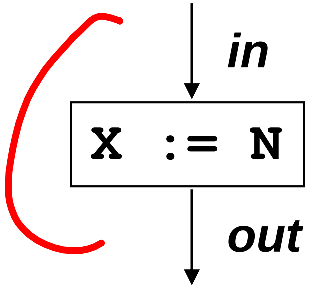
Another example: constant prop

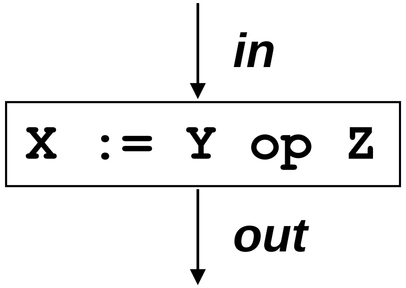
- Set D = $\mathcal{P}(\{x \to \underline{c}\})$

$\to \{a \to 2, b \to 4\}$
$a := a + b$
$\to \{a \to 6, b \to 4\}$

A. Yes
B. No
C. ??



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



$\text{in} - S_1$

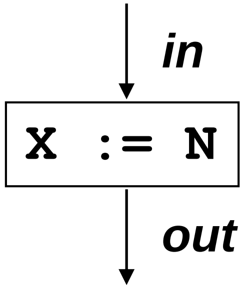
$\{a \to 2, b \to 4\}$

$F_{X := Y \text{ op } Z}(\text{in}) =$ $\text{in} \cup \{X \to c_1 \text{ op } c_2 \mid Y \to c_1 \in \text{in} \wedge Z \to c_2 \in \text{in}\}$ $- \{X \to *\}$

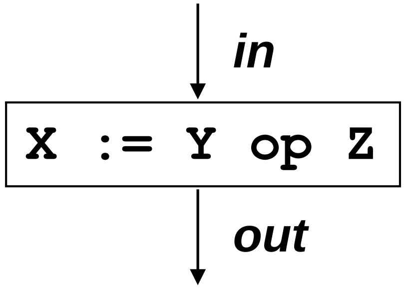
a a b

Another example: constant prop

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



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



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

Another example: constant prop

$MAYPT(Y) = \{z_1, z_2\}$

in

`X := *Y`

out

$F_{X := *Y}(in) = in - \{X \to *\} \cup \{X \to c \mid \forall z \in MAYPT(Y).\ z \to c \in in\}$

$\{Y \to c, Z \to c\}$

in

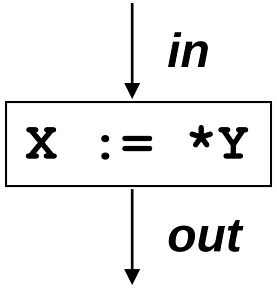
`*X := Y`

out

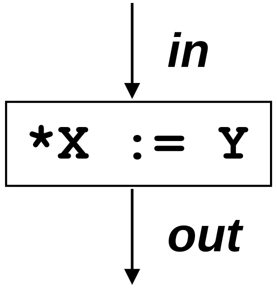
$F_{*X := Y}(in) = in - \{z \to * \mid z \in MAYPT(X)\}$

