

Static Program Analysis

Part 4 – flow sensitive analyses

<http://cs.au.dk/~amoeller/spa/>

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University

Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

Constant propagation optimization

```
var x,y,z;  
x = 27;  
y = input,  
z = 2*x+y;  
if (x<0) { y=z-3; } else { y=12 }  
output y;
```



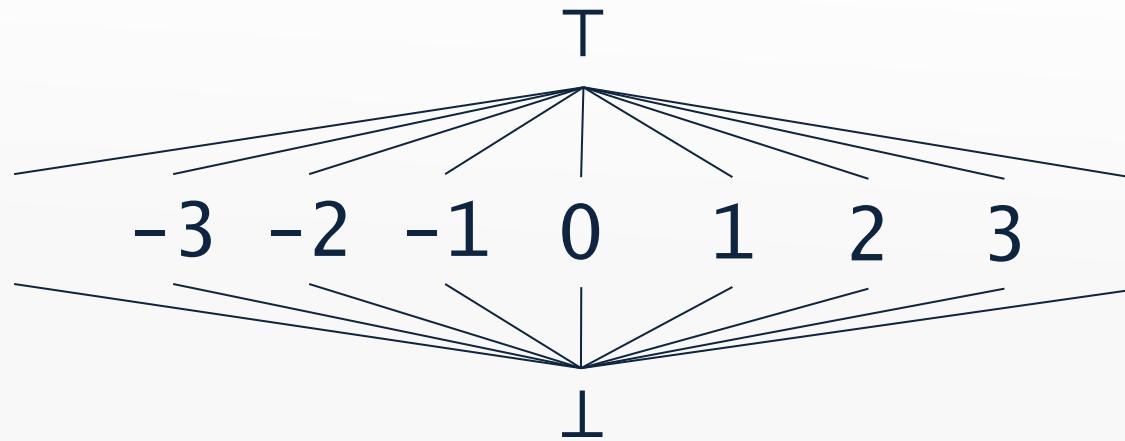
```
var x,y,z;  
x = 27;  
y = input;  
z = 54+y;  
if (0) { y=z-3; } else { y=12 }  
output y;
```



```
var y;  
y = input;  
output 12;
```

Constant propagation analysis

- Determine variables with a constant value
- Flat lattice:



Constraints for constant propagation

- Essentially as for the Sign analysis...
- Abstract operator for addition:

$$\overline{+}(n,m) = \begin{cases} \perp & \text{if } n=\perp \vee m=\perp \\ T & \text{else if } n=T \vee m=T \\ n+m & \text{otherwise} \end{cases}$$

Agenda

- Constant propagation analysis
- **Live variables analysis**
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

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 analysis must only answer “*dead*” if the variable is really dead
 - no need to store the values of dead variables

