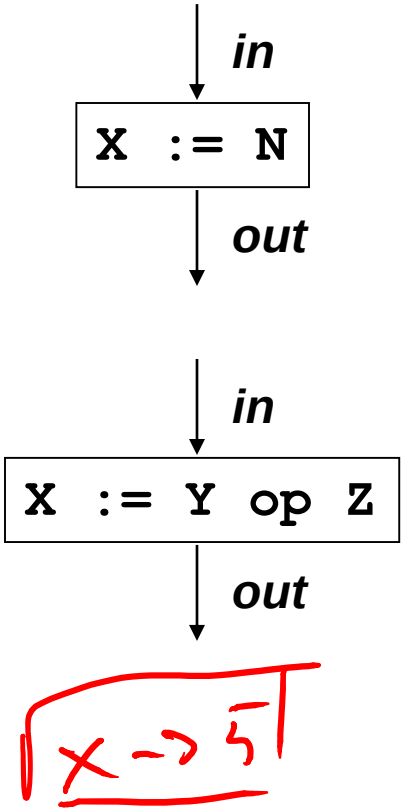
Another example: constant prop

- Set D = $\mathcal{P}(\{x \to c \mid x \in Var, c \in Const\})$

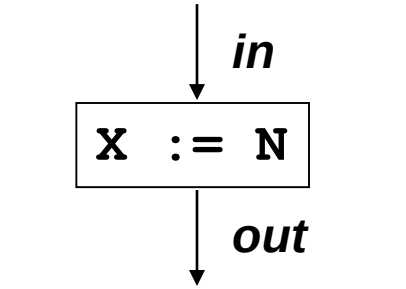


$F_{X := N}(in) = in - \{x \to *\} \cup \{x \to N\}$

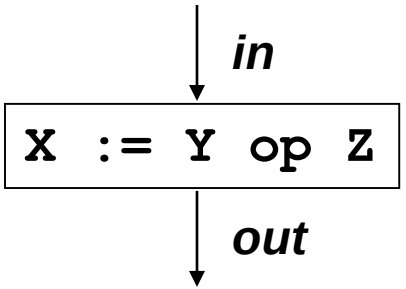
$F_{X := Y \text{ op } Z}(in) = in - \{x \to *\} \cup \{x \to N_1 \text{ op } N_2 \mid y \to N_1 \in in \wedge z \to N_2 \in in\}$

Another example: constant prop

- Set D = $2^{\{ x \to N \mid x \in Vars \wedge N \in Z \}}$

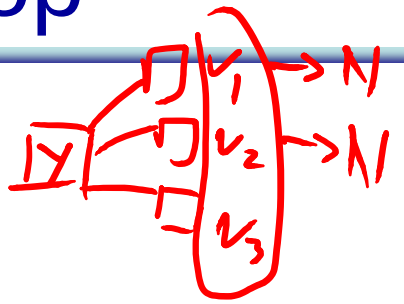


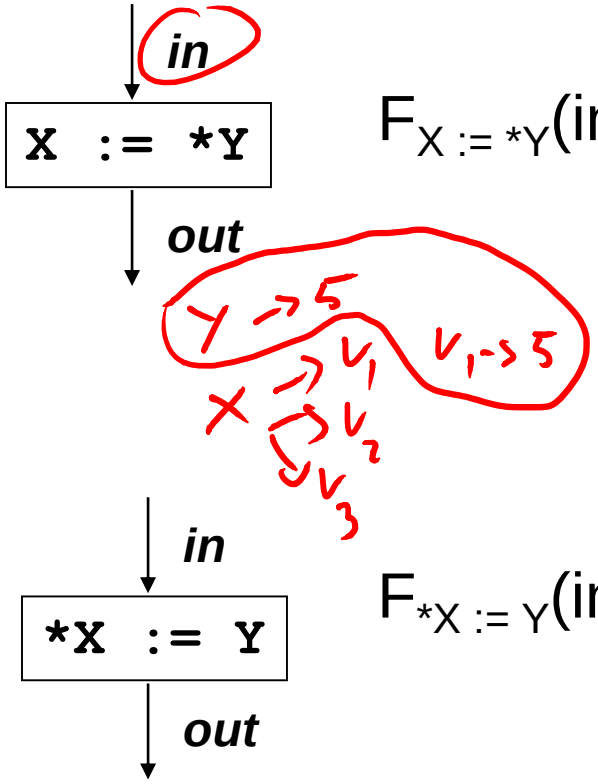
$$F_{X := N}(in) = in - \{ X \to * \} \cup \{ X \to N \}$$



$$F_{X := Y \text{ op } Z}(in) = in - \{ X \to * \} \cup \{ X \to N \mid (Y \to N_1) \in in \wedge (Z \to N_2) \in in \wedge N = N_1 \text{ op } N_2 \}$$

Another example: constant prop





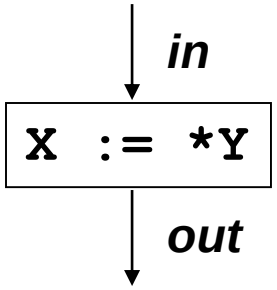
$F_{X := *Y}(in) =$ in − {x → *} ∪ {x → N | ∀ v ∈ mayPt(y) y → N ∈ in}

$F_{*X := Y}(in) =$ in − {w → * | w ∈ mayPt(x)} ∪ {v → N | v → N ∈ in ∧ y → N ∈ in}

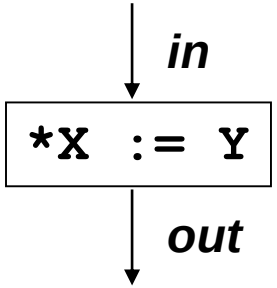
x = &v1 x = &v2

*x = ...

Another example: constant prop

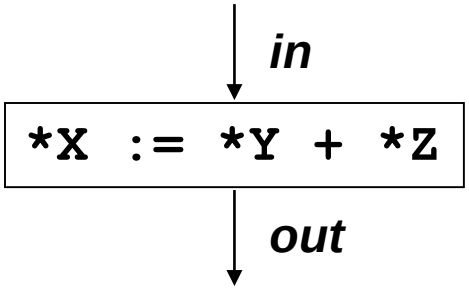


$$F_{X := *Y}(in) = in - \{ X \rightarrow * \} \cup \{ X \rightarrow N \mid \forall Z \in \text{may-point-to}(Y) . (Z \rightarrow N) \in in \}$$

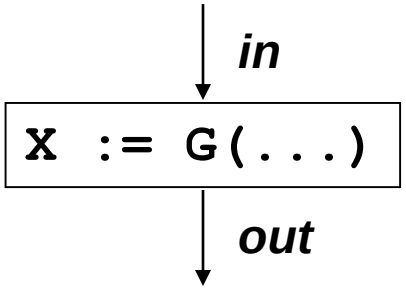


$$F_{*X := Y}(in) = in - \{ Z \rightarrow * \mid Z \in \text{may-point}(X) \} \cup \{ Z \rightarrow N \mid Z \in \text{must-point-to}(X) \wedge Y \rightarrow N \in in \} \cup \{ Z \rightarrow N \mid (Y \rightarrow N) \in in \wedge (Z \rightarrow N) \in in \}$$