$Z \to c$

Another example: constant prop



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



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

Another example: constant prop

in

```
*X := *Y + *Z
```

out

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

in

```
X := G(...)
```

out

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

Another example: constant prop

in

```
*X := *Y + *Z
```

out

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

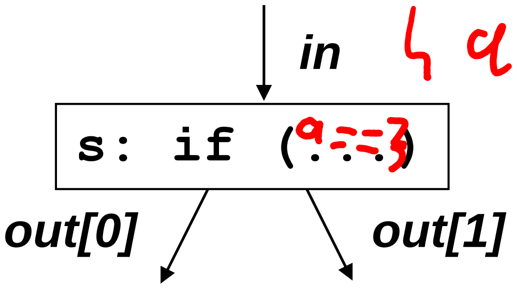
in

```
X := G(...)
```

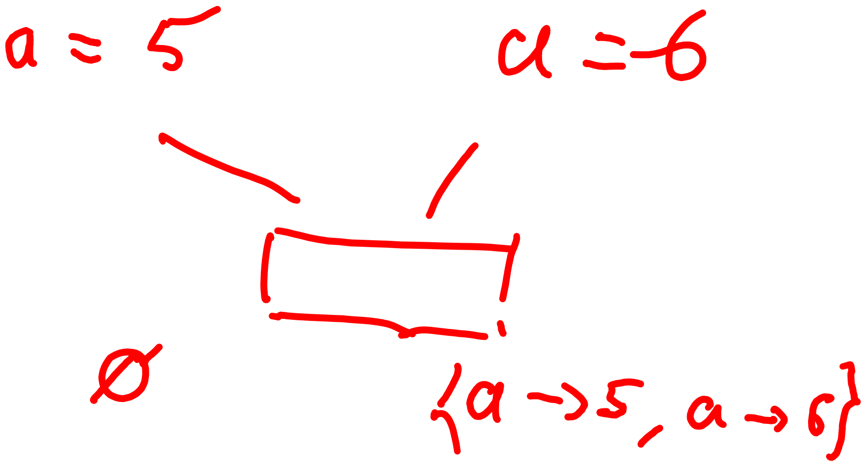
out

