

Static Single Assignment Form

Last Time

- Static single assignment (SSA) form

Today

- Applications of SSA

Dead Code Elimination for SSA

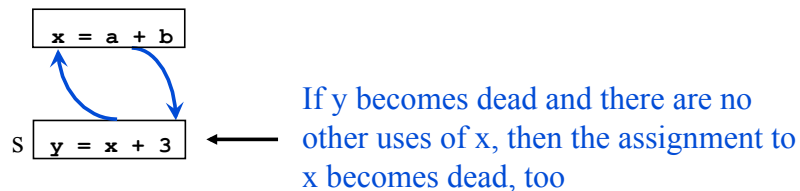
Dead code elimination

while \exists a variable v with no uses and whose def has no other side effects

Delete the statement s that defines v

for each of s 's uses w

Delete the use from list of uses of variable w



- Contrast this approach with one that uses liveness analysis
 - This algorithm updates information incrementally
 - With liveness, we need to invoke liveness and dead code elimination iteratively until we reach a fixed point

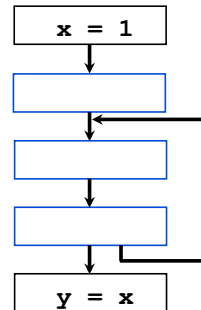
Implementing Simple Constant Propagation

Standard worklist algorithm

- Identifies simple constants
- For each program point, maintains one constant value for each variable

Problem

- Inefficient, since constants may have to be propagated through irrelevant nodes



Solution

- Exploit a sparse dependence representation (e.g., SSA)

Sparse Simple Constant Propagation

Reif and Lewis algorithm

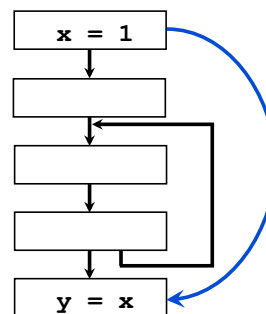
- Identifies simple constants
- Faster than Simple Constants algorithm

SSA edges

- Explicitly connect defs with uses
- How would you do this?

Main Idea

- Iterate over SSA edges instead of over all CFG edges



Sparse Simple Constants Algorithm (Ch. 19 in Appel)

worklist = all statements in SSA

while worklist $\neq \emptyset$

 Remove some statement S from worklist

 if S is $x = \text{phi}(c, c, \dots, c)$ for some constant c

 replace S with $v = c$

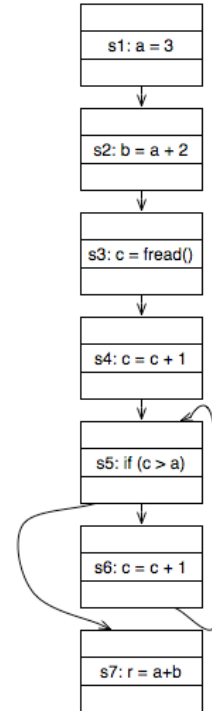
 if S is $x = c$ for some constant c

 delete S from program

 for each statement T that uses x

 substitute c for x in T

 worklist = worklist union {T}



Copy Propagation

Algorithm

worklist = all statements in SSA

while worklist $\neq \emptyset$

 Remove some statement S from worklist

 if S is $x = \text{phi}(y)$ or $x = y$

 for each statement T that uses x

 replace all use of x with y

 worklist = worklist union {T}

 delete S

Reuse Optimization

Idea

- Eliminate redundant operations in the dynamic execution of instructions

How do redundancies arise?

- Loop invariant code (*e.g.*, index calculation for arrays)
- Sequence of similar operations (*e.g.*, method lookup)
- Same value be generated in multiple places in the code

Types of reuse optimization

- Value numbering
- Common subexpression elimination
- Partial redundancy elimination

Local Value Numbering

Idea

- Each variable, expression, and constant is assigned a unique number
- When we encounter a variable, expression or constant, see if it's already been assigned a number
 - If so, use the value for that number
 - If not, assign a new number
- Same number \Rightarrow same value

Example

```
a := b + c
d := b
b := a
e := d + c a
```

```
b → #1 #3
c → #2
b + c is #1 + #2 → #3
a → #3
d → #1
d + c is #1 + #2 → #3
e → #3
```

