CS738: Advanced Compiler Optimizations

Static Single Assignment (SSA)

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs738
Department of CSE, IIT Kanpur

Agenda

- SSA Form
- Constructing SSA form
- Properties and Applications

SSA Form

- Developed by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck,

SSA Form

- Developed by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck,
  - in 1980s while at IBM.

SSA Form

- Developed by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck,
  - in 1980s while at IBM.
- *Static Single Assignment* – A variable is assigned only once in program text

SSA Form

- Developed by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck,
  - in 1980s while at IBM.
- *Static Single Assignment* – A variable is assigned only once in program text
  - May be assigned multiple times if program is executed

What is SSA Form?

- An Intermediate Representation

What is SSA Form?

- An Intermediate Representation
- Sparse representation

What is SSA Form?

- An Intermediate Representation
- Sparse representation
  - Definitions sites are directly associated with use sites

What is SSA Form?

- An Intermediate Representation
- Sparse representation
  - Definitions sites are directly associated with use sites
- Advantage

What is SSA Form?

- An Intermediate Representation
- Sparse representation
  - Definitions sites are directly associated with use sites
- Advantage
  - Directly access points where relevant data flow information is available

SSA IR

- In SSA Form

SSA IR

- In SSA Form
  - Each variable has exactly one definition

SSA IR

- In SSA Form
  - Each variable has exactly one definition

  $\Rightarrow$ A use of a variable is reached by exactly one definition

SSA IR

- In SSA Form
  - Each variable has exactly one definition

  $\Rightarrow$ A use of a variable is reached by exactly one definition
- Control flow like traditional programs

SSA IR

- In SSA Form
  - Each variable has exactly one definition
  - $\Rightarrow$ A use of a variable is reached by exactly one definition
- Control flow like traditional programs
- Some *magic* is needed at *join* nodes

Example

```
i = 0;
...
i = i + 1;
...
j = i * 5;
...
```

Example

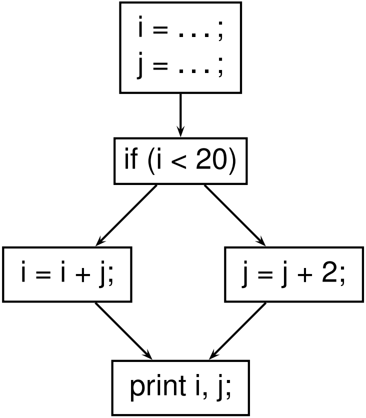
```
i = 0;
...
i = i + 1;
...
j = i * 5;
...
```

$\xrightarrow{SSA}$

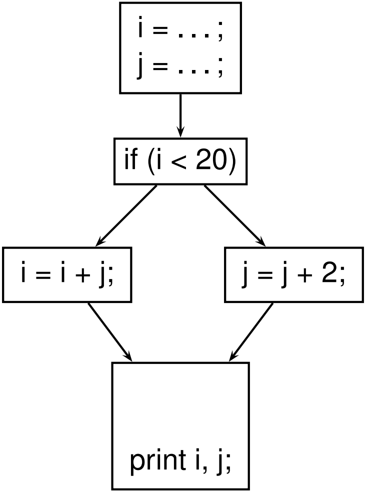
```
i1 = 0;
...
i2 = i1 + 1;
...
j1 = i2 * 5;
...
```

SSA Example

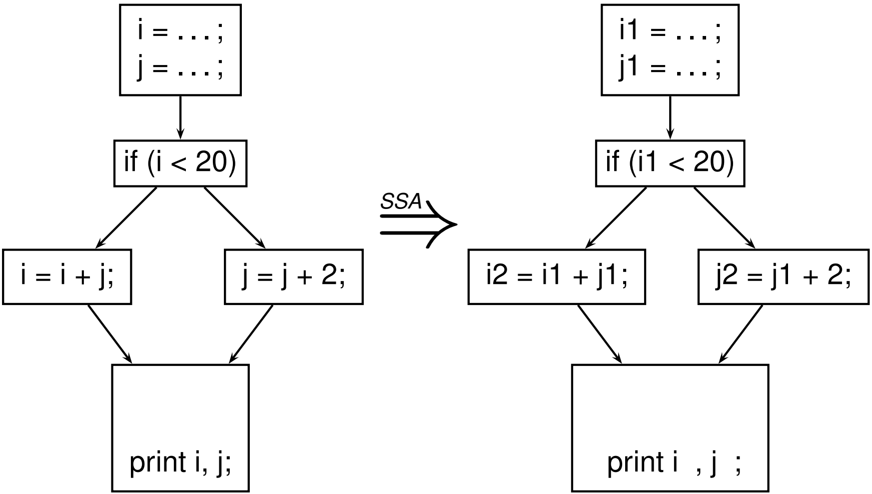
```
i = ...;
j = ...;
if (i < 20)
  i = i + j;
else
  j = j + 2;
print i, j;
```



