**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

\[
\begin{align*}
t_1 &= a + b \\
c &= a \\
t_2 &= c + b
\end{align*}
\]

Available expression analysis won't recognize that \( t_1 \) and \( t_2 \) must be equivalent, since it doesn't track the fact that \( a = c \) at \( t_2 \).

---

**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, given

\[
\begin{align*}
t_1 &= a + b \\
c &= a \\
t_2 &= c + b
\end{align*}
\]

is analyzed as

\[
\begin{align*}
v_1 &= a \\
v_2 &= b \\
t_1 &= v_1 + v_2 \\
c &= v_1 \\
t_2 &= v_1 + v_2
\end{align*}
\]

Clearly \( t_2 \) is equivalent to \( t_1 \) (and hence need not be computed).

---

In contrast, given

\[
\begin{align*}
t_1 &= a + b \\
a &= 2 \\
t_2 &= a + b
\end{align*}
\]

the analysis creates

\[
\begin{align*}
v_1 &= a \\
v_2 &= b \\
t_1 &= v_1 + v_2 \\
c &= v_1 \\
t_2 &= v_3 + v_2
\end{align*}
\]

Clearly \( t_2 \) is not equivalent to \( t_1 \) (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
\[ x_i = y_j \]
does mean the two values are always equivalent after the assignment. If \( y_j \) reaches a use of \( x_i \), that use of \( x_i \) can be replaced with \( y_j \).

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,
\[
\begin{align*}
t_1 &= a_1 + b_1 \\
c_1 &= a_1 \\
t_2 &= c_1 + b_1
\end{align*}
\]
becomes
\[
\begin{align*}
t_1 &= a_1 + b_1 \\
t_2 &= a_1 + b_1
\end{align*}
\]
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
\[
\begin{align*}
v_1 &= 2 \\
w_1 &= a_2 + 1
\end{align*}
\]
are always considered inequivalent. But,
\[
\begin{align*}
v_3 &= a_1 + b_2 \\
w_1 &= d_1 + e_2
\end{align*}
\]
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 \( b_j \) and \( c_k \) op \( d_l \) are in the same group (because they both have the same operator, op) and \( a_i \neq c_k \) or \( b_j \neq d_l \) 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**

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

Initial Groupings:

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

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=\phi(a1,a2)
        b3=\phi(b1,b2)
        c2=*a3
        d2=*b3
    }
    else {
        b4=10
        a5=\phi(a0,a3)
        b5=\phi(b3,b4)
        c3=*a5
        d3=*b5
        e3=*a5
    }
}
```

**Final Groupings:**

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

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

```
Variable e_3 can use c_3's value and a_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

```
a_1=1
b_1=a_1+b_0
t_1=a_1+b_0
```

```
a_2=2
b_2=a_2+b_0
t_2=a_2+b_0
```

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:

$$
\begin{array}{c}
a_1=1 \\
t_1=a_1+b_0 \\
e_1=a_1+b_0 \\
\hline
a_2=2 \\
t_2=a_2+b_0 \\
e_2=a_2+b_0 \\
\hline
e_3=\phi(e_1,e_2) \\
t_3=e_3
\end{array}
$$

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 moving $a+b$ above the phi produces

$$
\begin{array}{c}
a_1=1 \\
t_1=a_1+b_0 \\
a_2=2 \\
t_2=a_2+b_0 \\
\hline
a_3=\phi(a_1,a_2) \\
t_3=a_3+b_0
\end{array}
$$

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:

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

Transpi ≡ ¬Killi

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

Compi ≡ Geni

AntLoci (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 AntLoci 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:

Avlni = Available In for block i

Avlni = 0 (false) for b0

Avlni = AND AvOutpi

Avlni = AND Transpi

AvOuti = Comp i OR (Avlni AND Transpi)

AvOuti = Comp i OR (Avlni AND ¬Killi)

