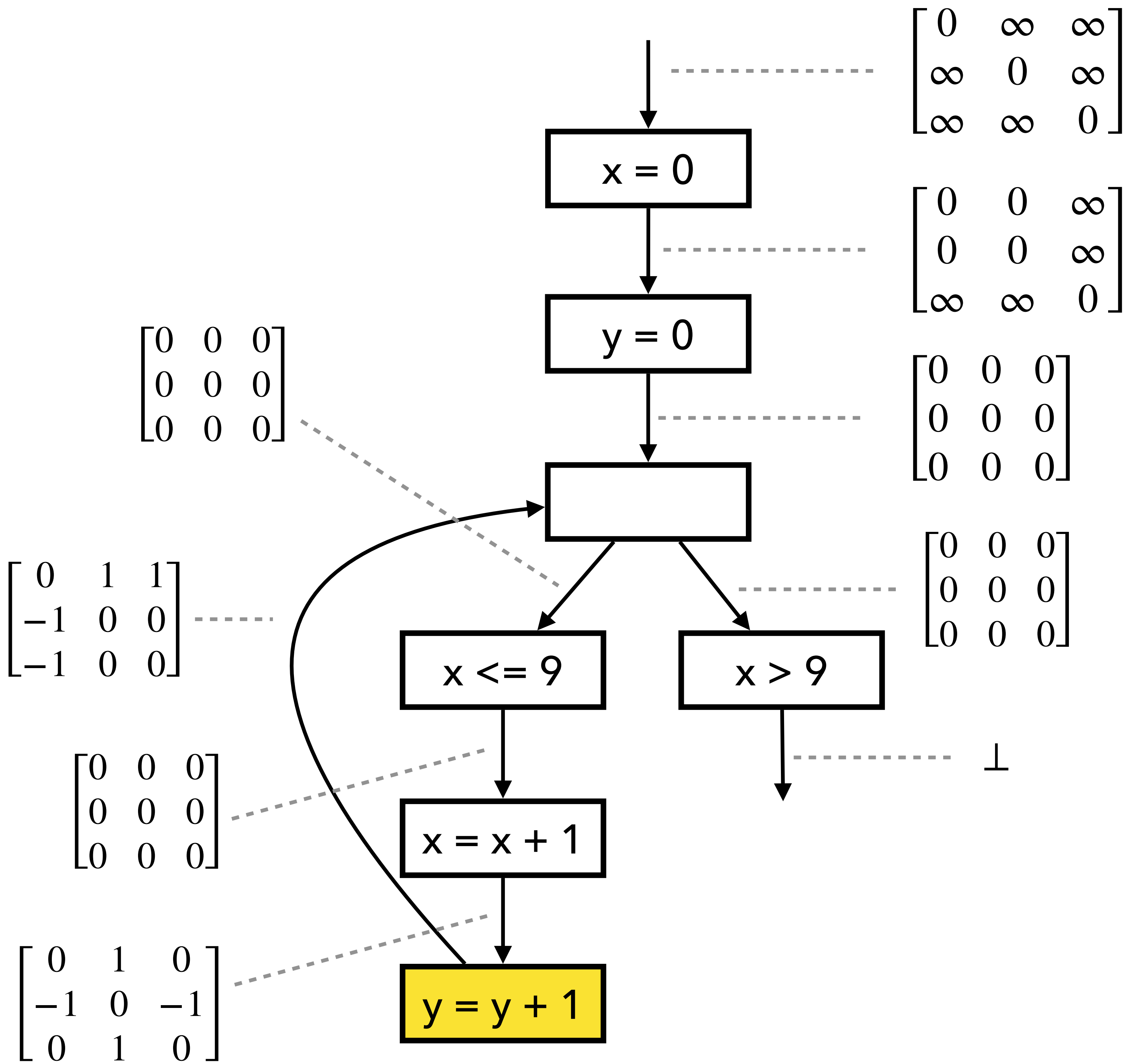
Fixed Point Comp. with Widening



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

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

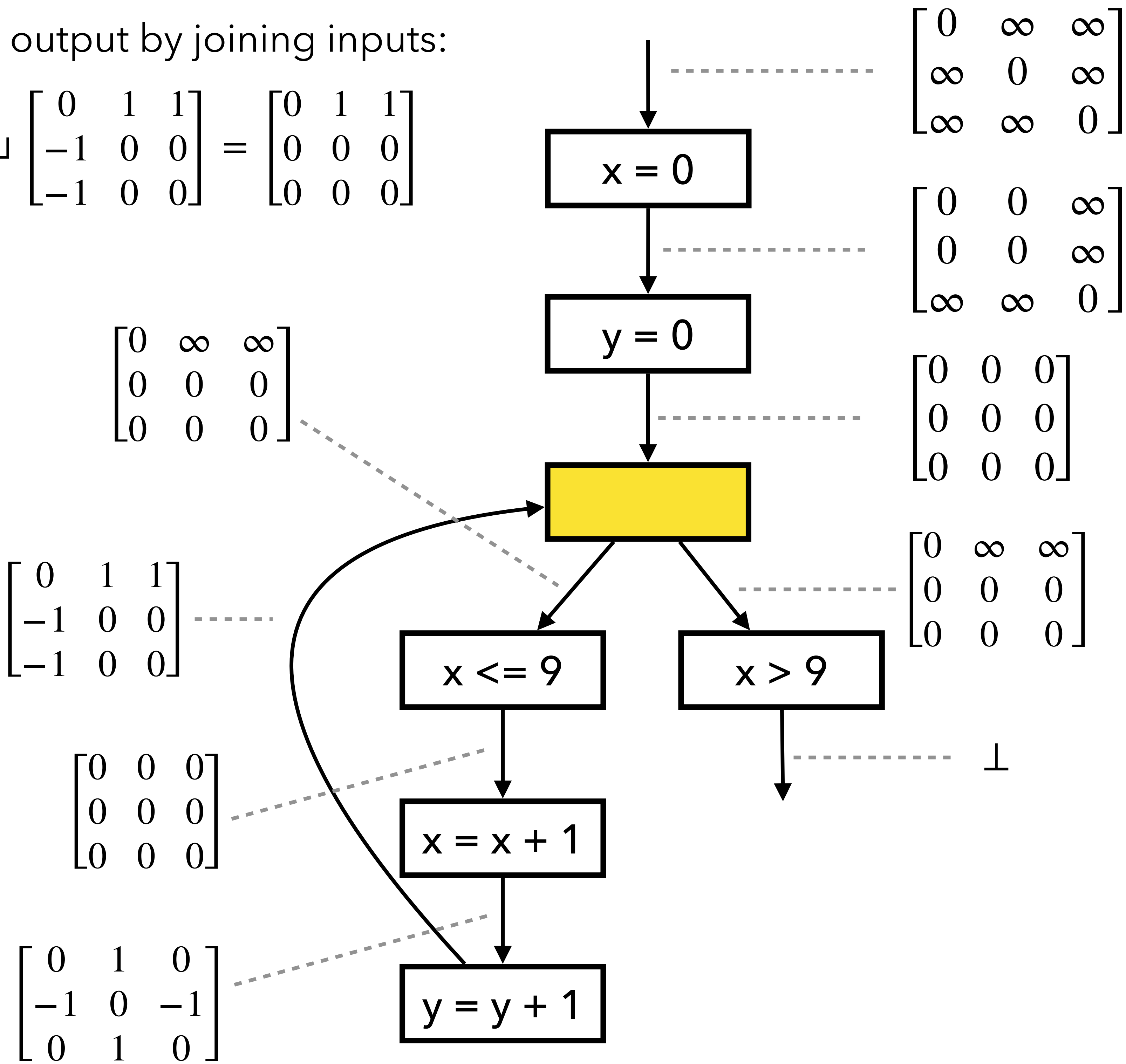
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

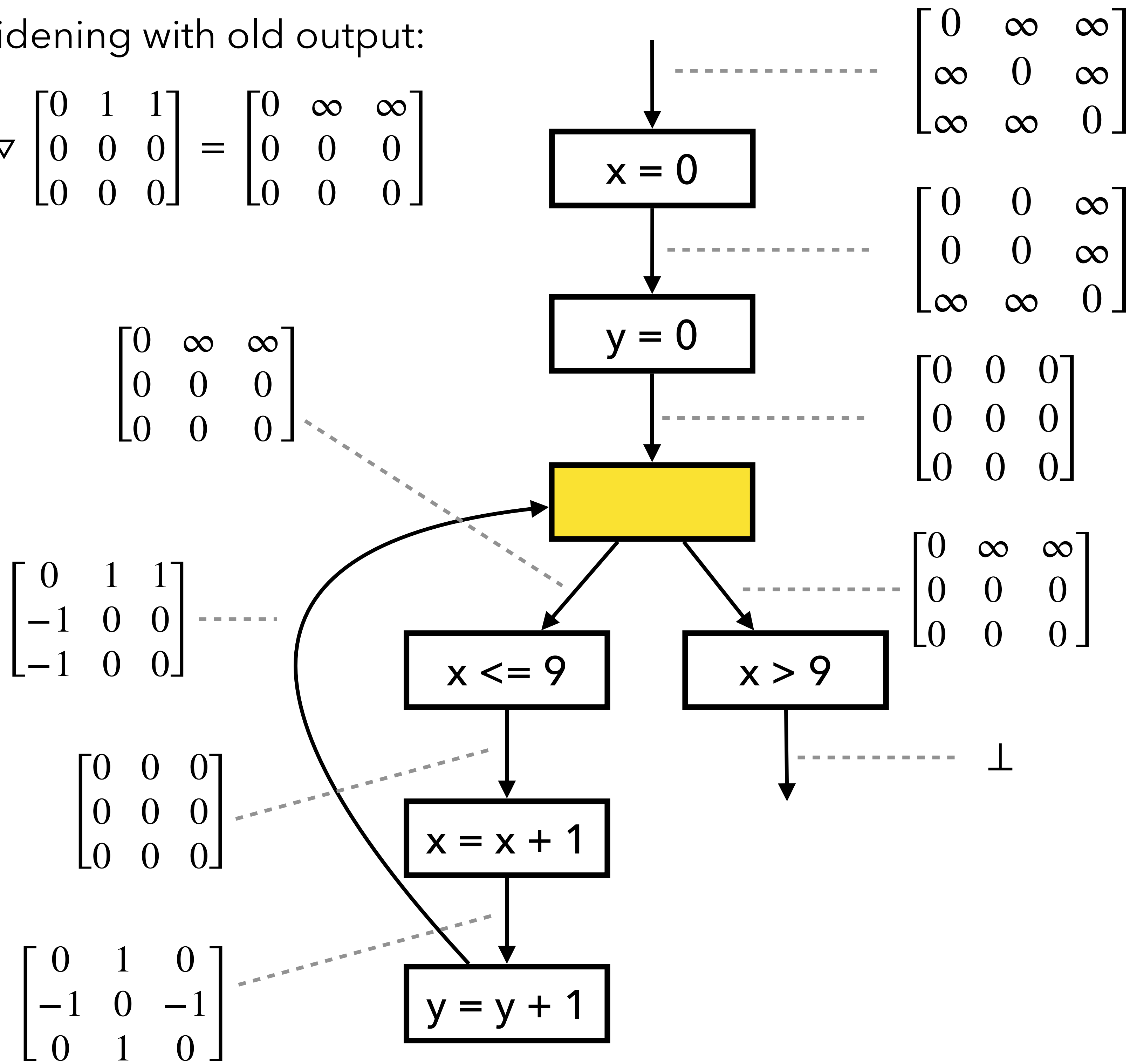
$$\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} \sqcup \begin{bmatrix}0 & 1 & 1\\-1 & 0 & 0\\-1 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & 1 & 1\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