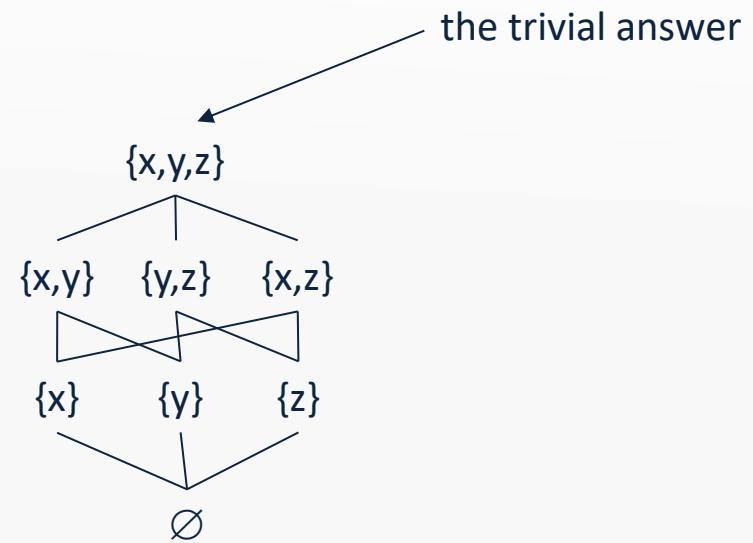


A lattice for liveness

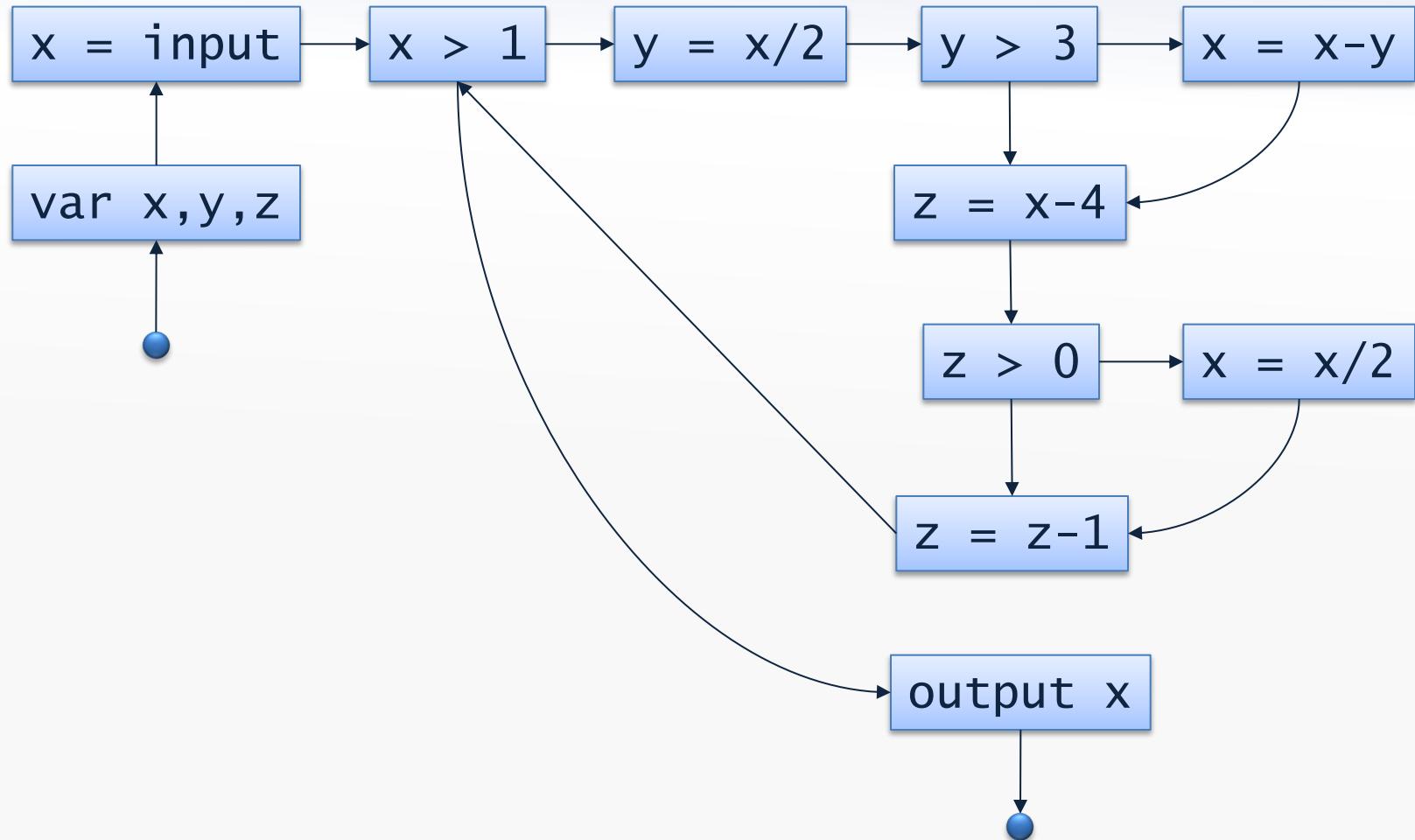
A powerset lattice of program variables

```
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;
```

