15-745 Lecture 2b

Dataflow Analysis
Basic Blocks
Related Optimizations
Clarifications and more

An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Can we combine the gen and kill sets into one set per BB?

```
1: i <- 1          1     8,4
2: j <- 2          2
3: k <- 3 + i      3     11
4: i <- j          4     1,8
5: m <- i + k      5
```
(Gen / kill)

- Gen[BB]={2,3,4,5}
- Kill[BB]={1,8,11}
- Gen[s1;s2]=
- Kill[s1;s2]=

An Improvement: Basic Blocks

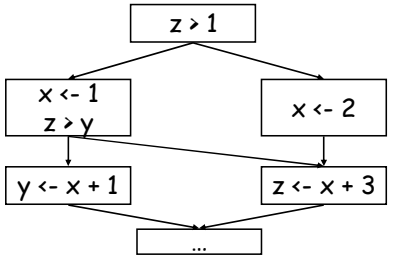
- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Can we combine the gen and kill sets into one set per BB?

```
1: i <- 1          1     8,4
2: j <- 2          2
3: k <- 3 + i      3     11
4: i <- j          4     1,8
5: m <- i + k      5
```
(Gen / kill)

- Gen[BB]={2,3,4,5}
- Kill[BB]={1,8,11}
- Gen[s1;s2]=(Gen[s1]-Kill[s2]) +Gen[s2]
- Kill[s1;s2]=(Kill[s1]-Gen[s2]) + Kill[s2]

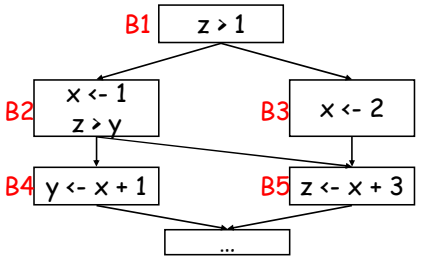
Def-use chains are valuable too

- Def-use chain: for each definition of var x, a list of all uses of that definition
- Computed from ~~liveness analysis~~ *upward-exposed uses*, a backward dataflow problem
- Def-use is NOT symmetric to use-def?



Def-use chains are valuable too

- Def-use is IS symmetric to use-def.
- Iff d is in u's UD chain, then u is in d's DU chain.
- Can compute DU chains from UD chains – same basic info, a set of <u,d> pairs, just packaged differently.



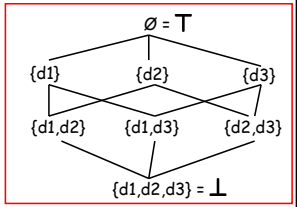
Pairs for x:
<d.B2,u.B4>, <d.B2,u.B5>,
<d.B3,u.B5>

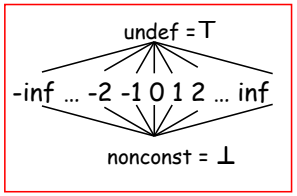
Use-def lists:
u.B4: d.B2
u.B5: d.B2, d.B3

Def-use lists:
d.B2: u.B4, u.B5
d.B3, u.B5

What the … is a Lattice?

- Represents values: for one item, or vector of all (often boolean to powerset)
- Has a defined top and bot
- According to ASU:
  - Top is least info: top $\wedge$ X = X
  - Bot is end: bot $\wedge$ X = bot
  - Init in[b] with top, out[b] with F_b(top).





Dragon versus Muchnick

- From now on, follow Dragon:
- Meet function = confluence for any problem
- Always start at top of lattice, meaning least information
- Meet (x $\wedge$ y) is the closest point where x and y run into each other going down the lattice; might be x, y, or other

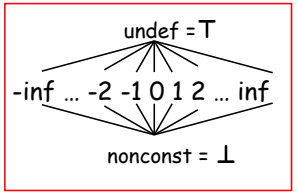
  (Muchnick seems to allow either direction on lattice, so either starting point, and defines join in addition to meet…)

Better Constant Propagation

- What about:

```
x <- 1
if (y > z)
    x <- 1
a <- x
```

- Use a better lattice



- Meet:
  - a $\wedge$ top -> a
  - a $\wedge$ bot -> bot
  - c $\wedge$ c -> c
  - c $\wedge$ d (if c $\neq$ d) -> bot
- Init all vars to: ~~bot or~~ top?

Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

```
a <- b+c
b <- a-d
c <- b+c
d <- a-d
```


Computing Available Expressions

- Forward or backward?
- Values?
- Lattice?
- gen[b] =
- kill[b] =
- in[b] =
- out[b] =
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, then all exprs using x are killed.
- out[b] = in[b] – kill[b] ∪ gen[b]
- in[b] = what to do at a join point?
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, exprs(x) are killed
  out[b] = (in[b] – kill[b]) ∪ gen[b]
- in[b] = An expr is avail only if avail on ALL edges, so: in[b] = ∩ over all p∈ pred(b), out[p]
- Initialization
  - All nodes except entry are set to ALL avail
  - Entry is set to NONE avail

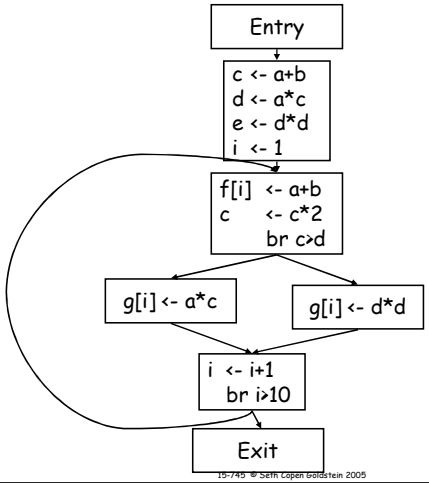
Constructing Gen & Kill

Stmt s	gen[s]	kill[s]
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	
M[a] <- b		
f(a, …)		{M[x] for all x}
t <- f(a,…)		

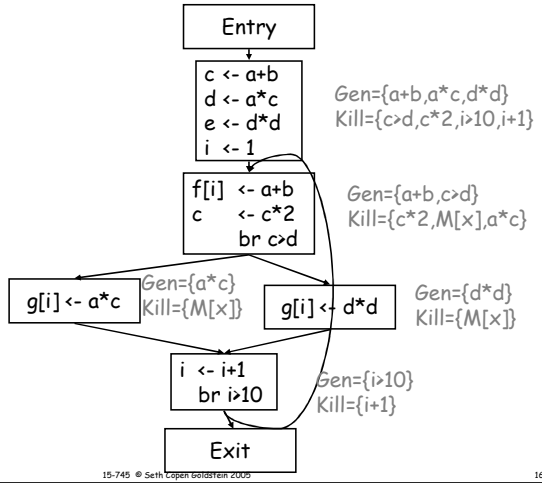
Constructing Gen & Kill

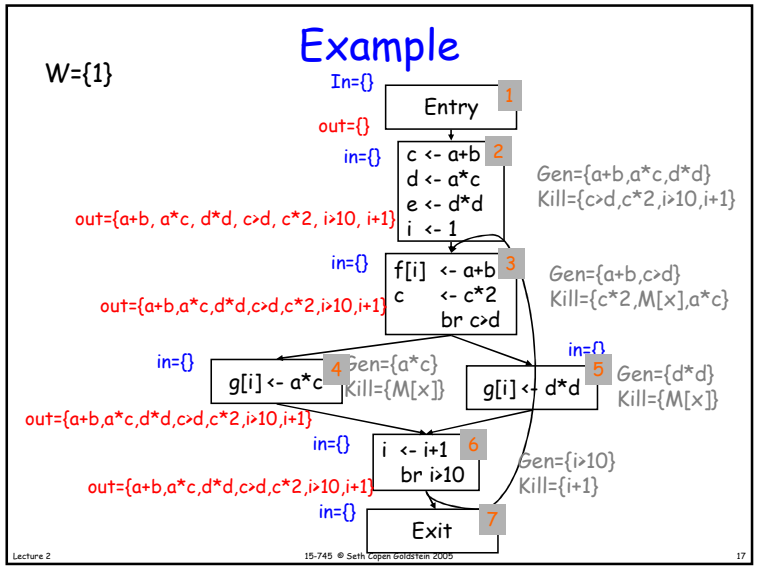
Stmt s	gen[s]	kill[s]
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	{exprs containing t}
M[a] <- b	{}	{for all x, M[x]}
f(a, …)	{}	{for all x, M[x]}
t <- f(a,…)	{}	{exprs containing t for all x, M[x]}

