Putting Programs into SSA Form

Assume we have the CFG for a program, which we want to put into SSA form. We must:

- Rename all definitions and uses of variables
- Decide where to add phi functions

Renaming variable definitions is trivial—each assignment is to a new, unique variable.

After phi functions are added (at the heads of selected basic blocks), only one variable definition (the most recent in the block) can reach any use. Thus renaming uses of variables is easy.

Placing Phi Functions

Let b be a block with a definition to some variable, v. If b contains more than one definition to v, the last (or most recent) applies.

What is the first basic block following b where some other definition to v *as well as* b's definition can reach?

In blocks dominated by b, b's definition *must* have been executed, though other later definitions may have overwritten b's definition.

Domination Frontiers (Again)

Recall that the Domination Frontier of a block b, is defined as

$$DF(N) = \{Z \mid M \rightarrow Z \ \& \ (N \text{ dom } M) \ \& \ \neg(N \text{ sdom } Z)\}$$

The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren't themselves strictly dominated by N.

Assume that an initial assignment to all variables occurs in b_0 (possibly of some special "uninitialized value.")

We will need to place a phi function at the start of all blocks in b's Domination Frontier.

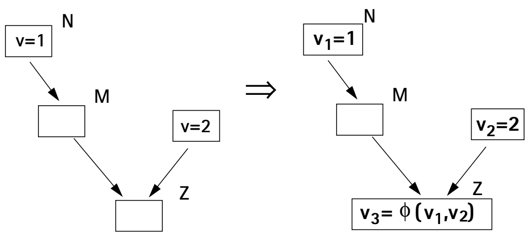
The phi functions will join the definition to v that occurred in b (or in a block dominated by b) with definitions occurring on paths that don't include b.

After phi functions are added to blocks in DF(b), the domination frontier of blocks with newly added phi's will need to be computed (since phi functions imply assignment to a new v_i variable).

Examples of How Domination Frontiers Guide Phi Placement

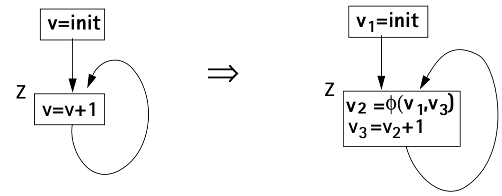
DF(N) =
{Z | M→Z & (N dom M) & ¬(N sdom Z)}

Simple Case:



Here, (N dom M) but ¬(N sdom Z), so a phi function is needed in Z.

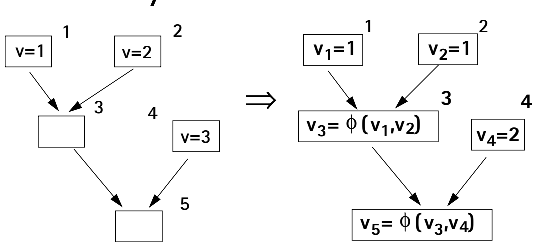
Loop:



Here, let M = Z = N. M→Z, (N dom M) but ¬(N sdom Z), so a phi function *is* needed in Z.

DF(N) =
{Z | M→Z & (N dom M) & ¬(N sdom Z)}

Sometimes Phi's must be Placed Iteratively



Now, DF(b1) = {b3}, so we add a phi function in b3. This adds an assignment into b3. We then look at DF(b3) = {b5}, so another phi function must be added to b5.

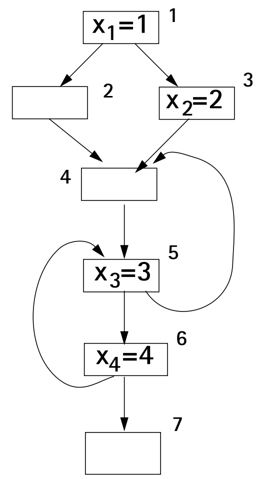
Phi Placement Algorithm

To decide what blocks require a phi function to join a definition to a variable v in block b:

1. Compute D_1 = DF(b).
   Place Phi functions at the head of all members of D_1.
2. Compute D_2 = DF(D_1).
   Place Phi functions at the head of all members of D_2-D_1.
3. Compute D_3 = DF(D_2).
   Place Phi functions at the head of all members of D_3-D_2-D_1.
4. Repeat until no additional Phi functions can be added.

```
PlacePhi{
 For (each variable v ∈ program) {
    For (each block b ∈ CFG ){
      PhiInserted(b) = false
      Added(b) = false }
    List = φ
    For (each b ∈ CFG that assigns to V ){
      Added(b) = true
      List = List U {b}   }
    While (List ≠ φ) {
       Remove any b from List
       For (each d ∈ DF(b)){
          If (! PhiInserted(d)) {
             Add a Phi Function to d
             PhiInserted(d) = true
             If (! Added(d)) {
                  Added(d) = true
                  List = List U {d}
             }
          }
       }
    }
  }
}
```


Example



Initially, List={1,3,5,6}

Process 1: DF(1) = φ

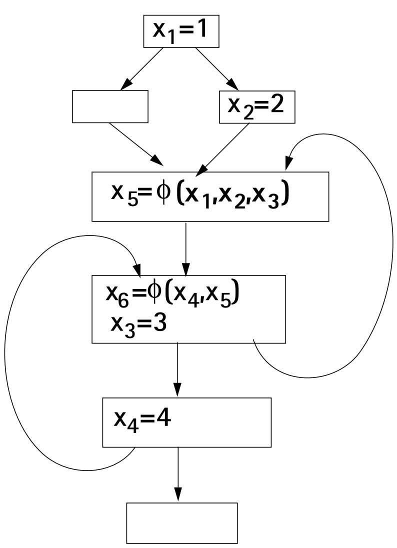
Process 3: DF(3) = 4, so add 4 to List and add phi fct to 4.