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



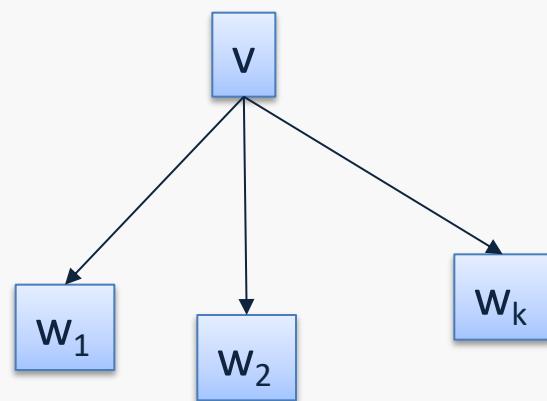
The control flow graph



Setting up

- For every CFG node, v , we have a variable $\llbracket v \rrbracket$:
 - 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:

$$JOIN(v) = \bigcup_{w \in succ(v)} \llbracket w \rrbracket$$



Liveness constraints

- For the exit node:

$$\text{vars}(E) = \text{variables occurring in } E$$

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

- For conditions and output:

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

- For assignments:

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

- For variable declarations:

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

- For all other nodes:

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

right-hand sides are monotone
since JOIN is monotone, and ...

Generated constraints

```
[[var x,y,z]] = [[z=input]] \ {x,y,z}
[[x=input]] = [[x>1]] \ {x}
[[x>1]] = ([[y=x/2]] ∪ [[output x]]) ∪ {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z}
[[output x]] = [[exit]] ∪ {x}
[[exit]] = ∅
```

Least solution

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

$\llbracket \text{var } x, y, z \rrbracket = \emptyset$

$\llbracket x = \text{input} \rrbracket = \emptyset$

$\llbracket x > 1 \rrbracket = \{x\}$

$\llbracket y = x/2 \rrbracket = \{x\}$

$\llbracket y > 3 \rrbracket = \{x, y\}$

$\llbracket x = x - y \rrbracket = \{x, y\}$

$\llbracket z = x - 4 \rrbracket = \{x\}$

$\llbracket z > 0 \rrbracket = \{x, z\}$

$\llbracket x = x/2 \rrbracket = \{x, z\}$

$\llbracket z = z - 1 \rrbracket = \{x, z\}$

$\llbracket \text{output } x \rrbracket = \{x\}$

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

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 assignment 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 (for the naive algorithm)

- With n CFG nodes and k variables:
 - the lattice L^n has height $k \cdot n$
 - so there are at most $k \cdot n$ iterations
- Subsets of Vars (the variables in the program) can be represented as bitvectors:
 - each element has size k
 - each $\cup, \setminus, =$ operation takes time $O(k)$
- Each iteration uses $O(n)$ bitvector operations:
 - so each iteration takes time $O(k \cdot n)$
- Total time complexity: $O(k^2 n^2)$
- Exercise: what is the complexity for the worklist algorithm?

Agenda

- Constant propagation analysis
- Live variables analysis
- **Available expressions analysis**
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables analysis

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 generally includes *too few* expressions
 - the analysis can only report “*available*” if the expression is definitely available
 - no need to re-compute available expressions (e.g. common subexpression elimination)

