Lattice Theory
Control Flow Graphs
Dataflow Analysis

Static Analysis 2009

Michael I. Schwartzbach
Computer Science, University of Aarhus

Partial Orders

- A partial order is a structure $L = (S, \sqsubseteq)$
- S is a set
- $\sqsubseteq$ is a binary relation that satisfies:
  - reflexivity: $\forall x \in S: x \sqsubseteq x$
  - transitivity: $\forall x,y,z \in S: x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - anti-symmetry: $\forall x,y \in S: x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Upper and Lower Bounds

- Let $X \subseteq S$ be a subset
- We say that $y \in S$ is an *upper* bound ($X \sqsubseteq y$) when:

  $\forall\, x \in X: x \sqsubseteq y$
- We say that $y \in S$ is a *lower* bound ($y \sqsubseteq X$) when:

  $\forall\, x \in X: y \sqsubseteq x$

- A *least* upper bound $\sqcup X$ is defined by:

  $X \sqsubseteq \sqcup X \wedge \forall y \in S: X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$
- A *greatest* lower bound $\sqcap X$ is defined by:

  $\sqcap X \sqsubseteq X \wedge \forall y \in S: y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$

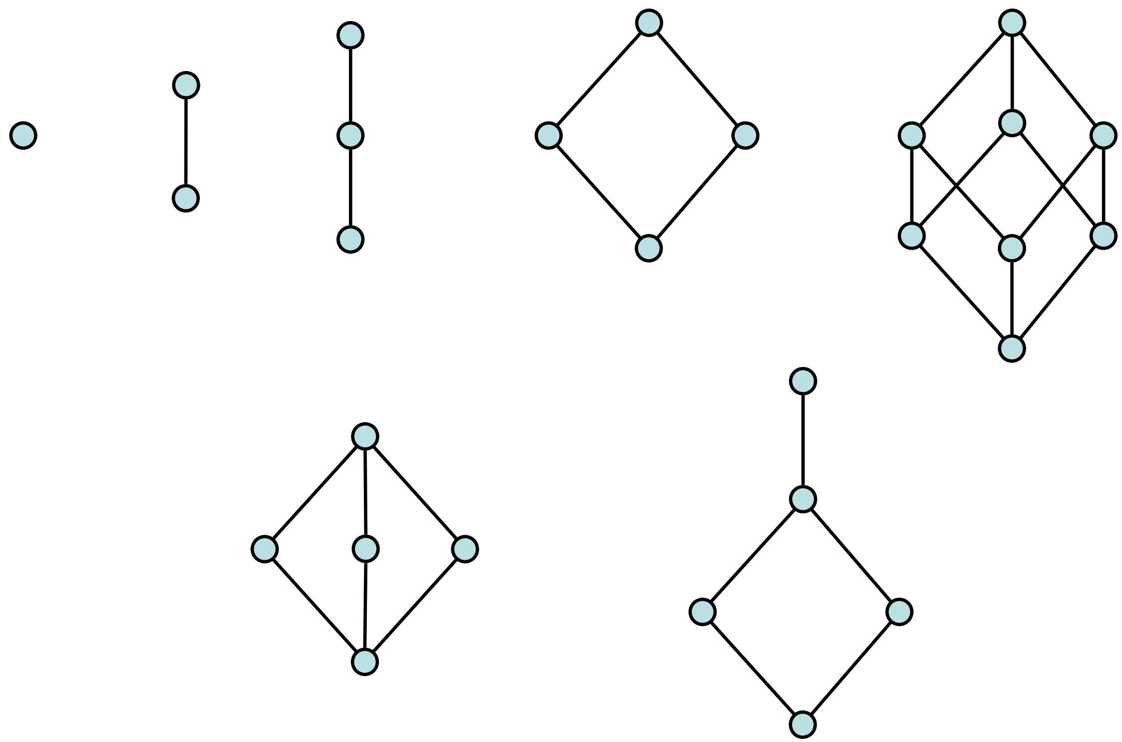
Lattices

- A *lattice* is a partial order where:
  $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$

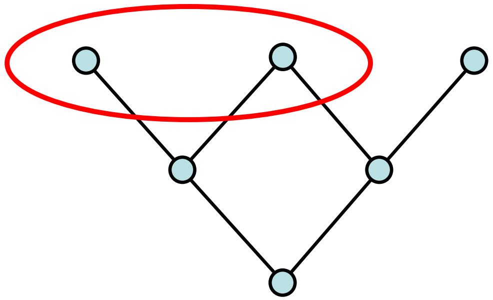
- A lattice must have:
  - a unique largest element, $\top = \sqcup S$
  - a unique smallest element, $\bot = \sqcap S$

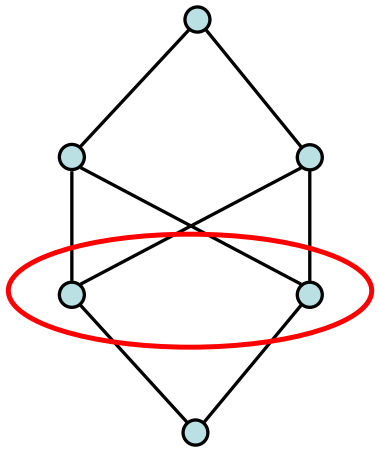
- If S is a finite set, then it is a lattice iff:
  - $\top$ and $\bot$ exist
  - $x \sqcup y$ and $y \sqcap x$ exist for all $x,y \in S$