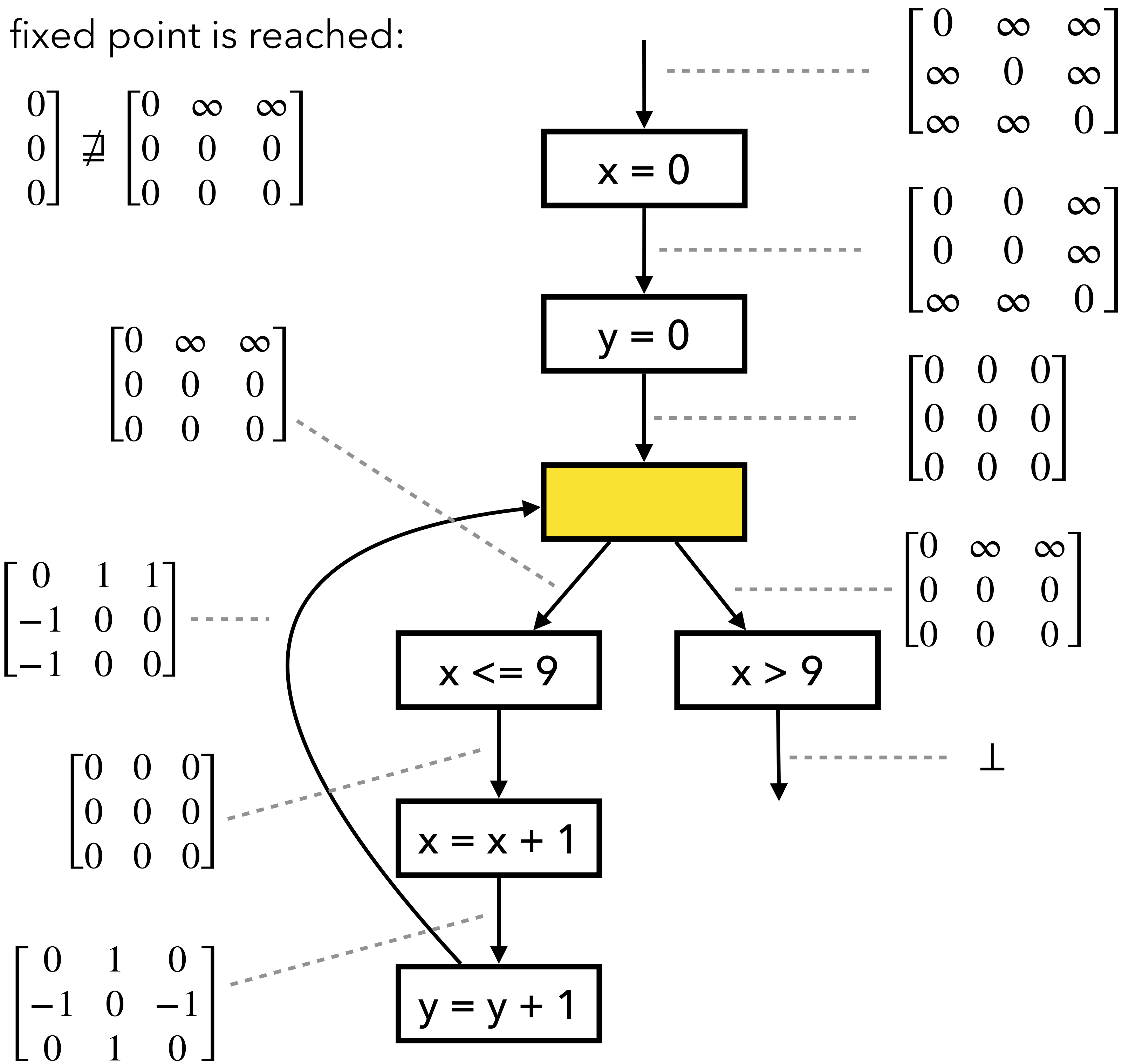
$$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix} \triangledown \begin{bmatrix}0&1&1\\0&0&0\\0&0&0\end{bmatrix} = \begin{bmatrix}0&\infty&\infty\\0&0&0\\0&0&0\end{bmatrix}$$



Fixed Point Comp. with Widening

3. Check if fixed point is reached:

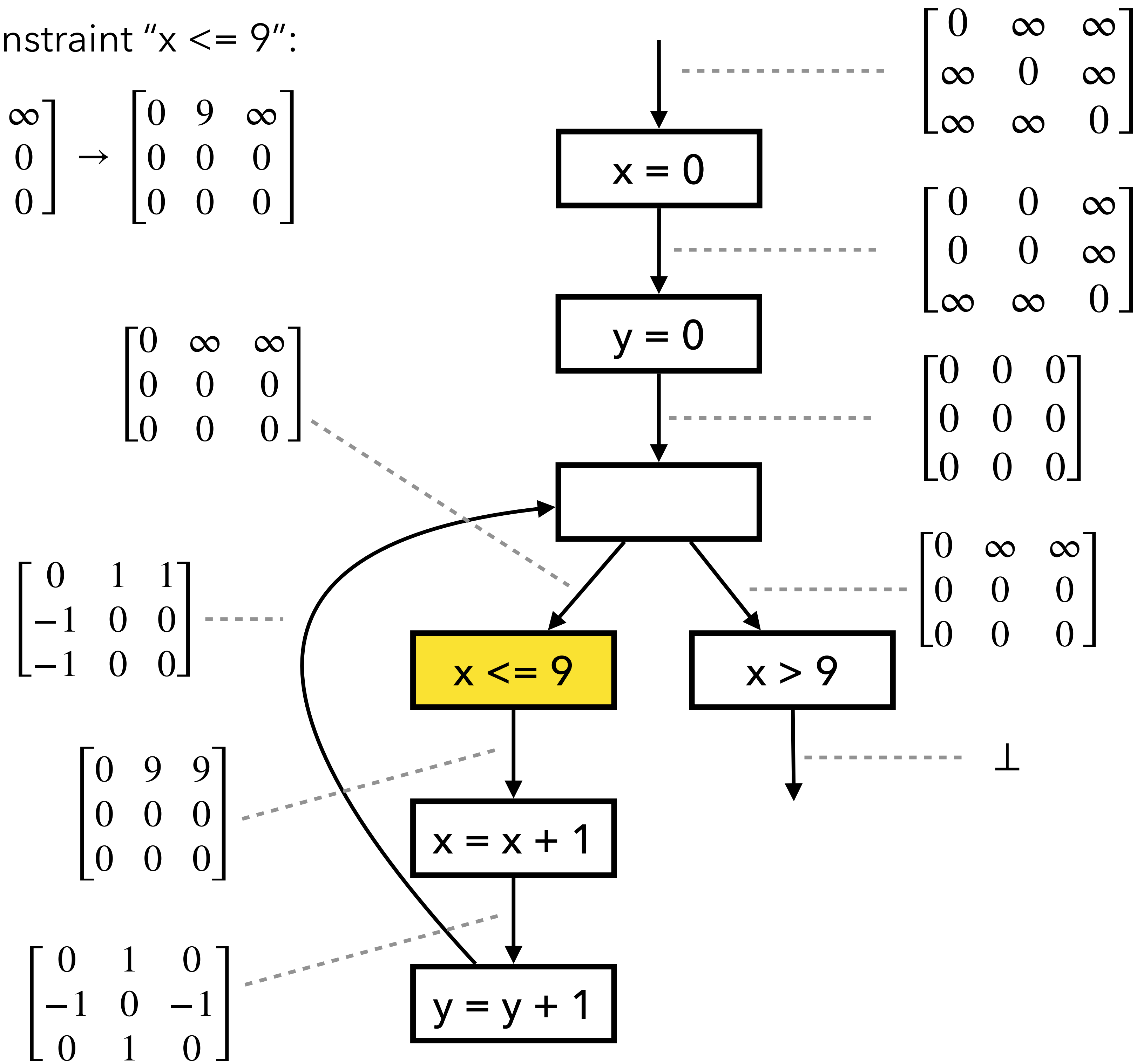
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \not\sqsupseteq \begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

