

Code analysis  
*and*  
transformation



Simone Campanoni  
simone.campanoni@northwestern.edu

# DFA foundation



We have seen several examples of DFAs

- Are they correct?
- Are they precise?
- Will they always terminate?
- How long will they take to converge?

# Outline

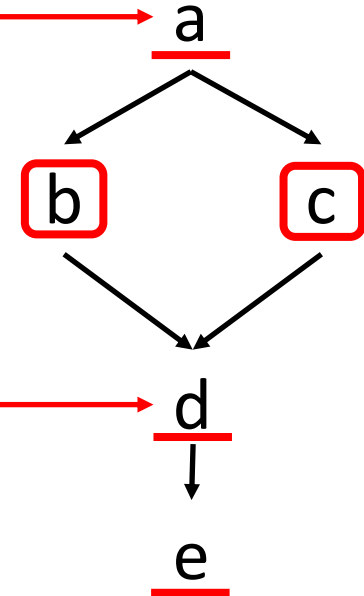
- Lattice and data-flow analysis
- DFA correctness
- DFA precision
- DFA complexity

# Understanding DFAs

- We need to understand **all** of them
  - Liveness analysis: is it correct? Precision? Convergence?
  - Reaching definitions: is it correct? Precision? Convergence?
  - ...
- **Idea:** create a framework to help reasoning about them
  - Provide a single formal model that describes all data-flow analyses
  - Formalize the notions of “correctness,” “conservativeness,” and “optimality”
  - Correctness proof for DFAs
  - Place bounds on the time complexity of iterative DFAs
  - This is not to drive the implementation, but to reason about data-flow equations

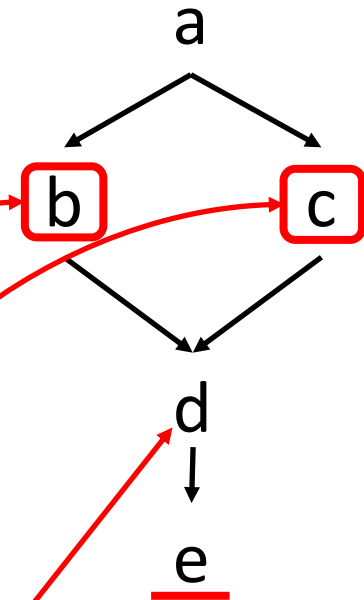
# Lattice

- Lattice  $L = (V, \leq)$ :
  - $V$  is a (possibly infinite) set of elements
  - $\leq$  is a binary relation over elements of  $V$
- Lower bound
  - $z$  is a lower bound of  $x$  and  $y$  iff  $z \leq x$  and  $z \leq y$
- Upper bound
  - $z$  is an upper bound of  $x$  and  $y$  iff  $x \leq z$  and  $y \leq z$
- Operations: meet ( $\wedge$ ) and join ( $\vee$ )
  - $b \vee c$ : least upper bound
  - $b \wedge c$ : greatest lower bound



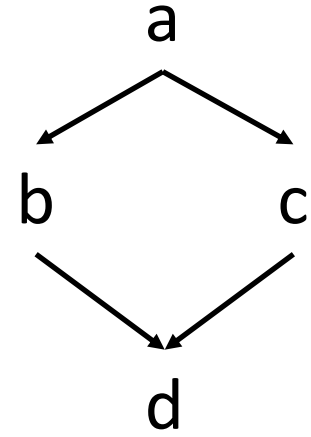
# Lattice

- Lattice  $L = (V, \leq)$ :
  - $V$  is a (possibly infinite) set of elements
  - $\leq$  is a binary relation over elements of  $V$
- Lower bound
  - $z$  is a lower bound of  $x$  and  $y$  iff  $z \leq x$  and  $z \leq y$
- Upper bound
  - $z$  is an upper bound of  $x$  and  $y$  iff  $x \leq z$  and  $y \leq z$
- Operations: meet ( $\wedge$ ) and join ( $\vee$ )
  - $b \vee c$ : least upper bound
  - $b \wedge c$ : greatest lower bound
  - An useful property: if  $e \leq b$  and  $e \leq c$ , then  $e \leq b \wedge c$



# Lattice

- Lattice  $L = (V, \leq)$ :
  - $V$  is a (possibly infinite) set of elements
  - $\leq$  is a binary relation over elements of  $V$
- Properties of  $\leq$ :
  - $\leq$  is a partial order (reflexive, transitive, anti-symmetric)
  - Every pair of elements in  $V$  has
    - An unique **greatest lower bound** (a.k.a. meet) and
    - An unique **least upper bound** (a.k.a. join)
- Top ( $T$ ) = unique greatest element of  $V$  (if it exists)
- Bottom ( $\perp$ ) = unique least element of  $V$  (if it exists)
- Height of  $L$ : longest path from  $T$  to  $\perp$ 
  - Infinite large lattice can still have finite height



*If you know nothing,  
this is still a correct,  
but conservative, solution*

# Lattice and DFA

- A lattice  $L = (V, \leq)$  describes all possible solutions of a given DFA
    - A lattice for reaching definitions
    - Another lattice for liveness analysis
    - ...
    - For DFAs that look for solutions **per point** in the CFG: one “lattice instance” per point
  - The relation  $\leq$  connects all solutions of its related DFA from the best one ( $\top$ ) to the worst one --most conservative one--( $\perp$ )
    - Liveness analysis: *variables that might be used after a given point in the CFG*
- Why?*  $\top$  = no variable is alive =  $\{ \}$   
 $\perp$  = all variables are alive =  $V$
- To solve a data-flow analysis: we traverse the lattice of a given DFA to find the correct solution **in a given point** of the CFG
    - We repeat it **for every point** in the CFG



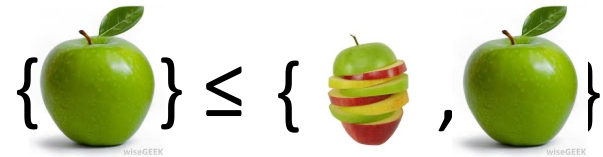
# Lattice example

- How many apples I must have?

