Basic Blocks

Definition

- sequence of code
- control enters at top, exits at bottom
- no branch/halt except at end

Construction algorithm (for 3-address code)

1. determine set of leaders
   (a) first statement
   (b) target of goto or conditional goto
   (c) statement following goto or conditional goto

2. add to basic block all statements following leader up to next leader or end of program

Example:

A := 0;
if (<cond>) goto L;
A := 1;
B := 1;
L: C := A;
Local vs. Global Optimization

Scope

- *local* — within basic block
- *global* — across basic blocks
- refers to both analyses and optimizations

Some optimizations may be applied locally or globally (e.g., dead code elimination):

\[
\begin{align*}
A & := 0; & A & := 0; \\
A & := 1; & \text{if } (<\text{cond}>) & \text{goto L;} \\
B & := A; & A & := 1; \\
B & := A;
\end{align*}
\]

Some optimizations require global analysis (e.g., loop-invariant code motion):

\[
\begin{align*}
\text{while } (<\text{cond}>) & \text{ do} \\
A & := B + C; \\
\text{foo}(A); \\
\text{end};
\end{align*}
\]
Local optimization

Value numbering

- another basic-block level optimization
- combines common subexpression elimination & constant folding
- avoids the graph manipulation involved in \textit{dag} construction

References

- John Cocke and Jack Schwartz in “Programming Languages and Their Compilers; Preliminary Notes” (1970)
- See also Aho, Sethi, and Ullman, pages 292, 293, and 635
Value numbering

Assumptions

- can find basic blocks
- input is in triples
- no knowledge about world before or after the block
- reference’s type is textually obvious
  (tag lhs and rhs)

Input

- basic block \((n\text{ instructions})\)
- symbol table \((w/\text{constant\ bit})\)

Output

- improved basic block \((cse, constant\ folding)\)
- table of available expressions \(\dagger\)
- table of constant values

\(\dagger\) an expression is \textit{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 $\Rightarrow$ same value
  - based solely on information from within the block
  - stored in different places
    * variables and constants $\rightarrow$ symbol table (SYMBOLS)
    * expressions $\rightarrow$ available expression table (AVAIL) & triple

- if an expression’s value is *available* (already computed), we should *not* recompute it
  $\Rightarrow$ re-write subsequent references (*subsumption*)

- constants denoted with a bit in SYMBOLS and in the triple
Value numbering

Principal data structures

CODE

- array of *triples*
- Fields: result, lhs, op, rhs

SYMBOLS

- hash table keyed by variable name
- Fields: name, val, isConstant

AVAIL

- hash table keyed by \((val, op, val)\)
- Fields: lhsVal, op, rhsVal, resultVal, isConstant, instruction

CONSTANTS

- table to hold funky, machine-specific values
- important in cross-compilation
- Fields: val, bits

(a nit)
Value numbering

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 then
      
      SYMBOLS[$lhs[i]$].val $\leftarrow r$
   
   if $r$ is constant then
      
      SYMBOLS[$lhs[i]$].isConstant $\leftarrow$ true

else /* an expression */

   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 CONSTANTS($l, op[i], r$)
      CONSTANTS($l, op[i], r$).bits $\leftarrow$ eval($l \ op[i] \ r$)
      CONSTANTS($l, op[i], r$).val $\leftarrow$ new value number
      $op[i] \leftarrow$ “constant ($l \ op[i] \ r$)”

else

   if ($l, op[i], r$) $\in$ AVAIL then
      
      $op[i] \leftarrow$ “ref AVAIL($l, op[i], r$).resultVal”

   else

      create AVAIL($l, op[i], r$)
      AVAIL($l, op[i], r$).val $\leftarrow$ 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>Triples</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: a ← C4</td>
<td>a ← 4</td>
</tr>
<tr>
<td>T2: i × j</td>
<td></td>
</tr>
<tr>
<td>T3: T2 + C5</td>
<td></td>
</tr>
<tr>
<td>T4: k ← T3</td>
<td>k ← i × j + 5</td>
</tr>
<tr>
<td>T5: C5 × a</td>
<td></td>
</tr>
<tr>
<td>T6: T5 × k</td>
<td></td>
</tr>
<tr>
<td>T7: l ← T6</td>
<td>l ← 5 × a × k</td>
</tr>
<tr>
<td>T8: m ← i</td>
<td>m ← i</td>
</tr>
<tr>
<td>T9: m × j</td>
<td></td>
</tr>
<tr>
<td>T10: i × a</td>
<td></td>
</tr>
<tr>
<td>T11: T9 + T10</td>
<td></td>
</tr>
<tr>
<td>T12: b ← T11</td>
<td>b ← m × j + i × a</td>
</tr>
<tr>
<td>T13: a ← T12</td>
<td>a ← b</td>
</tr>
</tbody>
</table>
Value numbering

Safety

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

Profitability

- assume that load of constant is cheaper than $op$
- assume that reference (or copy) is cheaper than $op$
- 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, *not* name
  – DAG construction has same property
• eliminate duplicate evaluations
• evaluate and fold constant expressions

Can we extend this idea across blocks?
Value numbering across blocks

What would we need to value number across multiple blocks?

1. a *control flow* ordering on the blocks
2. *AVAIL* information for logical predecessors
3. uniform naming scheme for values  (*confluence*)
4. formal definition of *availability*

Terminology

- this kind of analysis is called *data-flow analysis*
- it requires a *control flow graph*
  - nodes represent basic blocks
  - edges represent possible control flow paths
  - an algorithm to construct the control flow graph