

Data flow analysis

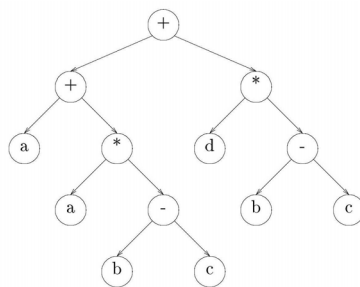
- compile-time - reasoning about the run-time
 - > flow of values in the program
 - > represent facts about run-time behavior
 - > represent effect of executing each basic block
 - > propagate facts around control flow graph
- Formulated as a set of simultaneous equations
 - > sets attached to the nodes and edges
 - > lattice to describe relation between values
 - > usually represented as bit or bit vector
- Limitations
 - > answers must be conservative
 - > often need to approximate information
 - > assume all possible paths can be taken

CMSC430 Spring 2007

1

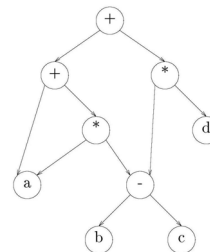
Common subexpressions

- Both a and $b-c$ are common subexpressions (CSE)
 - > Compute same value
 - > Should compute the value once
 - > A simple and general form of code improvement



$a + a * (b - c) + (b - c) * d$

The directed acyclic graph is a useful Representation for such expressions.



The *dag* clearly exposes the cses

CMSC430 Spring 2007

2

Directed acyclic graph

- A *directed acyclic graph* is a tree with sharing
 - > a tree is a directed acyclic graph where each node has at most one parent
 - > a *dag* allows multiple parents for each node
 - > both a tree and a *dag* have a distinguished *root*
 - > no cycles in the graph!
- To find common subexpressions within a statement
 - > build the dag
 - > generate code from the dag
 - > This should lead to faster evaluation
- How do we build a dag for an expression?
 - > Use construction primitives for building trees
 - > Teach primitives to catch cses
 - Mkleaf() and mknnode()
 - Hash on <op, l, r>
 - > Unique name for each node – its value number
- Anywhere we build a tree, we could build a dag
 - > Initialize hash table on each expression
 - > Catch only cses within expressions

CMSC430 Spring 2007

3

Directed acyclic graph

- What about assignment?
 - > Complicates cse detection
 - > Each value has a unique node
 - > Add subscripts to variables
- While building the dag, an assignment
 - > Creates new nodes for lhs – a new x_i
 - > Kills all nodes built from x_{i-1}
- Example: $a_1 = a_0 + b$
 - > Can we go beyond a single statement?
 - > Use a single dag for an entire basic block
- A dag for a basic block has labeled nodes
 - > Leaves are labeled with unique identifiers (either variable names or constants) (Leaves represent values on entry)
 - > Interior nodes are labeled with operators
 - > Nodes have optional identifier labels
 - Interior nodes represent computed values
 - Identifier label represents assignment

CMSC430 Spring 2007

4

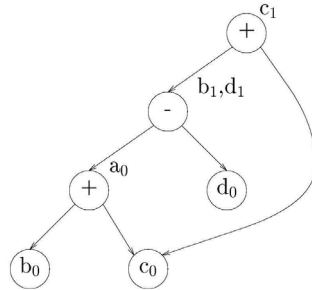
Example

- Code

$a = b + c$
 $b = a - d$
 $c = b + c$
 $d = a - d$

- After renaming

$a_0 = b_0 + c_0$
 $b_1 = a_0 - d_0$
 $c_1 = b_1 + c_0$
 $d_1 = a_0 - d_0$



CMSC430 Spring 2007

5

Building a dag

$\text{node}(\langle \text{id} \rangle) \rightarrow$ current dag for $\langle \text{id} \rangle$

1. Set $\text{node}(y)$ to undefined for each symbol y
2. for each statement $x = y \text{ op } z$, repeat steps 3, 4, and 5
3. If $\text{node}(y)$ is undefined,
 - > create a leaf for y
 - > set $\text{node}(y)$ to the new node
 - > do the same for z
4. if $\langle \text{op}, \text{node}(y), \text{node}(z) \rangle$ doesn't exist, create it and let n point to that node
5. delete x from the list of labels for $\text{node}(x)$
 - > append x to the list of labels for n
 - > set $\text{node}(x)$ to n