Local Value Numbering (cont)

Temporaries may be necessary

```

a := b + c
a := b
d := a + c
    
```

```

b → #1
c → #2
b + c is #1 + #2 → #3
a → #3 #1
a + c is #1 + #2 → #3
d → #3
    
```

```

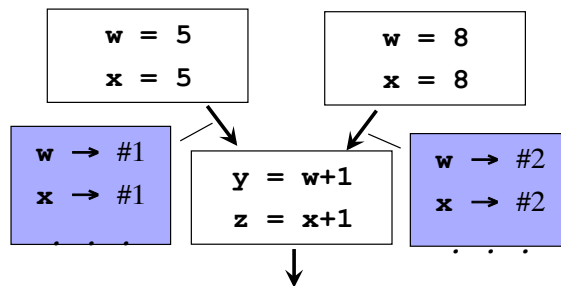
t := b + c
a := b
d := b + c t
    
```

```

b → #1
c → #2
b + c is #1 + #2 → #3
t → #3
a → #1
a + c is #1 + #2 → #3
d → #3
    
```

Global Value Numbering

How do we handle control flow?



Global Value Numbering (cont)

Idea [Alpern, Wegman, and Zadeck 1988]

- Partition program variables into **congruence classes**
- All variables in a particular congruence class have the same value
- SSA form is helpful

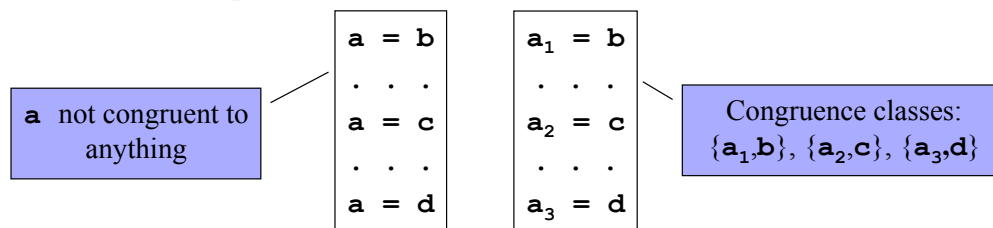
Approaches to computing congruence classes

- Pessimistic
 - Assume no variables are congruent (start with n classes)
 - Iteratively coalesce classes that are determined to be congruent
- Optimistic
 - Assume all variables are congruent (start with one class)
 - Iteratively partition variables that contradict assumption
 - Slower but better results

Role of SSA Form

SSA form is helpful

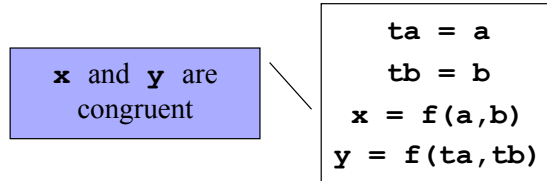
- Allows us to avoid data-flow analysis
- Variables correspond to values



Basis

Idea

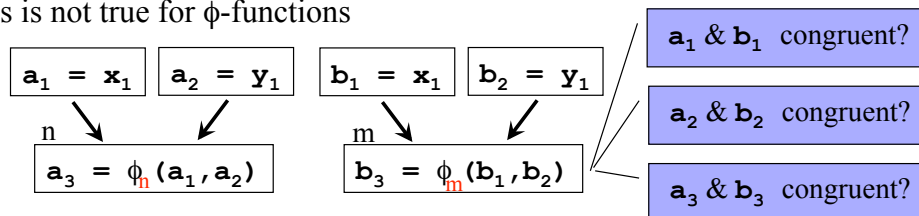
- If x and y are congruent then $f(x)$ and $f(y)$ are congruent



- Use this fact to combine (pessimistic) or split (optimistic) classes

Problem

- This is not true for ϕ -functions



Solution: Label ϕ -functions with join point

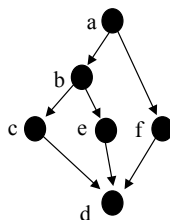
Pessimistic Global Value Numbering

Idea

- Initially each variable is in its own congruence class
- Consider each assignment statement s (reverse postorder in CFG)
 - Update LHS value number with hash of RHS
- Identical value number \Rightarrow congruence

Why reverse postorder?