These Partial Orders Are Lattices



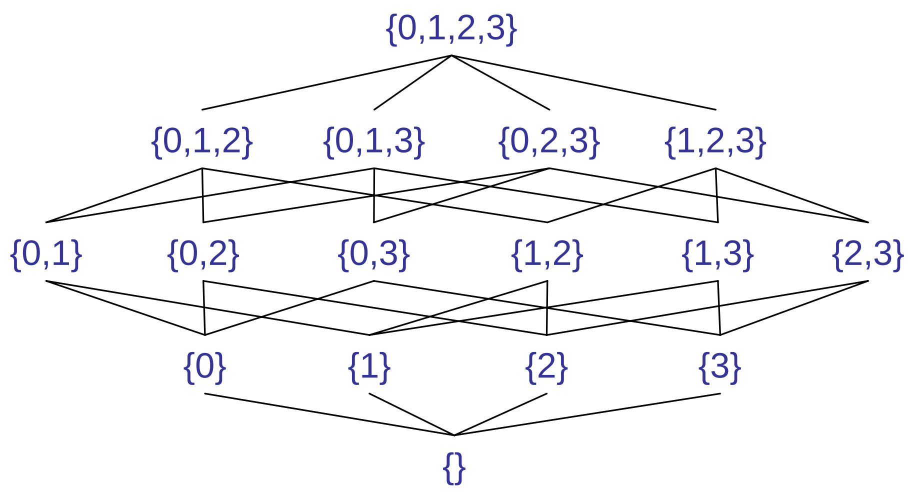
These Partial Orders Are Not Lattices





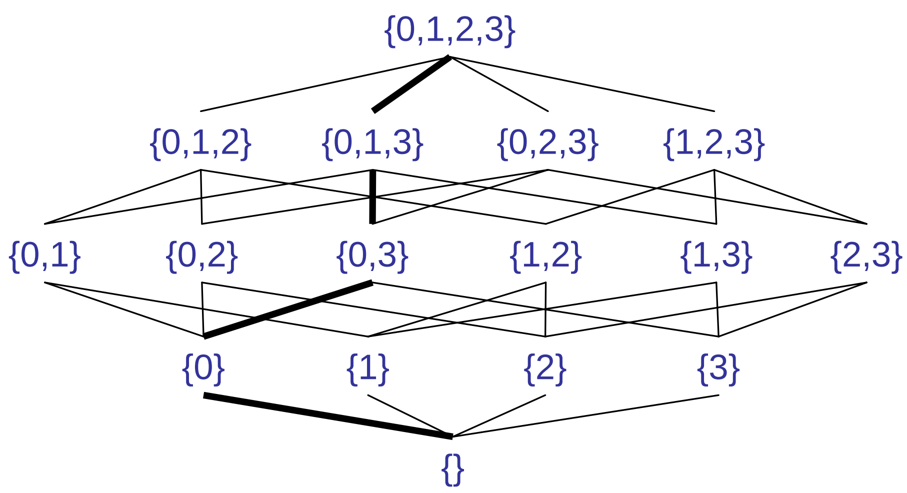
The Subset Lattice

- Every finite set A defines a lattice $(2^A,\subseteq)$, where:
  - $\bot = \varnothing$
  - $\top = A$
  - $x \sqcup y = x \cup y$
  - $x \sqcap y = x \cap y$



Lattice Height

- The *height* of a lattice is the length of the longest path from $\perp$ to $\top$
- The lattice $(2^A, \subseteq)$ has height $|A|$



Monotone and Increasing Functions

- A function f: L $\rightarrow$L is *monotone* when:

  $\forall x,y \in S: x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

- Monotone functions are closed under composition
- As functions, $\sqcup$ and $\sqcap$ are both monotone

- A function is *increasing* when:

  $\forall x \in S: x \sqsubseteq f(x)$

- Monotone is different from increasing
  - e.g. all constant functions are monotone

The Fixed-Point Theorem

- In a lattice with finite height, every monotone function f has a unique least fixed-point:

$$fix(f) = \bigsqcup_{i \geq 0} f^i(\perp)$$

such that $f(fix(f)) = fix(f)$

Proof of Existence

- Clearly, $\perp \sqsubseteq f(\perp)$
- Since f is monotone, we also have $f(\perp) \sqsubseteq f^2(\perp)$
- By induction, $f^i(\perp) \sqsubseteq f^{i+1}(\perp)$
- This means that:

  $$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \ \dots \ f^i(\perp) \dots$$

  is an increasing chain
- L has finite height, so for some k: $f^k(\perp) = f^{k+1}(\perp)$
- But then $\mathit{fix}(f) = f^k(\perp)$

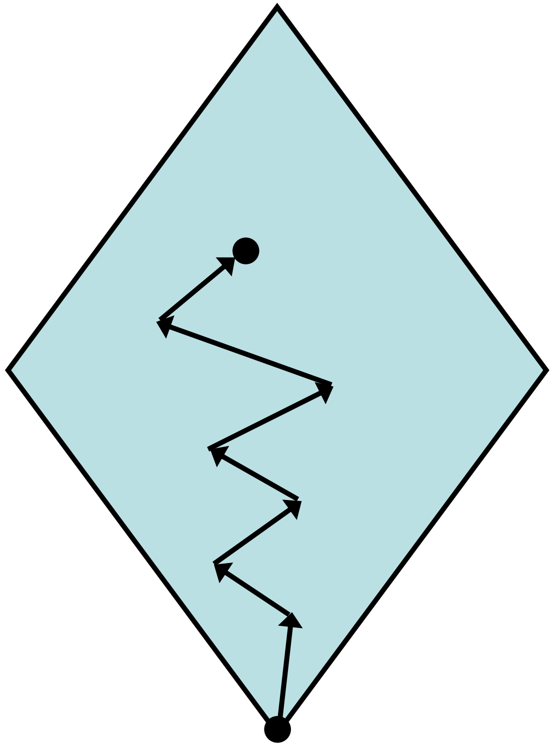
Proof of Unique Least

- Assume that x is another fixed-point: $x = f(x)$
- Clearly, $\perp \sqsubseteq x$
- By induction, $f^i(\perp) \sqsubseteq f^i(x) = x$
- In particular, $fix(f) = f^k(\perp) \sqsubseteq x$
- Uniqueness then follows from anti-symmetry

Computing Fixed-Points

- The time complexity of *fix*(f) depends on:
  - the height of the lattice
  - the cost of computing f
  - the cost of testing equality

```
x = ⊥ ;
do { t = x; x = f(x); }
while (x ≠ t);
```



Product Lattice

- If $L_1, L_2, \ldots, L_n$ are lattices, then so is the *product*:

$$L_1 \times L_2 \times \ldots \times L_n = \{ (x_1, x_2, \ldots, x_n) \mid x_i \in L_i \}$$

where $\sqsubseteq$ is defined pointwise

- Note that $\sqcup$ and $\sqcap$ can be computed pointwise
- *height*$(L_1 \times L_2 \times \ldots \times L_n) =$ *height*$(L_1) + \ldots +$ *height*(L_n)

Sum Lattice

- If $L_1, L_2, \ldots, L_n$ are lattices, then so is the *sum*:

$$L_1+L_2+ \ldots +L_n = \{ (i,x_i) \mid x_i \in L_i \backslash \{\bot, \top\}\} \cup \{\bot, \top\}$$

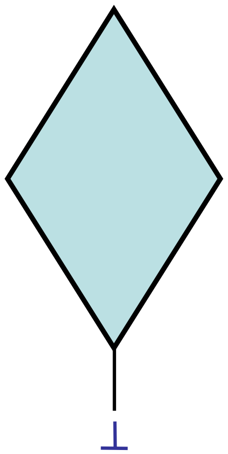
where:

- $\bot$ and $\top$ are as expected
- $(i,x) \sqsubseteq (j,y)$ if and only if $i=j$ and $x \sqsubseteq y$

