PROGRAMMING LANGUAGES LABORATORY

Universidade Federal de Minas Gerais - Department of Computer Science

LATTICES

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from the "Dragon Book", and from Michael Schwartzbach's "Lecture Notes in Static Analysis".

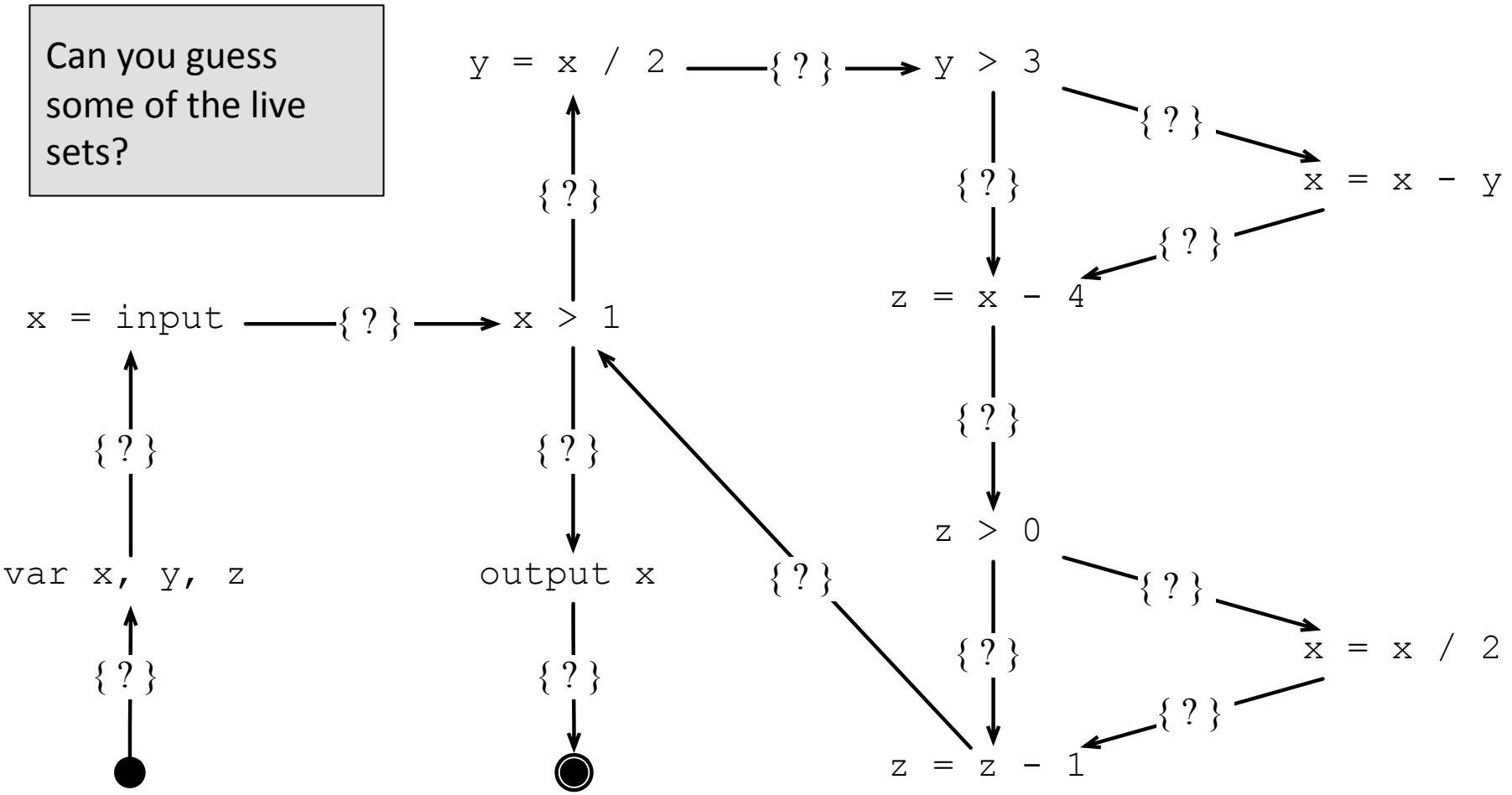
The Dataflow Framework

- We use an iterative algorithm to find the solution of a given dataflow problem.
  - How do we know that this algorithm terminates?
  - How precise is the solution produced by this algorithm?
- The purpose of this class is to provide a proper notation to deal with these two questions.
- To achieve this goal, we shall look into algebraic bodies called *lattices*.
  - But before, we shall provide some intuition on why the dataflow algorithms are correct.

My being a teacher had a decisive influence on making language and systems as simple as possible so that in my teaching, I could concentrate on the essential issues of programming rather than on details of language and notation. (N. Wirth)

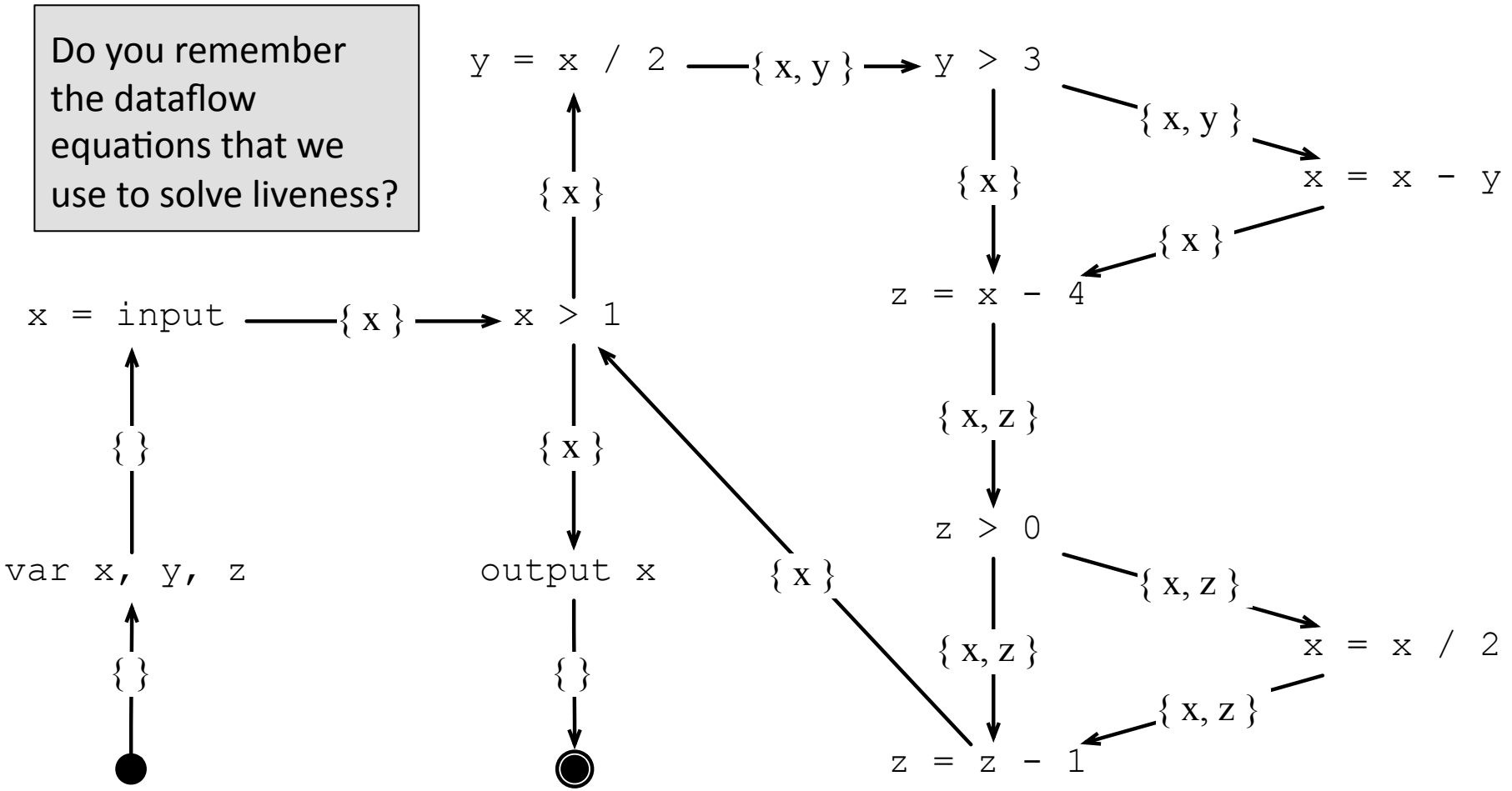
How do we know that liveness terminates?

- Given a control flow graph G, find, for each edge E in G, the set of variables alive at E.



Example of Liveness Problem

- Let's solve liveness analysis for the program below:



Dataflow Equations for Liveness

$$p : v = E$$

$$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$$

$$OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$$

- *IN(p)* = the set of variables alive immediately before p
- *OUT(p)* = the set of variables alive immediately after p
- *vars(E)* = the variables that appear in the expression E
- *succ(p)* = the set of control flow nodes that are successors of p

- The algorithm that solves liveness keeps iterating the application of these equations, until we reach a fixed point.
  - If *f* is a function, then *p* is a fixed point of f if *f(p) = p*.

Why every IN(p) and OUT(p) sets always reach a fixed point?

Dataflow Equations for Liveness

$$p : v = E$$

$$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$$

$$OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$$

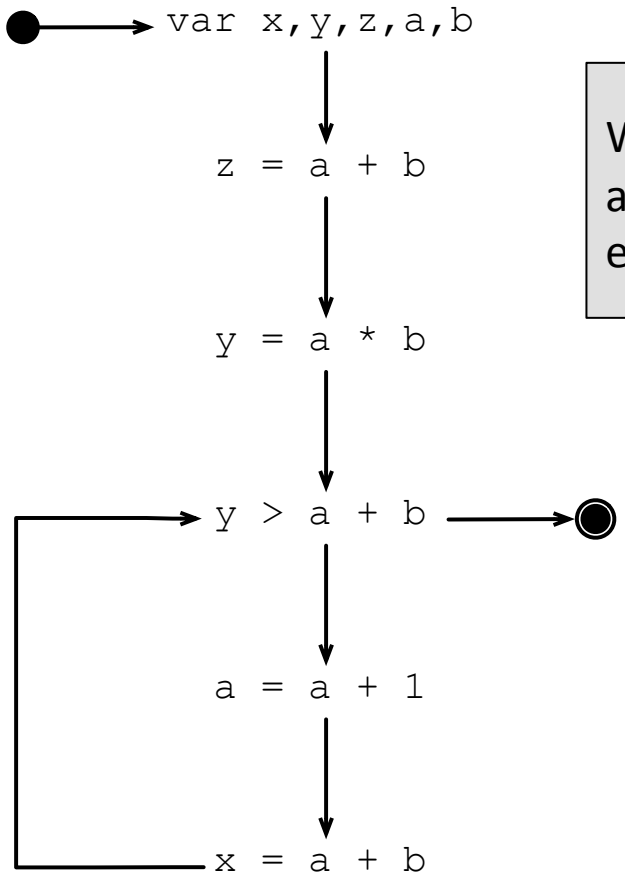
1. The key observation is that none of these equations take information away of its result.
    1. The result that they produce is always the same, or larger than the previous result.
2. Given this property, we eventually reach a fixed point, because the INs and OUTs sets cannot grow forever.

Can you explain why 1.1 is true?

And property 2, why these sets cannot grow forever?

Correctness of Available Expressions

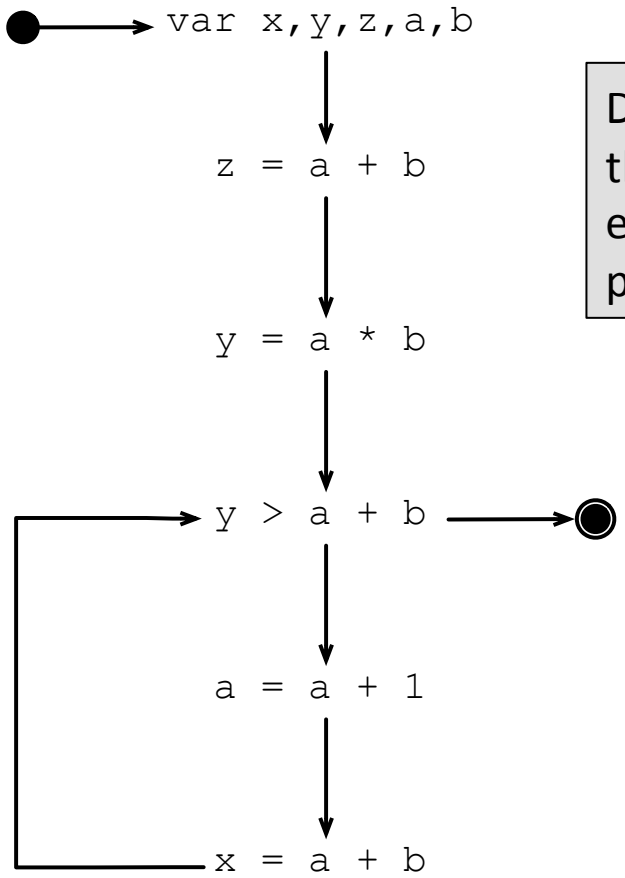
- An expression is available at a program point if its current value has already been computed earlier in the execution.



