

# LANGUAGE BASED SECURITY (LBT)

## LIVENESS ANALYSIS

Chiara Bodei, Gian-Luigi Ferrari

Lecture April, 4 2025



# Outline

Liveness Analysis

Liveness: Set Theory Approach

Liveness: Monotone Framework  
approach

Other analyses

# Outline

Liveness Analysis

# Liveness: previously mentioned

## Examples

**Code liveness (aka please compiler do not remove it)**

The static analysis decides if a given function is ever called at runtime:

- if “no”, remove the function from the code
- if “yes”, don’t do anything
- but the “no” answer *must* always be correct if given

False positives are ok (dead code left in)

False negatives are not ok (removing live code)

# Liveness analysis

When a variable is still “needed”,  
for a future use?

What is its **live** range?

Determine the possible **live** variables of  
a given program

Track live variables at each  
point of your program

# Liveness analysis

When a variable is still “needed”,  
for a future use?  
What is its **live** range?

Determine the possible **live** variables of  
a given program

## Applications

Compiler optimization: dead code elimination  
Register Allocation

If you have dead variables, you can  
discard them!

# Liveness analysis: uses

## Register allocation

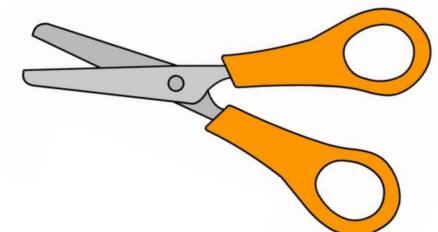
If variable  $x$  is live in a basic block, it is a potential candidate for register allocation

Two variables can share the same register if they are not live at the same time



## Dead code elimination

If variable  $x$  is dead (not live) after an assignment  $x = \dots$ , then the assignment is redundant and can be deleted as dead code



# Example 1

Are there any **dead** variables?

```
x = 2;  
x = 5;  
<< use x >>
```

Live variables: something important that you cannot remove without impacting overall behavior of program.

First assignment is useless, because first value is not used before update.

# Example1

Are there any **dead** variables?

```
x = 2;  
x = 5;  
<< use x >>
```

The value of **x** is not read (used)  
before the following assignment and  
therefore, **x** is **dead**

## Example 2

Are there any **dead** variables?

```
x = 2;  
if (y > 0) {  
    x = y+5;  
} else {  
    x = y+10;  
}  
<< use x >>
```

First assignment of x is not useful because we rewrite x regardless

## Example 2

Are there any **dead** variables?

```
x = 2;  
if (y > 0) {  
    x = y+5;  
} else {  
    x = y+10;  
}  
<< use x >>
```

The value of **x** is not read (used)  
before the following assignment and  
therefore, **x is dead**

# Example 3

Are there any **dead** variables?

```
x = 2;  
if (y > 50) {  
    x = 1;    // x def  
} else {  
    x = x+2; // x read  
}  
<< use x >>
```

The value of **x** is not read (used)  
before the following assignment and  
therefore, **x** is **dead**

# Example 3

Are there any **dead** variables?

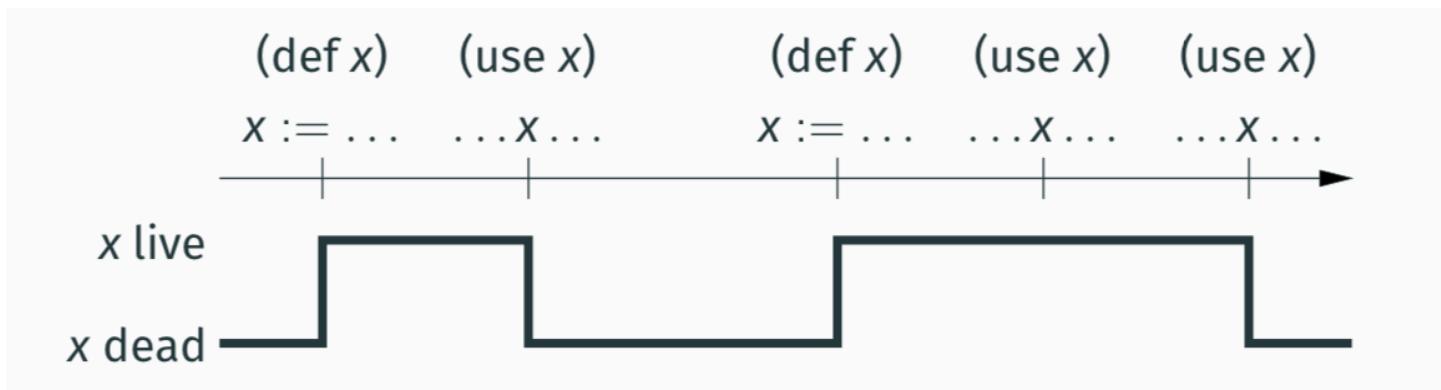
```
x = 2;  
if (y > 50) {  
    x = 1;    // x def  
} else {  
    x = x+2; // x read  
}  
<< use x >>
```

The value of **x** is read (used)  
read before one of the following  
assignments and  
therefore, **x** is **live**

# Semantic Liveness

A variable  $x$  is “semantically”

- live at a program point p IFF there exists an execution path where its value is read (used) later with no writes (re-definitions) to  $x$  in between
- dead otherwise: its value has no impact on the rest of the program execution!

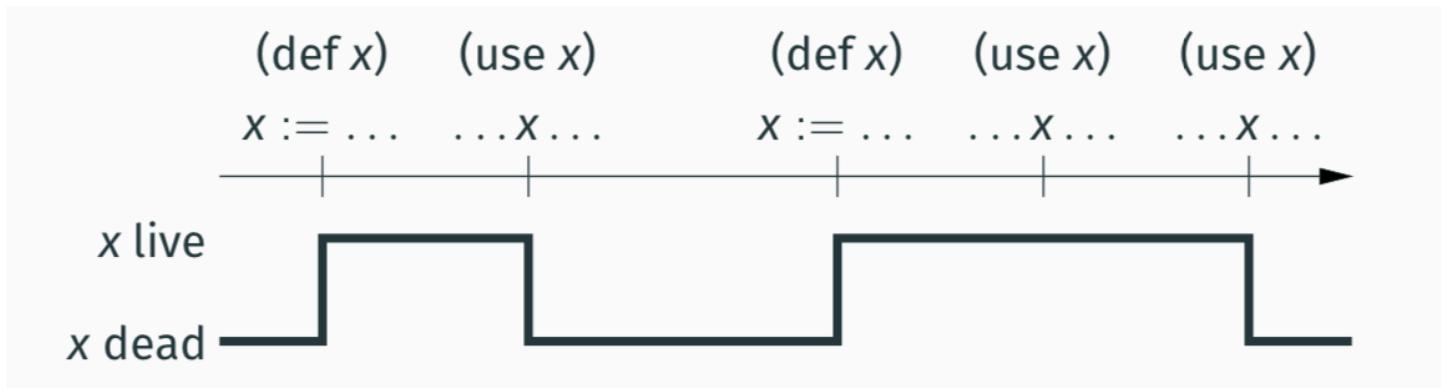


# Semantic Liveness

A variable  $x$  is “semantically”

- **live** at a program point p IFF there exists an **execution path** where its value is read (used) **later** with no writes (re-definition) to  $x$  in between
- **dead** otherwise: its value has no effect on the rest of the program execution!

**UNDECIDABLE!**



14

?  
Thus is not a trivial property!  
Undecidable. Try to approximate  
with static analysis.

# Syntactic liveness

A variable **x** is “syntactically”

- **live** at a program point p IFF if there is a **path** to the exit of the **Control Flow Graph (CFG)** along which its value may be used before it is redefined
- **dead** otherwise

not in general to  
the execution

# Syntactic liveness

A variable **x** is “**syntactically**”

- **live** at a program point p IFF if there is a **path** to the exit of the **Control Flow Graph (CFG)** along which its value may be used before it is redefined
- **dead** otherwise

## Syntactic liveness

- is **decidable** (because it is concerned with properties of the syntactic structure of the **program**)
- is a **computable approximation** of semantic liveness

# How to compute live variables

## Naïve solution

Analyse all paths from each use of each variable  
Look for each variables up and down  
to see if they are live

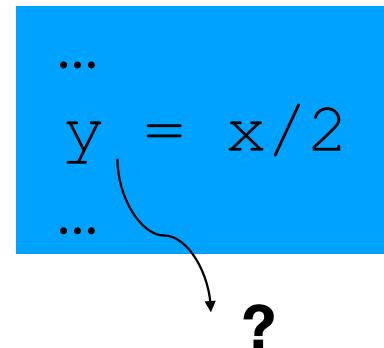
Inefficient: it explores the same paths many times

# Liveness static analysis

We focus on a **static analysis** called **liveness analysis**

**Liveness** is a **data-flow property** of variables

“Is the value of this variable needed **later**? ”



# Liveness static analysis (cont.)

We focus on a **static analysis** called **liveness analysis**

**Liveness** is a **data-flow property** of variables:

“Is the value of this variable needed **later**?”

$$y = x/2$$

If you find a path  
in which in CFG you use  $y$ ,  
you can say  $y$  is alive.

$$x = x - y$$

Suppose our analysis associates label for each node with the set of live variables at that point. What can you do? What is important is see to know what something will be used later.

19 You do a backward analysis because you are interested in what comes afterward. You cannot do that by only knowing past.

# Liveness static analysis (cont.)

We focus on a **static analysis** called **liveness analysis**

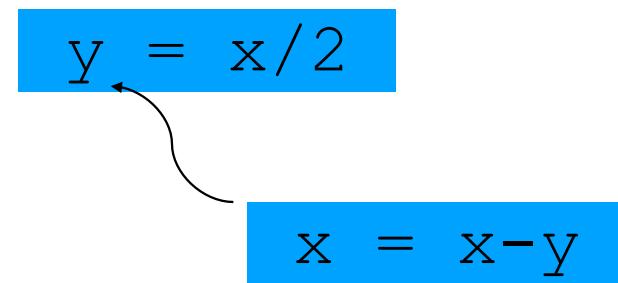
**Liveness** is a **data-flow property** of variables:  
“Is the value of this variable needed **later**?”

$$\begin{array}{c} y = x/2 \\ \swarrow \\ x = x - y \end{array}$$

# Liveness static analysis (cont.)

We focus on a **static analysis** called **liveness analysis**

**Liveness** is a **data-flow property** of variables:  
“Is the value of this variable needed **later**?”

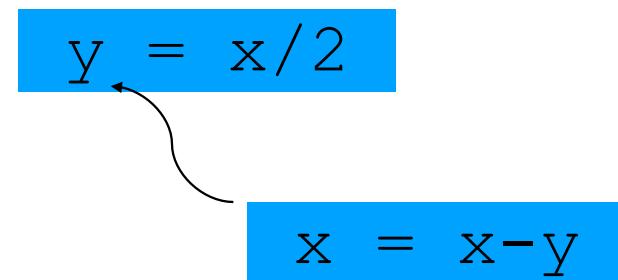


- In which direction the information propagate?