CMSC430 Spring 2007

6

Common subexpressions

- Going beyond basic blocks
 - > Can no longer build dags
 - > Must consider control flow
- Examples
 - > Use
 - C = A + B
 - D = A + B
 - > Intervening kill
 - C = A + B
 - A = ...
 - D = A + B
 - > Possible use
 - C = A + B
 - If (...)
 - D = A + B
- Possible kill
 - C = A + B
 - If (...)
 - A = ...
 - D = A + B
- Possible gen
 - If (...)
 - C = A + B
 - D = A + B
- Multiple gen
 - If (...)
 - C = A + B
 - Else
 - C = A + B
 - D = A + B

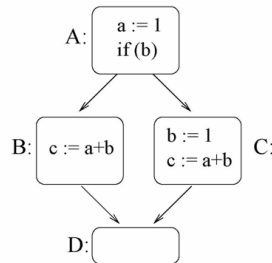
We generalize these conditions as data flow analysis

Algorithm

- build control flow graph (CFG)
 - > initial (local) data gathering
 - > propagate information around the graph
 - > post-processing (if needed)
- Example control flow graph

```

a := 1
if (b) then
  c := a+b
else
  b := 1
  c := a+b
...
  
```



Available expressions

- An expression is *defined* at point p if its value is computed at p.
- An expression is *killed* at a point p if one of its argument variables is defined at p.
- An expression e is *available* at a point p in a procedure if every path leading to p contains a prior definition of e that is not killed between its definition and p.

Global common subexpression elimination

- If, at some definition point for $p = e$, e is available with name x, we can replace the evaluation with a reference to x.
- Requires a global naming scheme and a natural analog to parts of value numbering

CMSC430 Spring 2007

9

Available expressions

For a block b

let AVAIL(b) be the set of expressions available on entry to b.

let KILL(b) be the set of expressions killed in b.

let GEN(b) be the set of expressions defined in b and not subsequently killed in b.

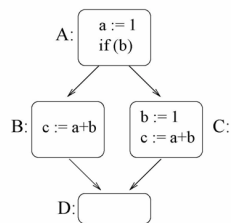
Note: GEN(b) can be calculated from the list of live expressions from DAG construction.

- KILL(b) is harder to construct, since it requires knowledge of all potential expressions in the program.
- Now, AVAIL can be defined as:
$$AVAIL(b) = \bigcap_{x \in \text{pred}(b)} (\text{GEN}(x) \cup (\text{AVAIL}(x) - \text{KILL}(x)))$$
- Note: initializations must be conservative.

CMSC430 Spring 2007

10

Available expression example



Node	Kill	Gen
A	a+b	∅
B	∅	a+b
C	a+b	a+b
D	∅	∅

- $AVAIL(A) = \emptyset$
- $AVAIL(B) = GEN(A) \cup (AVAIL(A) - KILL(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset$
- $AVAIL(C) = GEN(A) \cup (AVAIL(A) - KILL(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset$
- $AVAIL(D) = (GEN(B) \cup (AVAIL(B) - KILL(B))) \cap (GEN(C) \cup (AVAIL(C) - KILL(C)))$
 $= (\{a+b\} \cup (\emptyset - \emptyset)) \cap (\{a+b\} \cup (\emptyset - \{a+b\})) = \{a+b\}$

CMSC430 Spring 2007

11

Partial redundancy elimination

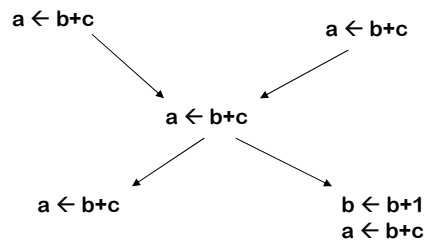
- **Partial redundancy elimination (PRE)** is an optimization that
 - > Discovers partially redundant expressions
 - > Converts them to fully redundant expressions
 - > Removes redundant expressions
- **Intuition**
 - > PRE moves computation back (against the control flow) as far as possible to make their effects universal as possible,
- **How does it work?**
 - > Anticipability → expression can be precomputed at point p
 - > Use data-flow analysis to find availability and anticipability
 - > Solve a data-flow problem to discover where to insert code
 - > Insert the code and remove redundant expression

CMSC430 Spring 2007

12

Redundant expressions

- An expression ϵ is redundant at point p if every path to p :
 - > ϵ is evaluated before reaching p , and
 - > None of the constituent values of ϵ are redefined before p .

