Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 16

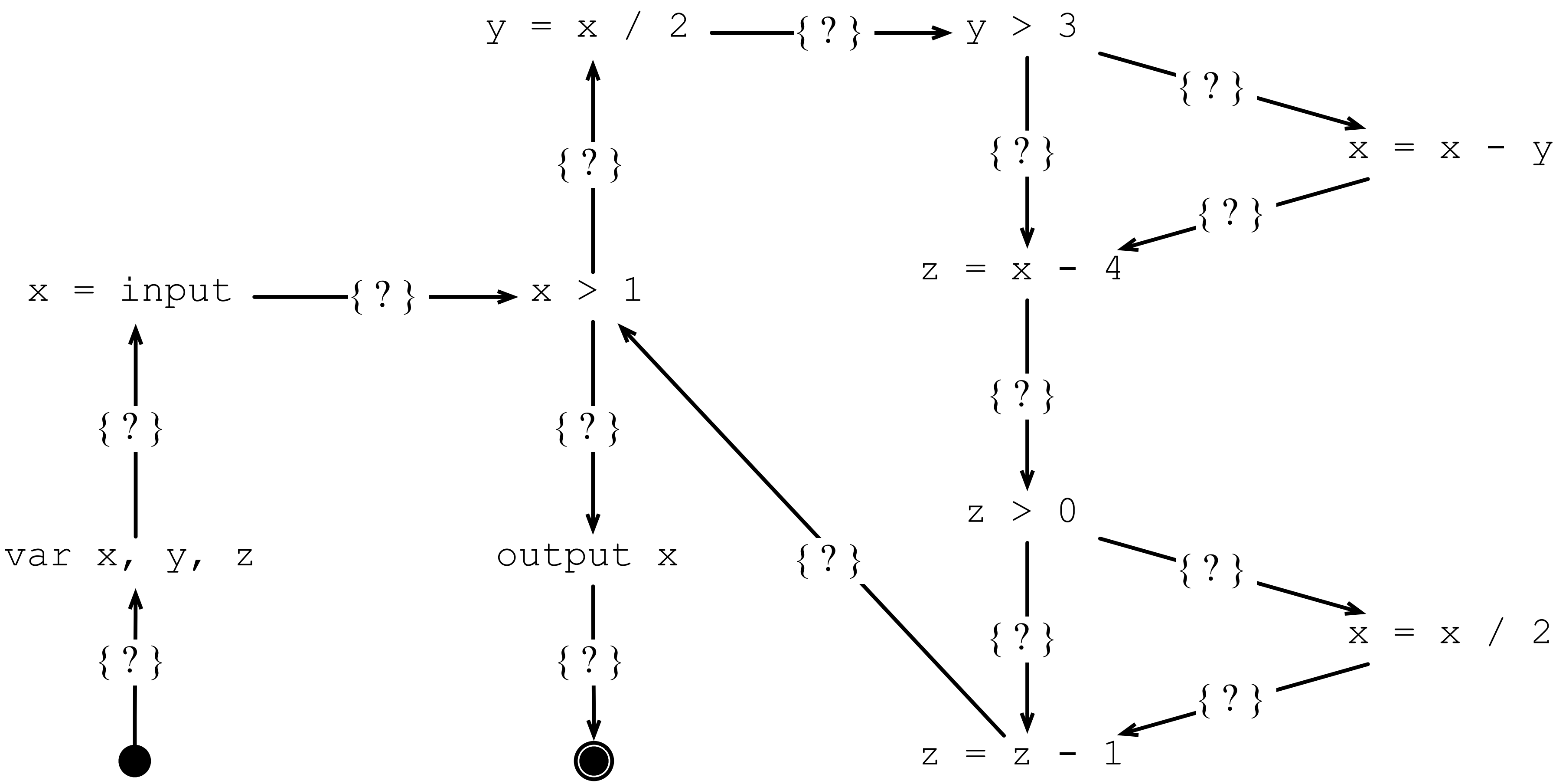
- The Dataflow framework for Global Analysis

The Dataflow Framework

- We will present a simple iterative algorithm to find the solution of a given dataflow problem.
- Main issues:
  - How do we know that this algorithm terminates?
  - How precise is the solution produced by this algorithm?
- We provide a formal ground to the theory using the algebraic notion of *semilattice*

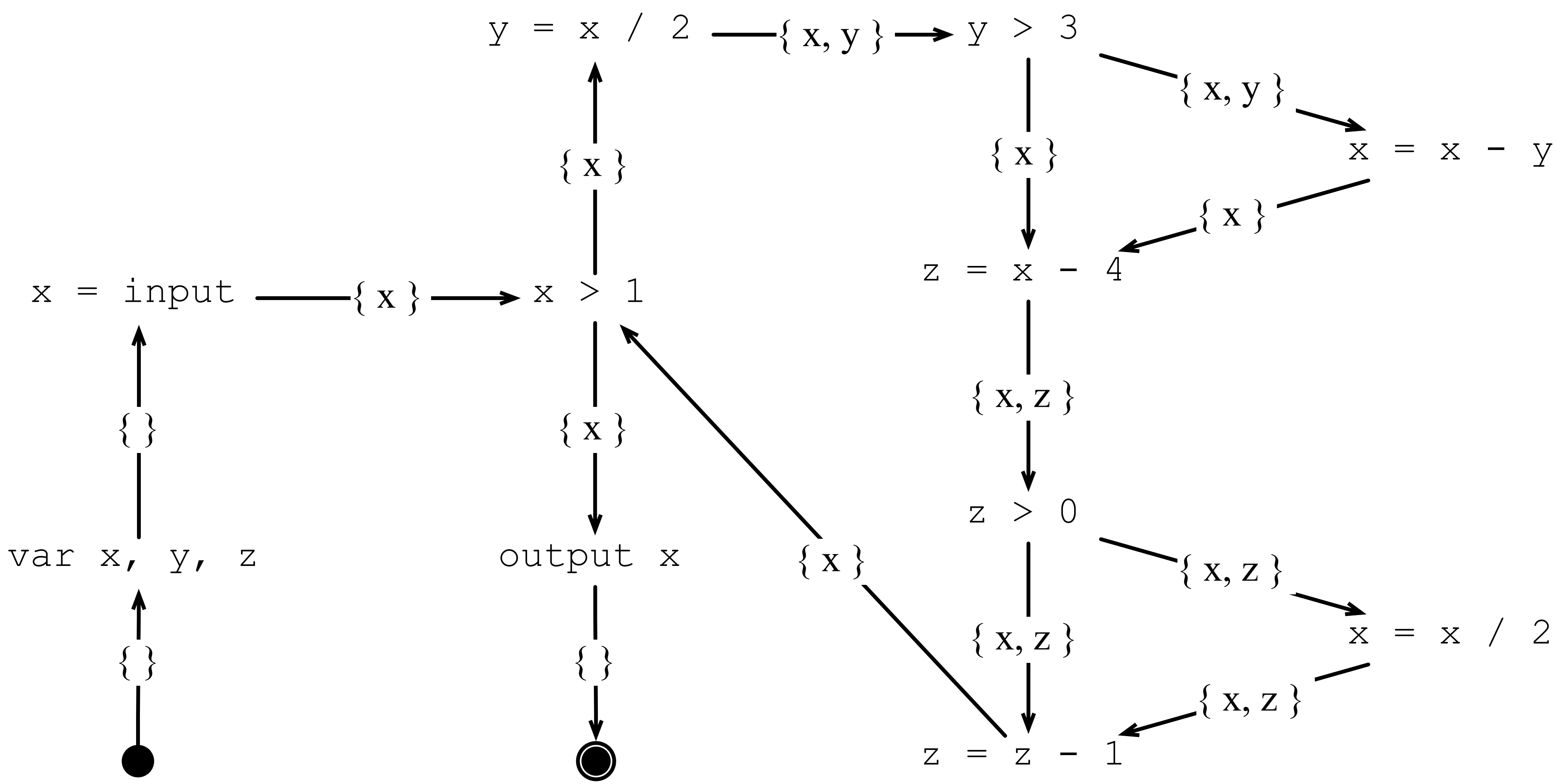
How do we know that liveness terminates?

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



Example of Liveness Problem

- Solution of the liveness analysis for the sample program



Dataflow Equations for Liveness

$$p : v = E$$

$$\begin{aligned} IN(p) &= (OUT(p) \setminus \{v\}) \cup vars(E) \\ OUT(p) &= \bigcup IN(p_s), p_s \in succ(p) \end{aligned}$$

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

(Meet) Semilattices

- All dataflow analyses map program points to elements of algebraic structures called *semilattices*.
- A *(meet) semilattice* L = (S, ≤, $\wedge$, T) is formed by:
  - A set S
  - A partial order ≤ between elements of S.
  - A greatest element T (**top**)
  - A binary **meet** operator $\wedge$
- The meet has to satisfy
  - $e_1 \wedge e_2 \leq e_1$ and $e_1 \wedge e_2 \leq e_2$
  - For any $e' \in S$, if $e' \leq e_1$ and $e' \leq e_2$, then $e' \leq e_1 \wedge e_2$
- It follows that $x \leq y$ if, and only if $x \wedge y = x$
- Thus we could also define a semilattice omitting ≤

Properties of Meet

- Meet is idempotent: $x \wedge x = x$
- Meet is commutative: $y \wedge x = x \wedge y$
- Meet is associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- The top element T is the unit: $(T \wedge x) = x$

By the above properties the meet can be extended to arbitrary finite subsets of S:

- $\bigwedge \{\} = T$
- $\bigwedge \{s_1, \ldots, s_{n+1}\} = (\bigwedge \{s_1, \ldots, s_n\}) \wedge s_{n+1}$
- $\bigwedge X$ is the **greatest lower bound** of X, for $X \subseteq S$