# Liveness static analysis (cont.)

We focus on a **static analysis** called **liveness analysis**

**Liveness** is a **data-flow property** of variables:  
“Is the value of this variable needed **later**?”



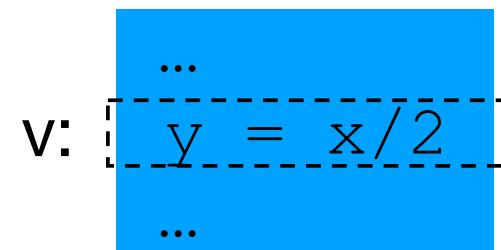
- In which direction the information propagate? **Backward**

# Liveness static analysis

- At each instruction, each variable in the program is either **live** or **dead**
- Each instruction, or node of the Control Flow Graph (CFG), has an associated set of **live variables**

## Idea

- An instruction makes a variable
  - **live** when it references (uses) it
  - **dead** when it defines (assigns to) it

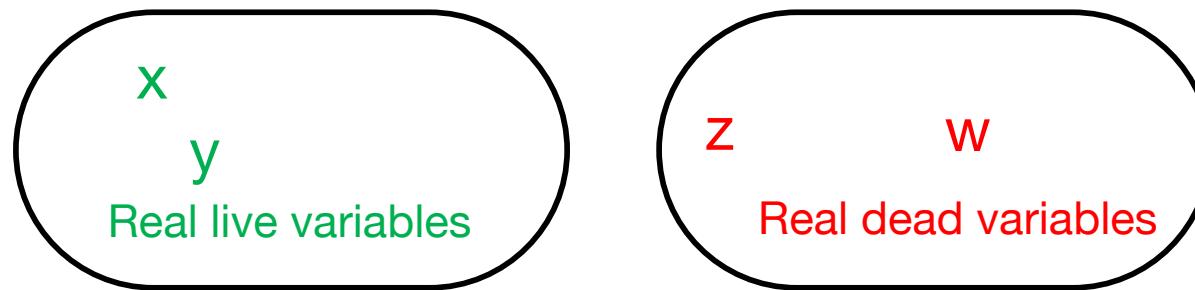


- Variables **live** at v:  $[[v]] = \{x\}$
- Variables **dead** at v:  $\{y\}$

# Liveness analysis

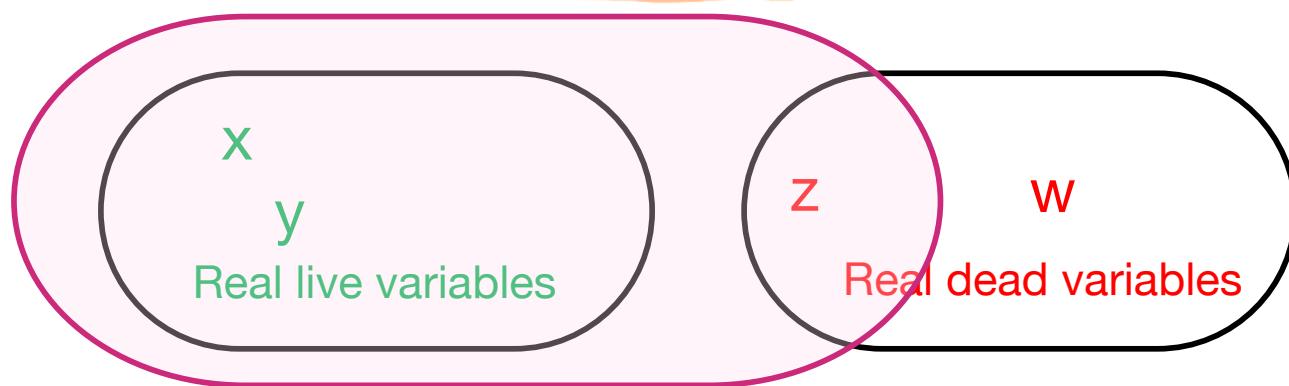
- We want a **conservative analysis**: it must only answer “dead” if the variable is **really dead**
- Optimization: no need to store the values of dead variables
- It is possible to compute a **safe over-approximation** of liveness
- It is a particular case of the previously mentioned code liveness
  - Why is this not a perfect analysis if CFG can show all executive paths?  
Because you lose something with over approximation

# Conservative analysis



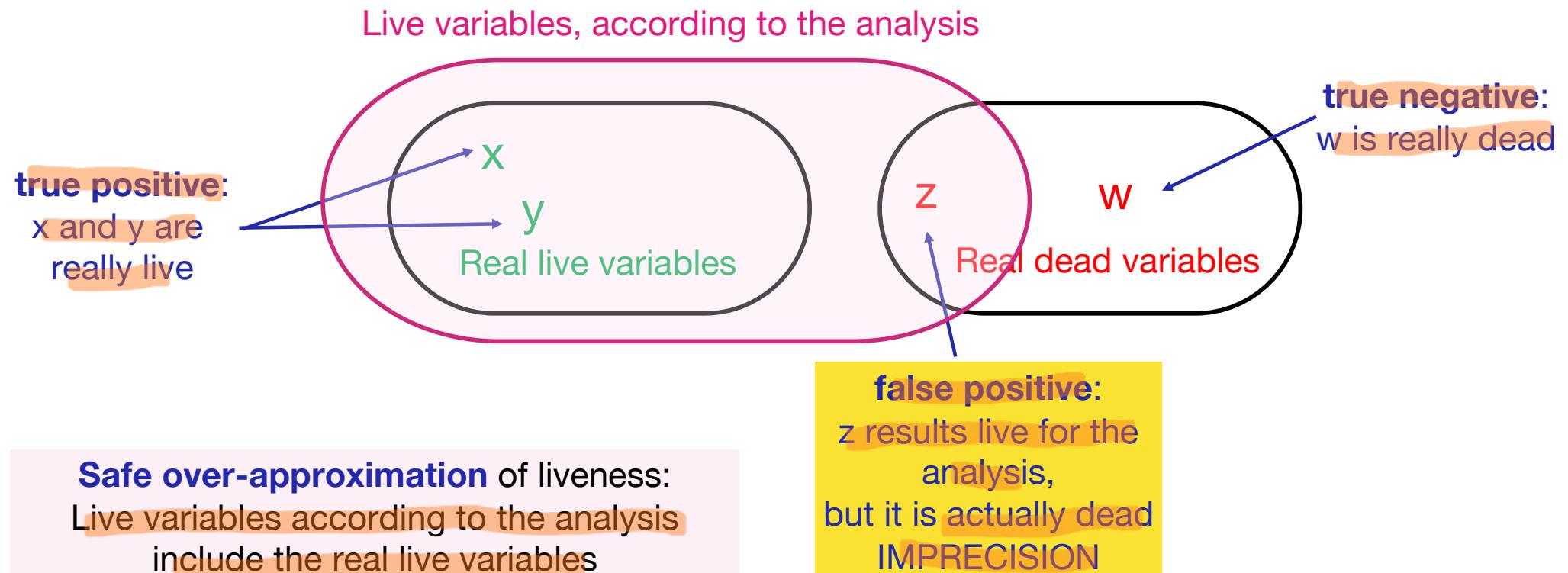
# Conservative analysis

Live variables, according to the analysis



Safe over-approximation of liveness

# Conservative analysis



# Outline

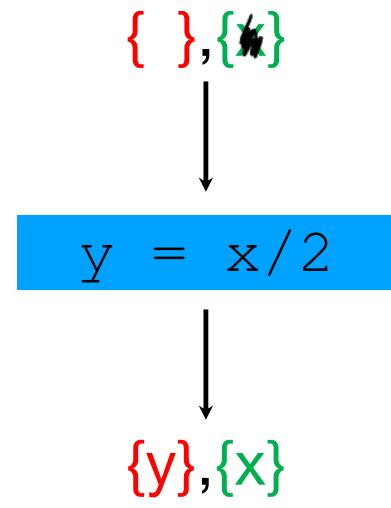
Liveness Analysis

Liveness: Set Theory Approach

# Liveness analysis: set theory approach

(Also called  
BFR Vector  
Framework)

## Intuition



- An instruction makes a variable:
- **live** when it references /uses it
  - **dead** when it defines/assigns to it

Liveness information at each node is updated by:

- adding any variables that become **live** and
- removing the variables that become **dead**
- we say that  $y = x/2$  kills  $y$  and generates  $x$

$y$  has been redefined so it is dead.

Between reassignment and first use variables are dead [?] We'll see

# Liveness analysis: set theory approach

How the statement affects the analysis state Approach

Analysis state = live variables

Sets as analysis information

Key questions:

- In which direction the information propagate? Backward

# Liveness analysis: set theory approach

How the statement affects the analysis state

Analysis state = live variables

Sets as analysis information

**Key questions:**

- In which **direction** the information propagate? **Backward** or **forward**?

# Liveness analysis: set theory approach

How the statement affects the analysis state

Analysis state = live variables

Sets as analysis information

## Key questions:

- In which **direction** the information propagate? **Backward** or **forward**?
- In which way information coming from other nodes can be **merged**?

# Liveness analysis: set theory approach

How the statement affects the analysis state

Analysis state = live variables

Sets as analysis information

## Key questions:

- In which **direction** the information propagate? **Backward** or **forward**?
- In which way information coming from other nodes can be **merged**? Using **union** or **intersection**?

# Set theory approach: local data flow

Two functions are used for **local data flow** properties

- **gen** computes live variables **generated** by a statement
- **kill** computes live variables **killed** by a statement (stop the propagation)

A statement **generates** a live variable x

- if the statement uses x, e.g.,  
 $\text{gen}(y = x/2) = \{x\}$

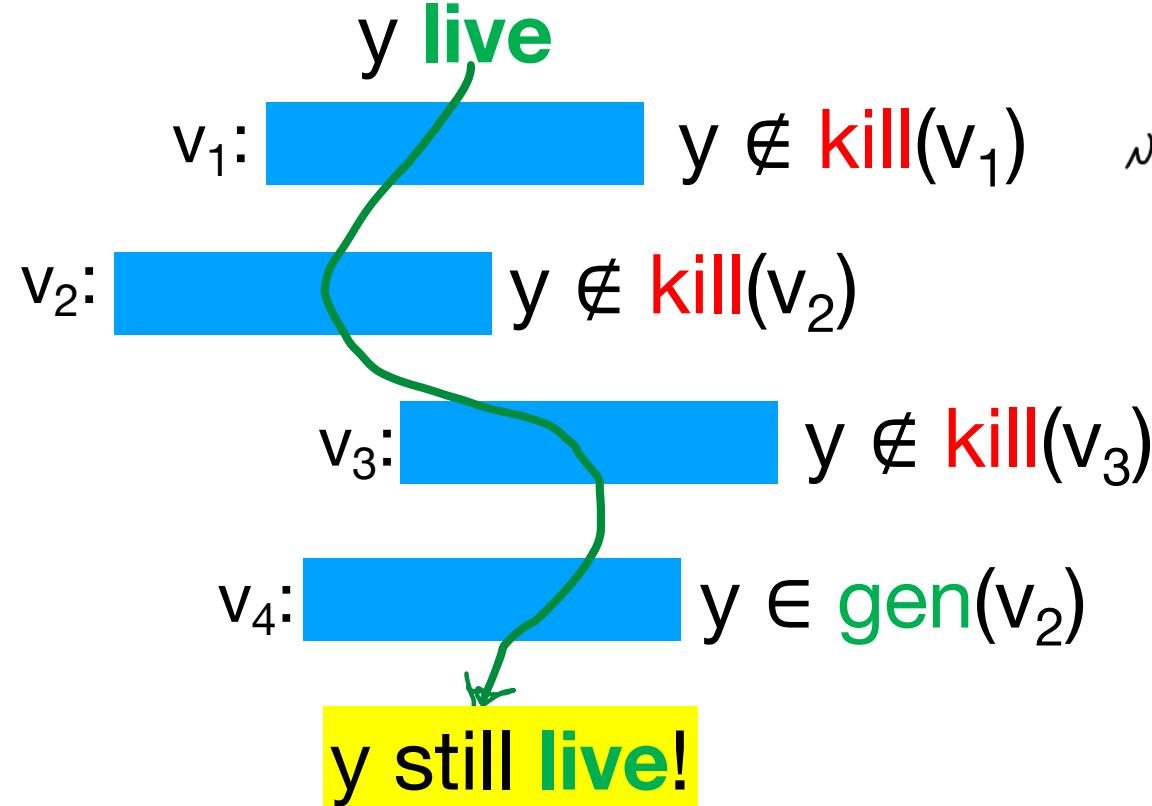
A statement **kills** a live variable x

- if the statement defines x, e.g.,  
 $\text{kill}(y = x/2) = \{y\}$

An instruction makes a variable:

- **live** when it references /uses it
- **dead** when it defines/assigns to it

# Liveness



Rough idea:  
start with  $y$  live and  
if  $y$  has not been  
redefined, it is still  
alive