- $\textit{height}(L_1+L_2+ \ldots +L_n) = \textit{max}\{\textit{height}(L_i)\}$

Lift Lattice

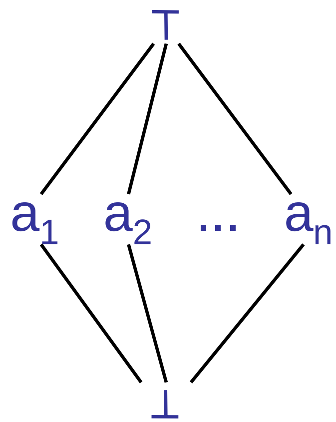
- If L is a lattice, then so is *lift*(L), which is:



- *height*(*lift*(L)) = *height*(L)+1

Flat Lattice

- If A is a finite set, the *flat*(A) is a lattice:



- *height*(*flat*(A)) = 2

Map Lattice

- If A is a finite set and L is a lattice, then we obtain the map lattice:

  $$A \rightarrow L = \{ [a_1 \rightarrow x_1, \ldots, a_n \rightarrow x_n] \mid x_i \in L_i \}$$

  ordered pointwise

- $\mathit{height}(A \rightarrow L) = |A| \cdot \mathit{height}(L)$

Lattice Equations

- Let L be a lattice with finite height

- A *equation system* is of the form:

$x_1 = F_1(x_1, ..., x_n)$
$x_2 = F_2(x_1, ..., x_n)$
...
$x_n = F_n(x_1, ..., x_n)$

where x_i are variables and F_i: $L^n \rightarrow L$ is monotone

Solving Equations

- Every equation system has a *unique least solution*, which is the least fixed-point of the function $F: L^n \to L^n$ defined by:

$$F(x_1,...,x_n) = (F_1(x_1,...,x_n), ..., F_n(x_1,...,x_n))$$

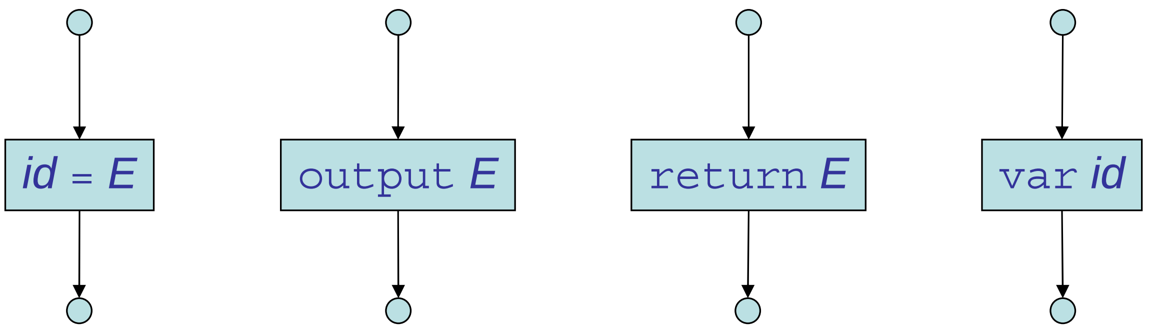
- The F function plugs into the right-hand sides
- A solution is always a fixed-point
  - this is true for any kind of equation

Control Flow Graphs

- A *control flow graph* (CFG) is a directed graph:
  - *nodes* correspond to program points
  - *edges* represent possible flow of control
- A CFG always has:
  - a single point of *entry*
  - a single point of *exit*
- Let v be a node in a CFG:
  - *pred*(v) is the set of predecessor nodes
  - *succ*(v) is the set of successor nodes

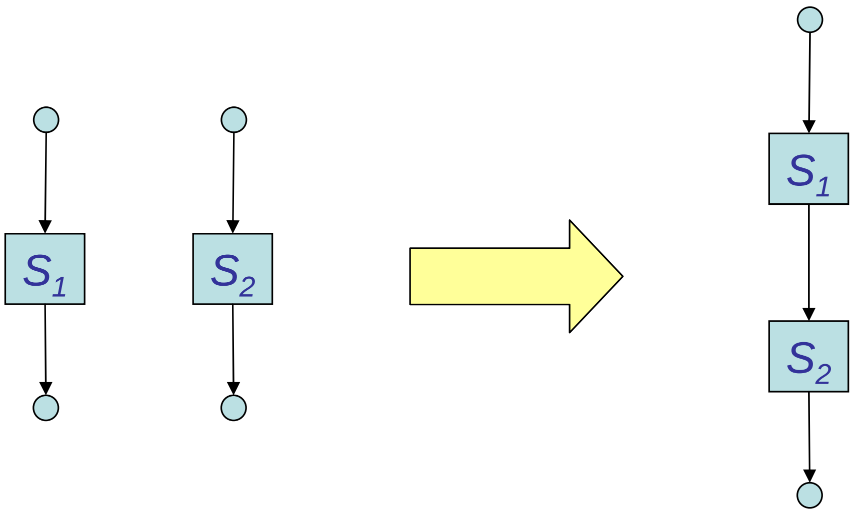
CFG Construction (1/3)

- For the simpel `while`-fragment, CFGs are constructed inductively
- CFGs for simple statements:



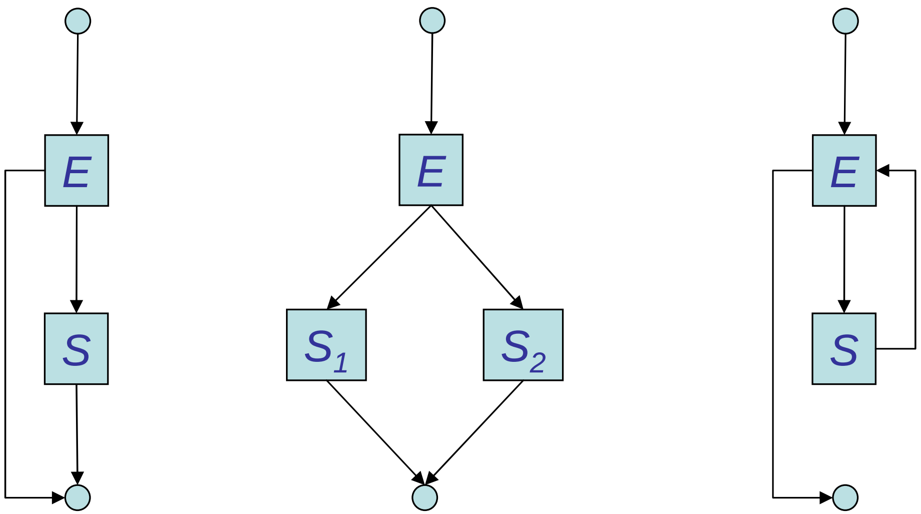
CFG Construction (2/3)