Avlni = Geni OR (Avlni AND ¬Killi)
We *anticipate* an expression if it is very busy:

\[
\text{AntOut}_i = \text{VeryBusyOut}_i = 0 \text{ (false) if } i \text{ is an exit block}
\]

\[
\text{AntIn}_i = \text{VeryBusyIn}_i = \text{AntLoc}_i \text{ OR } \\
\quad (\text{Trans}_i \text{ AND AntOut}_i)
\]

---

**Partial Availability**

Partial availability is similar to available expression analysis except that an expression must be computed (and not killed) along *some* (not necessarily *all*) paths:

\[
\text{PavOut}_i = 0 \text{ (false) for } b_0 = \\
\text{PavOut}_p = \text{Compi OR } \\
\quad (\text{PavIn}_i \text{ AND Trans}_p)
\]

---

**Where are Computations Added?**

The key to partial redundancy elimination is deciding where to add computations of an expression to change partial redundancies into full redundancies (which may then be optimized away).

We’ll start with an “enabling term.”

\[
\text{Const}_i = \text{AntIn}_i \text{ AND } \\
\quad [\text{PavIn}_i \text{ OR } (\text{Trans}_p \text{ AND } \neg \text{AntLoc}_i)]
\]

This term say that we require the expression to be:

1. Anticipated at the start of block i (somebody wants the expression) and
2a. The expression must be partially available (to perhaps transform into full availability)

or

2b. The block neither kills nor computes the expression.
Next, we compute PPIni and PPOuti.
PP means “possible placement” of a computation at the start (PPIni) or end (PPOuti) of a block.

These values determine whether a computation of the expression would be “useful” at the start or end of a basic block.

PPOuti = 0 (false) for all exit blocks

We try to move computations “up” (nearer the start block).

It makes sense to compute an expression at the end of a block if it makes sense to compute at the start of all the block’s successors.

PPIni = 0 (false) for b₀.

= Constᵢ
    AND (AntLocᵢ OR (Transpᵢ AND PPOutiᵢ))
    AND (PPOutpᵢ OR AvOutpᵢ)

p ∈ Pred(i)

To determine if PPIniᵢ is true, we first check the enabling term. It makes sense to consider a computation of the expression at the start of block i if the expression is anticipated (wanted) and partially available or if the expression is anticipated (wanted) and it is neither computed nor killed in the block.

We then check that the expression is anticipated locally or that it is unchanged within the block and possibly positioned at the end of the block.

Finally, we check that all the block’s predecessors either have the expression available at their ends or are willing to position a computation at their end.

Note also, the bi-directional nature of this equation.

**Inserting New Computations**

After PPIni and PPOuti are computed, we decide where computations will be inserted:

Insertᵢ = PPOutiᵢ AND (¬ AvOutᵢ) AND (¬ PPIniᵢ OR ¬ Transpᵢ)

This rule states that we really will compute the expression at the end of block i if this is a possible placement point and the expression is not already computed and available and moving the computation still earlier doesn’t work because the start of the block isn’t a possible placement point or because the block kills the expression.
Removing Existing Computations

We've added computations of the expression to change partial redundancies into full redundancies. Once this is done, expressions that are fully redundant can be removed.
But where?
Remove_i = AntLoc_i and PPIni_i

This rule states that we remove computation of the expression in blocks where it is computed locally and might be moved to the block's beginning.

