

◆ **Homework due today**

◆ **Reading assignment for next Tues**

- **Chapters 1 and 2 of a PhD thesis on formalizing the semantics of C**
- **Linked to class web page**

◆ **Next week: Semantics**

◆ **Week after: Abstract interpretation**

Last Time

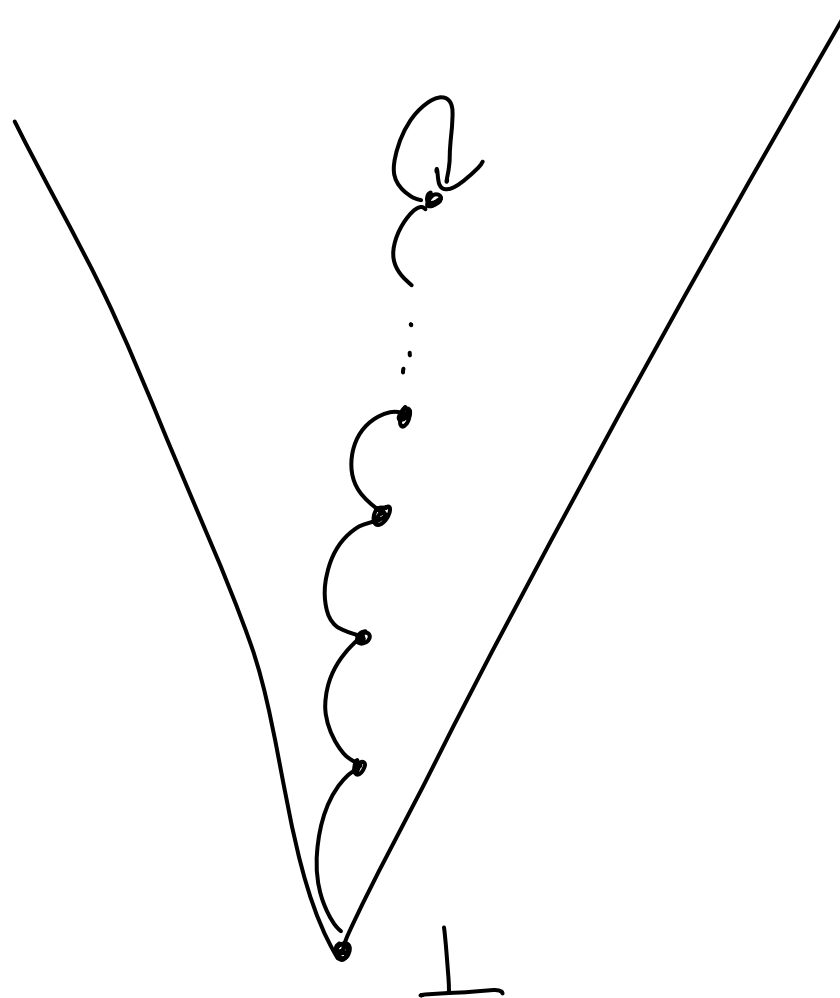
- ◆ **We want to find a fixed point of F**
 - I.e. a map m such that $m = F(m)$
- ◆ **Compute $F(\perp)$, then $F(F(\perp))$, then $F(F(F(\perp)))$, ... until the result doesn't change anymore**
- ◆ **If F is monotonic and height of lattice is finite:**
 - Iterative algorithm terminates
- ◆ **If F is monotonic:**
 - Fixpoint we find is the (unique) least fixpoint
 - Via Knaster-Tarski theorem

Today

◆ More dataflow

- Example analysis
- Analysis categories
- Dataflow vs. MOP

Graphical Analysis View



What about if we start at top?

- ◆ **What if we start with \top ?**

- $F(\widetilde{\top}) = F(F(\widetilde{\top})) = F(F(F(\widetilde{\top}))) = \dots$

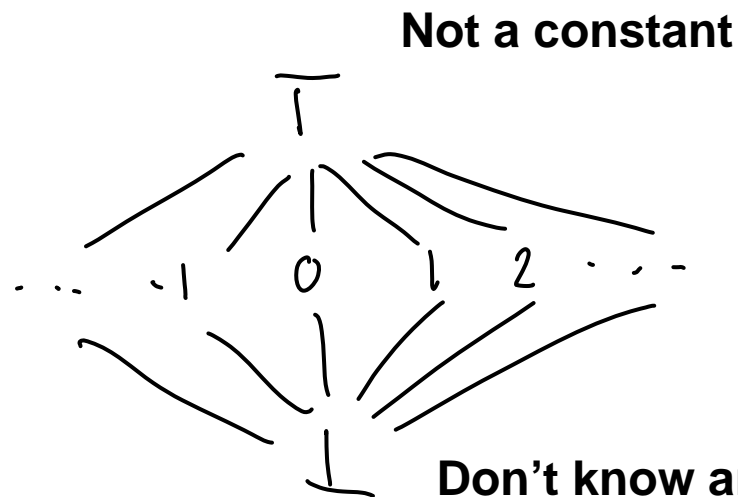
- ◆ **We get the greatest fixed point**

- That is: Least precise possible result

- This is useless: Transfer functions are not even used in any nontrivial way

Constant Propagation Lattice

- ◆ Suppose there is just one variable:



