CSE 401 – Compilers

Dataflow Analysis

Hal Perkins

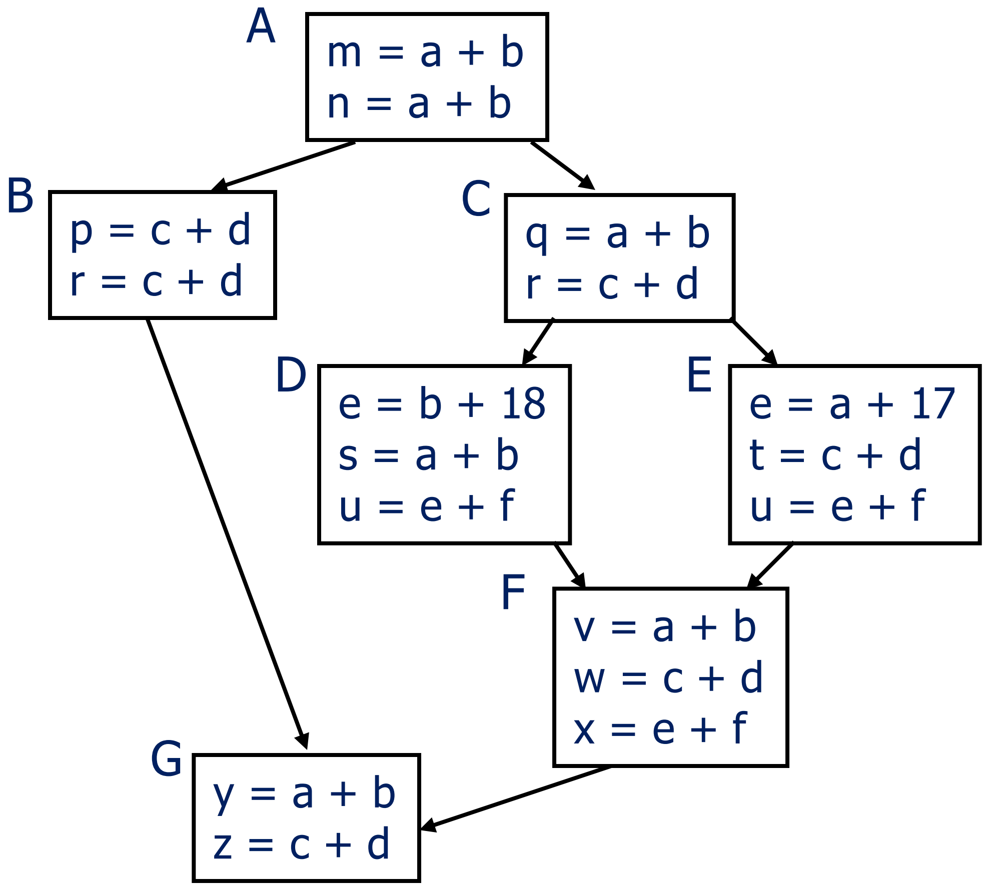
Winter 2011

Agenda

- Initial example: dataflow analysis for common subexpression elimination
- Other analysis problems that work in the same framework

Available Expressions

- Goal: use dataflow analysis to find common subexpressions
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation
  - Simple inside a single block; more complex dataflow analysis used across bocks



"Available" and Other Terms

- An expression *e* is *defined* at point *p* in the CFG if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands is defined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*

Available Expression Sets

- For each block *b*, define
  - AVAIL(b) – the set of expressions available on entry to *b*
  - NKILL(b) – the set of expressions <u>not killed</u> in *b*
  - DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions

- AVAIL(b) is the set

  $$\text{AVAIL}(b) = \cap_{x \in preds(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$

  - preds(b) is the set of b's predecessors in the control flow graph
- This gives a system of simultaneous equations – a dataflow problem

Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(b) and NKILL(b)
    - This only needs to be done once
  - Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing DEF and NKILL (1)

- For each block *b* with operations $o_1, o_2, \ldots, o_k$

```
KILLED = ∅
DEF(b) = ∅
for i = k to 1
   assume o_i is "x = y + z"
   if (y ∉ KILLED and z ∉ KILLED)
      add "y + z" to DEF(b)
   add x to KILLED
…
```

Computing DEF and NKILL (2)

- After computing DEF and KILLED for a block b,

  NKILL(b) = { all expressions }
  for each expression *e*
      for each variable $v \in$ e
          if $v \in$ KILLED then
              NKILL(b) = NKILL(b) - *e*