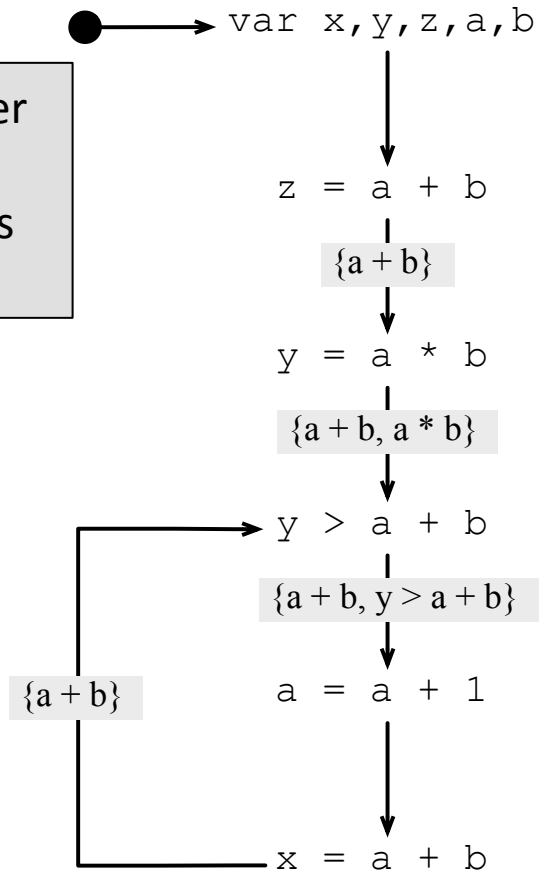
Which expressions are available in this example?

Correctness of Available Expressions

- An expression is available at a program point if its current value has already been computed earlier in the execution.



Do you remember the data-flow equations for this problem?



Dataflow Equations for Availability

$$p : v = E$$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$

- *IN(p)* = the set of expressions available immediately before p
- *OUT(p)* = the set of expressions available immediately after p
- *pred(p)* = the set of control flow nodes that are predecessors of p
- *Expr(v)* = the expressions that use variable v

- The argument on why availability analysis terminates is similar to that used in liveness, but it goes on the opposite direction.
  - In liveness, we assume that every IN and OUT set is empty, and keep adding information to them.
  - In availability, we assume that every IN and OUT set contains every expression in the program, and then we remove some of these expressions.

Dataflow Equations for Availability

$$p : v = E$$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$

1) Can you show that each IN and OUT set always decreases?

2) And why can't these sets decrease forever?

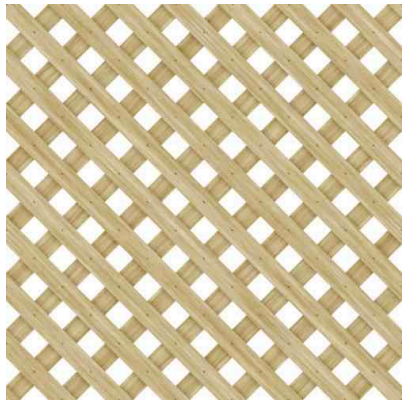
3) In the worst case, how long would this system take to stabilize?

Lattices

- All our dataflow analyses map program points to elements of algebraic bodies called *lattices*.
- A *complete lattice* L = (S, ≤, ⋁, ⋀, ⊥, T) is formed by:
  - A set S
  - A partial order ≤ between elements of S.
  - A least element ⊥
  - A greatest element T
  - A join operator ⋁
  - A meet operator ⋀

What is a partial order?

Lattice: structure consisting of strips of wood or metal crossed and fastened together with square or diamond-shaped spaces left between, used typically as a screen or fence or as a support for climbing



Can you imagine why we use this name?

Least Upper Bounds and Joins

This operator is called "Join"

- If $L = (S, \leq, \bigvee, \bigwedge, \bot, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then we let $e_{lub} = (e_2 \bigvee e_1) \in S$ be the least upper bound of the set $\{e_1, e_2\}$. The least upper bound e_{lub} has the following properties:
  - $e_1 \leq e_{lub}$ and $e_2 \leq e_{lub}$
  - For any element $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e_{lub} \leq e'$
- Similarly, we can define the least upper bound of a subset S' of S as the pairwise least upper bound of every element of S'

In order of L to be a lattice, it is a necessary condition that every subset S' of S has a least upper bound $e_{lub} \in S$. Notice that e_{lub} may not be in S'.

Greatest Lower Bounds and Meets

This operator is called "Meet"

- If $L = (S, \leq, \bigvee, \bigwedge, \bot, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then we let $e_{glb} = (e_2 \bigwedge e_1)$ be the greatest lower bound of the set $\{e_1, e_2\}$. The greatest lower bound e_{glb} has the following properties:
  - $e_{glb} \leq e_1$ and $e_{glb} \leq e_2$
  - For any element $e' \in S$, if $e' \leq e_1$ and $e' \leq e_2$, then $e' \leq e_{glb}$
- Similarly, we can define the greatest lower bound of a subset S' of S as the pairwise greatest lower bound of every element of S'

In order of L to be a lattice, it is a necessary condition that every subset S' of S has a greatest lower bound $e_{glb} \in S$. Notice that e_{glb} may not be in S'.

Properties of Join and Meet

- Join is idempotent: $x \vee x = x$
- Join is commutative: $y \vee x = x \vee y$
- Join is associative: $x \vee (y \vee z) = (x \vee y) \vee z$
- Join has a multiplicative one: for all x in S, $(\bot \vee x) = x$
- Join has a multiplicative zero: for all x in S, $(\top \vee x) = \top$

- Meet is idempotent: $x \wedge x = x$
- Meet is commutative: $y \wedge x = x \wedge y$
- Meet is associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- Meet has a multiplicative one: for all x in S, $(\top \wedge x) = x$
- Meet has a multiplicative zero: for all x in S, $(\bot \wedge x) = \bot$

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Joins, Meets and Semilattices

- Some dataflow analyses, such as live variables, use the join operator. Others, such as available expressions, use the meet operator.
  - In the case of liveness, this operator is set union.
  - In the case of availability, this operator is set intersection.
- If a lattice is well-defined for only one of these operators, we call it a *semilattice*.
- The vast majority of the dataflow analyses require only the semilattice structure to work.

Partial Orders

- The meet operator (or the join operator) of a semilattice defines a partial order:

$$x \leq y \text{ if, and only if } x \wedge y = x^{\clubsuit}$$

- This partial order has the following properties:
  - $x \leq x$ (The partial order is *reflexive*)
  - If $x \leq y$ and $y \leq x$, then $x = y$ (The partial order is *antisymmetric*)
  - If $x \leq y$ and $y \leq z$, then $x \leq z$ (The partial order is *transitive*)

$^{\clubsuit}$: For a proof that this equivalence is true, see the "Dragon Book".

The Semilattice of Liveness Analysis

```
var x,y,z;

x = input;

while (x > 1) {

    y = x / 2;

    if (y > 3)

        x = x - y;

    z = x - 4;

    if (z > 0)

        x = x / 2;

    z = z - 1;

}

output x;
```

- Given the program on the left, we have the semillatice L = ($P^{\{x, y, z\}}$, $\subseteq$, $\cup$, {}, {a, b, c})
- As this is a semilattice, we have only defined the join operator, which, in our case, is set union.
- The least element is the empty set, which is contained inside every other subset of {x, y, z}
- The greatest element is {x, y, z} itself, which contains every other subset.

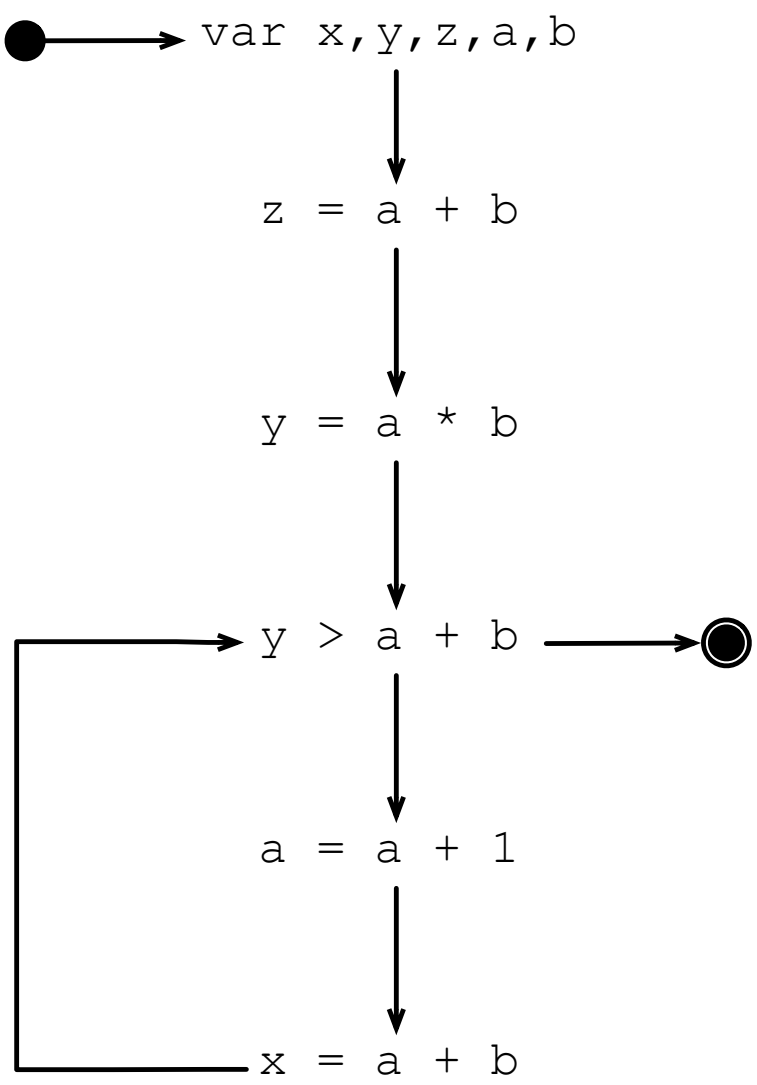
Checking the Properties of a Semilattice

- Is $L = (P^S, \subseteq, \cup, \{\}, S)$ really a semilattice♤?
  1. Is U a join operator? In other words, is it the case that "$x \leq y$ if, and only if $x \vee y = y$" for $\subseteq$ and U?
     - We want to know if "$x \subseteq y$ if, and only if, $x \cup y = y$"
  2. Is {} a "Least Element"?
     - The empty set is contained within any other set; hence, for any S' in P^S, we have that $\{\} \subseteq S'$
  3. Is S a "Greatest Element"?
     1. Any set in P^S is contained in S; thus, for any S' in Ps, we have that $S' \subseteq S$

Can you use similar reasoning to show that $(P^S, \supseteq, \cap, S, \{\})$ is also a lattice?

♤: S is any finite set of elements, e.g., S = {x, y, z} in our previous example.

