Thus at the beginning of the while, the solution is \{b1\}.
At the head if the If, the solution is
\[
\text{Id} \cup (\text{Id} \circ f_C \circ \text{Id}) \cup
(\text{Id} \circ f_C \circ \text{Id} \circ f_C \circ \text{Id}) \cup \ldots \circ \{b1\}
\]
\[= \{b1\} \cup \{b4,b5\} \cup \{b4,b5\} \cup \ldots = \{b1,b4,b5\}\]
At the head of the then part of the If, the solution is \text{Id}(\{b1,b4,b5\}) = \{b1,b4,b5\}.

Static Single Assignment Form

Many of the complexities of optimization and code generation arise from the fact that a given variable may be assigned to in many different places.
Thus reaching definition analysis gives us the set of assignments that may reach a given use of a variable.
Live range analysis must track all assignments that may reach a use of a variable and merge them into the same live range.
Available expression analysis must look at all places a variable may be assigned to and decide if any kill an already computed expression.

What If

each variable is assigned to in only one place?
(Much like a named constant).
Then for a given use, we can find a single unique definition point.
But this seems impossible for most programs—or is it?
In Static Single Assignment (SSA)
Form each assignment to a variable, v, is changed into a unique assignment to new variable, v_i.
If variable v has n assignments to it throughout the program, then (at least) n new variables, v_1 to v_n, are created to replace v. All uses of v are replaced by a use of some v_i.

Phi Functions

Control flow can’t be predicted in advance, so we can’t always know which definition of a variable reached a particular use.
To handle this uncertainty, we create phi functions.
As illustrated below, if v_i and v_j both reach the top of the same block, we add the assignment
\[v_k \leftarrow \phi(v_i,v_j)\]
to the top of the block.
Within the block, all uses of v become uses of v_k (until the next assignment to v).
What does $\phi(v_i, v_j)$ Mean?

One way to read $\phi(v_i, v_j)$ is that if control reaches the phi function via the path on which $v_i$ is defined, $\phi$ “selects” $v_i$; otherwise it “selects” $v_j$.

Phi functions may take more than two arguments if more than two definitions might reach the same block.

Through phi functions we have simple links to all the places where $v$ receives a value, directly or indirectly.

Example

Original CFG

CFG in SSA Form

In SSA form computing live ranges is almost trivial. For each $x_i$ include all $x_j$ variables involved in phi functions that define $x_i$.

Initially, assume $x_1$ to $x_6$ (in our example) are independent. We then union into equivalence classes $x_i$ values involved in the same phi function or assignment.

Thus $x_1$ to $x_3$ are unioned together (forming a live range). Similarly, $x_4$ to $x_6$ are unioned to form a live range.

Constant Propagation in SSA

In SSA form, constant propagation is simplified since values flow directly from assignments to uses, and phi functions represent natural “meet points” where values are combined (into a constant or $\perp$).

Even conditional constant propagation fits in. As long as a path is considered unreachable, it variables are set to T (and therefore ignored at phi functions, which meet values together).
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

\[
\text{DF}(N) = \{Z \mid M \rightarrow Z \land (N \text{ dom } M) \land \neg (N \text{ sdom } Z)\}
\]

The Domination Frontier of a basic block \( N \), \( \text{DF}(N) \), 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 \( \text{DF}(b) \), 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

\[
\text{DF}(N) = \{Z \mid M \rightarrow Z \land (N \text{ dom } M) \land \neg (N \text{ sdom } Z)\}
\]

Simple Case:

Here, \((N \text{ dom } M)\) but \(\neg(N \text{ sdom } Z)\), so a phi function is needed in \( Z \).