- For a statement sequence S_1 S_2:
  - eliminate the exit node of S_1 and the entry node of S_2
  - glue the statements together



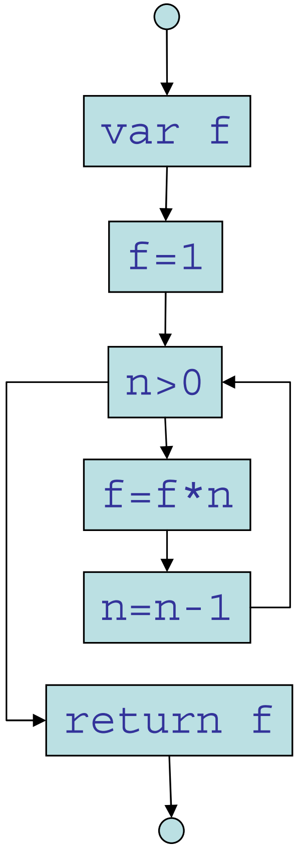
CFG Construction (3/3)

- Similarly for the other control structures:



An Example CFG

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```



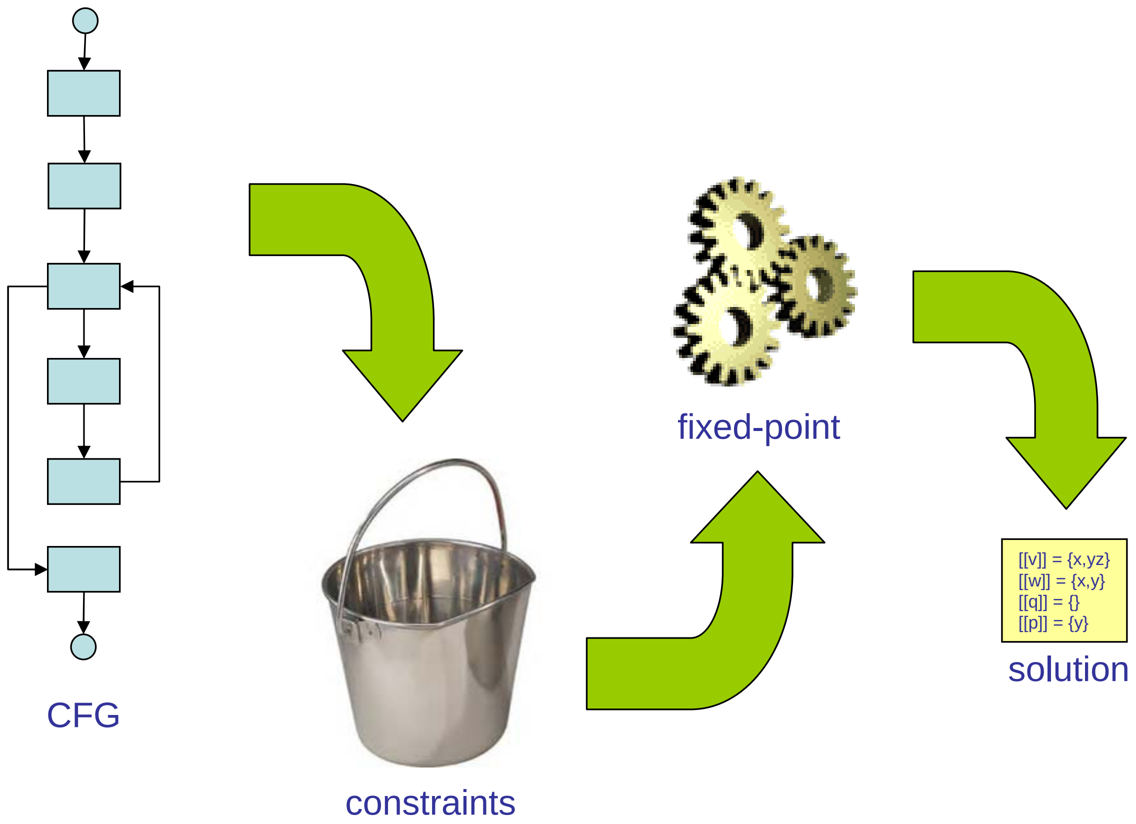
The Monotone Framework (1/2)

- A CFG to be analyzed, nodes $V = \{v_1, v_2, \ldots, v_n\}$
- A finite-height lattice L of possible answers
  - fixed or parametrized by the given program
- A variable $[[v]] \in L$ for every CFG node v
- A dataflow constraint for each syntactic construct
  - relates the value of $[[v_i]]$ to the variables for other nodes
  - typically a node is related to its neighbors
  - the constraints must be monotone functions:
    $[[v_i]] = F_i([[v_1]], [[v_2]], \ldots, [[v_n]])$

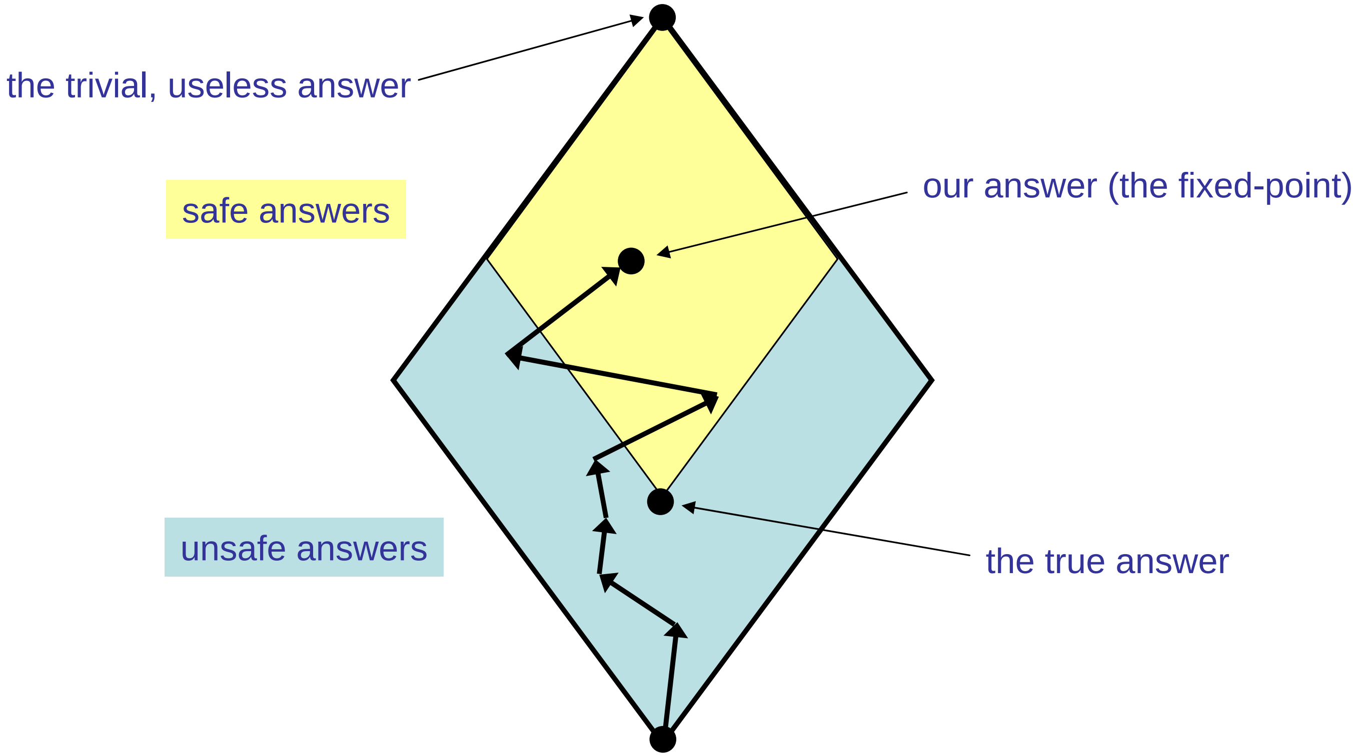
The Monotone Framework (2/2)