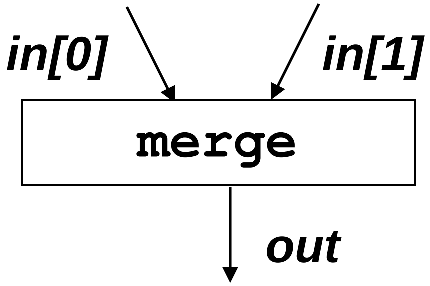
$$F_{X\ :=\ G(...)}(in) = \emptyset$$

Another example: constant prop



{a → 3}





A. out = in[0] ∩ in[1]

B. out = in[0] ∪ in[1]

Lattice

$\top = \emptyset$

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

$\mathcal{P}(\{x \to c\})$

$\emptyset \qquad FS = \{x \to c\}$

$A\ \bot = \emptyset$

$B\ \bot = FS$

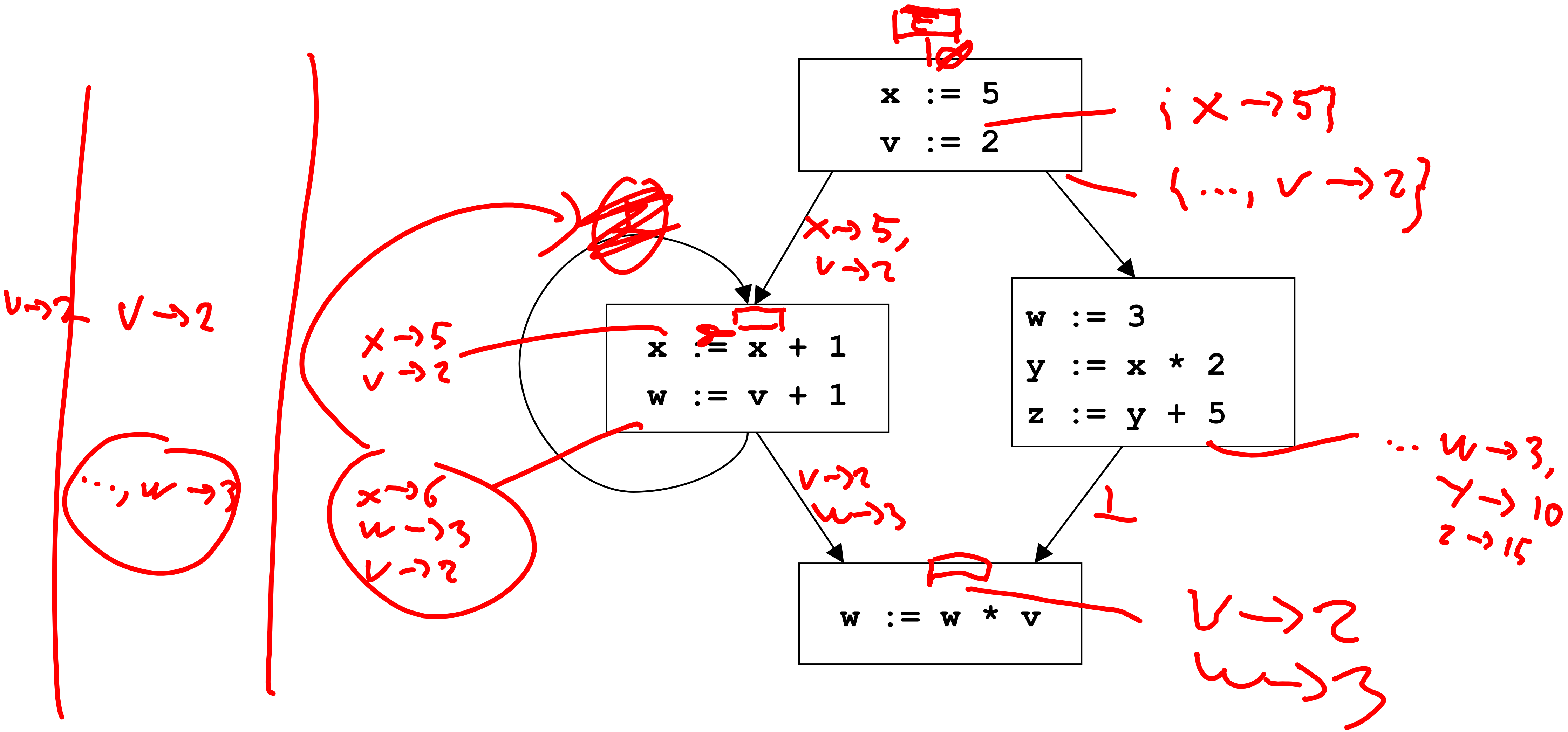
Lattice

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

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

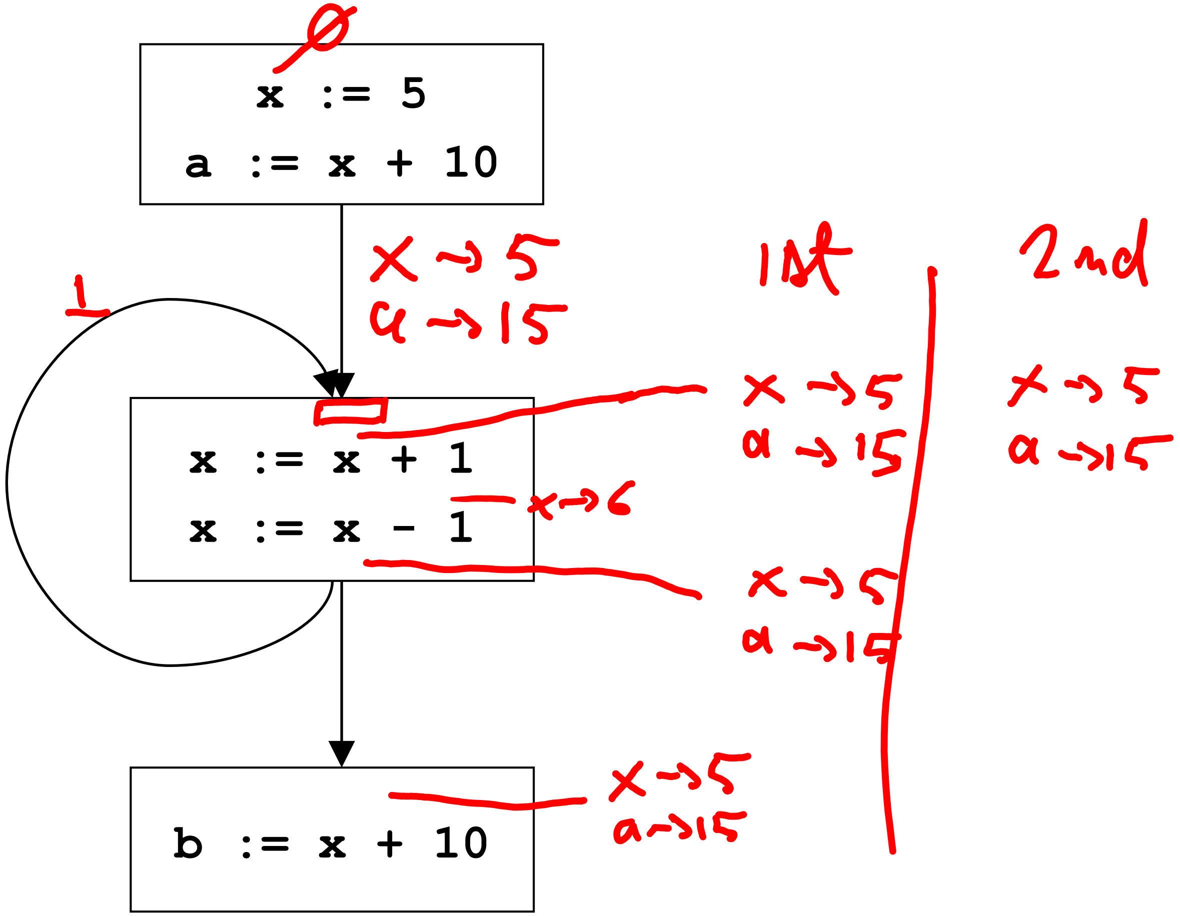
  where $A = \{ x \to N \mid x \in Vars \land N \in Z \}$