1. Add constraint “x <= 9”:

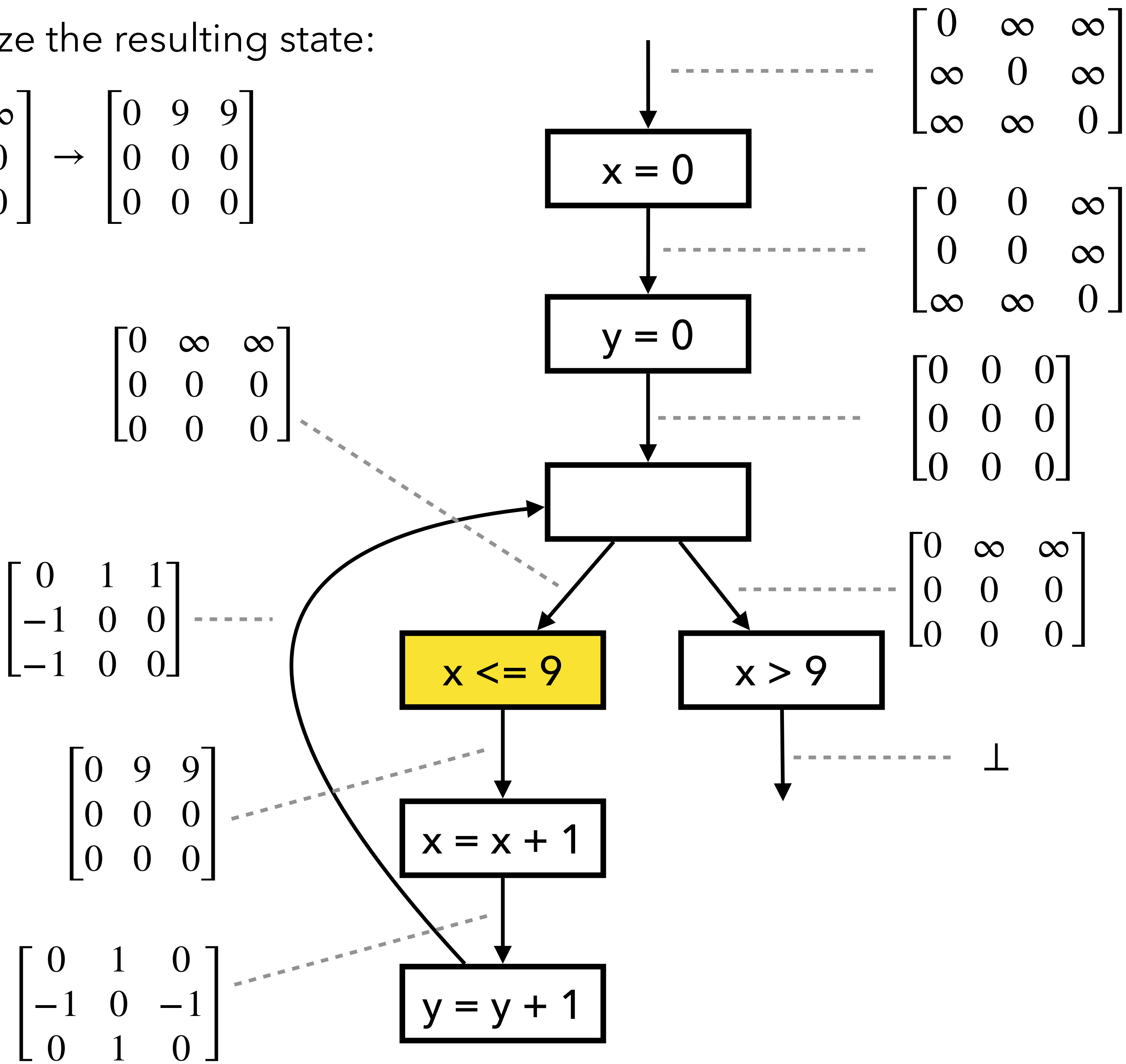
$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 9 & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



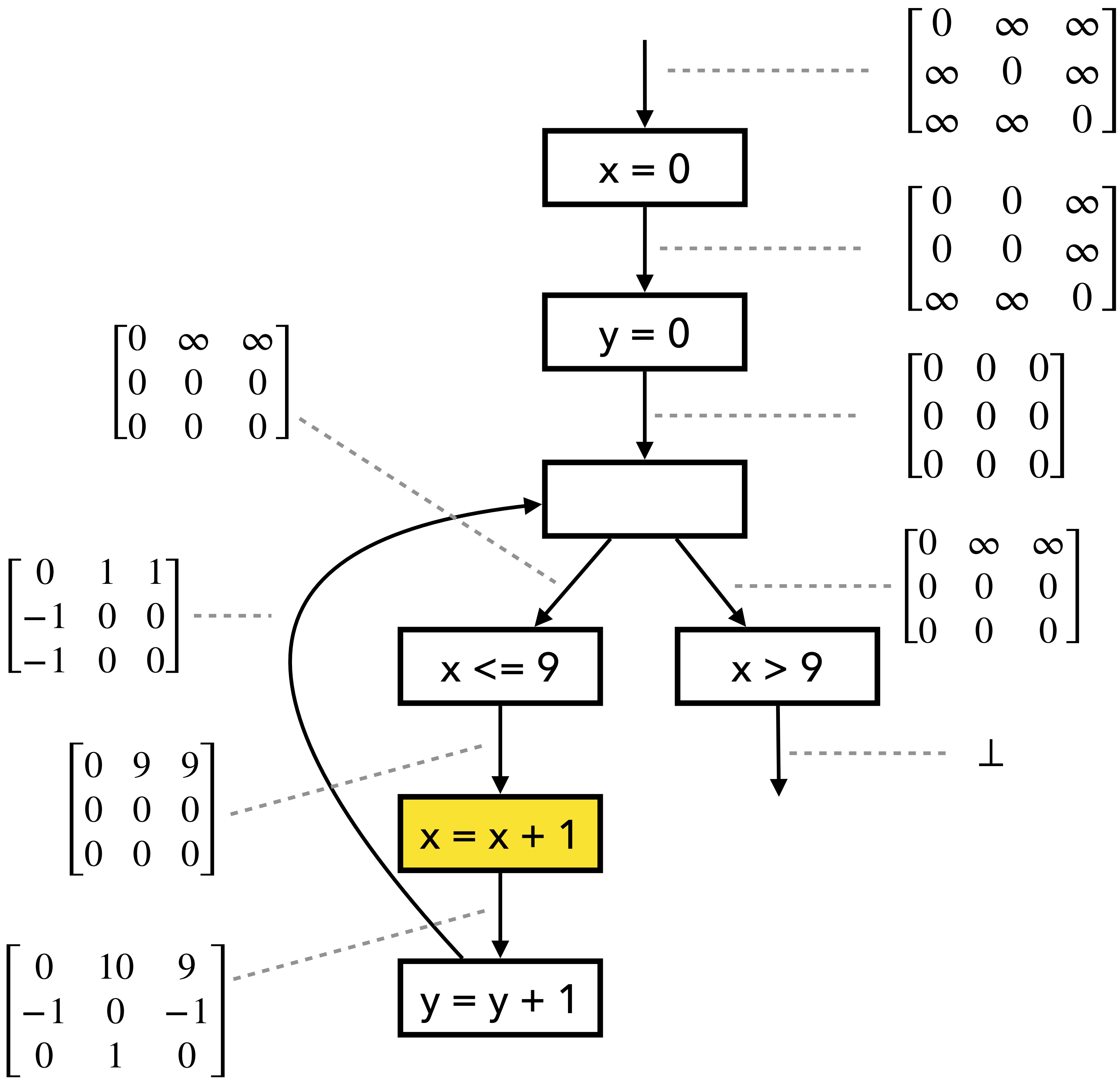
Fixed Point Comp. with Widening

2. Normalize the resulting state:

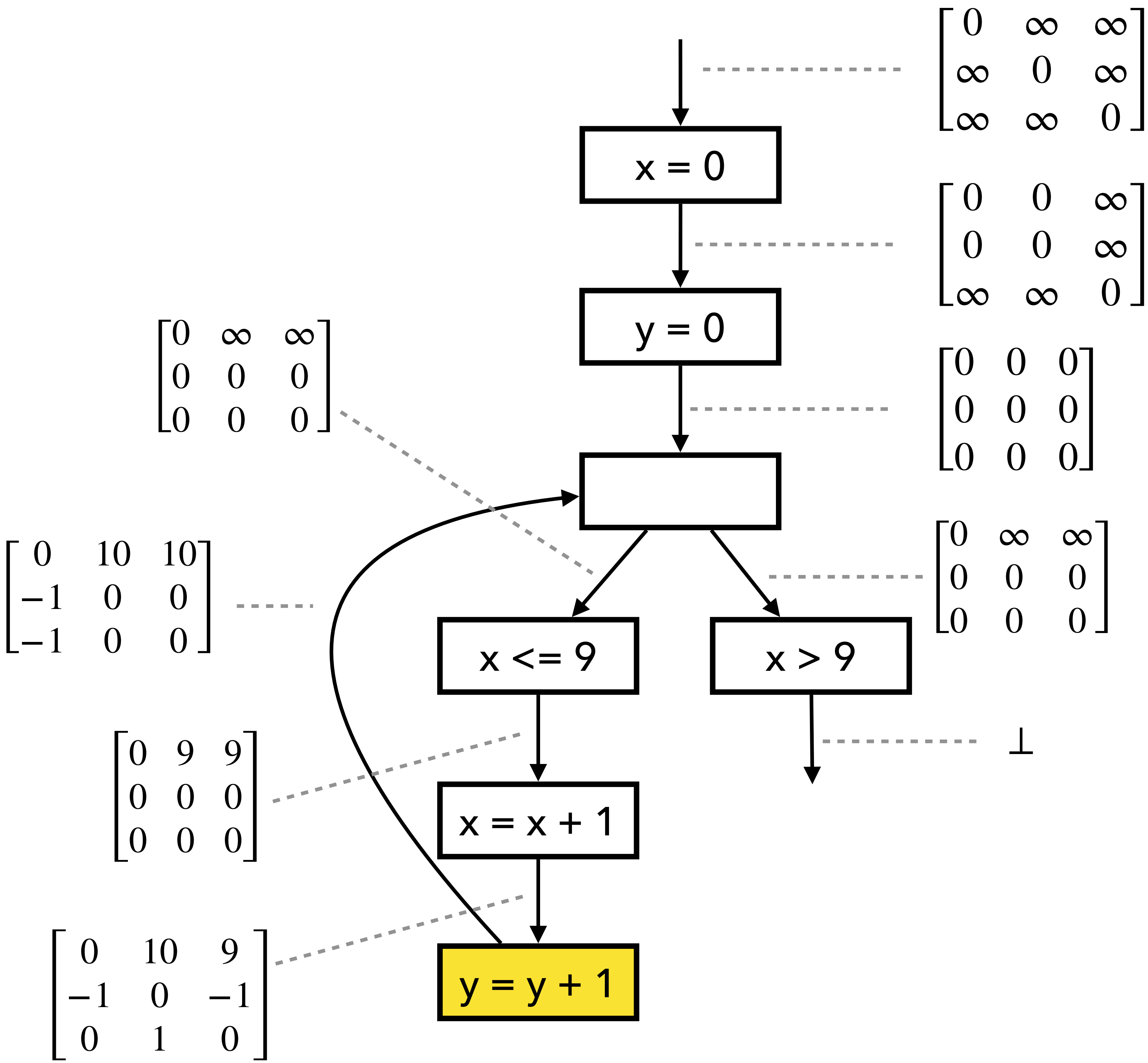
$$\begin{bmatrix} 0 & 9 & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 9 & 9 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening



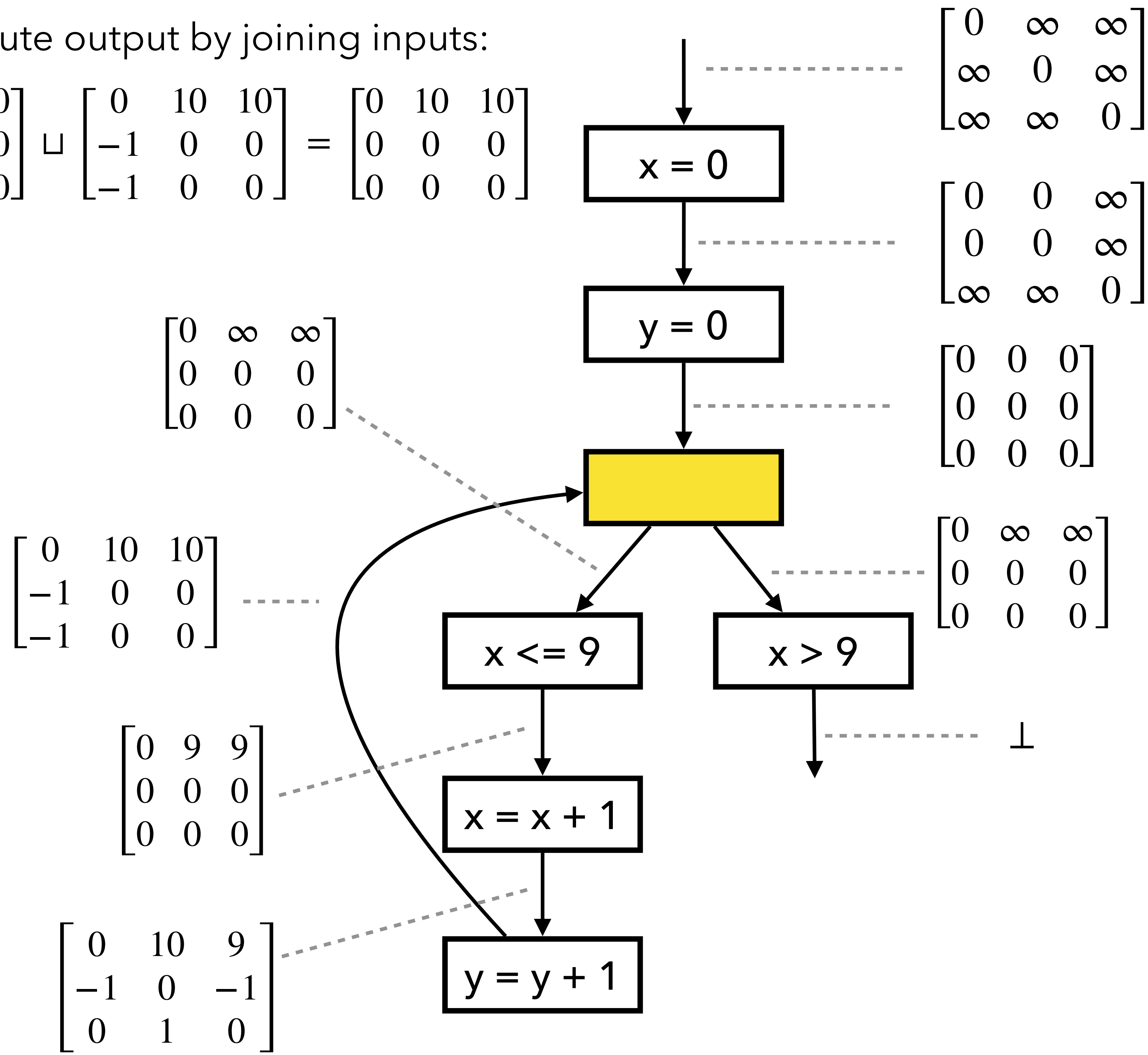
Fixed Point Comp. with Widening



Fixed Point Comp. with Widening

1. Compute output by joining inputs:

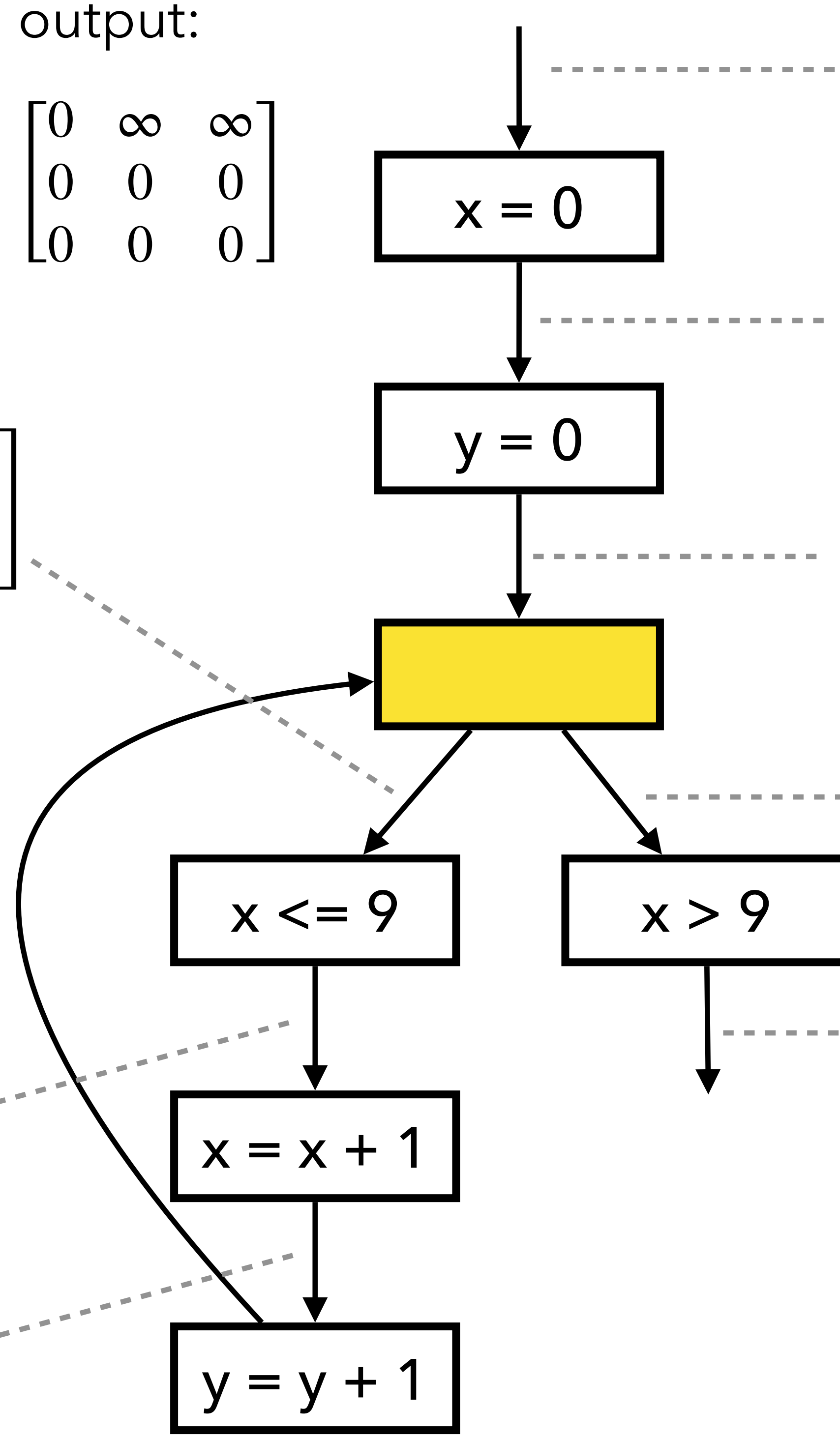
$$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix} \sqcup \begin{bmatrix}0&10&10\\-1&0&0\\-1&0&0\end{bmatrix} = \begin{bmatrix}0&10&10\\0&0&0\\0&0&0\end{bmatrix}$$