- Extract all constraints for the complete CFG
- Solve constraints using the fixed-point algorithm:
  - we work in the lattice L^n
  - computing the fixed-point of the combined function: $F(x_1,...,x_n) = (F_1(x_1,...,x_n), ..., F_n(x_1,...,x_n))$
- This solutions gives an answer from L for each program point

Generating and Solving Constraints



Lattice Points as Answers



The Naive Algorithm

```
x = (⊥,⊥, ..., ⊥ );
do { t = x; x = F(x); }
while (x≠t);
```

- Does not exploit any special structure

Chaotic Iteration

- Exploits the special structure of L^n

$x_1 = \perp$; ... $x_n = \perp$;
do {
  $t_1 = x_1$; ... $t_n = x_n$;
  $x_1 = F_1(x_1, ..., x_n)$;
  ...
  $x_n = F_n(x_1, ..., x_n)$;
} while $(x_1 \neq t_1 \vee ... \vee x_n \neq t_n)$;

The Worklist Algorithm (1/2)

- Exploits the special structure of right-hand sides
- Most right-hand sides are quite sparse:
  - constraints on CFG nodes do not involve all others
- Use a map:

$$dep\colon V \to 2^V$$

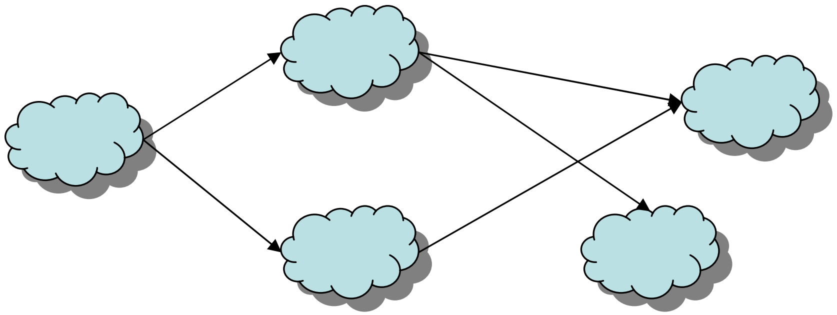
that for $v \in V$ gives the variables w where v occurs on the right-hand side of the constraint for w

The Worklist Algorithm (2/2)

$x_1 = \perp$; ... $x_n = \perp$;
q = [v_1, ..., v_n];
while (q≠[]) {
  assume q = [v_i, ...];
  y = $F_i(x_1, ..., x_n)$;
  q = q.*tail*();
  if ($y \neq x_i$) {
    for (v ∈ *dep*(v_i)) q.*append*(v);
    x_i = y;
  }
}

Further Improvements

- Use a priority queue instead of a FIFO queue:
  - find clever heuristics for priorities
- Look at the graph of dependency edges:
  - build strongly-connected components
  - solve constraints bottom-up in the resulting DAG



Liveness Analysis

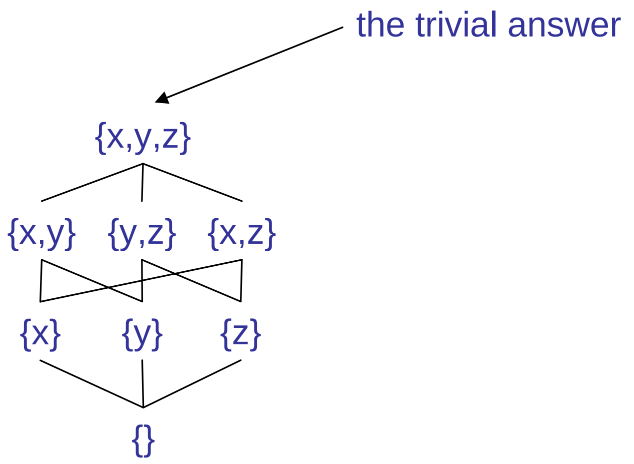
- A variable is *live* at a program point if its current value may be read in the remaining execution
- This is clearly undecidable, but the property can be conservatively approximated
- The answer ”*dead*” must be the true one
  - dead variables may be ignored