Semilattice structures on powersets

- The four examples of dataflow analysis that we have seen all use *subsets* of a suitable set as values for the INs and the OUTs of program points (sets of variables, of expressions, or of definitions)
- Some dataflow analyses, such as *live variables*, use the *union* of sets for joining information
- Others, such as *available expressions*, use the *intersection of sets* for joining information
- As a matter of fact, the powerset of a set is a *lattice*, and depending on our needs we can consider it as a *meet semilattice* in two different ways

The Semilattice of Liveness Analysis

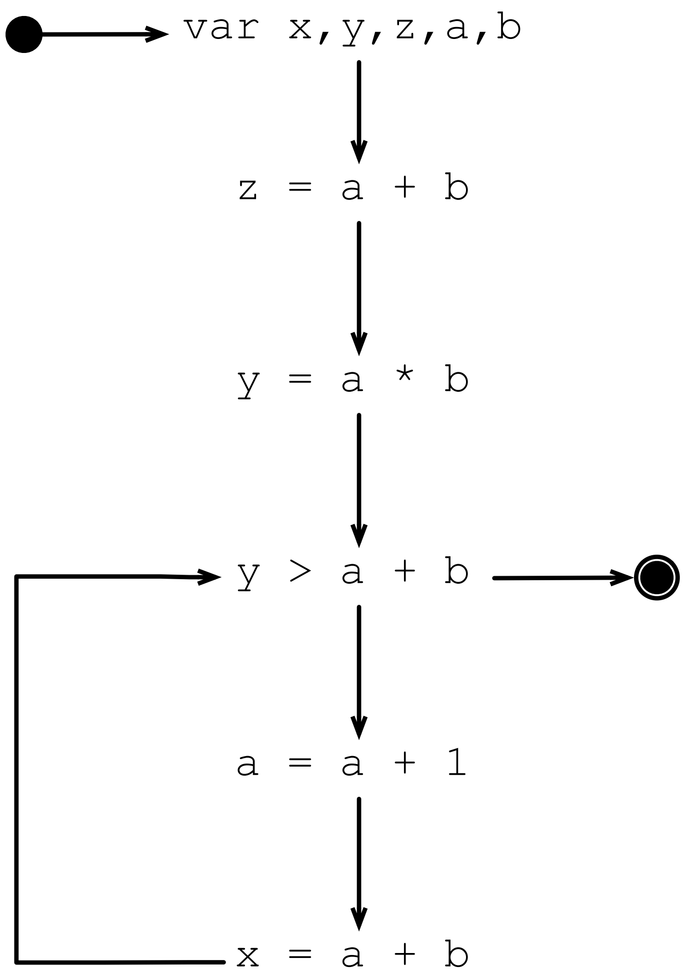
```
var x,y,z;

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 consider the semilattice $L = (P^{\{x, y, z\}}, \supseteq, \cup, \{\})$
- The **partial order** is set inclusion, i.e. $A \leq B$ iff $A \supseteq B$ for all $A,B \subseteq \{x, y, z\}$
- The **meet** operator is union
- The **top element** is the empty set

The expected properties hold:

- $A \leq B$ if and only if $A \wedge B = A$
  - $A \supseteq B$ if and only if $A \cup B = A$
- Top is the unit of meet: $(T \wedge A) = A$
  - $\{\} \cup A = A$

The Semilattice of Available Expressions



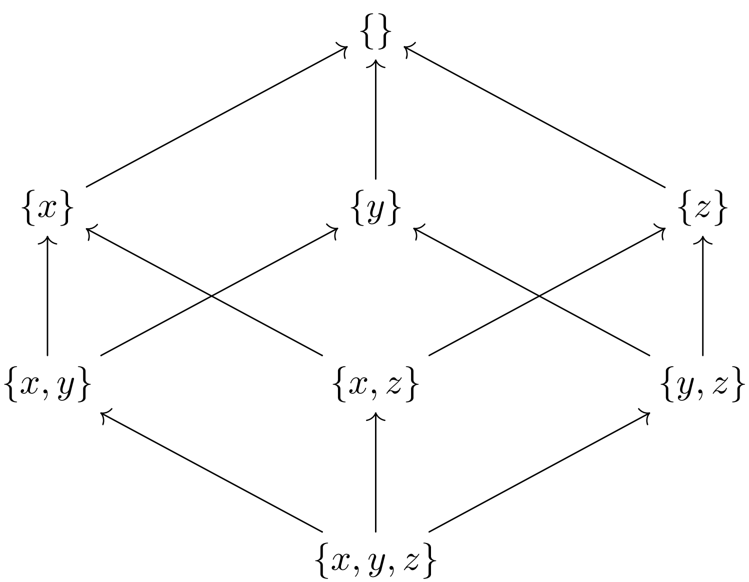
- Given the program on the left, we have the semilattice $L = (P^{\{a+b,\ a*b,\ a+1\}}, \subseteq, \cap, \{a+b, a*b, a+1\})$
- Let $D = \{a+b, a*b, a+1\}$
- The **partial order** is set containement, i.e. $A \leq B$ iff $A \subseteq B$ for all $A,B \subseteq D$
- The **meet** operator is intersection
- The **top element** is D

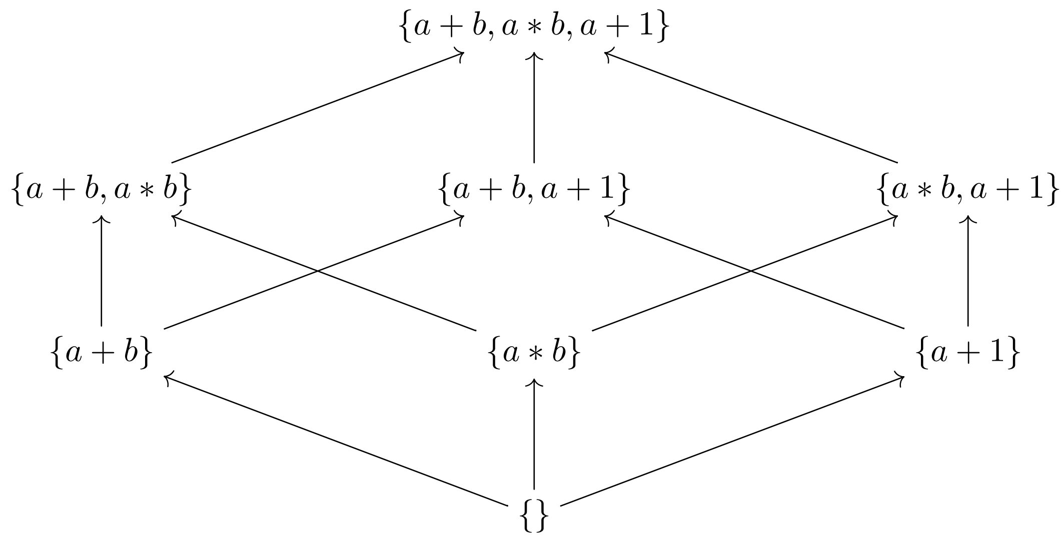
The expected properties hold:

- $A \leq B$ if and only if $A \wedge B = A$
  - $A \subseteq B$ if and only if $A \cap B = A$
- Top is the unit of meet: $(T \wedge A) = A$
  - $D \cap A = A$ for all $A \subseteq D$

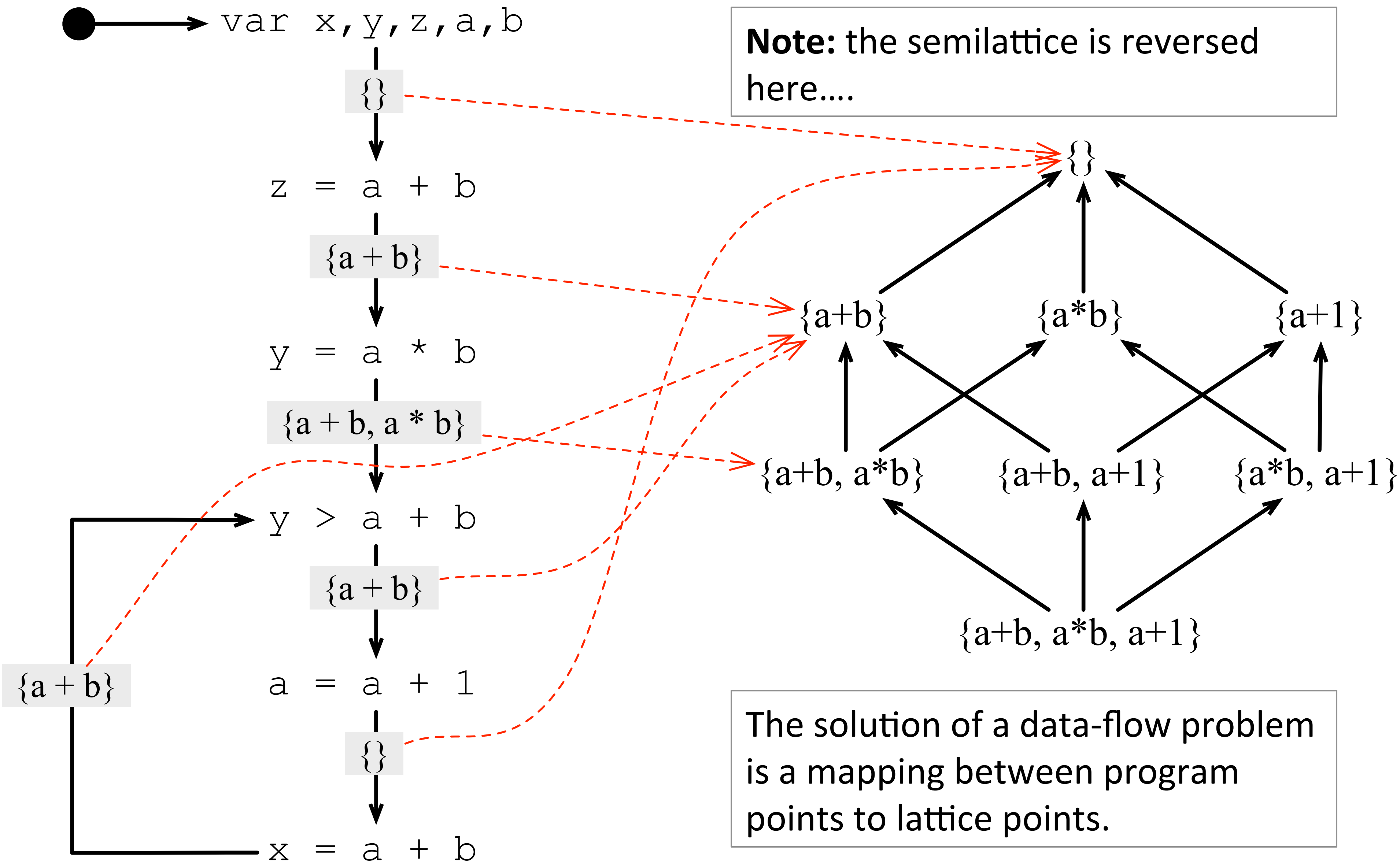
Lattice Diagrams

- We can represent the partial order between the elements of a (semi)lattice as a *Hasse Diagram*.
  - If there exists a path in this diagram, from an element e_1 to another element e_2, then we say that $e_1 \leq e_2$
  - The greatest element is at the top





Mapping Program Points to Lattice Points