Partial Redundancy Subsumes Available Expression Analysis

Using partial redundancy analysis, we can find (and remove) ordinary fully redundant available expressions.
Consider a block, b, in which:
(1) The expression is computed (anticipated) locally
and
(2) The expression is available on entrance
Point (1) tells us that AntLoc_b is true

Moreover, recall that
PPIni_b = Const_b AND (AntLoc_b OR ... )
AND (AvOut_p OR ... )
P ∈ Pred(b)

Const_b = AntIn_b AND [PavIn_b OR ... ]
We know AntLoc_b is true ⇒ AntIn_b = true.
Moreover, AvIn_b = true ⇒ PavIn_b = true.
Thus Const_b = true.
If AvIn_b is true, AvOut_p is true for all P ∈ Pred(b).
Thus PPIni_b AND AntLoc_b = true = Remove_b

Are any computations added earlier (to any of b's ancestors)?
No:
Insert_i = PPOut_i AND (¬ AvOut_i) AND (¬ PPIni_i OR ¬ Transp_i)
But for any ancestor, i, between the computation of the expression and b, AvOut_i is true, so Insert_i must be false.
Examples of Partial Redundancy Elimination

At block 3, x+3 is partially, but not fully, redundant.

\[ \text{PPIn}_3 = \text{Const}_3 \text{ AND } \left( \text{AntLoc}_3 \text{ OR ... } \right) \]
\[ \text{AND} \left( \text{PPOut}_p \text{ OR AvOut}_p \right) \]
\[ p \in \text{Pred}(3) \]
\[ \text{Const}_3 = \text{AntIn}_3 \text{ AND } [\text{PavIn}_3 \text{ OR ...}] \]

Now AntIn$_3$ = true and PavIn$_3$ = true.
\[ \text{Const}_3 = \text{true AND true = true} \]

Remove$_3$ = AntLoc$_3$ and PPIn$_3$ = true AND true = true, so x+3 is removed from block 3.

Is x+3 inserted at the end of block 2? (It shouldn’t be).

Insert$_2$ = PPOut$_2$ AND (¬ AvOut$_2$)
\[ \text{AND} (\neg \text{PPIn}_2 \text{ OR } \neg \text{Transp}_2) = \]
\[ \text{PPOut}_2 \text{ AND false AND } \neg \text{PPIn}_2 \text{ OR } \neg \text{Transp}_2 = \text{false.} \]

We now have

Computations May Move Up Several Blocks

Again, at block 4, x+3 is partially, but not fully, redundant.

\[ \text{PPIn}_4 = \text{Const}_4 \text{ AND } \left( \text{AntLoc}_4 \text{ OR ... } \right) \]
\[ \text{AND} \left( \text{PPOut}_p \text{ OR AvOut}_p \right) \]
\[ p \in \text{Pred}(4) \]
Const₄ = AntIn₄ AND [PavIn₄ OR ...]
Now AntIn₄ = true and PavIn₄ = true.
Const₄ = true AND true = true
PPOut₃ = PPIn₄.
AntLoc₄ = true.
PPIn₄ = true AND true
AND (PPOutₚ OR AvOutₚ) =
ₚ ∈ Pred(4)
PPOut₃ = true.
PPIn₃ = Const₃ AND
((Transp₃ AND PPOut₃) OR ... )
AND (PPOutₚ OR AvOutₚ)
ₚ ∈ Pred(3)
Const₃ = AntIn₃ AND [PavIn₃ OR ...]
AntIn₃ = true and PavIn₃ = true.

Insert₃ = PPOut₃ AND (¬ AvOut₃)
AND (¬ PPIn₃ OR ¬ Transp₃) =
true AND (true) AND
(false OR false) = false
so x+3 is not inserted at the end of block 3.
Insert₂ = PPOut₂ AND (¬ AvOut₂)
AND (¬ PPIn₂ OR ¬ Transp₂) =
PPOut₂ AND (false)
AND (¬ PPIn₂ OR ¬ Transp₂)=false,
so x+3 is not inserted at the end of block 2.

Where Do We Insert Computations?

Insert₁ = PPOut₁ AND (¬ AvOut₁)
AND (¬ PPIn₁ OR ¬ Transp₁) =
true AND (true) AND
(¬ PPIn₁ OR true) = true
so x+3 is inserted at the end of block 3.
Remove₄ = AntLoc₄ and PPIn₄
= true AND true = true, so x+3 is removed from block 4.

We finally have