# Set theory approach: gen and kill definitions

- **gen** computes live variables **generated** by a statement
- **kill** computes live variables **killed** by a statement (stop the propagation)

gen  $\llbracket \text{var } x_1, \dots, x_n \rrbracket = \emptyset$

gen  $\llbracket \text{entry} \rrbracket = \llbracket \text{exit} \rrbracket = \emptyset$

gen  $\llbracket \text{if } (E) \rrbracket = \llbracket \text{while } (E) \rrbracket = \llbracket \text{output } E \rrbracket = \bigcup \text{vars}(E)$  (variables occurring in E)

gen  $\llbracket x = E \rrbracket = \text{vars}(E)$  generate every variable mentioned in expression

kill  $\llbracket \text{var } x_1, \dots, x_n \rrbracket = \{x_1, \dots, x_n\}$

kill  $\llbracket \text{entry} \rrbracket = \llbracket \text{exit} \rrbracket = \emptyset$

kill  $\llbracket \text{if } (E) \rrbracket = \llbracket \text{while } (E) \rrbracket = \llbracket \text{output } E \rrbracket = \emptyset$

kill  $\llbracket x = E \rrbracket = \{x\}$

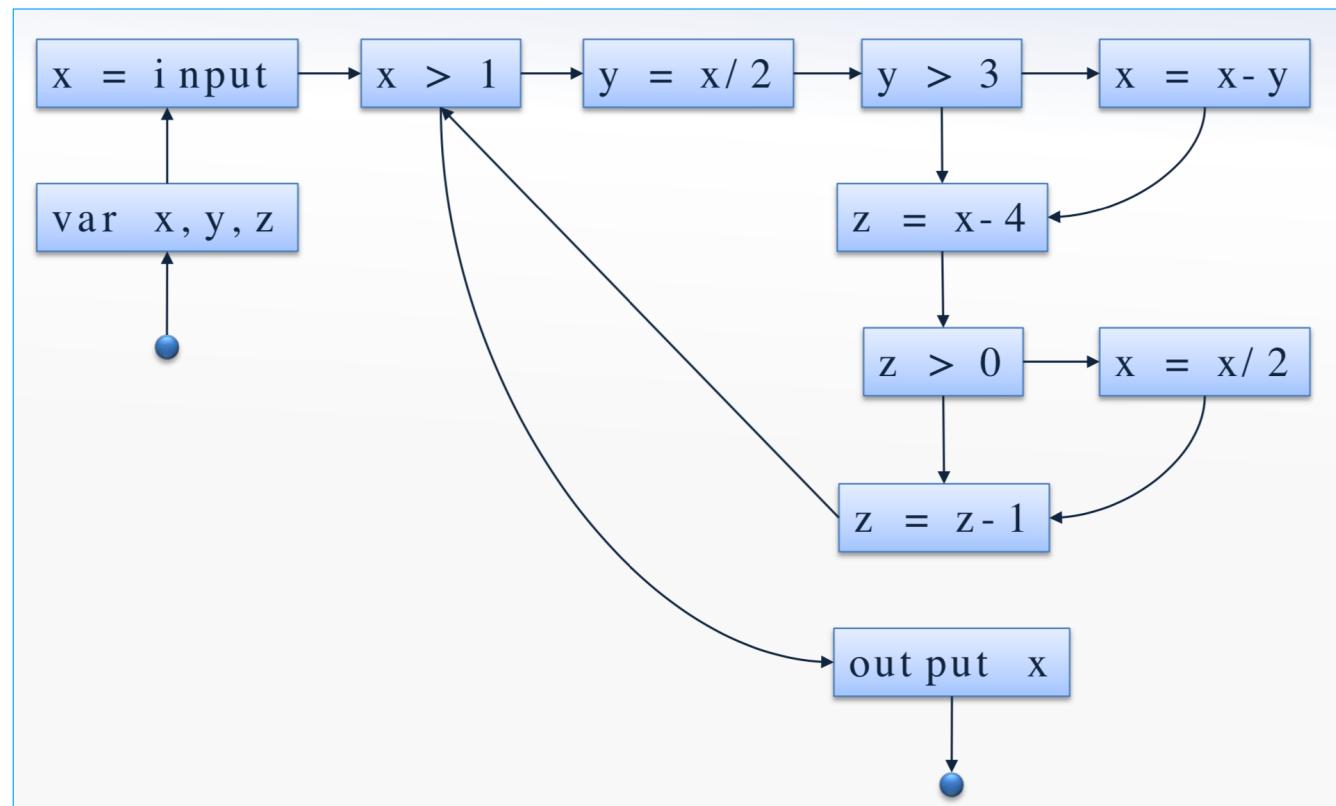
If you do  $y = y + 1$ , via this definition

gen  $\llbracket y = y + 1 \rrbracket = \{y\}$

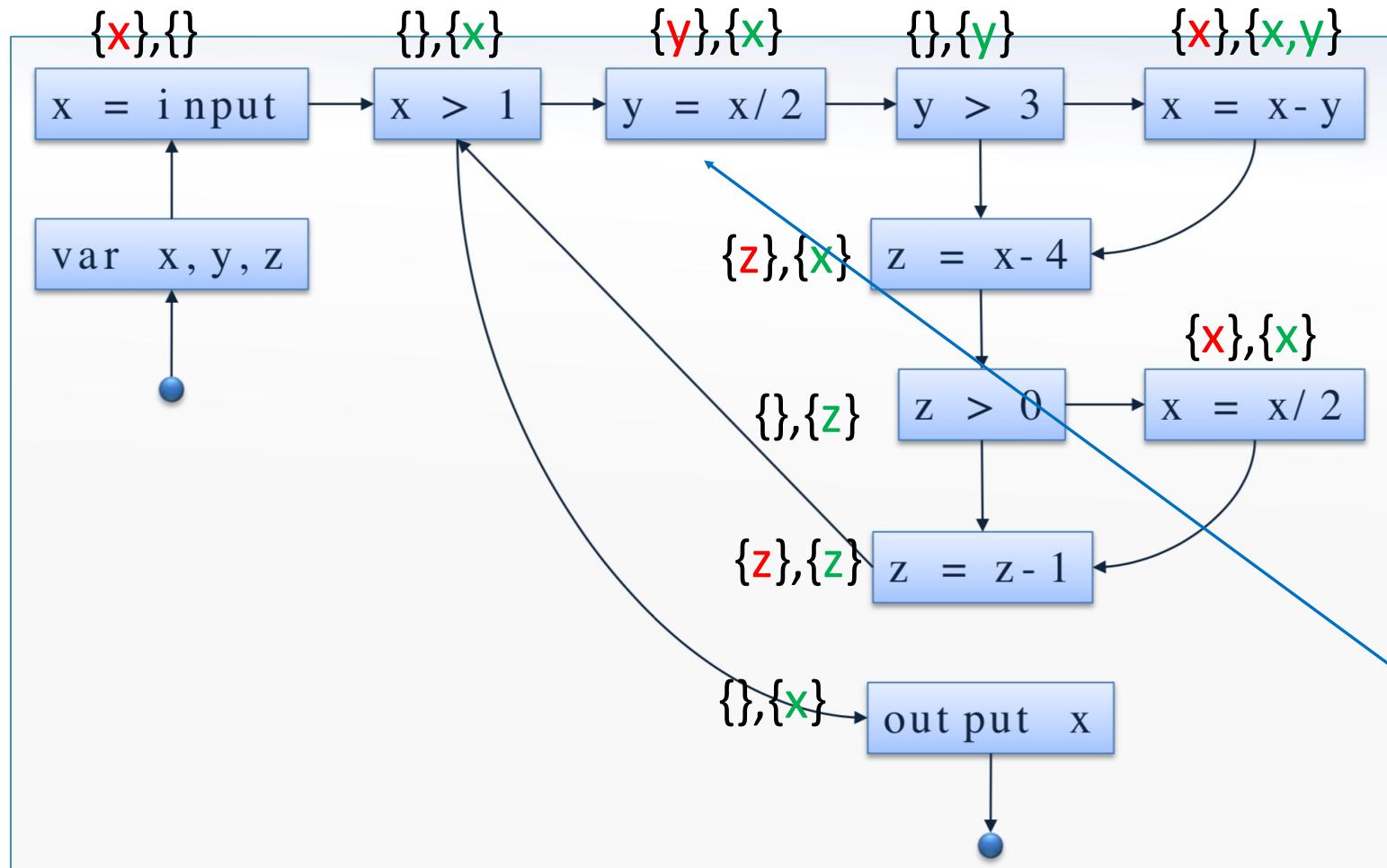
kill  $\llbracket y = y + 1 \rrbracket = \{y\}$

# Our running example

```
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;
```



# Liveness analysis: annotated CFG



An instruction makes a variable:

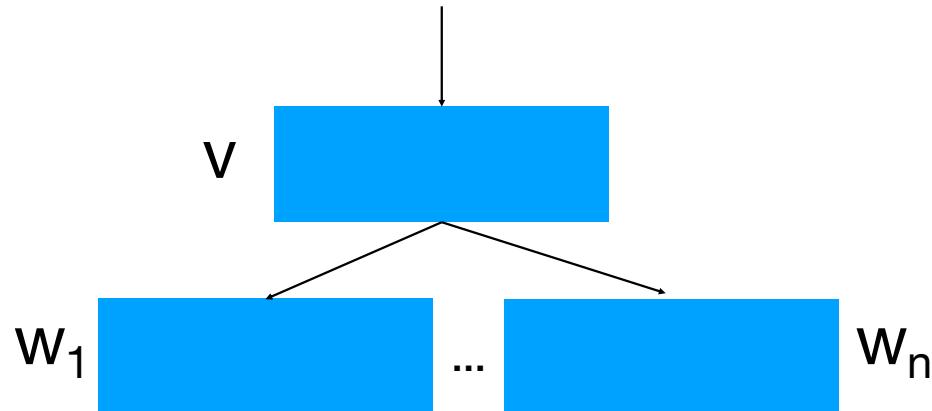
- **live** when it references /uses it
- **dead** when it defines/assigns to it
- **$y = x/2$  kills y and generates x**

# Set theory approach: global data flow

## Global data flow properties

We need to know which are the sets of live variables at the entry and at the exit of a node  $v$  and how they flow through the graph

- How do we combine information coming from other nodes?



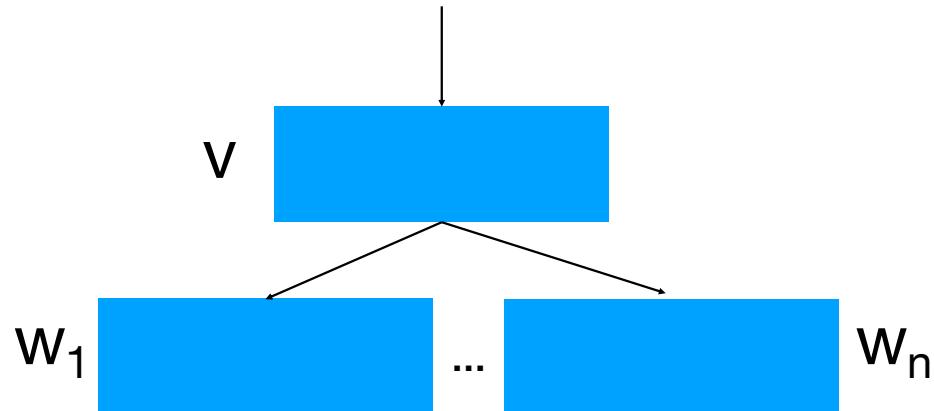
↳ Note that this is a backwards flow.  
Take the Union: if at least in one part there is something, you cannot forget it.

# Set theory approach: global data flow

## Global data flow properties

We need to know which are the sets of live variables at the entry and at the exit of a node  $v$  and how they flow through the graph

- How do we combine information coming from other nodes?