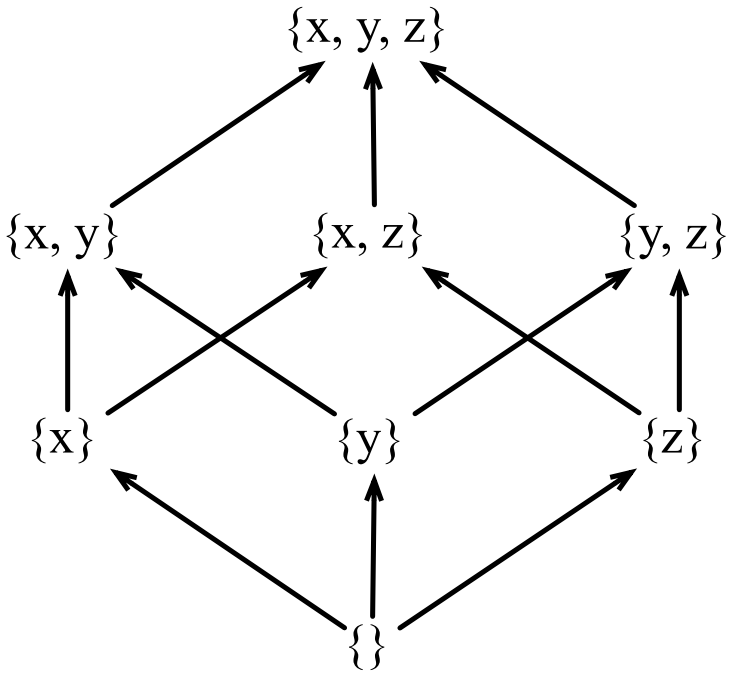
The Semillatice of Available Expressions

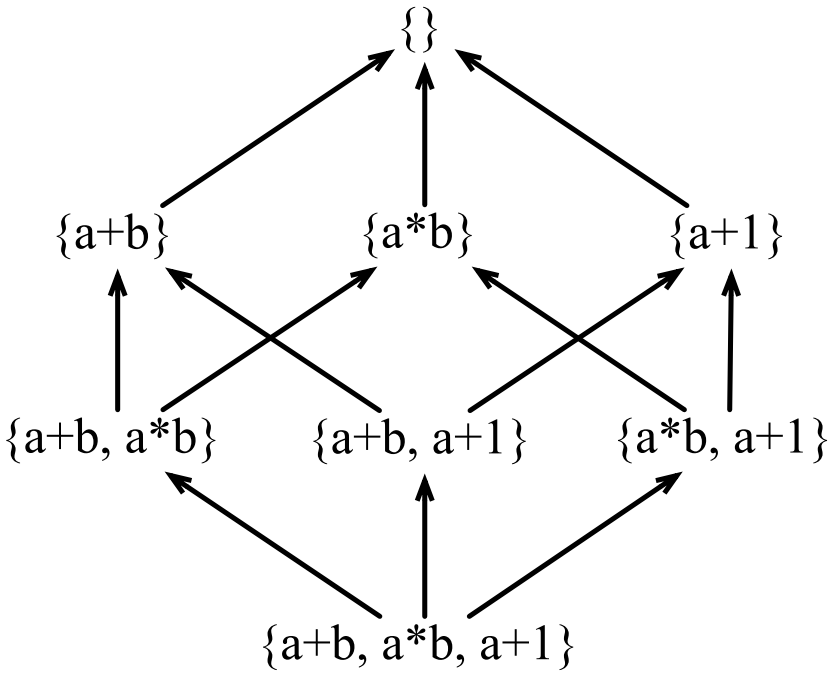


- Given the program on the left, we have the semillatice L = ($P^{\{a+b,\ a*b,\ a+1\}}$, $\supseteq$, $\cap$, {a+b, a*b, a+1}, {})
- As this is a semilattice, we have only defined the meet operator, which, in our case, is set intersection.
- The least element is the set of all the expressions (remember, we are dealing with intersection).
- The greatest element is the empty set, as it is the intersection of every subset.

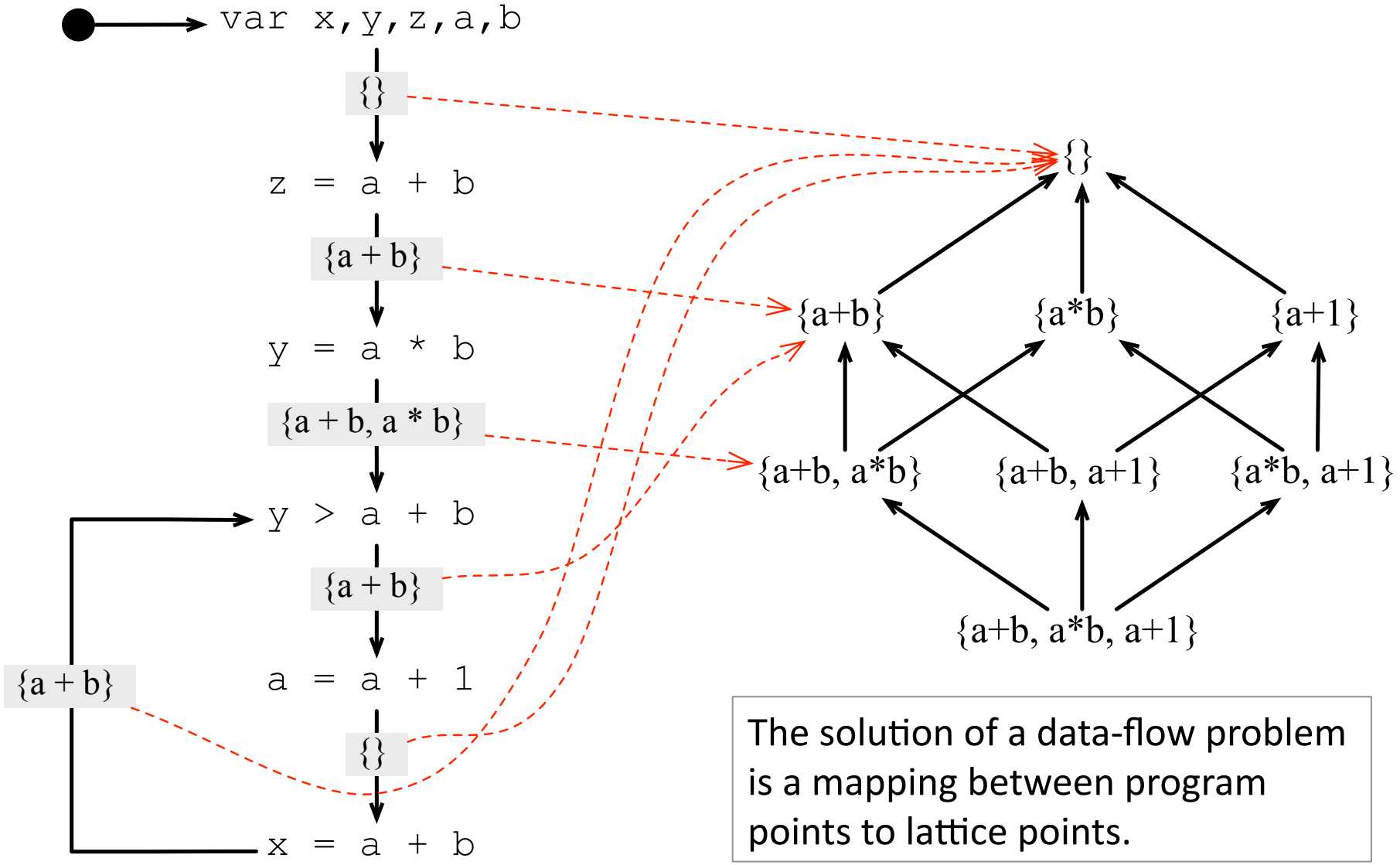
Lattice Diagrams

- We can represent the partial order between the elements of a lattice as a *Hasse Diagram*.
  - If an element e_1 is above another element e_2, then we say that $e_1 \leq e_2$.



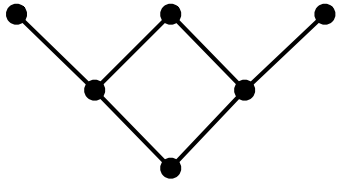


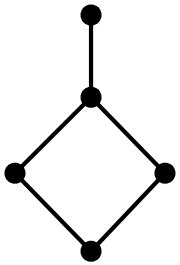
Mapping Program Points to Lattice Points

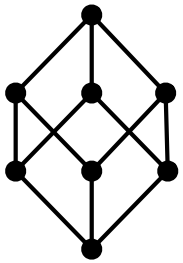


The solution of a data-flow problem is a mapping between program points to lattice points.

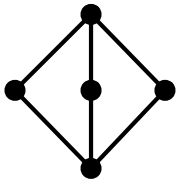
Lattice Diagrams

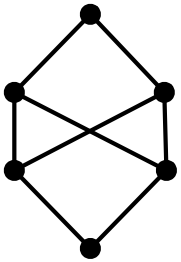


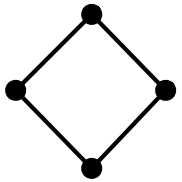




Which diagrams do not represent lattices?







Transfer Functions

- As we have seen in a previous class, the solution of a dataflow problem is given by a set of equations, such as:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

- Each function F_i is a transfer function.
- If the dataflow problem ranges over a lattice *L*, then each F_i is a function of type $L \rightarrow L$.

Which property must be true about each function F_i and the lattice *L*, so that a worklist algorithm that solves this system terminates?

Monotone Transfer Functions

- We can guarantee that the worklist algorithm that solves the constraint system terminates if these two conditions are valid:
  - Each transfer function F_i always produces larger results, i.e., given l_1 and l_2 in the lattice L, we have that if $l_1 \leq l_2$, then $F_1(l_1) \leq F_1(l_2)$
    - Functions that meet this property are called monotone.
  - The semilattice L must have finite height
    - It is not possible to have an infinite chain of elements in L, e.g., $l_0 \leq l_1 \leq l_2 \leq \ldots$ such that every element in this chain is different.
    - Notice that the lattice can still have an infinite number of elements.

Can you give an example of a lattice that has an infinite number of elements, and still has finite height?

Monotone Transfer Functions

- Let $L = (S, \leq, \bigvee, \bigwedge, \bot, T)$, and consider a family of functions F, such that for all $f \in F$, $f : L \rightarrow L$. These properties are equivalent*:
  - Any element $f \in F$ is monotonic
  - For all $x \in S$, $y \in S$, and $f \in F$, $x \leq y$ implies $f(x) \leq f(y)$
  - For all x and y in S and f in F, $f(x \bigwedge y) \leq f(x) \bigwedge f(y)$
  - For all x and y in S and f in F, $f(x) \bigvee f(y) \leq f(x \bigvee y)$

* for a proof, see the "Dragon Book", 2nd edition, Section 9.3

Monotone Transfer Functions

- Prove that the transfer functions used in the liveness analysis problem are monotonic:

$$p : v = E$$

$$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$$

$$OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$$

- Now do the same for available expressions: prove that these transfer functions are monotonic:

$$p : v = E$$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$

Correctness of the Iterative Solver

- We shall show the correctness of a algorithm that solves a forward-must analysis, i.e., an analysis that propagates information from IN sets to OUT sets, and that uses the semilattice with the meet operator. The solver is given on the right:

OUT[ENTRY] = v_{ENTRY}

for each block B, but ENTRY

  OUT[B] = T

while (any OUT set changes)

  for each block B, but ENTRY

    IN[B] = $\bigwedge_{p \in pred(B)} OUT[P]$

    OUT[B] = $f_b(IN[B])$

1) Does this algorithm solve a forward or a backward analysis?

2) How can we show that the values taken by the IN and OUT sets can only decrease?

Induction

- Usually we prove properties of algorithms using induction.
  - In the base case, we must show that after the first iteration the values of each IN and OUT set is no greater than the initial values.

1) Why is this statement trivial to show?

  - Induction step: assume that after the k^{th} iteration, the values are all no greater than those after the $(k-1)^{st}$ iteration.

2) You need now to extend this property to iteration k + 1

Induction

- Continuing with the induction step:

1. Let $IN[B]^i$ and $OUT[B]^i$ denote the values of IN[B] and OUT[B] after iteration i

2. From the hypothesis, we know that $OUT[B]^k \leq OUT[B]^{k-1}$

3. But (2) gives us that $IN[B]^{k+1} \leq IN[B]^k$, by the definition of the meet operator, and line 6 of our algorithm

4. From (3), plus the fact that f_b is monotonic, plus line 7 of our algorithm, we know that $OUT[B]^{k+1} \leq OUT[B]^k$

The Asymptotic Complexity of the Solver

If we assume a lattice of height H, and a program with B blocks, what is the asymptotic complexity of the iterative solver on the right?

1: $OUT[ENTRY] = v_{ENTRY}$

2: for each block B, but ENTRY

3: $\quad OUT[B] = T$

4: while (any OUT set changes)

5: $\quad$ for each block B, but ENTRY

6: $\quad\quad IN[B] = \bigwedge_{p \in pred(B)} OUT[P]$

7: $\quad\quad OUT[B] = f_b(IN[B])$

The Asymptotic Complexity of the Solver

If we assume a lattice of height H, and a program with B blocks, what is the asymptotic complexity of the iterative solver on the right?

1. The IN/OUT set associated with a block can change at most H times; hence, the loop at line 4 iterates H*B times
2. The loop at line 5 iterates B times
3. Each application of the meet operator, at line 6, can be done in O(H)
4. A block can have B predecessors; thus, line 6 is O(H*B)
5. By combining (1), (2) and (4), we have $O(H*B*B*H*B) = O(H^2*B^3)$

1: $OUT[ENTRY] = v_{ENTRY}$

2: for each block B, but ENTRY

3: OUT[B] = T

4: while (any OUT set changes)

5: for each block B, but ENTRY

6: $IN[B] = \bigwedge_{p \in pred(B)} OUT[P]$

7: $OUT[B] = f_b(IN[B])$

This is a pretty high complexity, yet most real-world implementations of dataflow analyses tend to be linear on the program size, in practice.