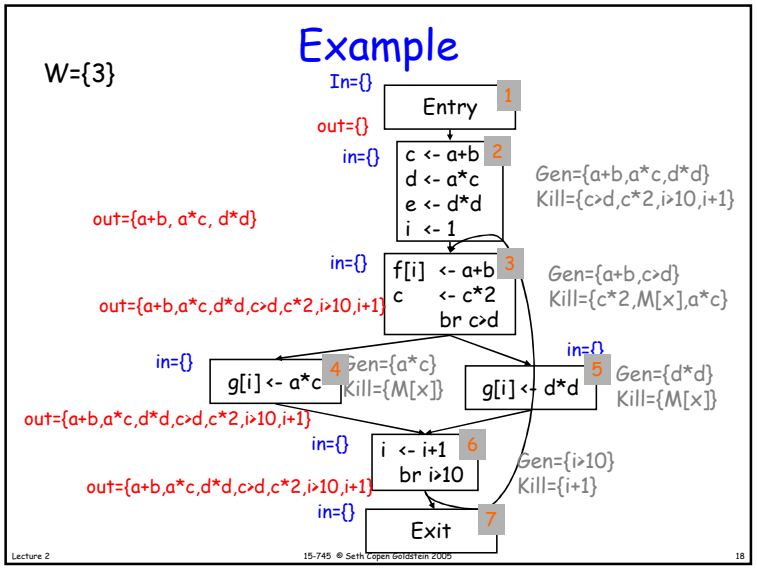
Example

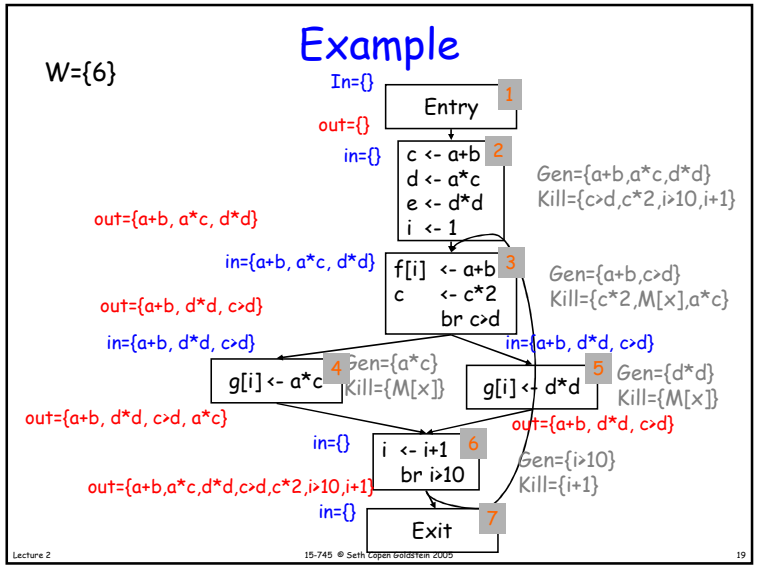


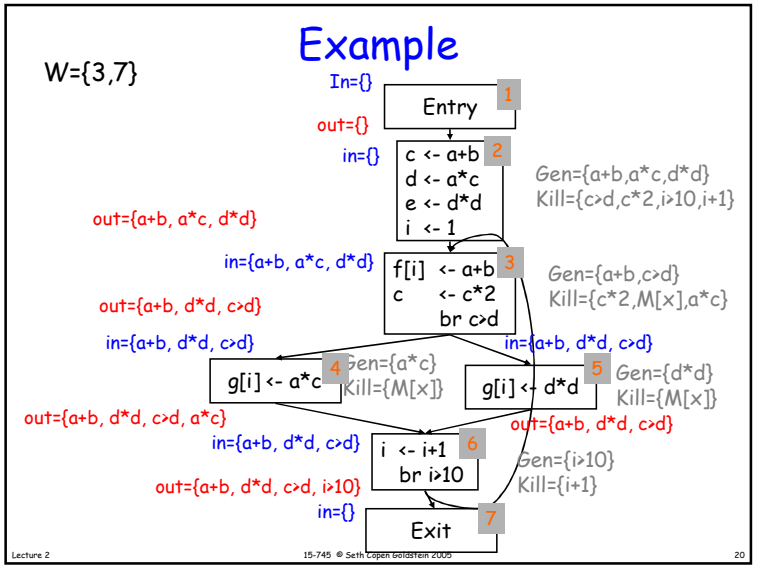
Example




Example
W={1}
In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d, c>d, c*2, i>10, i+1}
in={}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7
Lecture 2
15-745 © Seth Copen Goldstein 2005
17

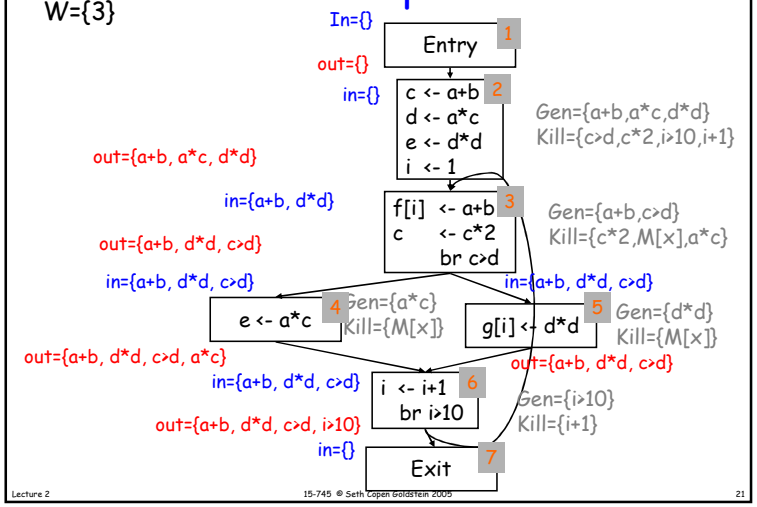

Example
W={3}
In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7
Lecture 2
15-745 © Seth Copen Goldstein 2005
18


Example
W={6}
In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b, a*c, d*d}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
in={a+b, d*d, c>d}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b, d*d, c>d, a*c}
out={a+b, d*d, c>d}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7
Lecture 2
15-745 © Seth Copen Goldstein 2005
19


Example
W={3,7}
In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b, a*c, d*d}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
in={a+b, d*d, c>d}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b, d*d, c>d, a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b, d*d, c>d, i>10}
in={}
Exit
7
Lecture 2
15-745 © Seth Copen Goldstein 2005
20

Example

W={3}



CSE

- Calculate Available expressions
- For every stmt in program

  If expression, x op y, is available {

  Compute reaching expressions for x op y at this stmt

  foreach stmt in RE of the form t <- x op y

  rewrite at: t' <- x op y

  t <- t'

