CS293S Iterative Data-Flow Analysis

Yufei Ding

Review: Computing Available Expressions

The Big Picture

1. Build a control-flow graph
2. Gather the initial data: DEEXPR(b) & EXPRKILL(b)
3. Propagate information around the graph, evaluating the equation

$$\underbrace{\textsc{Avail}(b)}_{\text{Entry point of block } b} = \bigcap_{x \in pred(b)} \underbrace{\left(\textsc{DEExpr}(x) \cup \left(\textsc{Avail}(x) \cap \overline{\textsc{ExprKill}(x)}\right)\right)}_{\text{Exit point of block } x}$$

Works for loops through an iterative algorithm: finding the fixed-point.

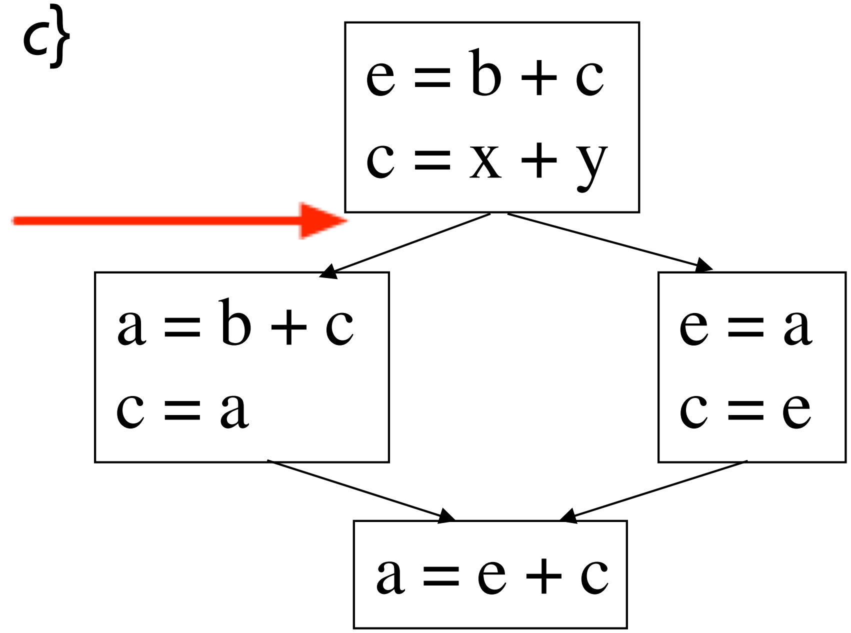
All data-flow problems are solved, essentially, this way.

Live Variables

- A variable v is live at a point p if there is a path from p to a use of v, and that path does not contain a redefinition of v

- Example: **I:** *a* <- *b* + *c*
  - A statement/instruction **I** is a definition of a variable *v* if it may write to <u>*v*</u>. def[**I**] = *a*
  - A statement is a use of variable *v* if it may read from *v*. use[**I**] = {*b*, *c*}

Point p



Live Variables

- A variable **v** is live at point **p** if and only if there is a path from **p** to a use of **v** along which **v** is not redefined.
- Usage
  - Global register allocation
  - Improve SSA construction
    - reduce # of f-functions
  - Detect references to uninitialized variables & defined but not used variables
  - Drive transformations
    - useless-store elimination

Live Variables at Special Points

- For an instruction I
  - LIVEIN[I]: live variables at program point before I
  - LIVEOUT[I]: live variables at program point after I

- For a basic block B
  - LI VEIN[B]: live variables at the entry point of B
  - LIVEOUT[B]: live variables at the exit point of B

- If I = first instruction in B, then LIVEIN[B] = LIVEIN[I]
- If I = last instruction in B, then LIVEOUT[B] = LIVEOUT[I]

How to Compute Liveness?

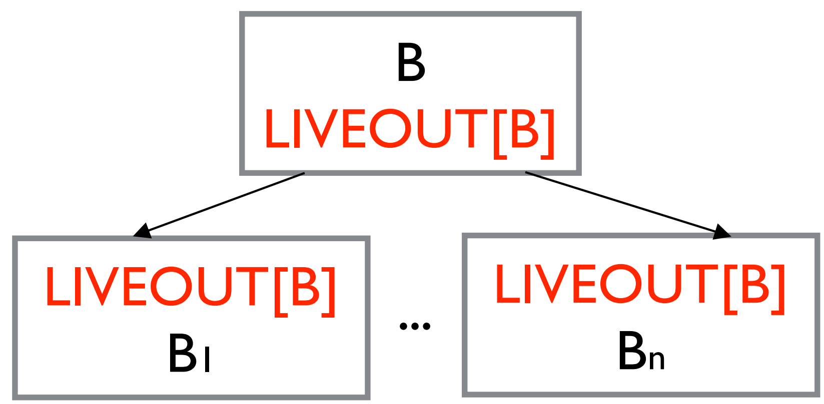
- Question 1: for each instruction I, what is the relation between LIVEIN[I] and LIVEOUT[I]?