- $V$  = sets of apples



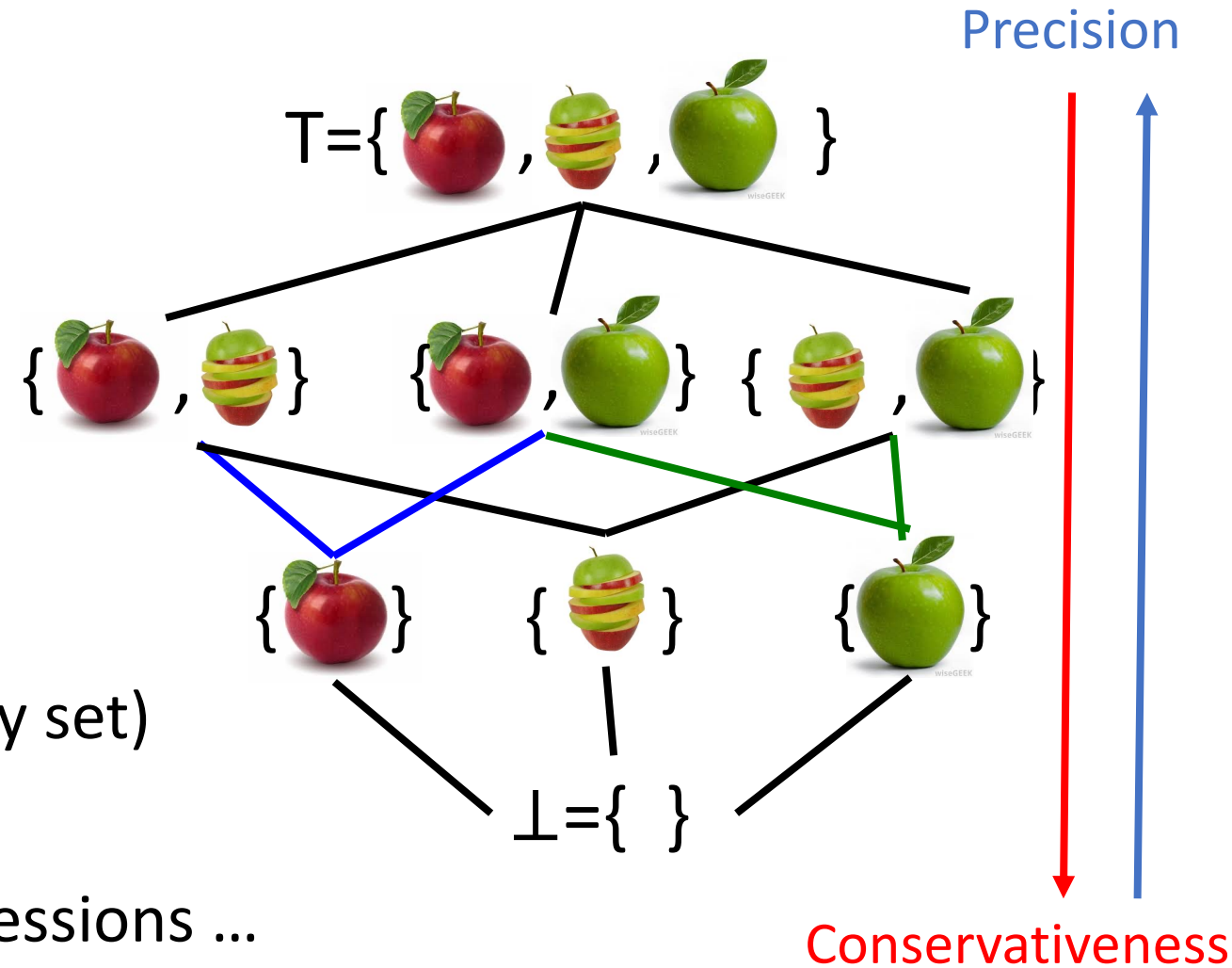
- $\leq$  = set inclusion



- $T$  = (best case) = all apples

- $\perp$  = (worst case) no apples (empty set)

Apples, definitions, variables, expressions ...



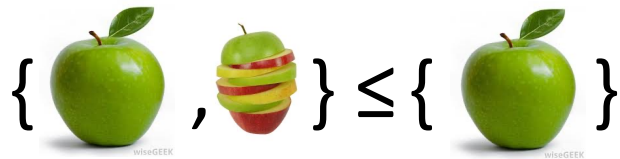
# Another lattice example

- How many apples I may have?

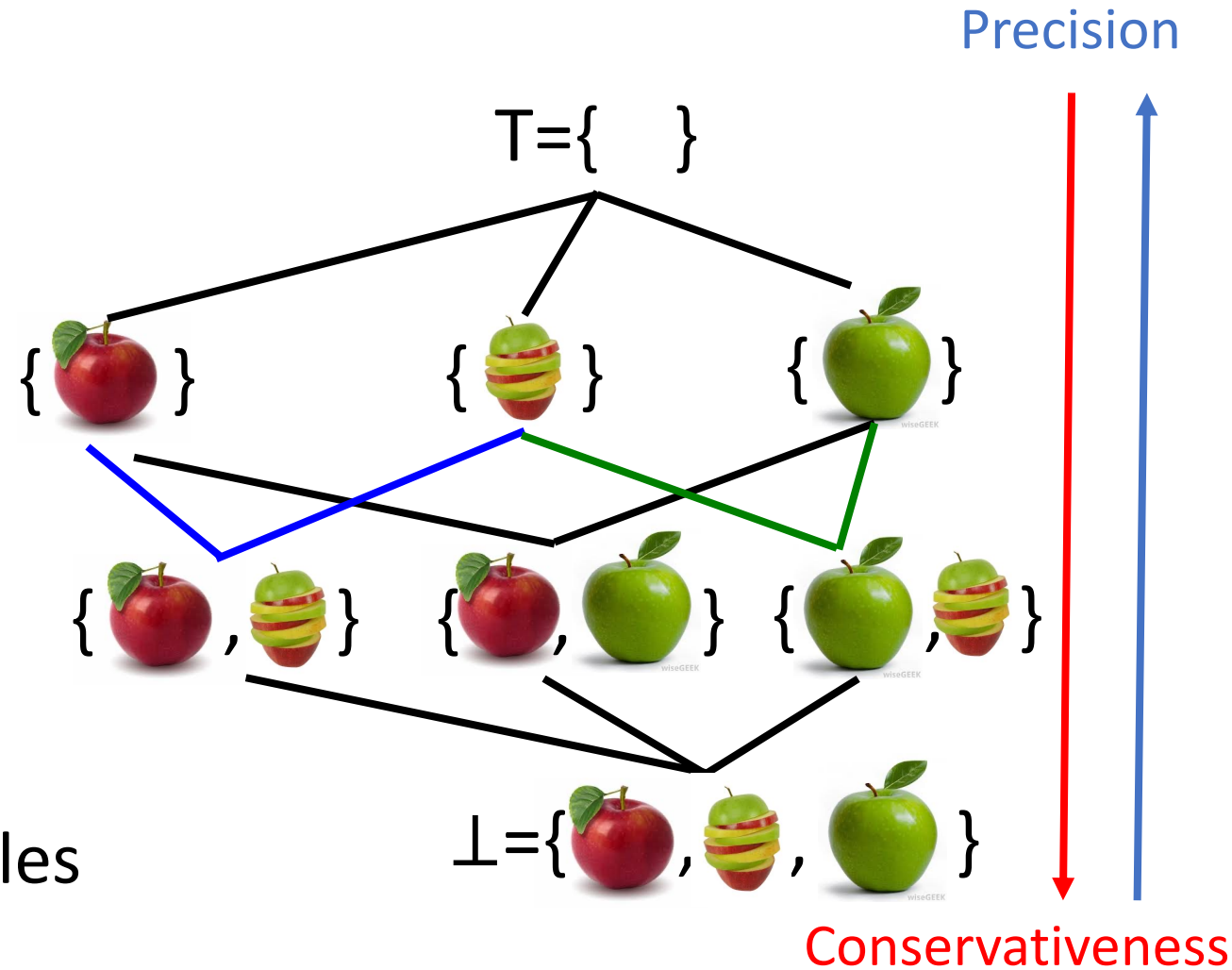
- $V$  = sets of apples



- $\leq$  = set inclusion



- $T$  = no apples (empty set)
- $\perp$  = (most conservative) all apples



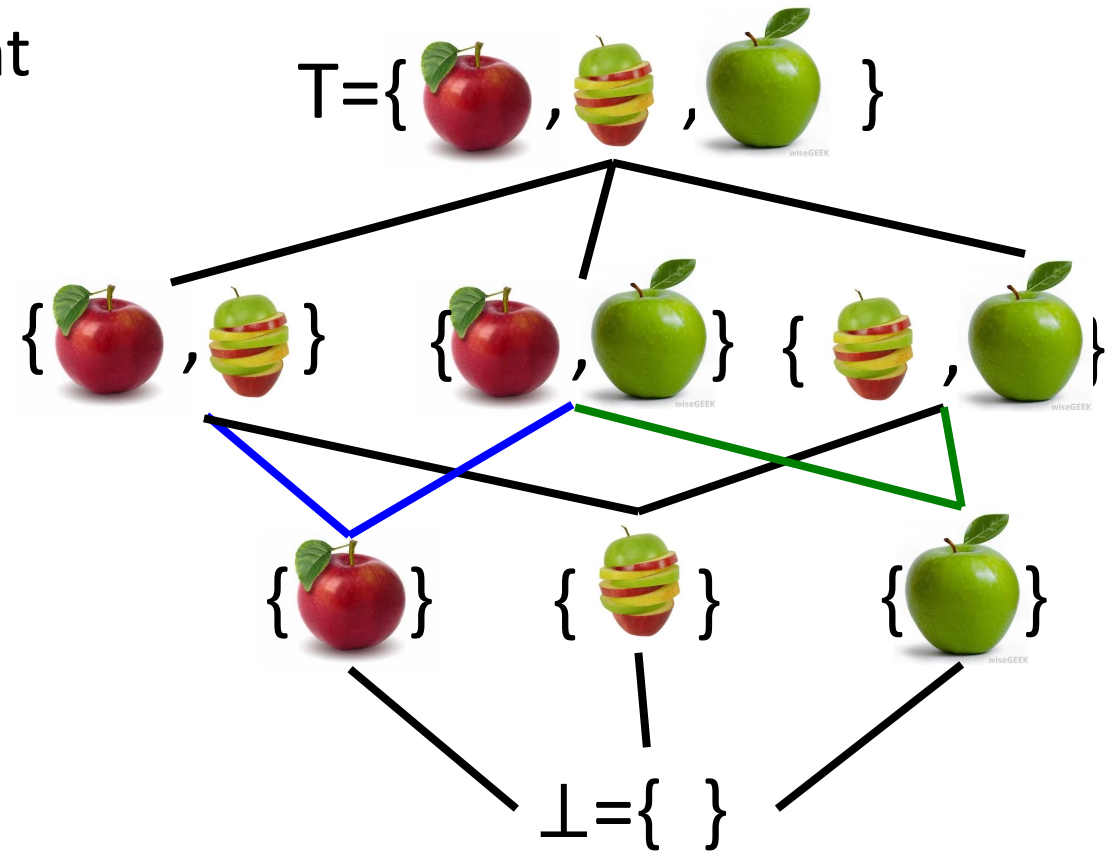
**How can we use this mathematical framework  
, lattice,  
to study a DFA?**