Another example: constant prop

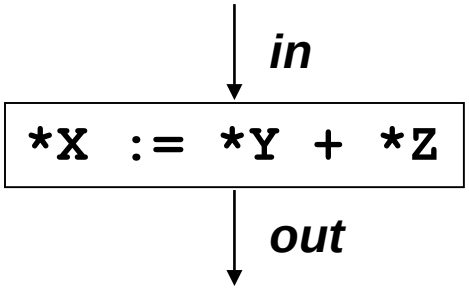


$F_{*X := *Y + *Z}(in) =$

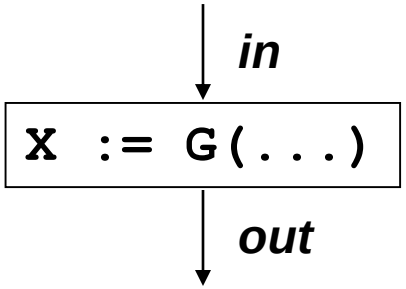


$F_{X := G(...)}(in) = \emptyset$

Another example: constant prop

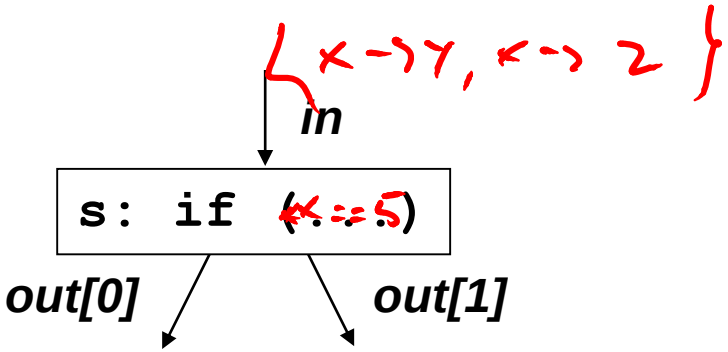


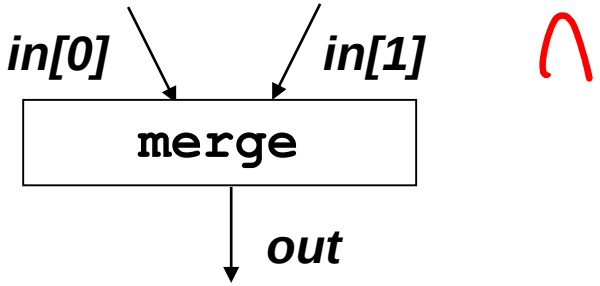
$F_{*X := *Y + *Z}(in) = F_{a := *Y;b := *Z;c := a + b; *X := c}(in)$