Fixed Point Comp. with Widening

2. Apply widening with old output:

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



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

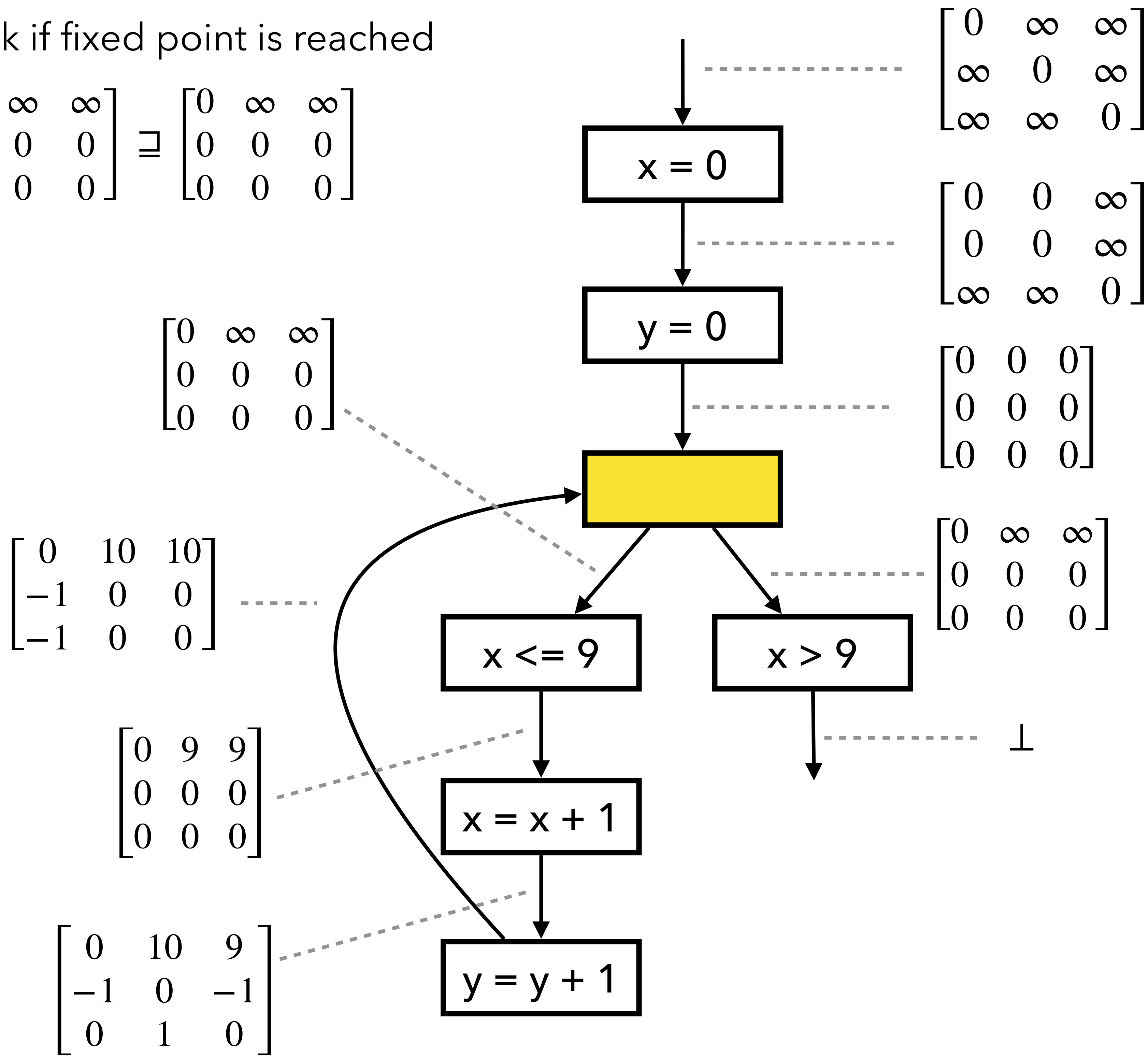
$$\begin{bmatrix}0&\infty&\infty\\0&0&0\\0&0&0\end{bmatrix}\quad \begin{bmatrix}0&10&10\\-1&0&0\\-1&0&0\end{bmatrix}\quad \begin{bmatrix}0&9&9\\0&0&0\\0&0&0\end{bmatrix}\quad \begin{bmatrix}0&10&9\\-1&0&-1\\0&1&0\end{bmatrix}$$

$\perp$

Fixed Point Comp. with Widening

3. Check if fixed point is reached

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

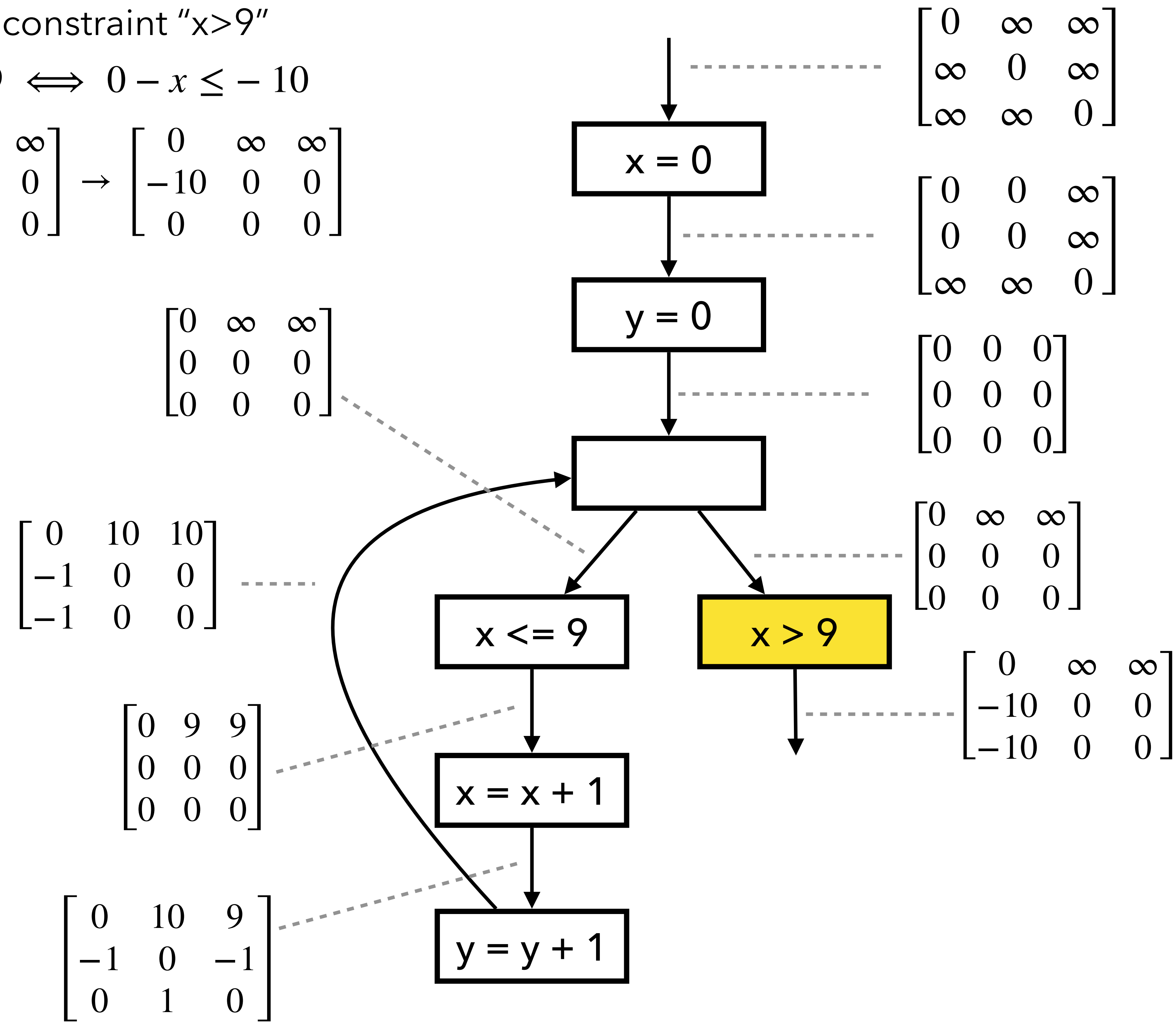


Fixed Point Comp. with Widening

1. Add constraint "x>9"

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

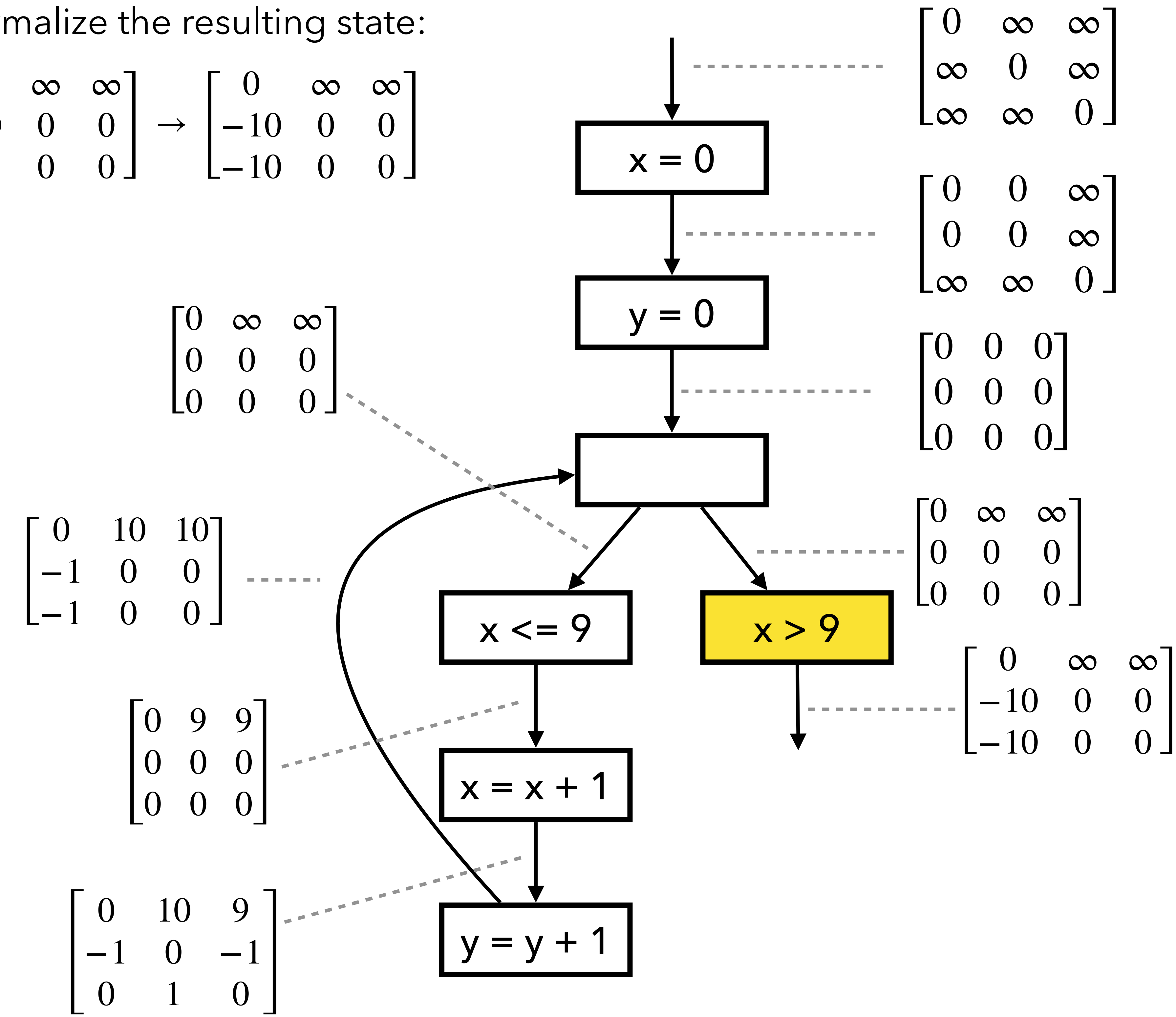
$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Widening