# Use of lattice for DFA

- Define domain of program properties (flow values --- apple sets) computed by data-flow analysis, and organize the domain of elements as a **lattice**
- Define how to traverse this domain to compute the final solution using lattice operations
- Exploit lattice theory in achieving goals

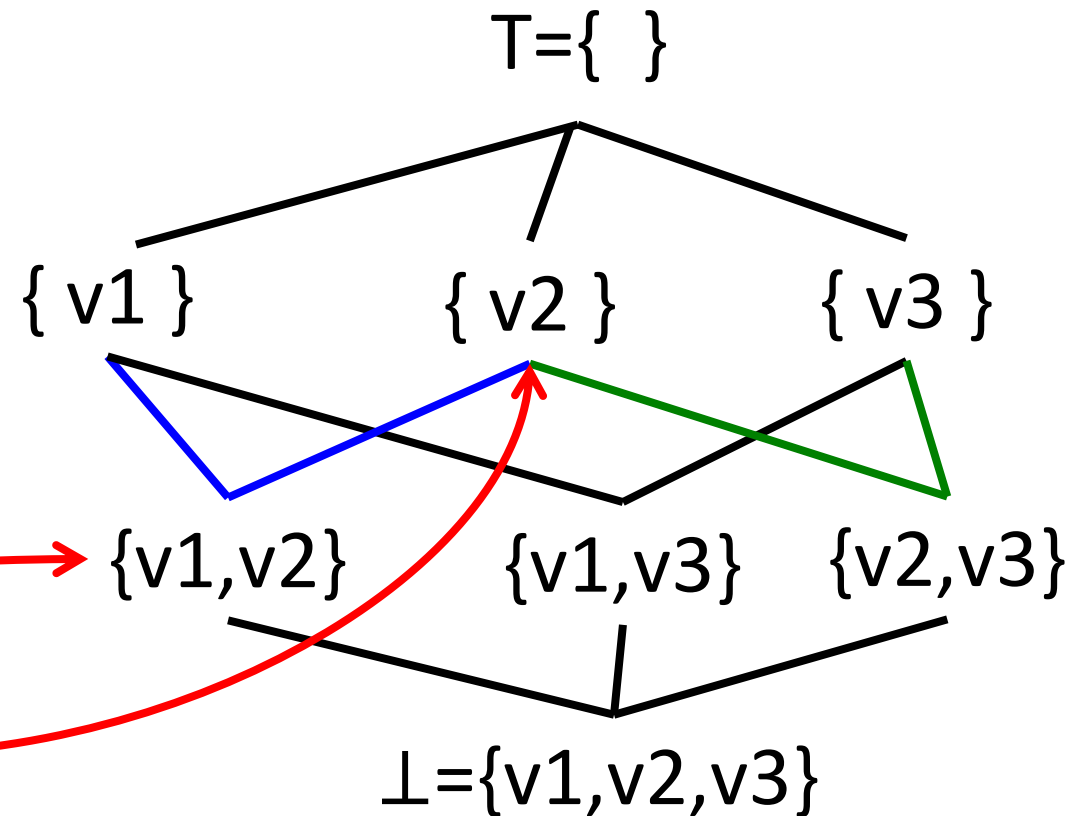
# Data-flow analysis and lattice

- Elements of the lattice (V) represent flow values (e.g., an  $IN[]$  set)
  - e.g., Sets of apples



# Data-flow analysis and lattice

- Elements of the lattice ( $V$ ) represent flow values (e.g., an  $IN[]$  set)
  - e.g., Sets of live variables for liveness
- $\perp$  “worst-case” information
  - e.g., Universal set
- $T$  “best-case” information
  - e.g., Empty set
- If  $x \leq y$ , then  $x$  is a conservative approximation of  $y$ 
  - e.g., Superset



# Data-flow analysis and lattice (reaching defs)

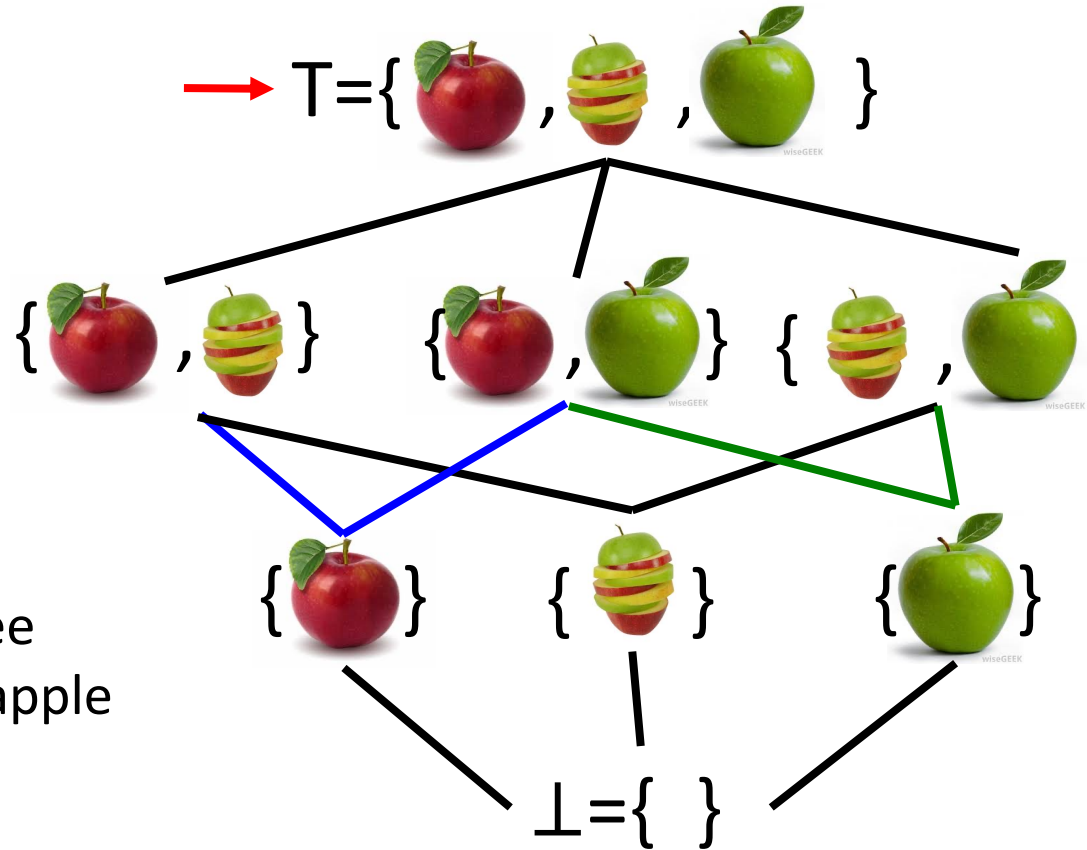
- Elements of the lattice ( $V$ ) represent flow values ( $IN[]$ ,  $OUT[]$ )
  - *e.g.*, Sets of definitions
- $T$  represents “best-case” information
  - *e.g.*, Empty set
- $\perp$  represents “worst-case” information
  - *e.g.*, Universal set
- If  $x \leq y$ , then  $x$  is a conservative approximation of  $y$ 
  - *e.g.*, Superset