A Lattice for Liveness

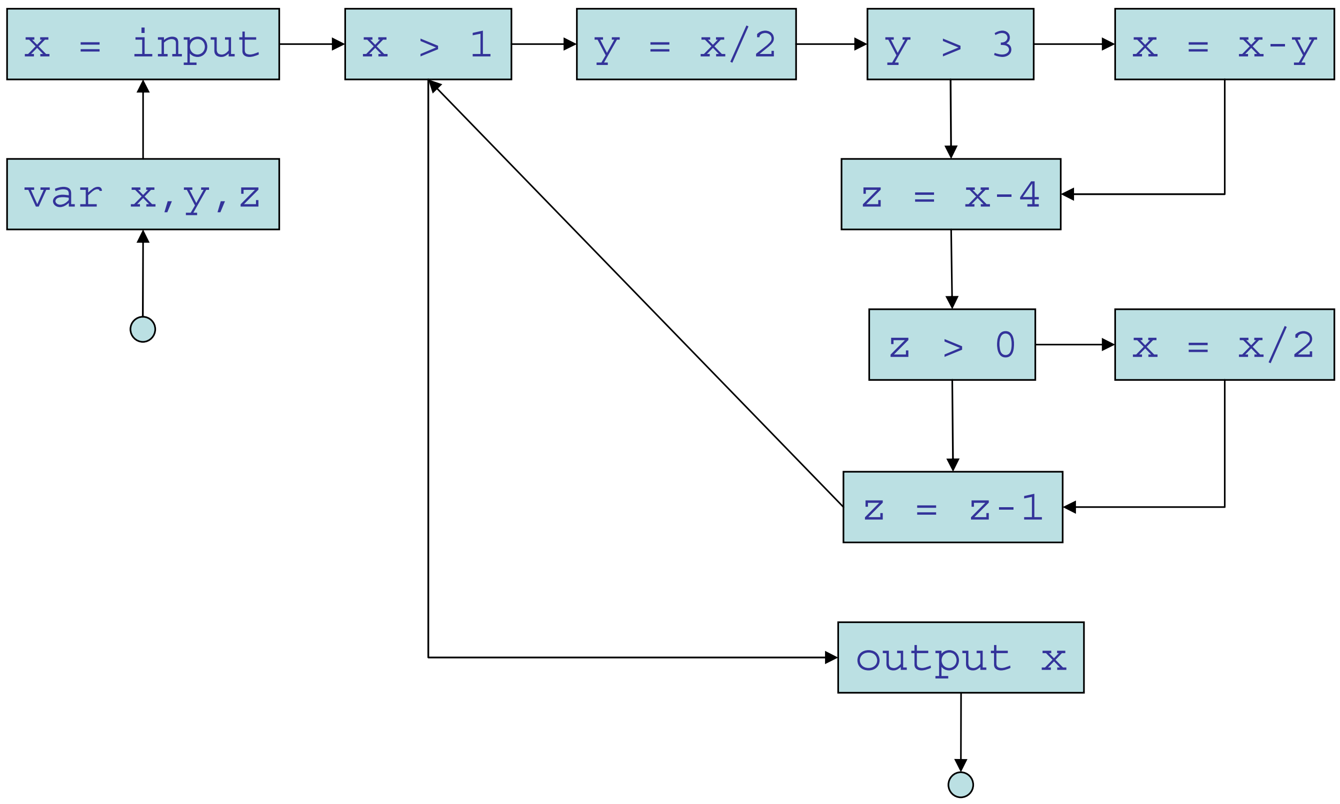
- A subset lattice of program variables

```
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;
```

$$L = (2^{\{x,y,z\}}, \subseteq)$$



The Control Flow Graph



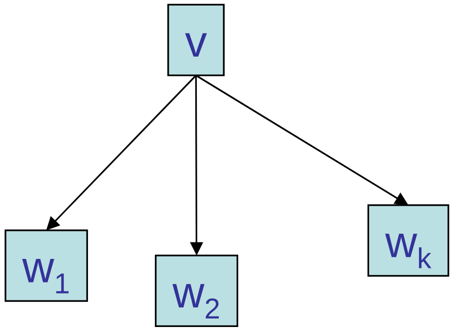
Setting Up

- For every CFG node, v, we have a variable [[v]]:
  - the subset of program variables that are live at the program point *before* v

- Since the analysis is conservative, the computed sets may be *too large*

- Auxiliary definition:

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



Liveness Constraints

- For the exit node:

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

- For conditions and `output`:

  $[[v]] = JOIN(v) \cup vars(E)$

- For assignments:

  $[[v]] = JOIN(v) \setminus \{id\} \cup vars(E)$

- For variable declarations:

  $[[v]] = JOIN(v) \setminus \{id_1, ..., id_n\}$

- For all other nodes:

  $[[v]] = JOIN(v)$

vars(*E*) = variables occurring in *E*

right-hand sides are monotone since *JOIN* is monotone

Generated Constraints

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

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

$[\![$`x>1`$]\!] = ([\![$`y=x/2`$]\!] \cup [\![$`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`$]\!] \setminus \{x\}) \cup \{x\}$

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

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

$[\![$`output x`$]\!] = [\![exit]\!] \cup \{x\}$

$[\![exit]\!] = \{\}$

Least Solution

[[*entry*]] = {}	[[`z>0`]] = {x,z}
[[`var x,y,z`]] = {}	[[`x=x/2`]] = {x,z}
[[`x=input`]] = {}	[[`z=z-1`]] = {x,z}
[[`x>1`]] = {x}	[[`output x`]] = {x}
[[`y=x/2`]] = {x}	[[*exit*]] = {}
[[`y>3`]] = {x,y}	
[[`x=x-y`]] = {x,y}	
[[`z=x-4`]] = {x}	

- Many non-trivial answers!

Optimizations

- Variables `y` and `z` are never simultaneously live

  ⇒ they can share the same variable location
- The value assigned in `z=z-1` is never read

  ⇒ the assigment can be skipped

```
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;
}
output x;
```

- better register allocation
- a few clock cycles saved

Time Complexity

- With n CFG nodes and k variables:
  - the lattice has height $k \cdot n$
- Subsets can be represented as bitvectors:
  - each lattice element has size k
  - each $\cup$, \, = operation takes time O(k)
- Each iteration uses O(n) operations:
  - each iteration takes time O($k \cdot n$)
- There are at most $k \cdot n$ iterations
- Total time complexity: O($k^2 n^2$)

Available Expressions Analysis

- A (nontrivial) expression is *available* at a program point if its current value has already been computed earlier in the execution

- The approximation includes *too few* expressions
  - the answer ”*available*” must be the true one
  - available expression may not be re-computed