Don't know anything about this variable yet; its value cannot affect the computation

- ◆ $D = \{\perp, \top\} \cup \mathbf{Z}$
- ◆ $\forall i \in \mathbf{Z}. \perp \sqsubseteq i \wedge i \sqsubseteq \top$
- ◆ height = 3
- ◆ This easily generalizes to non-integer datatypes

Extending to Multiple Vars

- ◆ Use lattice product
- ◆ Given lattices $L_1 = (D_1, \sqsubseteq_1, \perp_1, \top_1, \sqcup_1, \sqcap_1) \dots L_n = (D_n, \sqsubseteq_n, \perp_n, \top_n, \sqcup_n, \sqcap_n)$ create:

product lattice $L^n = ((D_1 \times \dots \times D_n), \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ where

$$\perp = (\perp_1, \dots, \perp_n)$$

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

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

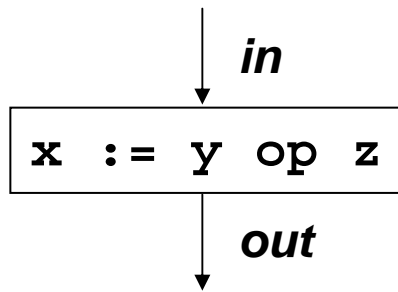
$$(a_1, \dots, a_n) \sqcap (b_1, \dots, b_n) = (a_1 \sqcap_1 b_1, \dots, a_n \sqcap_n b_n)$$

- ◆ Result is guaranteed to be a lattice
- ◆ height = height (L_1) + ... + height (L_n)
- ◆ So for 10 variables and CP, height = 30

Extending to Multiple Vars

- ◆ Think of product lattice as a mapping from variables to lattice elements
- ◆ In other words, information about individual variables, pulled out of the lattice, retains its intuitive meaning
- ◆ Sub-lattices that go into the larger lattice do not interact