**How do we choose  
which element in our lattice  
is the data-flow value  
of a given point of the input program?**



# We traverse the lattice

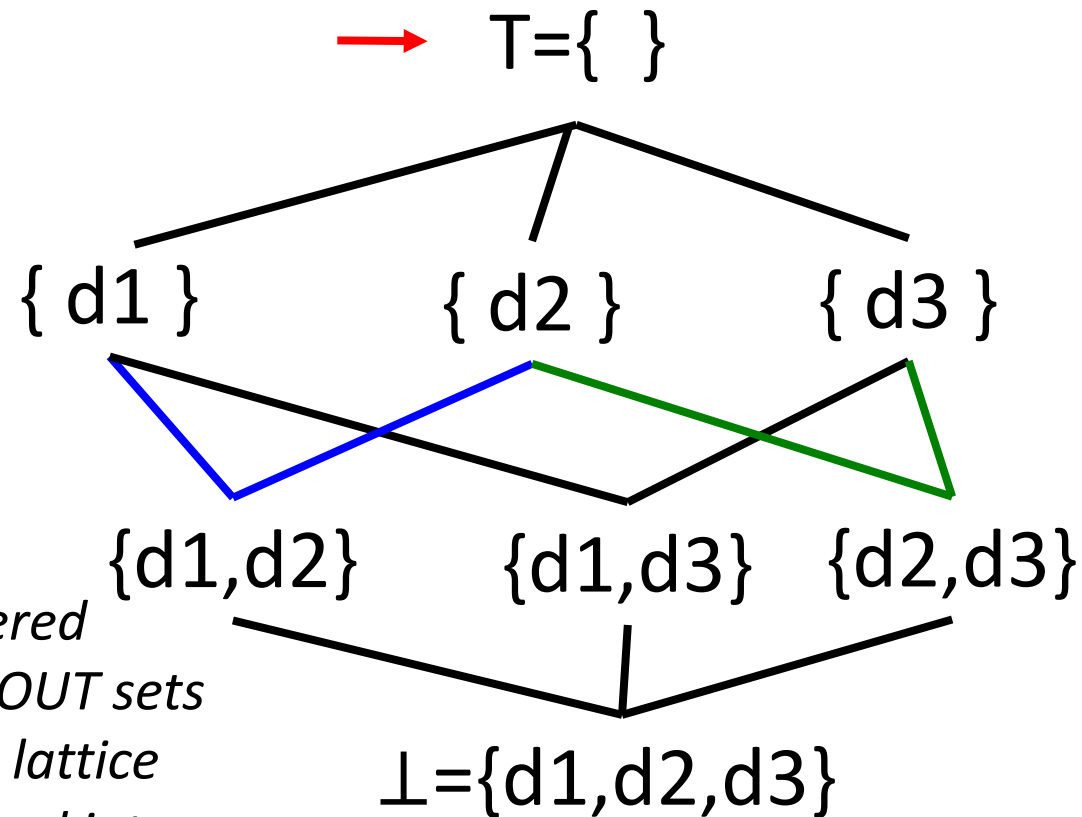
How many apples I must have?



We found out  
there is no guarantee  
we have the green apple

# We traverse the lattice

**for (each instruction  $i$  other than ENTRY)  $OUT[i] = \{ \};$**



- *New information discovered while computing the IN/OUT sets will bring us down in the lattice*
- *New information is merged into the current knowledge/state/current-point-in-the-lattice*

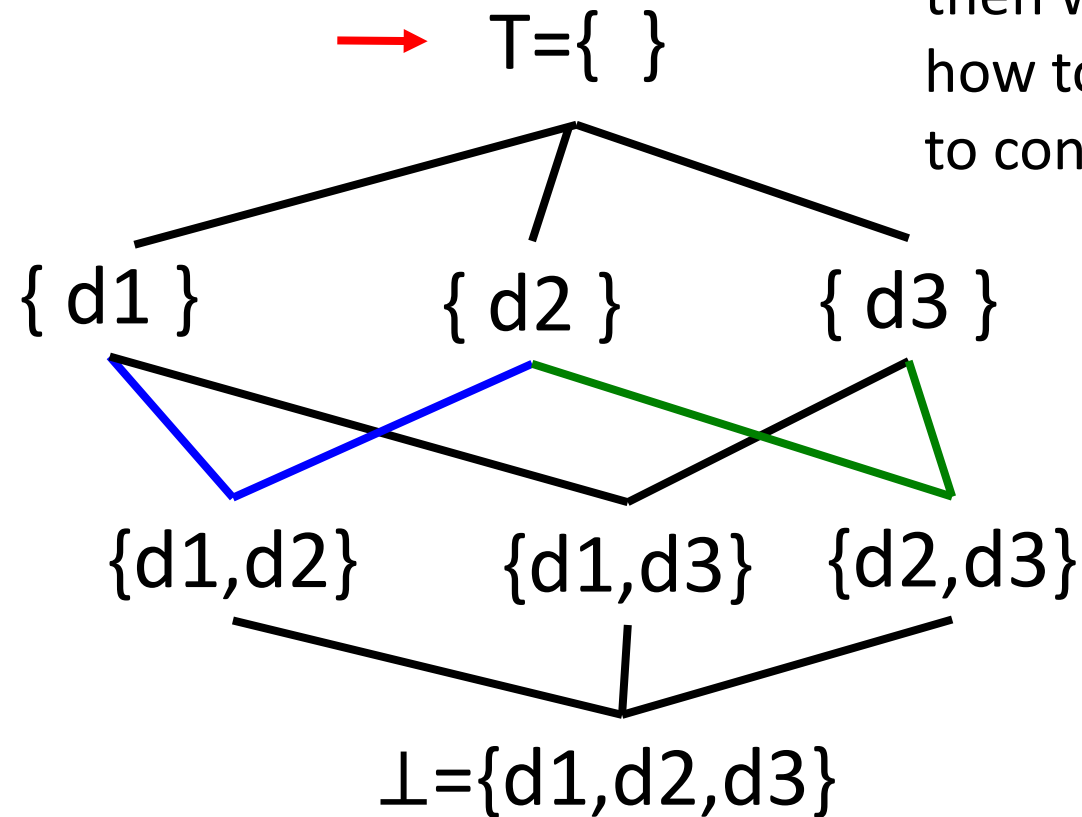
...let's see how

# Merging information

- New information is found
  - e.g., a new definition (d1) reaches a given point in the CFG
- New information is described as a point in the lattice
  - e.g. {d1}
- We use the "meet" operator ( $\wedge$ ) of the lattice to merge the new information with the current one
  - e.g., set union
  - Current information: {d2}
  - New information: {d1}
  - Result:  $\{d1\} \cup \{d2\} = \{d1, d2\}$

# We traverse the lattice

As long as we know  
how to get new information,  
then we know  
how to traverse the lattice  
to converge to the final solution



We discover:

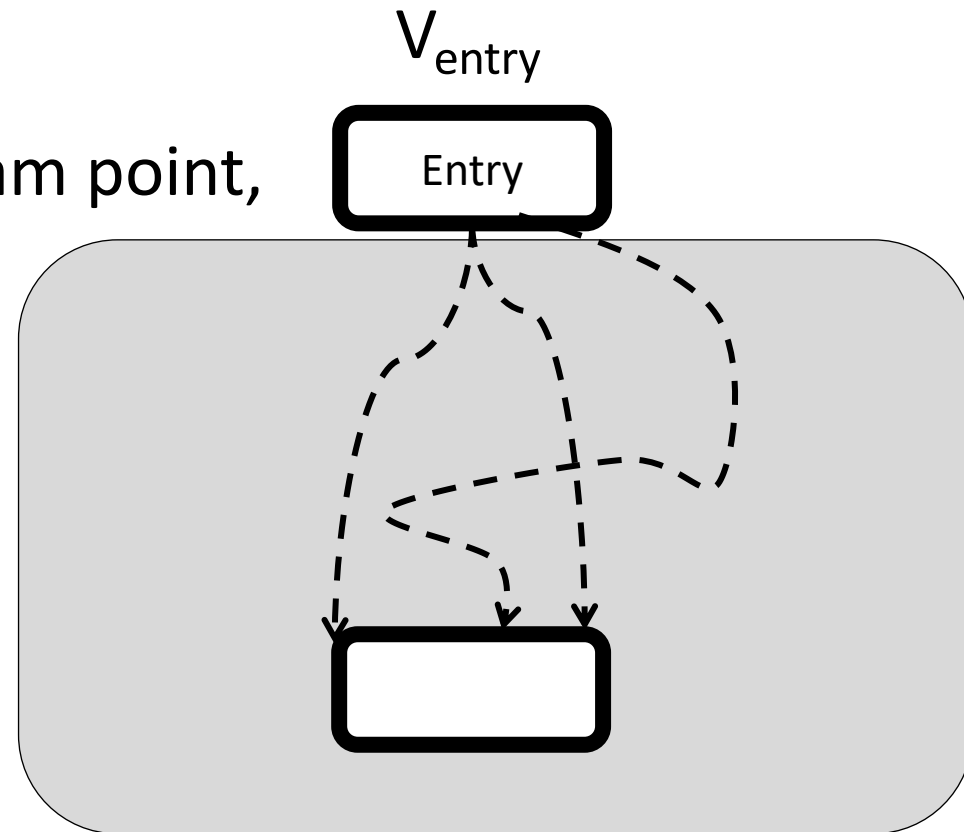
a new definition,  $d1$ ,  
reaches our point  
in the CFG

- New fact =  $\{d1\}$
- $\{ \} \wedge \{d1\} = \{d1\}$

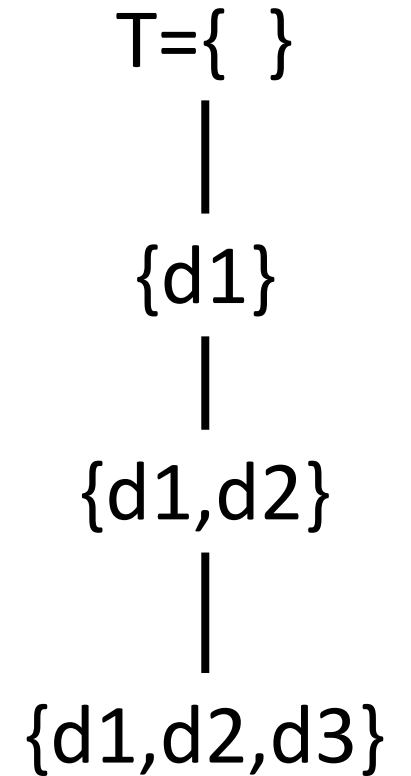
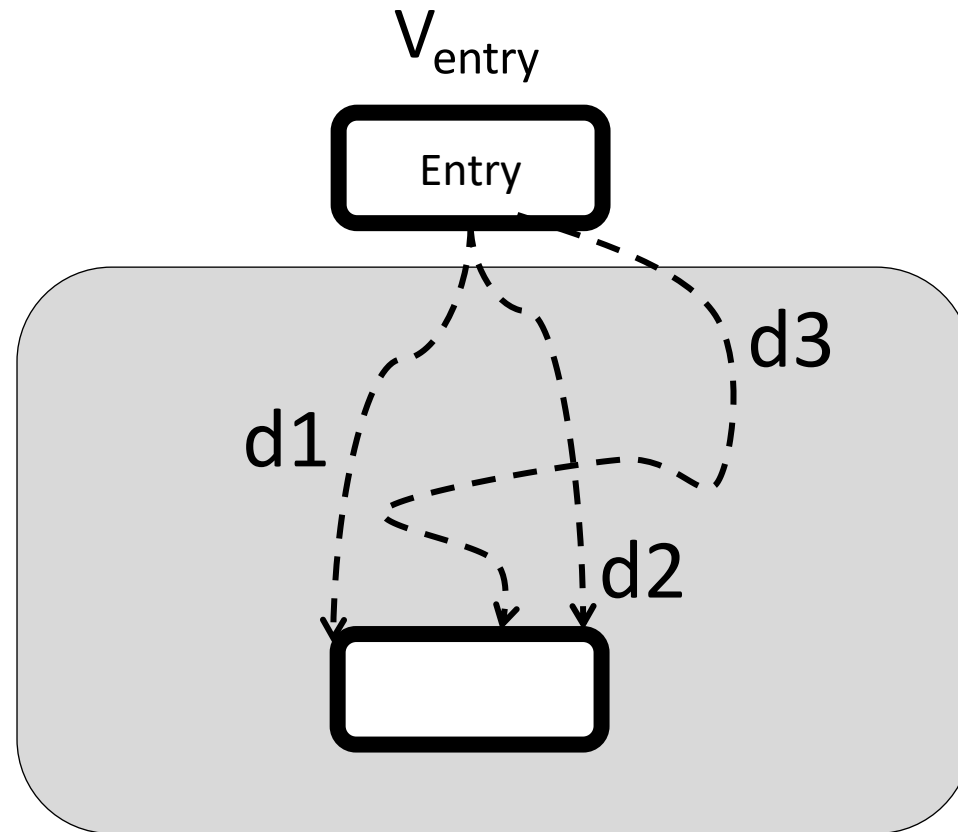
**How can we find new facts/information?**

# Computing a data-flow value (ideal)

- For a forward problem, consider all possible paths from the entry to a given program point, compute the flow values at the end of each path, and then **meet** these values together
- Meet-over-all-paths (MOP) solution at each program point
  - It's a correct solution



# Computing MOP solution for reaching definitions



# The problem of ideal solution

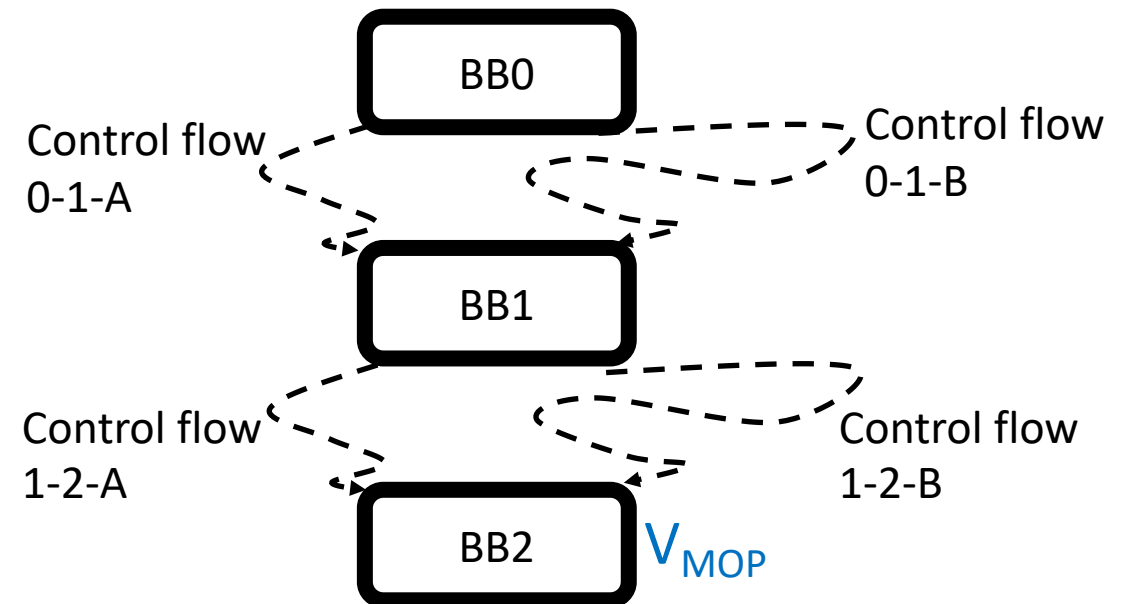
- **Problem:** all preceding paths must be analyzed
  - Exponential blow-up
- To compute the MOP solution in BB2:

0-1-A, 1-2-A

0-1-A, 1-2-B

0-1-B, 1-2-A

0-1-B, 1-2-B



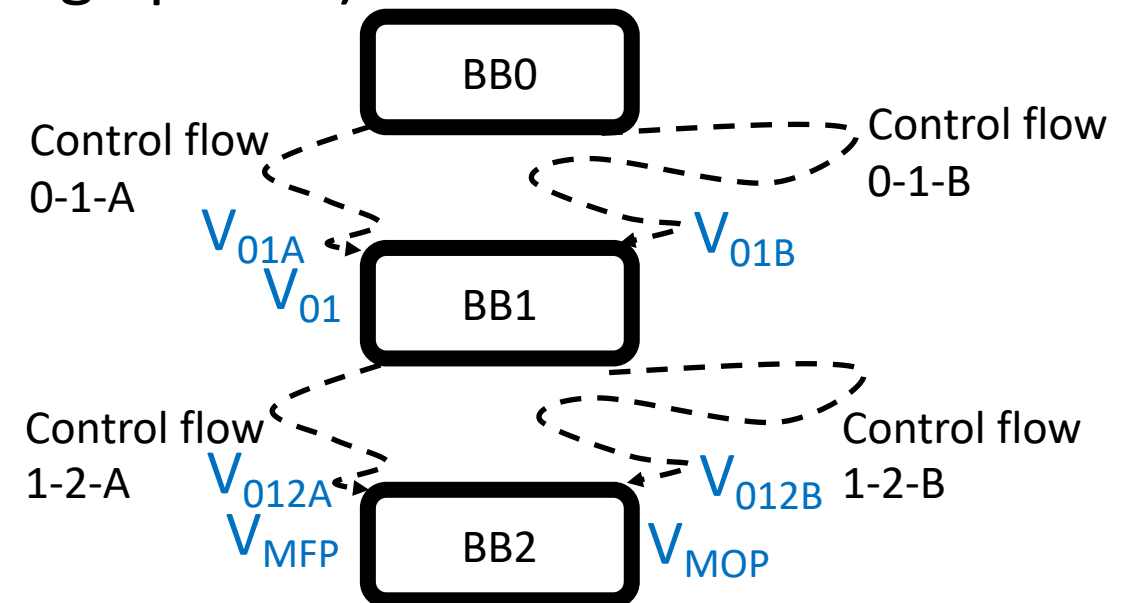


# From ideal to practical solution

- **Problem:** all preceding paths must be analyzed
  - Exponential blow-up
- **Solution:** compute meets early (at merge points) rather than at the end
  - Maximum fixed-point (MFP)

$$\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p];$$

- **Questions:**
  - Is MFP correct?
  - What's the precision of MFP?

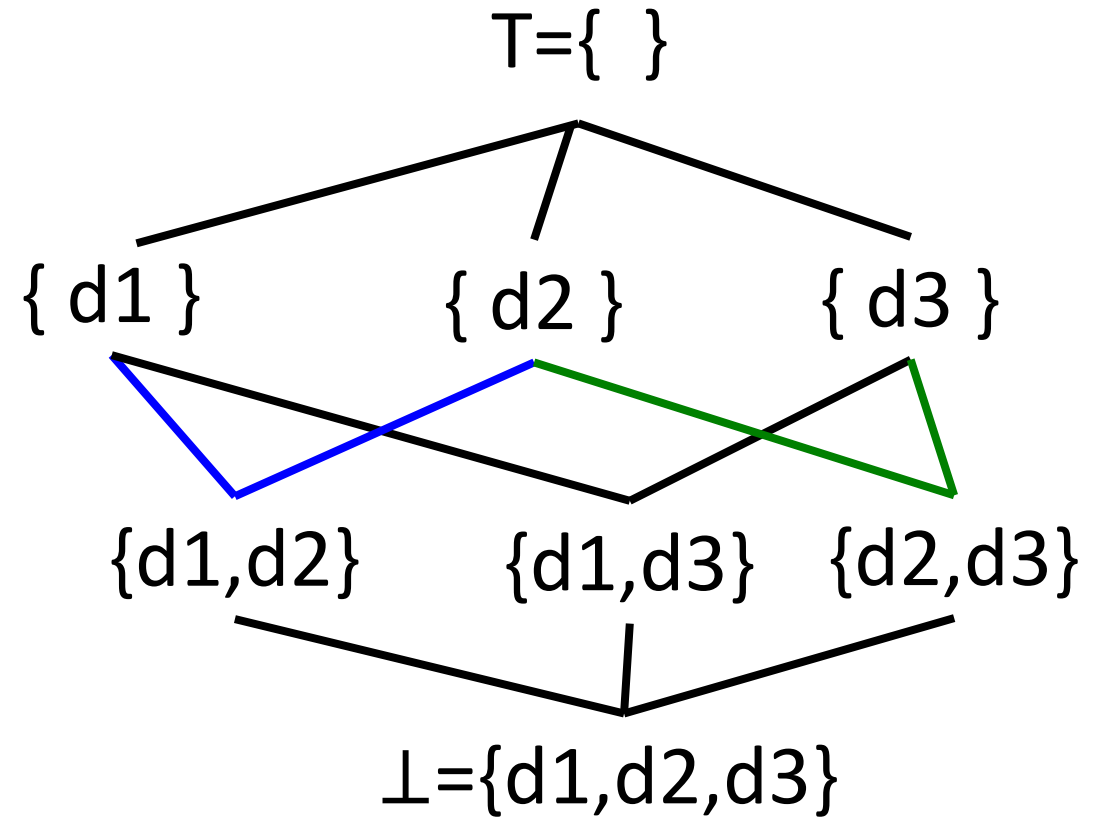
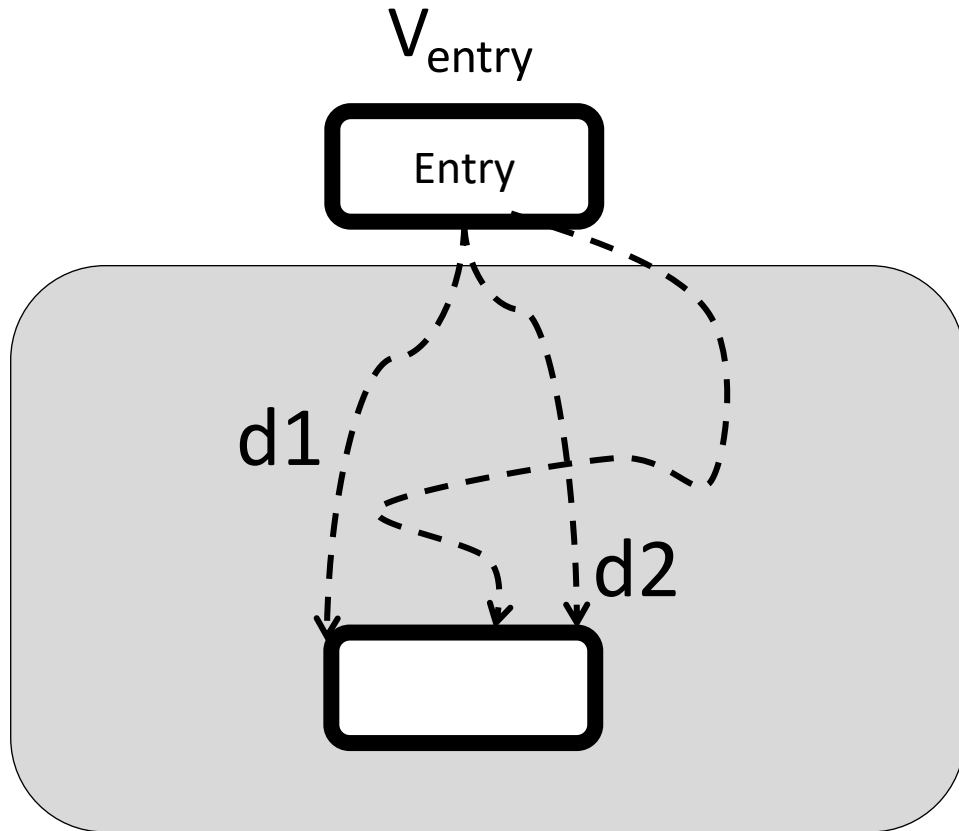


