

Lecture 11: Monotone Frameworks

Jeff Smits

CS4200 Compiler Construction

TU Delft

November 2018

Reading Material

Chapter 10: Liveness Analysis

Chapter 17: Dataflow Analysis

The applied/“cookbook” version

class NilExp extends Exp {}
class IntExp extends Exp {int value;}
class StringExp extends Exp {String value;}
class CallExp extends Exp {Symbol func; ExpList args;}
class OpExp extends Exp {Exp left; right; int oper;}
class RecordExp extends Exp {Symbol typ; FieldExpList fields;}
class SeqExp extends Exp {ExpList list;}
class AssignExp extends Exp {Var var; Exp exp;}
class IfExp extends Exp {Exp test; Exp thenclause; Exp elseclause;}
class WhileExp extends Exp {Exp test; Exp body;}
class ForExp extends Exp {VarDec var; Exp init; Exp cond; Exp body;}
class BreakExp extends Exp {}
class LetExp extends Exp {DecList decs; Exp body;}
class ArrayExp extends Exp {Symbol typ; Exp size; init;}

abstract class Dec {
 class FunctionDec extends Dec {Symbol name; FieldList params; NameTy results;
 Exp body; FunctionDec next;}
 abstract class Var {}
 class VarDec extends Var {Symbol name; boolean escape; FieldList params; NameTy results;
 TypeDec type; Exp init;}
 class SubscriptVar extends Var {Var var; Exp index;}
 class FieldVar extends Var {Var var; Symbol field;}
}

abstract class Ty {
 class FunctionTy extends Ty {Symbol name; FieldList params; NameTy results;}
 class VarTy extends Ty {Symbol name;}
 class RecordTy extends Ty {FieldList fields;}
 class ArrayTy extends Ty {Symbol name; TypeDec elem; int value;}
 class SeqTy extends Ty {TypeDec[] types;}
 class OpTy extends Ty {Symbol func; ExpList args;}
 class RecordTy extends Ty {FieldList fields;}
 class AssignTy extends Ty {Var var; Exp exp;}
 class IfTy extends Ty {Exp test; Exp thenclause; Exp elseclause;}
 class WhileTy extends Ty {Exp test; Exp body;}
 class ForTy extends Ty {VarDec var; Exp init; Exp cond; Exp body;}
 class BreakTy extends Ty {}
 class LetTy extends Ty {Dec[] decs; Exp body;}
 class FunctionTy extends Ty {Symbol name; FieldList params; NameTy results;}
 class VarTy extends Ty {Symbol name;}
 class RecordTy extends Ty {FieldList fields;}
 class ArrayTy extends Ty {Symbol name; TypeDec elem; int value;}
 class SeqTy extends Ty {TypeDec[] types;}
 class OpTy extends Ty {Symbol func; ExpList args;}
 class RecordTy extends Ty {FieldList fields;}
 class AssignTy extends Ty {Var var; Exp exp;}
 class IfTy extends Ty {Exp test; Exp thenclause; Exp elseclause;}
 class WhileTy extends Ty {Exp test; Exp body;}
 class ForTy extends Ty {VarDec var; Exp init; Exp cond; Exp body;}
 class BreakTy extends Ty {}
 class LetTy extends Ty {Dec[] decs; Exp body;}
}

modern compiler implementation in Java
second edition



andrew w. appel

Chapter 1: Introduction

Chapter 2: Data Flow Analysis

FLEMMING NIELSON
HANNE RIIS NIELSON
CHRIS HANKIN

Principles of Program Analysis



 Springer

The theoretical/“general”version

Data-Flow Analysis

Previous Lecture

Control-Flow

- Order of execution
- Reasoning about what is reachable

Data-Flow

- Flow of data through a program
- Reasoning about data, and dependencies between data

FlowSpec

- Control-Flow rules to construct the graph
- Annotate with information from analysis by Data-Flow rules

What is Data-Flow Analysis?

Static approximation of runtime behaviour

- What has or will be computed
- What extra invariants do some data adhere to
- Data dependence between data/variables where the data lives

Available Expressions

```
let
  var x : int := a + b
  var y : int := a * b
in
  while y > a + b then
    (
      a := a + 1;
      x := a + b
    )
end
```

- $a + b$ is already computed when you get to the condition
- There is no need to compute it again

Live variables

```
x := 2;  
y := 4;  
x := 1;  
if y > x then  
    z := y  
else  
    z := y * y;  
x := z
```

- The first value of x is never observed, because it isn't read after the assignment

What is Data-Flow Analysis?

Static approximation of runtime behaviour

- What has or will be computed
- What extra invariants do some data adhere to
- Data dependence between data/variables where the data lives

Flow-sensitive types

```
void hello(String? name) {  
    if (is String name) {  
        // name is of type String here  
        print("Hello, ``name``!");  
    }  
    else {  
        print("Hello, world!");  
    }  
}
```

- Ceylon (<https://ceylon-lang.org/>)
- Union and intersection types
- `String?` \equiv `String | Null`
- `is` like Java's `instanceof`
- General name: path-sensitive data-flow analysis

What is Data-Flow Analysis?

Static approximation of runtime behaviour

- What has or will be computed
- What extra invariants do some data adhere to
- Data dependence between data/variables where the data lives

Reaching definitions

```
let
  var x : int := 5
  var y : int := 1
in
  while x > 1 do
    (
      y := x * 2;
      x := y - 1
    )
end
```

- The inverse relation of live variables
- RD gives us the possible origins of the current value of a variable

Reaching definitions

```
1 let
2   var x : int := 5 _____ x ↦ 2
3   var y : int := 1 _____ x ↦ 2 ; y ↦ 3
4 in
5   while x > 1 do
6     (
7       y := x * 2 ; _____ x ↦ 2 , 8 ; y ↦ 3 , 7
8       x := y - 1 _____ x ↦ 8 ; y ↦ 7
9     )
10 end
```

- Analysis result is a map (shown here after each statement)
- Propagate information along the control-flow graph

Traditional Kill/Gen Sets

Traditional set based analysis

Traditional set based analysis has

- Sets as the type of information that's calculated
- A transfer function of:
 - ▶ `previousSet \ kill(currentNode) ∪ gen(currentNode)`
- Usually the initial set is empty

Depends on the direction of the analysis

Available Expressions

“An **expression** is **available** if it *must* have already been computed, and not later modified, on all paths to the program point”

```
kill(Assign(var, e1)) :=  
{ e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=  
{ e2 ∈ SE(e1) | var ∉ FV(e2) }
```