### Key questions:

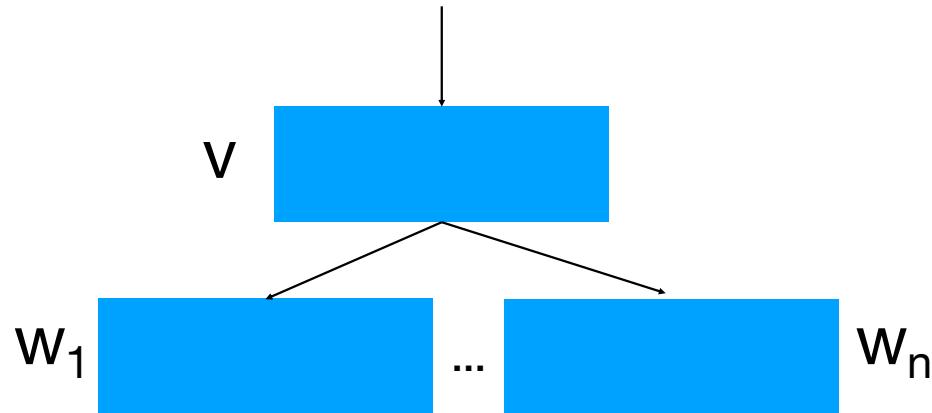
- In which **direction** the information propagate?

# Set theory approach: global data flow

## Global data flow properties

We need to know which are the sets of live variables at the entry and at the exit of a node  $v$  and how they flow through the graph

- How do we combine information coming from other nodes?



### Key questions:

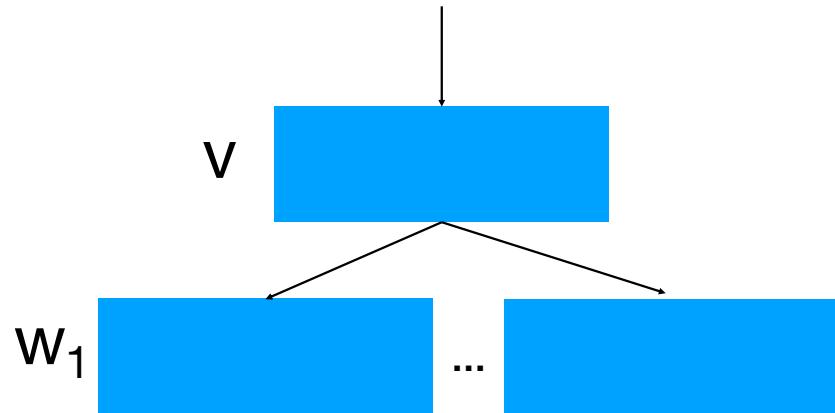
- In which **direction** the information propagate?  
**Backward**

# Set theory approach: global data flow

## Global data flow properties

We need to know which are the sets of live variables at the entry and at the exit of a node  $v$  and how they flow through the graph

- How do we combine information coming from other nodes?



### Key questions:

- In which **direction** the information propagate?  
**Backward**

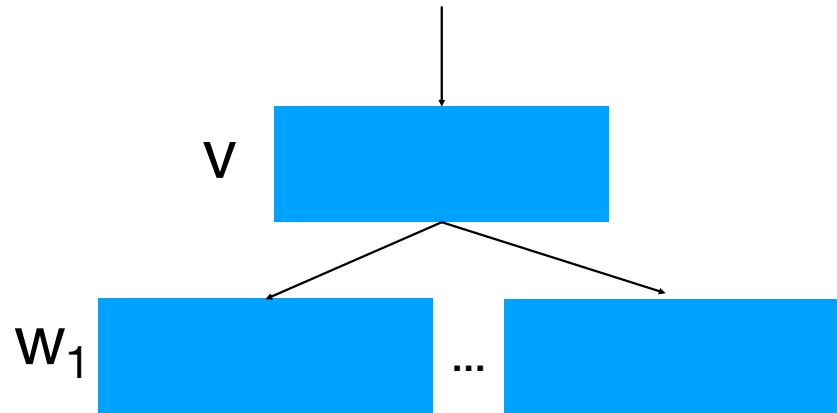
- $w_n$  - In which way information coming from other nodes can be **merged**?

# Set theory approach: global data flow

## Global data flow properties

We need to know which are the sets of live variables at the entry and at the exit of a node  $v$  and how they flow through the graph

- How do we combine information coming from other nodes?



### Key questions:

- In which **direction** the information propagate?  
**Backward**

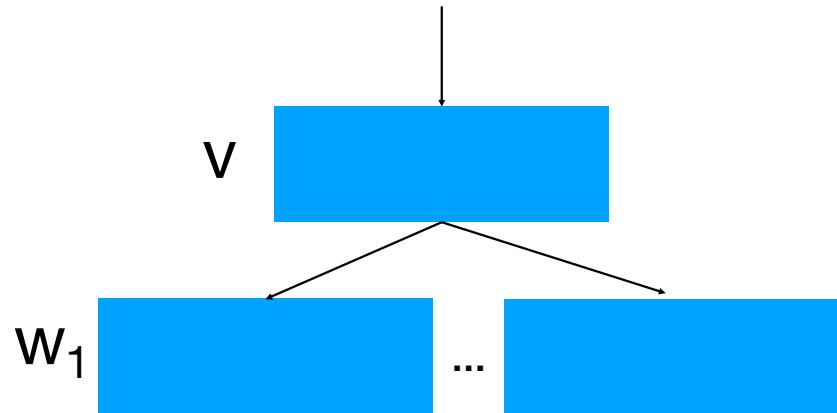
$w_1 \dots w_n$  - In which way information coming from other nodes can be **merged**?  
**Union**

# Set theory approach: global data flow

## Global data flow properties

We need to know which are the sets of live variables at the entry and at the exit of a node  $v$  and how they flow through the graph

- How do we combine information coming from other nodes?



**Key questions:**

- In which **direction** the information propagate?  
**Backward**

w<sub>1</sub> ... w<sub>n</sub> - In which way information coming from other nodes can be **merged**?  
**Union**

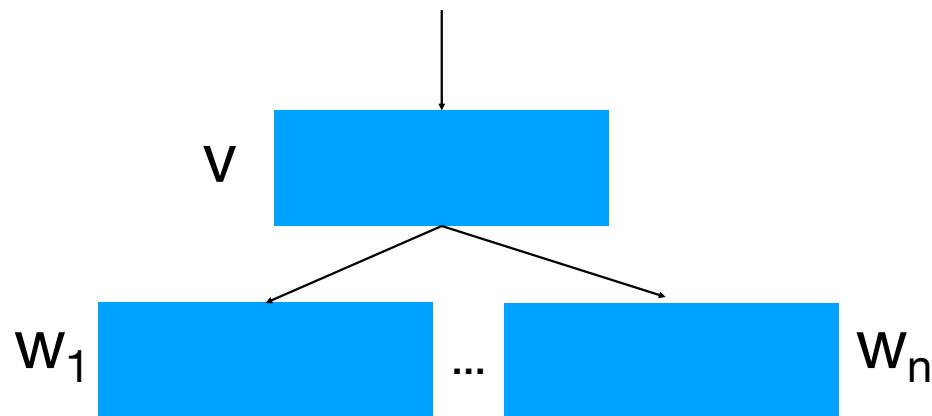
- This is because a variable is **live** if it **may** be used **later** (at least in a path)

↳ There is at least one path in which we use it!

# Set theory approach: flow equations

Given outgoing variables, compute ingoing ones

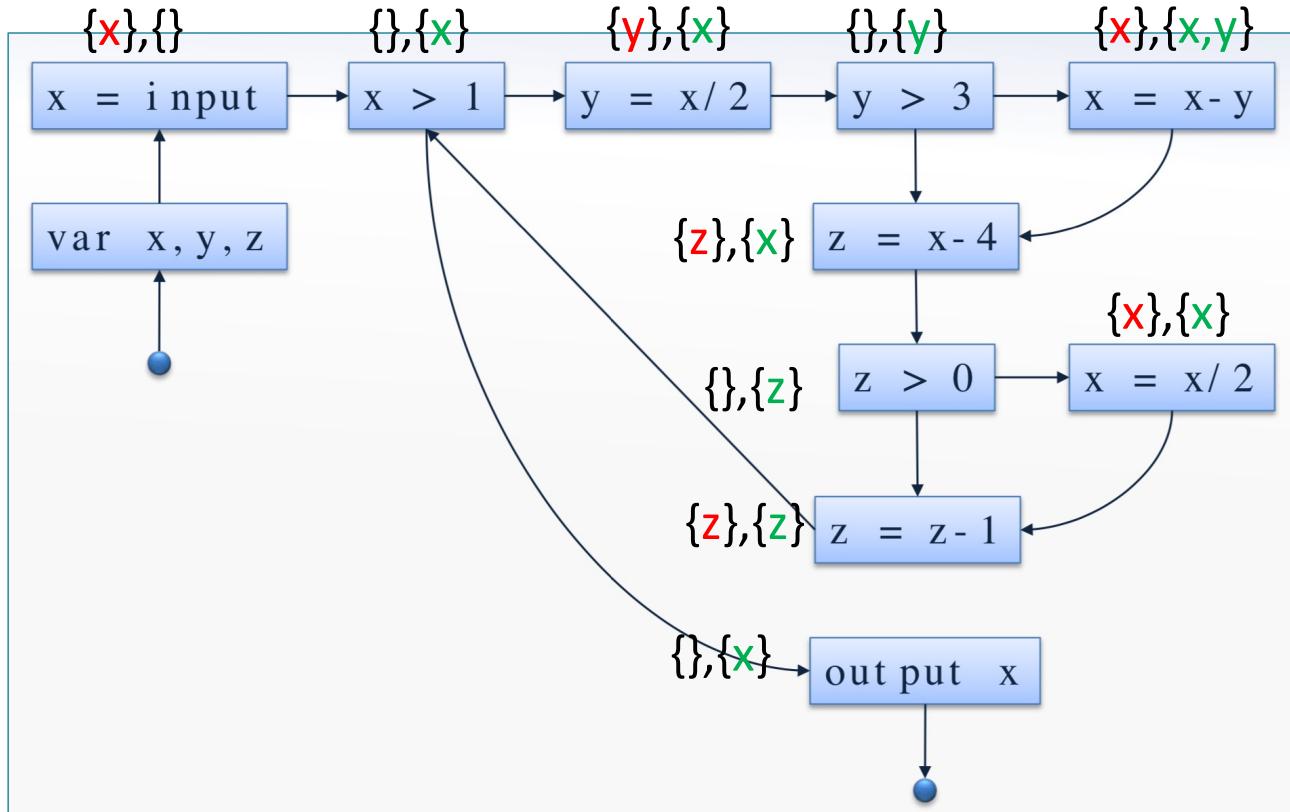
- $\text{Out}(v) = \bigcup_{w \in \text{succ}(v)} \text{In}(w)$
- $\text{In}(v) = \text{Gen}(v) \cup (\text{Out}(v) \setminus \text{Kill}(v))$



# Liveness analysis: set theory approach

- Initially: no live variables ( $\emptyset$ )
- Backward analysis: propagate live variables in the opposite direction of control flow
- Compute ingoing and outgoing live variables for each node  $v$
- When control flow splits, propagate live variables both ways
- May analysis: when control flows merge, joins with union the incoming live variables
- Solve the equations, iterating until fix-point is reached

# Liveness analysis: annotated CFG



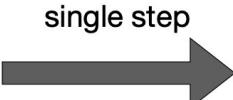
$\text{OUT}([\text{var } x, y, z]) = \text{IN}([\text{var } x, y, z]) = \emptyset$   
 $\text{OUT}([x=\text{input}]) = \text{IN}([x>1])$   
... (TO BE CONTINUED)

# Beyond states: Constant Propagation analysis

**Goal:** determine for each node whether or not a variable has a **constant** value whenever execution reaches that point (So we don't have to recompute it)

**Optimisation:** replacing uses of a variable with its value if that value is guaranteed to be constant

```
let
  var a : int := 0
  var b : int := a + 1
in
  c := c + b;
  a := 2 * b
end
```



```
let
  var a : int := 0
  var b : int := 0 + 1
in
  c := c + b;
  a := 2 * b
end
```



```
let
  var a : int := 0
  var b : int := 0 + 1
in
  c := c + 1;
  a := 2 * 1
end
```