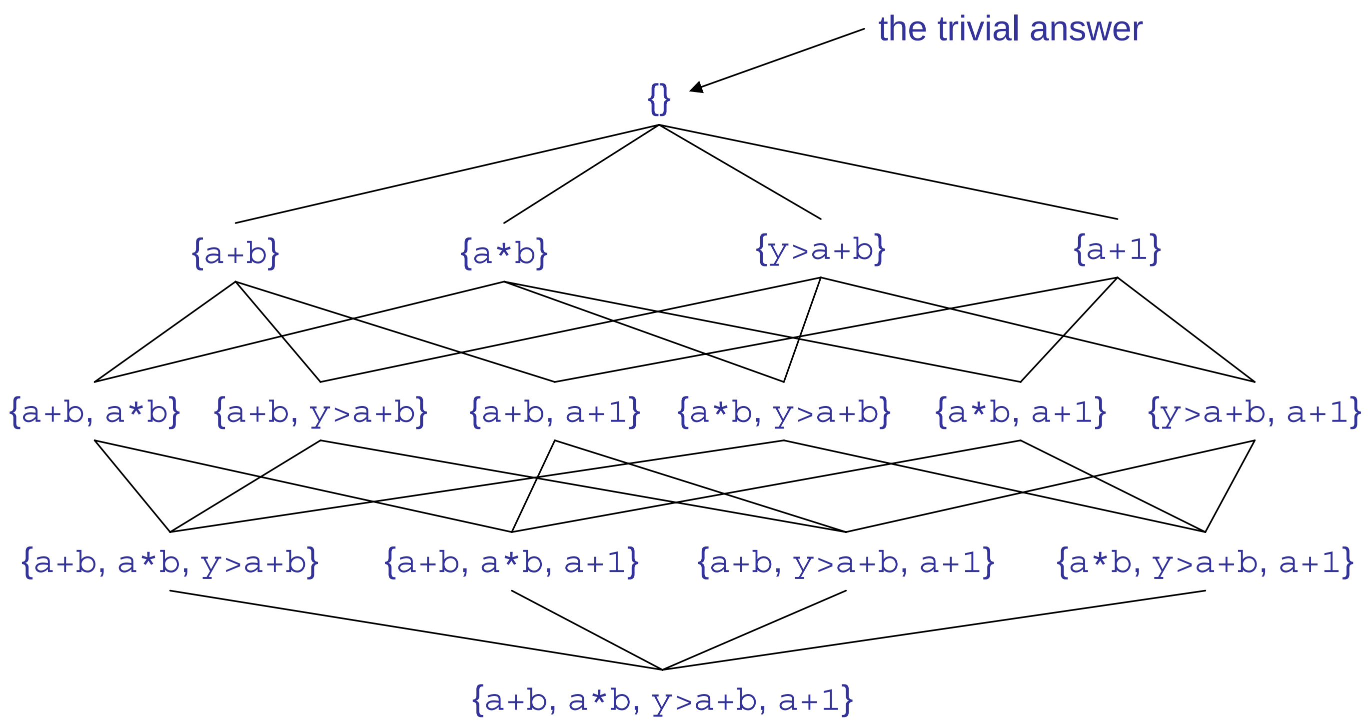
A Lattice for Available Expressions

- A reverse subset lattice of nontrivial expressions

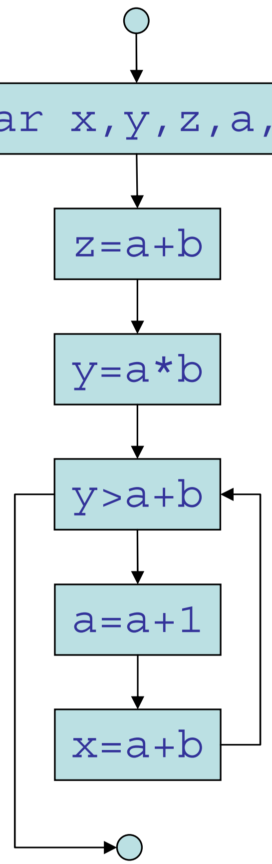
```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

$$L = (2^{\{a+b,\ a*b,\ y>a+b,\ a+1\}}, \supseteq)$$

Reverse Subset Lattice



The Flow Graph



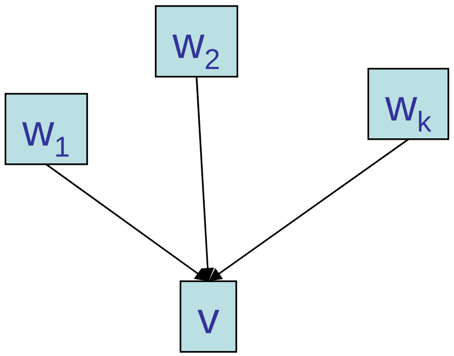
Setting Up

- For every CFG node, v, we have a variable [[v]]:
  - the subset of program variables that are available at the program point *after* v

- Since the analysis is conservative, the computed sets may be *too small*

- Auxiliary definition:

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



Auxiliary Functions

- The function $\downarrow id$ removes all expressions that contain a reference to the variable *id*

- The function $exps(E)$ is defined as:
  - $exps(intconst) = \emptyset$
  - $exps(id) = \emptyset$
  - $exps($`input`$) = \emptyset$
  - $exps(E_1\ op\ E_2) = \{E_1\ op\ E_2\} \cup exps(E_1) \cup exps(E_2)$ but don't include expressions containing `input`

Availability Constraints

- For the *entry* node:

  $[[entry]] = \{\}$

- For conditions and `output`:

  $[[v]] = JOIN(v) \cup exps(E)$

- For assignments:

  $[[v]] = (JOIN(v) \cup exps(E))\downarrow id$

- For all other nodes:

  $[[v]] = JOIN(v)$

Generated Constraints

[[*entry*]] = {}
[[`var x,y,z,a,b`]] = [[*entry*]]
[[`z=a+b`]] = *exps*(`a+b`)↓`z`
[[`y=a*b`]] = ([[`z=a+b`]] ∪ *exps*(`a*b`))↓`y`
[[`y>a+b`]] = ([[`y=a*b`]] ∩ [[`x=a+b`]]) ∪ *exps*(`y>a+b`)
[[`a=a+1`]] = ([[`y>a+b`]] ∪ *exps*(`a+1`))↓`a`
[[`x=a+b`]] = ([[`a=a+1`]] ∪ *exps*(`a+b`))↓`x`
[[*exit*]] = [[`y>a+b`]]

Least Solution

[[*entry*]] = {}
[[`var x,y,z,a,b`]] = {}
[[`z=a+b`]] = {`a+b`}
[[`y=a*b`]] = {`a+b`, `a*b`}
[[`y>a+b`]] = {`a+b`, `y>a+b`}
[[`a=a+1`]] = {}
[[`x=a+b`]] = {`a+b`}
[[*exit*]] = {`a+b`}

- Many nontrivial answers!

Optimizations

- We notice that `a+b` is available before the loop
- The program can be optimized (slightly):

```
var x,y,x,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

Very Busy Expressions Analysis

- A (nontrivial) expression is *very busy* if it will definitely be evaluated before its value changes

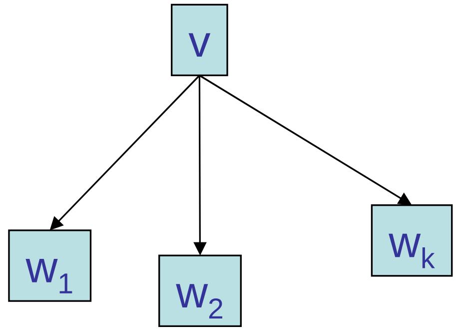
- The approximation includes *too few* expressions
  - the answer ”*very busy*” must be the true one
  - very busy expressions may be pre-computed