$F_{X := G(...)}(in) = \emptyset$

Another example: constant prop





∩

Lattice

- $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) =$

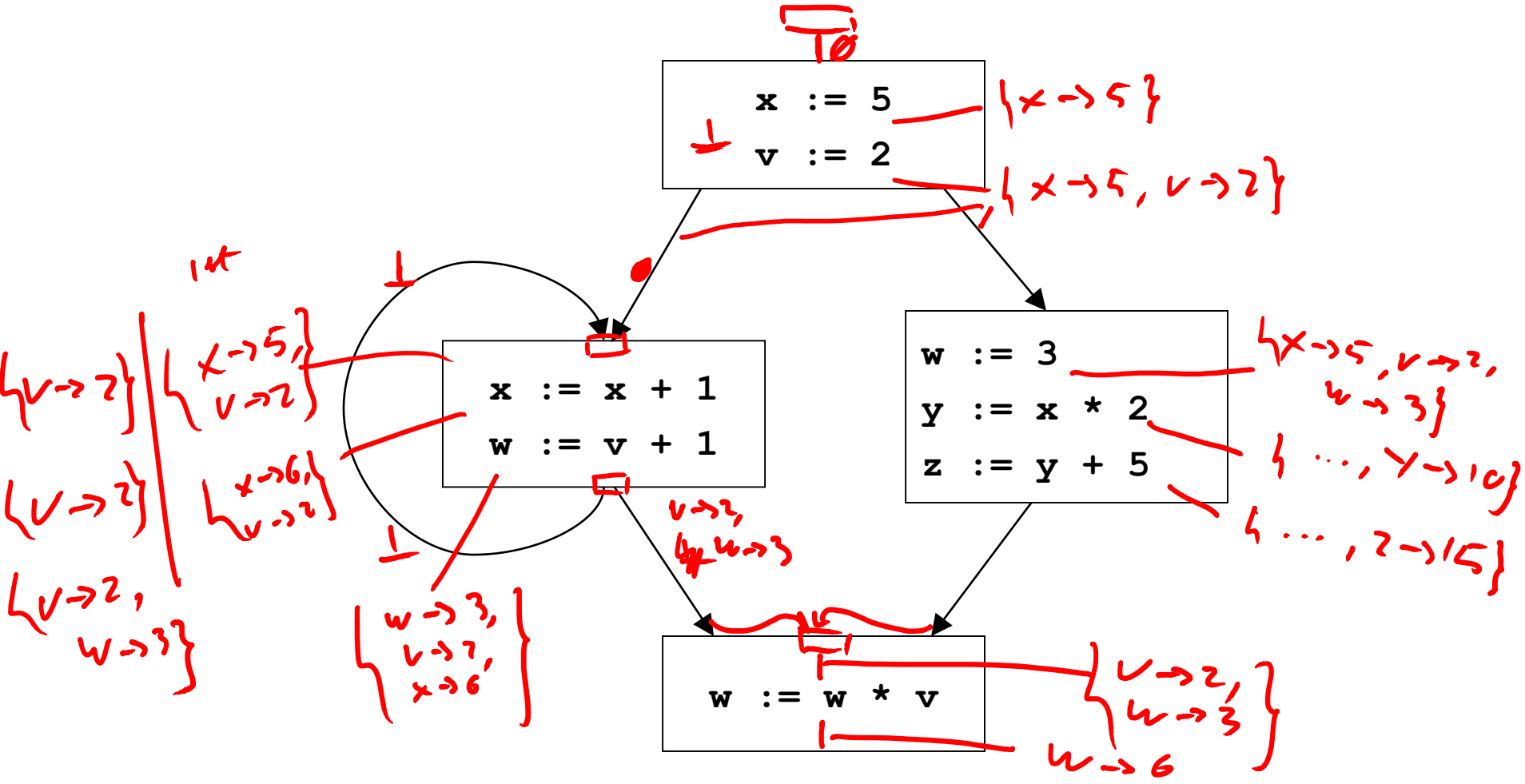
Lattice

- $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) =$

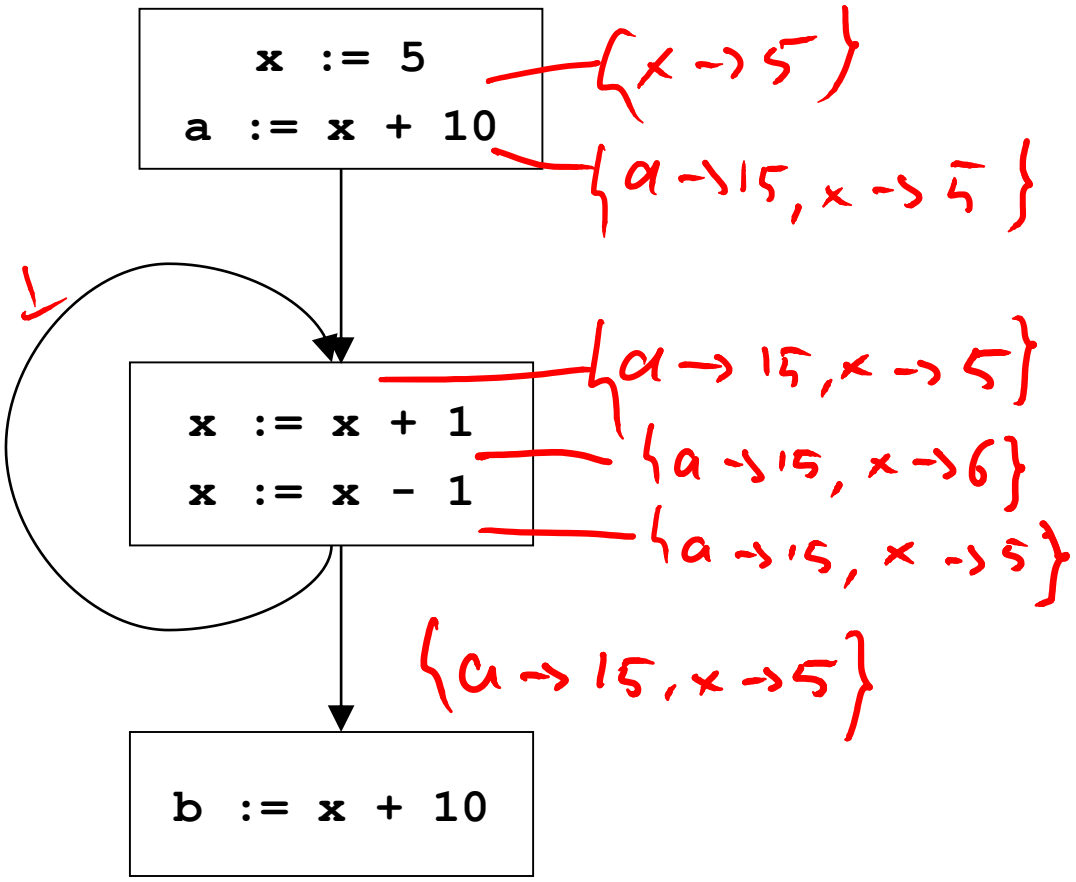
  $(2^A, \supseteq, A, \emptyset, \cap, \cup)$

  where $A = \{ x \rightarrow N \mid x \in Vars \wedge N \in Z \}$

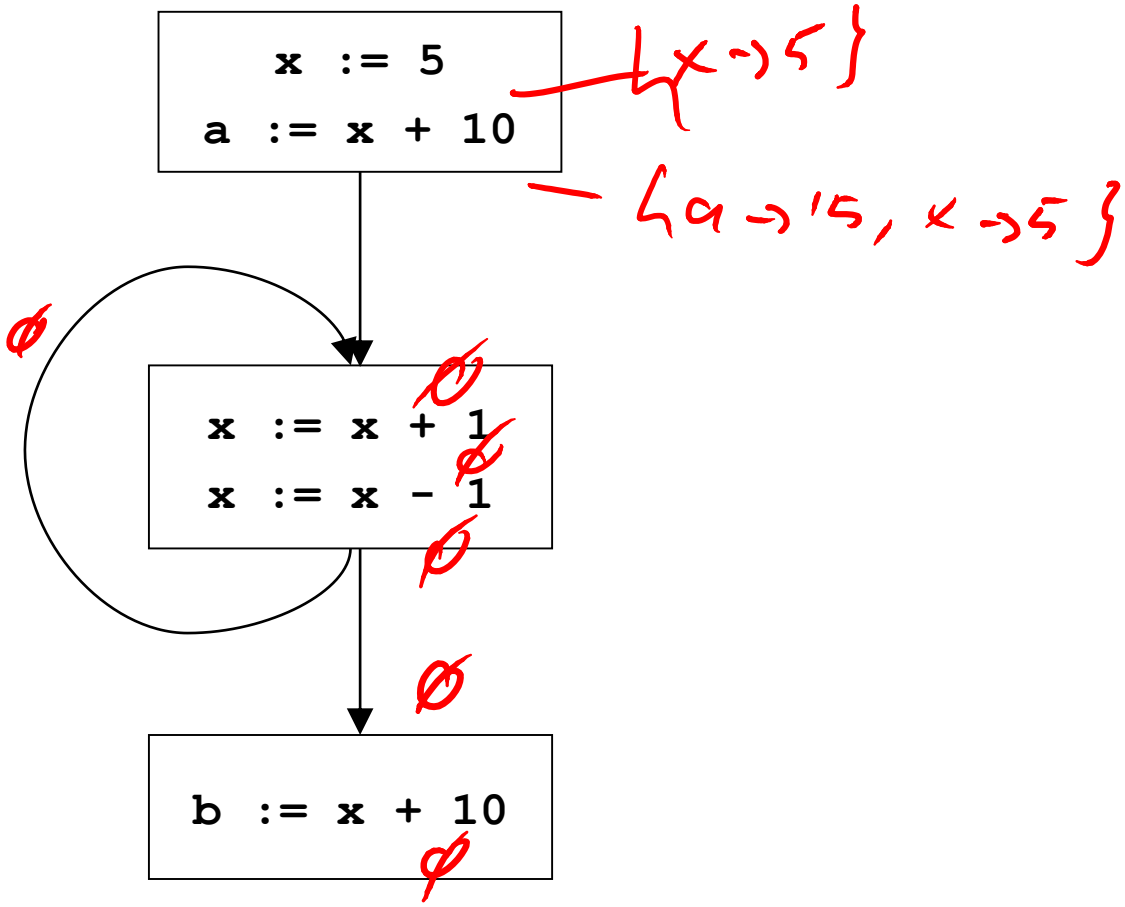
Example



Another Example



Another Example starting at top



Back to lattice

- $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) =$

  $(2^A, \supseteq, A, \emptyset, \cap, \cup)$

  where $A = \{ x \rightarrow N \mid x \in Vars \wedge N \in Z \}$

- What's the problem with this lattice?

Back to lattice

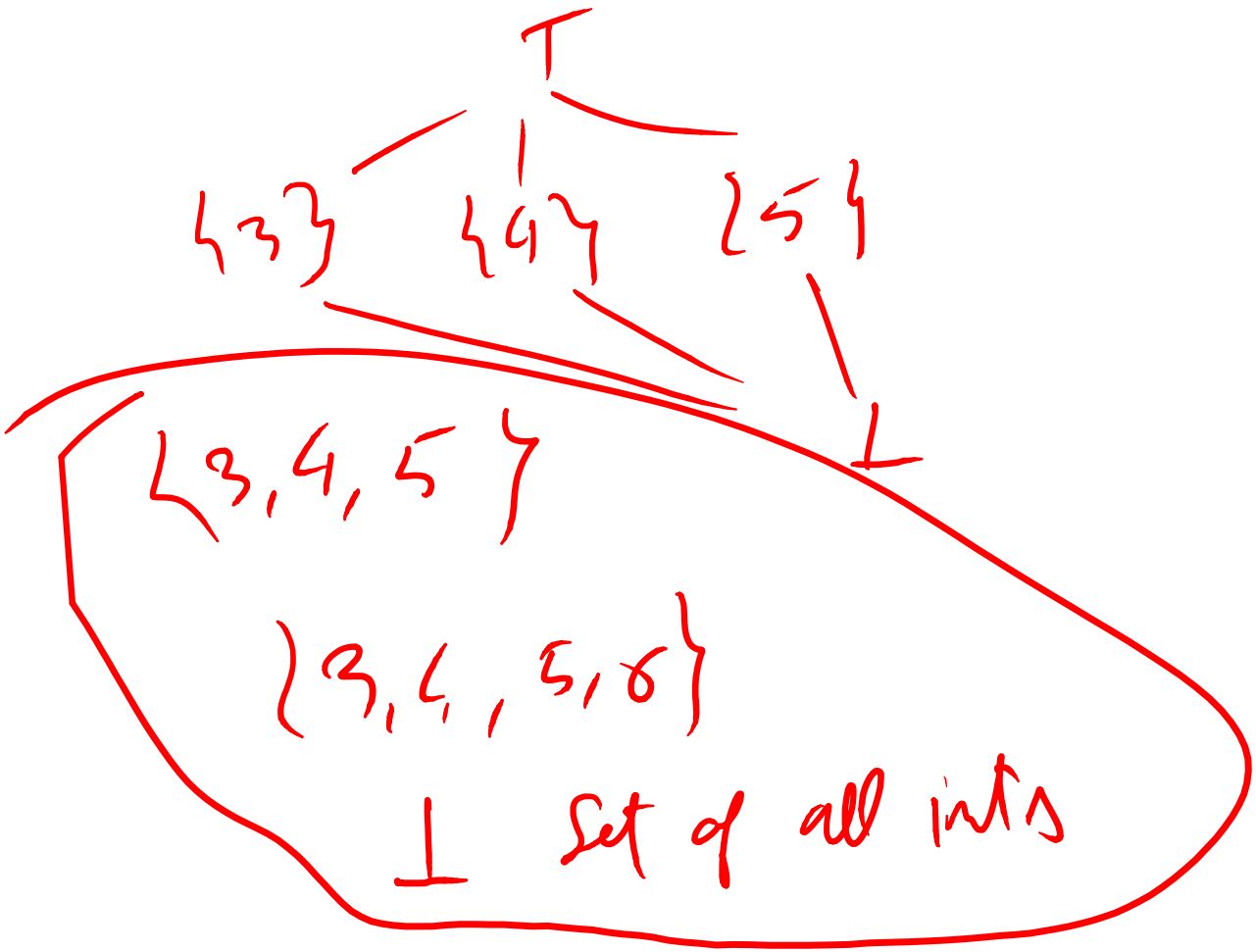
- $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) =$
  $(2^A, \supseteq, A, \emptyset, \cap, \cup)$
  where $A = \{ x \rightarrow N \mid x \in Vars \wedge N \in Z \}$