# Beyond states: Constant Propagation analysis

```
let
  var a : int := 0
  var b : int := a + 1
in
  c := c + b;
  a := 2 * b
end
```

single step

```
let
  var a : int := 0
  var b : int := 0 + 1
in
  c := c + b;
  a := 2 * b
end
```

full propagation

```
let
  var a : int := 0
  var b : int := 0 + 1
in
  c := c + 1;
  a := 2 * 1
end
```

**Goal:** determine for each node whether or not a variable has a constant value whenever execution reaches that point

let	a $\mapsto$ 0
var a : int := 0	a $\mapsto$ 0, b $\mapsto$ 1
var b : int := a + 1	a $\mapsto$ 0, b $\mapsto$ 1, c $\mapsto$ ?
in	a $\mapsto$ 2, b $\mapsto$ 1, c $\mapsto$ ?
c := c + b;	
a := 2 * b	
end	

Kill/gen do not work here: we need the previous information to compute the current one

Can we use a set for this map? Keys map to single values, so no  
But what if we keep multiple values? Analysing loops may not terminate

# Beyond states: Constant Propagation analysis

But what if we keep **multiple values**? Analysing loops may not terminate  
The (possibly) infinite kill set makes this impractical

<b>let</b>		
<b>var</b> a : int := 0	a $\mapsto$ 0	
<b>var</b> b : int := a + 1	a $\mapsto$ 0; b $\mapsto$ 1	
<b>in</b>		
<b>while</b> y > a + b <b>do</b>	a $\mapsto$ 0; b $\mapsto$ 1	a $\mapsto$ 0,1; b $\mapsto$ 1
a := a + 1	a $\mapsto$ 0,1; b $\mapsto$ 1	a $\mapsto$ 0,1,2; b $\mapsto$ 1
<b>end</b>		

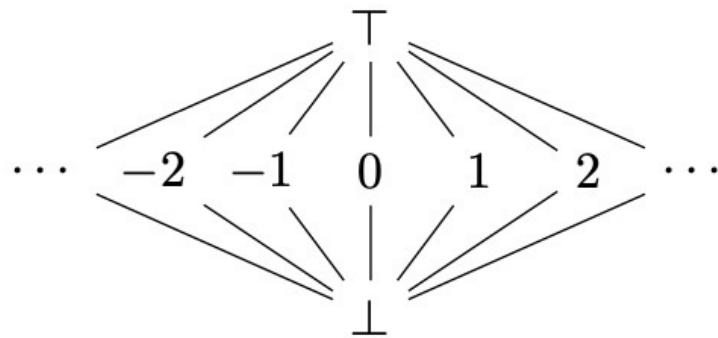
There is impracticality  
and we need to use  
monotone FW

**Monotone framework** to the rescue!

# Constant Propagation analysis: Monotone framework

It is a **forward** flow problem

Similar to sign analysis



Ym khs FW we  
consider n vegans

$$a \hat{op} b = \begin{cases} \perp & \text{if } a = \perp \text{ or } b = \perp \\ \top & \text{otherwise, if } a = \top \text{ or } b = \top \\ a op b & \text{if } a \in \mathbb{Z} \text{ and } b \in \mathbb{Z} \end{cases}$$

...

# Outline

Liveness Analysis

Liveness: Set Theory Approach

Liveness: Monotone Framework  
approach

# Dataflow Analysis

**Dataflow analysis** is a static program analysis technique used to infer properties about program variables (or other program elements) at various points in the control flow

The reasoning is about data, and dependencies between data

It systematically tracks how information “flows” through the program based on its structure, without actually executing the code

# Dataflow Analysis

**Dataflow analysis** is a static program analysis technique used to infer properties about program variables (or other program elements) at various points in the control flow

The reasoning is about data, and dependencies between data

It systematically tracks how information “flows” through the program based on its structure, without actually executing the code

**Monotone framework:** combination of

- a **complete lattice** and
  - a **space of monotone functions**
- } This is enough to find solutions for our properties

# Monotone Framework: Core Concepts

## Starting Point

- **CFG** (Control Flow Graph) of the program: analysis information is propagated along CFG edges
- A complete **lattice** of finite height, that describes abstract information to infer at each node (it may be parameterized by the program)

↳ Provides struct. of program

# Monotone Framework: Core Concepts

## Starting Point

- **CFG** (Control Flow Graph) of the program: analysis information is propagated along CFG edges
- A complete **lattice** of finite height, that describes abstract information to infer at each node (it may be parameterized by the program)

## Constraint Variables

- Each CFG **node**  $v$  (representing each program point) gets a **variable**  $[[v]]$
- Values range over lattice elements

# Monotone Framework: Core Concepts

## Starting Point

- **CFG** (Control Flow Graph) of the program: analysis information is propagated along CFG edges
- A complete **lattice** of finite height, that describes abstract information to infer at each node (it may be parameterized by the program)

## Constraint Variables

- Each CFG **node**  $v$  (representing each program point) gets a **variable**  $[[v]]$
- Values range over lattice elements

## Dataflow Constraints: generation and solving

- **Constraints** relate values of a node to its neighbors, defining how each statement impacts the analysis state
- Based on language constructs represented in the CFG

# Monotone Framework: Core Concepts

## Starting Point

- **CFG** (Control Flow Graph) of the program: analysis information is propagated along CFG edges
- A complete **lattice** of finite height, that describes abstract information to infer at each node (it may be parameterized by the program)

## Constraint Variables

- Each CFG **node**  $v$  (representing each program point) gets a **variable**  $[[v]]$
- Values range over lattice elements

## Dataflow Constraints: generation and solving

- **Constraints** relate values of a node to its neighbors, defining how each statement impacts the analysis state
- Based on language constructs represented in the CFG

## Fixed-Point Computation

- If constraints are **equations** with monotone RHS, we can use **fixed-point algorithm** that iterate until fix-point is reached: this yields the **least solution** (most precise)

# Monotone Framework approach

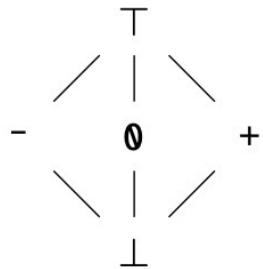
For a given program to be analyzed, a monotone framework can be instantiated by specifying:

- the CFG and
- the rules for assigning dataflow constraints to its nodes

An analysis is **sound** if all solutions to the constraints correspond to correct information about the program

# Sign Analysis in the Monotone Framework

- Lattice



- States =  $Vars \rightarrow Sign$
- For each CFG node  $v$  we assign a constraint variable  $[[v]]$  denoting an abstract state that gives the sign values for all variables at the program point immediately after  $v$
- The dataflow constraints model the effects of program execution on the abstract states
- Auxiliary function that combines the abstract states from the predecessors of a node  $v$ :

$$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} [[w]]$$

← set of all predecessors of node  $v$

$X = E$

$$[[v]] = \text{JOIN}(v)[X \mapsto \text{eval}(\text{JOIN}(v), E)]$$

eval computes the sign of the expression

# Equations

A program with  $n$  CFG nodes,  $v_1, \dots, v_n$ , is thus represented by  $n$  equations:

- $[[v_1]] = afv_1([[v_1]], \dots, [[v_n]])$
- $[[v_2]] = afv_2([[v_1]], \dots, [[v_n]])$
- ...
- $[[v_n]] = afv_n([[v_1]], \dots, [[v_n]])$

in general we need  
a constraint for each  
node

where  $afv: States^n \rightarrow States$  for each CFG node  $v$

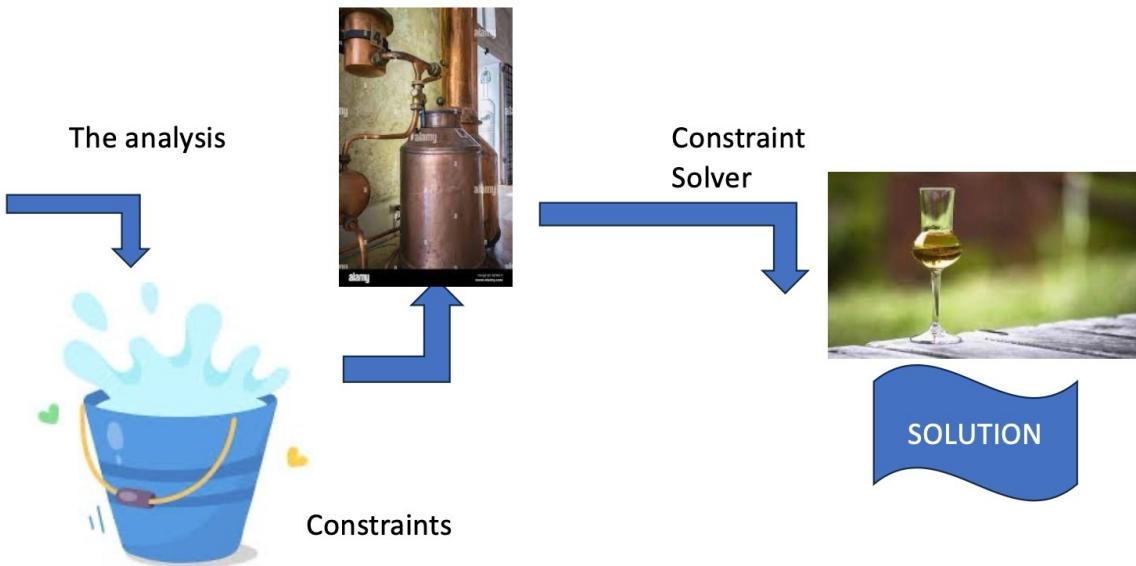
The lattice and the constraints form a **monotone framework**

Note that all the right-hand sides of our constraints correspond to monotone function

The basic idea is to separate the analysis specification from algorithmic aspects and implementation details

```
1 #include <iostream>
2 using namespace std;
3
4 void bubblesort(int arr[])
5 {
6     for (int i = 0; i < 5; i++) {
7         for (int j = 0; j < (5 - i - 1); j++) {
8             if (arr[j] > arr[j + 1]) {
9                 int temp = arr[j];
10                arr[j] = arr[j + 1];
11                arr[j + 1] = temp;
12            }
13        }
14    }
15
16    int max() {
17        int myarray[5];
18        cout << "Enter 5 integers in any orders " << endl;
19        for (int i = 0; i < 5; i++) {
20            cin >> myarray[i];
21        }
22        cout << "Before Sorting" << endl;
23        for (int i = 0; i < 5; i++) {
24            cout << myarray[i] << " ";
25        }
26
27        bubblesort(myarray); // sorting
28
29        cout << endl << "After Sorting" << endl;
30        for (int i = 0; i < 5; i++) {
31            cout << myarray[i] << " ";
32        }
33    }
34
35    return 0;
36}
37
```

Program to analyze



Use powerset of sets  
of variables you have  
at your disposal  
as live/alive

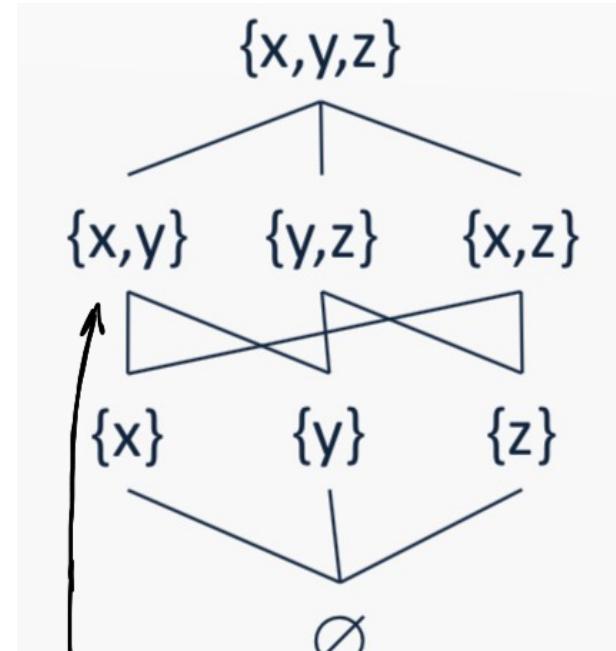
## Liveness analysis

- This **data-flow analysis** determines the set of potentially **live variables in each point** of a given program
- Applications: **remove dead stores and dead code**

