Objectives

- Understand what SSA form is and how to convert a program into SSA
- Define dominance frontiers and an efficient algorithm for computing DFs
- Use DFs to efficiently covert a program into SSA form
- Understand the notion of control-dependence and how to compute it

Static Single Assignment (SSA) Form

References

- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. on Programming Languages and Systems*, 13(4), Oct. 1991, pp. 451–490.
- Muchnick, Section 8.11 (*partially covered*).

What is SSA?

- Informally, a program can be converted into *SSA form* as follows:
  - Each assignment to a variable is given a unique name
  - All of the uses reached by that assignment are renamed.
- Easy for straight-line code:

$$V \leftarrow 4$$

Static Single Assignment (SSA) Form

References

- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. on Programming Languages and Systems*, 13(4), Oct. 1991, pp. 451–490.
- Muchnick, Section 8.11 (*partially covered*).

What is SSA?

- Informally, a program can be converted into *SSA form* as follows:
  - Each assignment to a variable is given a unique name
  - All of the uses reached by that assignment are renamed.
- Easy for straight-line code:

$V \leftarrow 4$	$V_0 \leftarrow 4$
$\leftarrow V + 5$	$\leftarrow V_0 + 5$
$V \leftarrow 6$	$V_1 \leftarrow 6$
$\leftarrow V + 7$	$\leftarrow V_1 + 7$

Static Single Assignment with Control Flow

- *2-way branch*:

```
if (...)
        X = 5;
else
        X = 3;

Y = X;
```

Static Single Assignment with Control Flow

- *2-way branch*:

```
if (...)
        X = 5;
else
        X = 3;

Y = X;
```

```
if (...)
        X0 = 5;
else
        X1 = 3;
X2 = φ(X0, X1);
Y0 = X2;
```

Static Single Assignment with Control Flow

- *2-way branch*:

```
if (...)
        X = 5;
else
        X = 3;

Y = X;
```

```
if (...)
        X0 = 5;
else
        X1 = 3;
X2 = φ(X0, X1);
Y0 = X2;
```

- *While loop*:

```
j=1;
while (j < X)
    ++j;
N = j;
```

Static Single Assignment with Control Flow

- *2-way branch*:

```
if (...)
      X = 5;
else
      X = 3;

Y = X;
```

```
if (...)
      X0 = 5;
else
      X1 = 3;
X2 = φ(X0, X1);
Y0 = X2;
```

- *While loop*:

```
j=1;
while (j < X)
    ++j;
N = j;
```

```
      j = 1;
      if (j ≥ X)
          goto E;
S:
      j = j+1;
      if (j < X)
          goto S;
E:
      N = j;
```

Static Single Assignment with Control Flow

- *2-way branch*:

```
if (...)
      X = 5;
else
      X = 3;

Y = X;
```

```
if (...)
      X0 = 5;
else
      X1 = 3;
X2 = φ(X0, X1);
Y0 = X2;
```

- *While loop*:

```
j=1;
while (j < X)
    ++j;
N = j;
```

```
      j = 1;
      if (j ≥ X)
          goto E;
S:
      j = j+1;
      if (j < X)
          goto S;
E:
      N = j;
```

```
      j0 = 1;
      if (j0 < X0)
          goto E;
S:
      j1 = φ(j0, j2);
      j2 = j1+1;
      if (j2 < X0)
          goto S;
E:
      j3 = φ(j0, j2);
      N0 = j3;
```

Review of Some Basic Terms

- **Value**:
- **Storage location** (register or memory)
- **Variable**:
- **Pointer**:
- **Alias**:
- **Reference to a variable**:

- **Use of a variable**: A use of variable X is a reference that *may read* the value stored in the location named X.
- **Definition of a variable**: A definition (def) of a variable X is a reference that *may store* a value into the location named X.

  Examples: Assignment; FOR; input I/O

- **Ambiguity**:

Unambiguous def : *guaranteed* to store to X — *must*

Ambiguous def : *may* store to X — *may*

Similarly, ambiguous/unambiguous use.

Q. Where does ambiguity come from?

Definition of SSA Form

- **Definition (ϕ Functions)**:
  In a basic block B with N predecessors, $P_1,\ P_2,\ \ldots,\ P_N$,

  $$X = \phi(V_1, V_2, \ldots, V_N)$$

  assigns $X = V_j$ if control enters block B from P_j, $1 \leq j \leq N$.