Example

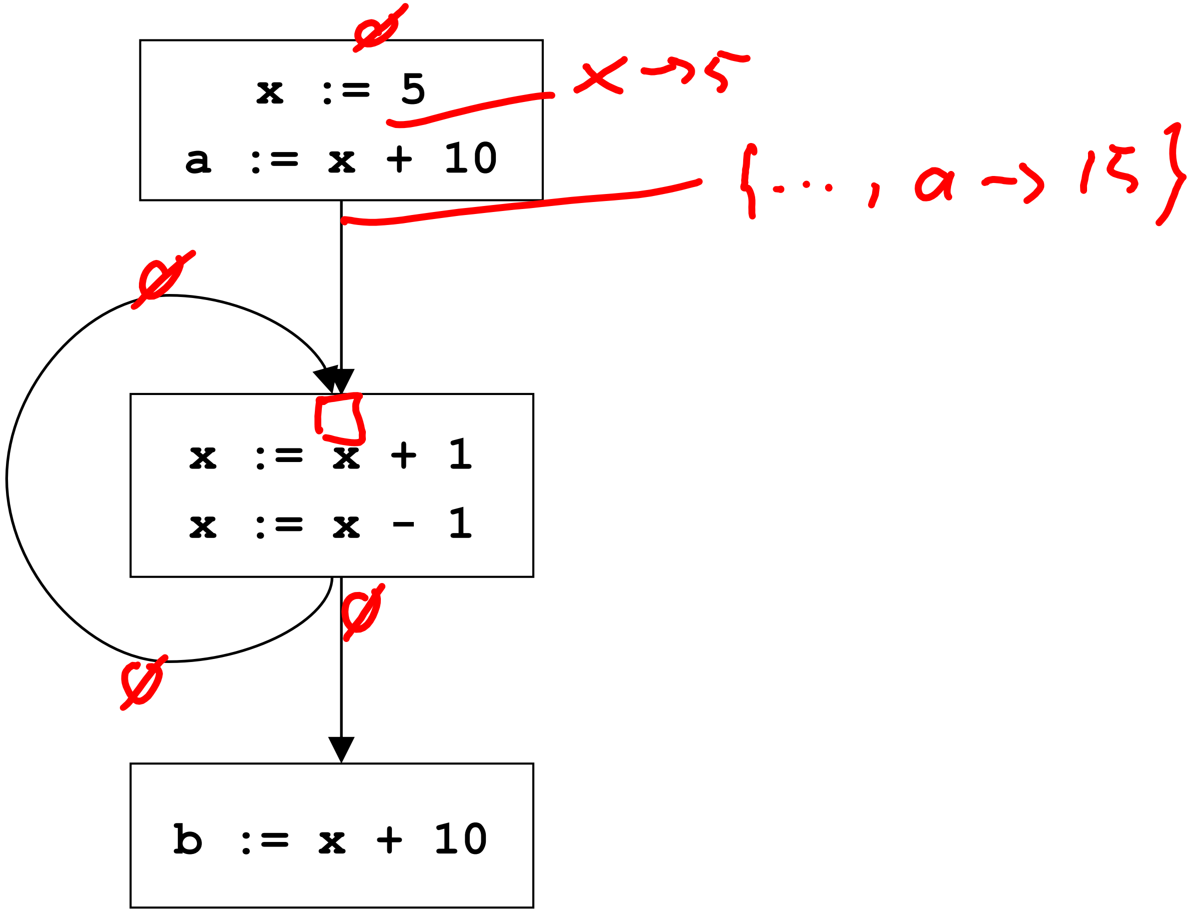


x := 5
v := 2
x := x + 1
w := v + 1
w := 3
y := x * 2
z := y + 5
w := w * v
{x → 5}
{..., v → 2}
x → 5, v → 2
v → 2
w → 3
⊥
... w → 3, y → 10, z → 15
v → 2
w → 3
x → 5
v → 2
x → 6
w → 3
v → 2
..., w → 3
v → 2

Another Example



Another Example starting at top



Back to lattice

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

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

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

- What's the problem with this lattice?