- Ensures that when we consider an assignment statement, we have already considered definitions that reach the RHS operands



Postorder: d, c, e, b, f, a

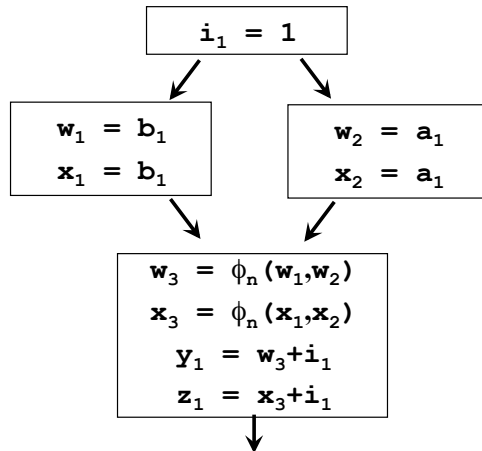
Algorithm

for each assignment of the form: “ $\mathbf{x} = \mathbf{f}(\mathbf{a}, \mathbf{b})$ ”

ValNum[x] \leftarrow UniqueValue() // same for a and b

for each assignment of the form: “ $\mathbf{x} = \mathbf{f}(\mathbf{a}, \mathbf{b})$ ” (in reverse postorder)

ValNum[x] \leftarrow Hash($f \oplus$ ValNum[a] \oplus ValNum[b])



a_1	#1
b_1	#2
i_1	#3
w_1	#4 #2
x_1	#5 #2
w_2	#6 #1
x_2	#7 #1
w_3	#8 $\phi_n(\#2, \#1) \rightarrow \#12$
x_3	#9 $\phi_n(\#2, \#1) \rightarrow \#12$
y_1	#10 $+(\#12, \#3) \rightarrow \#13$
z_1	#11 $+(\#12, \#3) \rightarrow \#13$

CS553 Lecture

Value Numbering

15

Snag!

Problem

- Our algorithm assumes that we consider operands before variables that depend upon it
- Can't deal with code containing loops!

Solution

- Ignore back edges
- Make conservative (worst case) assumption for previously unseen variable (*i.e.*, assume its in its own congruence class)

CS553 Lecture

Value Numbering

16

Optimistic Global Value Numbering

Idea

- Initially all variables in one congruence class
- Split congruence classes when evidence of non-congruence arises
 - Variables that are computed using different functions
 - Variables that are computed using functions with non-congruent operands

Splitting

Initially

- Variables computed using the same function are placed in the same class

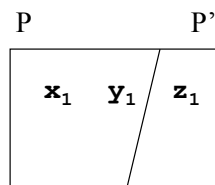
$$x_1 = f(a_1, b_1)$$

...

$$y_1 = f(c_1, d_1)$$

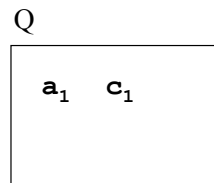
...

$$z_1 = f(e_1, f_1)$$



Iteratively

- *Split* classes when corresponding operands are in different classes
- Example: assume a_1 and c_1 are congruent, but e_1 is congruent to neither

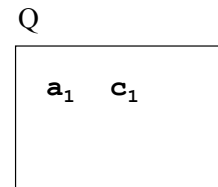
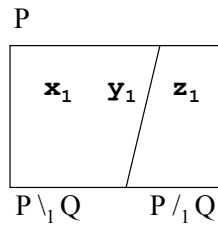


Splitting (cont)

Definitions

- Suppose P and Q are sets representing congruence classes
- Q **splits** P for each i into two sets
 - $P \setminus_i Q$ contains variables in P whose ith operand is in Q
 - $P /_i Q$ contains variables in P whose ith operand is not in Q
- Q **properly splits** P if neither resulting set is empty

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{f}(\mathbf{a}_1, \mathbf{b}_1) \\ &\dots \\ \mathbf{y}_1 &= \mathbf{f}(\mathbf{c}_1, \mathbf{d}_1) \\ &\dots \\ \mathbf{z}_1 &= \mathbf{f}(\mathbf{e}_1, \mathbf{f}_1) \end{aligned}$$