# Liveness analysis: the lattice

The **lattice** modelling our **abstract states** is  $(\wp(\{\text{Var}\}); \sqsubseteq)$  with **Var**, variables in the program (**parameterized lattice**). Again, a powerset for **live variables**

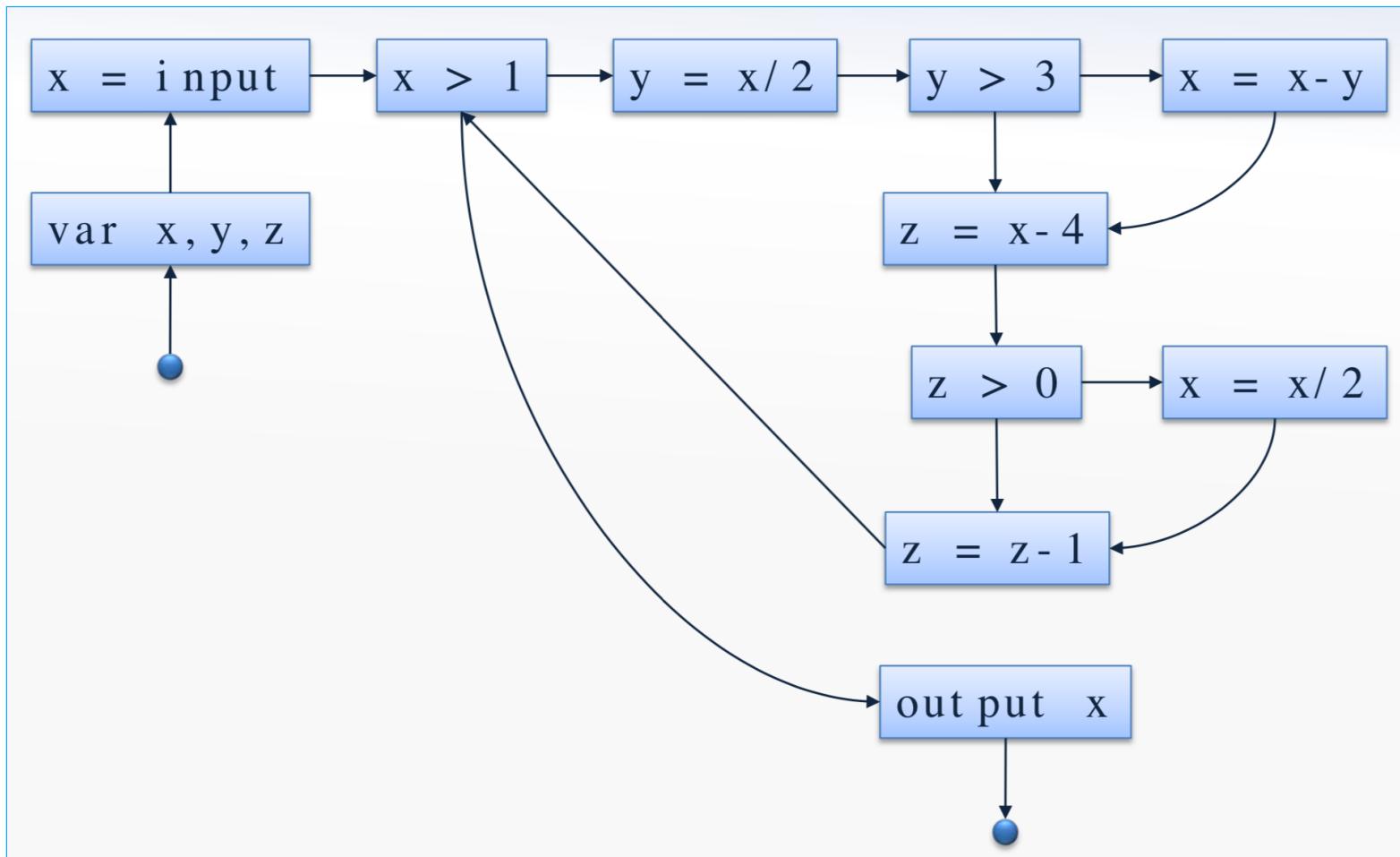
```
var x,y,z;  
x = input;  
while (x>1) {  
    y = x/2;  
    if (y>3) x = x-y;  
    z = x-4;  
    if (z>0) x = x/2;  
    z = z-1;  
}  
output x;
```



69

Variables in this state are live.  
They correspond to Set approach 110 (x,y,z)

# Liveness analysis: the Control Flow Graph



# Setting up the monotone framework

For every node  $v$  of the control flow graph we have a variable yielding the set of variables that are live at the program point after  $v$

$$[v]$$

the set of expressions that are live at the program before  $v$

- In which direction the information propagate?

• Potentially live,  
MAY ANALYSIS

# Setting up the monotone framework

For every node  $v$  of the control flow graph we have a variable yielding the set of variables that are live at the program point after  $v$

$\llbracket v \rrbracket$

the set of expressions that are **live** at the program before  $v$

- In which **direction** the information propagate? **Backward**

# Setting up the monotone framework

For every node  $v$  of the control flow graph we have a variable yielding the set of variables that are live at the program point after  $v$

$\llbracket v \rrbracket$

the set of expressions that are **live** at the program before  $v$

- In which **direction** the information propagate? **Backward**
- In which way information coming from other nodes can be **merged**?

This is because a variable is live if it **may** be used **later** (at least in one path)

# Setting up the monotone framework

For every node  $v$  of the control flow graph we have a variable yielding the set of variables that are live at the program point after  $v$

$\llbracket v \rrbracket$

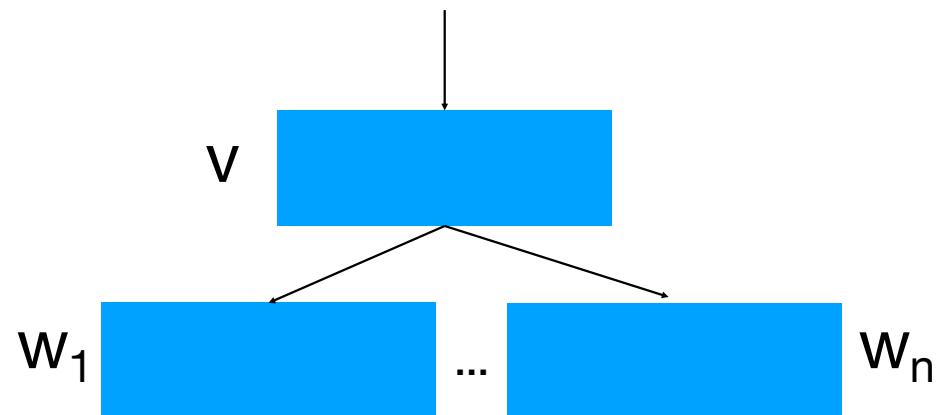
the set of expressions that are **live** at the program before  $v$

- In which **direction** the information propagate? **Backward**
- In which way information coming from other nodes can be **merged**? **Union**

This is because a variable is live if it **may** be used **later** (at least in one path)

# JOIN

$$\text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} [w]$$



# Termination

The set of all variables is finite, and

- the JOIN operation (i.e., meet) is union, hence
- the set associated with a data flow variable can only grow

This implies that termination is guaranteed (Think of monad framework. To make it understandable, by how we defined it, we never actually shrink ( $\set{x} \cup \set{x}$ , so that's true))

As initial value is  $\emptyset$ , live variables analysis converges on the smallest set

- Union : things always grow and you are going to converge
- we know thanks to theorems that we get to least fixed point

## Why the Set Only Grows (Monotonicity)

In backward static analysis of liveness:

- We start with the empty set at the end of the control flow.
- At each program point, we compute the live variables set by **taking the union of the successors**.
- If a statement **uses** a variable, that variable must be added to the live set.
- If a statement **defines** (kills) a variable, it is removed from the set.

Now, the key part: since we are solving this as a **fixed-point iteration**, the set of live variables at each program point **only grows across iterations**. The reason is:

1. Initially, every point starts with the empty set.
2. At each iteration, we update the live variable set by taking the **union** of the successors' sets.
3. Since union **only adds elements**, the set never shrinks between iterations.
4. Eventually, we reach a fixed point where further iterations do not change the set.

## Why Killing a Variable Doesn't Contradict Growth

You're right that a variable may be removed (killed) from the live set at a particular program point, but that happens **within one evaluation of a statement**. However, in the fixed-point computation:

- The next iteration could bring back that variable if it appears in the union of a successor's live set.
- Since every iteration only takes unions, we never "undo" a previous addition in the iterative process.

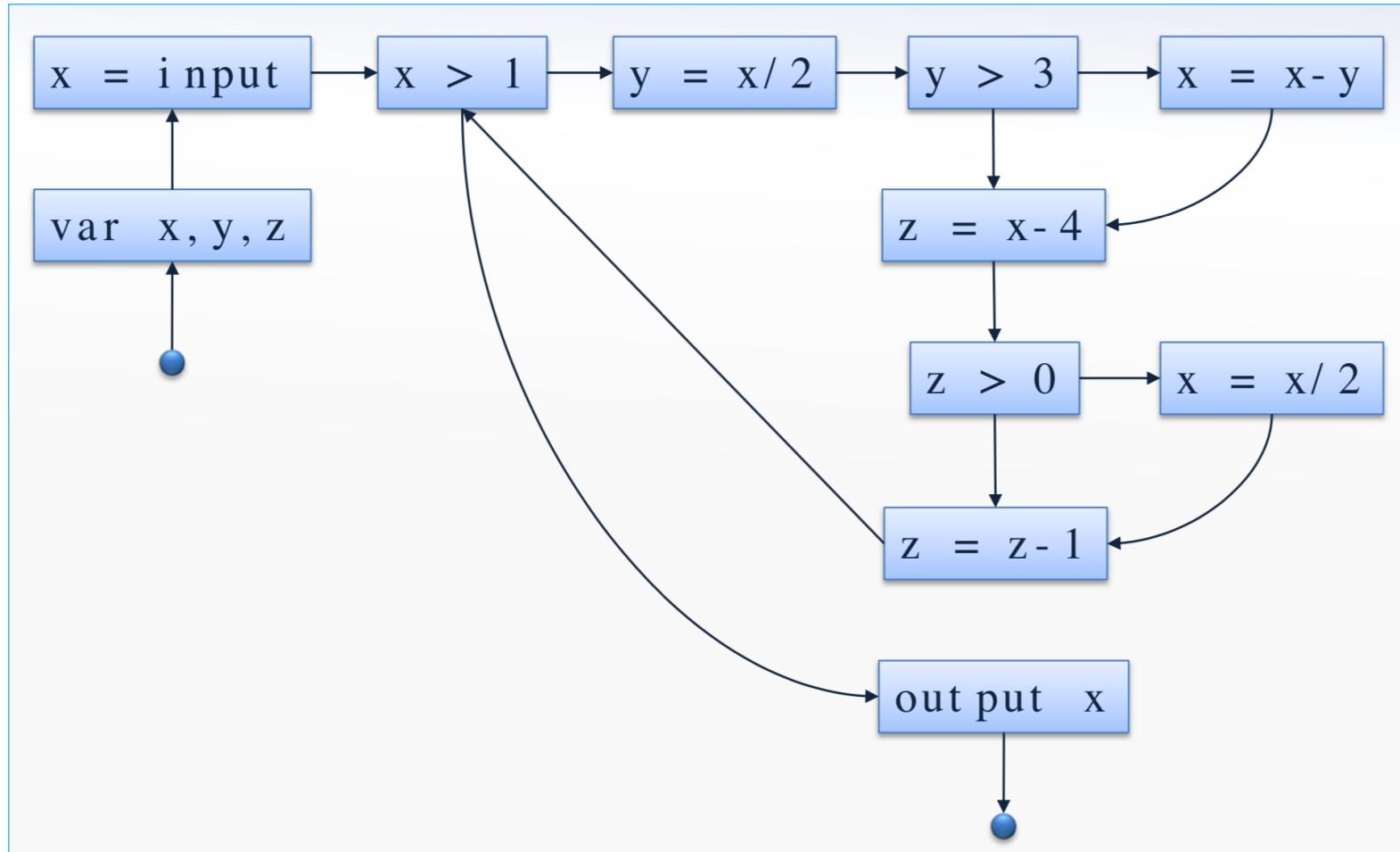
## Why This Guarantees Termination

1. The number of variables in the program is **finite**.
2. The powerset lattice (the structure we are working in) has a **finite height** (at most  $2^n$  elements, where  $n$  is the number of variables).
3. The iteration process **only increases sets** and thus moves in one direction (upward in the lattice).
4. Since we can only add elements (never remove from a previous iteration), we must eventually reach a fixed point where further updates do not change anything.