Note: the semilattice is reversed here....

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

Data-Flow Analysis Framework

- A *Data-Flow Analysis Framework* (D, S, $\wedge$, F) consists of:
  - A *direction D in {FORWARDS, BACKWARDS}*
  - A *domain of values* (S, $\wedge$) which forms a *meet semilattice*
  - A family *F* of *transfer functions* from S to S, including the identity function and closed under composition

Monotone Transfer Functions

- Given family of functions F over a semilattice S (i.e. such that for all $f \in F$, $f : S \rightarrow S$), these properties are equivalent:
  - Any element $f \in F$ is **monotonic**, that is
    for all $x \in S$, $y \in S$, and $f \in F$, $x \leq z\ y$ implies $f(x) \leq f(y)$
  - For all x and y in S and f in F, $f(x \wedge y) \leq f(x) \wedge f(y)$
- A dataflow analysis framework is **monotone** if all transfer functions *f* in *F* are monotonic
- It is **distributive** if for all *f* in *F* $f(x \wedge y) = f(x) \wedge f(y)$

Monotone Transfer Functions

- It is easy to check that the transfer functions used in the liveness analysis problem are monotonic:

$$p : v = E$$

$$\begin{array}{rcl} IN(p) & = & (OUT(p) \setminus \{v\}) \cup vars(E) \\ OUT(p) & = & \bigcup IN(p_s), p_s \in succ(p) \end{array}$$

- And the same for those used for available expressions:

$$p : v = E$$

$$\begin{array}{rcl} IN(p) & = & \bigcap OUT(p_s), p_s \in pred(p) \\ OUT(p) & = & (IN(p) \cup \{E\}) \setminus \{Expr(v)\} \end{array}$$

- Often *basic blocks* are annotated with values instead of individual statements: OUT[B] and IN[B]

Data-Flow Iterative Algorithm [Forward]

- Given:
  - a data-flow graph with ENTRY and EXIT nodes
  - one transfer function f_B for each basic block *B*
  - A "boundary condition" v_{ENTRY}
- Computes values IN[B] and OUT[B] for all blocks

1) OUT[ENTRY] = v_{ENTRY};
2) for each block B, but ENTRY {
3) OUT[B] = T }
4) while (changes to any OUT occur) {
5) for (each basic block B other than ENTRY){
6) IN[B] = $\bigwedge_{\text{P a predecessor of B}}$ OUT[P];
7) OUT[B] = f_B(IN[B]); } }

Example: Dataflow analysis for Reaching Definitions

- Each point in the program is associated with the set of definitions that are active at that point