- What's the problem with this lattice?

- Lattice is infinitely high, which means we can't guarantee termination

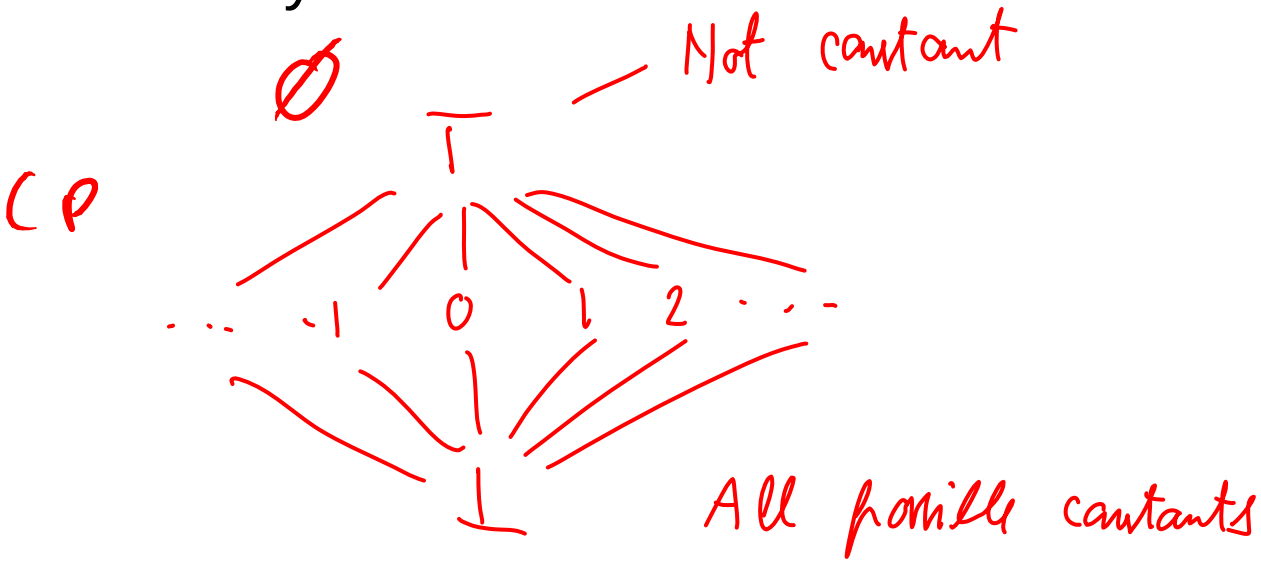
Better lattice

- Suppose we only had one variable



Better lattice

- Suppose we only had one variable



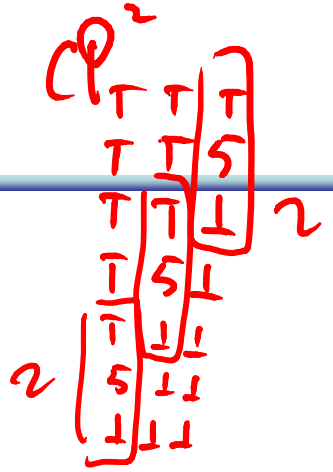
- $D = \{\bot, \top\} \cup Z$
- $\forall i \in Z . \bot \sqsubseteq i \wedge i \sqsubseteq \top$
- height = 3

For all variables

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices $(D_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ ... $(D_n, \sqsubseteq_n, \bot_n, \top_n, \sqcup_n, \sqcap_n)$ create:

tuple lattice D^n = $D_1 \times D_2 \cdots \times D_n \ni (d_1, d_2, d_3 \ldots d_n)$

For all variables



- Two possibilities
- Option 1: Tuple of lattices
- Given lattices $(D_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ ... $(D_n, \sqsubseteq_n, \bot_n, \top_n, \sqcup_n, \sqcap_n)$ create:

tuple lattice $D^n = ((D_1 \times ... \times D_n), \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where

$\bot = (\bot_1, ..., \bot_n)$

$\top = (\top_1, ..., \top_n)$

$(a_1, ..., a_n) \sqcup (b_1, ..., b_n) = (a_1 \sqcup_1 b_1, ..., a_n \sqcup_n b_n)$

$(a_1, ..., a_n) \sqcap (b_1, ..., b_n) = (a_1 \sqcap_1 b_1, ..., a_n \sqcap_n b_n)$

height = height(D_1) + ... + height(D_n)

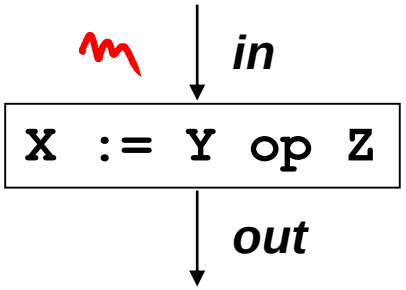
$(a,b) \sqsubseteq (c,d) \iff a \sqsubseteq c \wedge b \sqsubseteq d$

For all variables

- Option 2: Map from variables to single lattice
- Given lattice $(D, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and a set V, create:

map lattice $V \to D = (V \to D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$

Back to example



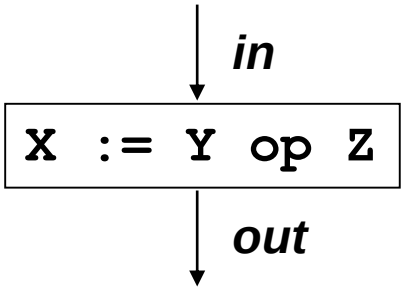
$F_{X := Y \text{ op } Z}(in) =$

$in[x \mapsto in(1) \hat{op} in(2)]$

$m[x \to m(1) \text{ op } m(2)]$

$\hat{op}$	$\perp$	c_1	$\top$
$\perp$	$\perp$	$\top$	$\top$
c_2	$\top$	$c_1 op c_2$	$\top$
$\top$	$\top$	$\top$	$\top$