Thus, termination is **guaranteed** because the iterative process cannot continue indefinitely—it must stabilize at some point.



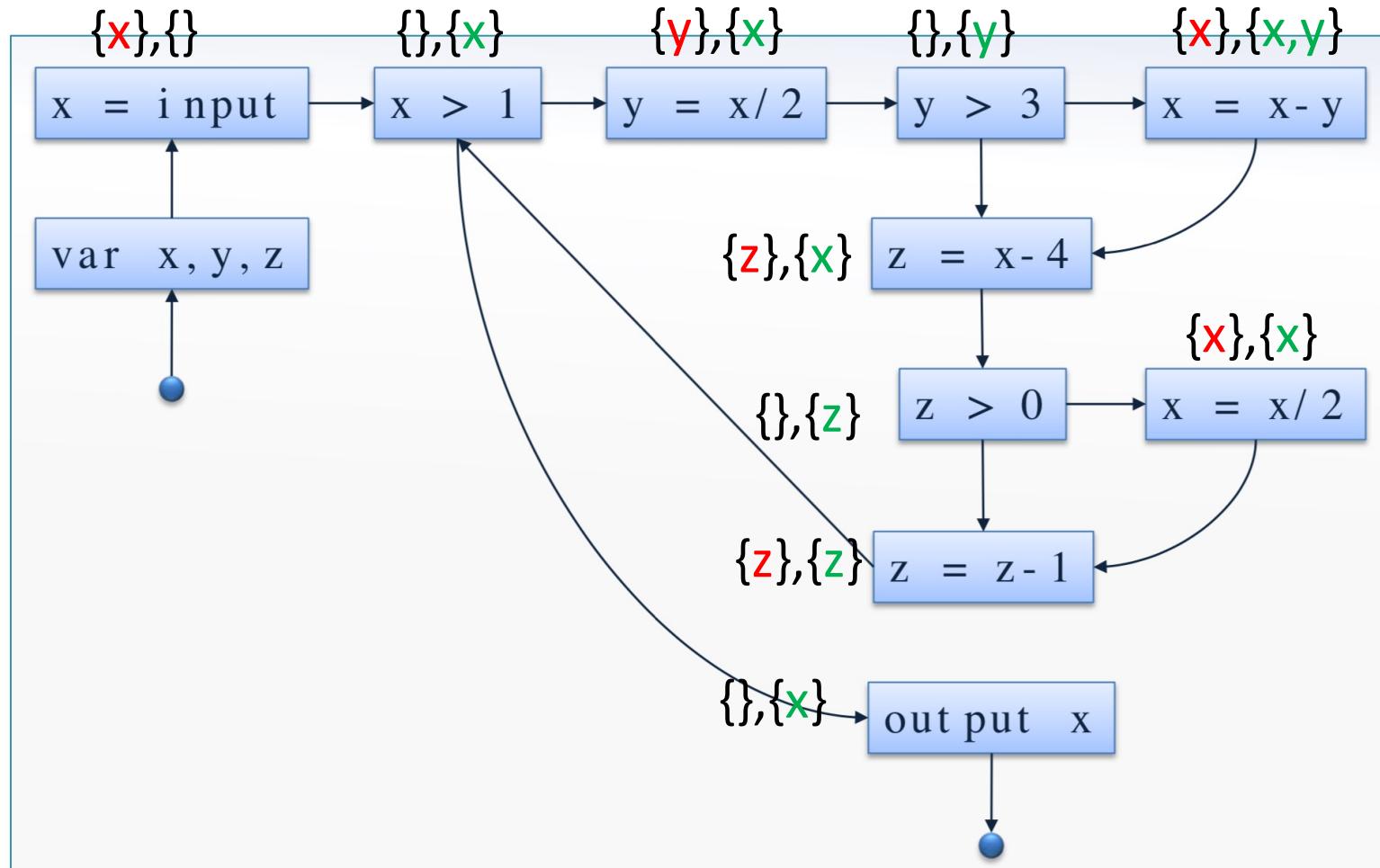
# Liveness analysis: annotated CFG



An instruction makes a variable:

- **live** when it references /uses it
- **dead** when it defines/assigns to it

# Liveness analysis: constraint variables



- For every node  $v$
- $[[v]]$  include the variables that are **live** at the program point before  $v$
  - Liveness flows **backwards** through the CFG

# Live Variables Analysis: dataflow equations

- Dataflow equations (transfer functions) tell us how liveness information is propagated
- For the variables declaration node

$$[\![\text{var } x_1, \dots, x_n]\!] = \text{JOIN}(v) \setminus \{x_1, \dots, x_n\}$$

- For the entry node

$$[\![\text{entry}]\!] = [\![\text{exit}]\!] = \emptyset$$

$$\text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} [\![w]\!]$$

- For conditions, loops and outputs

$$[\![\text{if (E)}]\!] = [\![\text{while (E)}]\!] = [\![\text{output E}]\!] = \text{JOIN}(v) \cup \text{vars}(E) \quad (\text{vars occurring in E})$$

- For assignment

$$[\![x = E]\!] = \left( \text{JOIN}(v) \setminus \{x\} \right) \cup \text{vars}(E)$$

*influence*

killed variable      generated variables

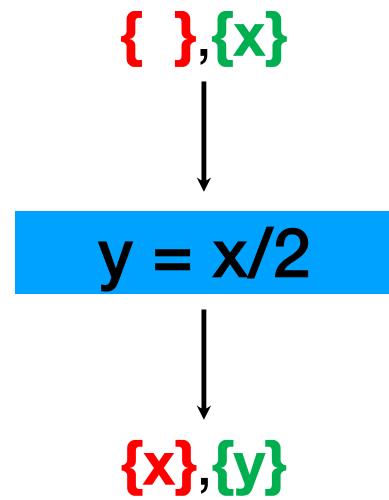
- For any other nodes

$$[\![v]\!] = \text{JOIN}(v)$$

Is this monotone?

where right-handsides are monotone

# Liveness analysis: annotated CFG


$$\llbracket y = x/2 \rrbracket = \text{JOIN}(v) \setminus \{x\} \cup \{y\} = \\ \bigcup_{w \in \text{succ}(v)} \llbracket w \rrbracket = \llbracket y > 3 \rrbracket \setminus \{x\} \cup \{y\}$$

- An instruction makes a variable:
- live when it references /uses it
  - dead when it defines/assigns to it
  - $y = x/2$  kills y and generates x

$$[\text{var } x, y, z] = [x = \text{input}] \setminus \{x, y, z\}$$

$$[x = \text{input}] = [x > 1] \setminus \{x\}$$

$$[x > 1] = ([y = x/2] \cup [\text{output } x]) \cup \{x\}$$

$$[y = x/2] = ([y > 3] \setminus \{y\}) \cup \{x\}$$

$$[y > 3] = ([x = x - y] \cup [z = x - 4]) \cup \{y\}$$

$$[x = x - y] = ([z = x - 4] \setminus \{x\}) \cup \{x, y\}$$

$$[z = x - 4] = ([z > 0] \setminus \{z\}) \cup \{x\}$$

$$[z > 0] = ([x = x/2] \cup [z = z - 1]) \cup \{z\}$$

$$[x = x/2] = ([z = z - 1] \setminus \{x\}) \cup \{x\}$$

$$[z = z - 1] = ([x > 1] \setminus \{z\}) \cup \{z\}$$

$$[\text{output } x] = [\text{exit.}] \cup \{x\}$$

$$[\text{exit.}] = \emptyset$$

Do not forget output and exit!

First iteration is of all empty sets as states

$$\textcircled{2} \quad [\text{Var } x, y, z] = \emptyset$$

$$[x = \text{input}] = \emptyset$$

$$[x > 1] = \{x\}$$

$$[y = x/2] = \{x\}$$

$$[y > 3] = \{y\}$$

$$[x = x - y] = \{x, y\}$$

$$[z = x - 1] = \{x\}$$

$$[z > 0] = \{z\}$$

$$[x = x/2] = \{x\}$$

$$[z = z - 1] = \{z\}$$

$$[\text{output } x] = \{x\}$$

$$[\text{exit}] = \emptyset$$

$$\textcircled{3} \quad [\text{Var } x, y, z] = \emptyset$$

$$[x = \text{input}] = \emptyset$$

$$[x > 1] = \{x\}$$

$$[y = x/2] = \{x\}$$

$$[y > 3] = \{x, y\}$$

$$[z = x - y] = \{x, y\}$$

$$[z = x - 1] = \{x\}$$

$$[z > 0] = \{x, z\}$$

$$[x = x/2] = \{z, x\}$$

$$[z = z - 1] = \{z, x\}$$

$$[\text{output } z] = \{x\}$$

$$[\text{exit}] = \emptyset$$

$$\textcircled{4} \quad [\text{Var } x, y, z] = \emptyset$$

$$[x = \text{input}] = \emptyset$$

$$[x > 1] = \{x\}$$

$$[y = x/2] = \{x\}$$

$$[y > 3] = \{x, y\}$$

$$[x = x - y] = \{x, y\}$$

$$[z = x - 1] = \{x\}$$

$$[z > 0] = \{z, x\}$$

$$[x = x/2] = \{z, x\}$$

$$[z = z - 1] = \{z, x\}$$

$$[\text{output } x] = \{x\}$$

$$[\text{exit}] = \emptyset$$

NOTE:  $[\text{empty}] = \emptyset$



We can make conclusion: y, z are never live at the same time, and z is not read after the assignment  $z = z - 1$ . It can be skipped. You can re

## Example optimization

- y and z are never live at the same time  $\Rightarrow$  they can share the same variable location
- the value assigned in  $z = z - 1$  is never read (used) after the assignment  $\Rightarrow$  the assignment can be skipped

```
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;
```

112

```
var x,yz;
x = input;
while (x>1) {
    yz = x/2;
    if (yz>3) x = x-yz;
    yz = x-4;
    if (yz>0) x = x/2;
    /* z = z-1; */
}
output x;
```

Note that there might be consequences: imagine you do  
key = 0. To wipe secrets. But if assignment is removed, that is bad! We  
need to find balance