AllAE

- All Available Expressions in the program

FV

- Free Variables of the argument

SE

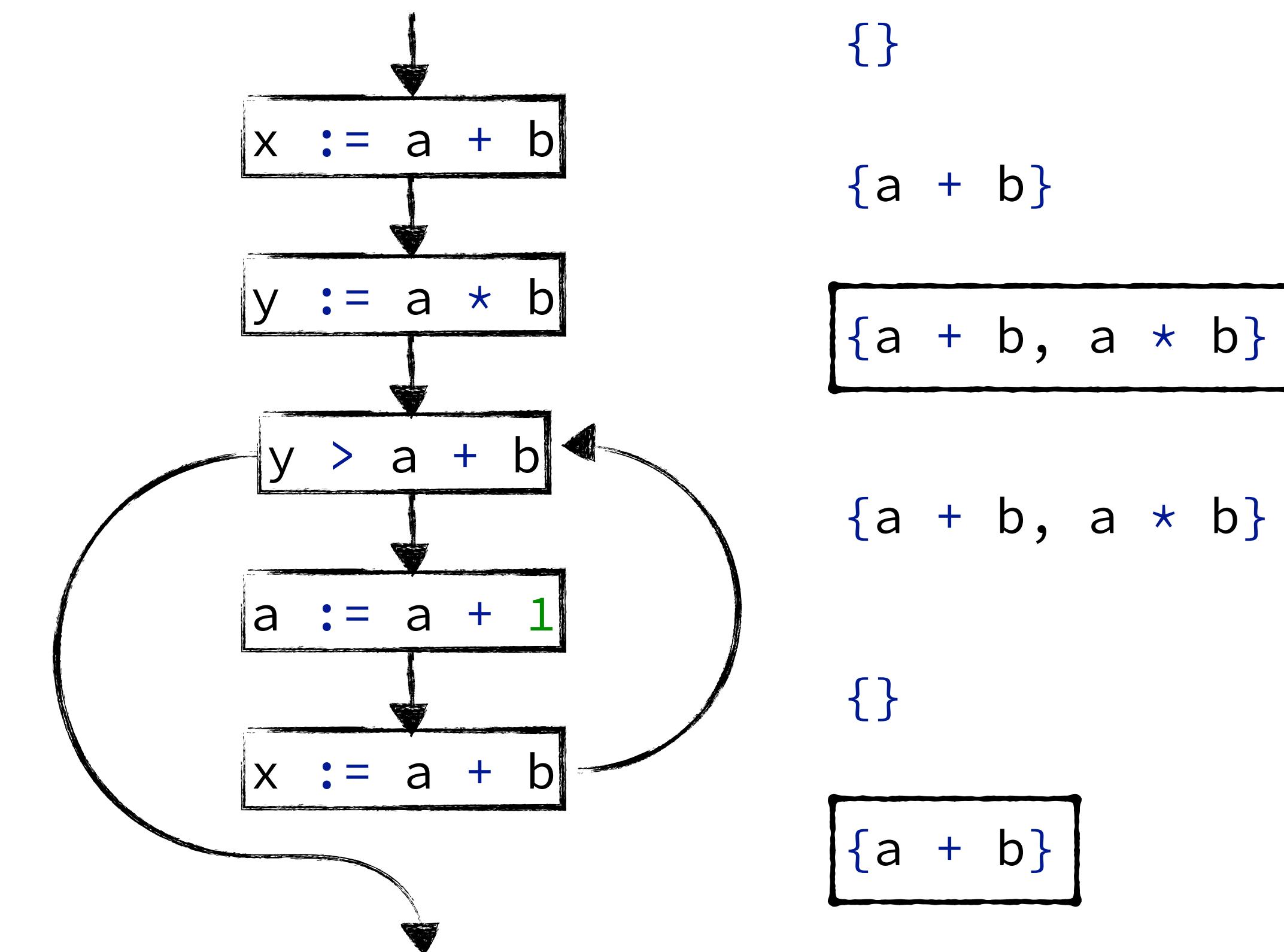
- Subexpressions of the argument

Available Expressions

“An **expression** is **available** if it *must* have already been computed, and not later modified, on all paths to the program point”

```
kill(Assign(var, e1)) :=  
{ e2 ∈ AllAE | var ∈ FV(e2) }
```

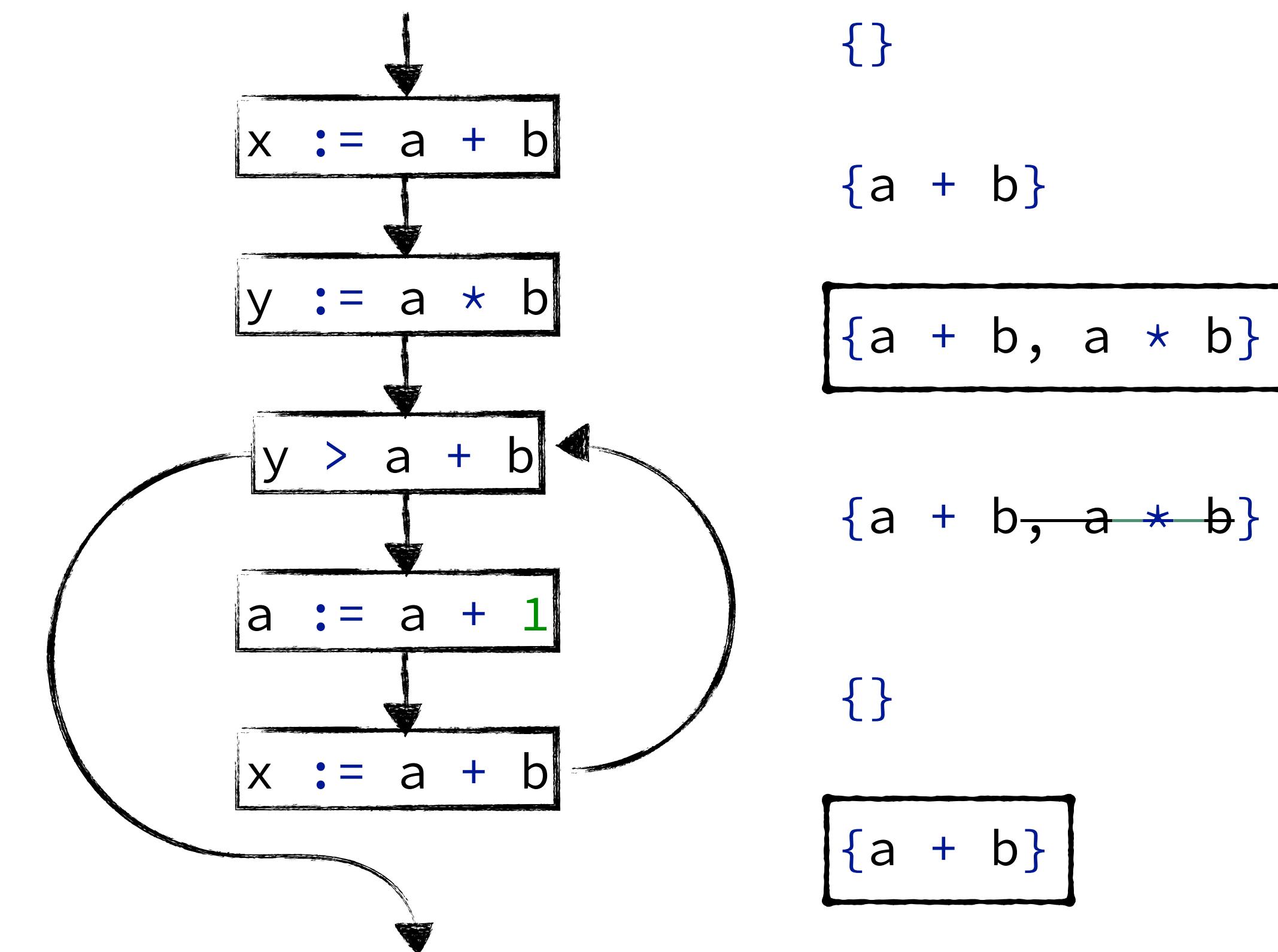
```
gen(Assign(var, e1)) :=  
{ e2 ∈ SE(e1) | var ∉ FV(e2) }
```