- Semilattice:
  - Powerset of definitions (assignments)
  - Meet operator: union. Top element: empty set
- The *transfer function* for a block kills definitions of variables that are redefined in the block and adds definitions of variables that occur in the block: $f_B(x) = gen_B \cup (x - kill_B)$
- The confluence operator is union.

Termination

The algorithm is ensured to terminate if

- The framework is monotonic
- The semilattice S has finite height
  - It is not possible to have an infinite chain of elements in *S*, e.g., $l_0 \leq l_1 \leq l_2 \leq \ldots$ such that every element in this chain is different
  - Notice that the semilattice can still have an infinite number of elements.

Asymptotic Complexity of the Solver

Assumption: a lattice of height H, and a program with B blocks.

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.

Constraint System and Fixed Point

- The application of a dataflow analysis framework to a Control Flow Graph determines a constraint system
  $$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$
  where the unknowns are the sets INs and OUTs
- The solution is a fixed point of F. i.e, a tuple of elements such that
  $$F(x_1, \ldots, x_n) = (x_1, \ldots, x_n)$$
- 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
- Such a solution is *conservative*

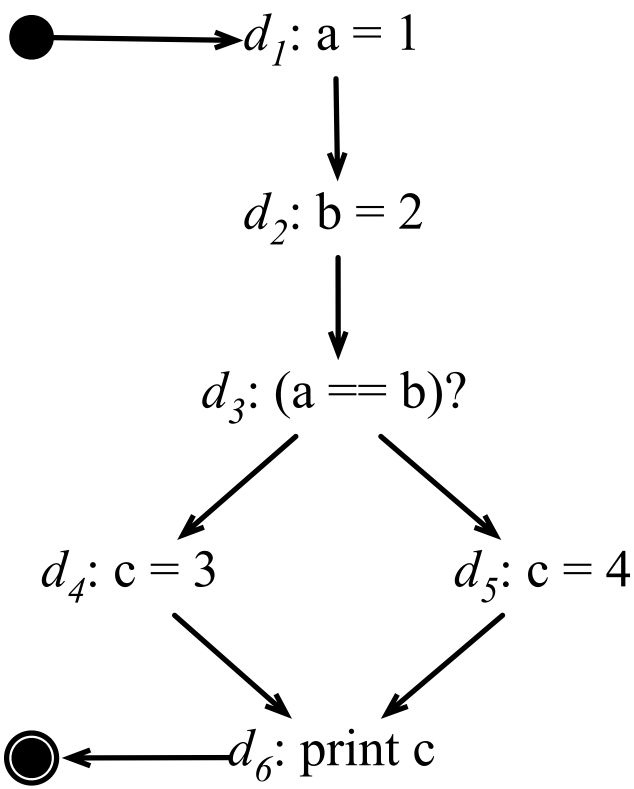
Accuracy, Safeness, and Conservative Estimations

In the framework of static analysis:

- *Conservative*: refers to making safe assumptions when insufficient information is available at compile time, i.e. the compiler has to guarantee not to change the meaning of the optimized code
- *Safe*: (similar) the values computed by the analysis approximate the exact ones in a way that does not affect the meaning of the optimized code
- *Accuracy*: refers to the fact that more and better information enables more code optimizations

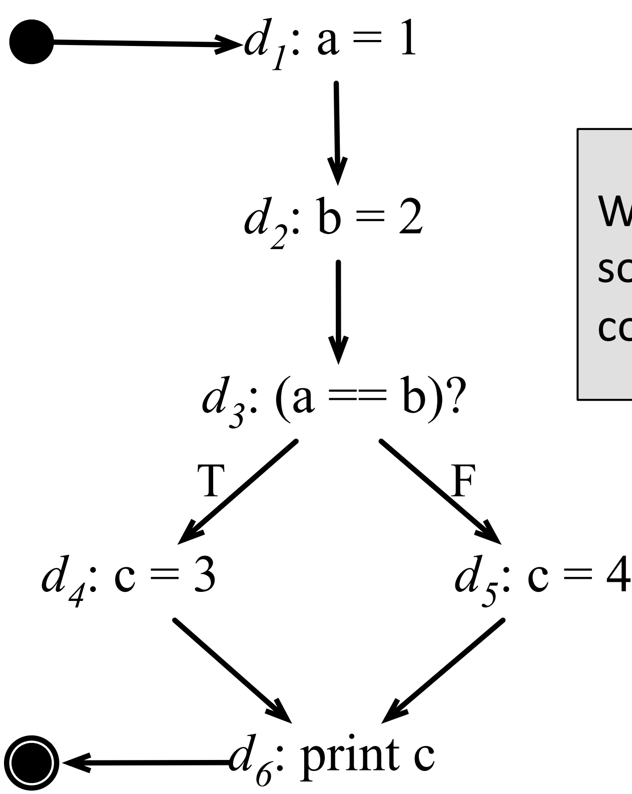
Example: Reaching Definitions

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

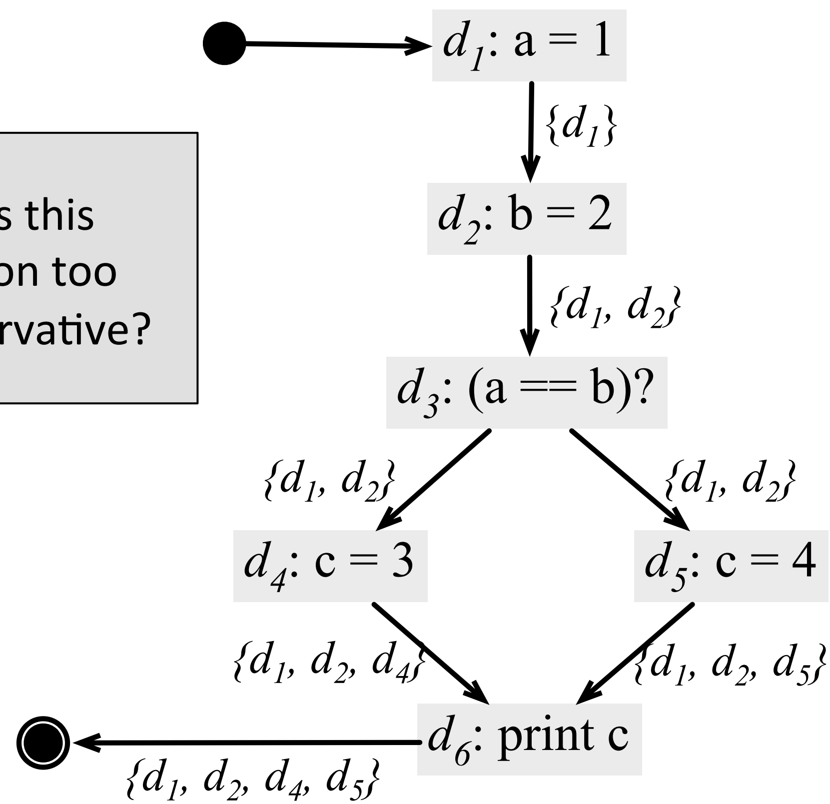


Example: Reaching Definitions

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



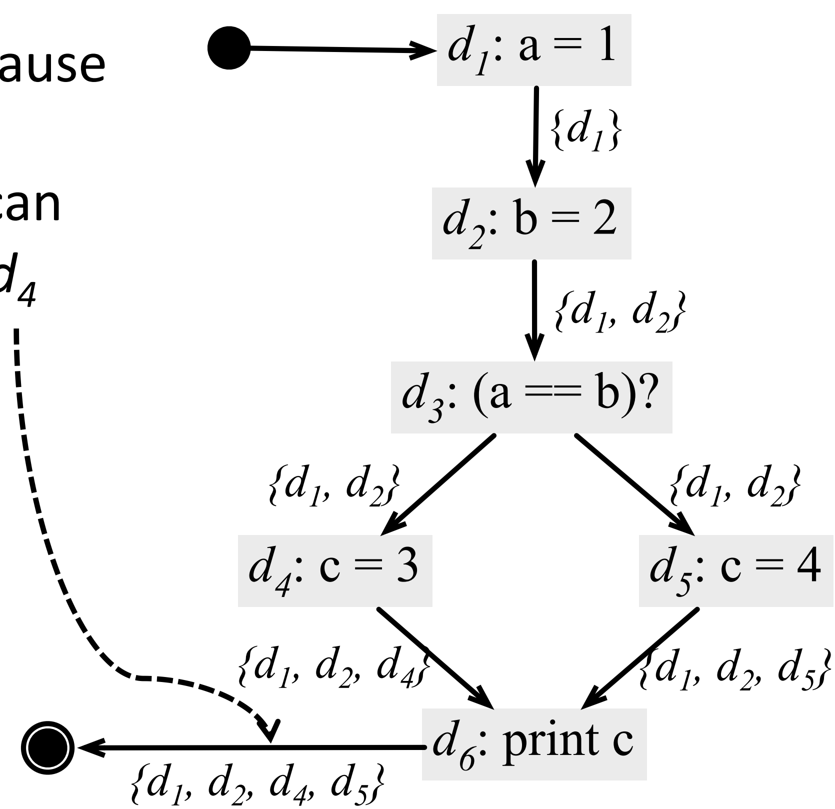
Why is this solution too conservative?