Computing Available Expressions

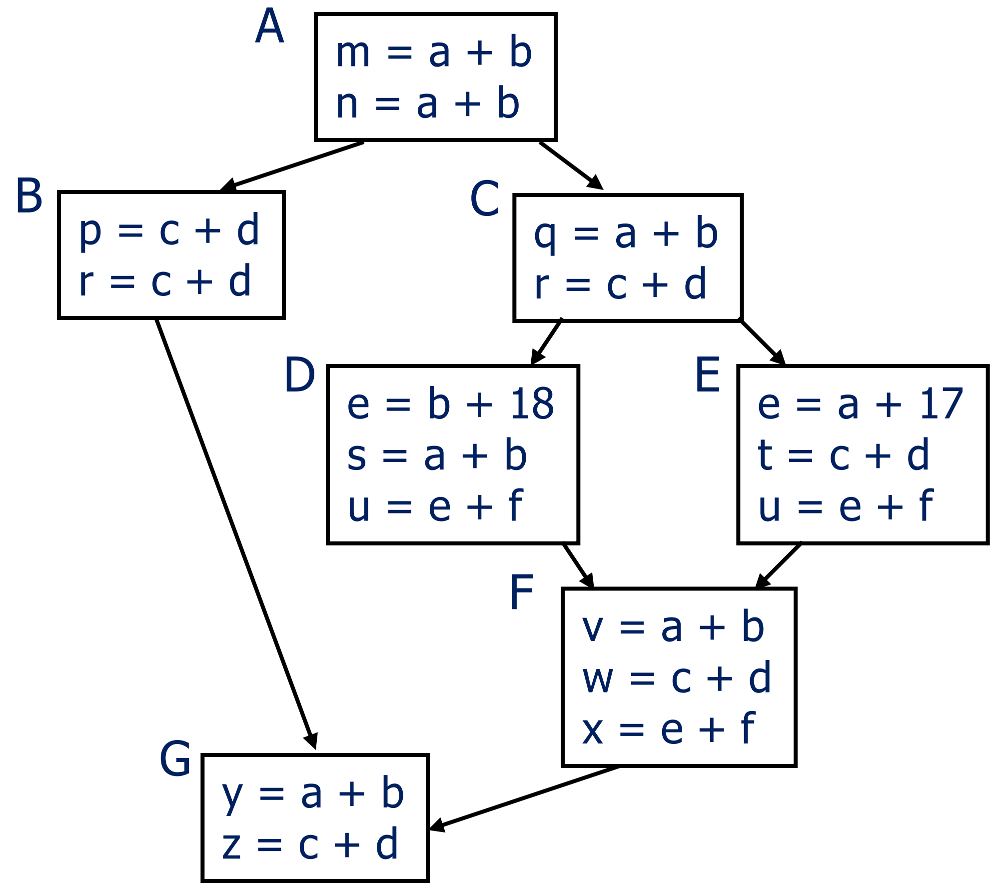
- Once DEF(b) and NKILL(b) are computed for all blocks b

```
Worklist = { all blocks b_i }
while (Worklist ≠ ∅)
    remove a block b from Worklist
    recompute AVAIL(b)
    if AVAIL(b) changed
        Worklist = Worklist ∪ successors(b)
```

Available Expressions

- AVAIL(b) – the set of expressions available on entry to *b*
- NKILL(b) – the set of exprs. <u>not killed</u> in *b*
- DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*
- AVAIL(b) =

  $\bigcap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$



Dataflow analysis

- Available expressions are an example of a *dataflow analysis* problem
- Many similar problems can be expressed in a similar framework
- Only the first part of the story – once we've discovered facts, we then need to use them to improve code

Characterizing Dataflow Analysis

- All of these algorithms involve sets of facts about each basic block b
  - IN(b) – facts true on entry to b
  - OUT(b) – facts true on exit from b
  - GEN(b) – facts created and not killed in b
  - KILL(b) – facts killed in b
- These are related by the equation

  $$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

  - Solve this iteratively for all blocks
  - Sometimes information propagates forward; sometimes backward

Example:Live Variable Analysis

- A variable *v* is *live* at point *p* iff there is *any* path from *p* to a use of *v* along which *v* is not redefined
- Some uses:
  - Register allocation – only live variables need a register (or temporary)
  - Eliminating useless stores
  - Detecting uses of uninitialized variables
  - Improve SSA construction – only need Φ-function for variables that are live in a block (later)

Liveness Analysis Sets

- For each block b, define
  - 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