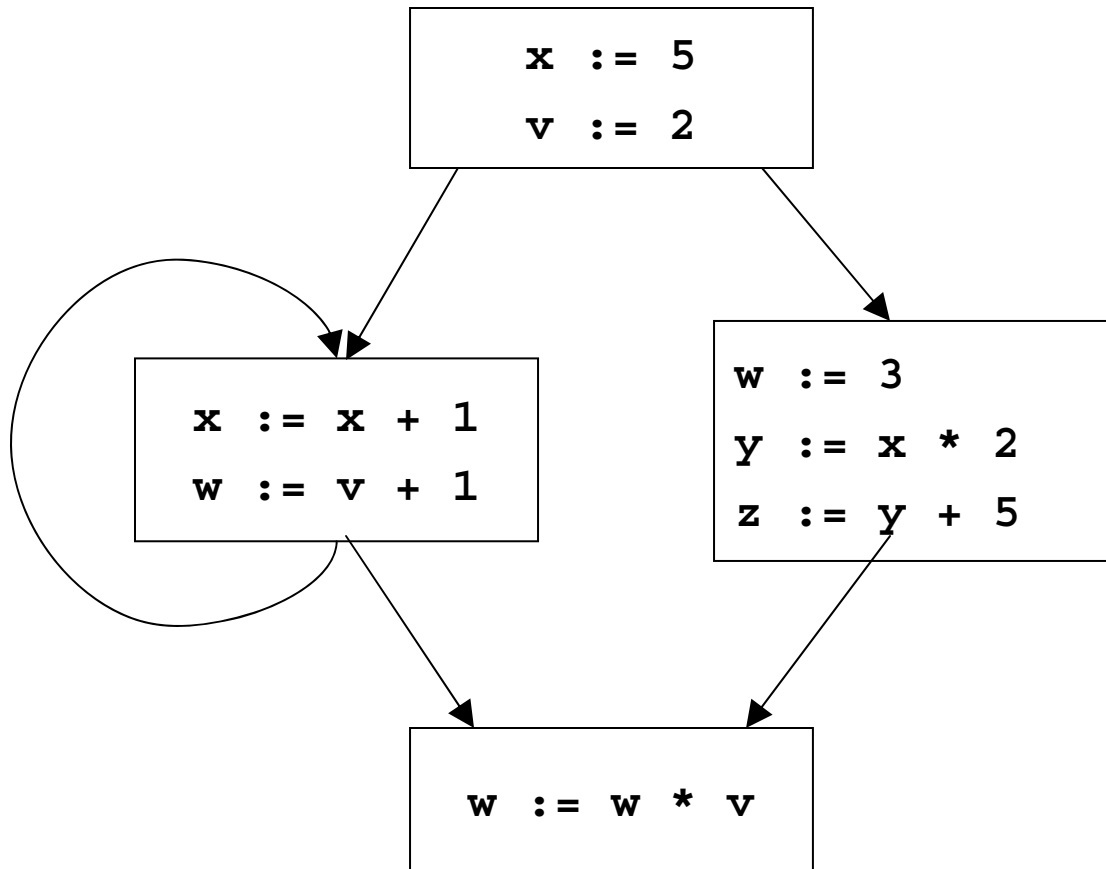
CP Transfer Functions



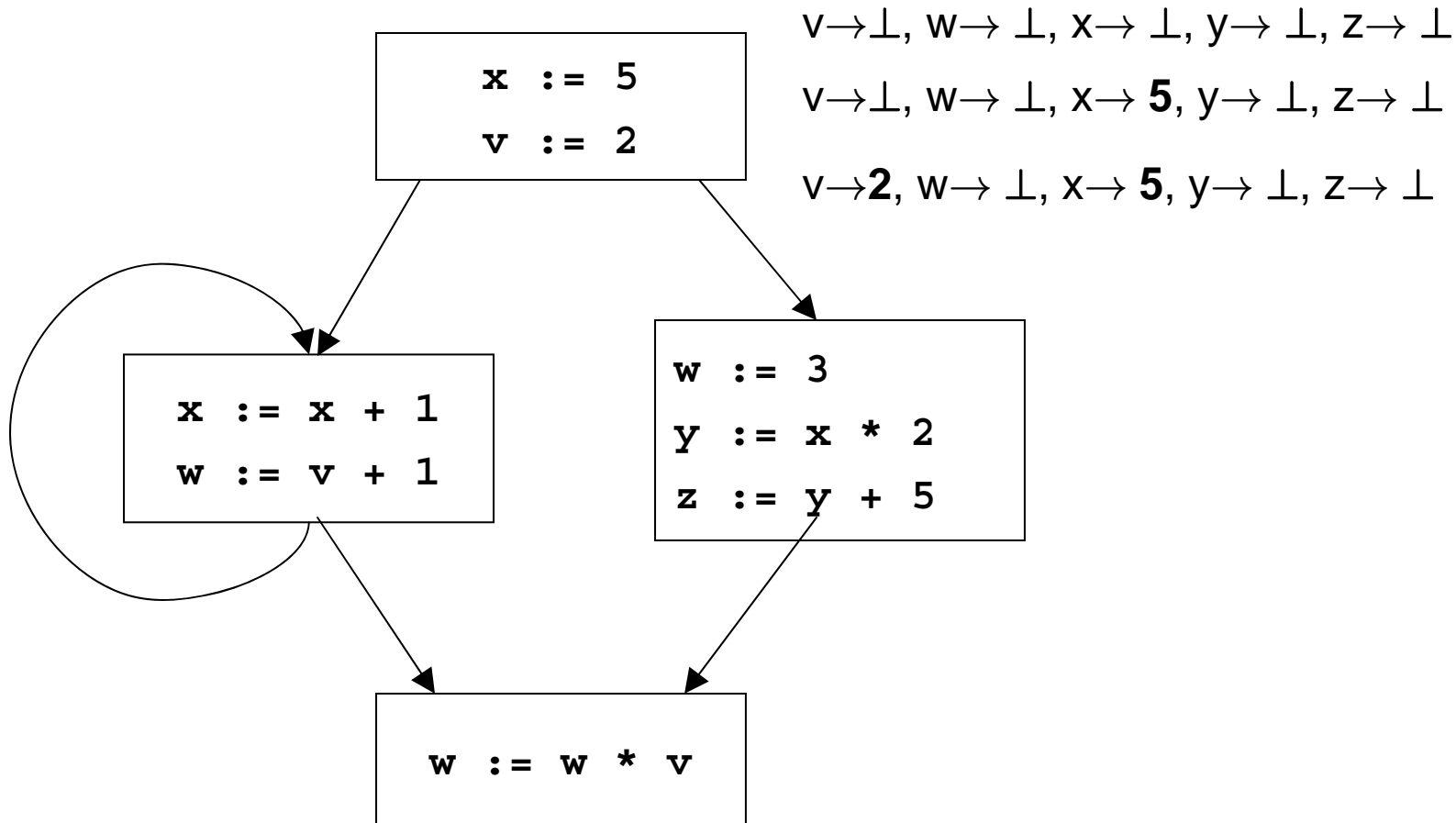
$$F_{x := y \text{ op } z}(\text{in}) = \text{in} [x \rightarrow \text{in}(y) \hat{\text{op}} \text{in}(z)]$$