SSA Example

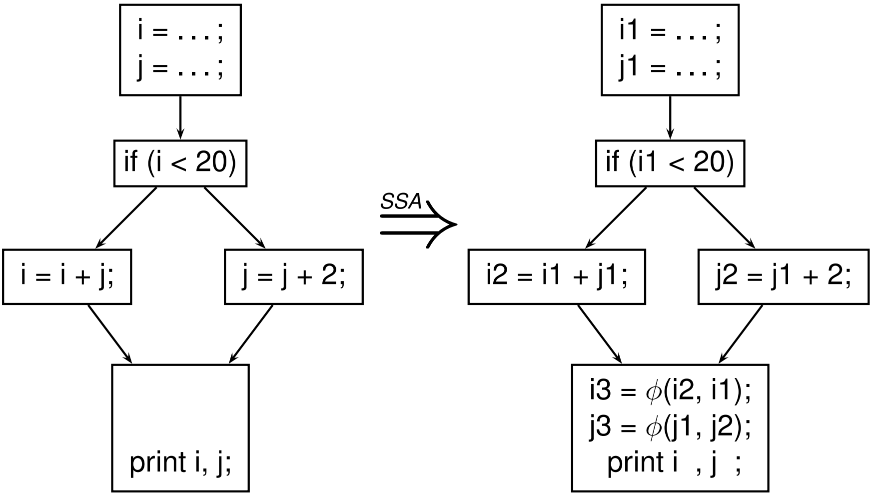


SSA Example



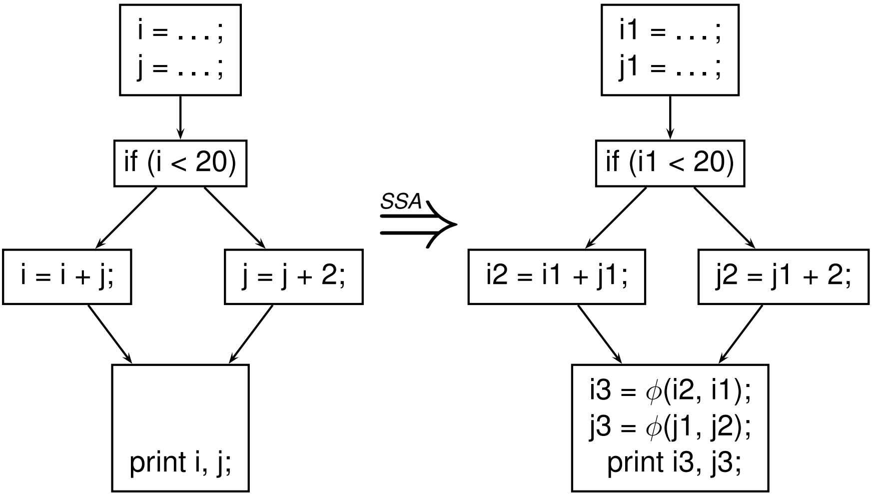
i = . . . ;
j = . . . ;
if (i < 20)
i = i + j;
j = j + 2;
print i, j;
SSA
i1 = . . . ;
j1 = . . . ;
if (i1 < 20)
i2 = i1 + j1;
j2 = j1 + 2;
print i , j ;

SSA Example



i = . . . ;
j = . . . ;
if (i < 20)
i = i + j;
j = j + 2;
print i, j;
SSA
i1 = . . . ;
j1 = . . . ;
if (i1 < 20)
i2 = i1 + j1;
j2 = j1 + 2;
i3 = φ(i2, i1);
j3 = φ(j1, j2);
print i , j ;

SSA Example



i = . . . ;
j = . . . ;
if (i < 20)
i = i + j;
j = j + 2;
print i, j;
SSA
i1 = . . . ;
j1 = . . . ;
if (i1 < 20)
i2 = i1 + j1;
j2 = j1 + 2;
i3 = φ(i2, i1);
j3 = φ(j1, j2);
print i3, j3;