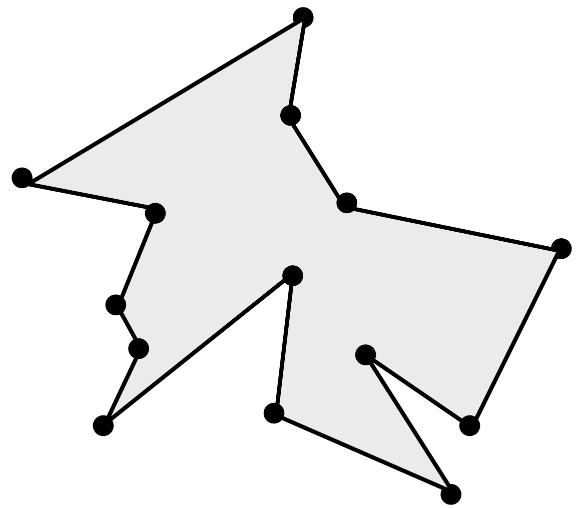
Example: Reaching Definitions

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

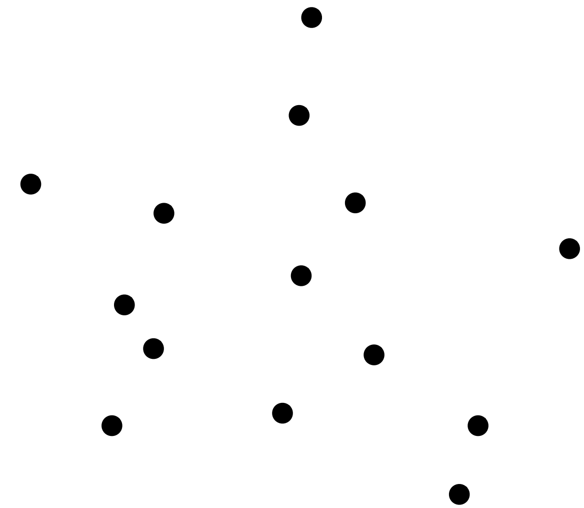
The solution is conservative because the branch is always false. Therefore, the statement c = 3 can never occur, and the definition d_4 will never reach the end of the program.



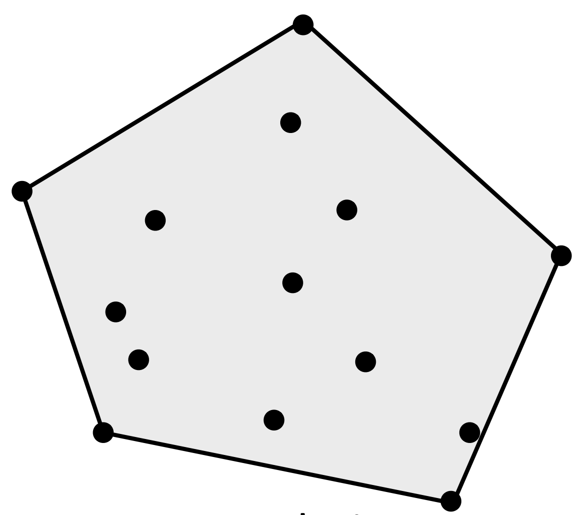
More Intuition on MFP Solutions



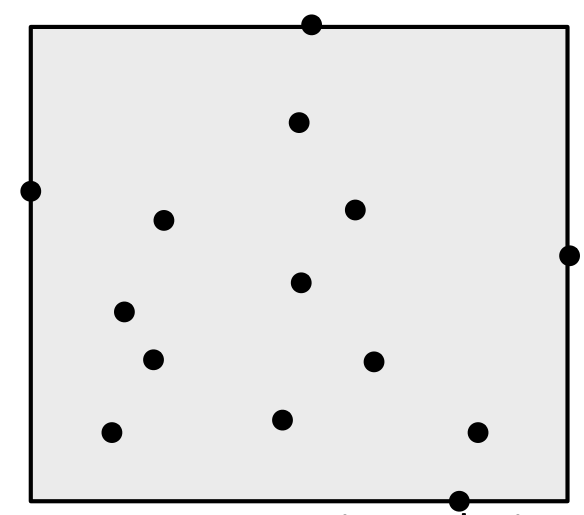
Actual Program Behavior



Constraints

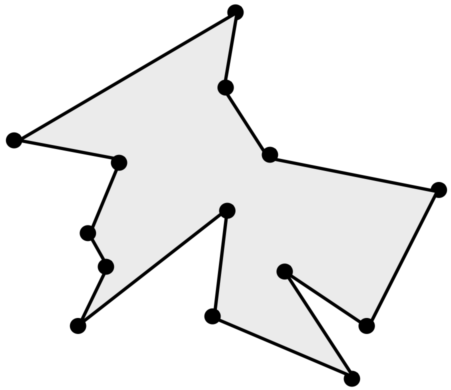


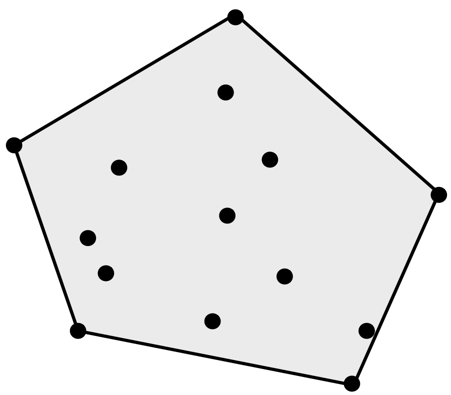
MFP Solution



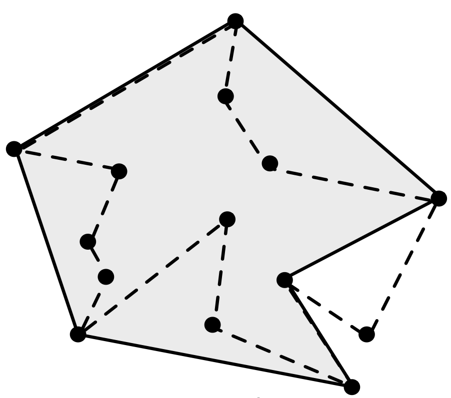
Over-Conservative Solution

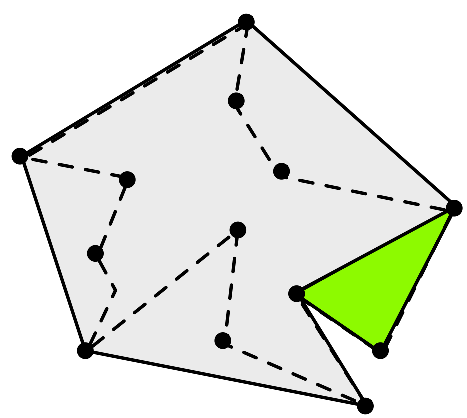
Wrong Solutions: False Negatives

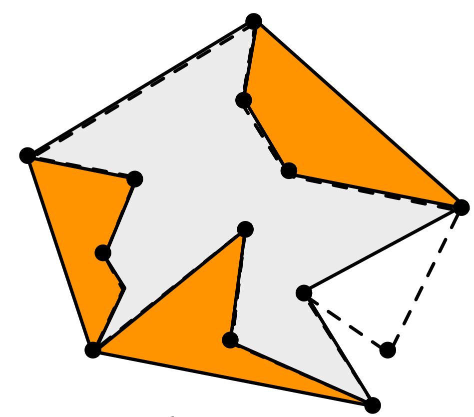

Actual Program Behavior


MFP 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

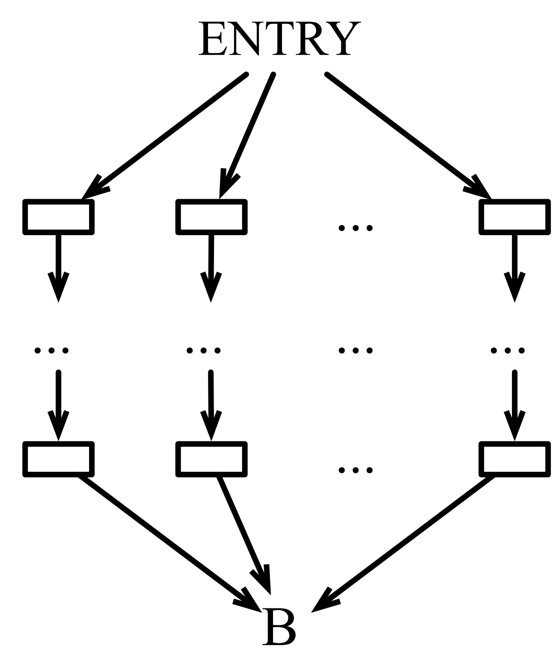

Wrong Solution


False Negative


False Positive

The Ideal Solution

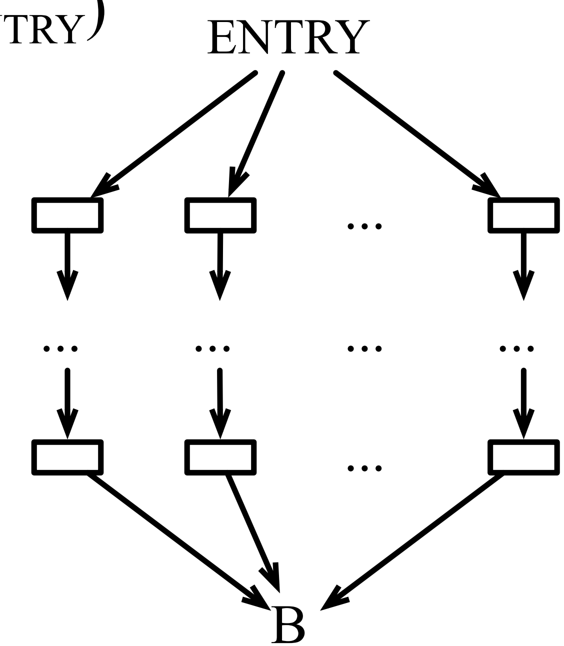
- The MFP solution fits the constraint system tightly, but it is a conservative solution.
  - What would be an ideal solution?
  - In other words, given a block B in the program, what would be an ideal solution of the dataflow problem for the IN set of B?
- The ideal solution computes dataflow information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

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



- 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.
- We define our meet over all paths (MOP) solution:

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

- Two natural questions:
  - 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
- **Note:** our four examples of data-flow analyses, e.g., liveness, availability, reaching defs and anticipability, are distributive. Let's see why….

Distributive Frameworks

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

$$f(x \wedge x') = ((x \wedge x') \setminus x_k) \cup x_g \quad \textbf{(i)}$$

$$= ((x \setminus x_k \wedge x' \setminus x_k)) \cup x_g \quad \textbf{(ii)}$$

$$= ((x \setminus x_k) \cup x_g) \wedge ((x' \setminus x_k) \cup x_g) \quad \textbf{(iii)}$$

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