Available Expressions

“An **expression** is **available** if it *must* have already been computed, and not later modified, on all paths to the program point”

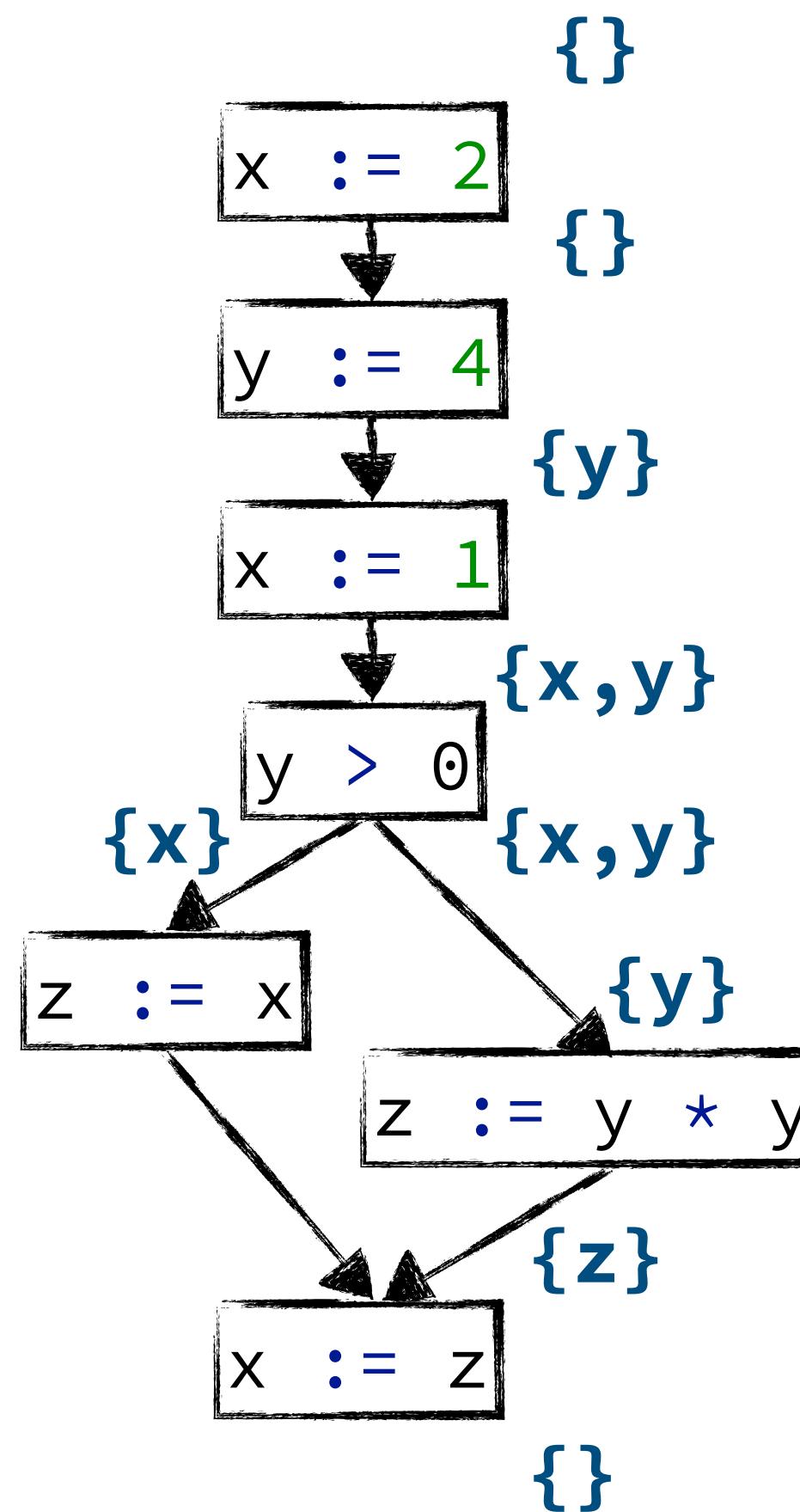
```
kill(Assign(var, e1)) :=  
{ e2 ∈ AllAE | var ∈ FV(e2) }  
  
gen(Assign(var, e1)) :=  
{ e2 ∈ SE(e1) | var ∉ FV(e2) }
```



Live Variables

“A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path.”

```
kill(Assign(var, e1)) :=  
{ var }  
  
gen(Assign(var, e1)) :=  
{ FV(e1) }  
  
gen(b@BinOp(_, _, _)) :=  
{ FV(b) }  
  
gen(u@UnOp(_, _)) :=  
{ FV(u) }
```



Traditional set based analysis

Sets as analysis information

Kill and gen sets per control node type

- $\text{previousSet} \setminus \mathbf{kill}(\text{currentNode}) \cup \mathbf{gen}(\text{currentNode})$

Can propagate either forward or backward

Can merge information with either union or intersection

- Respectively called **may** and **must** analyses

Beyond Sets

Constant propagation and folding

```
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
```

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

full propagation



Constant propagation and folding

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

Constant propagation and folding

The type of the analysis information

- Variables bound to either a particular *constant* or a *marker for non-constants*

The transfer functions per control node

- Basically an interpreter implementation for constants
- Needs to propagate markers when found

Monotone Frameworks

Termination

Data-Flow Analysis needs fixpoint computation

- Because of loops

To terminate, there needs to be a fixpoint

- **And** we need to actually get to that fixpoint
- How can we check this?