Fixed Point

- The fixed point of a function f:L→L is an element $\ell \in L$, such that $f(\ell) = \ell$
- A function can have more than one fixed point.
  - For instance, if our function is the identity, e.g., f(x) = x, then any element in the domain of f is a fixed point of it.

1. Our constraint system on $F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$ stabilizes in a fixed point of F.
2. If L is a lattice, and f is well-defined for every element e ∈ L, than we have a **maximum fixed point**.
3. Similarly, If L is a lattice, and f is well-defined for every element e ∈ L, than we have a **minimum fixed point**.

For each of these three bullets on the right, can you provide a rational on why it is true?

In other words, the glb and lub of the fixed points are fixed points themselves.

Maximum Fixed Point (MFP)

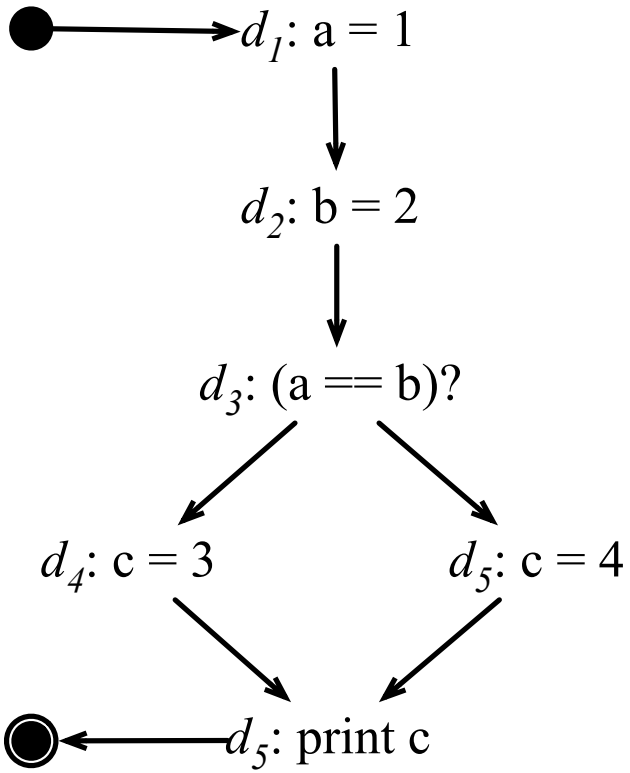
- The solution that we find with the iterative solver is the *maximum fixed point* of the constraint system, assuming monotone transfer functions, and a semilattice with meet operator of finite height♧.
- The MFP is a solution with the following property:
  - any other valid solution of the constraint system is less than the MFP solution.
- Thus, our iterative solver is quite precise, i.e., it produces a solution to the constraint system that wraps up each equation in the constraint system very tightly.
  - Yet, the MFP solution is still very conservative.

Can you give an example of a situation in which we get a conservative solution?

♧: For a proof, see "Principles of Program Analysis", pp 76-78.

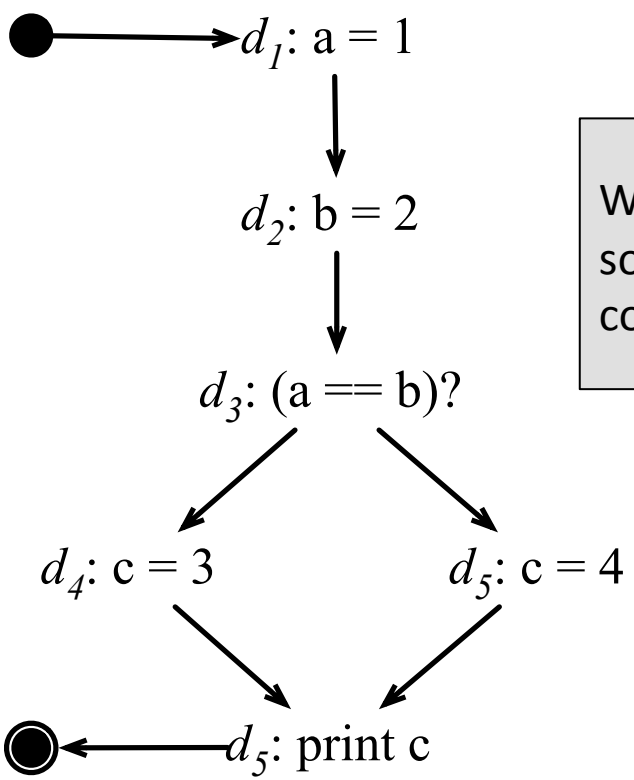
Example: Reaching Definitions

- What would be a solution that our iterative solver would produce for the reaching definitions problem in the program below?

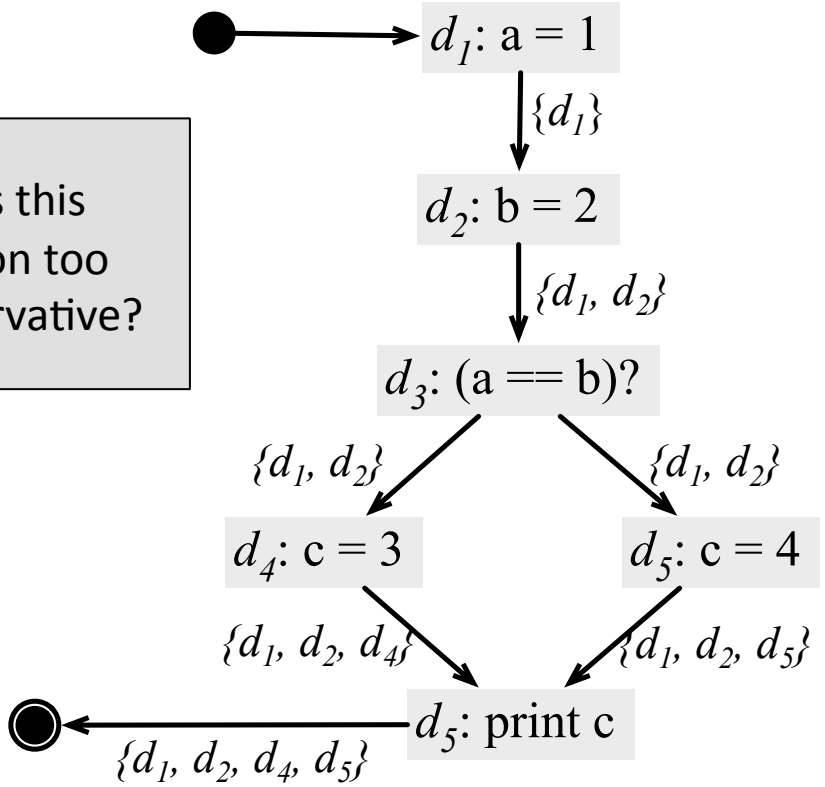


Example: Reaching Definitions

- What would be a solution that our iterative solver would produce for the reaching definitions problem in the program below?



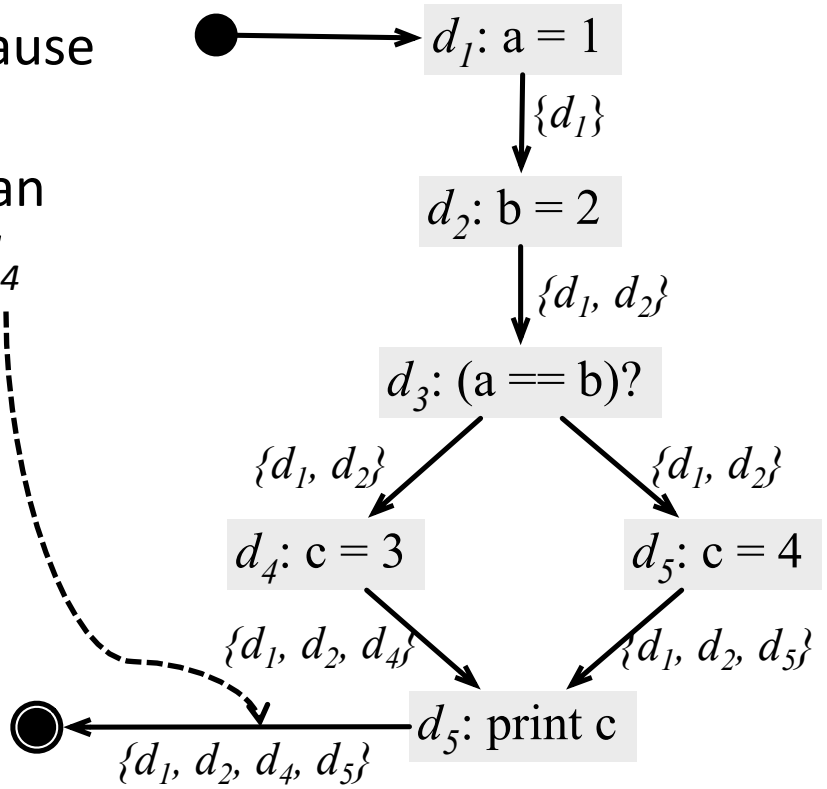
Why is this solution too conservative?



Example: Reaching Definitions

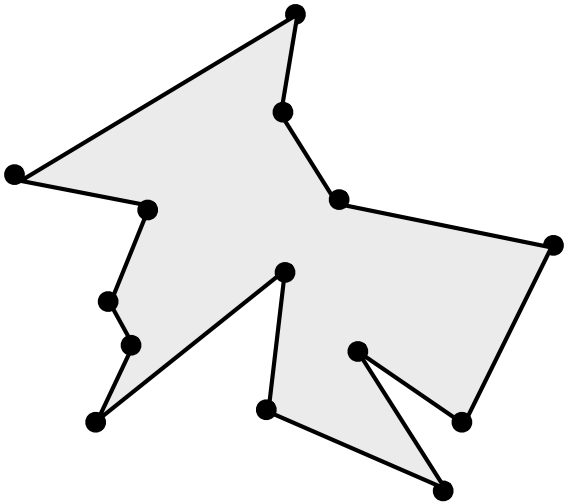
- What would be a solution that our iterative solver would produce for the reaching definitions problem in the program below?

The solution is conservative because the branch is always false.
Therefore, the statement c = 3 can never occur, and the definition d_4 will never reach the end of the program.

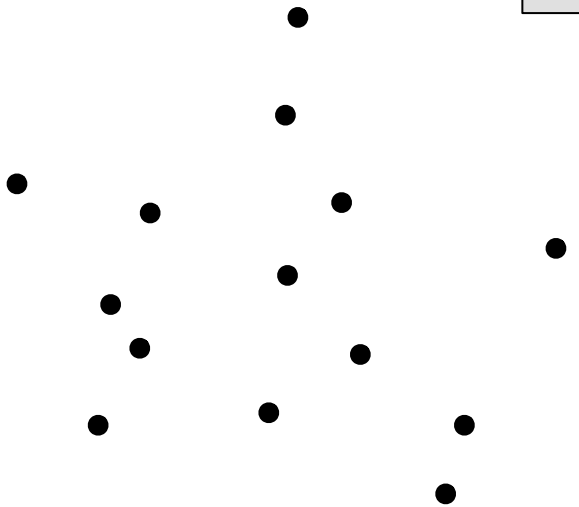


More Intuition on MFP Solutions

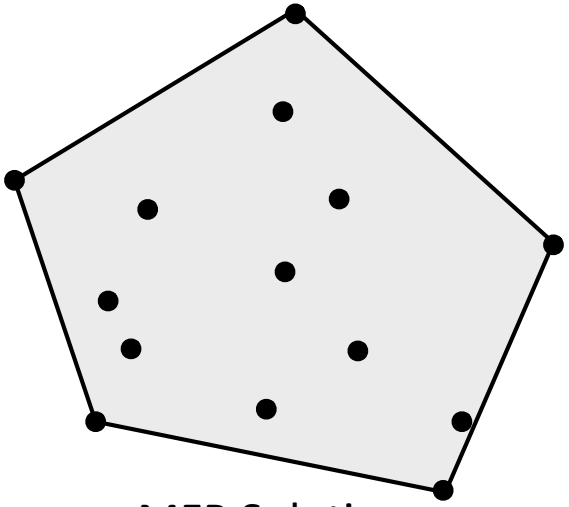
What would be a wrong solution?