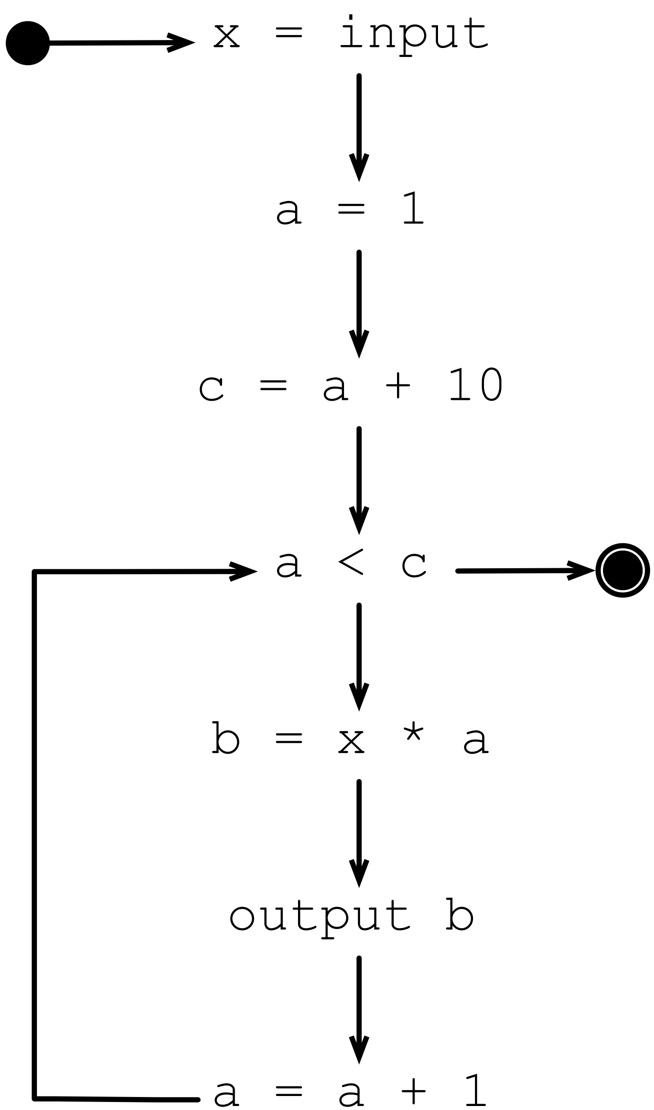
To see why (ii) and (iii) are true, just remember that in any of the four data-flow analyses, either $\wedge$ is $\cap$, or it is $\cup$.

Map Lattices

- If S is a set and L = (T, $\wedge$)♤ is a meet semilattice, then the structure $L_s = (S \to T, \wedge_s)$ is also a semilattice, which we call a *map semilattice*.
  - The domain is $S \to T$ (functions from S to T)
  - The meet is defined by $f \wedge f' = \lambda x.f(x) \wedge f'(x)$
    - equivalently, $(f \wedge f')(x) = f(x) \wedge f'(x)$
  - 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.

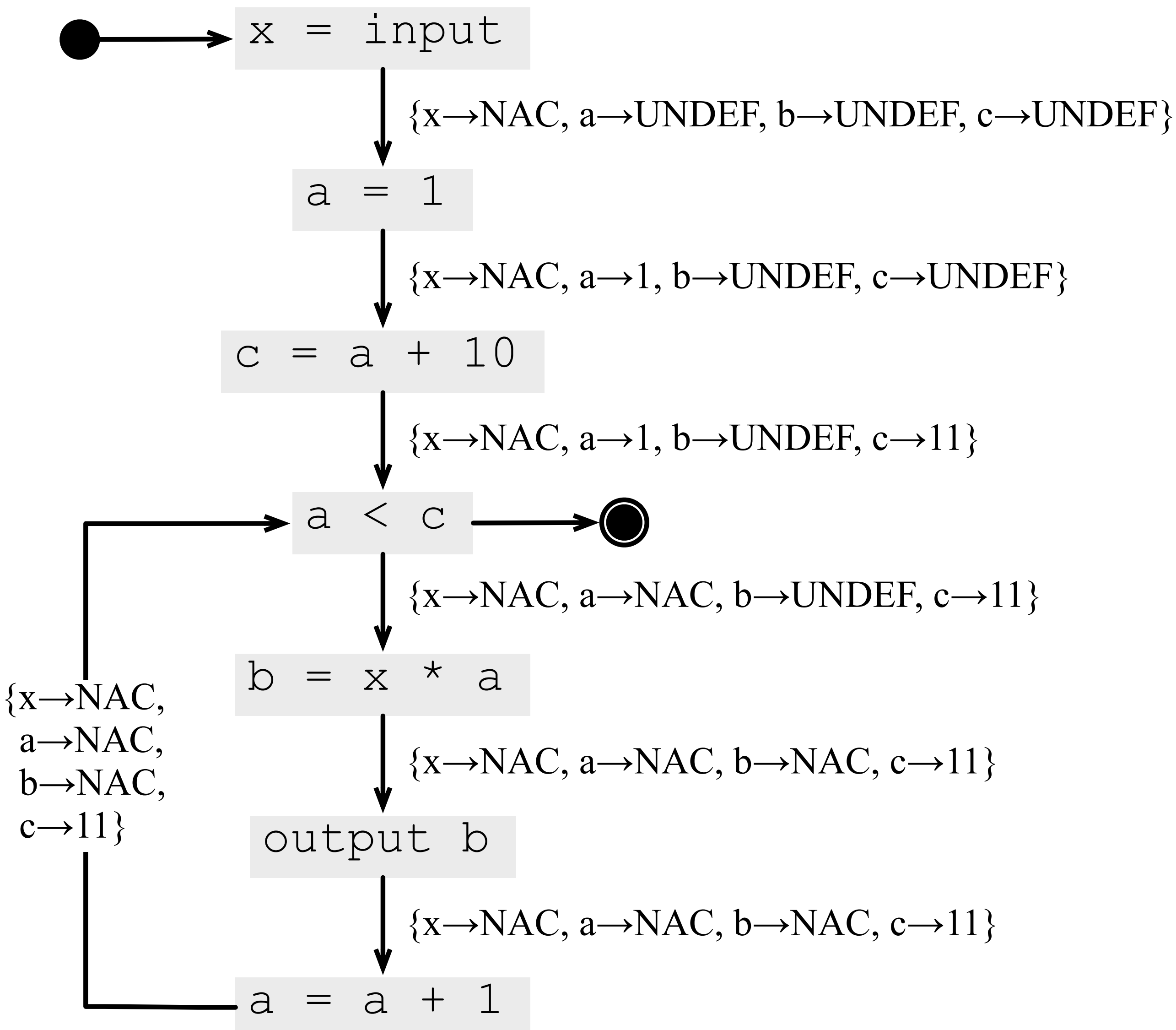
♤: This is set "T", not the symbol of "top"

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?

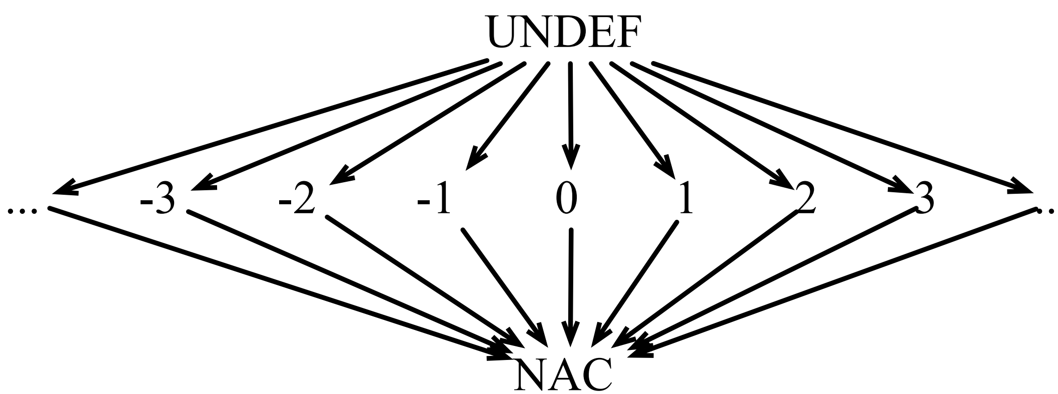
Constant Propagation



What is the semilattice 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:

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

$p : v = c$

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

$p : v = u$

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

$p : v = t + u$

$$\begin{aligned}
OUT(p) &= \lambda x.x = v \; ? \; IN(p)(t) + IN(p)(u) \; : \; IN(p)(x), \textit{if } IN(p)(t), IN(p)(u) \textit{ consts} \\
OUT(p) &= \lambda x.x = v \; ? \; NAC \; : \; IN(p)(x), \textit{if } IN(p)(t) \textit{ or } IN(p)(u) \textit{ not const} \\
OUT(p) &= \lambda x.x = v \; ? \; UNDEF \; : \; IN(p)(x), \textit{otherwise}
\end{aligned}$$

$---$

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

The meet operator

$\wedge$	UNDEF	c_1	NAC
UNDEF	UNDEF	c_1	NAC
c_2	c_2	$c_1 \wedge c_2$	NAC
NAC	NAC	NAC	NAC

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

Are these functions monotone?

$p : v = c$

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

$p : v = u$

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

$p : v = t + u$

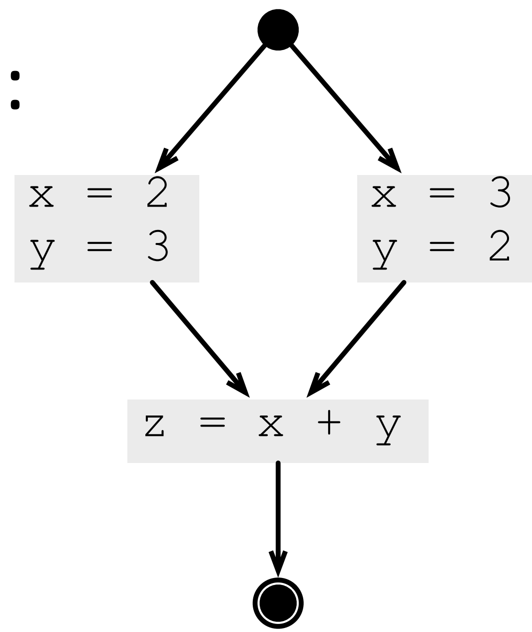
$$\begin{aligned} OUT(p) &= \lambda x.x = v \; ? \; IN(p)(t) + IN(p)(u) \; : \; IN(p)(x), \textit{if } IN(p)(t), IN(p)(u) \textit{ consts} \\ OUT(p) &= \lambda x.x = v \; ? \; NAC \; : \; IN(p)(x), \textit{if } IN(p)(t) \textit{ or } IN(p)(u) \textit{ not const} \\ OUT(p) &= \lambda x.x = v \; ? \; UNDEF \; : \; IN(p)(x), \textit{otherwise} \end{aligned}$$

