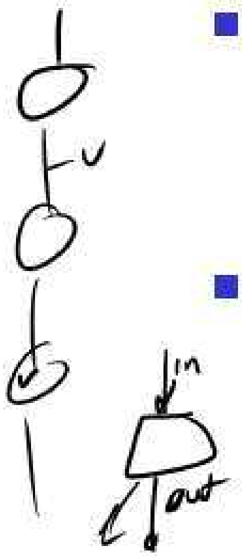
CSE P 501 – Compilers

Analysis & Optimization Examples

Hal Perkins

Winter 2008

Liveness Analysis – an example from last week

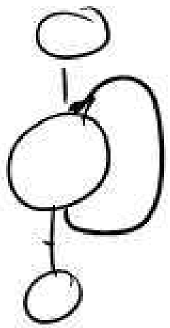


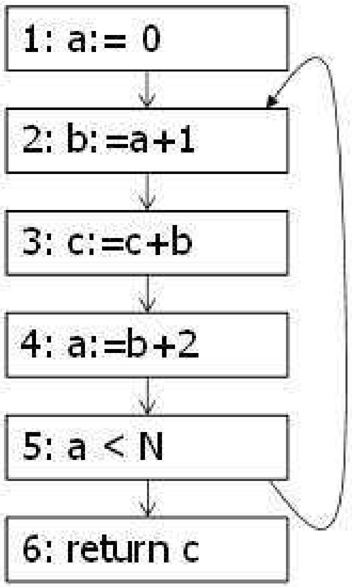
- Recall: A variable is *live* on an edge if there is a path from that edge to a use that does not go through any definition
- In a block, a variable is
  - *Live-in* if it is live on any in-edge
  - *Live-out* if it is live on any out-edge

Example (1 stmt per block)

- Code

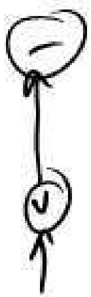
```
    a := 0
L:  b := a+1
    c := c+b
    a := b*2
    if a < N goto L
    return c
```





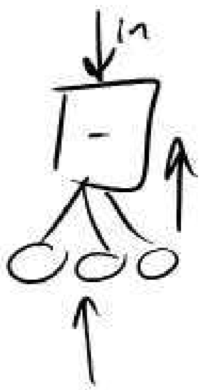
Liveness Analysis Sets

- For each block b
  - use[b] = variable used in b before any def
  - def[b] = variable defined in b & not killed
  - in[b] = variables live on entry to b
  - out[b] = variables live on exit from b



  - Information flows from the "future" to the "past"

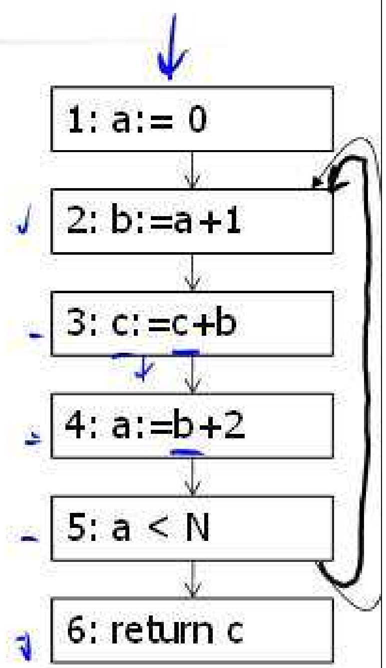
Dataflow equation



- Given the preceding definitions, we have
  - $in[b] = use[b] \cup (out[b] - def[b])$
  - $out[b] = \cup_{s \in succ[b]} in[s]$
- Algorithm
  - Set $in[b] = out[b] = \emptyset$
  - Update in, out until no change
- Evaluation order: back to front is best given information flow

Calculation

block (stmt)	use	def	1st out	1st in	2nd out	2nd in	3rd
6	c	–	—	c	—	c	same
5	a	—	c	ac	ac	ac	
4	b	a	ac	bc	ac	bc	
3	cb	c	bc	bc	bc	bc	
2	a	b	bc	ac	bc	ac	
1	—	a	ac	c	ac	c	



A few optimizing transformations

- A few examples with a bit more detail than last time….

Classic Common-Subexpression Elimination

- In a statement s: t := x op y, if x op y is *available* at s then it need not be recomputed
- Analysis: compute *reaching expressions* i.e., statements n: v := x op y such that the path from n to s does not compute x op y or define x or y

n x op y

s x op y

Classic CSE

n: v = x op y   w

s t = x op y   w

- If x op y is defined at n and reaches s
  - Create new temporary w
  - Rewrite n as

    n: w := x op y
    n': v := w
  - Modify statement s to be

    s: t := w
  - (Rely on copy propagation to remove extra assignments if not really needed)

Constant Propagation

d: t := c

- Suppose we have
  - t: Statement d: t := c, where c is constant
  - Statement n that uses t
- If d reaches n and no other definitions of t reach n, then rewrite n to use c instead of t

n: t

c

d: t := z

n: ~t~
z

Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t

Copy Propagation Tradeoffs

z

- Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic
  - Not worth doing if only reason is to eliminate copies – let the register allocate deal with that
- But it can expose other optimizations, e.g.,

z

```
a := y + z
u := y
c := u + z
```

y

  - After copy propagation we can recognize the common subexpression

Dead Code Elimination

- If we have an instruction

  s: a := b op c

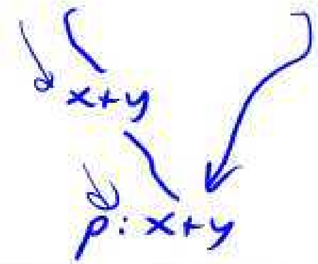
  and a is not live-out after s, then s can be eliminated
  - Provided it has no implicit side effects that are visible (output, exceptions, etc.)

a := b+c
x/y
c
Ada

Lazy Code Motion (LCM)

- Also known as partial-redundancy elimination
- More recent alternative to classic CSE and loop-invariant code motion

Partial Redundancy



- Informally, an expression is *partially redundant* if it is done more than once on some path through the flowgraph
- More specifically, a computation is partially redundant at point p if it occurs on some, but not all paths that reach p
- Idea: convert partially redundant expressions to fully redundant, then eliminate it, which moves it out of a loop or avoids recomputing it on some paths

Example