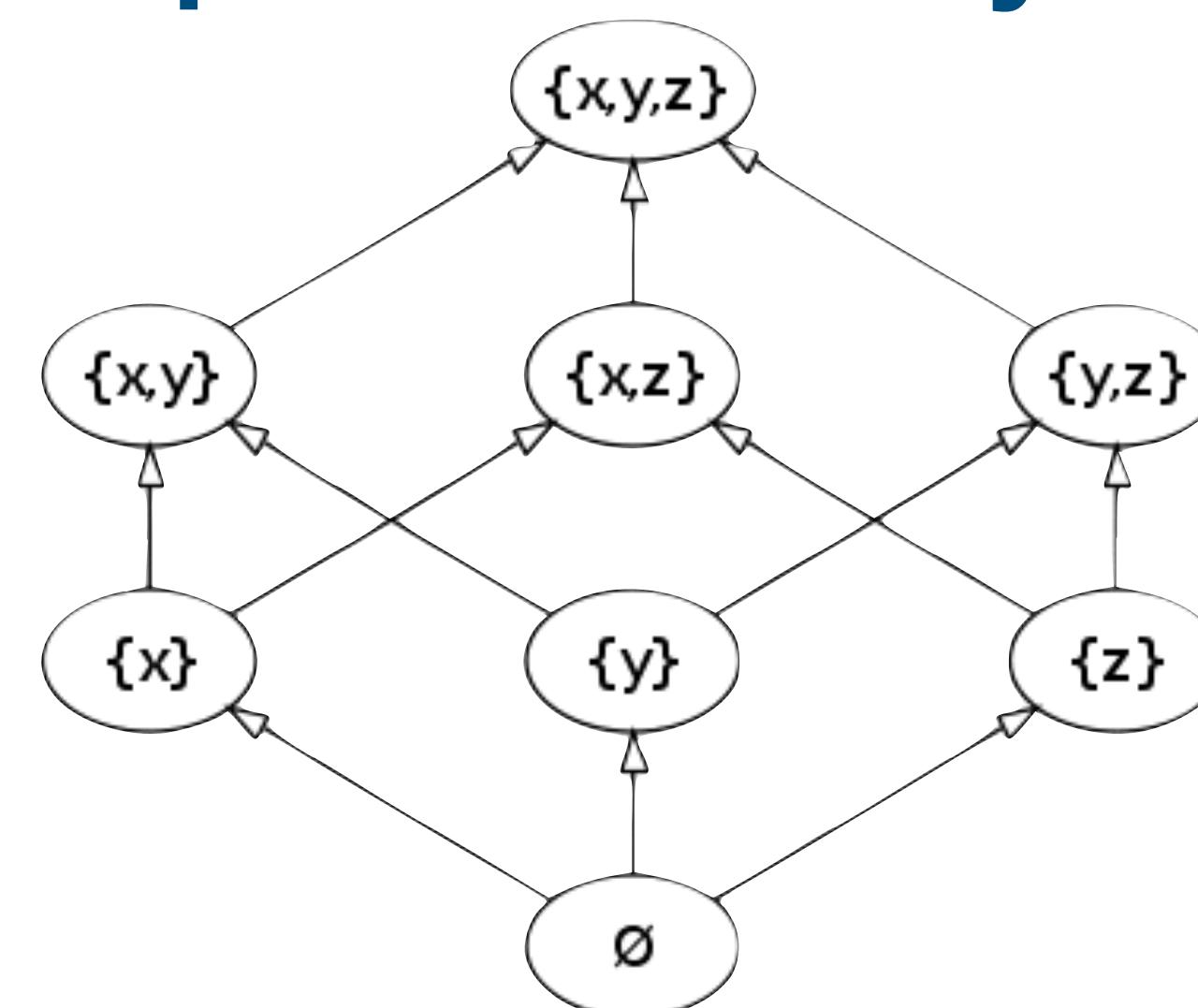
Lattice Theory

A set X is totally ordered under \leq if for $a, b, c \in X$

- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitivity)
- $a \leq b \vee b \leq a$ (totality)

A partial ordering drop the totality constraint

- e.g. subset inclusion:



Lattice Theory

A Lattice is a partially ordered set where

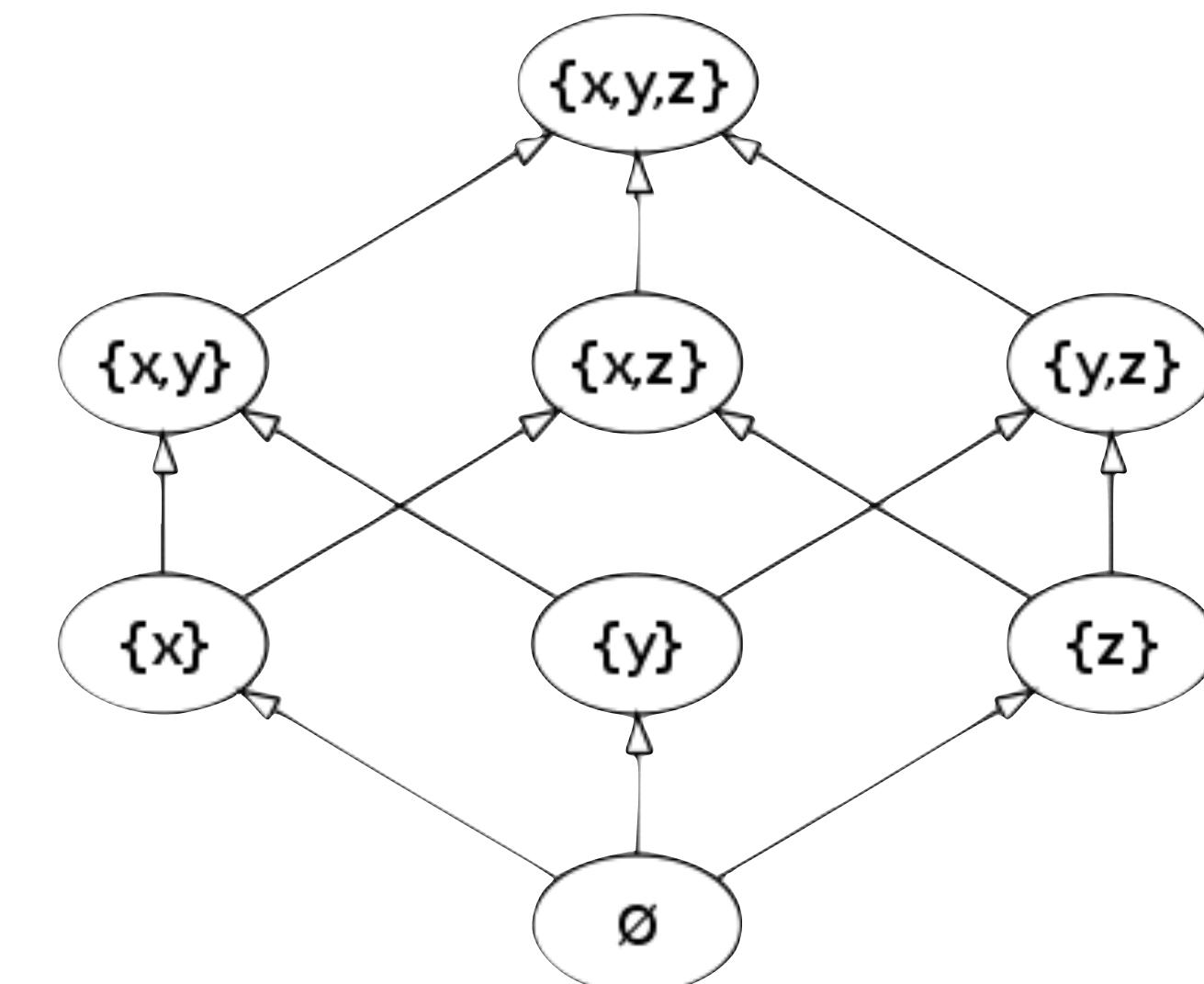
- every two elements have a unique least upper bound (or supremum or join)
- every two elements have a unique greatest lower bound (or infimum or meet)

Least upper bound (LUB)

- $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$
- $a \sqcup b = c \Rightarrow a \sqsubseteq c \wedge b \sqsubseteq c$

Greatest lower bound (GLB)

- $a \sqsubseteq b \Leftrightarrow a \sqcap b = a$
- $a \sqcap b = c \Rightarrow c \sqsubseteq a \wedge c \sqsubseteq b$



A bounded lattice has a top and bottom

- These are \top and \perp respectively

Lattices for data-flow analysis

Consider \top as the coarsest approximation

- It's a safe approximation, because it says we're not sure of anything

Then we can combine data-flow information with \sqcup

- It is the most information preserving combination of information

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f , $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing
- Worst case scenario: the loop reaches T
- *This **only** works if the lattice is of **finite height***

General interval analysis has an infinite lattice

- $T = [-\infty, \infty]$
- If a loop adds a finite number to a variable, you never get to ∞

Recap

An analysis consists of

- The type of the analysis information
- The *transfer functions* that express the ‘effect’ of a control node
- The initial analysis information

Recap

An analysis consists of

- The type of the analysis information, **and the lattice instance for that type**
- The *transfer functions* that express the ‘effect’ of a control node
 - ▶ **These should be monotone with respect to the lattice**
- The initial analysis information

Executing Monotone Frameworks

Executing Monotone Frameworks

Great formal model for reasoning

- Fairly simple
- Makes intuitive sense
- Has nice mathematical properties

But how to execute?