LIVEIN[I]
I
LIVEOUT[I]

- Question 2: for each basic block B, what is the relation between LIVEIN[B] and LIVEOUT[B]?

LIVEIN[B]
B
LIVEOUT[B]

- Question 3: for each basic block B with successor blocks B1, ..., Bn, what is the relation between LIVEOUT[B] and LIVEOUT[B1], ..., LIVEOUT[Bn]?



Part 1: Analyze Instructions

- Question: what is the relation between the sets of live variables before and after an instruction I?

LIVEIN[I]
I
LIVEOUT[I]

Examples:

LIVEIN[I] = {y,z}	LIVEIN[I] = {y,z,t}	LIVEIN[I] = {x,t}
x = y+z;	x = y+z;	x = x+1;
LIVEOUT[I] = {z}	LIVEOUT[I] = {x,t}	LIVEOUT[I] = {x,t}

… is there a general rule?

Analyze Instructions

- Two Rules:
  - Each variable live after I is also live before I, unless I defines (writes) it.
  - Each variable that I uses (reads) is also live before instruction I

- Mathematically:
  LIVEIN[I] = (LIVEOUT[I] – def[I]) ∪ use[I]
  where: def[I] = variables defined (written) by instruction I
  use[I] = variables used (read) by instruction I

- The information flows **backward!**

Analyze block

- Example: block B with three instructions I1, I2, I3:
  - Live1 = LIVEIN[B] = LIVEIN[I1]
  - Live2 = LIVEOUT[I1] = LIVEIN[I2]
  - Live3 = LIVEOUT[I2] = LIVEIN[I3]
  - Live4 = LIVEOUT[I3] = LIVEOUT[B]
- Relation between Live sets:
  - Live1 = (Live2-{x}) ∪ {y}
  - Live2 = (Live3-{y}) ∪ {x,z}
  - Live3 = (Live4-{t}) ∪ {d}

Block B

```
     Live1
I1   x = y + 1
     Live 2
I2   y = x * z
     Live 3
I3   t = d
     Live 4
```

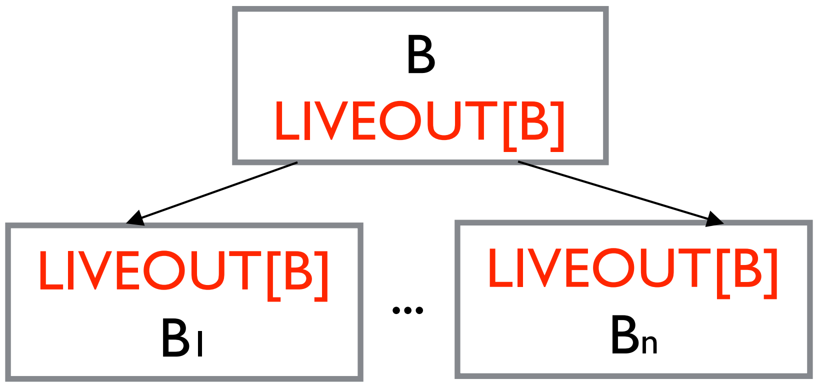
Analyze Block

- Two Rules:
  - Each variable live after B is also live before B, unless B defines (writes) it.
  - Each variable v that B uses (reads) before any redefinition in B is also live before B

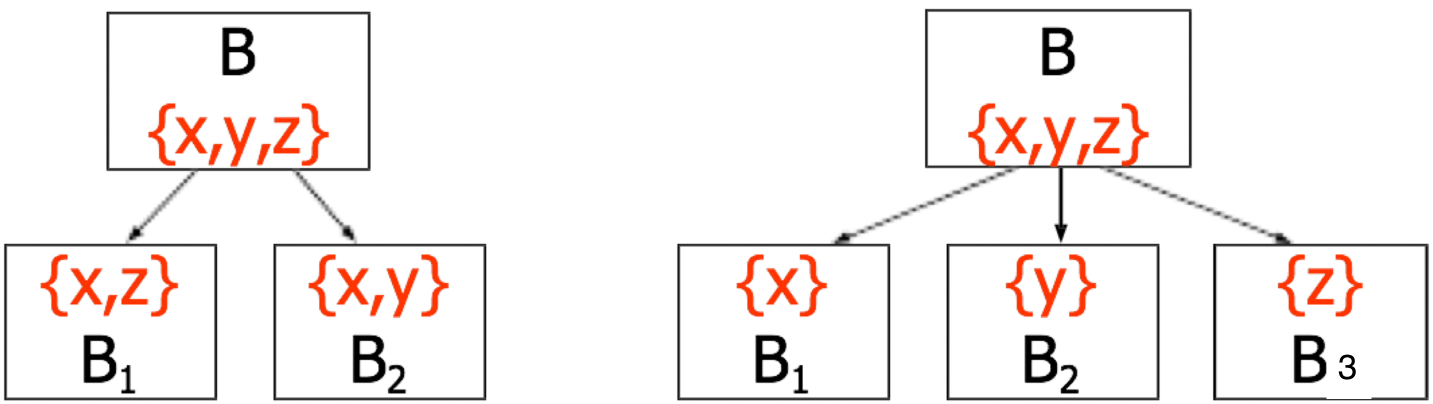
- Mathematically:
  LIVEIN[B] = (LIVEOUT[B] – VarKill(B)) ∪ UEVar(B)
  where:
  - VARKILL(B) = variables that are defined in B
  - UEVAR(B) variables that are used in B before any redefinition in B, i.e., <u>u</u>pward-<u>e</u>xposed <u>var</u>iables

