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):

```
A := 0;          A := 0;
A := 1;          if (<cond>) goto L;
B := A;          A := 1;
                 B := A;
```

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

```
while (<cond>) do
   A := B + C;
   foo(A);
end;
```

Local optimization

Value numbering

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

References

- John Cocke and Jack Schwartz in "Programming Languages and Their Compilers; Preliminary Notes" (1970)
- "A Survey of Data Flow Analysis Techniques" by K.W. Kennedy (in *Program Flow Analysis*, N.D. Jones and S.S. Muchnick, *editors*)
- 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 instructions*)

symbol table (*w/constant bit*)

Output

improved basic block (*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 $\Rightarrow$ same value
  - based solely on information from within the block
  - stored in different places
    - $\star$ variables and constants $\rightarrow$ symbol table (SYMBOLS)
    - $\star$ 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 (*a nit*)

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

Value numbering

```
for i ← 1 to n
    r ← value number for rhs[i]
    l ← value number for lhs[i]
    if op[i] is a store then
        SYMBOLS[lhs[i]].val ← r
        if r is constant then
            SYMBOLS[lhs[i]].isConstant ← 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 ← eval(l op[i] r)
            CONSTANTS(l, op[i], r).val ← new value number
            op[i] ← “constant (l op[i] r)”
        else
            if (l, op[i], r) ∈ AVAIL then
                op[i] ← “ref AVAIL(l, op[i], r).resultVal”
            else
                create AVAIL(l, op[i], r)
                AVAIL(l, op[i], r).val ← new value number
for i ← 1 to n
    if op[i] is ref or constant then delete instruction i
```

Example

Triples	*Source*
T1: a ← C4	a ← 4
T2: i × j	
T3: T2 + C5	
T4: k ← T3	k ← i × j + 5
T5: C5 × a	
T6: T5 × k	
T7: l ← T6	l ← 5 × a × k
T8: m ← i	m ← i
T9: m × j	
T10: i × a	
T11: T9 + T10	
T12: b ← T11	b ← m × j + i × a
T13: a ← T12	a ← b

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
- elminate 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