# Equations

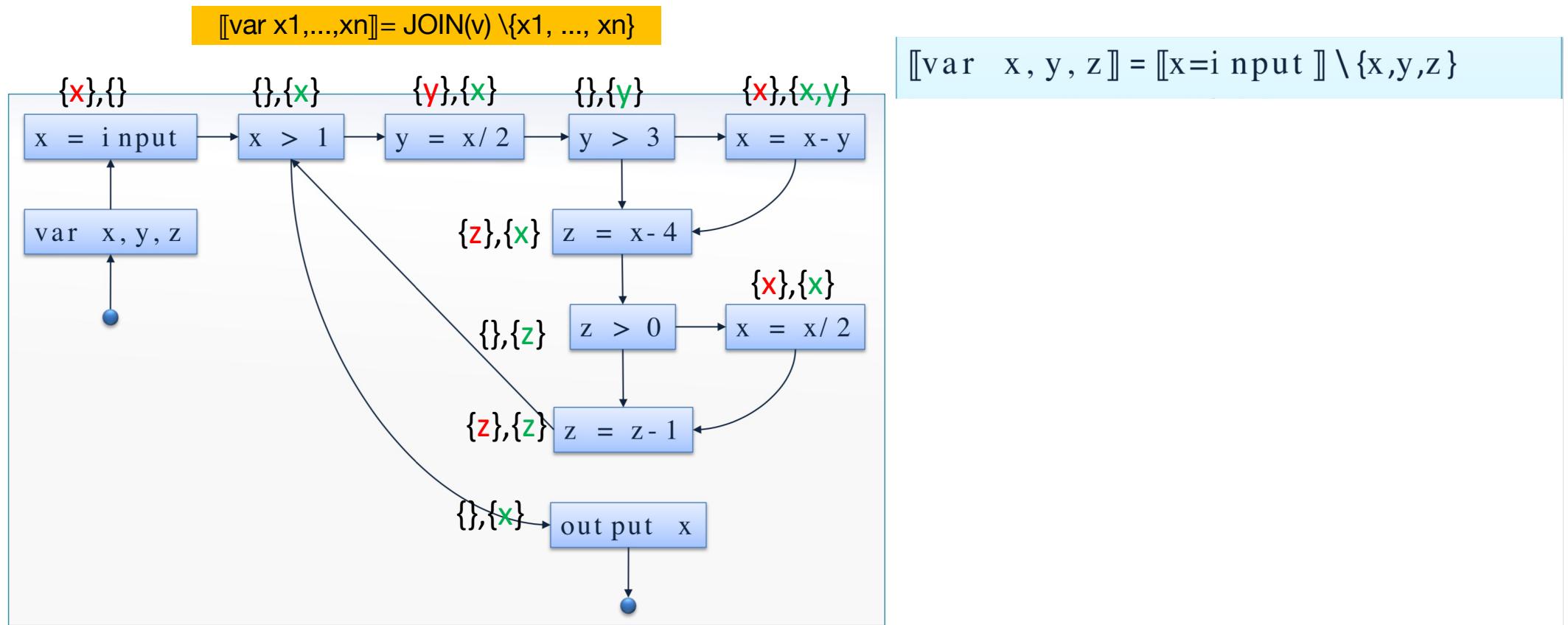
A program with  $n$  CFG nodes,  $v_1, \dots, v_n$ , is thus represented by  $n$  equations:

- $[[v_1]] = afv_1([[v_1]], \dots, [[v_n]])$
- $[[v_2]] = afv_2([[v_1]], \dots, [[v_n]])$
- ...
- $[[v_n]] = afv_n([[v_1]], \dots, [[v_n]])$

where  $afv: States^n \rightarrow States$  for each CFG node  $v$

Again, the lattice  $(\wp(\{Var\}); \subseteq)$  and the constraints extracted, using previously defined rules, form a **monotone framework**

# Back to our example



# Live Variables Analysis: algorithm

- Analysis: computing the set of live variables for each node  $n$  in the CFG such that the liveness data-flow equations are satisfied
- A simple way to solve the data-flow equations is to adopt an iterative strategy, starting from setting  $[[v]] = \emptyset$  for every node
- This algorithm is guaranteed to terminate since there are a finite number of variables in each program and the effect of one iteration is monotonic

```
Naïve algorithm
x = (⊥, ⊥, ..., ⊥);
do {
    t = x;
    x = f(x)
    while (x ≠ t)
```

# Live Variables Analysis: algorithm

- Analysis: computing the set of live variables for each node  $n$  in the CFG such that the liveness data-flow equations are satisfied
- A simple way to solve the data-flow equations is to adopt an iterative strategy, starting from setting  $[[v]] = \emptyset$  for every node
- This algorithm is guaranteed to terminate since there are a finite number of variables in each program and the effect of one iteration is monotonic

```
Naïve algorithm
x = (⊥, ⊥, ..., ⊥);
do {
    t = x;
    x = f(x)
  while (x ≠ t)
```

# Example optimization

- y and z are never live at the same time  $\Rightarrow$  they can share the same variable location
- the value assigned in  $z=z-1$  is never read (used) after the assignment  $\Rightarrow$  the assignment can be skipped

```
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;
```

```
var x,yz;
x = input;
while (x>1) {
    yz = x/2;
    if (yz>3) x = x-yz;
    yz = x-4;
    if (yz>0) x = x/2;
    /* z = z-1; */
}
output x;
```

# Live Variables Analysis: algorithm

- We can also use the **worklist** algorithm
- The **worklist W** contains nodes from the program graph (the ones we need to reconsider)
- The auxiliary function **dep** that gives the set of nodes w where v occurs on the right-hand side of the constraint for w.
- **dep v** contains the set of nodes that depend on v
- All the nodes provided by dep are inspected
- The analysis assignment will be updated if required: if this happens the target nodes are added to **W**

```
x1 = ⊥; ... xn = ⊥;  
W = {v1, ..., vn};  
while (W ≠ ∅) {  
    vi = W.removeNext();  
    y = fi(x1, ..., xn);  
    if (y ≠ xi) {  
        for (vj ∈ dep(vi)) W.add(vj);  
        xi = y;  
    }  
}
```

# Live Variables Analysis: recap

- Data-flow analysis collects information about how data moves through a program
- Variable liveness is a data-flow property
- Liveness variable analysis is a backwards dataflow analysis for determining variable liveness
- LVA may be expressed in terms of data-flow equations
- A simple iterative algorithm can be used to find the smallest solution to the LVA data-flow equations

# Types of Dataflow Analysis: back and forth

We distinguish between **forward** and **backward** analyses based on the **direction** in which information propagates through the control flow graph (CFG)

A **forward** analysis gathers information about what has happened before a program point. It starts at the entry node and moves forward through the CFG. The constraints at each node depend only on its predecessors. Classic examples include:

- **Sign analysis:** tracks the sign of variables
- **Available expressions:** determines which expressions have already been computed and not invalidated

A **backward** analysis collects information about what must hold after a program point. It starts at the exit node and proceeds backward through the CFG. The constraints here depend on successors. Examples include:

- **Live variable analysis:** tracks which variables will be used in the future

# Types of Dataflow Analysis: must and may

We distinguish between **may** and **must** analyses based, for analyses that are based on a powerset lattices

A **may** analysis aims to find **what may possibly be true** at a program point. It computes an **over-approximation** of the actual behavior and uses the lattice order  $\subseteq$  and union ( $\cup$ ) to combine information. These analyses are typically used when missing a possibility could lead to incorrect behavior. Examples include:

- Live variables

Property holds in **all paths**

A **must** analysis focuses on **what must definitely be true**. It computes an **under-approximation**, using the lattice order  $\supseteq$  and intersection ( $\cap$ ) to combine information. This kind of analysis is used when precision about guaranteed properties is crucial. Examples include:

- Available expressions

Property holds in **some paths**

# Outline

Liveness Analysis

Liveness: Set Theory Approach

Liveness: Monotone Framework  
approach

Other analyses

# Very busy expressions

An expression e is **very busy** at point p if

- On every path from p, expression e is evaluated before the value of e is changed

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
    output a*b-x;
    x = x-1;
}
output a*b;
```

# Very busy expressions

**Analysis:** same lattice and auxiliary functions as for available expressions

- In which **direction** the information propagate? Backward or forward?
- In which way information coming from other nodes can be **merged**? Union or intersection?

# Very busy expressions

**Analysis:** same lattice and auxiliary functions as for available expressions

- In which **direction** the information propagate? Backward or forward?  
**Backward**
- In which way information coming from other nodes can be **merged**? Union or intersection? **Intersection**

# Very busy expressions

**Optimization:** useful for code hoisting and saves code space

```
var x,a,b;  
x = input;  
a = x-1;  
b = x-2;  
while (x>0) {  
    output a*b-x;  
    x = x-1;  
}  
output a*b;
```

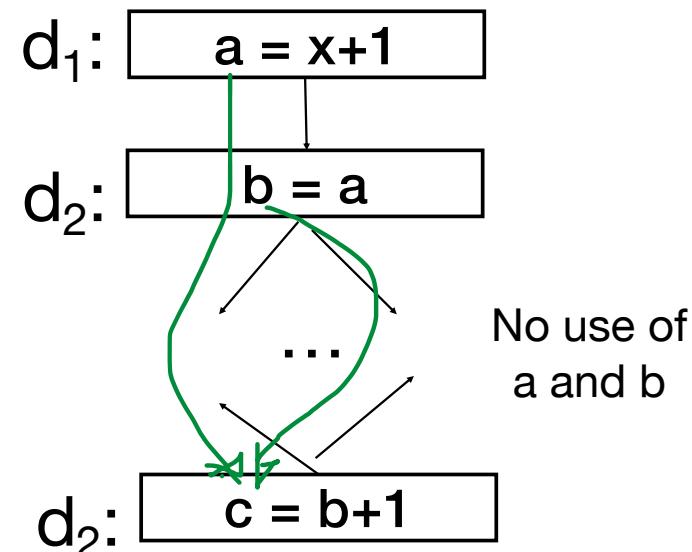
```
var x,a,b,atimesb;  
x = input;  
a = x-1;  
b = x-2;  
atimesb = a*b;  
while (x>0) {  
    output atimesb-x;  
    x = x-1;  
}  
output atimesb;
```

Here, the compiler can perform code hoisting and move the computation to the earliest program point where it is very busy

# Reachable definitions

- A definition of a variable v is an assignment to v
- A definition of variable v **reaches** point p if there is no intervening assignment to v (that contains a definition of v and therefore a kill)

```
void foo(int x) {  
    int a, b, c;  
    a = x + 1;  
    b = a;  
    if (...) { ... }  
    else { ... }  
    c = b + 1;  
    return c;  
}
```



# Reachable definitions

**Analysis:** powerset lattice of all assignments (represented as CFG nodes) occurring in the program

It can relates definitions (i.e. assignments) to its possible uses (def-use graph)

- In which **direction** the information propagate? Backward or forward?  
**Backward**
- In which way information coming from other nodes can be **merged**? Union or intersection? **Union**

# Reachable definitions

**Analysis:** powerset lattice of all assignments (represented as CFG nodes) occurring in the program

It can relates definitions (i.e. assignments) to its possible uses (def-use graph)

- In which **direction** the information propagate? Backward or forward?  
**Forward**
- In which way information coming from other nodes can be **merged**? Union or intersection? **Union**

# Reachable definitions

It can relates definitions (i.e. assignments) to its possible uses (def-use graph)

## Optimizations:

- dead code elimination and
- code motion

# Types of Dataflow Analysis

	<i>Forward</i>	<i>Backward</i>
<i>May</i>	Reaching Definitions	Live Variables
<i>Must</i>	Available Expressions	Very Busy Expressions

# Types of Dataflow Analysis: back and forth

We distinguish between **forward** and **backward** analyses based on the direction in which information propagates through the control flow graph (CFG)

A **forward** analysis gathers information about what has happened before a program point. It starts at the entry node and moves forward through the CFG. The constraints at each node depend only on its predecessors. Classic examples include:

- **Sign analysis:** tracks the sign of variables
- **Available expressions:** determines which expressions have already been computed and not invalidated

A **backward** analysis collects information about what must hold after a program point. It starts at the exit node and proceeds backward through the CFG. The constraints here depend on successors. Examples include:

- **Live variable analysis:** tracks which variables will be used in the future
- **Very busy expressions:** finds expressions that will definitely be evaluated later

# Types of Dataflow Analysis: must and may

We distinguish between **may** and **must** analyses based, for analyses that are based on a powerset lattices

A **may** analysis aims to find **what may possibly be true** at a program point. It computes an **over-approximation** of the actual behavior and uses the lattice order  $\subseteq$  and union ( $\cup$ ) to combine information. These analyses are typically used when missing a possibility could lead to incorrect behavior. Examples include:

- Live variables
- Reaching definitions

A **must** analysis focuses on **what must definitely be true**. It computes an **under-approximation**, using the lattice order  $\supseteq$  and intersection ( $\cap$ ) to combine information. This kind of analysis is used when precision about guaranteed properties is crucial. Examples include:

- Available expressions
- Very busy expressions

# Bibliography

- Anders Møller & Michael I. Schwartzbach. *Static Program Analysis*  
<https://cs.au.dk/~amoeller/spa/>
- F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*  
Springer-Verlag. 1998.