Back to example



$F_{X := Y \text{ op } Z}(in) = in [X \rightarrow in(Y) \widehat{op} in(Z)]$

where $a \widehat{op} b =$

$\widehat{op}$	$\perp$	d_1	$\top$
$\perp$	$\perp$	$\top$	$\top$
d_2	$\top$	$d_1 \widehat{op} d_2$	$\top$
$\top$	$\top$	$\top$	$\top$

General approach to domain design

- Simple lattices:
  - boolean logic lattice
  - powerset lattice
  - incomparable set: set of incomparable values, plus top and bottom (eg const prop lattice)
  - two point lattice: just top and bottom
- Use combinators to create more complicated lattices
  - tuple lattice constructor
  - map lattice constructor

May vs Must

- Has to do with definition of computed info

- Set of x $\rightarrow$ y must-point-to pairs
  - if we compute x $\rightarrow$ y, then, then during program execution, x must point to y
- Set of x$\rightarrow$ y may-point-to pairs
  - if during program execution, it is possible for x to point to y, then we must compute x $\rightarrow$ y

May vs must

	May	Must
most optimistic (bottom)	$\emptyset$	FS
most conservative (top)	FS	$\emptyset$
safe	Add	Remove
merge	$\cup$	$\cap$

May vs must

	May	Must
most optimistic (bottom)	empty set	full set
most conservative (top)	full set	empty set
safe	overly big	overly small
merge	$\cup$	$\cap$

Common Sub-expression Elim

- Want to compute when an expression is available in a var
- Domain:

Common Sub-expression Elim

- Want to compute when an expression is available in a var
- Domain:

$$S = \{ x \to E \mid x \in Var, E \in Expr \}$$

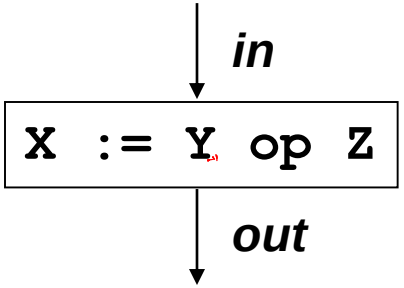
$$D = 2^S \quad \mathcal{P}(S)$$

$$\top = S$$

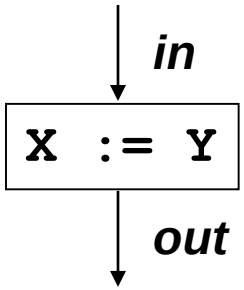
$$\bot = \emptyset$$

$$\sqcup = \cap$$

Flow functions

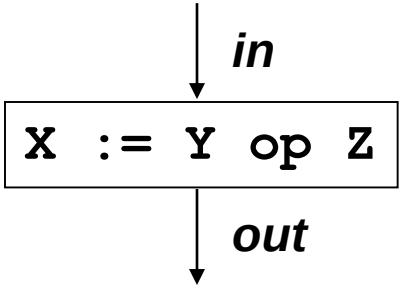


$F_{X := Y \text{ op } Z}(in) =$

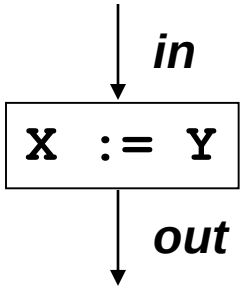


$F_{X := Y}(in) =$

Flow functions

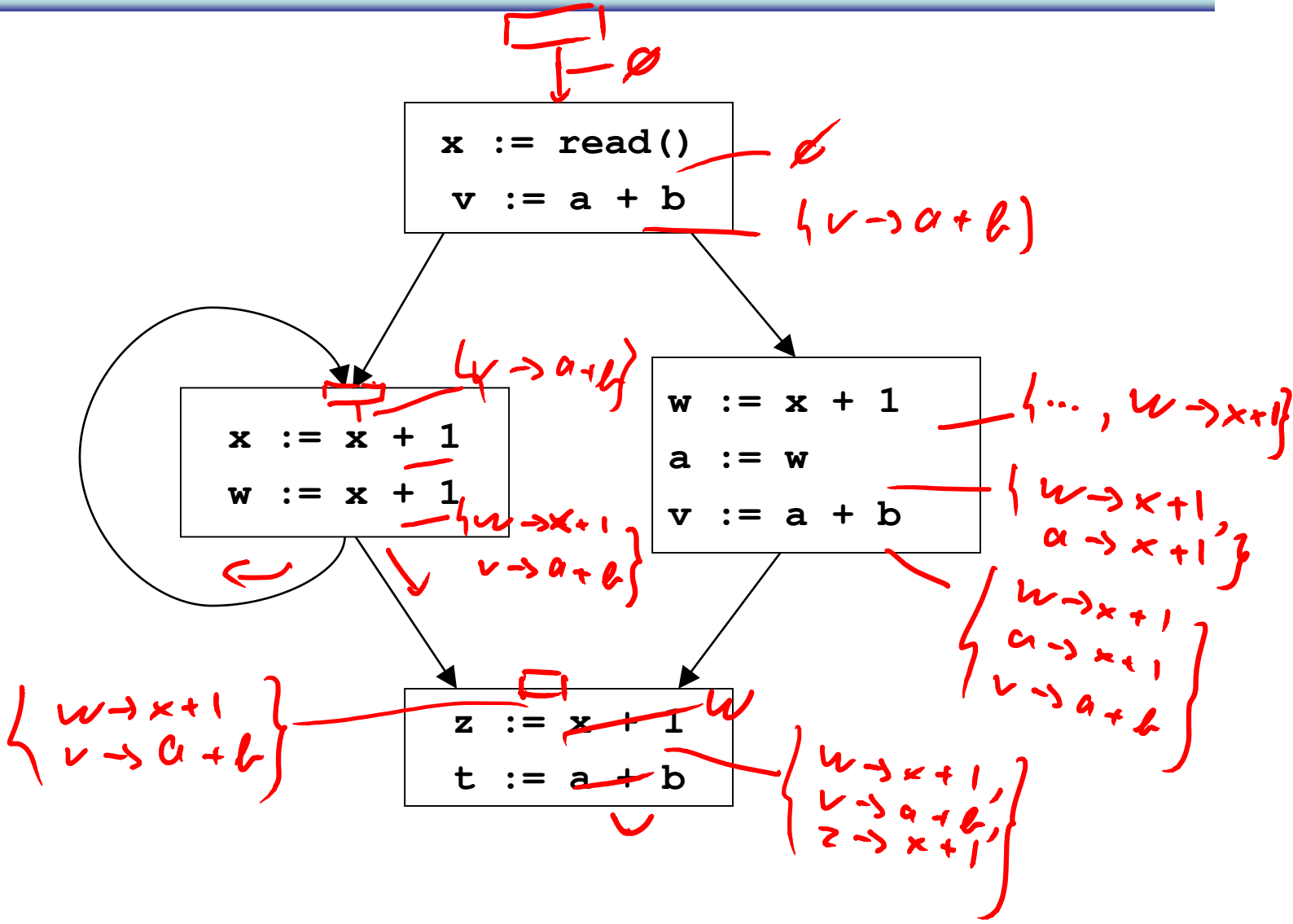


$$F_{X := Y \text{ op } Z}(in) = in - \{ X \rightarrow * \} - \{ * \rightarrow \ldots X \ldots \} \cup \{ X \rightarrow Y \text{ op } Z \mid X \neq Y \wedge X \neq Z \}$$



$$F_{X := Y}(in) = in - \{ X \rightarrow * \} - \{ * \rightarrow \ldots X \ldots \} \cup \{ X \rightarrow E \mid Y \rightarrow E \in in \}$$