Analyze CFG

- Question: for each basic block B with successor blocks B1, ..., Bn, what is the relation between LIVEOUT[B] and LIVEIN[B1], ..., LIVEIN[Bn]?



- Example:



- General rule?

Analyze CFG

- Rule: A variables is live at end of block B if it is live at the beginning of one (or more) successor blocks
- Mathematically:

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} LIVEIN[B']$$

$$= \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

- Again, information flows backward: from successors B’ of B to basic block

Equations for Live Variables

- LIVEOUT(B) contains the name of every variable that is live on exit from n (a basic block)
- UEVAR(B) contains the upward-exposed variables in n, i.e. those that are used in n before any redefinition in n
- VARKILL(B) contains all the variables that are defined in n
- Equation (n_f is the exit node of the CFG)

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Note: $A\text{-}B = A \cup \overline{B}$

Three Steps in Data-Flow Analysis

- Build a CFG
- Gather the initial information for each block (i.e., (UEVAR and VARKILL))
- Use an iterative fixed-point algorithm to propagate information around the CFG

Algorithm

// Get initial sets

```
for each block b
   UEVAR(b) = Ø
   VARKILL(b) = Ø
   for i=1 to number of instr in b
      (assuming inst I is “x= y op z”)
        if y ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {y}
        if z ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {z}
        VARKILL(b) = VARKILL(b) ∪ {x}
```

// update LiveOut version 1

```
set LIVEOUT(bi) to Ø for all blocks
Worklist ← {all blocks}
while (Worklist ≠ Ø)
        remove a block b from Worklist
        recompute LIVEOUT(b)
        if LIVEOUT(b) changed then
           Worklist ← Worklist ∪ pred(b)
```

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Algorithm

// Get initial sets

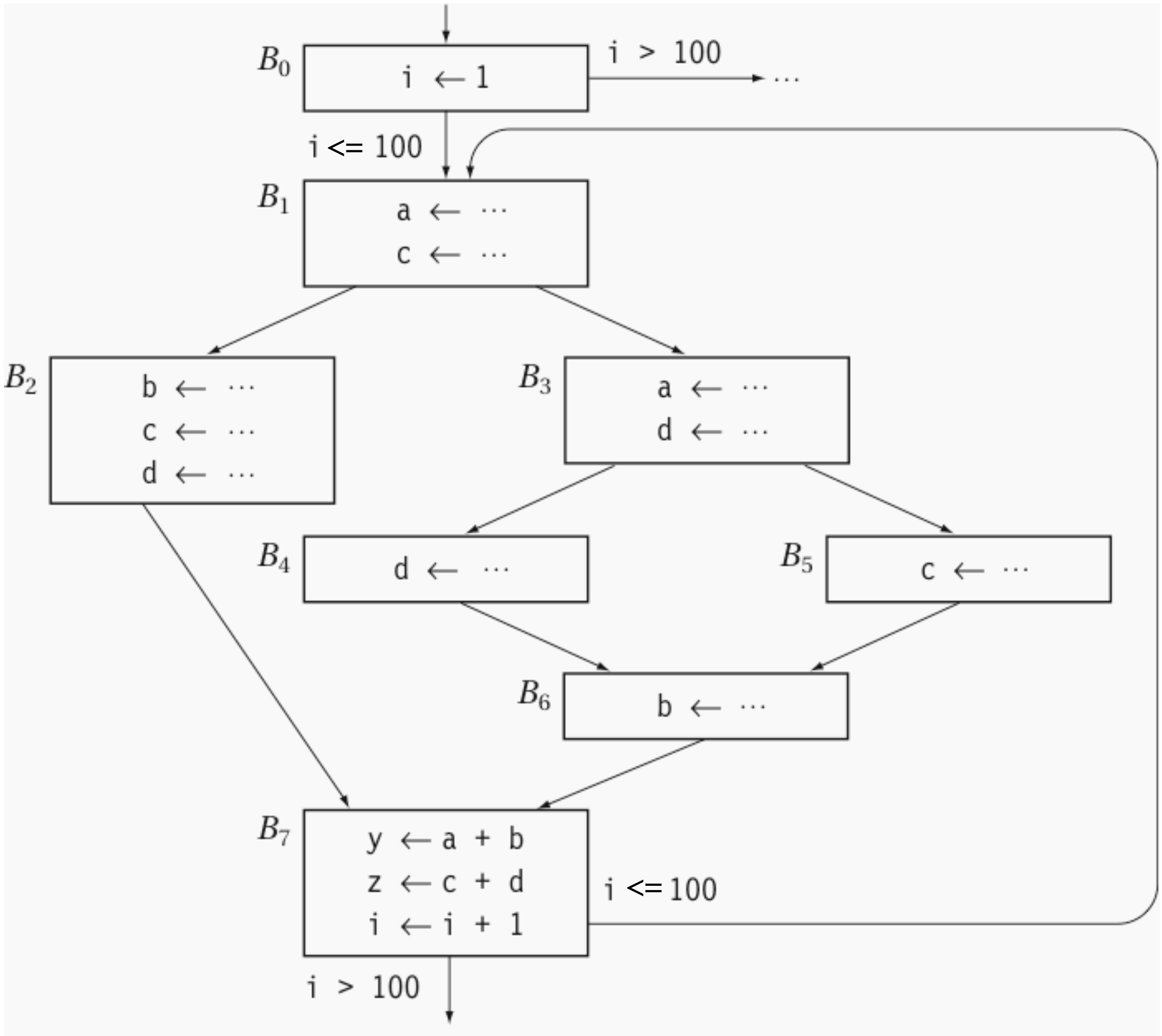
```
for each block b
    UEVAR(b) = Ø
    VARKILL(b) = Ø
    for i=1 to number of instr in b
        (assuming inst I is “x= y op z”)
          if y ∉VARKILL(b) then
             UEVAR(b) = UEVAR(b) ∪ {y}
          if z ∉VARKILL(b) then
             UEVAR(b) = UEVAR(b) ∪ {z}
          VARKILL(b) = VARKILL(b) ∪ {x}
```