Equations for Live Variables

- Given the preceding definitions, we have

  $\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$

  $\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$

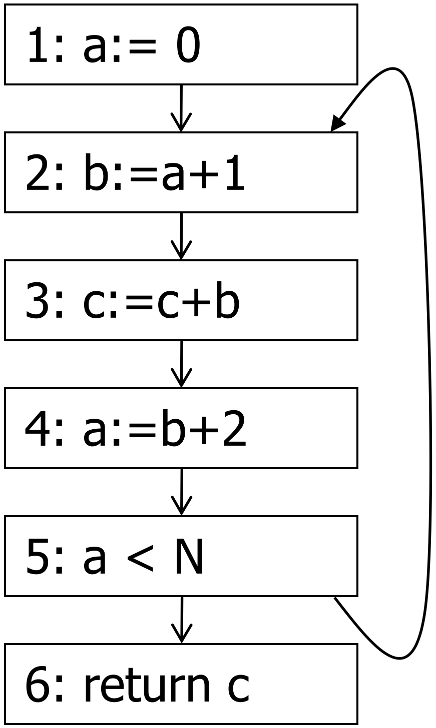
- Algorithm
  - Set in[b] = out[b] = $\varnothing$
  - Update in, out until no change

$in[b] = use[b] \cup (out[b] - def[b])$
$out[b] = \cup_{s \in succ[b]}\ in[s]$

Example (1 stmt per block)

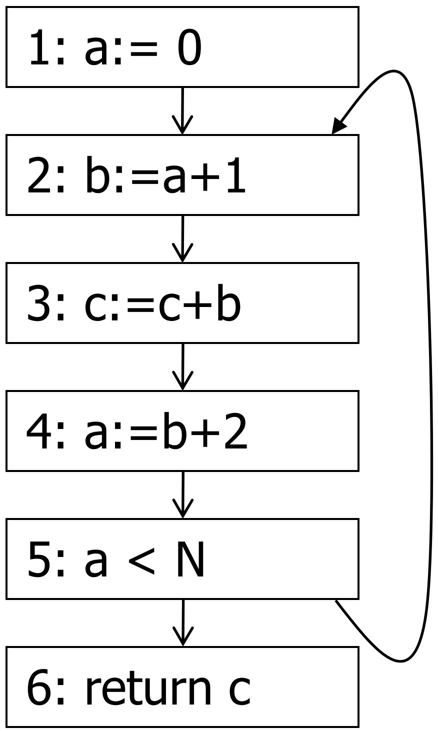
- Code

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



Calculation

$in[b] = use[b] \cup (out[b] - def[b])$
$out[b] = \cup_{s \in succ[b]} in[s]$



Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables…
- Sets
  - USED(b) – variables used in b before being defined in b
  - NOTDEF(b) – variables not defined in b
  - LIVE(b) – variables live on *exit* from b
- Equation

$$LIVE(b) = \cup_{s \in succ(b)} USED(s) \cup (LIVE(s) \cap NOTDEF(s))$$

Efficiency of Dataflow Analysis

- The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG
  - Forward problems – reverse postorder
  - Backward problems - postorder

Example: Reaching Definitions

- A definition *d* of some variable *v* *reaches* operation *i* iff *i* reads the value of *v* and there is a path from *d* to *i* that does not define *v*
- Uses
  - Find all of the possible definition points for a variable in an expression

Equations for Reaching Definitions

- Sets
  - DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  - SURVIVED(b) – set of all definitions not obscured by a definition in b
  - REACHES(b) – set of definitions that reach b
- Equation

$$\text{REACHES}(b) = \cup_{p \in preds(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))$$

Example: Very Busy Expressions

- An expression *e* is considered *very busy* at some point *p* if *e* is evaluated and used along every path that leaves *p*, and evaluating *e* at *p* would produce the same result as evaluating it at the original locations
- Uses
  - Code hoisting – move *e* to *p* (reduces code size; no effect on execution time)

Equations for Very Busy Expressions

- Sets
  - USED(b) – expressions used in b before they are killed
  - KILLED(b) – expressions redefined in b before they are used
  - VERYBUSY(b) – expressions very busy on exit from b
- Equation

$$\text{VERYBUSY}(b) = \bigcap_{s \in succ(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s))$$

Using Dataflow Information

- A few examples of possible transformations…

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

Classic CSE

- 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

- Suppose we have
  - 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

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

- 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.,

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

  - 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.)

Dataflow…

- General framework for discovering facts about programs
  - Although not the only possible story
- And then: facts open opportunities for code improvement
- To be continued…
  - SSA in sections Thursday
  - CSE 501!