Iterative Execution

Repeat Loop

Given the basic blocks:

we create a single composite block:

Here \( b_B \) is the loop body, and \( b_C \) is the loop control.
If the loop iterates once, the transfer function is $f_C \circ f_B$.

If the loop iterates twice, the transfer function is $(f_C \circ f_B) \circ (f_C \circ f_B)$.

Considering all paths, the transfer function is $(f_C \circ f_B) \land (f_C \circ f_B)^2 \land ...$

Define $\text{fix } f \equiv f \land f^2 \land f^3 \land ...$

The transfer function of repeat is then

$$f_{\text{repeat}} = \text{fix}(f_C \circ f_B)$$
While Loop.

Given the basic blocks:

\[ \text{while} \]

\[ b_C \]

\[ b_B \]

we create a single composite block:

\[ b_{\text{while}} \]

Here again \( b_B \) is the loop body, and \( b_C \) is the loop control. The loop always executes \( b_C \) at least once, and always executes \( b_C \) as the last block before exiting.
The transfer function of a while is therefore

\[ f_{\text{while}} = f_C \land \text{fix}(f_C \circ f_B) \circ f_C \]
Evaluating Fixed Points

For lattices of height \( H \), and monotone transfer functions, \( \text{fix } f \) needs to look at no more than \( H \) terms.

In practice, we can give \( \text{fix } f \) an operational definition, suitable for implementation:

Evaluate

\[
(fix \ f)(x) \!
\begin{align*}
\text{prev} &= \text{soln} = f(x); \\
\text{while} (\text{prev} \neq \text{new} = f(\text{prev})) \{
&\text{prev} = \text{new}; \\
&\text{soln} = \text{soln} \land \text{new};
\}
\}
\]

return soln;
}
Example—Reaching Definitions

The transfer functions are either constant-valued \( f_1 = \{b_1\}, f_4 = \{b_4\}, f_5 = \{b_5\} \) or identity functions \( f_2 = f_3 = f_6 = f_7 = \text{Id} \).
First we isolate and reduce the conditional:

\[ f_C = f_4 \circ f_3 \land f_5 \circ f_3 = \{b_4\} \circ \text{Id} \cup \{b_5\} \circ \text{Id} = \{b_4, b_5\} \]
Substituting, we get

We can combine $b_C$ and $b_6$, to get a block equivalent to $b_C$. That is,

$$f_6 \circ f_C = \text{Id} \circ f_C = f_C$$
We now have

We isolate and reduce the while loop formed by $b_2$ and $b_C$, creating $b_W$.

The transfer function is

$$f_W = f_2 \land (\text{fix}(f_2 \circ f_C)) \circ f_2 =$$

$$\text{Id} \cup (\text{fix}(\text{Id} \circ f_C)) \circ \text{Id} =$$

$$\text{Id} \cup (\text{fix}(f_C)) =$$

$$\text{Id} \cup (f_C \land f_C^2 \land f_C^3 \land ...) =$$

$$\text{Id} \cup \{b_4, b_5\}$$
We now have

\[ f_P = \text{Id} \circ (\text{Id} \cup \{b4,b5\}) \circ \{b1\} = \{b1,b4,b5\}. \]

We compose these three sequential blocks to get the whole solution, \( f_P \).

These are the definitions that reach the end of the program.

We can expand subgraphs to get the solutions at interior blocks.
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, vi.
If variable v has n assignments to it throughout the program, then (at least) n new variables, v₁ to vₙ, are created to replace v. All uses of v are replaced by a use of some vi.
**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 2 arguments if more than 2 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 ⊥).

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).
Example

\[
i = 6 \\
j = 1 \\
k = 1
\]
repeat
\[
\text{if } (i == 6) \\
\quad k = 0 \\
\text{else} \\
\quad i = i + 1 \\
\quad i = i + k \\
\quad j = j + 1
\]
until (i == j)

\[
i_1 = 6 \\
j_1 = 1 \\
k_1 = 1
\]
repeat
\[
i_2 = \phi (i_1, i_5) \\
j_2 = \phi (j_1, j_3) \\
k_2 = \phi (k_1, k_4)
\]
if (i_2 == 6)
\[
k_3 = 0
\]
else
\[
i_3 = i_2 + 1 \\
i_4 = \phi (i_2, i_3) \\
k_4 = \phi (k_3, k_2) \\
i_5 = i_4 + k_4 \\
j_3 = j_2 + 1
\]
until (i_5 == j_3)