// update LiveOut version2

```
set LIVEOUT(bi) to Ø for all blocks
changed = true
while (changed)
    changed = false
    for i = 1 to N (number of blocks)
        recompute LIVEOUT(i)
        if LIVEOUT(i) changed then
            changed = true
```

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Example



B_0
i ← 1
i > 100
...
i <= 100
B_1
a ← ...
c ← ...
B_2
b ← ...
c ← ...
d ← ...
B_3
a ← ...
d ← ...
B_4
d ← ...
B_5
c ← ...
B_6
b ← ...
B_7
y ← a + b
z ← c + d
i ← i + 1
i <= 100
i > 100
...

Example (cont.)

	B0	B1	B2	B3	B4	B5	B6	B7
UEVar	Ø	Ø	Ø	Ø	Ø	Ø	Ø	a,b,c,d,i
VarKill	i	a, c	b, c, d	a, d	d	c	b	y, z, i

Example (cont.)

Can the algorithm converge in fewer iterations?

LiveOut (b)

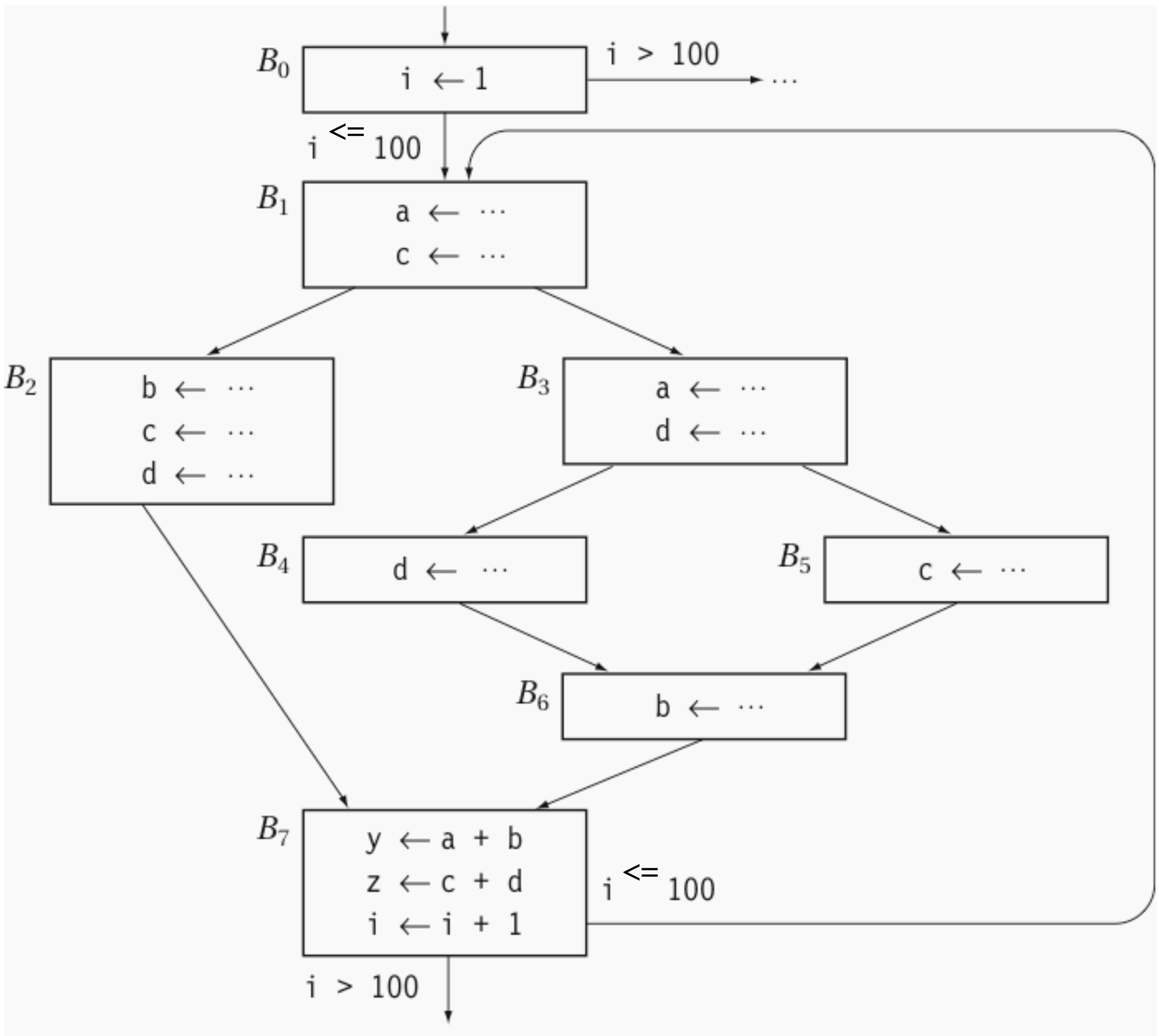
iteration	B0	B1	B2	B3	B4	B5	B6	B7
0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
1	Ø	Ø	a,b,c,d,i	Ø	Ø	Ø	a,b,c,d,i	Ø
2	Ø	a,i	a,b,c,d,i	Ø	a,c,d,i	a,c,d,i	a,b,c,d,i	i
3	i	a,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
4	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
5	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Preorder: parents first.