Framework overview

Control-flow graph

- graph
- start node
- reverse beforehand if backward analysis

Lattice instance for data-flow information:

- Lattice L
- Least Upper Bound \sqcup
- Bottom value $\perp \in L$

Initial data-flow information for start node

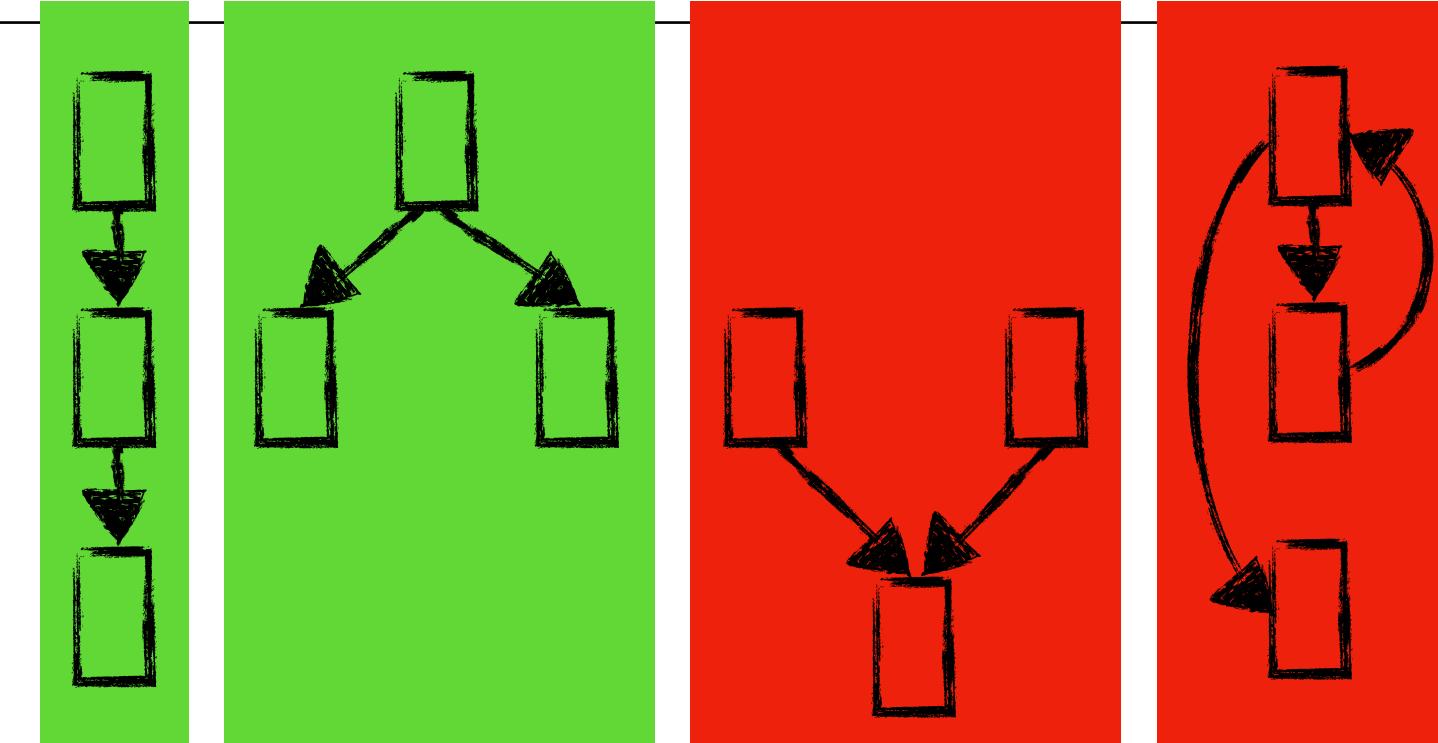
Transfer function $f: (L \rightarrow L)$ per control-flow graph node

- Denotes the data-flow effect of the CFG node

Naive approach

```
start_node.value = initial_value
walk(start_node)

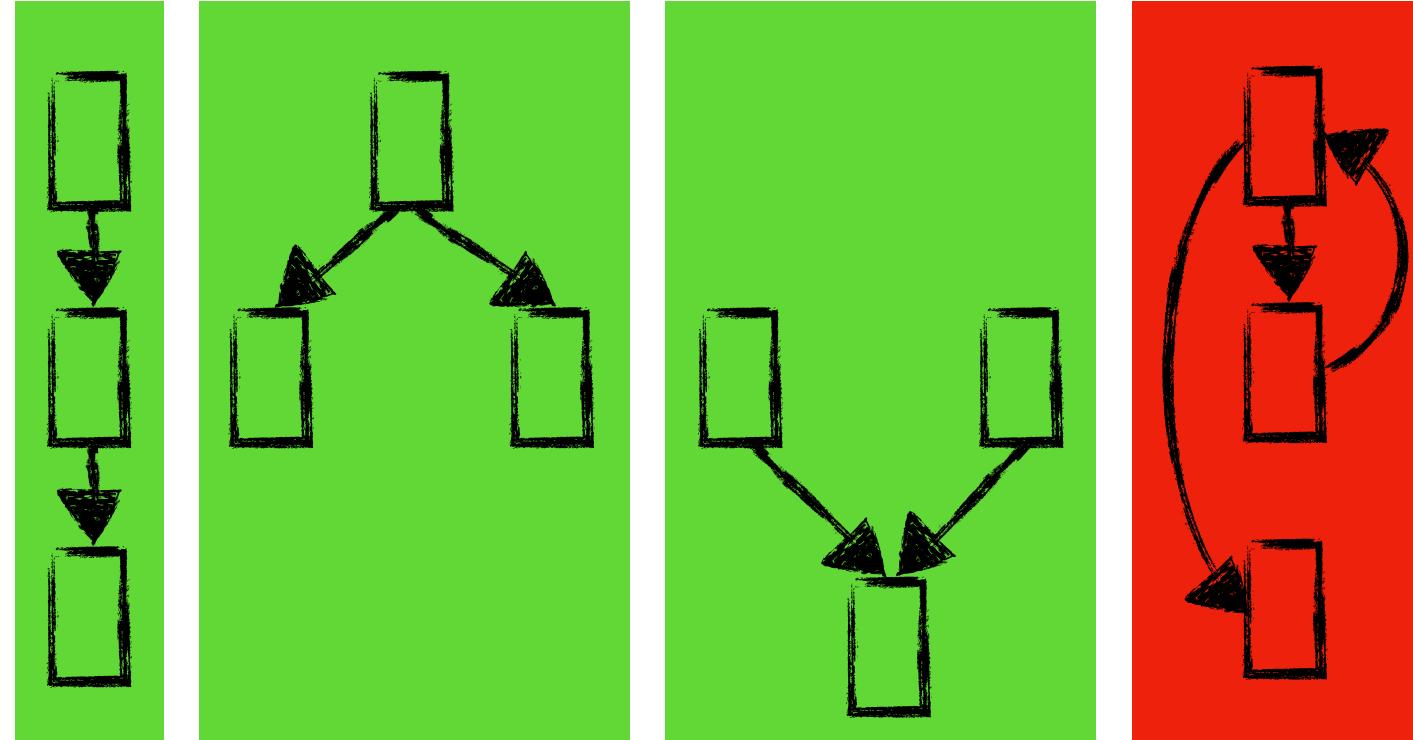
function walk(node) =
  for next in node.successors:
    next.value = node.transfer(node.value)
    walk(next)
```