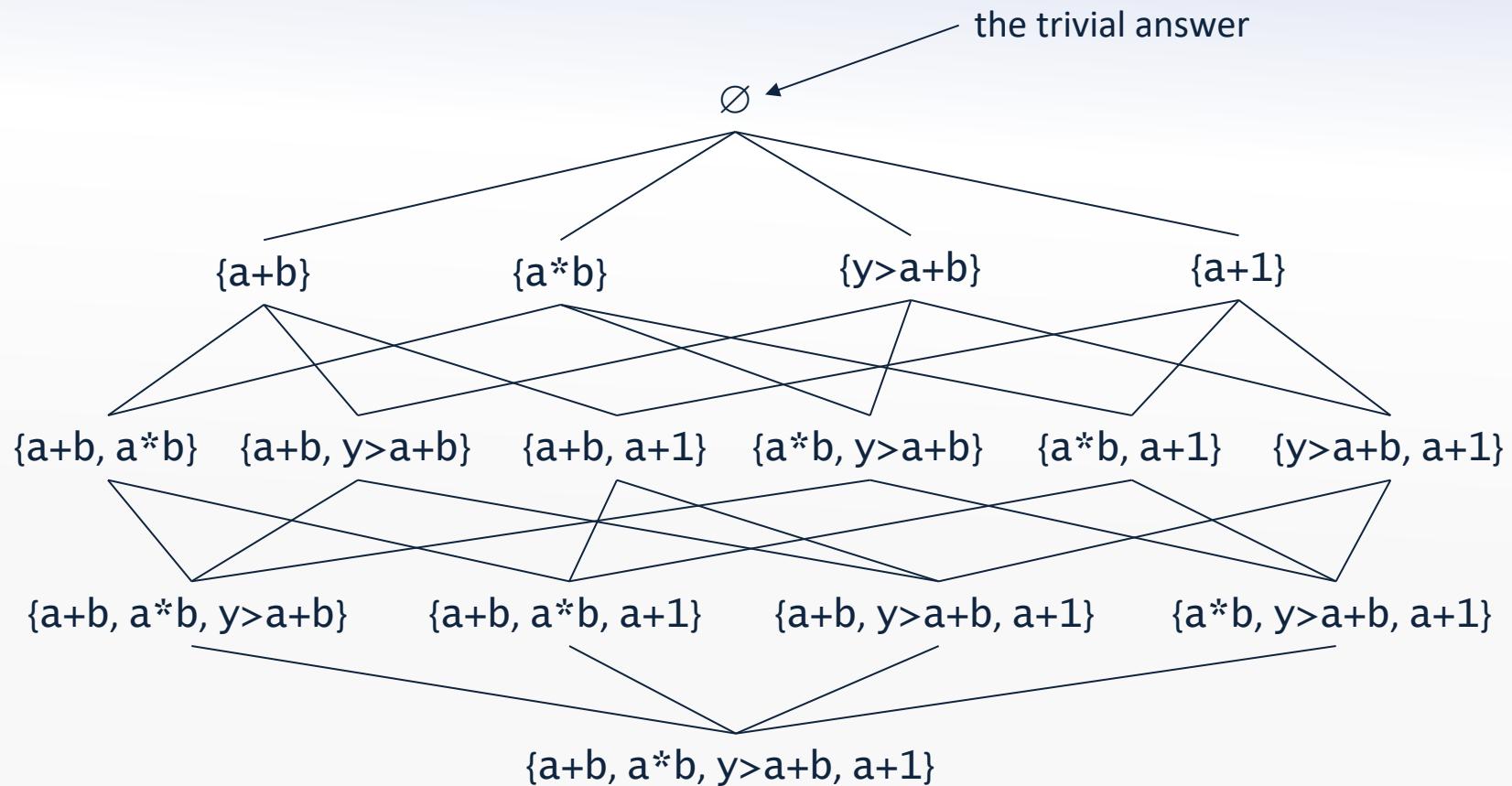
A lattice for available expressions

A reverse powerset lattice of nontrivial expressions

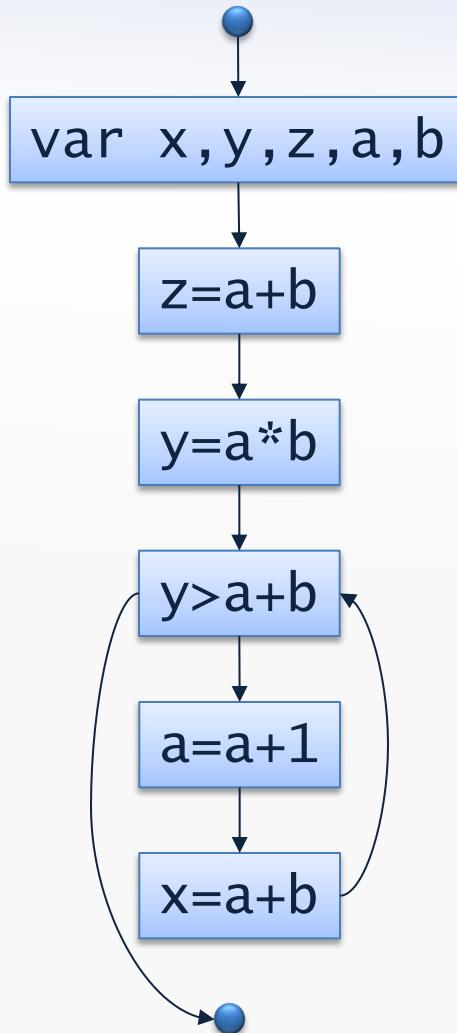
```
var x,y,z,a,b;  
z = a+b;  
y = a*b;  
while (y > a+b) {  
    a = a+1;  
    x = a+b;  
}
```