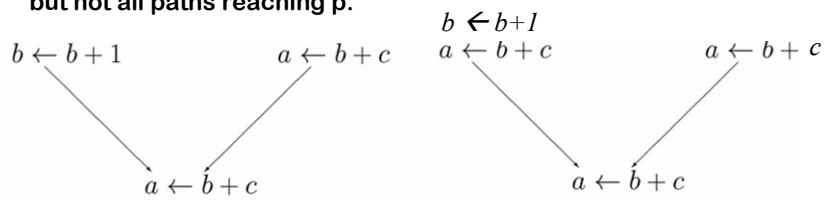


CMSC430 Spring 2007

13

Partially redundant expressions

- An expression is partially redundant at p if it is available on some, but not all paths reaching p .



CMSC430 Spring 2007

14

PRE equations

$$avin(b) = \begin{cases} \text{false} & \text{if } b \text{ is an entry block} \\ \bigcap_{x \in \text{pred}(b)} avout(x) & \text{otherwise} \end{cases}$$

$$avout(b) = comp(b) \cup (avin(b) \cap transp(b))$$

$$pavin(b) = \begin{cases} \text{false} & \text{if } b \text{ is an entry block} \\ \bigcup_{x \in \text{pred}(b)} pavout(x) & \text{otherwise} \end{cases}$$

$$pavout(b) = comp(b) \cup (pavin(b) \cap transp(b))$$

$$antout(b) = \begin{cases} \text{false} & \text{if } b \text{ is an exit block} \\ \bigcap_{x \in \text{succ}(b)} antin(x) & \text{otherwise} \end{cases}$$

$$antin(b) = antloc(b) \cup (antout(b) \cap transp(b))$$

$$ppout(b) = \begin{cases} \text{false} & \text{if } b \text{ is an exit block} \\ \bigcap_{x \in \text{succ}(b)} ppin(x) & \text{otherwise} \end{cases}$$

$$ppin(b) = \begin{aligned} & \text{antin}(b) \cap \text{pavin}(b) \cap \\ & (\text{antloc}(b) \cup (\text{ppout}(b) \cap \text{transp}(b))) \end{aligned}$$

$$insert(b) = \frac{\overline{(\text{ppin}(b) \cap \text{transp}(b))}}{\cap \text{ppout}(b) \cap \text{avout}(b)}$$

$$delete(b) = \text{ppin}(b) \cap \text{antloc}(b)$$

CMSC430 Spring 2007

15

Solving data flow equations

- Iterative algorithm

```
change = true;
while (change)
  change = false;
  for each basic block b // faster in reverse PostOrder
    solve data-flow equations for b
    if old != new then change = true;
  end for
end while
```
- Speed of solution
 - > Node may change only if some predecessor changes
 - > Try to visit node after all its predecessors
 - > Reverse PostOrder propagates information quickly
 - > Programs usually converge after 3-4 passes
 - > Use bit vectors for more efficiency

CMSC430 Spring 2007

16

PostOrder and reverse PostOrder

- Step 1: PostOrder
Main()
 count = 1;
 visit (root);
visit(n)
 mark n as visited
 for each successor s of n not yet visited, visit(s);
 PostOrder(n) = count;
 count = count+1;
- Step 2: Reverse Postorder(rPostOrder)
For each node n
 rPostOrder(n) = NumNodes – PostOrder(n)

Depth-first search ~ rPostOrder

Data-flow analysis framework

- Use same framework for all data-flow problems
 - > Given local information Gen, Kill
 - > Start with some initial values for In, Out
 - > Iterate through nodes in the flow graph, recompute transfer functions until sets stabilize
- Framework has 3 components
 - > Domain of values: L
 - > Operator for combining values: Δ
 - > A set of transfer functions (L \rightarrow L): F
- Usefulness of unified framework
 - > Defines a collection of properties that guarantee correctness and convergence
 - > Can describe speed of convergence and precision of result for a family of analysis problems
 - > Can reuse code to solve new analysis problems

Iterative algorithm

- What about loops?
 - > Circular dependencies between blocks
 - > Can initialize solutions, then solve repeatedly
- Example
 - c = a+b
 - L:
 - d = a+b
 - a= ...
 - if (...) goto L
- Termination
 - > Goal is for solutions to converge to a fixed point ($x = f(x)$)
 - > Can stop once solution stops changing
 - > Is this guaranteed?
 - If system is monotone (i.e., $f(x \wedge y) \leq f(x) \wedge f(y)$)