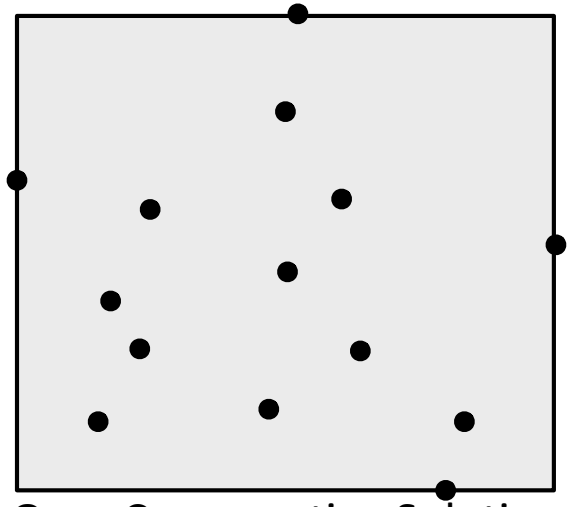
Actual Program Behavior



Constraints

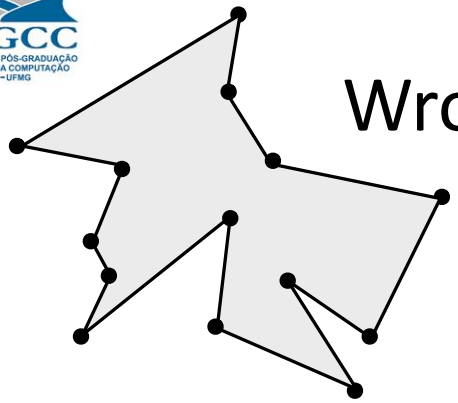


MFP Solution

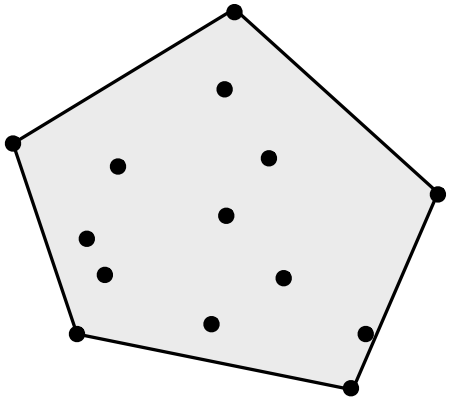


Over-Conservative Solution

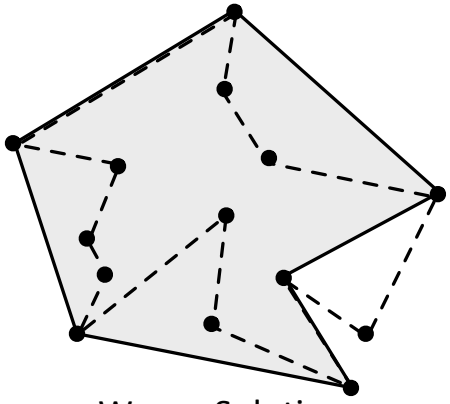
Wrong Solutions: False Negatives



Actual Program Behavior

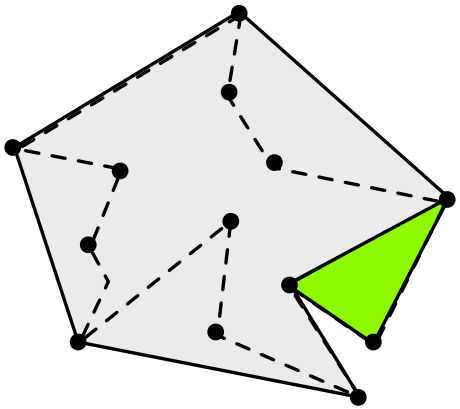


MFP Solution

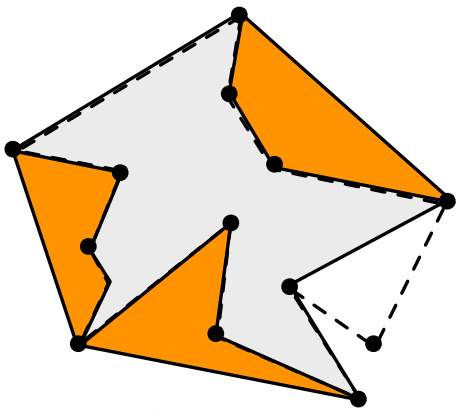


Wrong Solution

- It is ok if we say that a program does more than it really does.
  - This excess of information is usually called false positive
- The problem are the false negatives.
  - If we say that a program does not do something, and it does, we may end up pruning away correct behavior



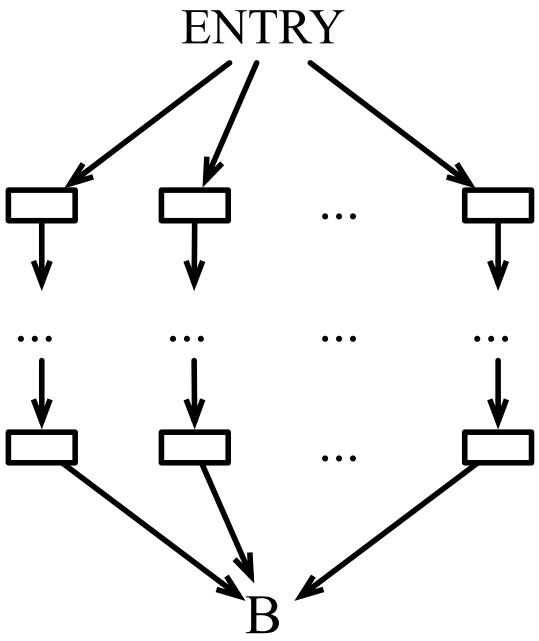
False Negative



False Positive

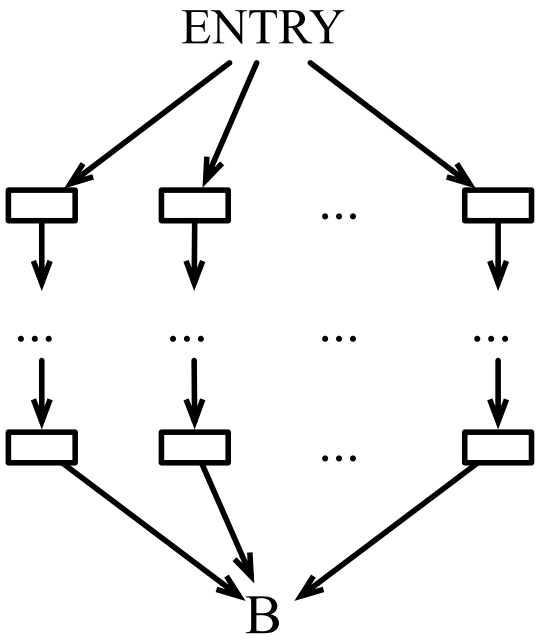
The Ideal Solution

- The MFP solution fits the constraint system tightly, but it is a conservative solution.
  - What would be an ideal solution?
  - In other words, given a block B in the program, what would be an ideal solution of the dataflow problem for the IN set of B?



The Ideal Solution

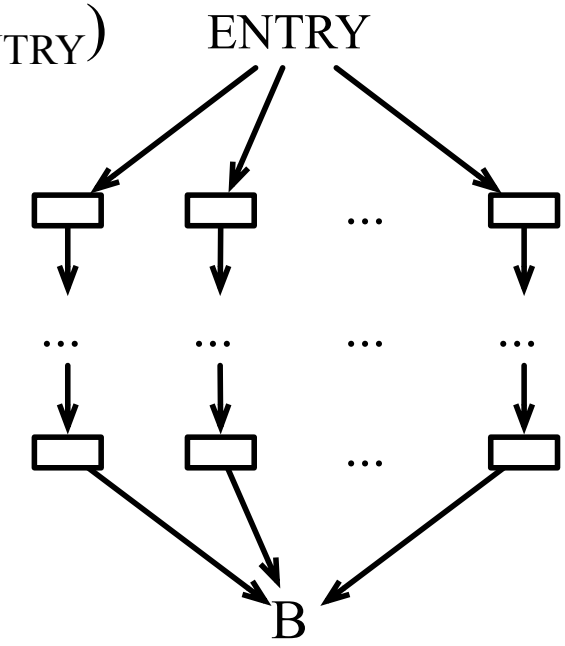
- The MFP solution fits the constraint system tightly, but it is a conservative solution.
  - What would be an ideal solution?
  - In other words, given a block B in the program, what would be an ideal solution of the dataflow problem for the IN set of B?
- The ideal solution computes dataflow information through each *possible path* from ENTRY to B, and then meets/joins this info at the IN set of B.
  - A path is possible if it is *executable*.



The Ideal Solution

- Each possible path P, e.g.: ENTRY → B_1 → … → B_K → B gives us a transfer function f_p, which is the composition of the transfer functions associated with each B_i.
- We can then define the ideal solution as:

$$\text{IDEAL[B]} = \bigwedge_{\text{p is a possible path from ENTRY to B}} f_p(v_{\text{ENTRY}})$$



- Any solution that is smaller than ideal is wrong.
- Any solution that is larger is conservative.

The Meet over all Paths Solution

- Finding the ideal solution to a given dataflow problem is undecidable in general.
  - Due to loops, we may have an infinite number of paths.
  - Some of these paths may not even terminate.
- Thus, we settle for the meet over all paths (MOP) solution:

$$\mathrm{MOP[B]} = \bigwedge_{\text{p is a path from ENTRY to B}} f_p(v_{\mathrm{ENTRY}})$$

Is MOP the solution that our iterative solver produces for a dataflow problem?

What is the difference between the ideal solution and the MOP solution?

Distributive Frameworks♥

- We say that a dataflow framework is distributive if, for all x and y in S, and every transfer function f in F we have that:

$$f(x \wedge y) = f(x) \wedge f(y)$$

- The MOP solution and the solution produced by our iterative algorithm are the same if the dataflow framework is distributive♣.
- If the dataflow framework is not distributive, but is monotone, we still have that IN[B] ≤ MOP[B] for every block B

Can you show that our four examples of data-flow analyses, e.g., liveness, availability, reaching defs and anticipability, are distributive?

♥ We are considering the meet operator. An analogous discussion applies to join operators.
♣ For a proof, see the "Dragon Book", 2nd edition, Section 9.3.4

Distributive Frameworks

Our analyses use transfer functions such as $f(\ell) = (\ell \setminus \ell_k) \cup \ell_g$. For instance, for liveness, if ℓ = *"v = E",* then we have that $IN[\ell] = OUT[\ell] \setminus \{v\} \cup vars(E)$. So, we only need a bit of algebra:

$$f(\ell \wedge \ell') = ((\ell \wedge \ell') \setminus \ell_k) \cup \ell_g \quad \textbf{(i)}$$

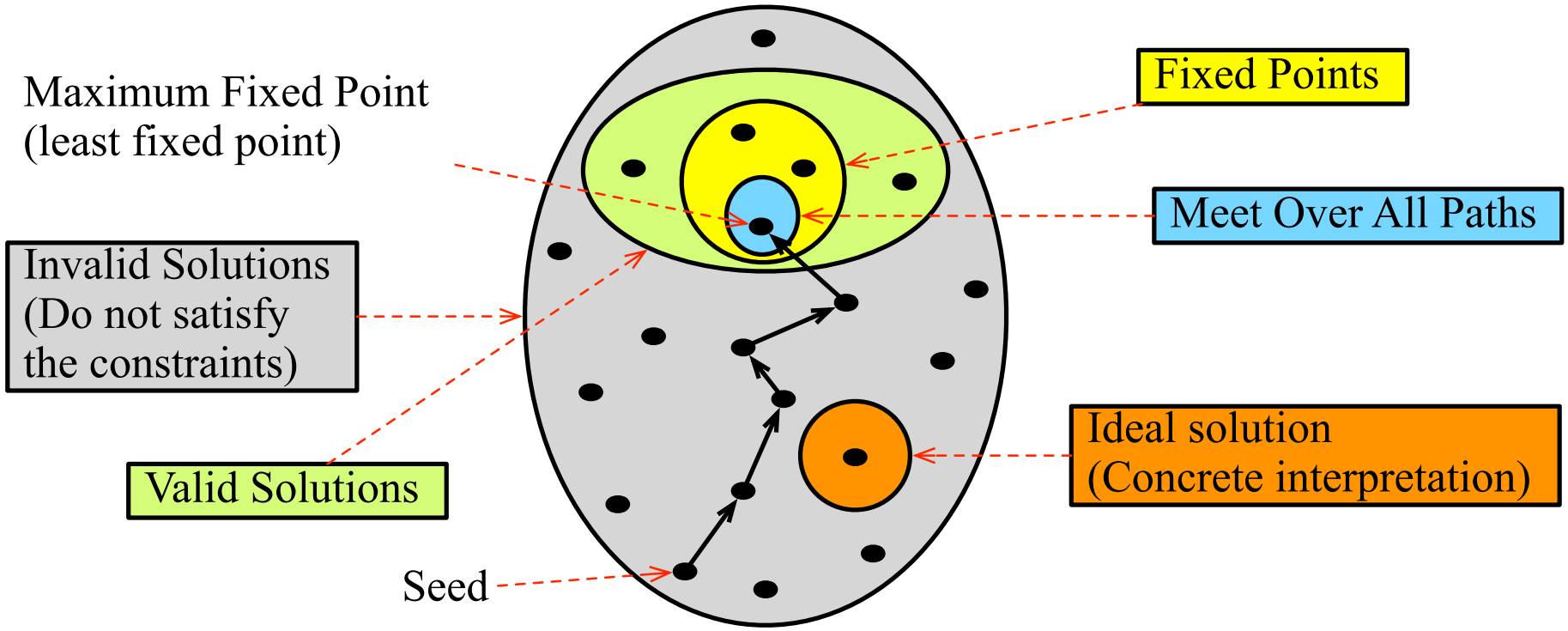
$$((\ell \setminus \ell_k \wedge \ell' \setminus \ell_k)) \cup \ell_g \quad \textbf{(ii)}$$

$$((\ell \setminus \ell_k) \cup \ell_g) \wedge ((\ell' \setminus \ell_k) \cup \ell_g) \quad \textbf{(iii)}$$

$$f(\ell) \wedge f(\ell') \quad \textbf{(iv)}$$

To see why (ii) and (iii) are true, just remember that in any of the four data-flow analyses, either $\wedge$ is $\cap$, or it is $\cup$.