Process 5: DF(5)={4,5} so add phi fct to 5.

Process 5: DF(6) = {5}

Process 4: DF(4) = {4}

We will add Phi's into blocks 4 and 5. The arity of each phi is the number of in-arcs to its block. To find the args to a phi, follow each arc "backwards" to the sole reaching def on that path.



SSA and Value Numbering

We already know how to do available expression analysis to determine if a previous computation of an expression can be reused.

A limitation of this analysis is that it can't recognize that two expressions that aren't syntactically identical may actually still be equivalent.

For example, given

```
t1 = a + b
c = a
t2 = c + b
```

Available expression analysis won't recognize that `t1` and `t2` must be equivalent, since it doesn't track the fact that `a = c` at `t2`.

Value Numbering

An early expression analysis technique called *value numbering* worked only at the level of basic blocks. The analysis was in terms of "values" rather than variable or temporary names.

Each non-trivial (non-copy) computation is given a number, called its *value number*.

Two expressions, using the same operators and operands with the same value numbers, must be equivalent.

For example,

```
t1 = a + b
c = a
t2 = c + b
```

is analyzed as

```
v1 = a
v2 = b
t1 = v1 + v2
c = v1
t2 = v1 + v2
```

Clearly `t2` is equivalent to `t1` (and hence need not be computed).

In contrast, given

```
t1 = a + b
a = 2
t2 = a + b
```

the analysis creates

```
v1 = a
v2 = b
t1 = v1 + v2
v3 = 2
t2 = v3 + v2
```

Clearly `t2` is not equivalent to `t1` (and hence will need to be recomputed).

Extending Value Numbering to Entire CFGs

The problem with a global version of value numbering is how to reconcile values produced on different flow paths. But this is exactly what SSA is designed to do!

In particular, we know that an ordinary assignment

```
x = y
```

does *not* imply that all references to x can be replaced by y after the assignment. That is, an assignment *is not* an assertion of value equivalence.

But,

in SSA form

$\mathbf{x_i = y_j}$

does mean the two values are *always* equivalent after the assignment. If $\mathbf{y_j}$ reaches a use of $\mathbf{x_i}$, that use of $\mathbf{x_i}$ *can* be replaced with $\mathbf{y_j}$.

Thus in SSA form, an assignment *is* an assertion of value equivalence.

We will assume that simple variable to variable copies are removed by substituting equivalent SSA names.

This alone is enough to recognize some simple value equivalences.

As we saw,

$\mathbf{t_1 = a_1 + b_1}$

$\mathbf{c_1 = a_1}$

$\mathbf{t_2 = c_1 + b_1}$

becomes

$\mathbf{t_1 = a_1 + b_1}$

$\mathbf{t_2 = a_1 + b_1}$

Partitioning SSA Variables

Initially, all SSA variables will be partitioned by the *form* of the expression assigned to them.

Expressions involving different constants or operators won't (in general) be equivalent, even if their operands happen to be equivalent.

Thus

$\mathbf{v_1 = 2}$ and $\mathbf{w_1 = a_2 + 1}$

are always considered inequivalent.

But,

$\mathbf{v_3 = a_1 + b_2}$ and $\mathbf{w_1 = d_1 + e_2}$

may *possibly* be equivalent since both involve the same operator.

Phi functions are potentially equivalent only if they are in the same basic block.

All variables are initially considered equivalent (since they all initially are considered uninitialized until explicit initialization).

After SSA variables are grouped by assignment form, groups are split.

If a_i op b_y and c_k op d_l

are in the same group (because they both have the same operator, op)

and $a_i \not\equiv c_k$ or $b_j \not\equiv d_l$

then we split the two expressions apart into different groups.

We continue splitting based on operand inequivalence, until no more splits are possible. Values still grouped are equivalent.