$$\text{where } a \hat{\text{op}} b = \begin{cases} x \text{ op } y & \text{if } a = x \text{ and } b = y \\ \top & \text{if } a = \top \text{ or } b = \top \\ \perp & \text{otherwise} \end{cases}$$

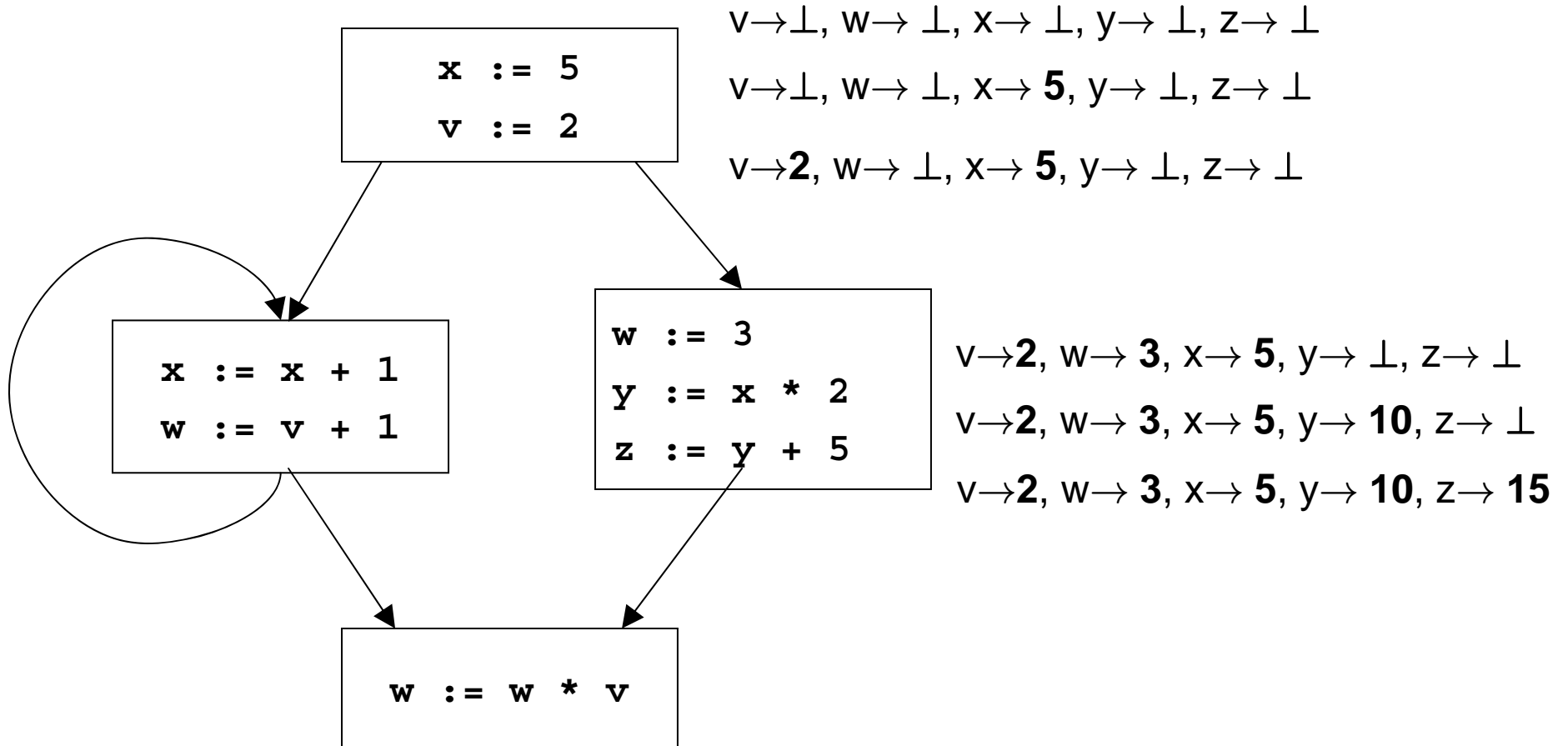
CP Example



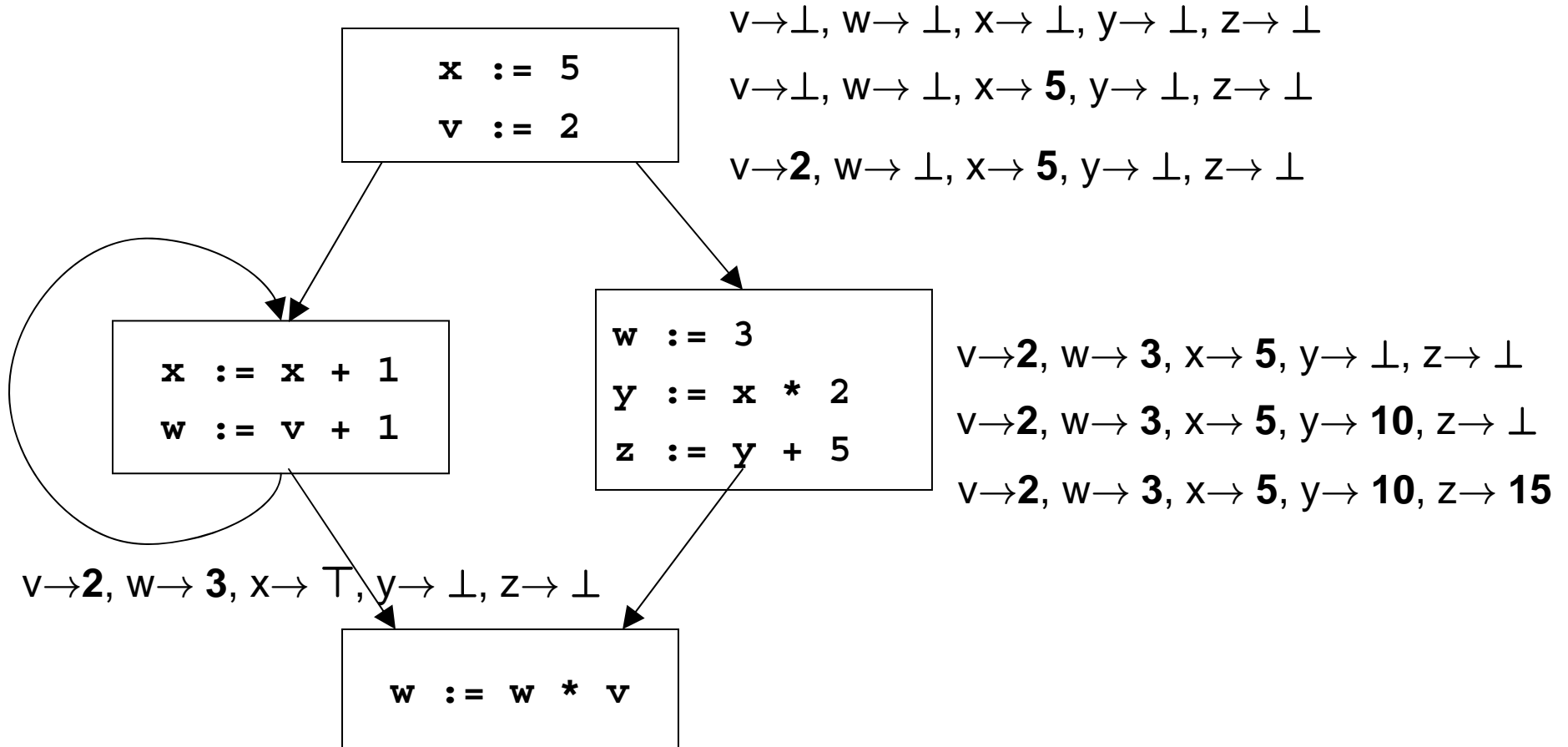
CP Example



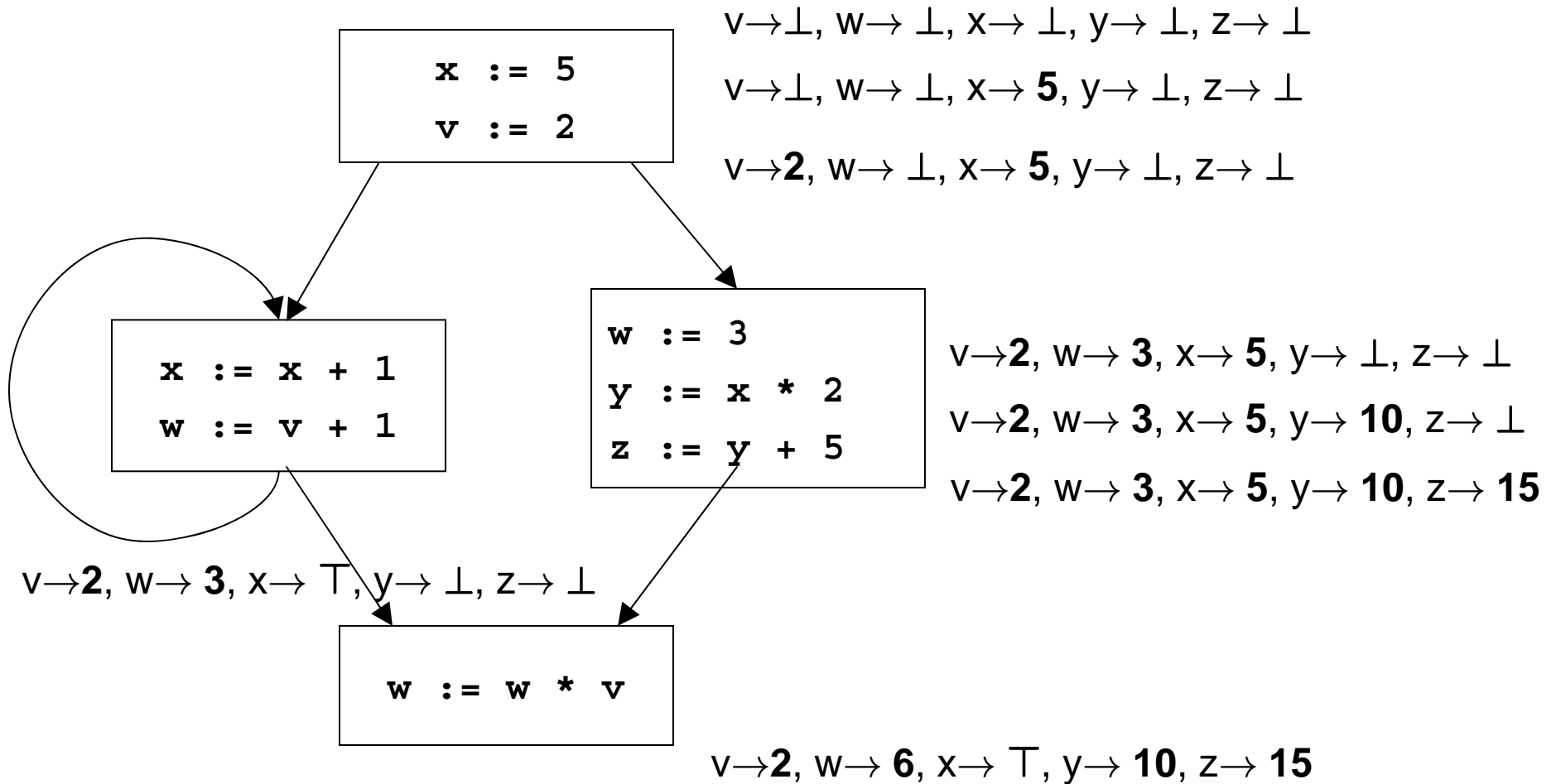
CP Example



CP Example



CP Example



A Bit of Domain Design

- ◆ **Start with simple lattices**
 - **Boolean logic lattice**
 - **Powerset lattice**