Back to lattice

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

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

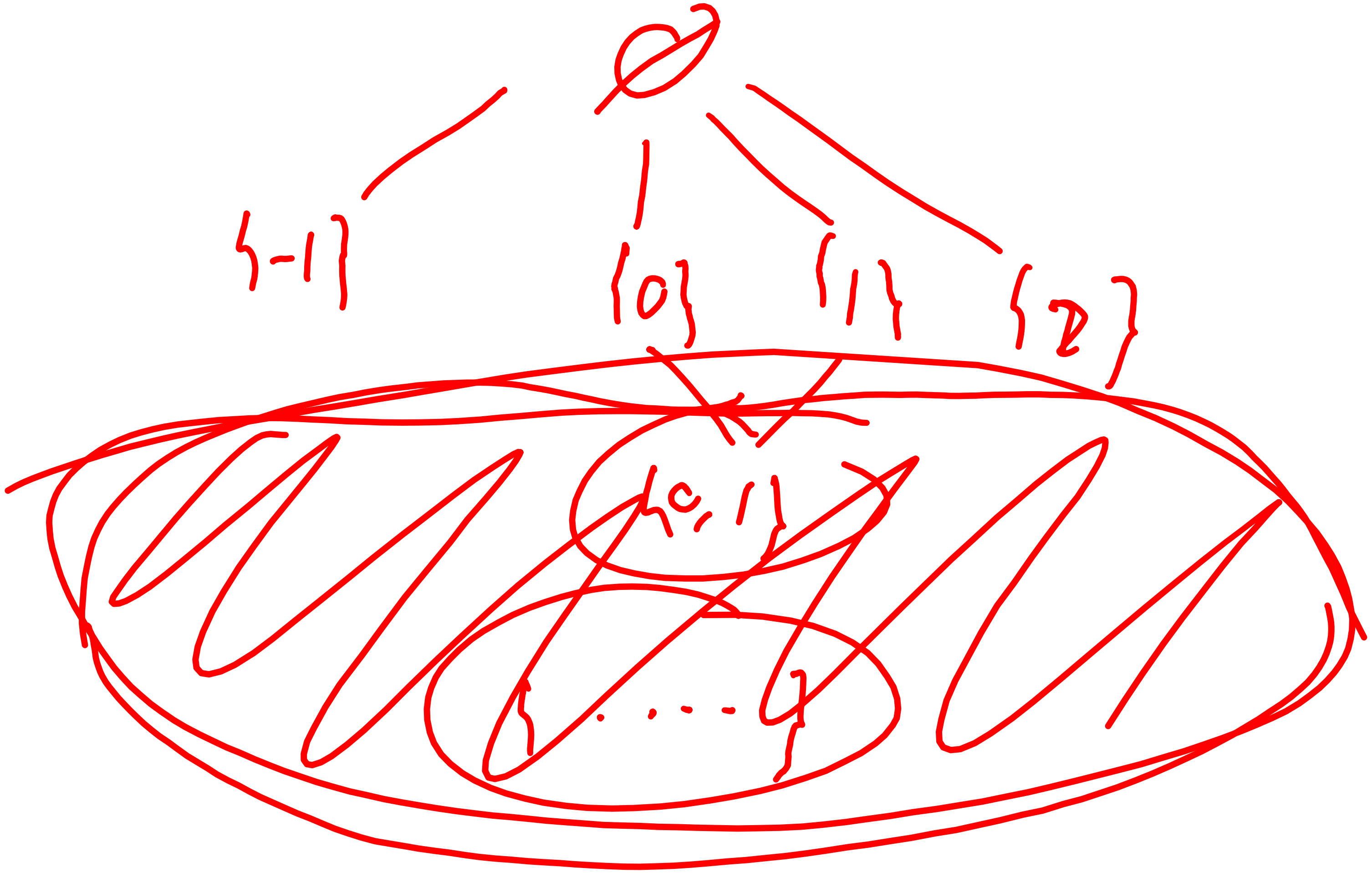
  where $A = \{ x \to N \mid x \in Vars \wedge N \in Z \}$

- What's the problem with this lattice?

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

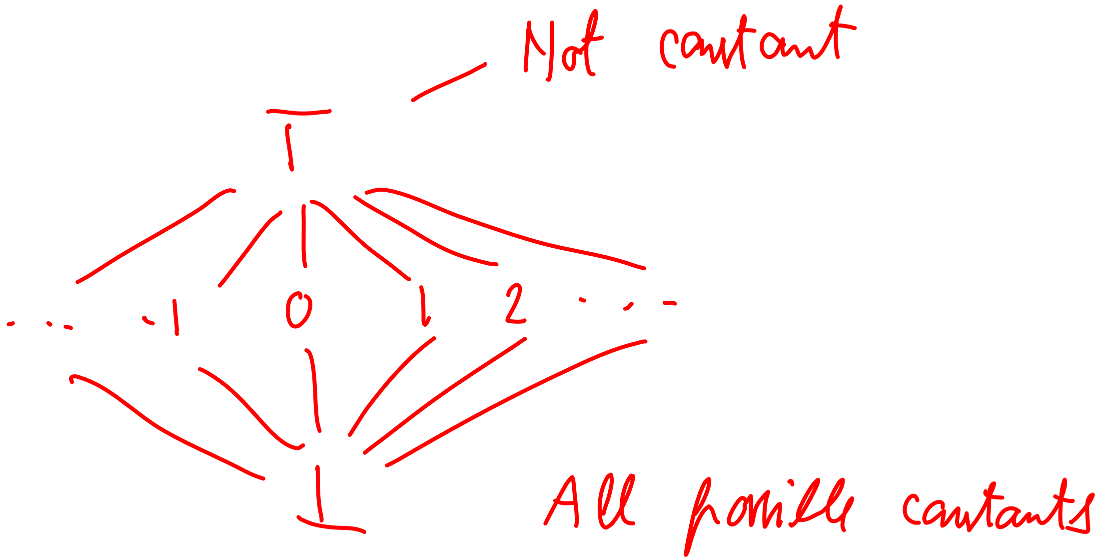
Better lattice

- Suppose we only had one variable



Better lattice

- Suppose we only had one variable



- D = {⊥, ⊤ } ∪ Z
- $\forall$ i ∈ Z . ⊥ ⊑ i ∧ i ⊑ ⊤
- 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 =

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)