Loop:

Here, let \( M = Z = N \). \( M \rightarrow Z \), \((N \text{ dom } M)\) but \(\neg(N \text{ sdom } Z)\), so a phi function is needed in \( Z \).

\[
\text{DF}(N) = \{Z \mid M \rightarrow Z \land (N \text{ dom } M) \land \neg (N \text{ sdom } Z)\}
\]
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

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

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,

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

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
\[ t_1 = a + b \]
\[ a = 2 \]
\[ t_2 = a + b \]
The analysis creates
\[ v_1 = a \]
\[ v_2 = b \]
\[ t_1 = v_1 + v_2 \]
\[ v_3 = 2 \]
\[ t_2 = v_3 + v_2 \]
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,
\[ t_1 = a_1 + b_1 \]
\[ c_1 = a_1 \]
\[ t_2 = c_1 + b_1 \]
becomes
\[ t_1 = a_1 + b_1 \]
\[ t_2 = a_1 + b_1 \]
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

\[ v_1 = 2 \quad \text{and} \quad w_1 = a_2 + 1 \]

are always considered inequivalent.

But,

\[ v_3 = a_1 + b_2 \quad \text{and} \quad w_1 = d_1 + e_2 \]

may possibly be equivalent since both involve the same operator.

Example

```cpp
if (...) {
  a_1 = 0
  if (...) {  
    b_1 = 0
    else {  
      a_2 = x_0
      b_2 = x_0
      a_3 = \phi(a_1, a_2)
      b_3 = \phi(b_1, b_2)
      c_2 = *a_3
      d_2 = *b_3  
    }
  }  
else {  
  b_4 = 10  
  a_5 = \phi(a_0, a_3)
  b_5 = \phi(b_1, b_4)
  c_3 = *a_5
  d_3 = *b_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 have

```
if (...) {
  a_1 = 0
  if (...) {  
    b_1 = 0
    else {  
      a_2 = x_0
      b_2 = x_0
      a_3 = \phi(a_1, a_2)
      b_3 = \phi(b_1, b_2)
      c_2 = *a_3
      d_2 = *b_3  
    }
  }  
else {  
  b_4 = 10  
  a_5 = \phi(a_0, a_3)
  b_5 = \phi(b_1, b_4)
  c_3 = *a_5
  d_3 = *b_5
  e_3 = *a_5  
}
```

Final Groupings:

```
if (...) {
  a_1 = 0
  if (...) {  
    b_1 = 0
    else {  
      a_2 = x_0
      b_2 = x_0
      a_3 = \phi(a_1, a_2)
      b_3 = \phi(b_1, b_2)
      c_2 = *a_3
      d_2 = *b_3  
    }
  }  
else {  
  b_4 = 10  
  a_5 = \phi(a_0, a_3)
  b_5 = \phi(b_1, b_4)
  c_3 = *a_5
  d_3 = *b_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

\[
\begin{align*}
a_1 &= 1 \\
t_1 &= a_1 + b_0 \\
a_2 &= 2 \\
t_2 &= a_2 + b_0 \\
a_3 &= \phi(a_1, a_2) \\
t_3 &= a_3 + b_0
\end{align*}
\]

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:

\[
\begin{align*}
a_1 &= 1 \\
t_1 &= a_1 + b_0 \\
e_1 &= a_1 + b_0 \\
a_2 &= 2 \\
t_2 &= a_2 + b_0 \\
e_2 &= a_2 + b_0 \\
e_3 &= \phi(e_1, e_2) \\
t_3 &= e_3
\end{align*}
\]

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

\[
\begin{align*}
a_1 &= 1 \\
t_1 &= a_1 + b_0 \\
a_2 &= 2 \\
t_2 &= a_2 + b_0 \\
a_3 &= \phi(a_1, a_2) \\
t_3 &= a_3 + b_0
\end{align*}
\]

moving \( a+b \) above the phi produces

\[
\begin{align*}
a_1 &= 1 \\
t_1 &= a_1 + b_0 \\
a_2 &= 2 \\
t_2 &= a_2 + b_0 \\
t_3 &= \phi(t_1, t_2)
\end{align*}
\]

Now \( a+b \) is computed only once on each path, an improvement.