$---$

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

Non-Distributivity

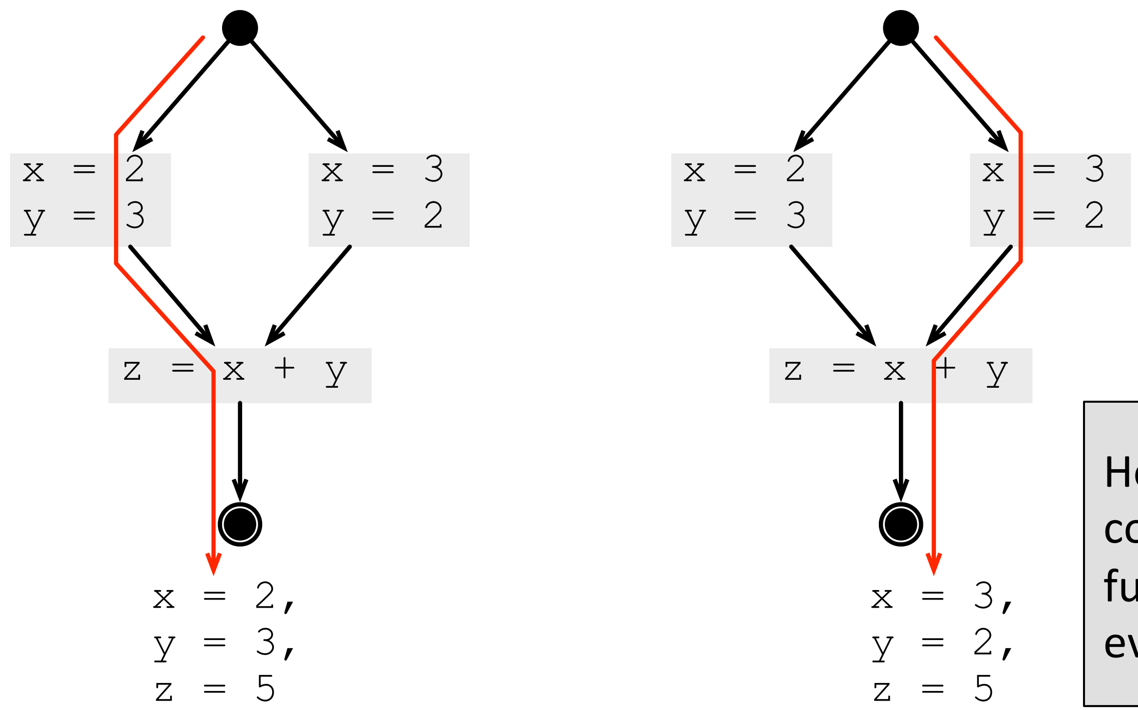
- The constant propagation framework is monotone, but it is not distributive.
  - As a consequence, the MOP solution is more precise than the iterative solution.
- Consider this program:



Is variable z a constant in this program?

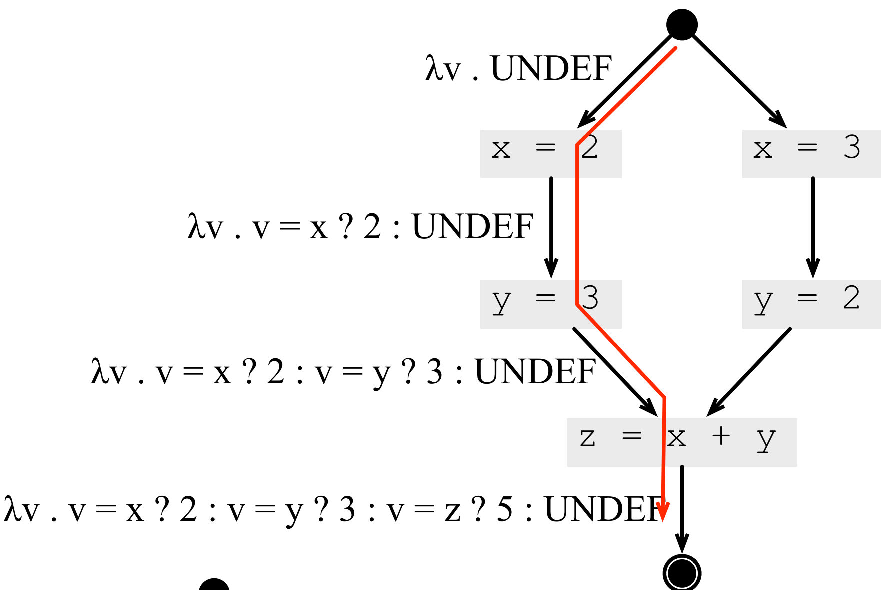
Non-Distributivity

- 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.

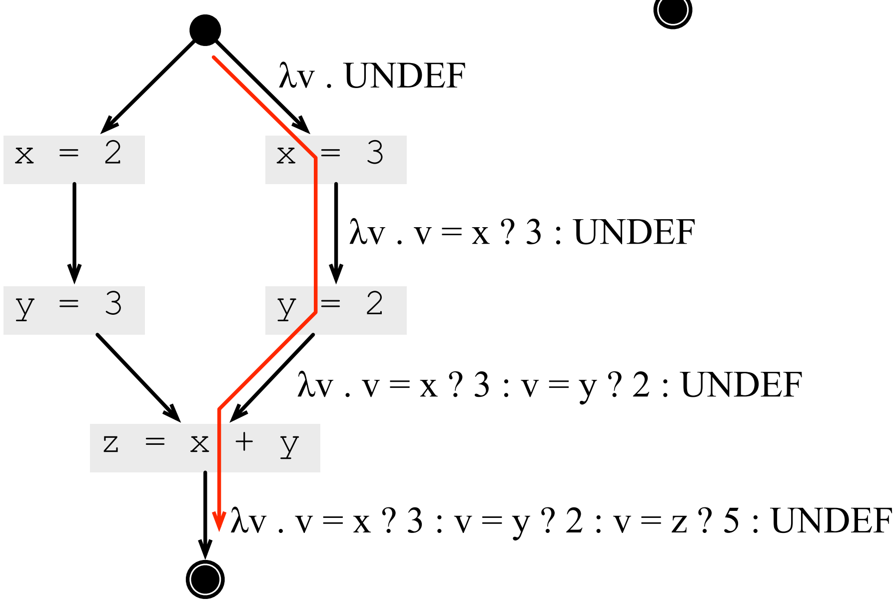


How is the composition of functions along every path?

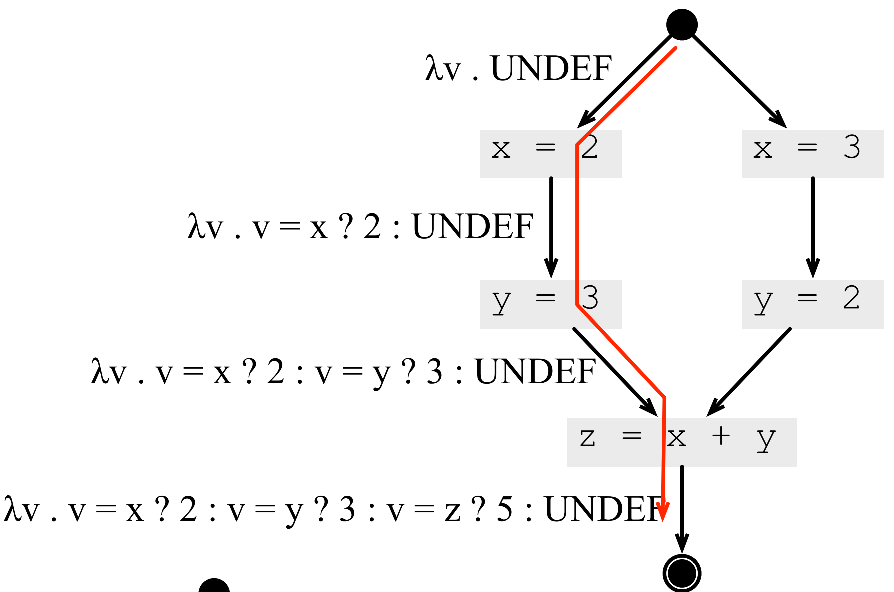
Non-Distributivity

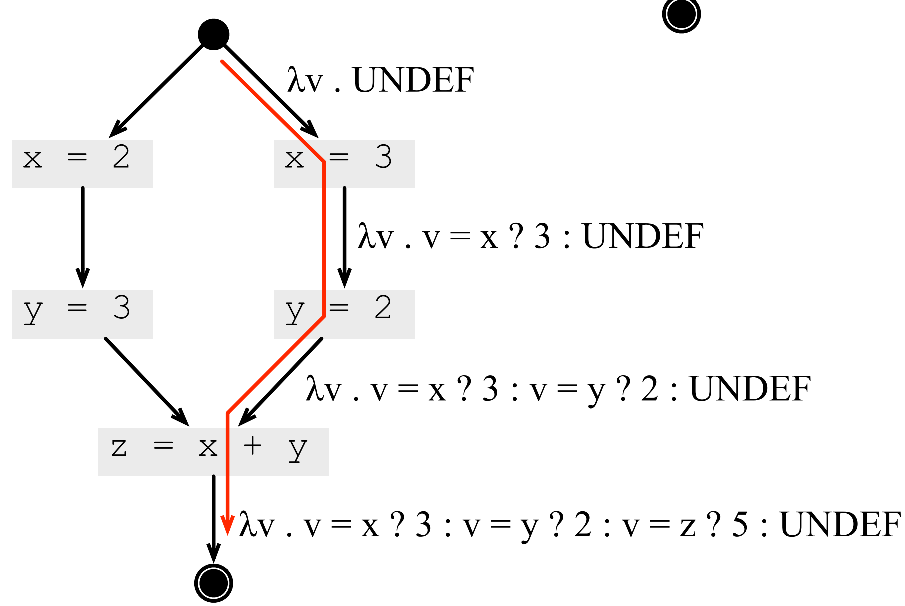


What is the meet of the functions
λv . v = x ? 3 : v = y ? 2 : v = z ? 5 : U
and
λv . v = x ? 2 : v = y ? 3 : v = z ? 5 : U?



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:

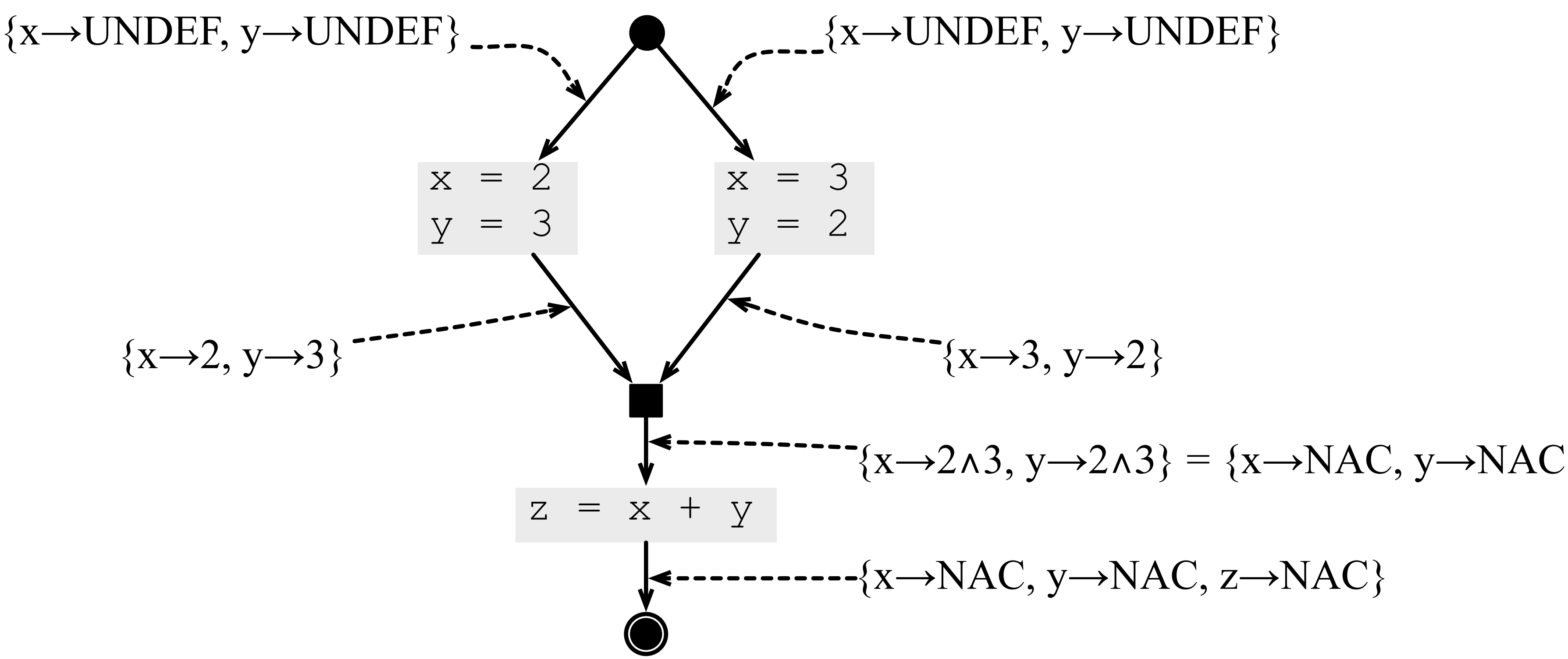
λv . v = x ? N : v = y ? N : v = z ? 5 : U,

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, $\wedge_A$) and (B, $\wedge_B$) are lattices, then a product lattice A×B has the following characteristics:
  - The domain is $A \times B$
  - The meet is defined by

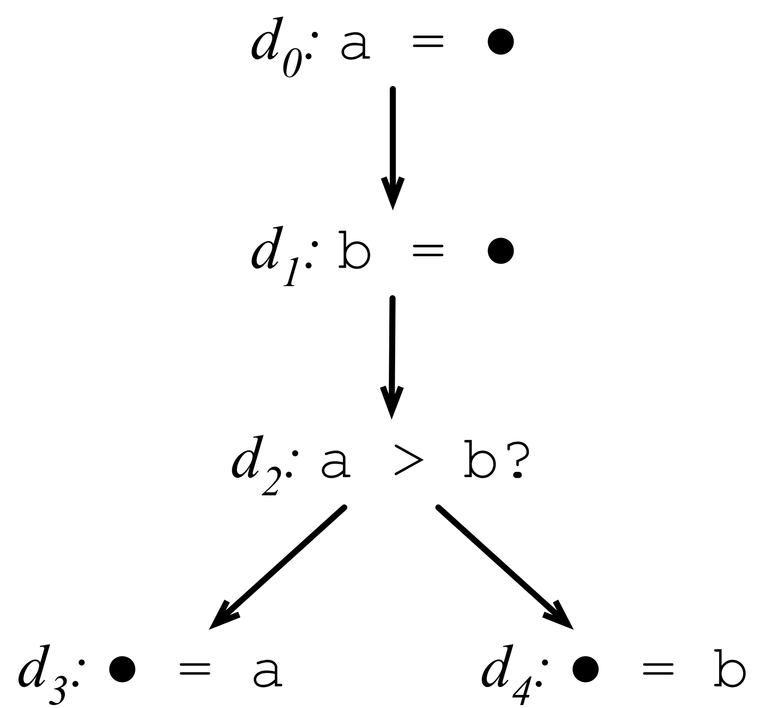
    $$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b')$$
  - The ordering is $(a, b) \leq (a', b') \Leftrightarrow a \leq a'$ and $b \leq b'$
- The system of data-flow equations ranges over a product lattice:

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

- 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,. $(L^1 \times \ldots \times L^n)$

Product Lattices

Let's show how this product lattice surfaces using the example on the right. What are the IN and OUT sets for liveness analysis for this example?



Product Lattices

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

$OUT[d_0] = IN[d_1]$

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

$OUT[d_1] = IN[d_2]$

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

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

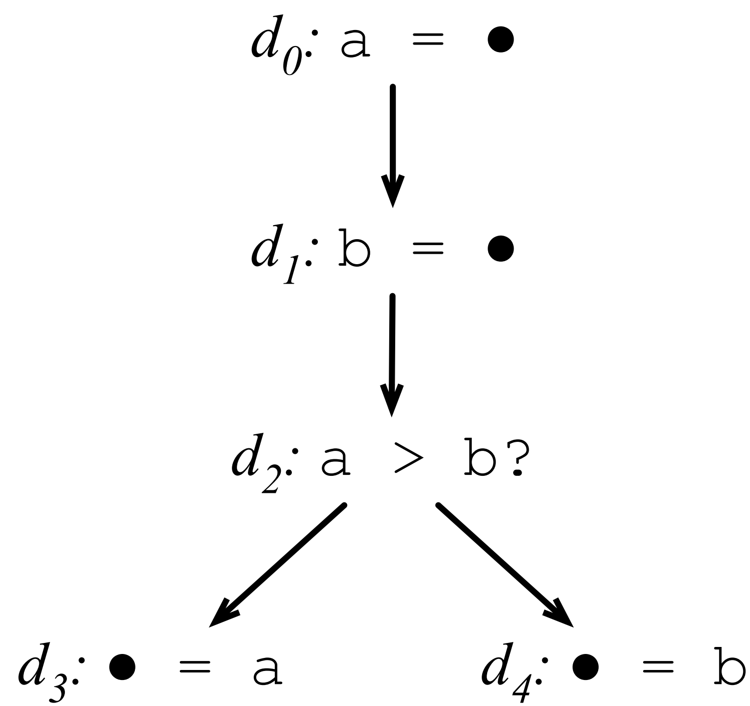
$IN[d_3] = OUT[d_3] \cup \{a\}$

$OUT[d_3] = \{\}$

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