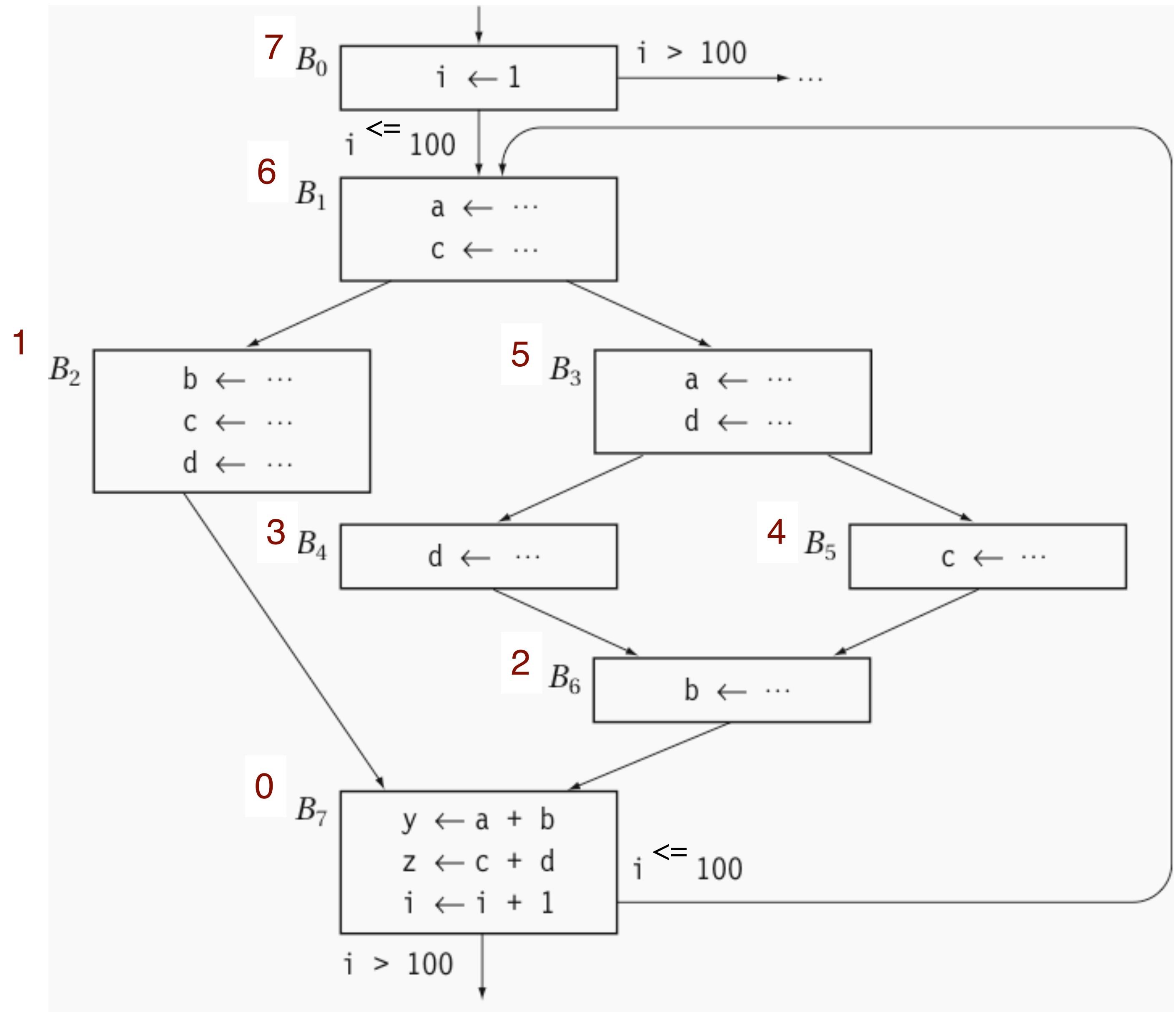
w/o considering backedges.