<table>
<thead>
<tr>
<th>Pass1</th>
<th>i_1</th>
<th>j_1</th>
<th>k_1</th>
<th>i_2</th>
<th>j_2</th>
<th>k_2</th>
<th>k_3</th>
<th>i_3</th>
<th>i_4</th>
<th>k_4</th>
<th>i_5</th>
<th>j_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>6\land T</td>
<td>1\land T</td>
<td>1\land T</td>
<td>0</td>
<td>T</td>
<td>6\land T</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Pass2</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>6\land 6</td>
<td>\bot</td>
<td>\bot</td>
<td>0</td>
<td>T</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>\bot</td>
</tr>
</tbody>
</table>

We have determined that i = 6 everywhere.
**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

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

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

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

Simple Case:

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

\[v_1=1 \quad \rightarrow \quad v_2=\phi(v_1,v_2)\]
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 $Z$.

$DF(N) = \{Z | M \rightarrow Z \& (N \text{ dom } M) \& \neg(N \text{ sdom } Z)\}$
Sometimes Phi’s must be Placed Iteratively

Now, $DF(b_1) = \{b_3\}$, so we add a phi function in $b_3$. This adds an assignment into $b_3$. We then look at $DF(b_3) = \{b_5\}$, so another phi function must be added to $b_5$. 
**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 \( v \in \text{program} \) ) {
  For (each block \( b \in \text{CFG} \)) {
    PhiInserted(b) = false
    Added(b) = false
  }
  List = \( \emptyset \)
  For (each \( b \in \text{CFG} \) that assigns to \( V \) ) {
    Added(b) = true
    List = List U \{ b \}
  }
  While (List \( \neq \emptyset \) ) {
    Remove any \( b \) from List
    For (each \( d \in \text{DF}(b) \) ) {
      If (! PhiInserted(d)) {
        Add a Phi Function to \( d \)
        PhiInserted(d) = true
        If (! Added(d)) {
          Added(d) = true
          List = List U \{ d \}
        }
      }
    }
  }
}
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.

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

Process 1: DF(1) = ∅

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}
$x_1 = 1$

$x_2 = 2$

$x_5 = \phi(x_1, x_2, x_3)$

$x_6 = \phi(x_4, x_5)$

$x_3 = 3$

$x_4 = 4$
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,
\[
\begin{align*}
t1 &= a + b \\
c &= a \\
t2 &= c + b
\end{align*}
\]
is analyzed as
\[
\begin{align*}
v1 &= a \\
v2 &= b \\
t1 &= v1 + v2 \\
c &= v1 \\
t2 &= v1 + v2
\end{align*}
\]
Clearly $t2$ is equivalent to $t1$ (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,

\[
\begin{align*}
t_1 &= a_1 + b_1 \\
c_1 &= a_1 \\
t_2 &= c_1 + b_1
\end{align*}
\]

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

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

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.
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 \( a_i \text{ op } b_y \) and \( c_k \text{ op } d_l \) 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

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:

G₁=[a₀,b₀,c₀,d₀,e₀,x₀]

G₂=[a₁=0, b₁=0]

G₃=[a₂=x₀, b₂=x₀]

G₄=[b₄=10]

G₅=[a₃=φ(a₁,a₂), b₃=φ(b₁,b₂)]

G₆=[a₅=φ(a₀,a₃), b₅=φ(b₃,b₄)]

G₇=[c₂=*a₃, 
    d₂=*b₃, 
    d₃=*b₅, 
    c₃=*a₅, 
    e₃=*a₅]

Now b₄ isn’t equivalent to anything, so split a₅ and b₅. In G₇ split operands b₃, a₅ and b₅. 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:
G₁=[a₀,b₀,c₀,d₀,e₀,x₀]  
G₂=[a₁=0, b₁=0]  
G₃=[a₂=x₀, b₂=x₀]  
G₄=[b₄=10]  
G₅=[a₃=φ(a₁,a₂),  
    b₃=φ(b₁,b₂)]  
G₆ₐ=[a₅=φ(a₀,a₃)]  
G₆ₜ=[b₅=φ(b₃,b₄)]  
G₇ₐ=[c₂=*a₃,  
    d₂=*b₃]  
G₇ₜ=[d₃=*b₅]  
G₇ₜ=[c₃=*a₅,  
    e₃=*a₅]  

Variable e₃ can use c₃'s value and a₂ can use c₂'s value.