2. Normalize the resulting state:

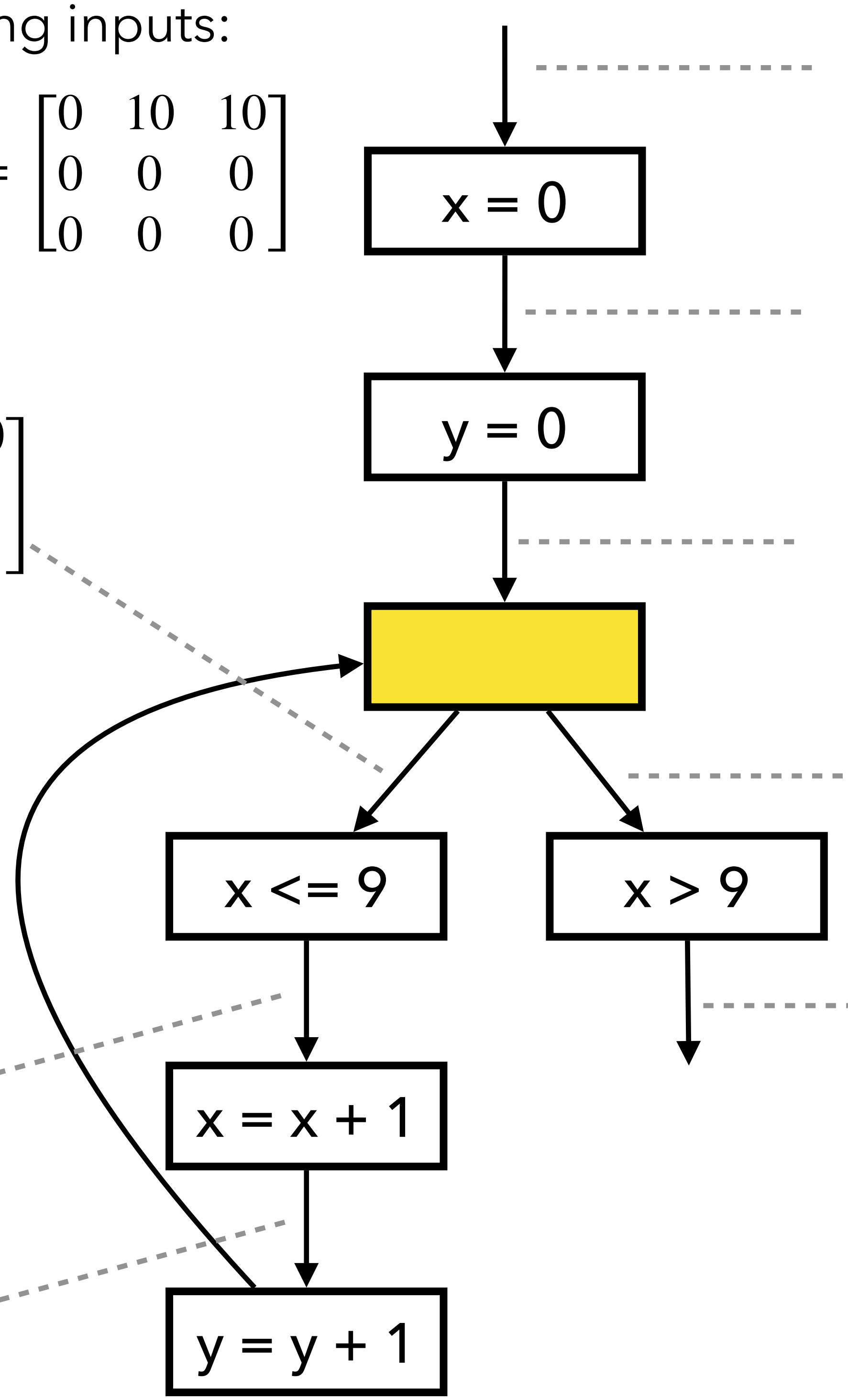
$$\begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ -10 & 0 & 0 \end{bmatrix}$$



Fixed Point Comp. with Narrowing

1. Compute output by joining inputs:

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



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

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

$$\begin{bmatrix} 0 & 10 & 10 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

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

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

$$\begin{bmatrix} 0 & 10 & 10 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

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

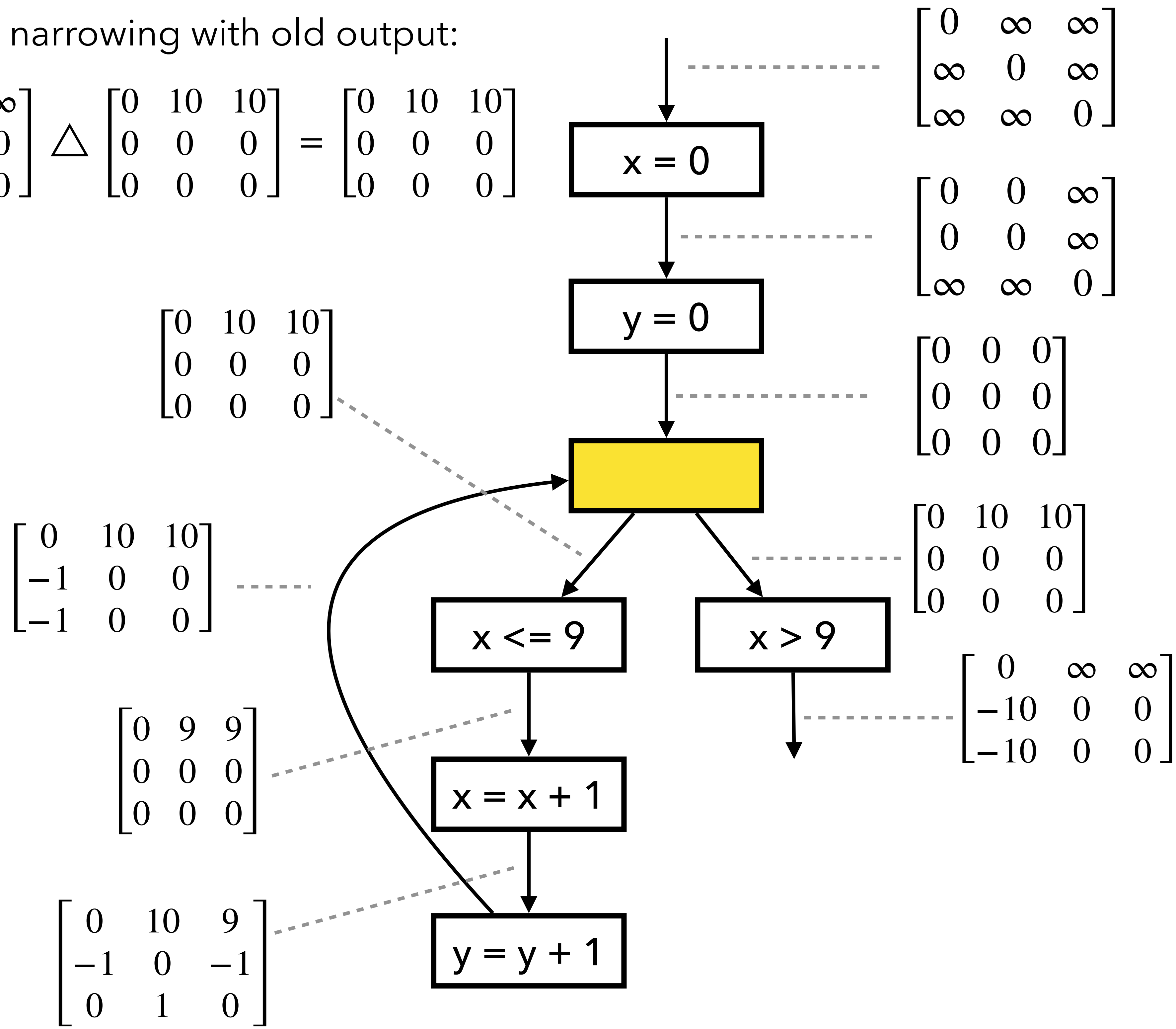
$$\begin{bmatrix} 0 & \infty & \infty \\ -10 & 0 & 0 \\ -10 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 10 & 9 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