$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Postorder: children first.

w/o considering backedges.



Algorithm

// Get initial sets

```
for each block b
   UEVAR(b) = Ø
   VARKILL(b) = Ø
   for i=1 to number of instr in b
      (assuming inst I is "x= y op z")
        if y ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {y}
        if z ∉VARKILL(b) then
           UEVAR(b) = UEVAR(b) ∪ {z}
          VARKILL(b) = VARKILL(b) ∪ {x}
```

// update LiveOut version2

```
set LIVEOUT(bi) to Ø for all blocks
changed = true
while (changed)
   changed = false
   for i = 0 to N
      // different orders could be used
      recompute LIVEOUT(i)
      if LIVEOUT(i) changed then
         changed = true
```

$$LIVEOUT[B] = \bigcup_{B' \in succ(B)} ((LIVEOUT[B'] - VARKILL(B')) \bigcup UEVAR(B'))$$

Postorder (5 iterations becomes 3)

iteration	B0	B1	B2	B3	B4	B5	B6	B7
0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
1	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	Ø
2	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i
3	i	a,c,i	a,b,c,d,i	a,c,d,i	a,c,d,i	a,c,d,i	a,b,c,d,i	i

Order

Parent relation does not consider backedges.

- **Preorder**: visit parents before children.
  - also called reverse postorder
- **Postorder**: visit children before parents.

- Forward problem (e.g., AVAIL):
  - A node needs the info of its predecessors.
  - Preorder on CFG.
- Backward problem (e.g., LIVEOUT):
  - A node needs the info of its successors.
  - Postorder on CFG.

Comparison with AVAIL

- Common
  - Three steps
  - Fixed-point algorithm finds solution
- Differences
  - AVAIL: domain is a set of expressions
    LIVEOUT: domain is a set of variables

    Domain
  - AVAIL: forward problem
    LIVEOUT: backward problem

    Direction
  - AVAIL: intersection of all paths (all path problem)
    - Also called Must Problem

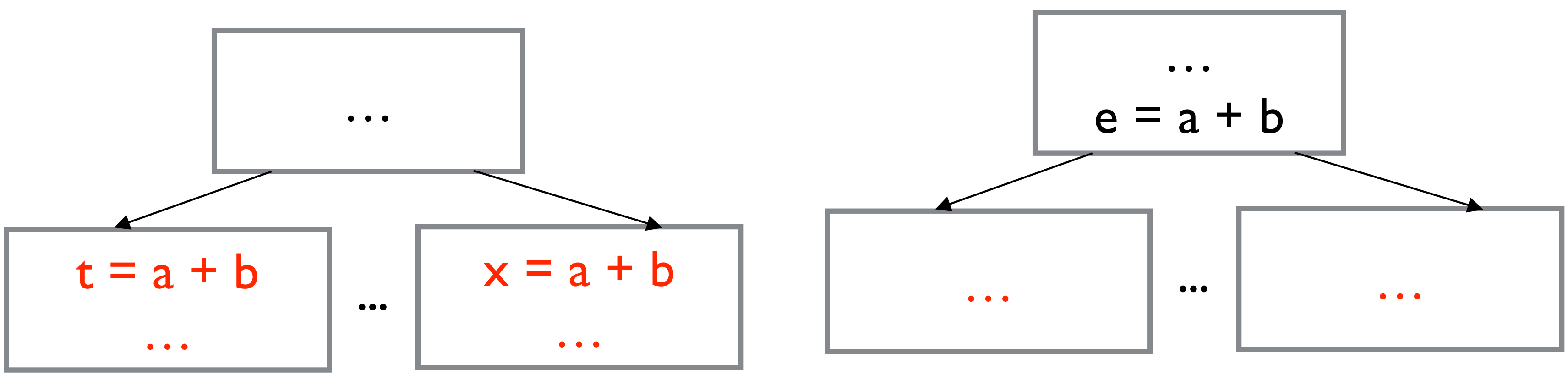
    LIVEOUT: union of all paths (any path problem)
    Also called May Problem