- *Properties of ϕ-functions*:
  - ϕ is not an executable operation.
  - ϕ has exactly as many arguments as the number of incoming BB edges
  - Think about ϕ argument V_i as being evaluated on CFG edge from predecessor P_i to B

- **Definition (SSA form)**:
  A program is in SSA form if:
  1. each variable is assigned a value in exactly one statement
  2. each use of a variable is *dominated* by the definition

Which Variables Can We Convert?

Which of these can we convert? And when?

Scalars?

Arrays?

Structures?

Which Variables Can We Convert?

Which of these can we convert? And when?

Scalars?

Arrays?

Structures?

General Criteria

Convert all variables to SSA form, *except* . . .

- *Arrays*: Array elements do not have an explicit name
- *Variables that may have aliases*: do not have a unique name
- *Volatile variables*: can be modified "unexpectedly"

E.g., In LLVM, only variables in virtual registers are in SSA form.

Advantages of SSA

- Explicit def-use and use-def chains:

  Def-use chain: The set of uses reached by a particular definition.

  Use-def chain: The set of defs reaching a particular use.

  These are foundation of many dataflow optimizations.

  $\Longrightarrow$ Specifically **enables sparse optimizations.**
- Compact, *flow-sensitive* def-use information
  $\Longleftarrow$ fewer def-use edges per variable: one per CFG edge
- No anti- and output dependences on SSA variables
- Explicit merging of values (ϕ): key additional information
- Can serve as IR for code transformations

Disadvantages of SSA

- Size of SSA program is O(N^2) for an ordinary program with N variables.
- Usually not used for structures and arrays
- May not be used for scalar variables with aliases
- *If used as IR*: Must be converted back to code (*Not too bad*)
- *Otherwise*: Must be recomputed frequently (*Often bad*)

Constructing *SSA* Form

Simple method

1. insert ϕ-functions for every variable at every join
2. solve REACHING DEFINITIONS
3. rename each use to the def that reaches it (*unique*)

What's wrong with this approach

1. too many ϕ-functions (*precision*)
2. too many ϕ-functions (*space*)
3. too many ϕ-functions (*time*)

Might be good enough for some transformations

Everything else is an optimization

The *SSA*-Construction Algorithm: Intuition

Key question: Where do we place ϕ-functions?

- **Example:**

```
1:  if (...) then {
2:      V = ...;
3:      if (...)  {
4:          U = V + 1;    // No phi for V needed here
5:      } else {
6:          U = V + 2;    // No phi for V needed here
7:      }
8:      W = U + 1;        // No phi for V needed here
9:  }
10: ...                   // phi for V _is_ needed here
```

The SSA-Construction Algorithm: Intuition(cont)

Informal Conditions:

If node X contains an assignment to a variable V, then a ϕ must be inserted in each node Z such that:

1. there is a non-null path X $\rightarrow^+$ Z, and
2. there is a path from ENTRY to Z that does not go through X,
3. Z is the first node on the path X $\rightarrow^+$ Z that satisfies (2).

The Dominance Frontier *of the node X is exactly the set of nodes Z that satisfy the above condition!*

Intuition for Placement Conditions:

(1) $\Longrightarrow$ the value of V computed in X reaches Z

(2) $\Longrightarrow$ there is a path that does not go through X, so some other value of V reaches Z along that path (ignore bugs due to uses of uninitialized variables). So, two values must be merged at X with a ϕ.

(3) $\Longrightarrow$ The ϕ for the value coming from X is placed in Z and not in some earlier node on the path X $\rightarrow^+$ Z.

The *SSA*-Construction Algorithm: Intuition (cont)

Iterating the Placement Conditions:

- After a ϕ is inserted at Z, the above process must be repeated for Z because the ϕ is effectively a new definition of V.
- For each node X and variable V, there must be at most one ϕ for V in X. This means that the above iterative process can be done with a single worklist of nodes for each variable V, initialized to handle all original assignment nodes X simultaneously.

Flavors of SSA:

Minimal SSA : As few as possible, subject to above condition

Pruned SSA : As few as possible, subject to above condition *and* no dead ϕ-functions

Dominance Frontiers

Dominance: *Recall definitions of:*

1. X *dominates* Y, or X *dom* Y or X $\underline{\gg}$ Y
2. X *strictly dominates* Y, or X $\gg$ Y
3. X is the *immediate dominator* of Y, or X = *idom*(Y)
4. *Dominator tree*

Dominance frontiers:

The *dominance frontier* of node X is the set of nodes Y such that X dominates a predecessor of Y, but X does not strictly dominate Y.