SSA Example

```
i = ...;
j = ...;
if (i < 20)
  i = i + j;
else
  j = j + 2;


print i, j;
```

$\xRightarrow{SSA}$

```
i1 = ...;
j1 = ...;
if (i1 < 20)
  i2 = i1 + j1;
else
  j2 = j1 + 2;
i3 = φ(i2, i1);
j3 = φ(j1, j2);
print i3, j3;
```

The *magic*: ϕ function

- ϕ is used for selection

The *magic*: ϕ function

- ϕ is used for selection
  - One out of multiple values at join nodes

The *magic*: ϕ function

- ϕ is used for selection
  - One out of multiple values at join nodes
- Not every join node needs a ϕ

The *magic*: ϕ function

- ϕ is used for selection
  - One out of multiple values at join nodes
- Not every join node needs a ϕ
  - Needed only if multiple definitions reach the node

The *magic*: ϕ function

- ϕ is used for selection
  - One out of multiple values at join nodes
- Not every join node needs a ϕ
  - Needed only if multiple definitions reach the node
- Examples?

But. . . What is ϕ?

- What does ϕ operation mean in a machine code?

But. . . What is ϕ?

- What does ϕ operation mean in a machine code?
- ϕ is a conceptual entity

But. . . What is ϕ?

- What does ϕ operation mean in a machine code?
- ϕ is a conceptual entity
- Statically equivalent to choosing one of the arguments "non-deterministicly"

But. . . What is ϕ?

- What does ϕ operation mean in a machine code?
- ϕ is a conceptual entity
- Statically equivalent to choosing one of the arguments "non-deterministicly"
- No direct translation to machine code

But... What is ϕ?

- What does ϕ operation mean in a machine code?
- ϕ is a conceptual entity
- Statically equivalent to choosing one of the arguments "non-deterministicly"
- No direct translation to machine code
  - typically mimicked using "copy" in predecessors

But. . . What is ϕ?

- What does ϕ operation mean in a machine code?
- ϕ is a conceptual entity
- Statically equivalent to choosing one of the arguments “non-deterministicly”
- No direct translation to machine code
  - typically mimicked using “copy” in predecessors
  - Inefficient

But. . . What is ϕ?

- What does ϕ operation mean in a machine code?
- ϕ is a conceptual entity
- Statically equivalent to choosing one of the arguments "non-deterministicly"
- No direct translation to machine code
  - typically mimicked using "copy" in predecessors
  - Inefficient
  - Practically, the inefficiency is compensated by dead code elimination and register allocation passes

Properties of ϕ

- Placed only at the entry of a join node

Properties of ϕ

- Placed only at the entry of a join node
- Multiple ϕ-functions could be placed

Properties of ϕ

- Placed only at the entry of a join node
- Multiple ϕ-functions could be placed
  - for multiple variables

Properties of ϕ

- Placed only at the entry of a join node
- Multiple ϕ-functions could be placed
  - for multiple variables
  - all such ϕ functions execute concurrently

Properties of ϕ

- Placed only at the entry of a join node
- Multiple ϕ-functions could be placed
  - for multiple variables
  - all such ϕ functions execute concurrently
- *n*-ary ϕ function at *n*-way join node