 - **Incomparable set: Set of incomparable values, plus top and bottom (e.g. constant propagation lattice)**
 - **Two point lattice: Just top and bottom**
- ◆ **Use combinators to create more complicated lattices**
 - **When lattices do not interact, this is easy**
 - **When lattices interact, this can be hard**
 - **We'll look into this more later**

May vs. Must Analyses

- ◆ 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 Analyses

	May	Must
most conservative (bottom)	Full set	Empty set
most optimistic (top)	Empty set	Full set
safe	Overly big	Overly small
join	\cup	\cap

Analysis Direction

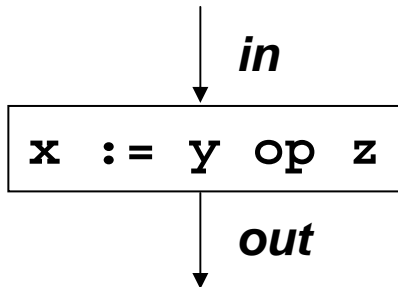
- ◆ Although constraints are not directional, transfer functions are
- ◆ All transfer functions we have seen so far are in the forward direction
- ◆ In some cases, the constraints are of the form
 $\text{in} = F(\text{out})$
 - These are called backward problems
- ◆ Example: Live variables
 - Compute the of variables that may be live