- Fine for straight-line programs
- Distributes information along splits in control-flow
- Overrides values from one path with those of another
- Recursive: great for stack overflows on loops

Using lattices

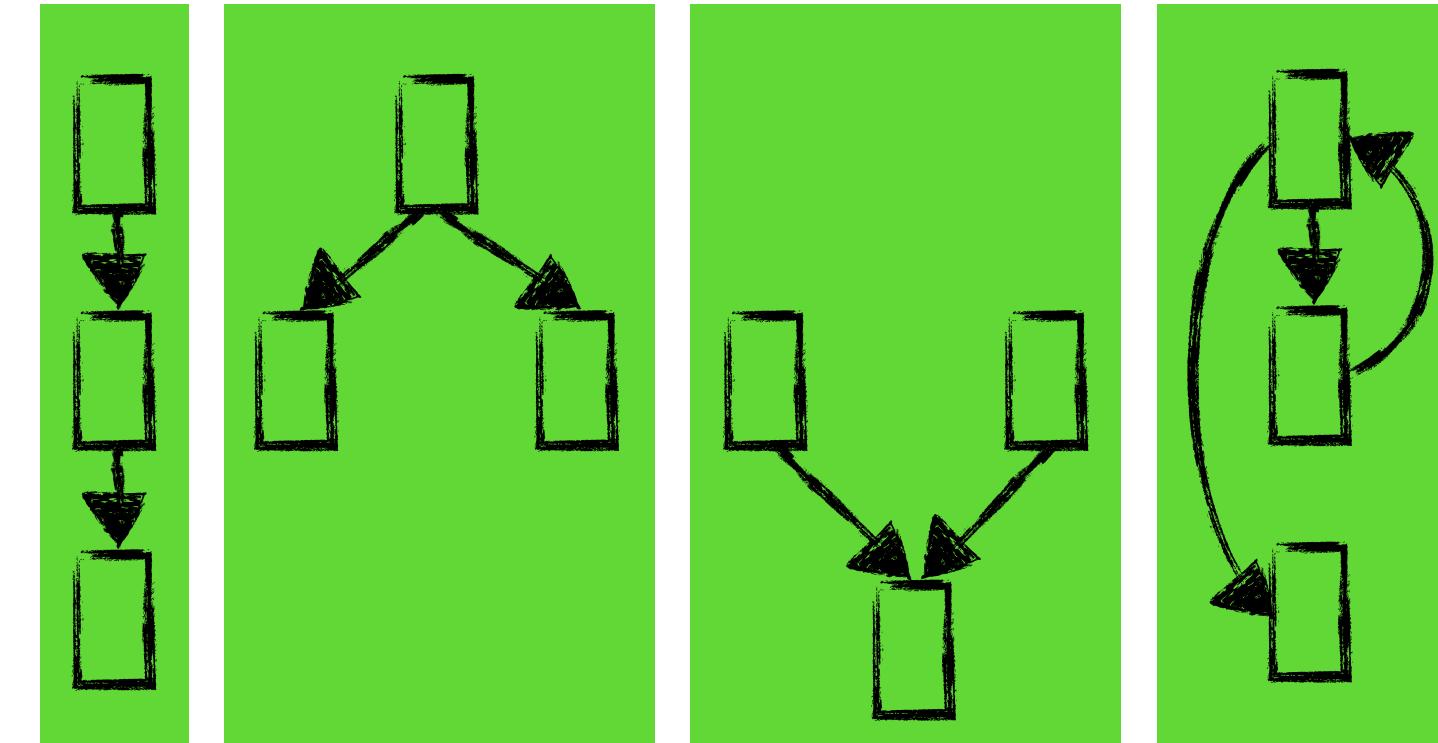
```
for node in nodes:  
    node.value = bottom  
start_node.value = initial_value  
walk(start_node)  
  
function walk(node) =  
    for next in node.successors:  
        next.value =  
            next.value  $\sqcup$  node.transfer(node.value)  
    walk(node)
```



- Fine for straight-line programs
- Distributes information along splits in control-flow
- Combines values from one path with those of another
- Recursive: great for stack overflows on loops

Worklist

```
for node in nodes:  
    node.value = bottom  
start_node.value = initial_value  
worklist = [nodes]  
  
while !worklist.empty():  
    node = worklist.pop()  
    for next in node.successors:  
        oldValue = next.value  
        newValue = node.transfer(node.value)  
        if !(newValue ⊑ oldValue):  
            next.value = oldValue ∪ newValue  
            worklist += [next]
```



If `initial_value == bottom` and a transfer function is identity: traversal will stop there, so don't just start from the `start_node`

- Fine for straight-line programs
- Distributes information along splits in control-flow
- Combines values from one path with those of another
- Worklist: works for loops too

FlowSpec Design

Framework overview

Control-flow graph

- graph Control-flow rules
- start node Root rule(s)
- reverse beforehand if backward analysis In edge direction of data-flow rules

Lattice instance for data-flow information:

- Lattice L In property definition
- Least Upper Bound \sqcup In lattice definition
- Bottom value $\perp \in L$

Initial data-flow information for start node In special data-flow rule

Transfer function $f: (L \rightarrow L)$ per control-flow graph node

- Denotes the data-flow effect of the CFG node In data-flow rule

Variants of Data-Flow Analysis

Many interacting features

- Intra-procedural or inter-procedural
 - ▶ Inter-procedural with dynamic dispatch means dynamic control flow analysis depending on the data-flow analysis
- Flow-insensitive, flow-sensitive or even path-sensitive
- Different kind of context-sensitivity for dynamic dispatch

Worklist optimizations in FlowSpec

Worklist algorithm optimizations

Filter irrelevant CFG nodes

- With transfer function $tf(x) = x$

Order nodes

- Topological order would make sense
- But there are cycles in our graphs
- Every cycle should be computed to a fixpoint
 - ▶ Really we need each strongly connected component (SCC)
- Tarjan's SCCs algorithm gives SCCs in reverse topological order!
- Within each SCC the order should also not be random:
 - ▶ We use the reverse post-order of the spanning tree

