DAG for Multiple Statements

DAGs can also handle *assignment* statements

- same variable has different *values*

Solution

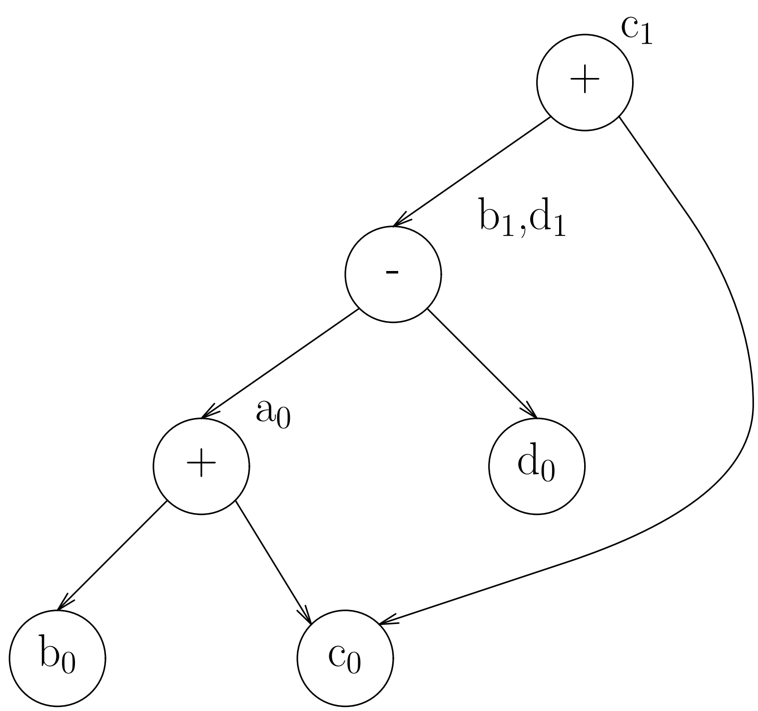
- creates new node for *lhs* — a new x_i
- kills all nodes built from x_{i-1}

A DAG for a *basic block* has labeled nodes

1. *leaves are labeled with unique identifier*
   — either variable names or constants
   — *lvalues* or *rvalues* (*obvious by context*)
   — 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

Example

Code	After Renaming
a ← b + c	a_0 ← b_0 + c_0
b ← a - d	b_1 ← a_0 - d_0
c ← b + c	c_1 ← b_1 + c_0
d ← a - d	d_1 ← a_0 - d_0



DAG Construction Algorithm

Building a DAG

node($< id >$) $\rightarrow$ current DAG for $< id >$

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

2. for each statement `x ← y 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

   do the same for z

4. if $< \texttt{op}, \text{node}(y), \text{node}(z) >$ 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 $\Rightarrow$ same value
  - based solely on information from within the block
    $\Rightarrow$ variables and constants in SYMBOLS
    $\Rightarrow$ 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 ← 1 to n
    r ← value number for rhs[i]
    l ← value number for lhs[i]
    if op[i] is a store /* l := r */ then
        SYMBOLS[lhs[i]].valNum ← r
        if r is constant then
            SYMBOLS[lhs[i]].isConstant ← 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 CONSTANTS(l, op[i], r)
            CONSTANTS(l, op[i], r).bits ← eval(l op[i] r)
            CONSTANTS(l, op[i], r).valNum ← 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).instr"
            else
```

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

CODE				
Instructions		resultValNum	isConst	
(t1)	t1 := 5 + i	3	no	
(t2)	a := t1	3	no	
(t3)	b := 3	4	yes	
(t4)	t4 := b + 6	6	yes	deleted
(t5)	m := t4	6	yes	

SYMBOLS		
name	ValNum	isConst
c5	1	yes
i	2	no
a	3	no
c3	4	yes
b	4	yes
c6	5	yes
m	6	yes

CONSTANTS	
ValNum	bits
1	5
4	3
5	6
6	9

AVAIL				
lhsValNum	op	rhsValNum	resultValNum	instr
1	+	2	3	(t1)

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

Value numbering's focus is on values, not names. This is an important distinction.

- It is easy to attain in *local* optimization.
- It is harder in *global* and *interprocedural* optimization.

Can we extend this idea across blocks?