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

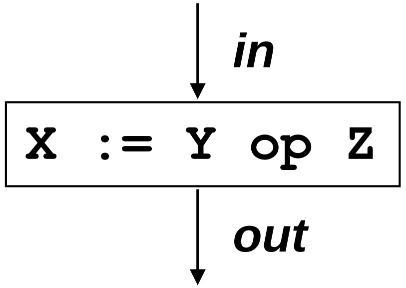
$$\bot = \lambda v \to \bot_1$$

$$\top = \lambda v \to \top_1$$

$$m_1 \sqcup m_2 = \lambda v \to m_1(v) \sqcup_1 m_2(v)$$

$$m_1 \sqsubseteq m_2 \iff \forall v.\ m_1(v) \sqsubseteq_1 m_2(v)$$

Back to example



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

Back to example

in

```
X := Y op Z
```

out

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

where a $\widehat{op}$ b =

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

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

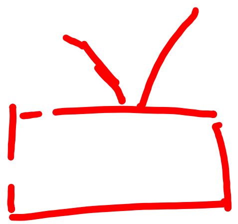
- Has to do with definition of computed info

- 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

A. $\cup$
B. $\cap$

	May	Must
most optimistic (bottom)	$\emptyset$	FS
most conservative (top)	FS	$\emptyset$
safe		
merge		



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:

$D = \mathcal{P}(\{x \to E\})$

$\top = \emptyset$

$\bot = FS$

$\sqsubseteq \; = \; \supseteq$

$\sqcup = \cap$

$\to \sqcap = \cup$

$a := b + c$

$\to \{a \to b + c\}$

$\ldots \cancel{b+c}^{\,a} \ldots$

Common Sub-expression Elim

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

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

$D = 2^S$

$\top = S$

$\bot = \emptyset$

$\sqcup = \cap$

Flow functions

a := 0
b := 1
a := a + b
z → d := a + b
a

∅ b := 1
a := a + b
{a → a + b}

a in b

```
X := Y op Z
```

out

$F_{X := Y \text{ op } Z}(\text{in}) = \text{in} - \{X \to *\} - \{* \to \ldots X \ldots\}$