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

Reverse powerset lattice



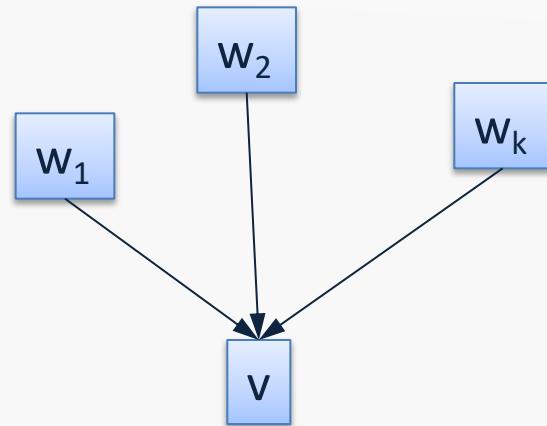
The flow graph



Setting up

- For every CFG node, v , we have a variable $\llbracket v \rrbracket$:
 - 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:

$$JOIN(v) = \bigcap_{w \in pred(v)} \llbracket w \rrbracket$$



Auxiliary functions

- The function $X \downarrow x$ removes all expressions from X that contain a reference to the variable x
- The function $\text{exp}(E)$ is defined as:
 - $\text{exp}(\text{intconst}) = \emptyset$
 - $\text{exp}(x) = \emptyset$
 - $\text{exp}(\text{input}) = \emptyset$
 - $\text{exp}(E_1 \text{ op } E_2) = \{E_1 \text{ op } E_2\} \cup \text{exp}(E_1) \cup \text{exp}(E_2)$
but don't include expressions containing `input`

Availability constraints

- For the *entry* node:

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

- For conditions and output:

$$[\![\text{if } (E)]\!] = [\![\text{output } E]\!] = \text{JOIN}(v) \cup \text{exp}(E)$$

- For assignments:

$$[\![x = E]\!] = (\text{JOIN}(v) \cup \text{exp}(E)) \downarrow x$$

- For any other node v :

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

Generated constraints

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

$\llbracket \text{var } x, y, z, a, b \rrbracket = \llbracket \text{entry} \rrbracket$

$\llbracket z=a+b \rrbracket = \text{exp}(a+b) \downarrow z$

$\llbracket y=a*b \rrbracket = (\llbracket z=a+b \rrbracket \cup \text{exp}(a*b)) \downarrow y$

$\llbracket y>a+b \rrbracket = (\llbracket y=a*b \rrbracket \cap \llbracket x=a+b \rrbracket) \cup \text{exp}(y>a+b)$

$\llbracket a=a+1 \rrbracket = (\llbracket y>a+b \rrbracket \cup \text{exp}(a+1)) \downarrow a$

$\llbracket x=a+b \rrbracket = (\llbracket a=a+1 \rrbracket \cup \text{exp}(a+b)) \downarrow x$