Example



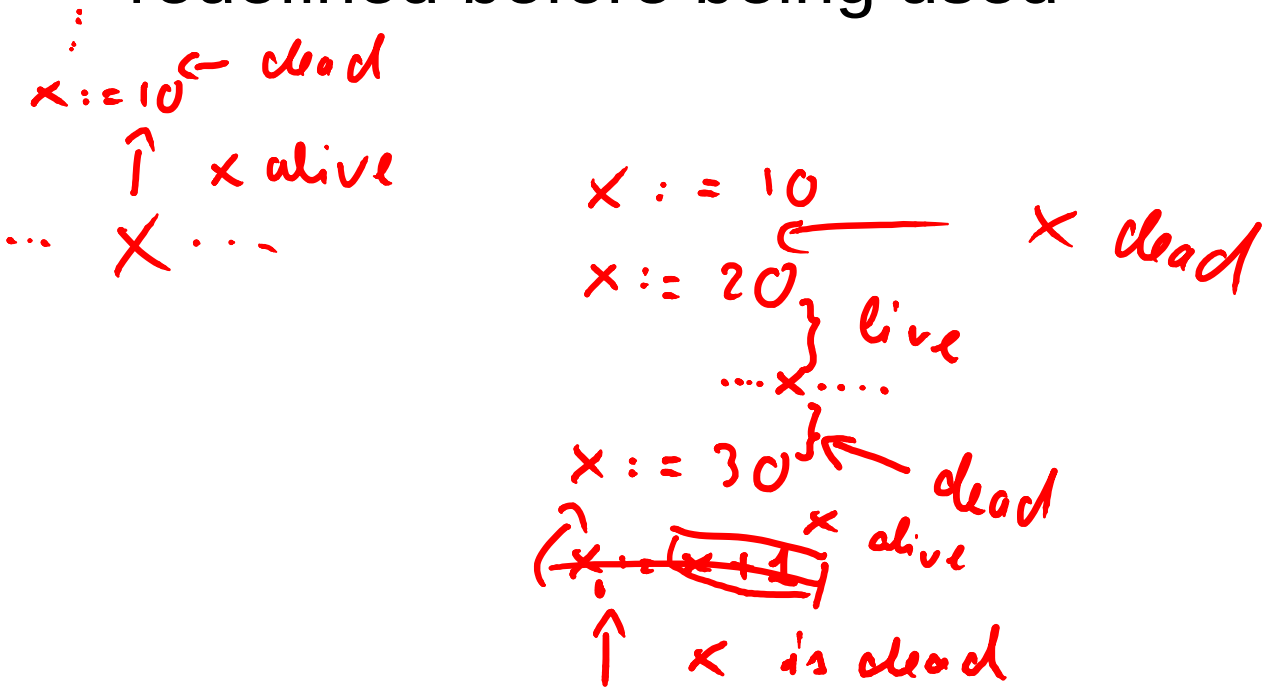
Direction of analysis

- Although constraints are not directional, flow functions are
- All flow functions we have seen so far are in the forward direction
- In some cases, the constraints are of the form

  in = F(out)
- These are called backward problems.
- Example: live variables
  - compute the set of variables that may be live

Live Variables

- A variable is live at a program point if it will be used before being redefined
- A variable is dead at a program point if it is redefined before being used

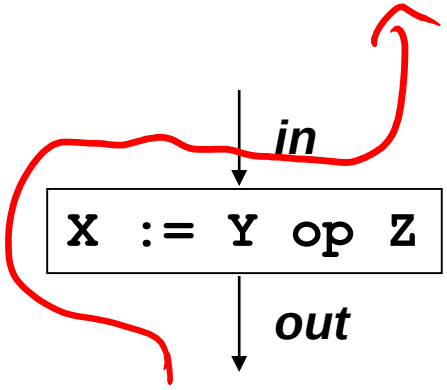


Example: live variables

- Set D = $\mathcal{P}(V)$
- Lattice: (D, $\sqsubseteq$, $\bot$, $\top$, $\sqcup$, $\sqcap$) = $\subseteq$ $\varnothing$ V $\cup$ $\cap$

Example: live variables

- Set D = 2^{Vars}
- Lattice: (D, $\sqsubseteq$, $\bot$, $\top$, $\sqcup$, $\sqcap$) = (2^{Vars}, $\subseteq$, $\emptyset$,Vars, $\cup$, $\cap$)