    May/Must

Other Data Flow Analysis

Very Busy Expressions

- Def: e is a very busy expression at the exit of block b if
  - e is evaluated and used along every path that leaves b, and
  - evaluating e at the end of b produces the same result
- useful for code hoisting
- saves code space



Very Busy Expressions

- VERYBUSY(b) contains expressions that are very busy at <u>end</u> of b
- UEEXPR(b): up exposed expressions (i.e. expressions defined in b and not subsequently killed in b)
- EXPRKILL(b): killed expressions

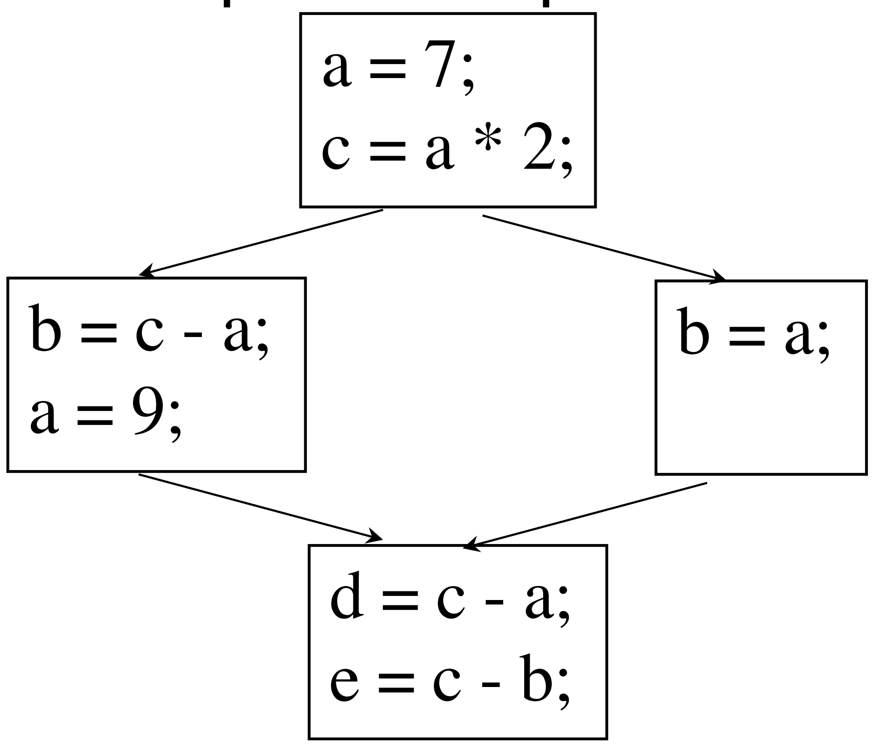
A backward flow problem, domain is the set of expressions

$$\text{VERYBUSY}(b) = \cap_{s \in succ(b)} \text{UEEXPR}(s) \cup (\text{VERYBUSY}(s) \cap \overline{\text{EXPRKILL}(s)})$$

$$\text{VERYBUSY}(n_f) = \emptyset$$

Constant Propagation

- Def of a constant variable v at point p:
  - Along every path to p, v has same known value
- Specialize computation at p based on v's value



Constant Propagation: Another Data Flow Problem

Domain is the set of pairs $<v_i,c_i>$ where v_i is a variable and $c_i \in C$

$$\text{CONSTANTS}(b) = \wedge_{p \in preds(b)} \; f_p(\text{CONSTANTS}(p))$$

- $\wedge$ performs a pairwise meet on two sets of pairs
- $f_p(x)$ is a block specific function that models the effects of block p on the $<v_i,c_i>$ pairs in x

A forward flow problem, domain is the set of pairs <v,c>.

C: constants or $\perp$.

$\perp$: non-constant or unknown value

$$\textit{Constants}(b) = \wedge_{p \in \text{preds}(b)} \ f_p(\textit{Constants}(\text{p}))$$

Meet operation $<v, c_1> \wedge <v, c_2>$

⊥: non-constant or unknown value

- $<v, c_1>$ if $c_1 = c_2$, else $<v, \perp>$

What about f_p ?