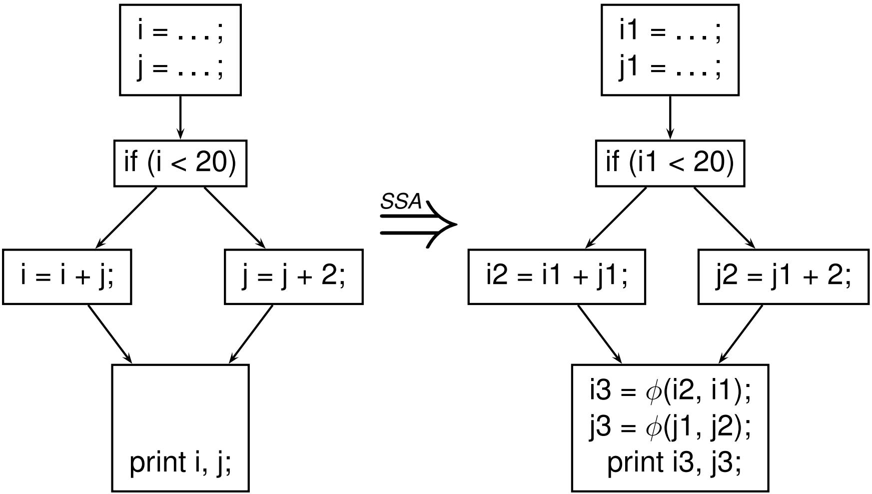
Properties of ϕ

- Placed only at the entry of a join node
- Multiple ϕ-functions could be placed
  - for multiple variables
  - all such ϕ functions execute concurrently
- *n*-ary ϕ function at *n*-way join node
- gets the value of *i*-th argument if control enters through *i*-th edge

Properties of ϕ

- Placed only at the entry of a join node
- Multiple ϕ-functions could be placed
  - for multiple variables
  - all such ϕ functions execute concurrently
- *n*-ary ϕ function at *n*-way join node
- gets the value of *i*-th argument if control enters through *i*-th edge
  - Ordering of ϕ arguments according to the edge ordering is important

SSA Example (revisit)



Construction of SSA Form

Assumptions

- Only scalar variables

Assumptions

- Only scalar variables
  - Structures, pointers, arrays could be handled

Assumptions

- Only scalar variables
  - Structures, pointers, arrays could be handled
  - Refer to publications

Dominators

- Nodes x and y in flow graph

Dominators

- Nodes x and y in flow graph
- x dominates y if **every** path from *Entry* to y goes through x

Dominators

- Nodes x and y in flow graph
- x dominates y if **every** path from *Entry* to y goes through x
  - $x \operatorname{dom} y$

Dominators

- Nodes x and y in flow graph
- x dominates y if **every** path from *Entry* to y goes through x
  - $x \operatorname{dom} y$
  - partial order?

Dominators

- Nodes x and y in flow graph
- x dominates y if **every** path from *Entry* to y goes through x
  - $x \operatorname{dom} y$
  - partial order?
- x strictly dominates y if $x \operatorname{dom} y$ and $x \neq y$

Dominators

- Nodes x and y in flow graph
- x dominates y if **every** path from *Entry* to y goes through x
  - $x \operatorname{dom} y$
  - partial order?
- x strictly dominates y if $x \operatorname{dom} y$ and $x \neq y$
  - $x \operatorname{sdom} y$

Computing Dominators

- Equation

$$\mathrm{DOM}(n) \quad = \quad \{n\} \cup \left(\bigcap_{m \in \mathrm{PRED}(n)} \mathrm{DOM}(m) \right),$$
$$\forall n \in N$$

Computing Dominators

- Equation

$$\text{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in \text{PRED}(n)} \text{DOM}(m) \right), \quad \forall n \in N$$

- Initial Conditions:

$$\begin{aligned} \text{DOM}(n_{Entry}) &= \{n_{Entry}\} \\ \text{DOM}(n) &= N, \forall n \in N - \{n_{Entry}\} \end{aligned}$$

where N is the set of all nodes, n_{Entry} is the node corresponding to the *Entry* block.

Computing Dominators

- Equation

$$\text{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in \text{PRED}(n)} \text{DOM}(m) \right), \quad \forall n \in N$$

- Initial Conditions:

$$\begin{aligned} \text{DOM}(n_{Entry}) &= \{n_{Entry}\} \\ \text{DOM}(n) &= N, \forall n \in N - \{n_{Entry}\} \end{aligned}$$

 where N is the set of all nodes, n_{Entry} is the node corresponding to the *Entry* block.

- Note that efficient methods exist for computing dominators

Immediate Dominators and Dominator Tree

- *x* is immediate dominator of *y* if *x* is the *closest strict dominator* of *y*

Immediate Dominators and Dominator Tree

- x is immediate dominator of y if x is the *closest strict dominator* of y
  - unique, if it exists

Immediate Dominators and Dominator Tree

- x is immediate dominator of y if x is the *closest strict dominator* of y
  - unique, if it exists
  - denoted $\mathrm{idom}[y]$