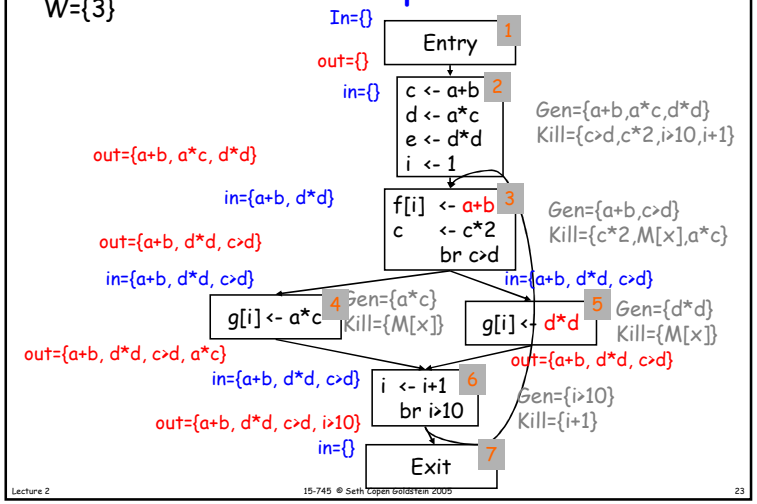
  }

  replace x op y in stmt with t'

  }

Example

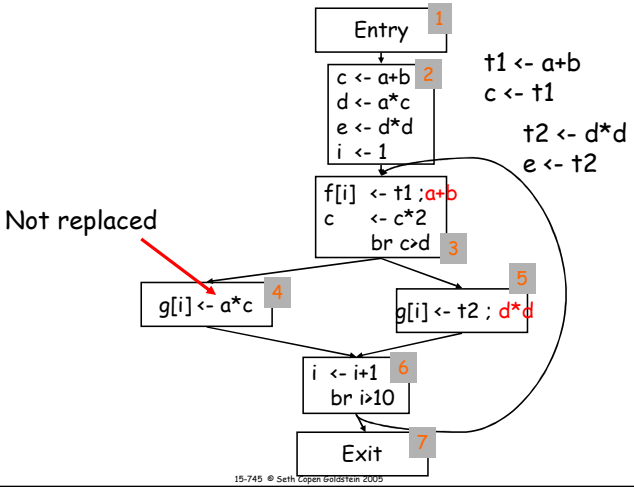
W={3}



Calculating RE

- Could be dataflow problem, but not needed enough, so …
- To find RE for x op y at stmt S
  - traverse cfg backward from S until
    - reach t <- x + y (& put into RE)
    - reach definition of x or y

Example



Dataflow Summary

	Union	intersection
Forward	Reaching defs	Available exprs
Backward	Live variables	

Later in course we look at bidirectional dataflow

Dataflow Framework

- Lattice
- Universe of values
- Meet operator
- Basic attributes (e.g., gen, kill)
- Traversal order
- Transfer function

Dataflow Framework

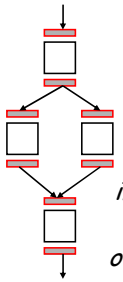
- Another formulation: "Meet over Paths" (MOP)
- To find in[B],
  - Enumerate <u>all</u> paths from entry to B to get P
  - Foreach path p_x in P, p_x={entry->b_i->b_j->….$b_{..}$}, calculate sum of transfer functions:

    $s_x = F_{b..}(\ldots\ldots F_{bj}(F_{bi}(\text{out[entry]})))..)$

  - Then do one big Meet over all the s_x values
  - Not practical; more of theoretical interest….

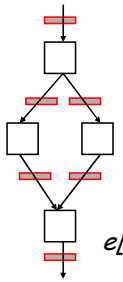
Dataflow Framework

- Finally…why not put the values on the edges?
  - (Muchnick thinks it's a good idea…)



$in[n] = \bigwedge_{p \text{ in } pred[n]} out[p]$

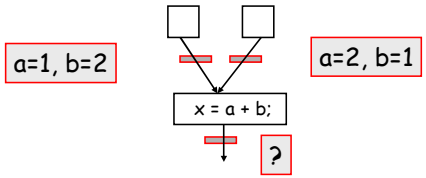
$out[n] = F_n(in[n])$



$e[n->s] = \bigwedge_{p \text{ in } pred[n]} F_{n,s}(e[p->n])$

Dataflow : Edges

- Muchnick's example: smart const prop



Dataflow : Edges

- My example (again, const prop)