$\llbracket \text{exit} \rrbracket = \llbracket y>a+b \rrbracket$

Least solution

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

$\llbracket \text{var } x, y, z, a, b \rrbracket = \emptyset$

$\llbracket z = a + b \rrbracket = \{a + b\}$

$\llbracket y = a * b \rrbracket = \{a + b, a * b\}$

$\llbracket y > a + b \rrbracket = \{a + b, y > a + b\}$

$\llbracket a = a + 1 \rrbracket = \emptyset$

$\llbracket x = a + b \rrbracket = \{a + b\}$

$\llbracket \text{exit} \rrbracket = \{a + b\}$

Again, 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,ap1usb;
ap1usb = a+b;
z = ap1usb;
y = a*b;
while (y > ap1usb) {
    a = a+1;
    ap1usb = a+b;
    x = ap1usb;
}
```

Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- **Very busy expressions analysis**
- Reaching definitions analysis
- Initialized variables analysis

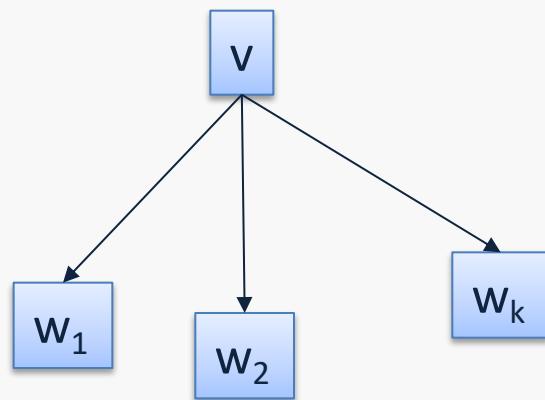
Very busy expressions analysis

- A (nontrivial) expression is *very busy* if it will definitely be evaluated before its value changes
- The approximation generally includes *too few* expressions
 - the answer “*very busy*” must be the true one
 - very busy expressions may be pre-computed (e.g. loop hoisting)
- Same lattice as for available expressions

Setting up

- For every CFG node, v , we have a variable $\llbracket v \rrbracket$:
 - 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:

$$JOIN(v) = \bigcap_{w \in succ(v)} \llbracket w \rrbracket$$



Very busy constraints

- For the *exit* node:

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

- For conditions and output:

$$\llbracket \text{if } (E) \rrbracket = \llbracket \text{output } E \rrbracket = \text{JOIN}(v) \cup \text{exp}(E)$$

- For assignments:

$$\llbracket x = E \rrbracket = \text{JOIN}(v) \downarrow x \cup \text{exp}(E)$$

- For all other nodes:

$$\llbracket v \rrbracket = \text{JOIN}(v)$$

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;
```

Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- **Reaching definitions analysis**
- Initialized variables analysis

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 powerset lattice of assignments

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

```
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;
```

Reaching definitions constraints

- For assignments:

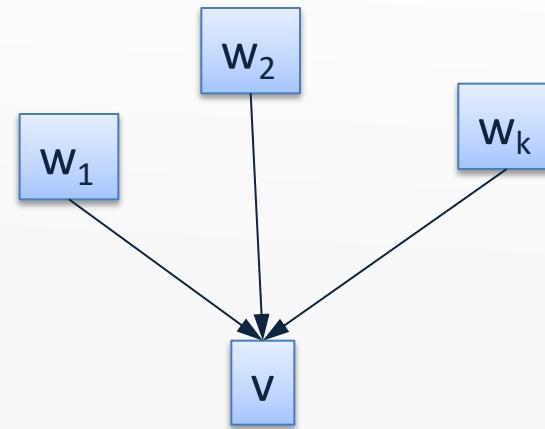
$$[\![x = E]\!] = JOIN(v) \downarrow x \cup \{ x = E \}$$

- For all other nodes:

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

- Auxiliary definition:

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

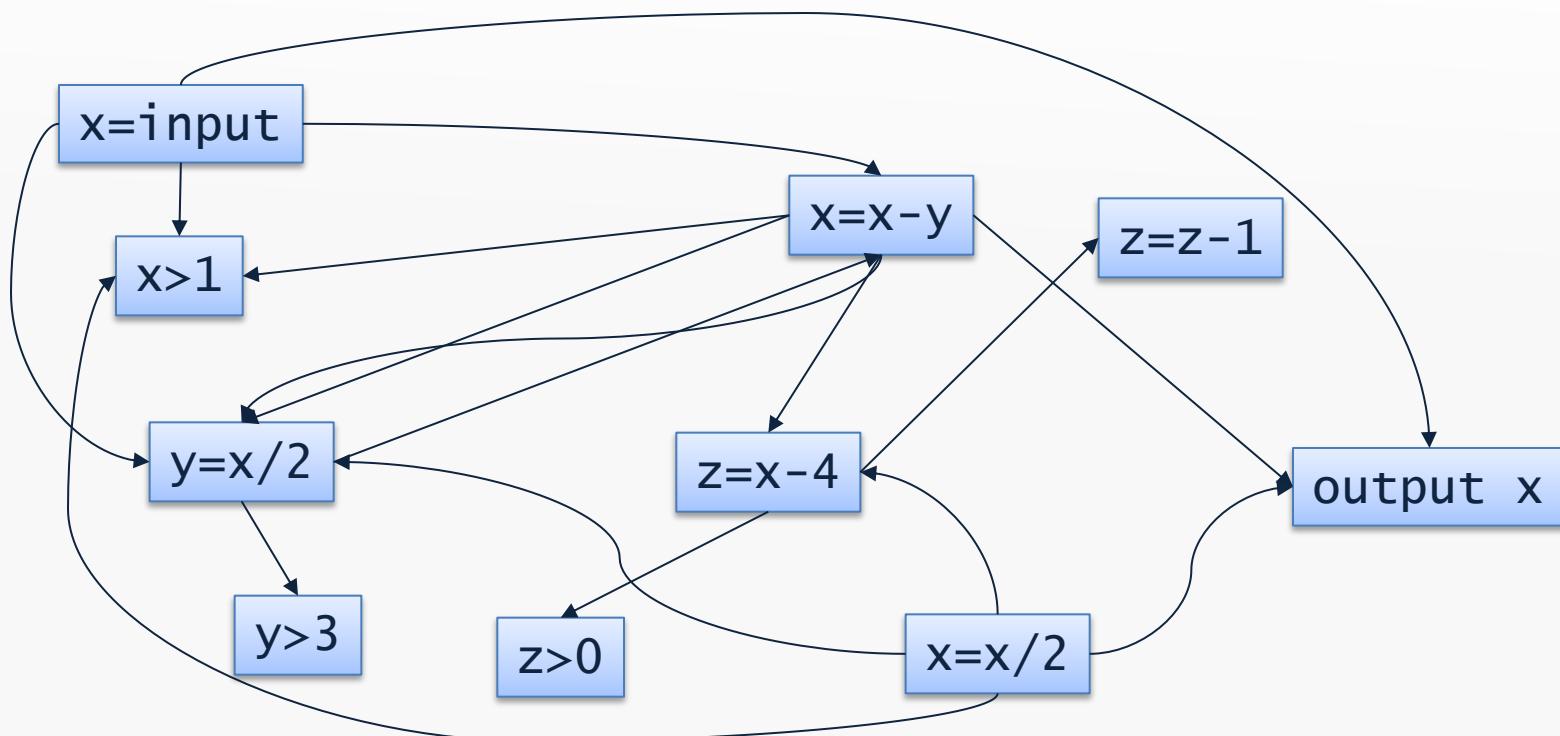


- The function $X \downarrow x$ removes assignments to x from X

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*



Forward vs. backward

- A *forward* analysis:
 - computes information about the *past* behavior
 - examples: available expressions, reaching definitions
- A *backward* analysis:
 - computes information about the *future* behavior
 - examples: liveness, very busy expressions

May vs. must

- A *may* analysis:
 - describes information that is *possibly* true
 - an *over*-approximation
 - examples: liveness, reaching definitions
- A *must* analysis:
 - describes information that is *definitely* true
 - an *under*-approximation
 - examples: available expressions, very busy expressions

Classifying analyses

	forward	backward
may	<p>example: reaching definitions</p> <p>$\llbracket v \rrbracket$ describes state after v</p> $\text{JOIN}(v) = \bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket$	<p>example: liveness</p> <p>$\llbracket v \rrbracket$ describes state before v</p> $\text{JOIN}(v) = \bigsqcup_{w \in \text{succ}(v)} \llbracket w \rrbracket = \bigcup_{w \in \text{succ}(v)} \llbracket w \rrbracket$
must	<p>example: available expressions</p> <p>$\llbracket v \rrbracket$ describes state after v</p> $\text{JOIN}(v) = \bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket = \bigcap_{w \in \text{pred}(v)} \llbracket w \rrbracket$	<p>example: very busy expressions</p> <p>$\llbracket v \rrbracket$ describes state before v</p> $\text{JOIN}(v) = \bigsqcup_{w \in \text{succ}(v)} \llbracket w \rrbracket = \bigcap_{w \in \text{succ}(v)} \llbracket w \rrbracket$

Agenda

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- **Initialized variables analysis**

Initialized variables analysis

- Compute for each program point those variables that have *definitely* been initialized in the *past*
- (Called *definite assignment* analysis in Java and C#)
- \Rightarrow *forward must analysis*
- Reverse powerset lattice of all variables

$$JOIN(v) = \bigcap_{w \in pred(v)} \llbracket w \rrbracket$$

- For assignments: $\llbracket x = E \rrbracket = JOIN(v) \cup \{x\}$
- For all others: $\llbracket v \rrbracket = JOIN(v)$