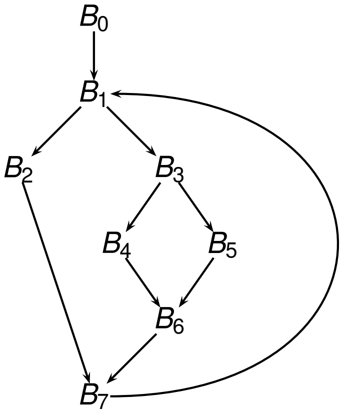
Immediate Dominators and Dominator Tree

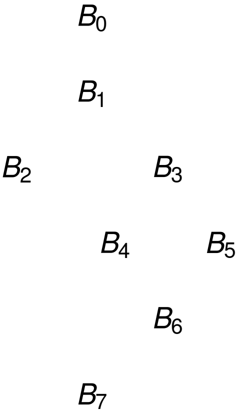
- x is immediate dominator of y if x is the *closest strict dominator* of y
  - unique, if it exists
  - denoted $\mathrm{idom}[y]$
- Dominator Tree

Immediate Dominators and Dominator Tree

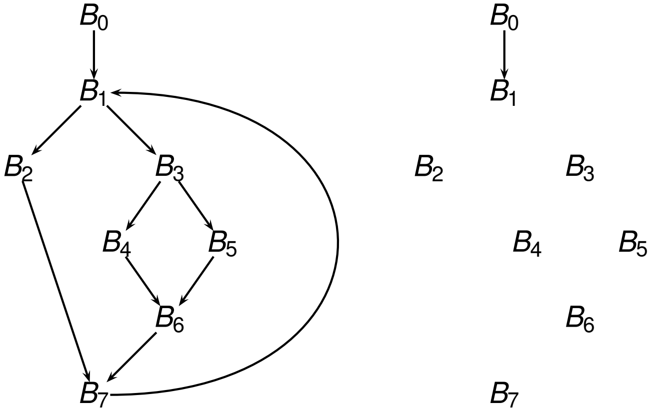
- x is immediate dominator of y if x is the *closest strict dominator* of y
  - unique, if it exists
  - denoted $\mathrm{idom}[y]$
- Dominator Tree
  - A tree showing all immediate dominator relationships