- if p has only one statement, update the constant set with
  - the results if operands are all constants
  - ⊥ if the result is unknown or non-constant
- If p has n statements then
  - fp(CONSTANTS(p)) = fn(fn-1(fn-2(…f2(f1(CONSTANTS(p)))…))),
    - where fi is the function generated by the ith statement in p

$\textit{Constants}(b) = \wedge_{p \in \text{preds}(b)} \ f_p(\textit{Constants}(p))$

Meet operation $<v, c_1> \wedge <v, c_2>$

- $<v, c_1>$ if $c_1 = c_2$, else $<v, \perp>$

$\perp$: non-constant or unknown value

Formal definition of p:

- If p has one statement then
  - $x \leftarrow y$ with $\text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots\}$
    then $f_p(\text{CONSTANTS}(p)) = \{\text{CONSTANTS}(p) - <x,l_1>\} \cup <x,l_2>$
  - $x \leftarrow y \text{ op } z$ with $\text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots,\ldots<z,l_3>\ldots\}$
    then $f_p(\text{CONSTANTS}(p)) = \{\text{CONSTANTS}(p) - <x,l_1>\} \cup <x, l_2 \text{ op } l_3>$
- If p has n statements then

  $f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(\text{CONSTANTS}(p)))\ldots)))$

  where f_i is the function generated by the i^{th} statement in p

f_p interprets p over CONSTANTS

Data-Flow Analysis Frameworks

- Generalizes and unifies data flow problems.
- Important components:
  - Direction D: forward or backward.
  - A *Semilattice:* a domain V and a *meet* operator ∧ that captures the effect of path confluence.
  - A transfer function F(m): compute the effect of passing through a basic block and include function value at boundary conditions.

A *semilattice* is an algebra $\mathcal{S} = (S, *)$ satisfying, for all $x, y, z \in S$,

(1) $x * x = x,$
(2) $x * y = y * x,$
(3) $x * (y * z) = (x * y) * z.$

Examples

- (D, V, F, ^)
- LIVE
  - D: backward
  - V: all variables
  - F_m: $UEVAR(m) \cup (LIVEOUT(m) \cap \overline{VARKILL(m)})$; $LIVEOUT(n_f) = \phi$
  - ^ : ∪
- AVAIL
  - D: forward, V: all expressions
  - F_m: $DEEXPR(m) \cup (AVAIL(m) \cap \overline{EXPRKILL(m)})$; $AVAIL(n_o) = \phi$
  - ^: ∩

Summary

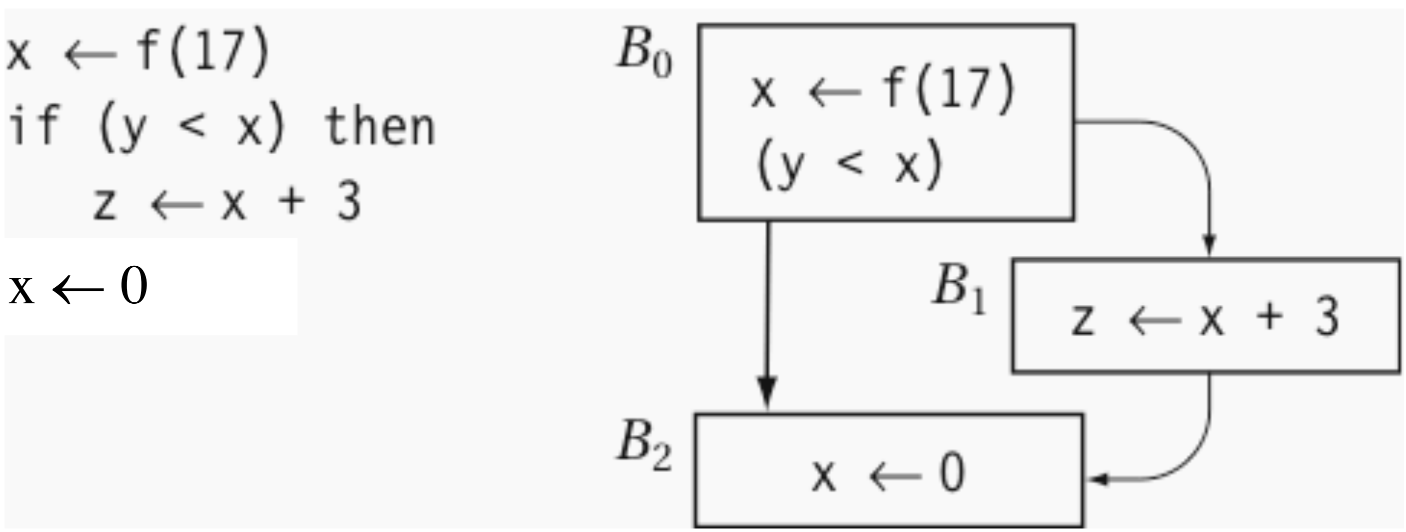
	Domain	Direction	Uses
AVAIL	Expressions	Forward	GCSE
LIVEOUT	Variables	Backward	Register alloc. Detect uninit. Construct SSA Useless-store Elim.
VERYBUSY	Expressions	Backward	Hoisting
CONSTANT	Pairs <v,c>	Forward	Constant folding

Why to Study Data Flow Analysis

- Data-flow analysis
  - A collection of techniques for compile-time reasoning about the run-time flow of values.
- Backbone of scalar optimizing compilers

Limitation of Data-Flow Analysis

- Imprecision from pointers, and procedure calls
- Assume all paths will be taken



If y is always no less than x, x is not live before B2.
But data-flow analysis may not figure that out.