Tarjan's SCC algorithm

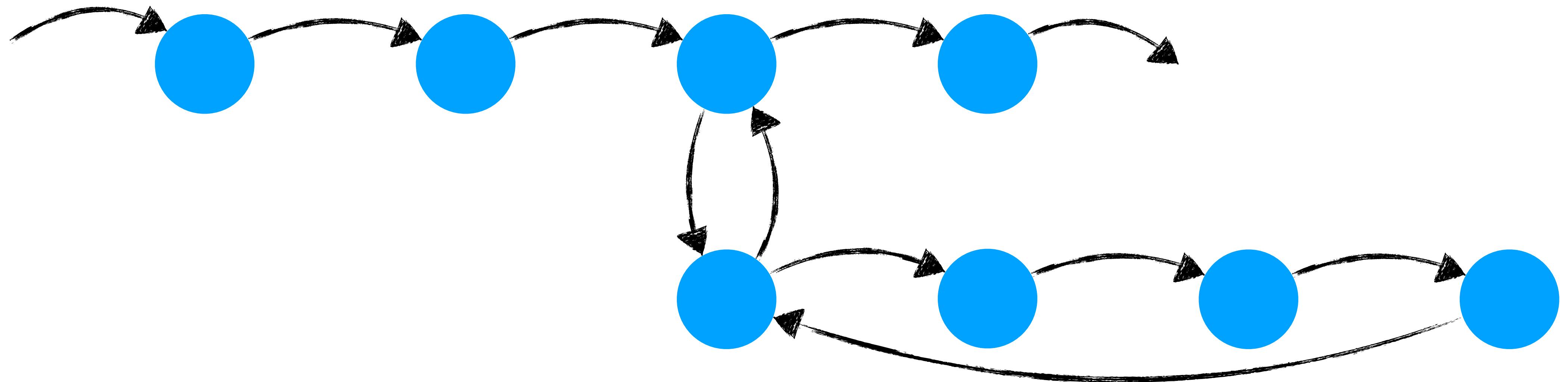
SCC identification

- Label nodes with increasing integers during a depth-first searches
 - ▶ Multiple searches to make sure you reach all nodes in the graph
- The depth-first spanning forest (spanning trees from the searches) holds SCCs as subtrees
- Nodes that can reach the same lowest label are an SCC together

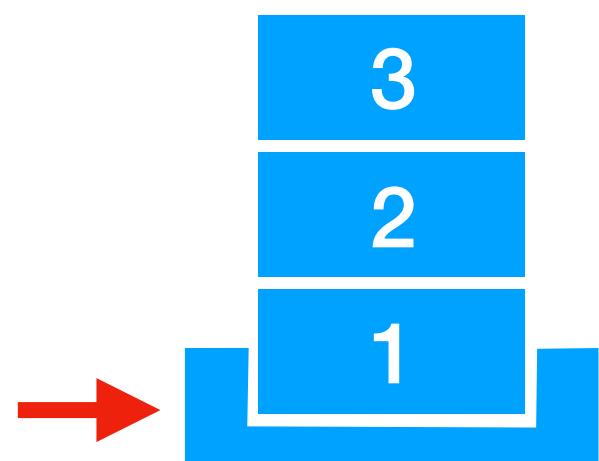
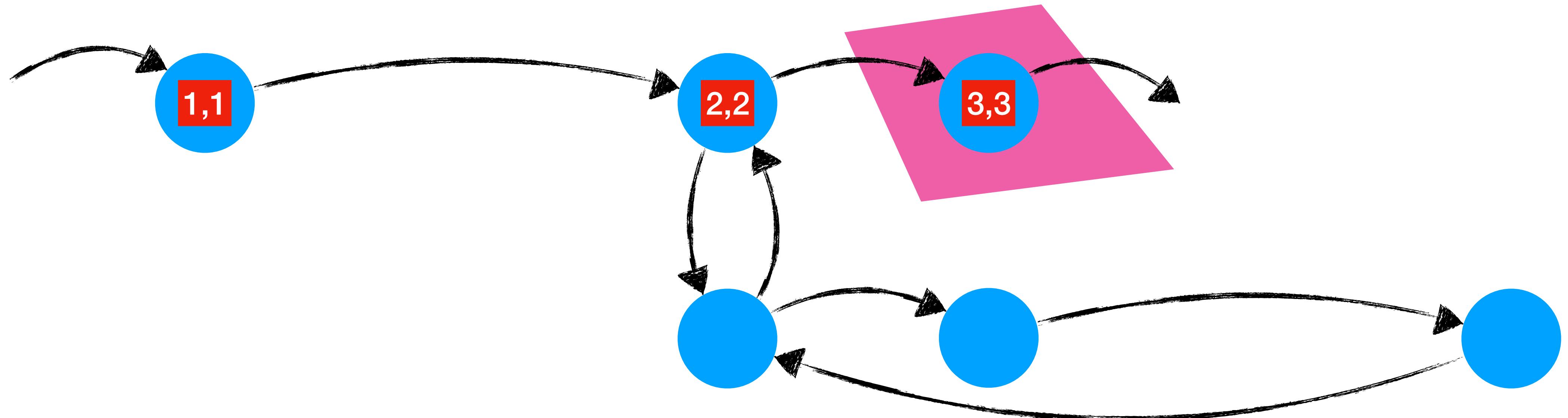
The version in FlowSpec is slightly adapted

- To return the topological order instead of the reverse topo order
- To have reverse postorder inside SCCs