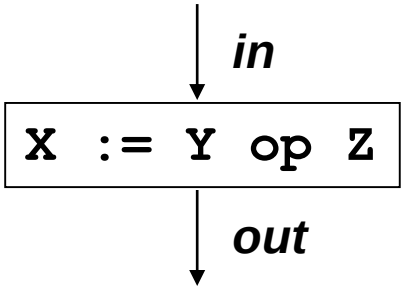


$F_{X := Y \text{ op } Z}(\text{out}) =$ out $- \{x\} \cup \{y, z\}$

$a := a + a$

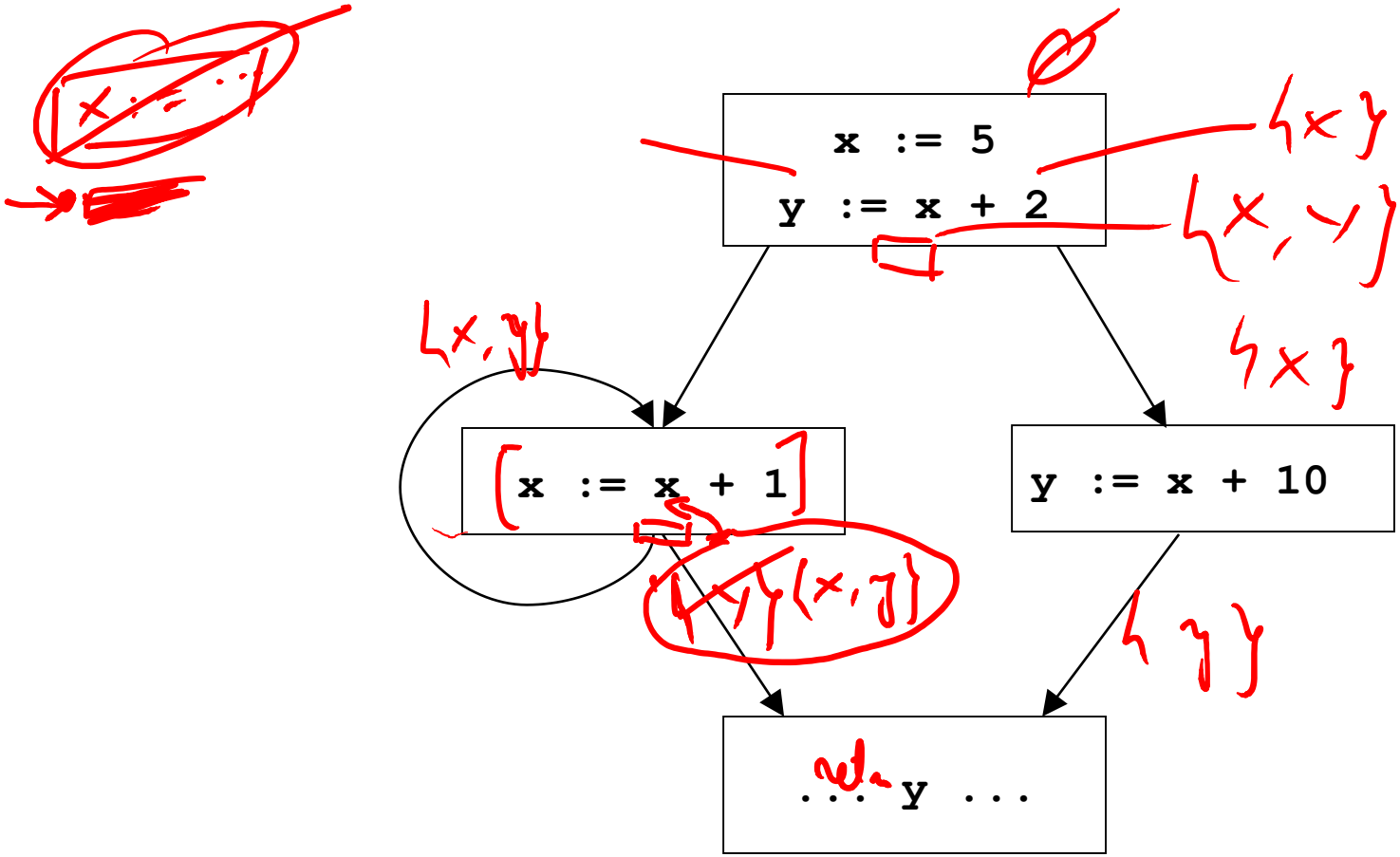
Example: live variables

- Set D = 2^{Vars}
- Lattice: (D, $\sqsubseteq$, $\bot$, $\top$, $\sqcup$, $\sqcap$) = (2^{Vars}, $\subseteq$, $\emptyset$,Vars, $\cup$, $\cap$)

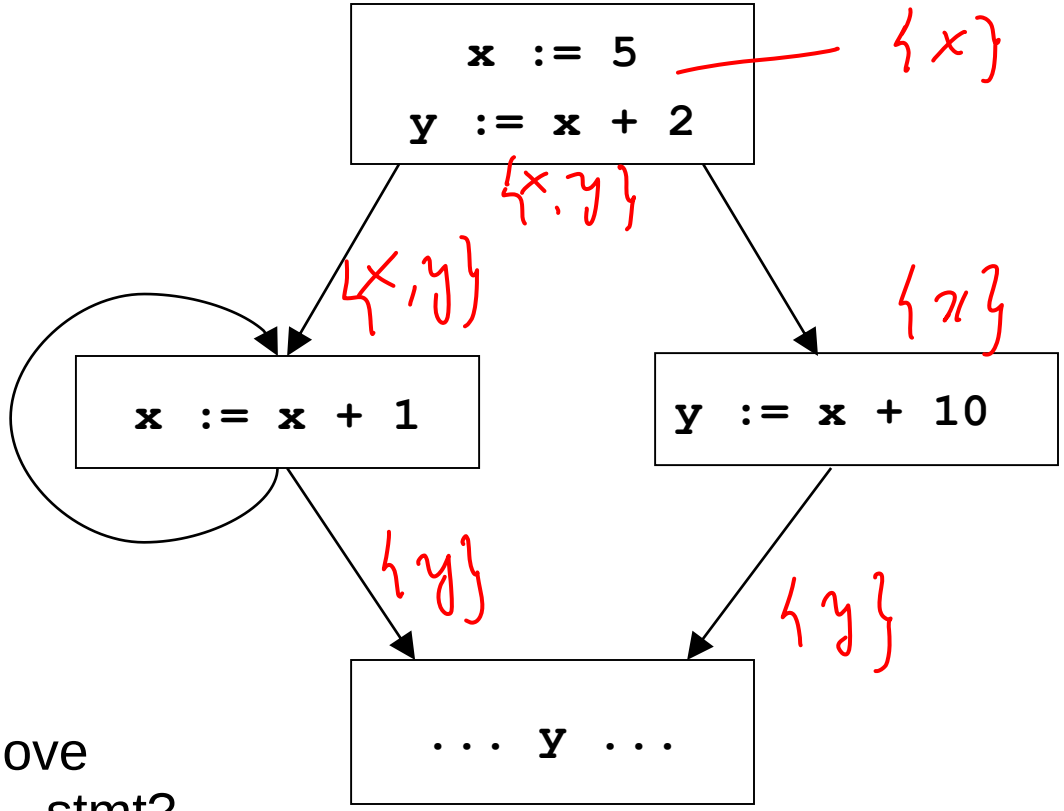


$F_{X := Y \text{ op } Z}(out) = out - \{ X \} \cup \{ Y, Z \}$

Example: live variables

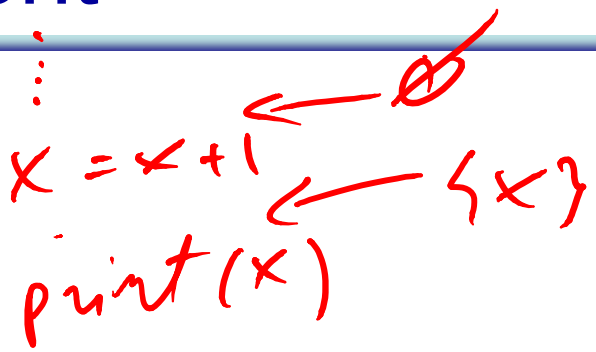


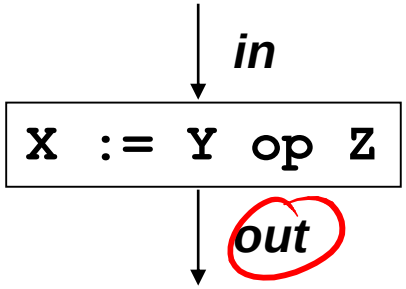
Example: live variables



How can we remove the `x := x + 1` stmt?

Revisiting assignment

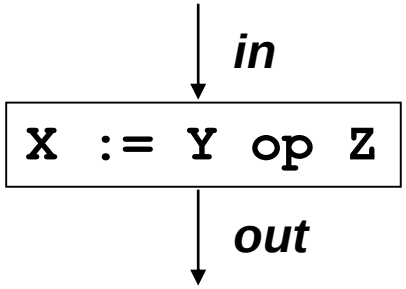




$F_{X := Y \text{ op } Z}(out) = out - \{ X \} \cup \{ Y, Z \}$

x = 15
→ x is dead

Revisiting assignment



$F_{X := Y \text{ op } Z}(out) = out - \{ X \} \cup \{ Y, Z \}$

out - {x} ∪

x ∉ out ? ∅ : {y, z}

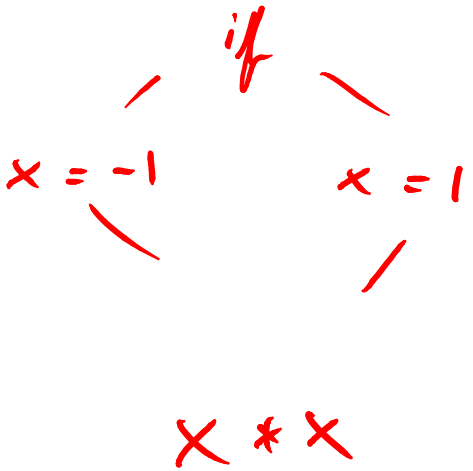
Theory of backward analyses

- Can formalize backward analyses in two ways
- Option 1: reverse flow graph, and then run forward problem
- Option 2: re-develop the theory, but in the backward direction

Precision

- Going back to constant prop, in what cases would we lose precision?

```
if (p) {
  x = 5
} else { x = 7 }
```




```
if (p) { x = x + 7 }
} else { x = x + 5 }
```

for (i = 0; i < 100; i++) {—}

Precision

- Going back to constant prop, in what cases would we lose precision?

```
x := 5
if (<expr>) {
   x := 6
}
... x ...

where <expr> is
equiv to false
```

```
if (p) {
   x := 5;
} else
   x := 4;
}
...
if (p) {
   y := x + 1
} else {
   y := x + 2
}
... y ...
```

```
if (...) {
   x := -1;
} else
   x := 1;
}
y := x * x;
... y ...
```