# Outline

- Lattice and data-flow analysis
- DFA correctness
- DFA precision
- DFA complexity

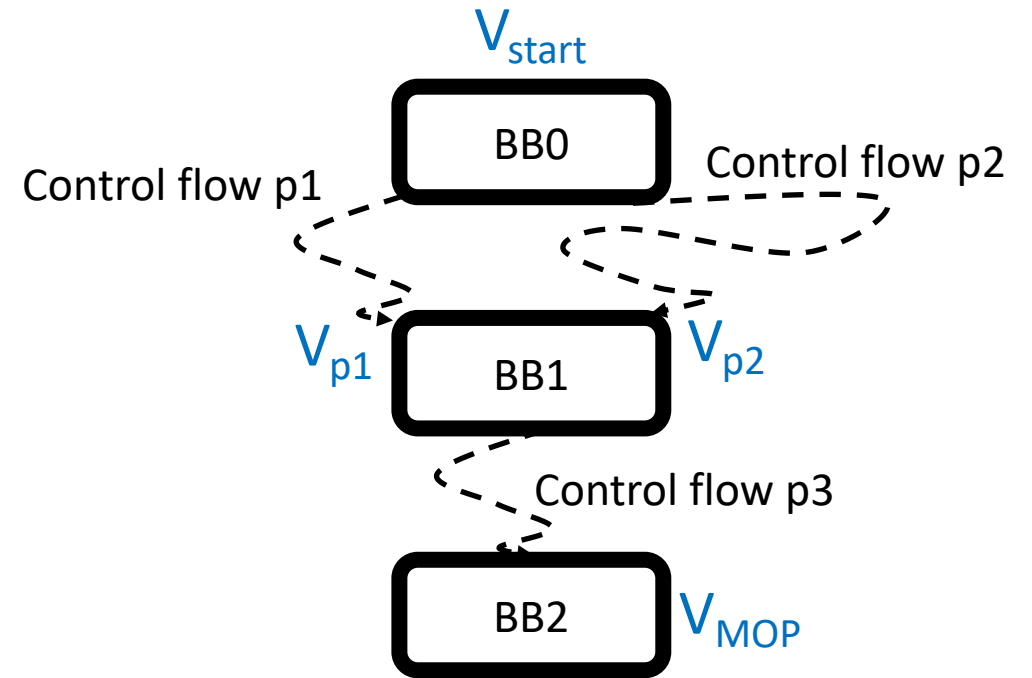
# Correctness

$$V_{\text{correct}} \leq V_{\text{MOP}}$$



# Correctness

- Key idea:
  - “Is MFP correct?” iff  $V_{MFP} \leq V_{MOP}$
- Both start from  $V_{start}$ 
  - $V_{p1} = fs_{p1}(V_{start})$
  - $V_{p2} = fs_{p2}(V_{start})$
  - $V_{MOP} = fs_{p3}(V_{p1}) \wedge fs_{p3}(V_{p2})$



# Correctness **$fs$ is monotonic $\Rightarrow$ MFP is correct!**

- Key idea:

- “Is MFP correct?” iff  $V_{MFP} \leq V_{MOP}$

- Focus on merges:  $IN[i] = \cup_{p \text{ a predecessor of } i} OUT[p];$

- $V_{MOP} = fs_{p3}(V_{p1}) \wedge fs_{p3}(V_{p2})$
- $V_{MFP} = fs_{p3}(V_{p1} \wedge V_{p2})$
- $V_{MFP} \leq V_{MOP}$  iff  $fs_{p3}(V_{p1} \wedge V_{p2}) \leq fs_{p3}(V_{p1}) \wedge fs_{p3}(V_{p2})$

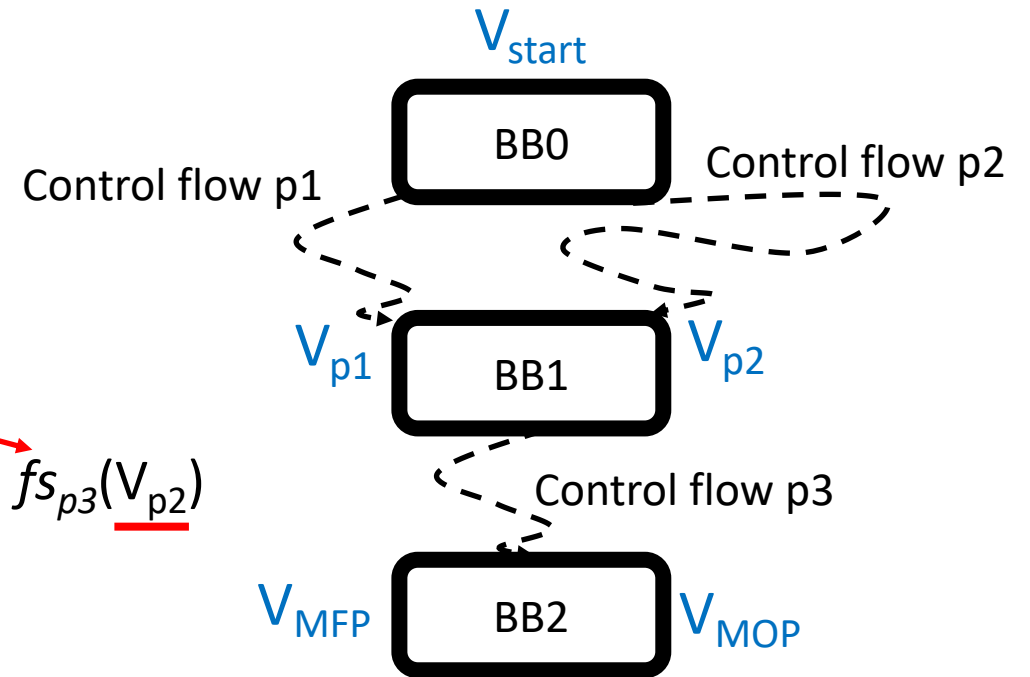
*Same function*

*Let us compare*

- If  $fs$  is monotonic:  $X \leq Y$  then  $fs(X) \leq fs(Y)$

- $(V_{p1} \wedge V_{p2}) \leq V_{p1}$  by definition of meet
- $(V_{p1} \wedge V_{p2}) \leq V_{p2}$  by definition of meet
- So  $fs_{p3}(V_{p1} \wedge V_{p2}) \leq fs_{p3}(V_{p1})$  and  $fs_{p3}(V_{p1} \wedge V_{p2}) \leq fs_{p3}(V_{p2})$
- Therefore  $fs(V_{p1} \wedge V_{p2}) \leq fs(V_{p1}) \wedge fs(V_{p2})$
- And therefore  $V_{MFP} \leq V_{MOP}$

An useful property: if  $e \leq b$  and  $e \leq c$ , then  $e \leq b \wedge c$



# Monotonicity

- $X \leq Y$  then  $fs(X) \leq fs(Y)$
- If the flow function  $f$  is applied to two members of  $V$ , the result of applying  $f$  to the “lesser” of the two members will be under the result of applying  $f$  to the “greater” of the two
- More conservative inputs leads to more conservative outputs (never more optimistic outputs)