Ascending Chains

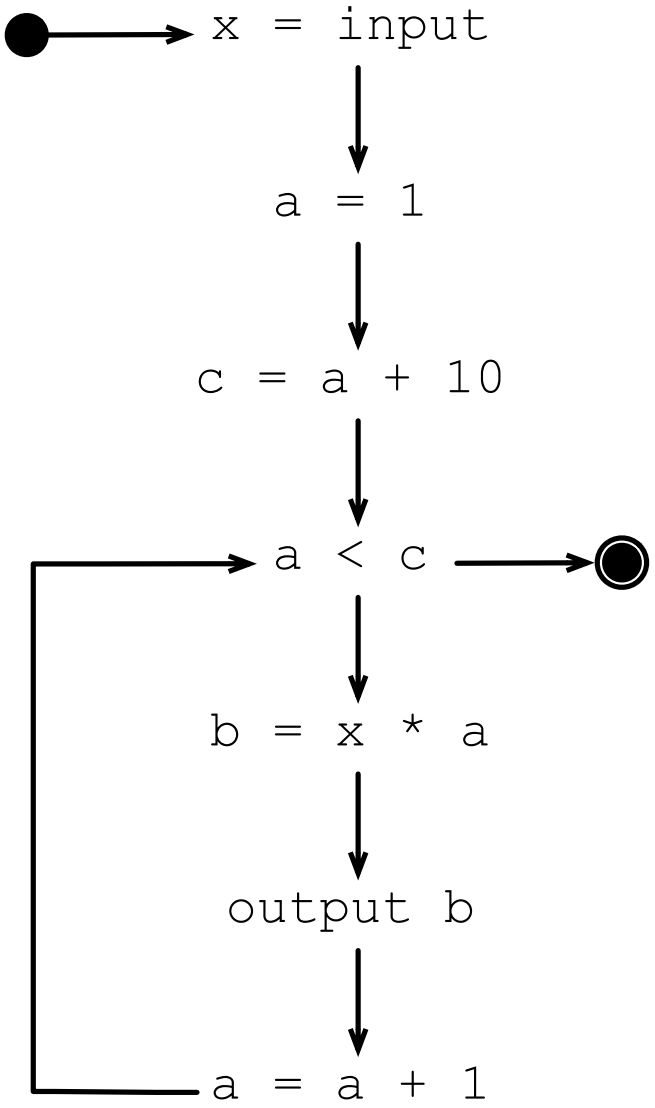


Our constraint solver finds solutions starting from a seed s, and then computing elements of the underlying lattice that are each time larger than s, until we find an element of this lattice that satisfies the constraints. We find the least among these elements, which traditionally we call the system's Maximum Fixed Point.

Map Lattices

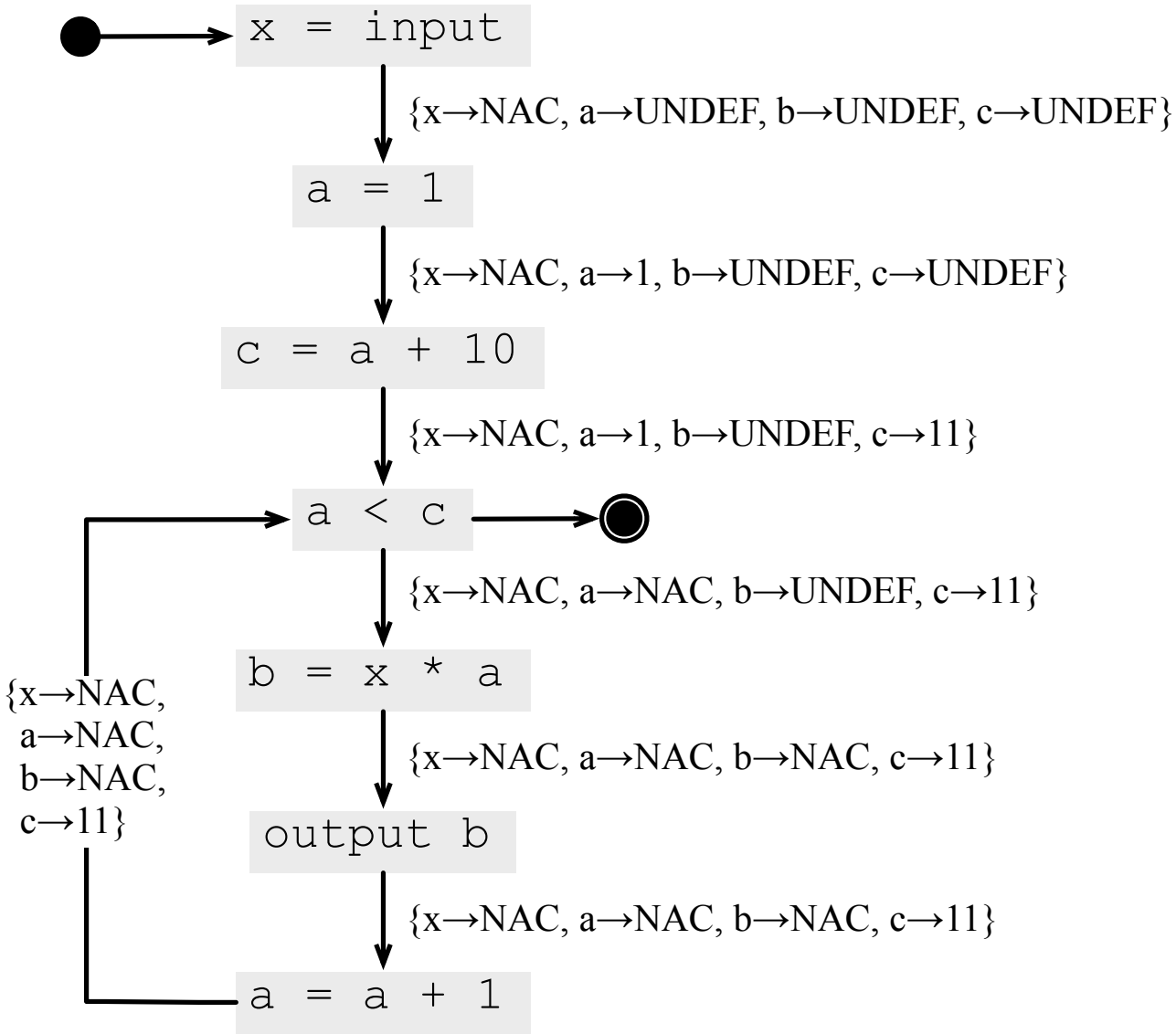
- If S is a set and $L = (T, \bigwedge)$ is a meet semilattice, then the structure $L_s = (S \to T, \bigwedge_s)$ is also a semilattice, which we call a *map semilattice*.
  - The domain is $S \to T$
  - The meet is defined by $f \bigwedge f' = \lambda x.f(x) \bigwedge f'(x)$
  - The ordering is $f \leq f' \Leftrightarrow \forall x, f(x) \leq f'(x)$
- A typical example of a map lattice is used in the constant propagation analysis.

Constant Propagation



- How could we optimize the program on the left?
- Which information would be necessary to perform this optimization?
- How can we obtain this information?

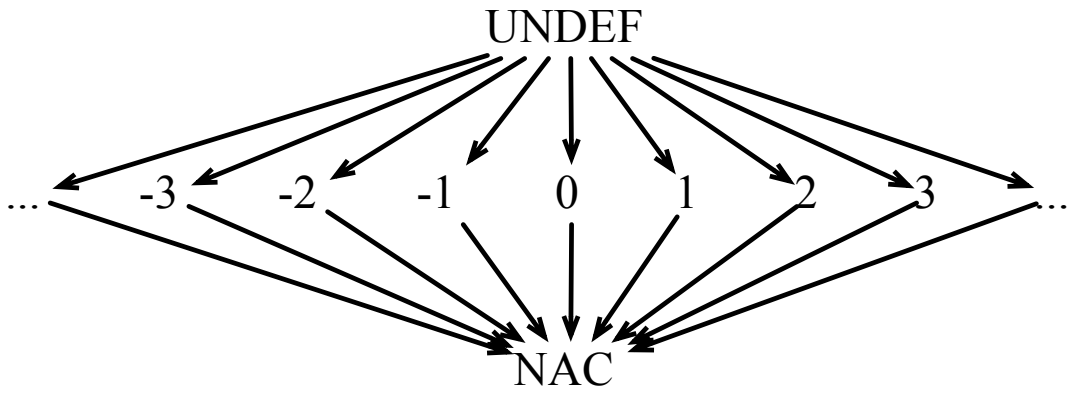
Constant Propagation



What is the lattice that we are using in this example?

Constant Propagation

- We are using a map lattice, that maps variables to an element in the lattice L below:



How are the transfer functions, assuming a meet operator?

Constant Propagation: Transfer Functions

- The transfer function depends on the statement *p* that is being evaluated. I am giving some examples, but a different transfer function must be designed for every possible instruction in our program representation:

$p : v = c$

$$OUT(p) = \lambda x.x = v \ ? \ c \ : \ IN(p)(x)$$

$p : v = u$

$$OUT(p) = \lambda x.x = v \ ? \ IN(p)(u) \ : \ IN(p)(x)$$

$p : v = t + u$

$$OUT(p) = \lambda x.x = v \ ? \ IN(p)(t) + IN(p)(u) \ : \ IN(p)(x), if \ IN(p)(t), IN(p)(u) \ consts$$
$$OUT(p) = \lambda x.x = v \ ? \ NAC \ : \ IN(p)(x), if \ IN(p)(t) \ or \ IN(p)(u) \ not \ const$$
$$OUT(p) = \lambda x.x = v \ ? \ UNDEF \ : \ IN(p)(x), otherwise$$

$- - -$

$$IN(p) = \bigwedge OUT(p_s), p_s \in pred(p)$$

How is the meet operator, e.g., $\bigwedge$, defined?

The meet operator

$\bigwedge$	UNDEF	c_1	NAC
UNDEF	UNDEF	c_1	NAC
c_2	c_2	$c_1 \bigwedge c_2$	NAC
NAC	NAC	NAC	NAC

If we have two constants, e.g., c_1 and c_2, then we have that $c_1 \bigwedge c_2 = c_1$ if $c_1 = c_2$, and NAC otherwise

$p : v = c$

$$OUT(p) = \lambda x.x = v \ ? \ c \ : \ IN(p)(x)$$

$p : v = u$

$$OUT(p) = \lambda x.x = v \ ? \ IN(p)(u) \ : \ IN(p)(x)$$

$p : v = t + u$

$$OUT(p) = \lambda x.x = v \ ? \ IN(p)(t) + IN(p)(u) \ : \ IN(p)(x), if \ IN(p)(t), IN(p)(u) \ consts$$
$$OUT(p) = \lambda x.x = v \ ? \ NAC \ : \ IN(p)(x), if \ IN(p)(t) \ or \ IN(p)(u) \ not \ const$$
$$OUT(p) = \lambda x.x = v \ ? \ UNDEF \ : \ IN(p)(x), otherwise$$

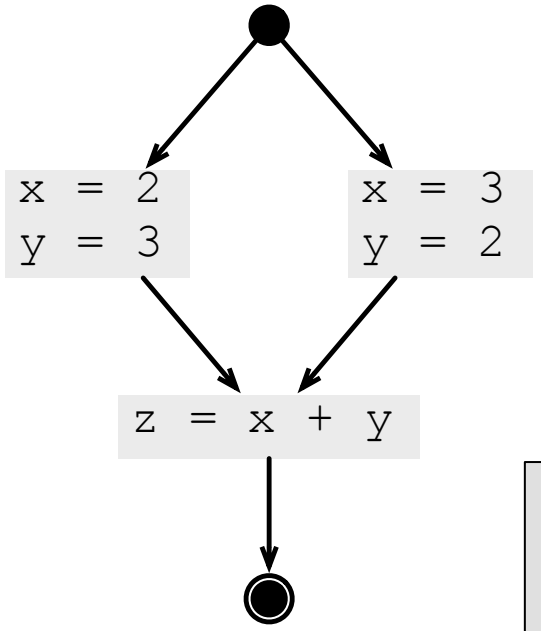
– – –

$$IN(p) = \bigwedge OUT(p_s), p_s \in pred(p)$$

Are these functions monotone?