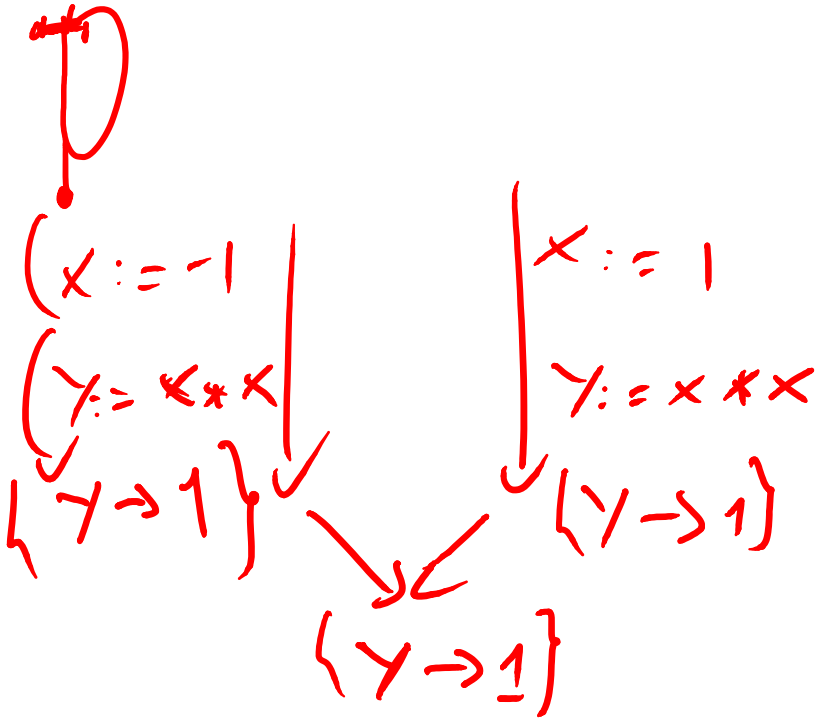
Precision

- The first problem: Unreachable code
  - solution: run unreachable code removal before
  - the unreachable code removal analysis will do its best, but may not remove all unreachable code
- The other two problems are path-sensitivity issues
  - Branch correlations: some paths are infeasible
  - Path merging: can lead to loss of precision

MOP: meet over all paths

- Information computed at a given point is the meet of the information computed by each path to the program point

```
if (...) {
    x := -1;
} else
    x := 1;
}
y := x * x;
... y ...
```



MOP

- For a path p, which is a sequence of statements $[s_1, ..., s_n]$, define: $F_p(in) = F_{s_n}(...F_{s_1}(in) ...)$
- In other words: $F_p = F_{s_1} \circ \cdots \circ F_{s_n}$
- Given an edge e, let paths-to(e) be the (possibly infinite) set of paths that lead to e
- Given an edge e, MOP(e) = $\bigsqcup_{p \in \text{paths-to}(e)} F_p(\bot)$
- For us, should be called JOP (ie: join, not meet)

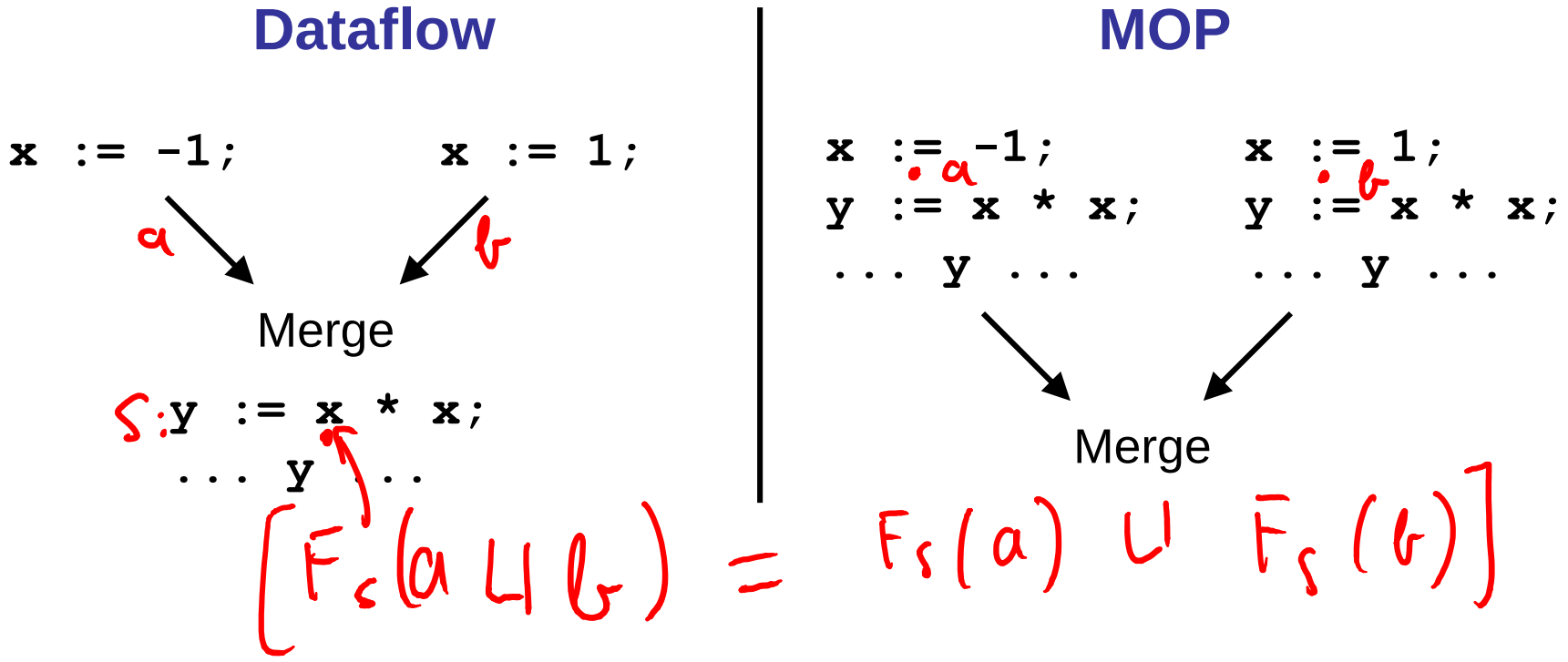
MOP vs. dataflow

- MOP is the “best” possible answer, given a fixed set of flow functions
  - This means that MOP $\sqsubseteq$ dataflow at edge in the CFG
- In general, MOP is not computable (because there can be infinitely many paths)
  - vs dataflow which is generally computable (if flow fns are monotonic and height of lattice is finite)
- And we saw in our example, in general, MOP $\neq$ dataflow

MOP vs. dataflow

$F(b \sqcup c) = F(b) \sqcup F(c)$

- However, it would be great if by imposing some restrictions on the flow functions, we could guarantee that dataflow is the same as MOP. What would this restriction be?



MOP vs. dataflow

- However, it would be great if by imposing some restrictions on the flow functions, we could guarantee that dataflow is the same as MOP. What would this restriction be?
- Distributive problems. A problem is distributive if:

$$\forall\ a, b\ .\ F(a \sqcup b) = F(a) \sqcup F(b)$$

- If flow function is distributive, then MOP = dataflow

Summary of precision

- Dataflow is the basic algorithm
- To basic dataflow, we can add path-separation
  - Get MOP, which is same as dataflow for distributive problems
  - Variety of research efforts to get closer to MOP for non-distributive problems
- To basic dataflow, we can add path-pruning
  - Get branch correlation
- To basic dataflow, can add both:
  - meet over all feasible paths