- Same lattice as for available expressions

Setting Up

- For every CFG node, v, we have a variable [[v]]:
  - the subset of program variables that are very busy at the program point *before* v
- Since the analysis is conservative, the computed sets may be *too small*
- Auxiliary definition:

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



Very Busy Constraints

- For the *exit* node:

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

- For conditions and `output`:

  $[[v]] = JOIN(v) \cup exps(E)$

- For assignments:

  $[[v]] = JOIN(v)\downarrow id \cup exps(E)$

- For all other nodes:

  $[[v]] = JOIN(v)$

An Example Program

```
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;
```

- The analysis shows that `a*b` is very busy

Code Hoisting

```
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;
```

- The analysis shows that `a*b` is very busy

Reaching Definitions Analysis

- The *reaching definitions* for a program point are those assignments that may define the current values of variables
- The conservative approximation may include *too many* possible assignments

A Lattice for Reaching Definitions

- The subset lattice of assignments

```
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;
```

$$L = (2^{\{\texttt{x=input}, \texttt{y=x/2}, \texttt{x=x-y}, \texttt{z=x-4}, \texttt{x=x/2}, \texttt{z=z-1}\}}, \subseteq)$$

Reaching Definitions Constraints

- For assignments:

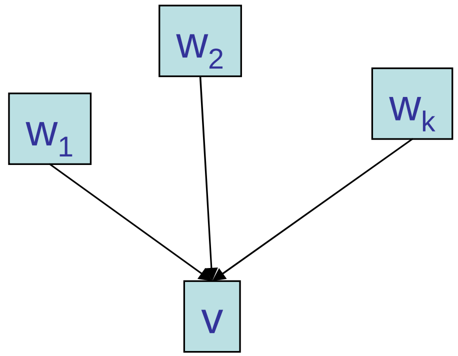
  $[[v]] = \mathit{JOIN}(v)\downarrow \mathit{id} \cup \{v\}$

- For all other nodes:

  $[[v]] = \mathit{JOIN}(v)$

- Auxiliary definition:

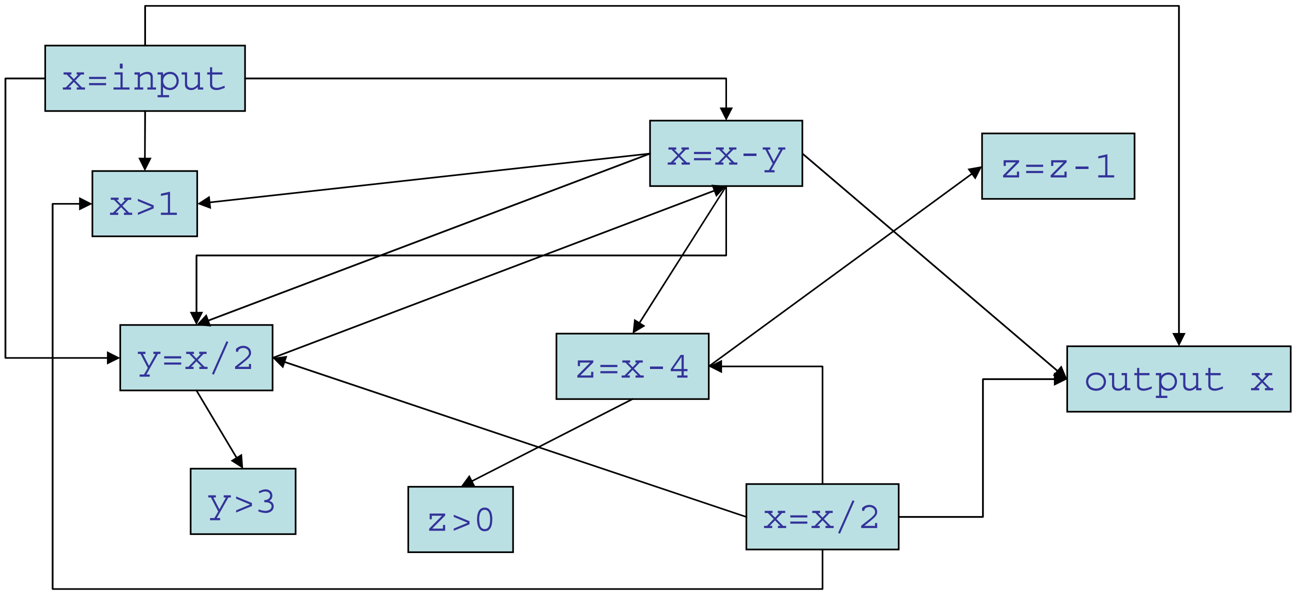
$$\mathit{JOIN}(v) = \bigcup_{w \in \mathit{pred}(v)} [[w]]$$



- The function $\downarrow \mathit{id}$ removes assignments to *id*

Def-Use Graph

- Reaching definitions define the def-use graph:
  - like a CFG but with edges from *def* to *use* nodes
  - basis for *dead code elimination* and *code motion*



Forwards vs. Backwards

- A *forwards* analysis:
  - computes information about the *past* behavior
  - available expressions, reaching definitions

- A *backwards* analysis:
  - computes information about the *future* behavior
  - liveness, very busy expressions

May vs. Must

- A *may* analysis:
  - describes information that *possibly* is true
  - an *upper* approximation
  - liveness, reaching definitions

- A *must* analysis:
  - describes information that *definitely* is true
  - a *lower* approximation
  - available expressions, very busy expressions

Classifying Analyses

	forwards	backwards
may	reaching definitions [[v]] describes state *after* v $JOIN(v) = \bigsqcup_{w \in pred(v)} [[w]] = \bigcup_{w \in pred(v)} [[w]]$	liveness [[v]] describes state *before* v $JOIN(v) = \bigsqcup_{w \in succ(v)} [[w]] = \bigcup_{w \in succ(v)} [[w]]$
must	available expressions [[v]] describes state *after* v $JOIN(v) = \bigsqcup_{w \in pred(v)} [[w]] = \bigcap_{w \in pred(v)} [[w]]$	very busy expressions [[v]] describes state *before* v $JOIN(v) = \bigsqcup_{w \in succ(v)} [[w]] = \bigcap_{w \in succ(v)} [[w]]$

Initialized Variables Analysis

- Compute for each program point those variables that have *definitely* been initialized in the *past*
- $\Rightarrow$ *forwards must analysis*
- Reverse subset lattice of all variables
- $JOIN(v) = \bigcap_{w \in pred(v)} [[w]]$
- $[[entry]] = \{\}$
- For assignments: $[[v]] = JOIN(v) \cup \{id\}$
- For all others: $[[v]] = JOIN(v)$