Non-Distributivity

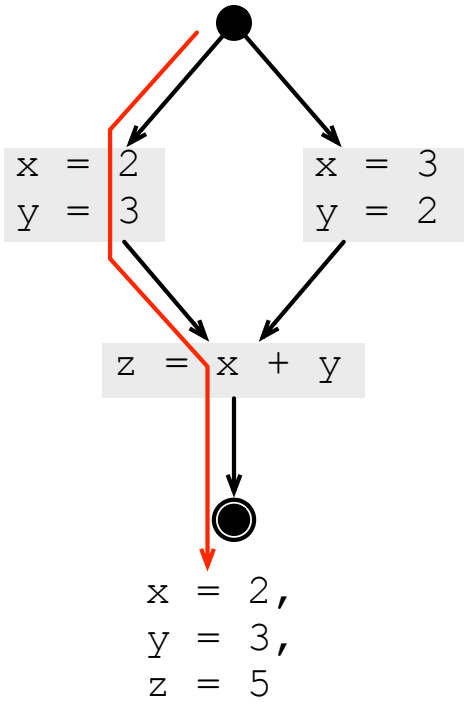
- The constant propagation framework is monotone, but it is not distributive.
  - As a consequence, the MOP solution is more precise than the iterative solution.
- Consider this program:

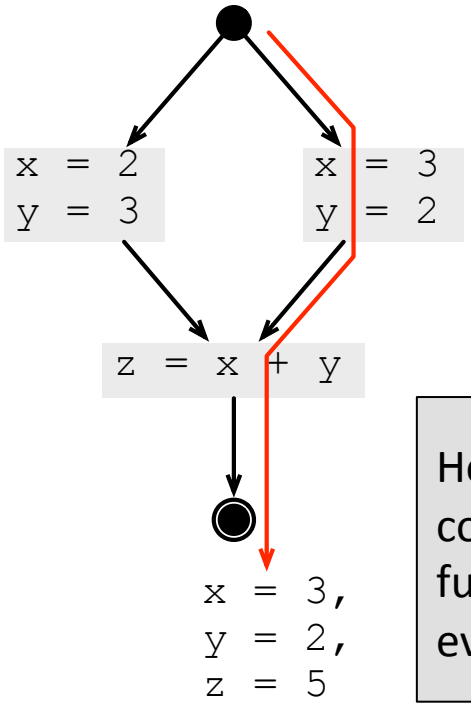


Is variable z a constant in this program?

Non-Distributivity

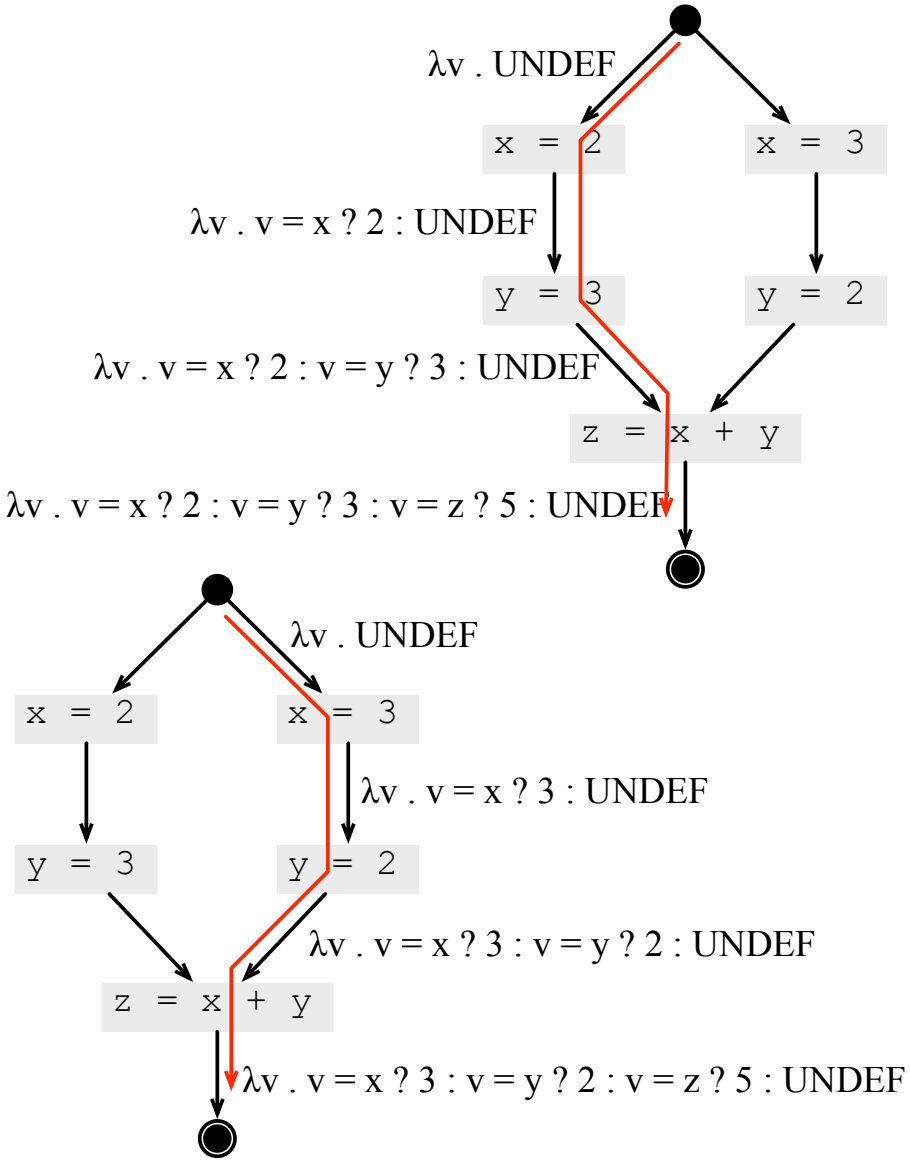
- This program has only two paths, and throughout any of them we find that z is a constant.
  - Indeed, variables are constants along any path in this program.





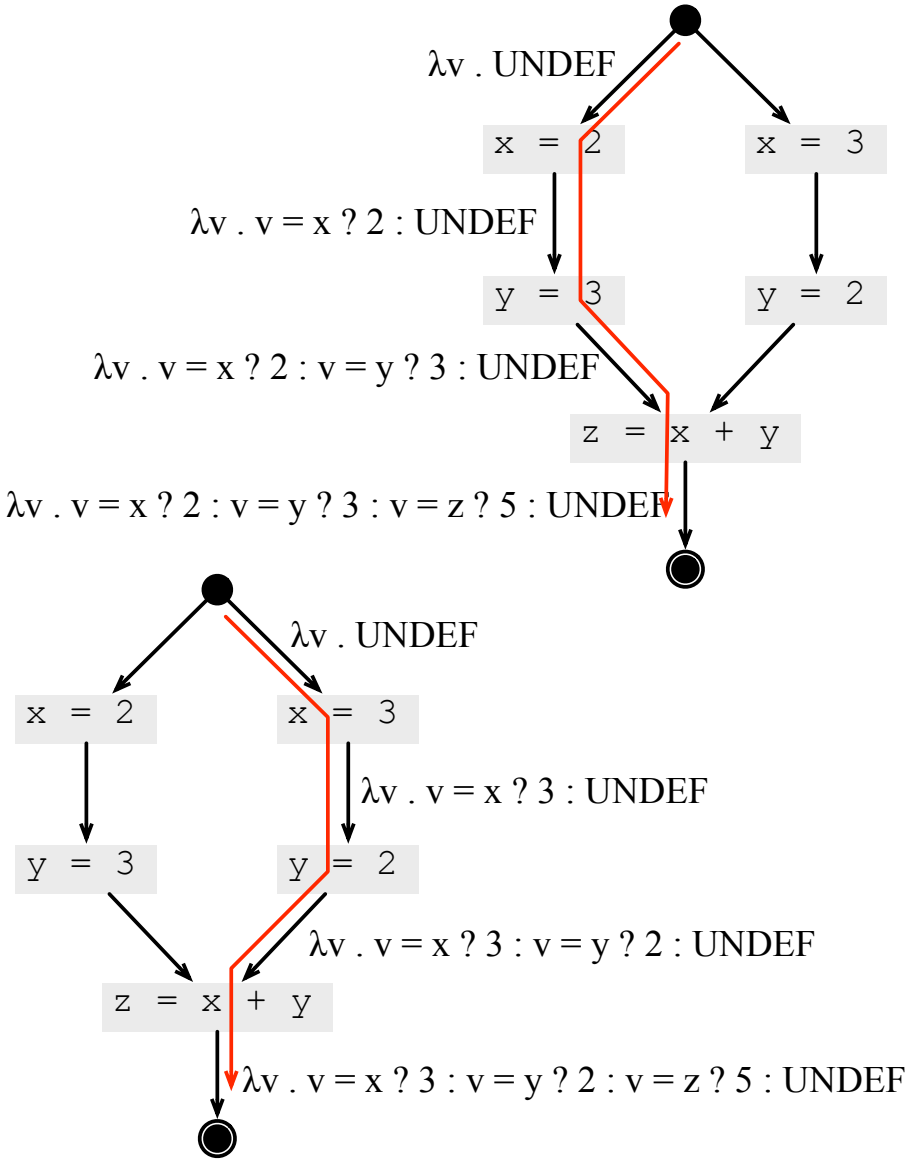
How is the composition of functions along every path?

Non-Distributivity



What is the meet of the functions
λv . v = x ? 3 : v = y ? 2 : v = z ? 5 : U
and
λv . v = x ? 2 : v = y ? 3 : v = z ? 5 : U?

Non-Distributivity



The meet of the functions

λv . v = x ? 3 : v = y ? 2 : v = z ? 5 : U

and

λv . v = x ? 2 : v = y ? 3 : v = z ? 5 : U

is the function:

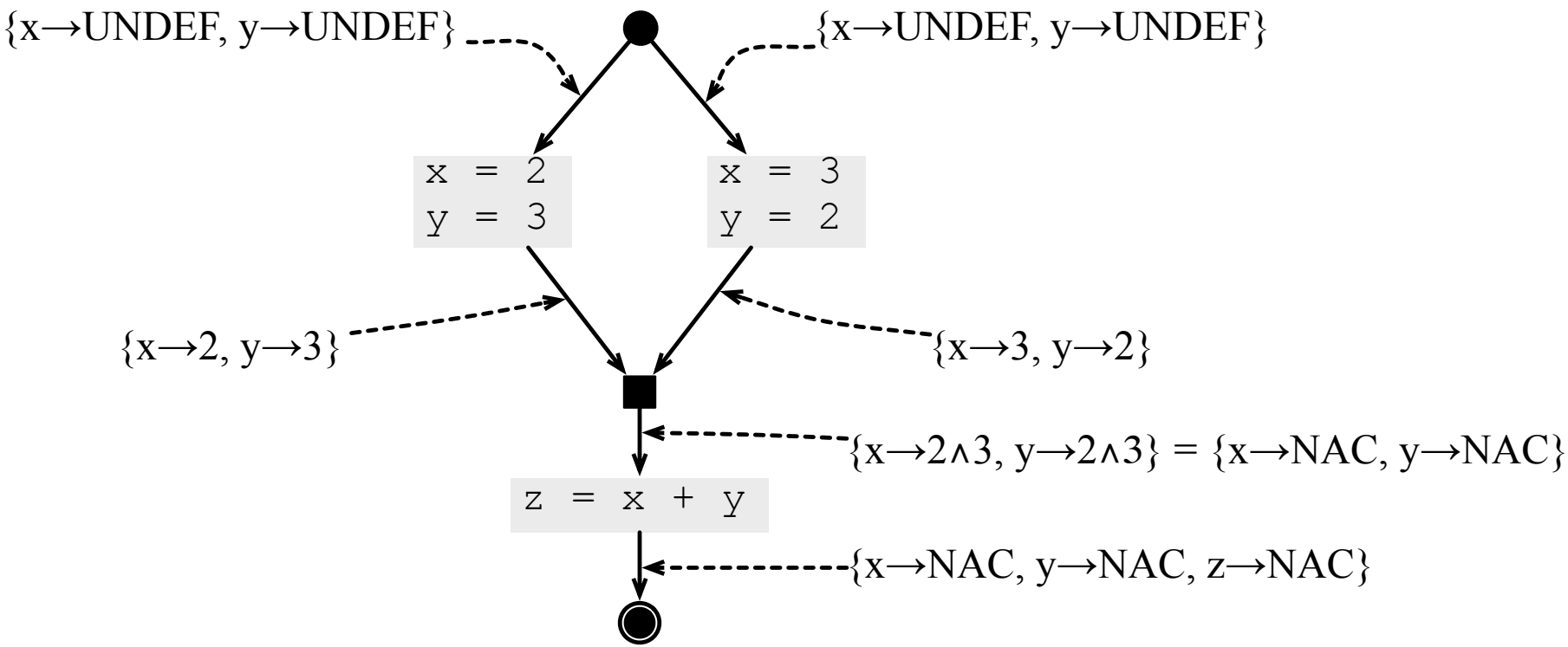
λv . v = x ? N : v = y ? N : v = z ? 5 : U,

which, in fact, points that neither x nor y are constants past the join point, but z indeed is.