$\cup \{X \to Y \text{ op } Z\}$

$X \neq Y \wedge X \neq Z$

in

```
X := Y
```

out

$F_{X := Y}(\text{in}) = \text{in} - \{X \to *\} - \{* \to \ldots X \ldots\}$

$\cup \{X \to E \mid Y \to E \in \text{in}\}$

Flow functions

$(a \to b+1),$

$b := a$

$b \to b+1$

in

```
X := Y op Z
```

out

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

$Y \to X+1$

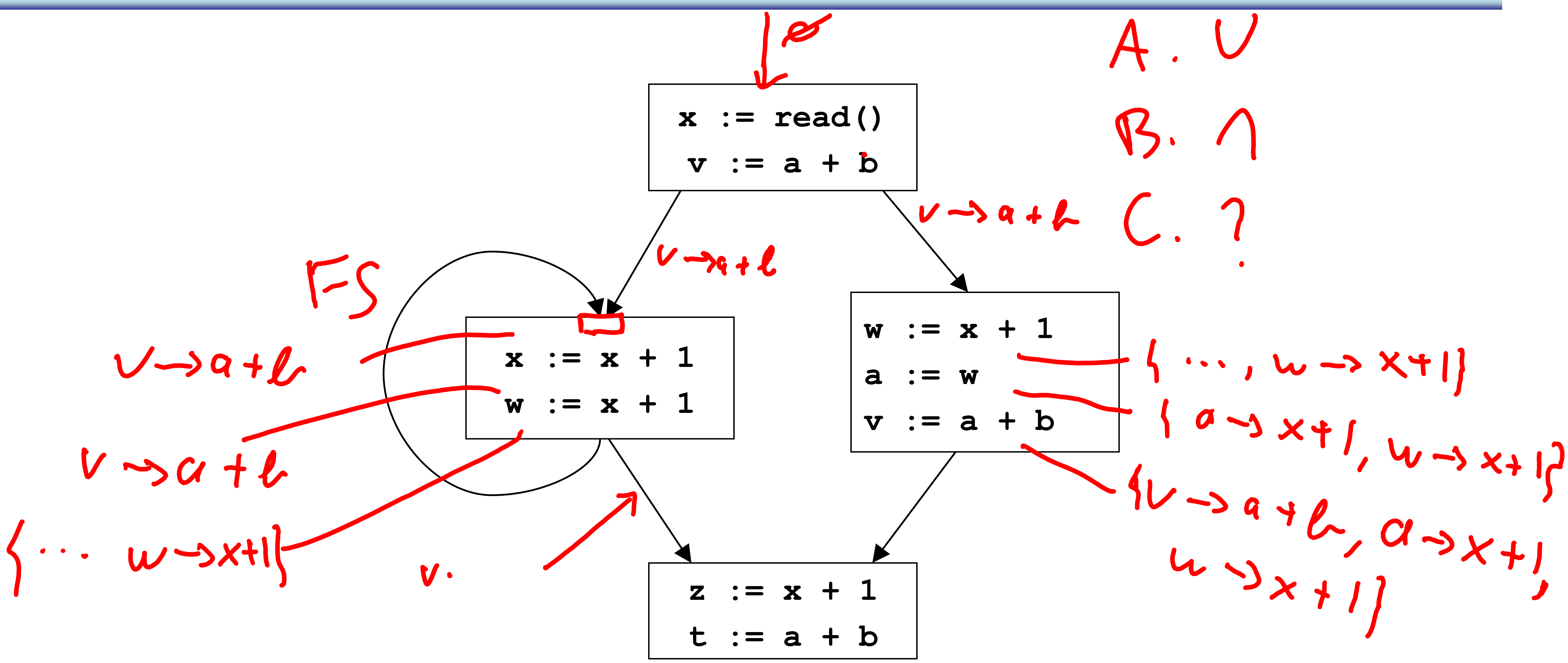
in

```
X := Y
```

out

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

Example



x := read()
v := a + b
x := x + 1
w := x + 1
w := x + 1
a := w
v := a + b
z := x + 1
t := a + b
∅
v -> a+b
FS
{..., w -> x+1}
{a -> x+1, w -> x+1}
{v -> a+b, a -> x+1, w -> x+1}
{... w->x+1}
v.

A. ∪
B. ∩
C. ?

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

a = 1
A. b = 1
B. print (a,b)
C. b = 10
D a = 20
E print (a,b)

a = 100
...
print (a)

print (a)