Fixed Point Comp. with Narrowing

2. Apply narrowing with old output:

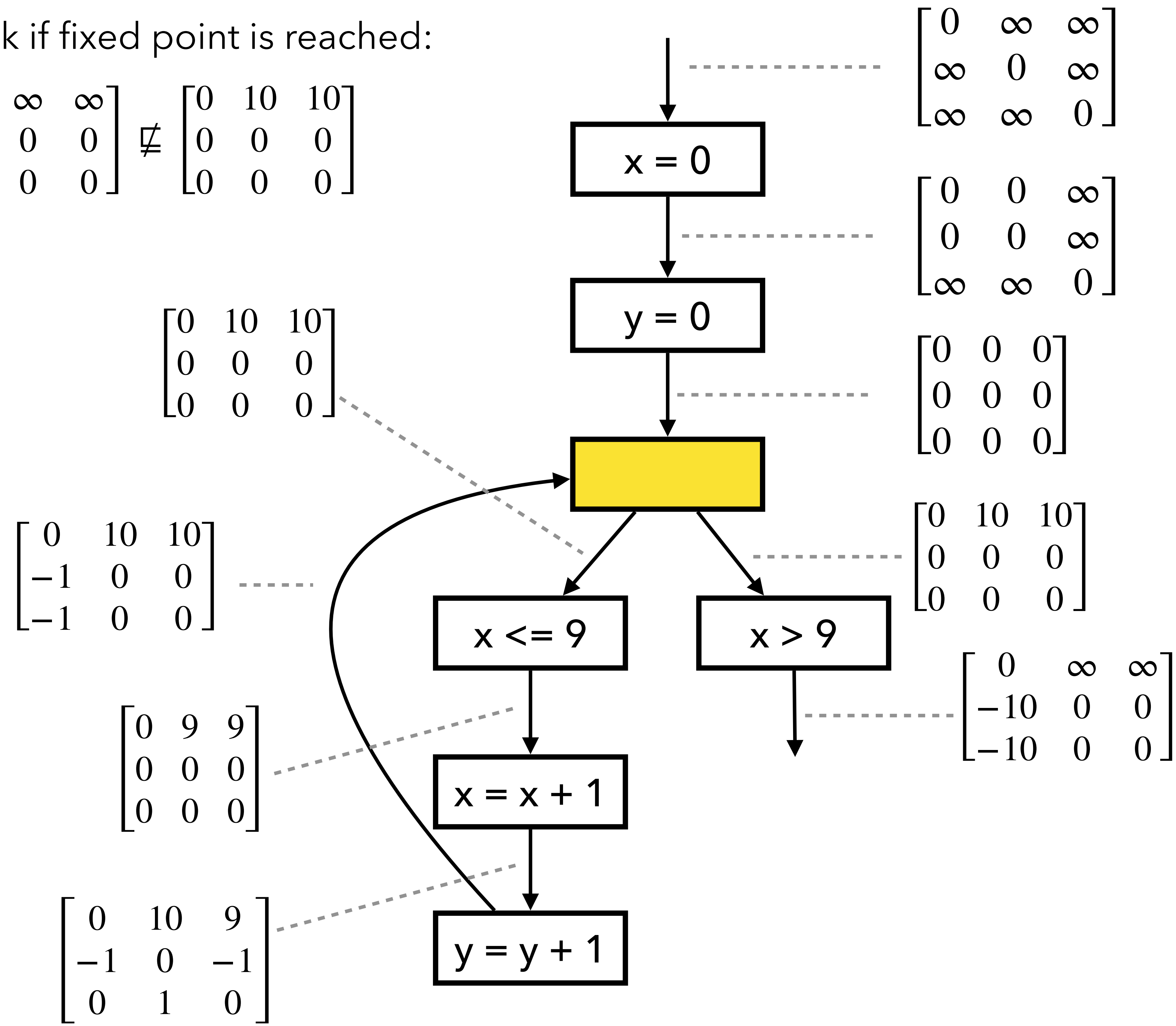
$$\begin{bmatrix} 0 & \infty & \infty \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \triangle \begin{bmatrix} 0 & 10 & 10 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 10 & 10 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



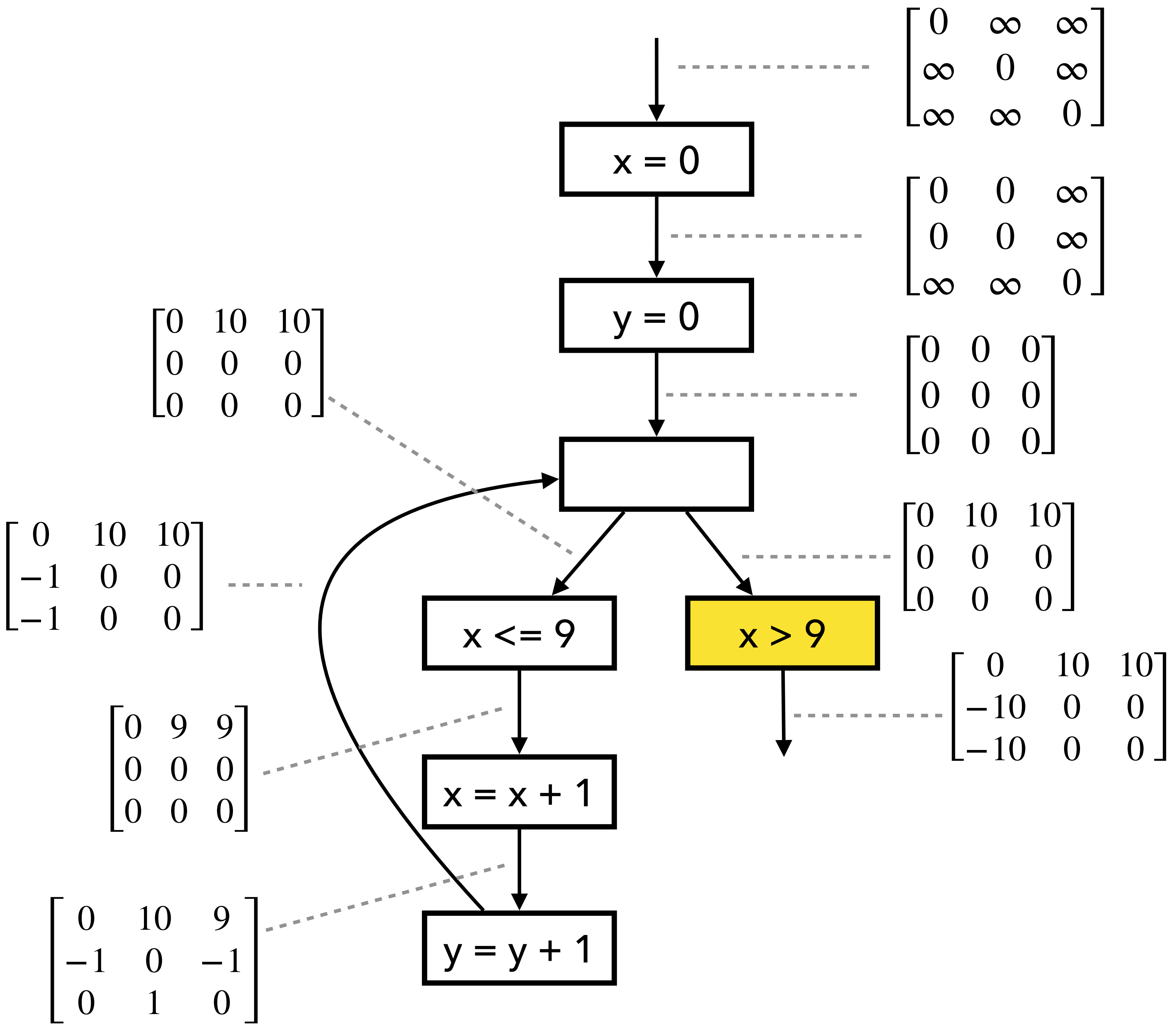
Fixed Point Comp. with Narrowing

3. Check if fixed point is reached:

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



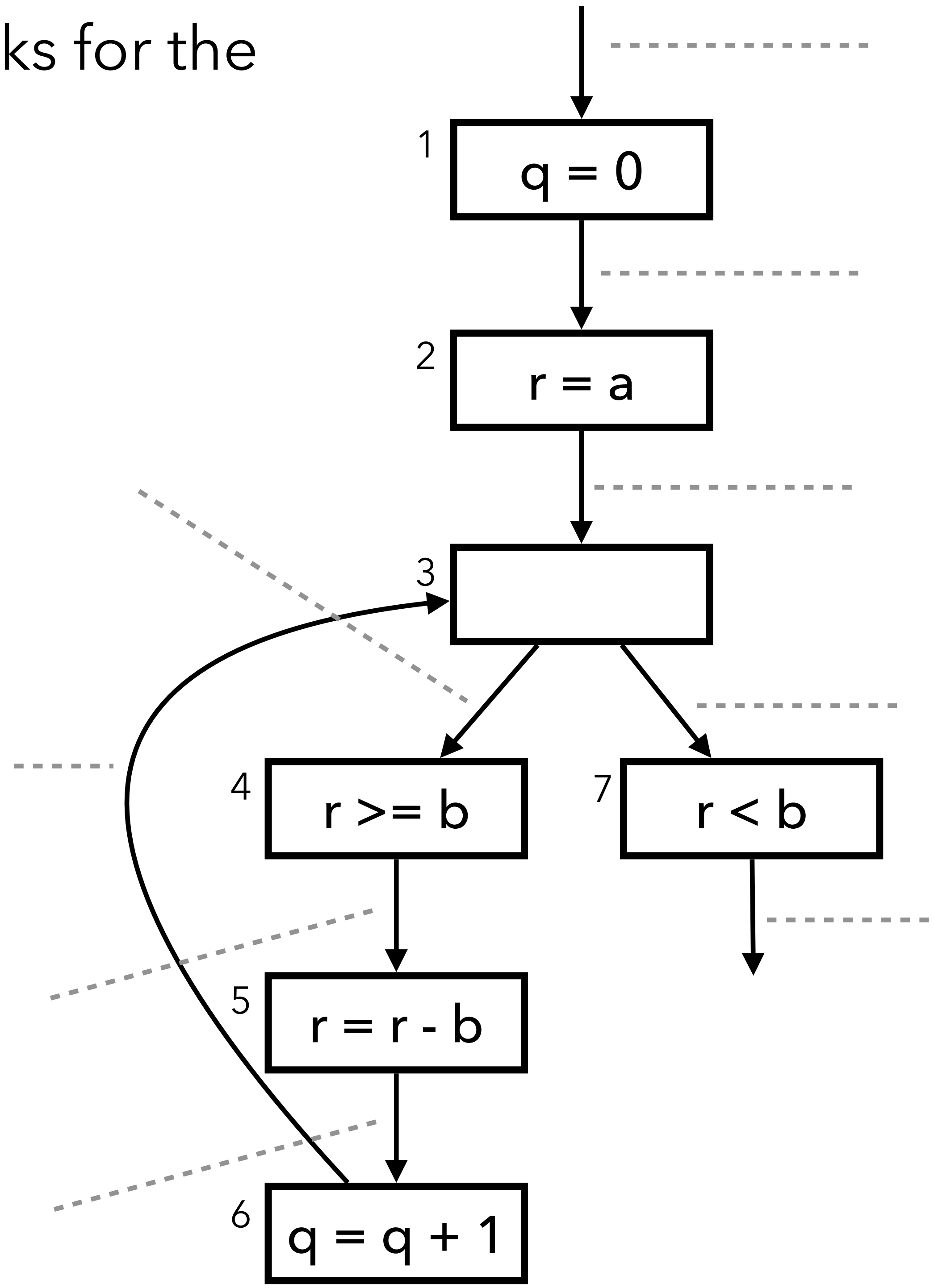
Fixed Point Comp. with Narrowing



Motivating Example

Describe how the zone analysis works for the following example.