Live Variables Analysis

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

Live Variables Analysis

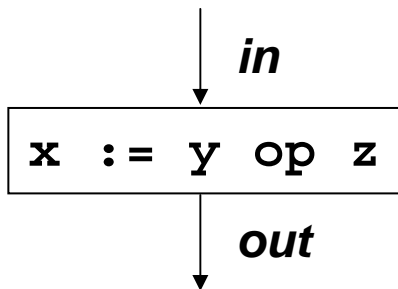
- ◆ Set $D = 2^{\text{Vars}}$
- ◆ Lattice: $(D, \subseteq, \perp, \top, \sqcup, \sqcap) = (2^{\text{Vars}}, \subseteq, \emptyset, \text{Vars}, \cup, \cap)$



$$F_{x := y \text{ op } z}(\text{out}) =$$

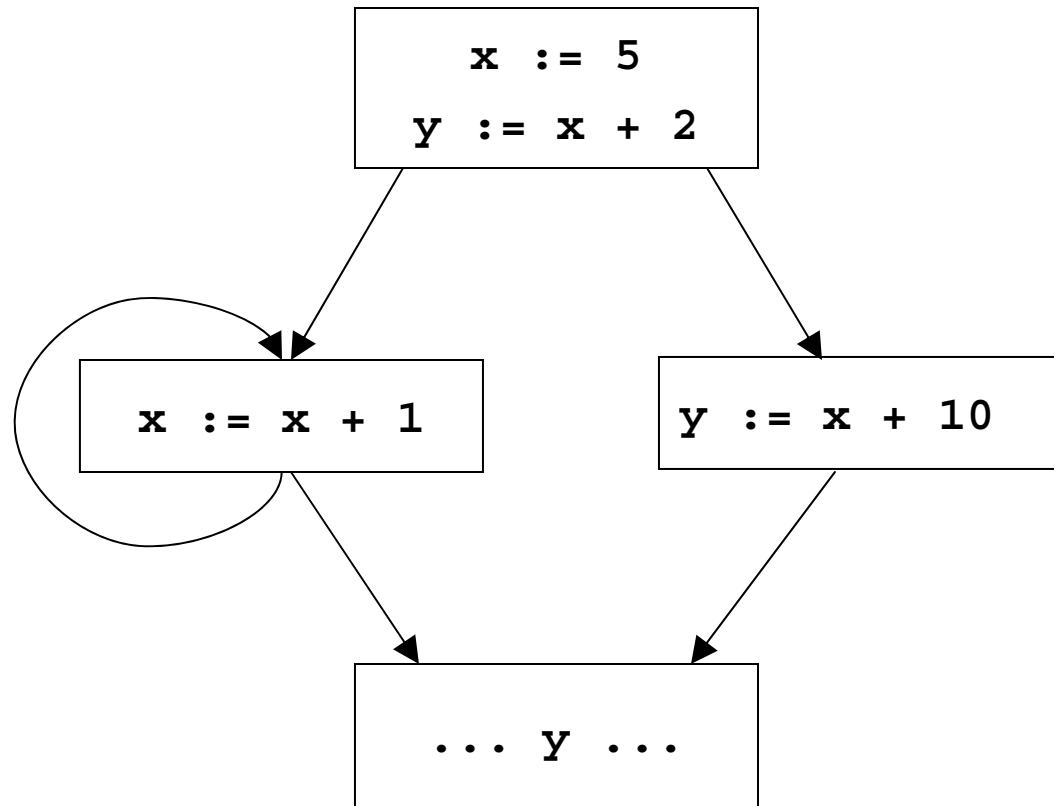
Live Variables Analysis

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

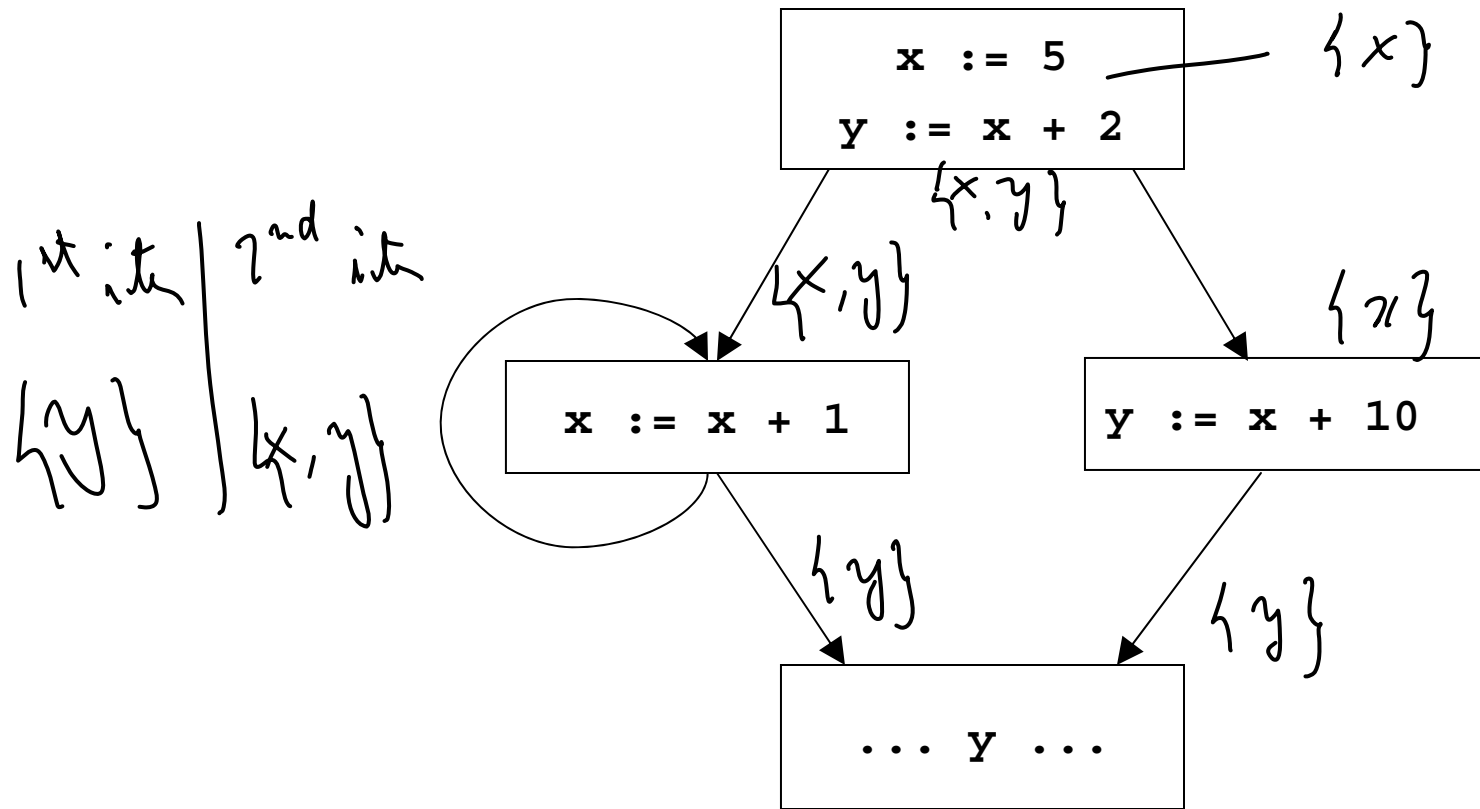


$$F_{x := y \text{ op } z}(\text{out}) = \text{out} - \{x\} \cup \{y, z\}$$

Live Variables Analysis



Live Variables Analysis



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

- ◆ Also: Can create bi-directional analyses
 - Run forwards and backwards at the same time

Precision

- ◆ Going back to constant prop, in what cases would we fail to identify a constant as constant?

```
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**
- **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 $[s_1, \dots, s_n]$, define: $F_p(\text{in}) = F_{s_n}(\dots F_{s_1}(\text{in}) \dots)$
- ◆ In other words: $F_p = F_{s_1} \circ \dots \circ F_{s_n}$
- ◆ Given an edge e , let $\text{paths-to}(e)$ be the (possibly infinite) set of paths that lead to e