# Convergence

- **From lattice theory**

If  $f$ s is monotonic,

then the maximum number of times  $f$ s can be applied w/o reaching a fixed point is  $\text{Height}(V) - 1$

- Iterative DFA is guaranteed to terminate if the  $f$ s is monotonic and the lattice has finite height

# Outline

- Lattice and data-flow analysis
- DFA correctness
- **DFA precision**
- **DFA complexity**

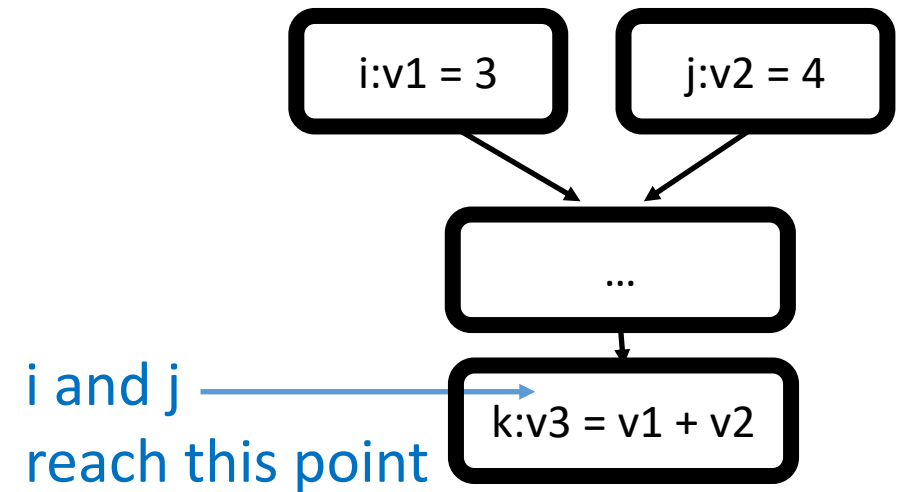


# Precision

- $V_{MOP}$ : the best solution
- $V_{MFP} \leq V_{MOP}$ 
  - $fs(V_{p1} \wedge V_{p2}) \leq fs(V_{p1}) \wedge fs(V_{p2})$
- Distributive  $fs$  over  $\wedge$ 
  - $fs(V_{p1} \wedge V_{p2}) = fs(V_{p1}) \wedge fs(V_{p2})$
  - $V_{MFP} = V_{MOP}$
- Is reaching definition  $fs$  distributive?
  - (did having performed  $\wedge$  earlier in the CFG change anything?)

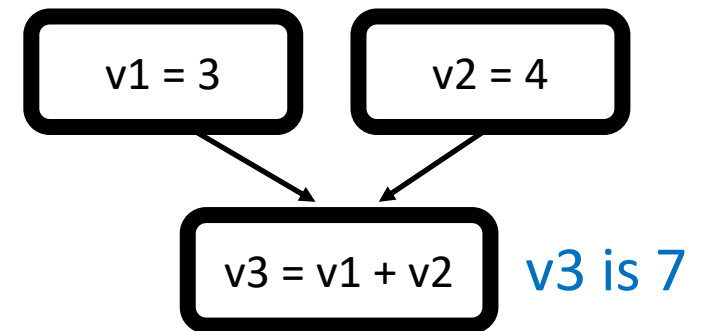
\* is distributive over +  
 $4 * (2 + 3) = 4 * (5) = 20$

$(4 * 2) + (4 * 3) = 8 + 12 = 20$



# A new DFA example: reaching constants

- Goal
  - Compute the value that a variable must have at a program point (no SSA)
- Flow values (V)
  - Set of (variable, constant) pairs
- Merge function
  - Intersection
- Data-flow equations
  - Effect of node n:  $x = c$ 
    - $KILL[n] = \{(x, k) \mid \forall k\}$
    - $GEN[n] = \{(x, c)\}$
  - Effect of node n:  $x = y + z$ 
    - $KILL[n] = \{(x, k) \mid \forall k\}$
    - $GEN[n] = \{(x, c) \mid c = \text{val}_y + \text{val}_z, (y, \text{val}_y) \in IN[n], (z, \text{val}_z) \in IN[n]\}$



# Reaching constants: characteristics

- $\perp = ?$
- $IN = ?$
- $OUT = ?$
- Let's study this analysis
  - Does it convergence?
    - is  $fs$  monotonic? Has the lattice a finite height?
  - What is the precision of the solution?
    - is  $fs$  distributive?

# Outline

- Lattice and data-flow analysis
- DFA correctness
- DFA precision
- **DFA complexity**

# Complexity

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]$ ;
    OUT[i] = GEN[i]  $\cup$  (IN[i] – KILL[i]);
  }
} while (changes to any OUT occur);
```

# Complexity

- N instructions (N definitions at most)
- Complexity of the computation of IN[i]
  - Each IN/OUT set has at most N elements
  - Each set-union operation takes O(N) time
  - Constant number of set operations per instruction
    - The computation of IN[i] is O(N) time

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\cup_{p \text{ a predecessor of } i} \text{OUT}[p]$ ;
    OUT[i] = GEN[i]  $\cup$  (IN[i] - KILL[i]);
  }
} while (changes to any OUT occur);
```

# Complexity

- N instructions (N definitions at most)
- Complexity of the computation of IN[i]
  - O(N)
- Complexity of an iteration
  - Constant number of set operations per iteration
  - O(N)

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\cup_{p \text{ a predecessor of } i} \text{OUT}[p]$ ;
    OUT[i] = GEN[i]  $\cup$  (IN[i] - KILL[i]);
  }
} while (changes to any OUT occur);
```

# Complexity

- N instructions (N definitions at most)
- Complexity of the computation of IN[i]
  - O(N)
- Complexity of an iteration
  - O(N)
- Complexity of an invocation
  - O(N) instructions  $\Rightarrow$  O(N<sup>2</sup>) time per invocation of the loop

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\cup_{p \text{ a predecessor of } i}$  OUT[p];
    OUT[i] = GEN[i]  $\cup$  (IN[i] - KILL[i]);
  }
} while (changes to any OUT occur);
```



# Complexity

- N instructions (N definitions at most)
- Complexity of the computation of  $IN[i]$ 
  - $O(N)$
- Complexity of an iteration
  - $O(N)$
- Complexity of an invocation
  - $O(N^2)$
- Complexity of do-while
  - Each do-while iteration modifies in the worst case only one set
  - Each modification can only add one element in the worst case
  - So the computation of a **single set** can take up to  $O(N^3)$
  - There are N sets:  $O(N^4)$

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
     $IN[i] = \bigcup_{p \text{ a predecessor of } i} OUT[p]$ ;
     $OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$ ;
  }
} while (changes to any OUT occur);
```

# Complexity

- N instructions (N definitions at most)
- Complexity of the computation of IN[i]
  - $O(N)$
- Complexity of an iteration
  - $O(N)$
- Complexity of an invocation
  - $O(N^2)$
- Complexity of do-while
  - $O(N^4)$

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\cup_{p \text{ a predecessor of } i}$  OUT[p];
    OUT[i] = GEN[i]  $\cup$  (IN[i] - KILL[i]);
  }
} while (changes to any OUT occur);
```

# Complexity

- N instructions (N definitions at most)
- Complexity of do-while
  - $O(N^4)$

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\cup_{p \text{ a predecessor of } i}$  OUT[p];
    OUT[i] = GEN[i]  $\cup$  (IN[i] - KILL[i]);
  }
} while (changes to any OUT occur);
```

# Complexity

- N instructions (N definitions at most)
- Complexity of do-while
  - Worst case:  $O(N^4)$

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] =  $\cup_{p \text{ a predecessor of } i}$  OUT[p];
    OUT[i] = GEN[i]  $\cup$  (IN[i] – KILL[i]);
  }
} while (changes to any OUT occur);
```

- Typical case: 2 to 3 invocations with good ordering, work-list, basic-block, and sparse sets
  - Between N and  $N^2$

N=500

Worst case: 62,500,000,000

Optimized average case:

500 – 250,000

Always have faith in your ability

Success will come your way eventually

**Best of luck!**