$OUT[d_4] = \{\}$

In order to reduce the equations, Let just work with OUT sets.



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, let's just call each constraint variable x_i

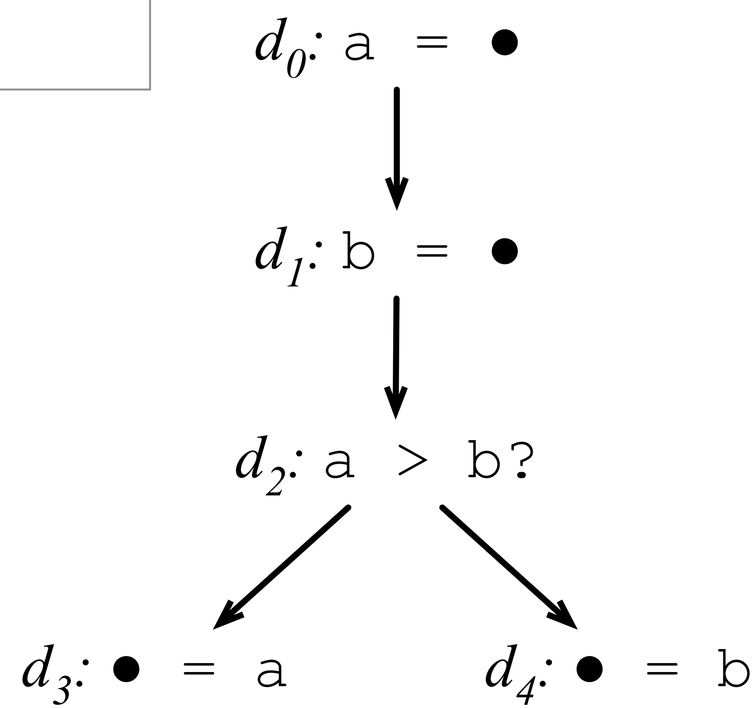
$x_0 = x_1 \setminus \{b\}$

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

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

$x_3 = \{\}$

$x_4 = \{\}$



Product Lattices

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

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

$x_3 = \{\}$

$x_4 = \{\}$

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

$$\downarrow$$

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

Dataflow analysis as solution of a constraint system

Which product lattice do we have in this equation?

Product Lattices

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

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

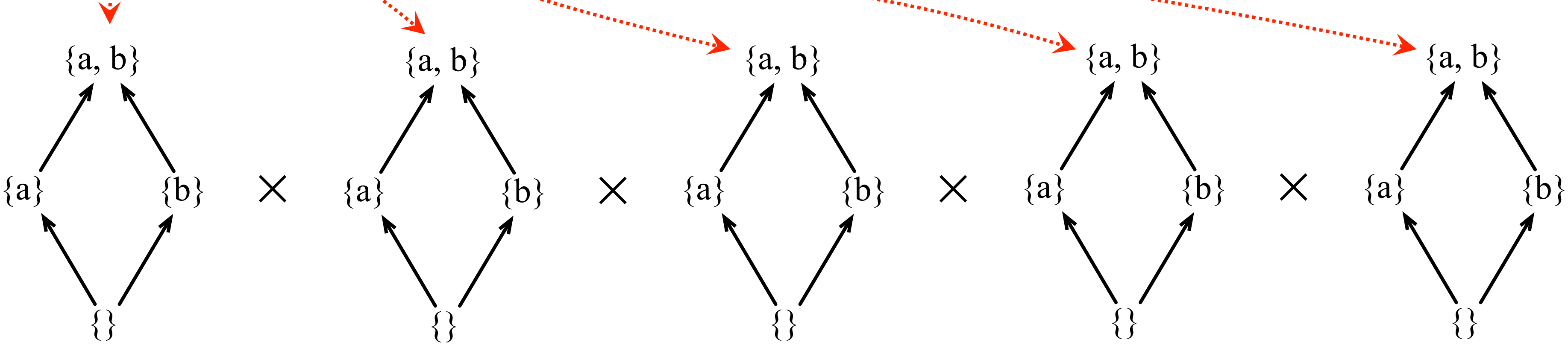
$x_3 = \{\}$

$x_4 = \{\}$

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

How many elements will our product lattice have?

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



On Partial-Redundancy Elimination

- Partial Redundancy Elimination is one of the most complex classic compiler optimizations.
  - Includes many dataflow analyses
  - Subsumes several compiler optimizations:
    - Common subexpression elimination
    - Loop invariant code motion
- Four steps "Lazy Code Motion" algorithm
  - Find blocks where evaluation of an expression can be anticipated (backwards) (*Very Busy Expressions*)
  - Check availability of expressions along all paths leading to a block needing it (forwards) (*Available Expressions)*
  - Postpone the expression as much as possible (forwards)
  - Eliminate assignments to temporaries that are used only once (backwards)

Properties of Lazy Code Motion

- Lazy Code Motion has the following properties:
  1. All redundant computations of expressions that can be eliminated without code duplication are eliminated.
  2. The optimized program does not perform any computation that is not in the original program execution.
  3. Expressions are computed at the latest possible time.
     - That is why it is called *lazy*.