How is the solution that our iterative data-flow solver produces for this example?

Non-Distributivity

- The iterative algorithm applies the meet operator too early, before computations would take place.
  - This early evaluation is necessary to avoid infinite program paths, but it may generate imprecision



Product Lattices

- If (A, $\bigwedge_A$) and (B, $\bigwedge_B$) are lattices, then a product lattice A×B has the following characteristics:
  - The domain is A × B
  - The meet is defined by (a, b) $\bigwedge$ (a', b') = (a $\bigwedge_A$ a', b $\bigwedge_B$ b')
  - The ordering is (a, b) ≤ (a', b') ⇔ a ≤ a' and b ≤ b'
- Our system of data-flow equations ranges over a product lattice:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

Which product lattice are we talking about in this case?

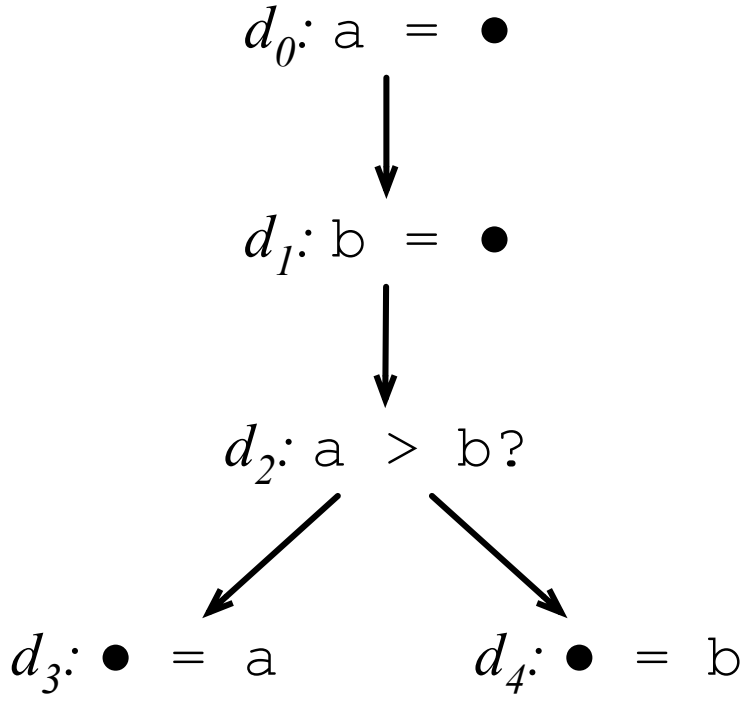
Product Lattices

- Our system of data-flow equations ranges over a product lattice:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

- If every x_i is a point in a lattice L, then the tuple $(x_1, \ldots, x_n)$ is a point in the product of n instances of the lattice L, e.g,. $(L^1 \times \ldots \times L^n)$

Let's show how this product lattice surfaces using the example on the right. Can you give me the IN and OUT sets for liveness analysis for this example?



Product Lattices

$$IN[d_0] = OUT[d_0] \setminus \{a\}$$

$$OUT[d_0] = IN[d_1]$$

$$IN[d_1] = OUT[d_1] \setminus \{b\}$$

$$OUT[d_1] = IN[d_2]$$

$$IN[d_2] = OUT[d_2] \cup \{a, b\}$$

$$OUT[d_2] = IN[d_3] \cup IN[d_4]$$

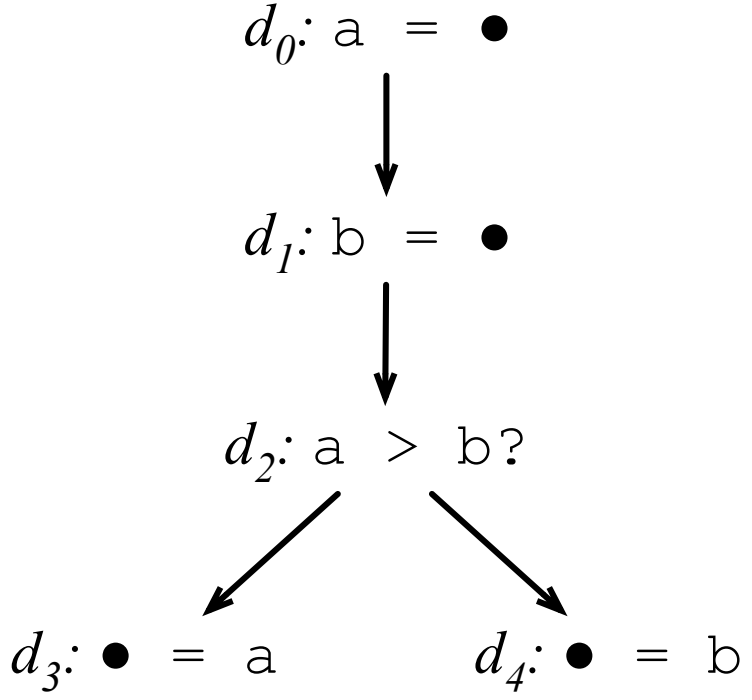
$$IN[d_3] = OUT[d_3] \cup \{a\}$$

$$OUT[d_3] = \{\}$$

$$IN[d_4] = OUT[d_4] \cup \{b\}$$

$$OUT[d_4] = \{\}$$

That is a lot of equations! Lets just work with OUT sets. Can you simplify the IN sets away?



Product Lattices

$OUT[d_0] = OUT[d_1] \setminus \{b\}$

$OUT[d_1] = OUT[d_2] \cup \{a, b\}$

$OUT[d_2] = OUT[d_3] \cup \{a\} \cup OUT[d_4] \cup \{b\}$

$OUT[d_3] = \{\}$

$OUT[d_4] = \{\}$

But, given that now we only have out sets, lets just call each constraint variable x_i

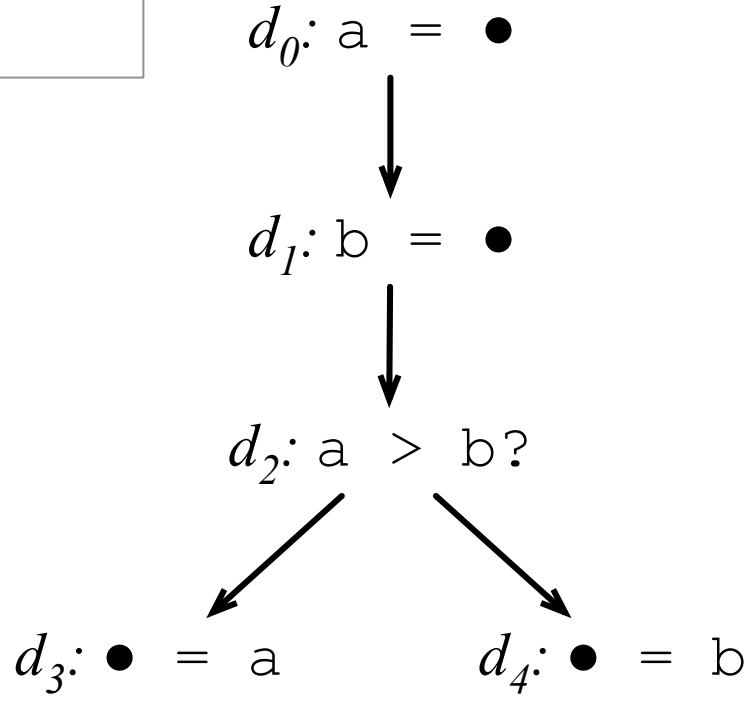
$x_0 = x_1 \setminus \{b\}$

$x_1 = x_2 \cup \{a, b\}$

$x_2 = x_3 \cup x_4 \cup \{a, b\}$

$x_3 = \{\}$

$x_4 = \{\}$



Product Lattices

$x_0 = x_1 \setminus \{b\}$ $\quad x_3 = \{\}$ $\quad x_2 = x_3 \cup x_4 \cup \{a, b\}$

$x_1 = x_2 \cup \{a, b\}$ $\quad x_4 = \{\}$

$$(x_0, x_2, x_3, x_4, x_5) = (x_1 \setminus \{b\}, x_2 \cup \{a, b\}, x_3 \cup x_4 \cup \{a, b\}, \{\}, \{\})$$

Which product lattice do we have in this equation?

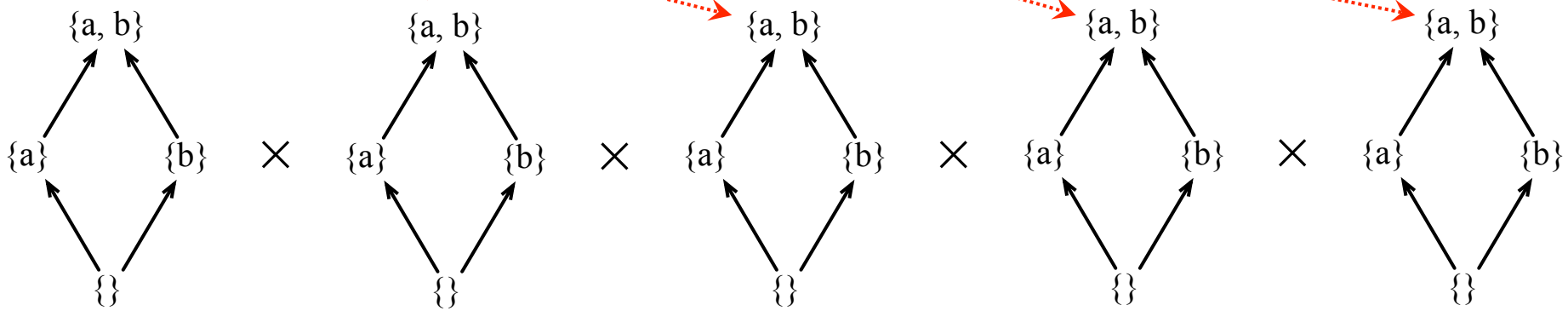
Product Lattices

$x_0 = x_1 \setminus \{b\}$ $\quad x_3 = \{\}$ $\quad x_2 = x_3 \cup x_4 \cup \{a, b\}$

$x_1 = x_2 \cup \{a, b\}$ $\quad x_4 = \{\}$

How many elements will our product lattice have?

$$(x_0, x_2, x_3, x_4, x_5) = (x_1 \setminus \{b\}, x_2 \cup \{a, b\}, x_3 \cup x_4 \cup \{a, b\}, \{\}, \{\})$$



A Bit of History

- Lattices existed in mathematics before compilers came to be, yet a lot of work has been done in this field motivated by program analysis. Among the pioneers, we have Vyssotsky, Allen, Cocke and Kildall

- Allen, F. E., "Control Flow Analysis", ACM Sigplan Notices 5:7 (1970), pp. 1-19
- Cocke, J., "Global Common Subexpression Elimination", ACM Sigplan Notices 5:7 (1970) pp. 1-19
- Kildall, G. "A Unified Approach to Global Program Optimizations", ACM Symposium on Principles of Programming Languages (1973), pp. 194-206
- Vyssotsky, V. and P. Wegner, "A Graph theoretical Fortran Source Language Analyzer", Technical Report, Bell Laboratories, 1963