Dominator Tree Example

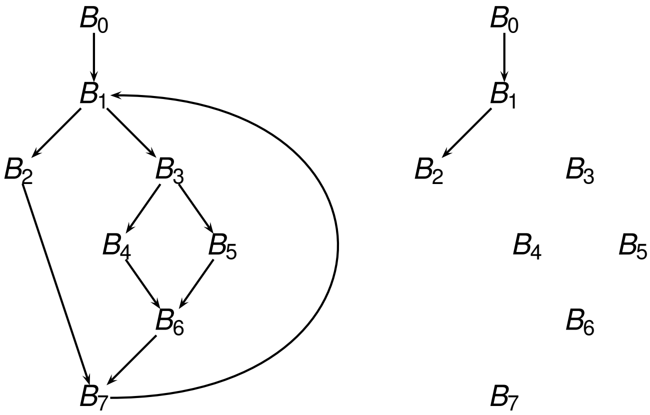




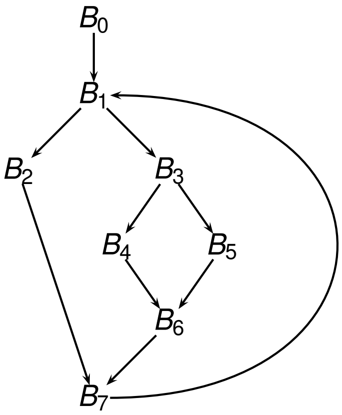
Dominator Tree Example

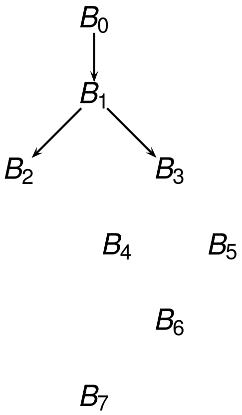


Dominator Tree Example

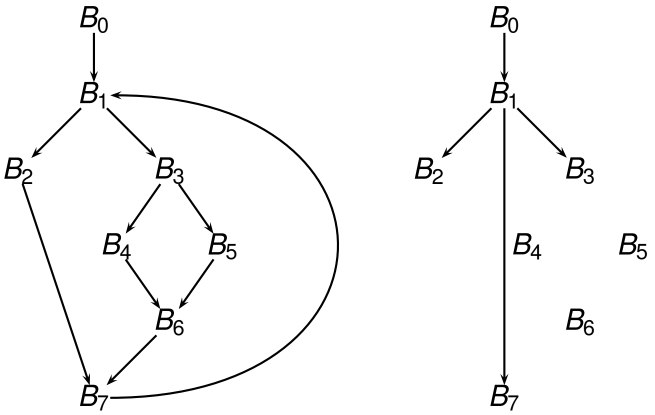


Dominator Tree Example

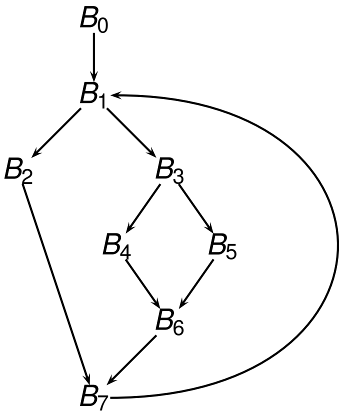


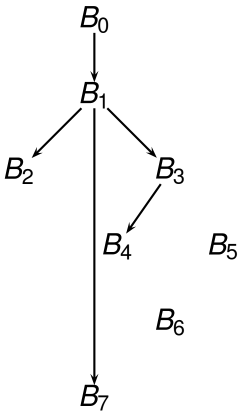


Dominator Tree Example

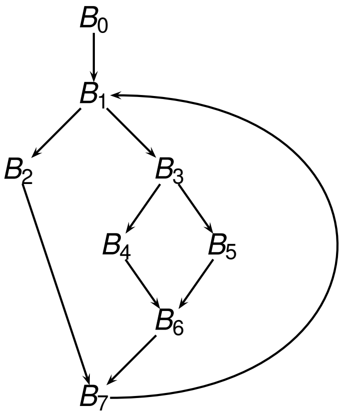


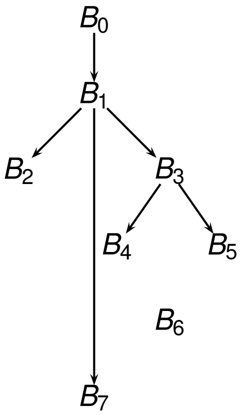
Dominator Tree Example



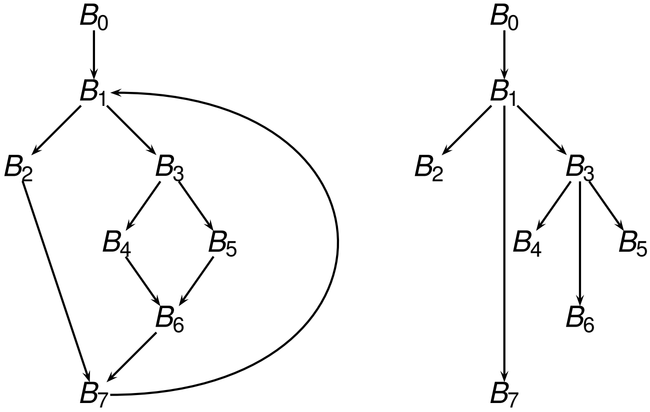


Dominator Tree Example





Dominator Tree Example



Dominance Frontier: DF

- Dominance Frontier of x is set of all nodes y s.t.

Dominance Frontier: DF

- Dominance Frontier of x is set of all nodes y s.t.
  - x dominates a predecessor of y AND

Dominance Frontier: DF

- Dominance Frontier of x is set of all nodes y s.t.
  - x dominates a predecessor of y AND
  - x does not strictly dominate y

Dominance Frontier: DF

- Dominance Frontier of x is set of all nodes y s.t.
  - x dominates a predecessor of y AND
  - x does not strictly dominate y
- Denoted $\mathrm{DF}(x)$

Dominance Frontier: DF

- Dominance Frontier of x is set of all nodes y s.t.
  - x dominates a predecessor of y AND
  - x does not strictly dominate y
- Denoted $\mathrm{DF}(x)$
- Why do you think $\mathrm{DF}(x)$ is important for any x?

Dominance Frontier: DF

- Dominance Frontier of x is set of all nodes y s.t.
  - x dominates a predecessor of y AND
  - x does not strictly dominate y
- Denoted DF(x)
- Why do you think DF(x) is important for any x?
  - Think about the information originated in x.