- ◆ Given an edge e , $\text{MOP}(e) = \bigcup_{p \in \text{paths-to}(e)} F_p(\perp)$
- ◆ For us, should be called JOP...

MOP vs. dataflow

- ◆ As we saw in our example, in general,
MOP \neq dataflow
- ◆ In what cases is MOP the same as dataflow?
 - Distributive problems
- ◆ A problem is distributive if
$$\forall a, b . F(a \sqcup b) = F(a) \sqcup F(b)$$
- ◆ In other words, joins lose no information
- ◆ Require that F is monotonic when
$$\forall a, b . a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b)$$
- ◆ Monotonicity also implies that
$$\forall a, b . F(a \sqcup b) \sqsubseteq F(a) \sqcup F(b)$$
- ◆ So distributivity is a stricter requirement

Distributive Analyses

- ◆ **Generally, analyses are distributive when they are about *how* the program computes**
 - **Live variables**
 - **Available expressions**
 - **Reaching definitions**
 - **Very busy expressions**

- ◆ **In contrast, analyses that are about *what* a program computes are generally not distributive**
 - **Constant propagation**

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**
- ◆ **To basic dataflow, we can add path-pruning**
 - **Get branch correlation**
 - **Will see example of this later**
- ◆ **To basic dataflow, can add both**
 - **Meet over all feasible paths**