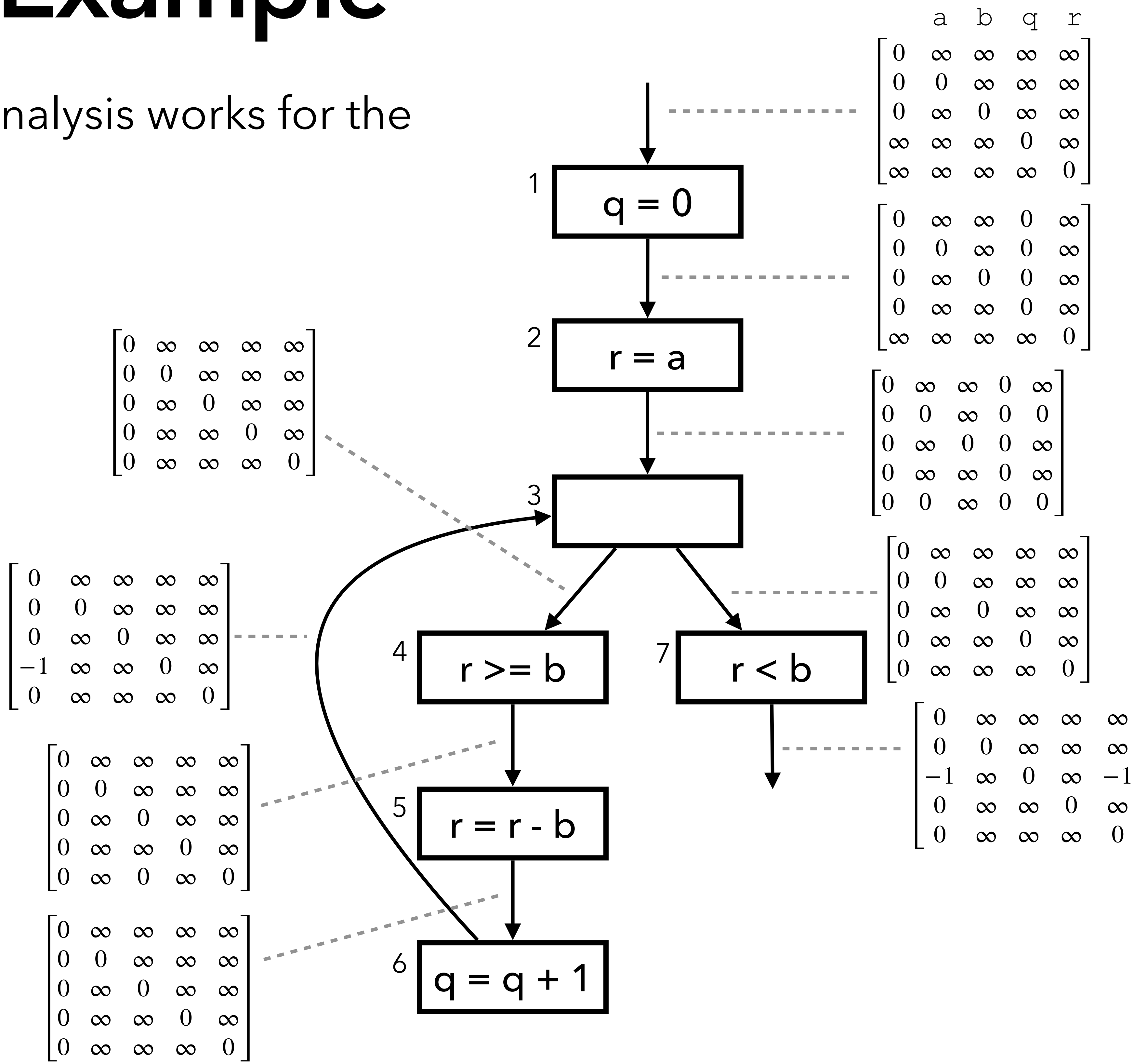
```
// a >= 0, b >= 0
q = 0;
r = a;
while (r >= b) {
  r = r - b;
  q = q + 1;
}
assert(q >= 0);
assert(r >= 0);
```



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);
```



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()}
```

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

$$
\begin{array}{lclclcl}
 & & & & & & \mathsf{Alloc}(a, l_1) \\
 & & & & \mathsf{Alloc}(a, l_1) & & \mathsf{Alloc}(b, l_2) \\
 & & \mathsf{Alloc}(a, l_1) & & \mathsf{Alloc}(b, l_2) & & \mathsf{Move}(c, a) \\
\mathsf{Alloc}(a, l_1) & & \mathsf{Alloc}(b, l_2) & & \mathsf{Move}(c, a) & & \mathsf{Store}(a, f, b) \\
\mathsf{Alloc}(b, l_2) & & \mathsf{Move}(c, a) & & \mathsf{Store}(a, f, b) & & \mathsf{Load}(d, c, f) \\
\mathsf{Move}(c, a) & \xrightarrow{(1)} & \mathsf{Store}(a, f, b) & \xrightarrow{(2),\,(3)} & \mathsf{Load}(d, c, f) & \xrightarrow{(4)} & \mathsf{VarPointsTo}(a, l_1) \\
\mathsf{Store}(a, f, b) & & \mathsf{Load}(d, c, f) & & \mathsf{VarPointsTo}(a, l_1) & & \mathsf{VarPointsTo}(b, l_2) \\
\mathsf{Load}(d, c, f) & & \mathsf{VarPointsTo}(a, l_1) & & \mathsf{VarPointsTo}(b, l_2) & & \mathsf{VarPointsTo}(c, l_1) \\
 & & \mathsf{VarPointsTo}(b, l_2) & & \mathsf{VarPointsTo}(c, l_1) & & \mathsf{FldPointsTo}(l_1, f, l_2) \\
 & & & & \mathsf{FldPointsTo}(l_1, f, l_2) & & \mathsf{VarPointsTo}(d, l_2)
\end{array}
$$

Pointer Analysis Rules

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

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

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

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

Interprocedural Analysis (First-Order)

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

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

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

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

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

$\text{Reachable}(global)$

$\text{Reachable}(m_1)$

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

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

Interprocedural Analysis (First-Order)

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

$\Rightarrow$

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)

$\Rightarrow$

InterProcAssign(p, a)

InterProcAssign(b, p)

VarPointsTo(a, l_1)

VarPointsTo(p, l_1)

VarPointsTo(b, l_1)

Input and Output Relations

- Input relations (program representation)

Alloc($var : V, heap : H, inMeth : M$)
Move($to : V, from : V$)
Load($to : V, base : V, fld : F$)
Store($base : V, fld : F, from : V$)
CallGraph($invo : I, meth : M$)
Reachable($meth : M$)
FormalArg($meth : M, i : \mathbb{N}, arg : V$)
ActualArg($invo : I, i : \mathbb{N}, arg : V$)
FormalReturn($meth : M, ret : V$)
ActualReturn($invo : I, var : V$)

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

VarPointsTo($var : V, heap : H$)
FldPointsTo($baseH : H, fld : F, heap : H$)
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)

$\xrightarrow{(1), (5), (6)}$

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)

$\xrightarrow{(7)}{}^{*}$

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
```

➡

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)

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
```

→

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)

→

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

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

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

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

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

$\text{VCall}(base : V, sig : S, invo : I, inMeth : M)$

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

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

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

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

$\text{ThisVar}(meth : M, this : V)$

$\text{HeapType}(heap : H, type : T)$

$\text{LookUp}(type : T, sig : S, meth : M)$

- Output relations

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

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

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

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

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

Analysis Rules

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

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

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

(3) $\text{FldPointsTo}(baseH, fld, heap) \leftarrow$

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

(4) $\text{VarPointsTo}(to, heap) \leftarrow$

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

(5) $\text{InterProcAssign}(to, from) \leftarrow$

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

(6) $\text{InterProcAssign}(to, from) \leftarrow$

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

(7) $\text{VarPointsTo}(to, heap) \leftarrow$

$\quad \text{InterProcAssign}(to, from), \text{VarPointsTo}(from, heap)$

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($toMeth, this$)

Analysis Rules

(8) Reachable($toMeth$),

$\quad$ VarPointsTo($this$, $heap$),

$\quad$ CallGraph($invo$, $toMeth$) $\leftarrow$

$\quad\quad$ VCall($base$, sig, $invo$, $inMeth$), Reachable($inMeth$),

$\quad\quad$ VarPointsTo($base$, $heap$),

$\quad\quad$ HeapType($heap$, $heapT$), LookUp($heapT$, sig, $toMeth$),

$\quad\quad$ 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.

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

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

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

$\xrightarrow{(7)}$ $\mathsf{VarPointsTo}(d, l_2)$, $\mathsf{VarPointsTo}(d, l_3)$, $\mathsf{VarPointsTo}(e, l_2)$, $\mathsf{VarPointsTo}(e, l_3)$

```
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
```

Summary

- Static analysis examples
  - Numerical analysis: Sign, Interval, Octagon domains
  - Pointer analysis
- Concepts covered
  - Abstract domain and semantics
  - Fixed point computation, acceleration, refinement