Example: live variables

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

Example: live variables

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

in

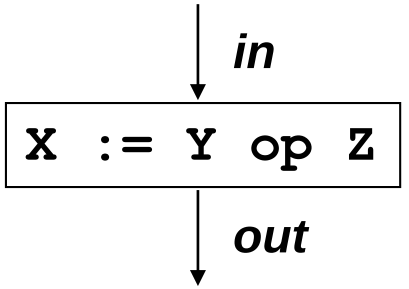
```
X := Y op Z
```

out

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

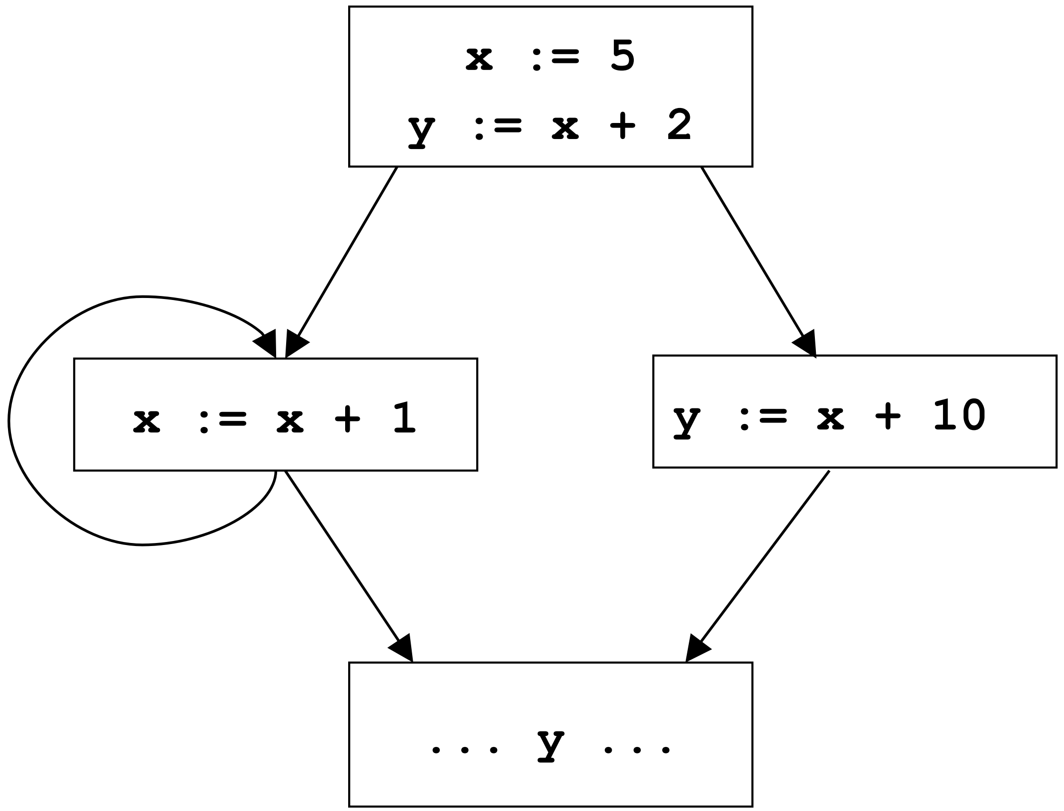
Example: live variables

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

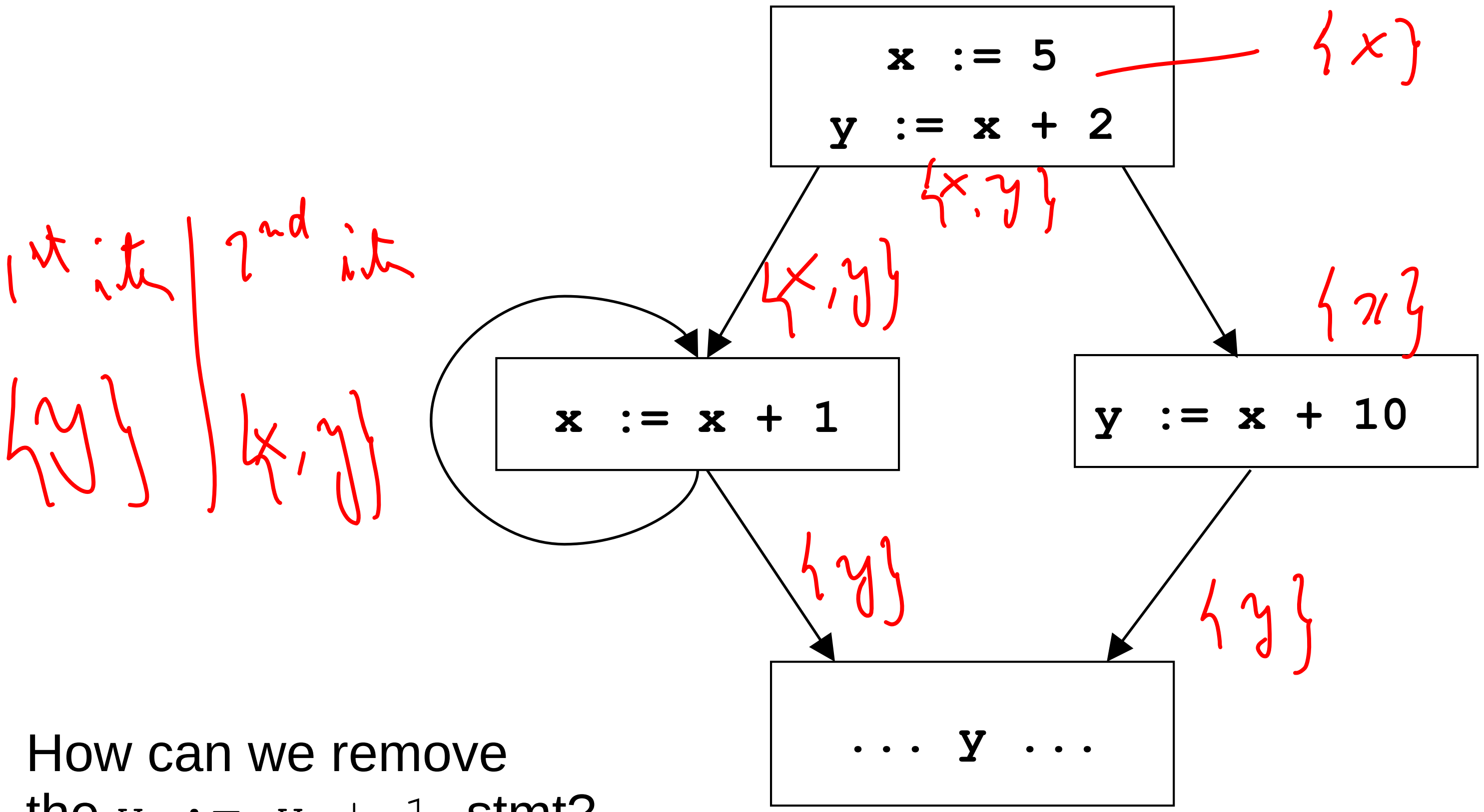


$$F_{X := Y \text{ op } Z}(\text{out}) = \text{out} - \{ X \} \cup \{ Y, Z\}$$

Example: live variables



Example: live variables



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

Revisiting assignment

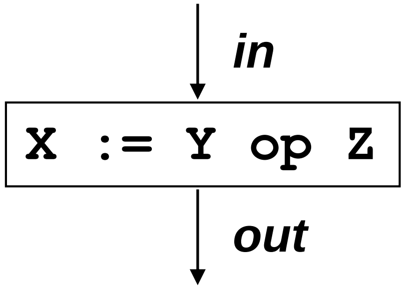
in

`X := Y op Z`

out

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

Revisiting assignment



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

$\text{out} - \{x\} \cup$

$x \notin \text{out} \; ? \; \emptyset : \{Y, Z\}$

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?

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: $F_p(in) = F_{s_n}(\ldots F_{s_1}(in) \ldots)$
- 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) = $\bigsqcup_{p \in \text{paths-to}(e)} F_p(\perp)$
- 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

- 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?

Dataflow

```
x := -1;        x := 1;
```

Merge

```
y := x * x;
... y ...
```

MOP

```
x := -1;        x := 1;
y := x * x;     y := x * x;
... y ...       ... y ...
```

Merge

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