DF(X) = {Y | $\exists$ p $\in$ Pred(Y) : X $\underline{\gg}$ p and X $\not\gg$ Y}

The *dominance frontier* can be subdivided into two components:

DF_{local}(X) $\equiv$ {Y $\in$ Succ(X) | X $\not\gg$ Y}

DF_{up}(Z) $\equiv$ {Y $\in$ DF(Z) | *idom*(Z) $\not\gg$ Y}

Then,

$$DF(X) = DF_{local}(X) \bigcup \bigcup_{Z \in Children(X)} DF_{up}(Z)$$

Computing Dominance Frontiers

Algorithm for computing dominance frontiers

for each X in a bottom-up traversal of the dominator tree
    DF(X) $\leftarrow \emptyset$
    for each Y $\in$ succ(X) /* *local* */
        if *idom*(Y) $\neq$ X then
            DF(X) $\leftarrow$ DF(X) $\bigcup$ {Y}
    for each Z $\in$ children(X) /* *up* */
        for each Y $\in$ DF(Z)
            if *idom*(Y) $\neq$ X then
                DF(X) $\leftarrow$ DF(X) $\bigcup$ {Y}

Theorem 1

The dominance frontier algorithm is correct.

Refer to paper for proof.

Iterated Dominance Frontiers

Dominance frontiers for a set of nodes

Extend the dominance frontier mapping from nodes to sets of nodes:

$DF(\mathcal{L}) = \bigcup_{X \in \mathcal{L}} DF(X)$

The *iterated* dominance frontier $DF^+(\mathcal{L})$ is the limit of the sequence:

$DF_1 = DF(\mathcal{L})$

$DF_{i+1} = DF(\mathcal{L} \bigcup DF_i)$

Theorem 2

The set of nodes that need ϕ-functions for any variable V is the iterated dominance frontier $DF^+(\mathcal{L})$, where $\mathcal{L}$ is the set of nodes that may modify V.

Refer to paper for proof.

Computing Static Single Assignment Form

Complete algorithm

Compute the dominance frontiers

Insert ϕ-functions

Rename the variables

Theorem 3

Any program can be put into minimal SSA form using this algorithm.

Refer to paper for proof.

Computing Static Single Assignment Form

Inserting ϕ-functions

```
for each variable V
    HasAlready ← ∅
    EverOnWorkList ← ∅
    WorkList ← ∅
    for each node X that may modify V
        EverOnWorkList ← EverOnWorkList ⋃ {X}
        WorkList ← WorkList ⋃ {X}

    while WorkList ≠ ∅
        remove X from W
        for each Y ∈ DF(X)
            if Y ∉ HasAlready then
                insert a φ-node for V at Y
                HasAlready ← HasAlready ⋃ {Y}
                if Y ∉ EverOnWorkList then
                    EverOnWorkList ← EverOnWorkList ⋃ {Y}
                    WorkList ← WorkList ⋃ {Y}
```

Renaming the Variables

Intuition:

Renaming defs is easy. To rename each use of V:

(a) *Use in a non-ϕ statement*: Use immediately dominating definition of V (+ ϕ nodes inserted for V).

$\Longrightarrow$ *preorder on Dominator Tree!*

(b) *Use in a ϕ operand*: Use definition that immediately dominates incoming CFG edge (not ϕ)

$\Longrightarrow$ rename the ϕ operand when processing the predecessor basic block!

Data Structures:

Stacks - an array of stacks, one for each original variable V

Stacks[V] = subscript of most recent definition of V. Initially, Stacks[V] = EmptyStack, $\forall$ V

Counters - an array of counters, one for each original variable

Counters[V] = number of assignments to V processed, Initially. Counters[V] = 0, $\forall$ V

procedure GenName(Variable V)

1. i $\leftarrow$ Counters[V]
2. replace V by V_i
3. push i onto Stacks[V]
4. Counters[V] $\leftarrow$ i + 1

Computing Static Single Assignment Form

The Renaming Algorithm Initially, call Rename(**Entry**)