Example

SSA code

$\mathbf{x}_0 = 1$
$\mathbf{y}_0 = 2$
$\mathbf{x}_1 = \mathbf{x}_0 + 1$
$\mathbf{y}_1 = \mathbf{y}_0 + 1$
$\mathbf{z}_1 = \mathbf{x}_0 + 1$

Congruence classes

S_0	$\{\mathbf{x}_0\}$
S_1	$\{\mathbf{y}_0\}$
S_2	$\{\mathbf{x}_1, \mathbf{y}_1, \mathbf{z}_1\}$
S_3	$\{\mathbf{x}_1, \mathbf{z}_1\}$
S_4	$\{\mathbf{y}_1\}$

Worklist: ~~$S_0 = \{\mathbf{x}_0\}$~~ , ~~$S_1 = \{\mathbf{y}_0\}$~~ , ~~$S_2 = \{\mathbf{x}_1, \mathbf{y}_1, \mathbf{z}_1\}$~~

~~$S_3 = \{\mathbf{x}_1, \mathbf{z}_1\}$~~ , ~~$S_4 = \{\mathbf{y}_1\}$~~

S_0 psplit S_0 ? **no** S_0 psplit S_1 ? **no** S_0 psplit S_2 ? **yes!**

$$S_2 \setminus_1 S_0 = \{\mathbf{x}_1, \mathbf{z}_1\} = S_3$$

$$S_2 /_1 S_0 = \{\mathbf{y}_1\} = S_4$$

Algorithm

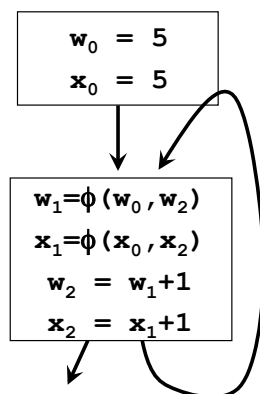
```
worklist  $\leftarrow \emptyset$ 
for each function f
   $C_f \leftarrow \emptyset$ 
  for each assignment of the form “ $x = f(a,b)$ ”
     $C_f \leftarrow C_f \cup \{x\}$ 
  worklist  $\leftarrow$  worklist  $\cup \{C_f\}$ 
  CC  $\leftarrow$  CC  $\cup \{C_f\}$ 
while worklist  $\neq \emptyset$ 
  Delete some D from worklist
  for each class C properly split by D (at operand i)
    CC  $\leftarrow$  CC  $- C$ 
    worklist  $\leftarrow$  worklist  $- C$ 
    Create new congruence classes  $C_j \leftarrow \{C \setminus_i D\}$  and  $C_k \leftarrow \{C /_i D\}$ 
    CC  $\leftarrow$  CC  $\cup C_j \cup C_k$ 
    worklist  $\leftarrow$  worklist  $\cup C_j \cup C_k$ 
```

Note: see paper for optimization

Comparing Optimistic and Pessimistic

Differences

- Handling of loops
- Pessimistic makes worst-case assumptions on back edges
- Optimistic requires actual contradiction to split classes



Role of SSA

Single global result

- Single def reaches each use
- No data (flow value) at each point

No data flow analysis

- Optimistic: Iterate over congruence classes, not CFG nodes
- Pessimistic: Visit each assignment once

ϕ -functions

- Make data-flow merging explicit
- Treat like normal functions after subscripting them for each merge point

Concepts

SSA construction

- Place phi nodes
- Variable renaming

Transformation from SSA to executable code depends on the optimizations dead-code elimination and coalescing in the register allocator to remove extra moves

Some optimizations that are simpler and more efficient with SSA

- dead-code elimination
- constant propagation
- copy propagation
- value numbering

Others that weren't covered

- induction variable detection, strength reduction, and elimination
- register allocation
- ...

Next Time

Lecture

- Parallelism and Data Locality, start reading chapter 11 in dragon book

Suggested Exercise

- Use any programming language to write the class declarations for the equivalence class and variable structures described at the bottom of page 6 in the value numbering chapter.
- Instantiate the data structures for the small example programs in Figures 7.2 and 7.6 and perform pessimistic value numbering. Remember to convert to SSA first.
- After performing value numbering, how will the program be transformed?