DAG for Multiple Statements

DAGs can also handle assignment statements

- same variable has different values

Solution

- creates new node for lhs — a new \( x_i \)
- kills all nodes built from \( x_{i-1} \)

A DAG for a basic block has labeled nodes

1. leaves are labeled with unique identifier
   - either variable names or constants
   - \( l\)values or \( r\)values
   - leaves represent values on entry, \( x_0 \)
2. interior nodes are labeled with operators
3. nodes have optional identifier labels
   - interior nodes represent computed values
   - identifier label represents assignment

(obvious by context)
DAG for Multiple Statements

Example

<table>
<thead>
<tr>
<th>Code</th>
<th>After Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \leftarrow b + c$</td>
<td>$a_0 \leftarrow b_0 + c_0$</td>
</tr>
<tr>
<td>$b \leftarrow a - d$</td>
<td>$b_1 \leftarrow a_0 - d_0$</td>
</tr>
<tr>
<td>$c \leftarrow b + c$</td>
<td>$c_1 \leftarrow b_1 + c_0$</td>
</tr>
<tr>
<td>$d \leftarrow a - d$</td>
<td>$d_1 \leftarrow a_0 - d_0$</td>
</tr>
</tbody>
</table>
DAG Construction Algorithm

Building a DAG

\[ \text{node}(\ <id>\ ) \rightarrow \text{current DAG for } <id> \]

1. set node(\( y \)) to undefined, for each symbol \( y \)

2. for each statement \( x \leftarrow y \ \text{op} \ z \), repeat steps 3, 4, and 5

3. if node(\( y \)) is undefined,
   
   create a leaf for \( y \)
   
   set node(\( y \)) to the new node
   
   \textit{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 node(\( x \))
   
   append \( x \) to the list of labels for \( n \)
   
   set node(\( x \)) to \( n \)

Aho, Sethi, and Ullman, Algorithm 9.2, in §9.8
Value Numbering

A different approach: Value numbering

- associate *unique* value number with each distinct value created or used within the block. Two variables have the same value *only* if they are provably identical.
- a classical (*i.e.*, *old*) way to fold constants and eliminate common subexpressions
- a little more flexible than building a DAG
- a gentle way to lead into data-flow analysis
Value Numbering

Assumptions

- can find basic blocks
- input is in triples: A OP B, where OP may be :=, +, −, ...
- no knowledge about surrounding context  
  (local method)
- reference’s type is textually obvious  
  (tag lhs and rhs)

Input

- basic block of triples  
  (n instructions)
- symbol table

Output

- improved basic block and result value #  
  (CSE, constant folding)
- table of available expressions†
- table of constant values

† An expression is available at point p if it is defined along each path leading to p and none of its constituent values has been subsequently redefined.
Value Numbering

Key Notions

- each variable, each expression, and each constant is assigned a unique number, its value number
  - same number ⇒ same value
  - based solely on information from within the block
    ⇒ variables and constants in SYMBOLS
    ⇒ expressions in AVAIL and triple

- if an expressions value is available (already computed), we should not recompute it

- constants denoted in both SYMBOLS and triples

Initializations

1. find the block
2. clear constant bits, val fields
Value Numbering

Principle data structures

CODE

- instructions, value number (ValNum) of result
- **Fields:** instr, resultValNum, isConst

SYMBOLS

- hash table keyed by variable name
- current value number of variable
- **Fields:** name, ValNum, isConst

AVAIL

- available expressions based on value number
- hash table keyed by \((val, op, val)\)
- **Fields:** (lhsValNum, op, rhsValNum), resultValNum, instr

CONSTANTS

- value number of each constant
- **Fields:** ValNum, bits
Value Numbering Algorithm

for $i \leftarrow 1$ to $n$
    $r \leftarrow$ value number for $rhs[i]$
    $l \leftarrow$ value number for $lhs[i]$
    if $op[i]$ is a store /* $l := r$ */ then
        SYMBOLS[$lhs[i]$].valNum $\leftarrow r$
        if $r$ is constant then
            SYMBOLS[$lhs[i]$].isConstant $\leftarrow$ true
    else /* an expression $l$ $OP$ $r$ */
        if $l$ is constant then replace $lhs[i]$ with constant
        if $r$ is constant then replace $rhs[i]$ with constant
        if $l$ is “ref $k$” then replace $lhs[i]$ with $k$
        if $r$ is “ref $k$” then replace $rhs[i]$ with $k$
        if $l$ and $r$ are both constant then
            create $\text{CONSTANTS}(l, op[i], r)$
            $\text{CONSTANTS}(l, op[i], r)$.bits $\leftarrow \text{eval}(l \ op[i] \ r)$
            $\text{CONSTANTS}(l, op[i], r)$.valNum $\leftarrow$ new value number
            $op[i]$ $\leftarrow$ “constant $(l \ op[i] \ r)$”
        else
            if $(l, op[i], r) \in \text{AVAIL}$ then
                $op[i]$ $\leftarrow$ “ref $\text{AVAIL}(l, op[i], r)$.instr”
            else

CMSC 430  Lecture 26, Page 8
create AVAIL($l, op[i], r$)
AVAIL($l, op[i], r$).instr $\leftarrow$ i
AVAIL($l, op[i], r$).resultValNum $\leftarrow$ new value number

for $i \leftarrow 1$ to $n$
    if $op[i]$ is ref or constant then delete instruction $i$
Example

<table>
<thead>
<tr>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
</tr>
<tr>
<td>(t1)</td>
</tr>
<tr>
<td>(t2)</td>
</tr>
<tr>
<td>(t3)</td>
</tr>
<tr>
<td>(t4)</td>
</tr>
<tr>
<td>(t5)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYMBOLS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>ValNum</td>
</tr>
<tr>
<td>c5</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>c3</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c6</td>
<td>5</td>
</tr>
<tr>
<td>m</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTANTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ValNum</td>
<td>bits</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AVAIL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lhsValNum</td>
<td>op</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
</tr>
</tbody>
</table>

CMSC 430
Lecture 26, Page 10
Value Numbering

Safety

- constant folding — applied only to constant arguments
- common subexpressions — construction ensures it

Profitability

- assume that load of constant is cheaper than $\text{op}$
- assume that reference (or copy) is cheaper than $\text{op}$
- forwarding mechanism (ref) does subsumption

Opportunity

- look at each instruction
- linear time, but assumes basic blocks are small
Value Numbering

What does value numbering accomplish?

- assign a value number to each available expression
  - identity based on value, \emph{not} name
  - DAG construction has same property
- eliminate duplicate evaluations
- evaluate and fold constant expressions

\emph{Value numbering’s focus is on values, not names. This is an important distinction.}

- It is easy to attain in \emph{local} optimization.
- It is harder in \emph{global} and \emph{interprocedural} optimization.

\emph{Can we extend this idea across blocks?}