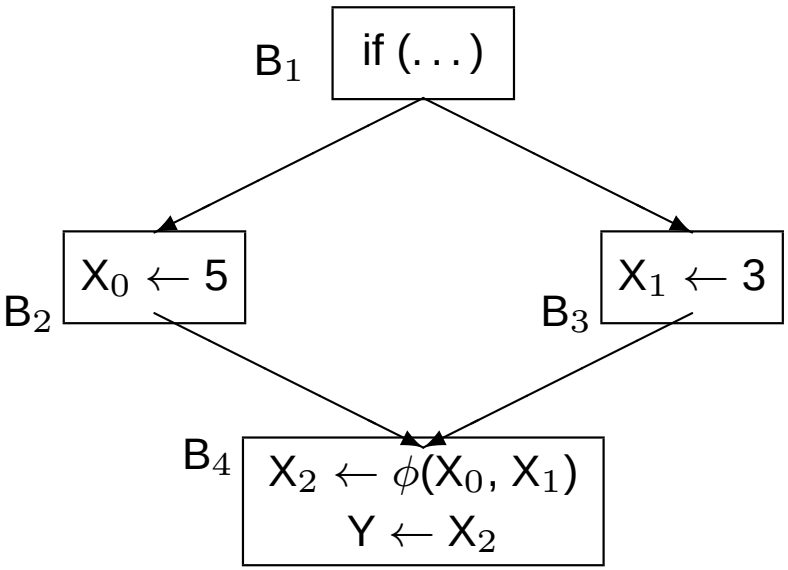
```
procedure Rename(Block X)
    for each φ-node P in X
        GenName(LHS(P))
    for each statement A in X
        for each variable V ∈ RHS(A)
            replace V by V_i, where i = Top(Stacks[V])
        for each variable V ∈ LHS(A)
            GenName(V)
    for each Y ∈ Succ(X)
        j ← position in Y's φ-functions corresponding to X
        for each φ-node P in Y
            replace the j^th operand of RHS(P) by V_i
                where i = Top(Stacks[V])
    for each Y ∈ Children(X)
        Rename(Y)
    for each φ-node or statement A in X
        for each V_i ∈ LHS(A)
            pop Stacks[V]
```

Translating out of Static Single Assignment form

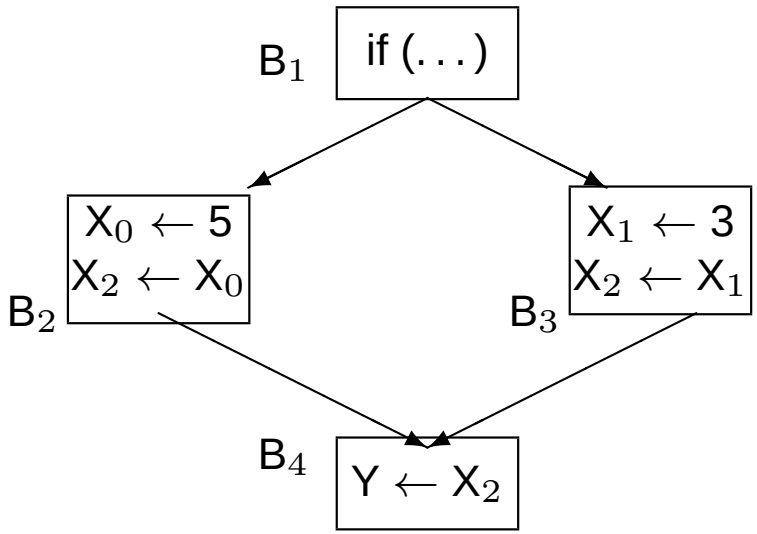
Overview:

1. Dead-code elimination (prune dead ϕs)
2. Replace ϕ-functions with copies in predecessors
3. Register allocation with copy coalescing

Before (2)



After (2)



Control Dependence

Definition: In a CFG, node Y is *control-dependent* on node B if

1. There is a non-empty path $N_0 = B, N_1, N_2, ..., N_k = Y$ such that Y postdominates $N_1 \ldots N_k$, *and*
2. Y does not strictly postdominate B

Definition: The *Reverse Control Flow Graph* (RCFG) of a CFG has the same nodes as CFG and has edge $Y \rightarrow X$ if $X \rightarrow Y$ is an edge in CFG.

Control Dependence (continued)

Computing Control Dependence

Key observation: Node Y is control-dependent on B *iff* $B \in DF(Y)$ in RCFG.

Algorithm:

1. Build RCFG
2. Build dominator tree for RCFG
3. Compute dominance frontiers for RCFG
4. Compute $CD(B) \equiv \{Y : B \in DF(Y)\}$.
   $CD(B)$ gives the nodes that are control-dependent on B.