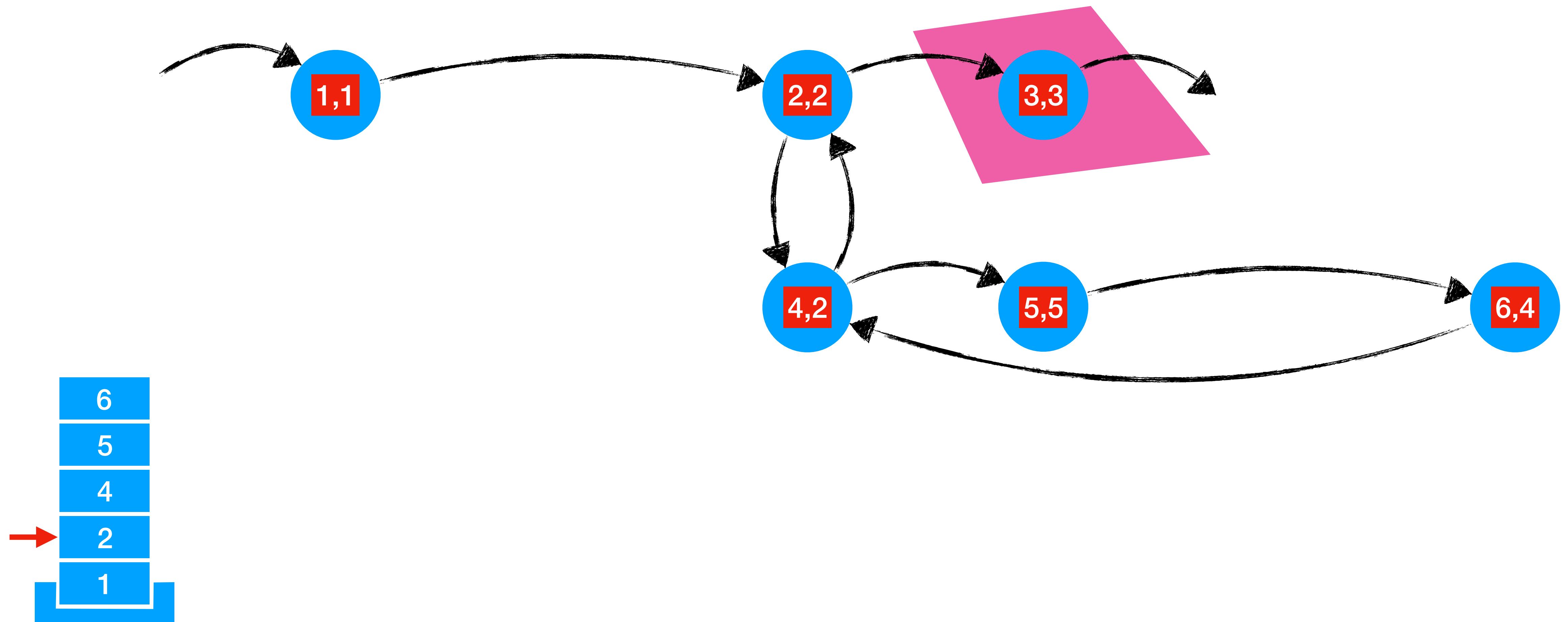
CFG filtering



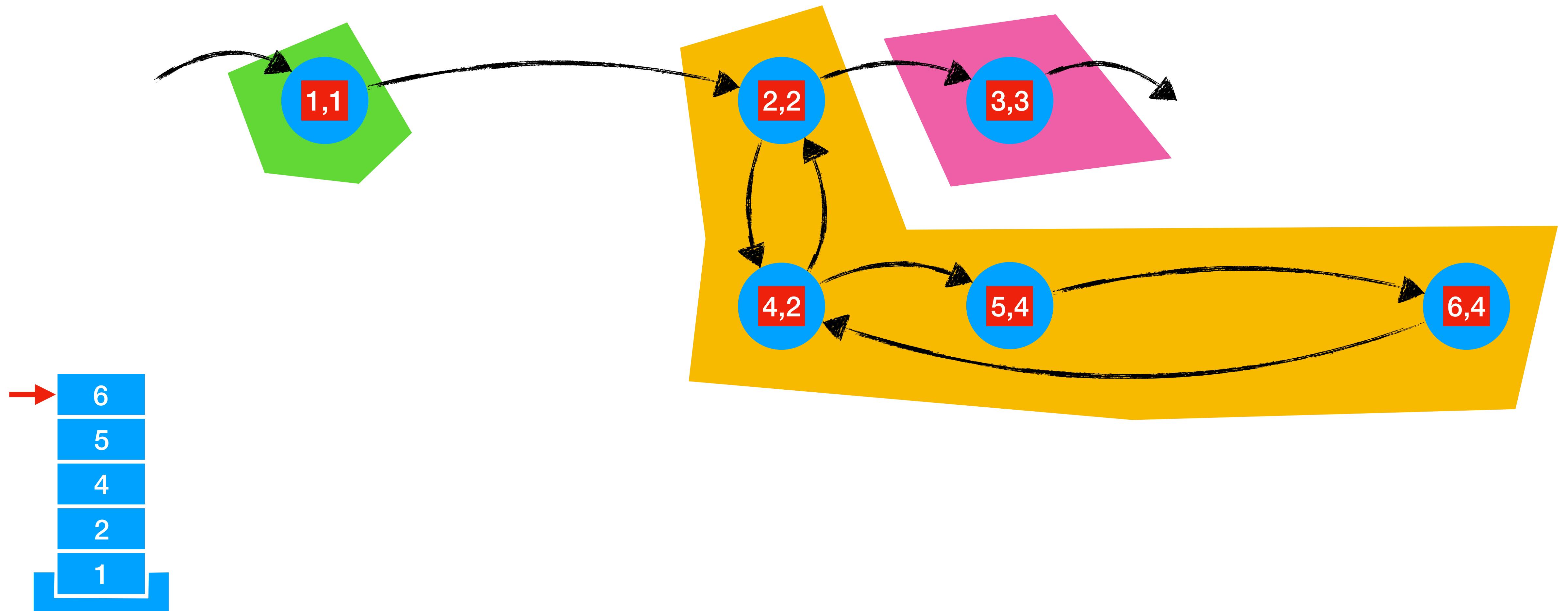
Tarjan's SCC algorithm



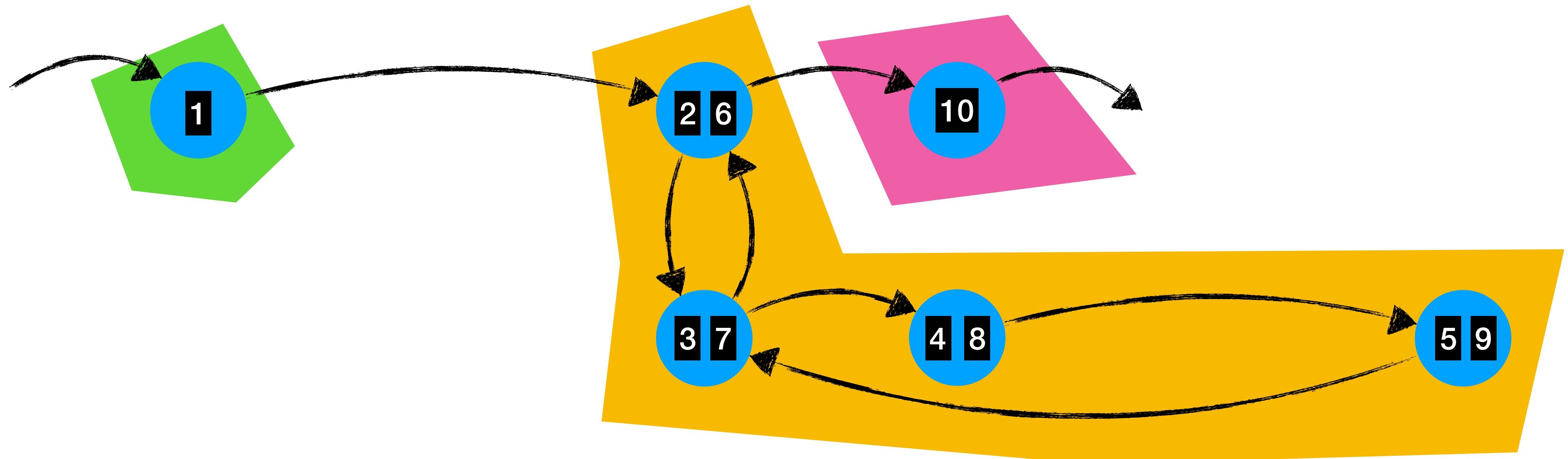
Tarjan's SCC algorithm



Tarjan's SCC algorithm



Reverse postorder in SCC



Conclusion

Summary

Data-flow analysis and its uses

- Sets are not enough
- Lattices are the generalization

Monotone Frameworks

- Finite height lattices and monotone transfer functions
- Execute by worklist algorithm

FlowSpec design

- FlowSpec only does intra-procedural, flow-sensitive analysis
- Worklist algorithm with optimizations:
 - ▶ SCCs, reverse post-order within SCC, CFG filtering