Example

```
if (...) {
  a1=0
  if (...)
    b1=0
  else {
    a2=x0
    b2=x0 }
  a3=φ(a1,a2)
  b3=φ(b1,b2)
  c2=*a3
  d2=*b3 }
else {
  b4=10 }
a5=φ(a0,a3)
b5=φ(b3,b4)
c3=*a5
d3=*b5
e3=*a5
```

Initial Groupings:

G_1=[a_0,b_0,c_0,d_0,e_0,x_0]
G_2=[a_1=0, b_1=0]
G_3=[a_2=x_0, b_2=x_0]
G_4=[b_4=10]
G_5=[a_3=$\phi(a_1,a_2)$, b_3=$\phi(b_1,b_2)$]
G_6=[a_5=$\phi(a_0,a_3)$, b_5=$\phi(b_3,b_4)$]
G_7=[c_2=*a_3, d_2=*b_3, d_3=*b_5, c_3=*a_5, e_3=*a_5]

Now b_4 isn't equivalent to anything, so split a_5 and b_5. In G_7 split operands b_3, a_5 and b_5. We now have


```
if (...) {
  a1=0
  if (...)
    b1=0
  else {
    a2=x0
    b2=x0 }
  a3=φ(a1,a2)
  b3=φ(b1,b2)
  c2=*a3
  d2=*b3 }
else {
  b4=10 }
a5=φ(a0,a3)
b5=φ(b3,b4)
c3=*a5
d3=*b5
e3=*a5
```

Final Groupings:

G_1=[a_0,b_0,c_0,d_0,e_0,x_0]
G_2=[a_1=0, b_1=0]
G_3=[a_2=x_0, b_2=x_0]
G_4=[b_4=10]
G_5=[a_3=$\phi(a_1,a_2)$, b_3=$\phi(b_1,b_2)$]
G_{6a}=[a_5=$\phi(a_0,a_3)$]
G_{6b}=[b_5=$\phi(b_3,b_4)$]
G_{7a}=[c_2=*a_3, d_2=*b_3]
G_{7b}=[d_3=*b_5]
G_{7c}=[c_3=*a_5, e_3=*a_5]

Variable e_3 can use c_3's value and d_2 can use c_2's value.

Limitations of Global Value Numbering

As presented, our global value numbering technique doesn't recognize (or handle) computations of the same expression that produce different values along different paths.

Thus in

```
[a1=1        ]      [a2=2        ]
[t1=a1+b0    ]      [t2=a2+b0    ]
        \              /
         [a3=φ(a1,a2)]
         [t3=a3+b0   ]
```

variable a_3 isn't equivalent to either a_1 or a_2.

But,
we can still remove a redundant computation of a+b by moving the computation of t_3 to each of its predecessors:

```
[a1=1        ]      [a2=2        ]
[t1=a1+b0    ]      [t2=a2+b0    ]
[e1=a1+b0    ]      [e2=a2+b0    ]
        \              /
         [e3=φ(e1,e2)]
         [t3=e3      ]
```

Now a redundant computation of a+b is evident in each predecessor block. Note too that this has a nice register targeting effect—e_1, e_2 and e_3 can be readily mapped to the same live range.

The notion of moving expression computations above phi functions also meshes nicely with notion of partial redundancy elimination. Given

a_1=1
t_1=a_1+b_0

a_2=2

a_3=ϕ(a_1,a_2)
t_3=a_3+b_0

moving **a+b** above the phi produces

a_1=1
t1=a_1+b_0

a_2=2
t_2=a_2+b_0

t_3=ϕ(t_1,t_2)

Now **a+b** is computed only once on each path, an improvement.

Reading Assignment

- Read "Global Optimization by Suppression of Partial Redundancies," Morel and Renvoise.
(Linked from the class Web page.)
- Read "Profile Guided Code Positioning," Pettis and Hansen.
(Linked from the class Web page.)

Partial Redundancy Analysis

Partial Redundancy Analysis is a boolean-valued data flow analysis that generalizes available expression analysis.

Ordinary available expression analysis tells us if an expression must already have been evaluated (and not killed) along *all* execution paths.

Partial redundancy analysis, originally developed by Morel & Renvoise, determines if an expression has been computed along *some* paths. Moreover, it tells us where to add new computations of the expression to change a partial redundancy into a full redundancy.

This technique *never* adds computations to paths where the computation isn't needed. It strives to avoid having any redundant computation on any path.

In fact, this approach includes movement of a loop invariant expression into a preheader. This loop invariant code movement is just a special case of partial redundancy elimination.

Basic Definition & Notation

For a Basic Block i and a particular expression, e:

$Transp_i$ is true if and only if e's operands aren't assigned to in i.

$Transp_i \equiv \neg Kill_i$

$Comp_i$ is true if and only if e is computed in block i and is not killed in the block after computation.

$Comp_i \equiv Gen_i$

$AntLoc_i$ (Anticipated Locally in i) is true if and only if e is computed in i and there are no assignments to e's operands prior to e's computation.

If $AntLoc_i$ is true, computation of e in block i will be redundant if e is available on entrance to i.

We'll need some standard data flow analyses we've seen before:

$AvIn_i$ = Available In for block i

= 0 (false) for b_0

$= \underset{p \in Pred(i)}{AND} AvOut_p$

$AvOut_i = Comp_i$ OR ($AvIn_i$ AND $Transp_i$)

$\equiv Gen_i$ OR ($AvIn_i$ AND $\neg Kill_i$)

We *anticipate* an expression if it is very busy:

$AntOut_i = VeryBusyOut_i$

= 0 (false) if i is an exit block

$= \underset{s \in Succ(i)}{AND} AntIn_s$

$AntIn_i = VeryBusyIn_i$

$= AntLoc_i$ OR ($Transp_i$ AND $AntOut_i$)