Computing DF

- PARENT(x) denotes parent of node x in the dominator tree.
- CHILDERN(x) denotes children of node x in the dominator tree.
- PRED and SUCC from CFG.

$$\mathrm{DF}(x) \quad = \quad \mathrm{DF}_{\mathrm{local}}(x) \cup \left(\bigcup_{z \in \mathrm{CHILDERN}(x)} \mathrm{DF}_{\mathrm{up}}(z) \right)$$

Computing DF

- PARENT(x) denotes parent of node x in the dominator tree.
- CHILDERN(x) denotes children of node x in the dominator tree.
- PRED and SUCC from CFG.

$$
\begin{aligned}
\mathsf{DF}(x) &= \mathsf{DF}_{\mathsf{local}}(x) \cup \left(\bigcup_{z \in \mathsf{CHILDERN}(x)} \mathsf{DF}_{\mathsf{up}}(z) \right) \\
\mathsf{DF}_{\mathsf{local}}(x) &= \{ y \in \mathsf{SUCC}(x) \mid \mathsf{idom}[y] \neq x \}
\end{aligned}
$$

Computing DF

- $\text{PARENT}(x)$ denotes parent of node x in the dominator tree.
- $\text{CHILDERN}(x)$ denotes children of node x in the dominator tree.
- PRED and SUCC from CFG.

$$
\begin{aligned}
\text{DF}(x) &= \text{DF}_{\text{local}}(x) \cup \left(\bigcup_{z \in \text{CHILDERN}(x)} \text{DF}_{\text{up}}(z) \right) \\
\text{DF}_{\text{local}}(x) &= \{ y \in \text{SUCC}(x) \mid \text{idom}[y] \neq x \} \\
\text{DF}_{\text{up}}(z) &= \{ y \in \text{DF}(z) \mid \text{idom}[y] \neq \text{PARENT}(z) \}
\end{aligned}
$$

Iterated Dominance Frontier

- Transitive closure of Dominance frontiers on a set of nodes

Iterated Dominance Frontier

- Transitive closure of Dominance frontiers on a set of nodes
- Let S be a set of nodes

$$\mathrm{DF}(S) = \bigcup_{x \in S} \mathrm{DF}(x)$$

Iterated Dominance Frontier

- Transitive closure of Dominance frontiers on a set of nodes
- Let S be a set of nodes

$$\mathrm{DF}(S) = \bigcup_{x \in S} \mathrm{DF}(x)$$

Iterated Dominance Frontier

- Transitive closure of Dominance frontiers on a set of nodes
- Let S be a set of nodes

$$\begin{aligned}
\mathrm{DF}(S) &= \bigcup_{x \in S} \mathrm{DF}(x) \\
\mathrm{DF}^1(S) &= \mathrm{DF}(S) \\
\mathrm{DF}^{i+1}(S) &= \mathrm{DF}(S \cup \mathrm{DF}^i(S))
\end{aligned}$$

- $\mathrm{DF}^+(S)$ is the fixed point of DF^i computation.

Minimal SSA Form Construction

- Compute DF^{+} set for each flow graph node

Minimal SSA Form Construction

- Compute DF^+ set for each flow graph node
- Place trivial ϕ-functions for each variable in the node

Minimal SSA Form Construction

- Compute DF^+ set for each flow graph node
- Place trivial ϕ-functions for each variable in the node
  - trivial ϕ-function at n-ary join: $x = \phi(\overbrace{x, x, \ldots, x}^{n\text{-times}})$

Minimal SSA Form Construction

- Compute DF^+ set for each flow graph node
- Place trivial ϕ-functions for each variable in the node
  - trivial ϕ-function at n-ary join: $x = \phi(\overbrace{x, x, \ldots, x}^{n\text{-times}})$
- Rename variables

Minimal SSA Form Construction

- Compute DF^+ set for each flow graph node
- Place trivial ϕ-functions for each variable in the node
  - trivial ϕ-function at n-ary join: $x = \phi(\overbrace{x, x, \ldots, x}^{n\text{-times}})$
- Rename variables
- Why DF^+? Why not only DF?

Inserting ϕ-functions

```
foreach variable v {
```

Inserting ϕ-functions

```
foreach variable v {
    S = Entry ∪ {B_n | v defined in B_n}
```

$S = Entry \cup \{B_n \mid v \texttt{ defined in } B_n\}$

Inserting ϕ-functions

```
foreach variable v {
    S = Entry ∪ {B_n | v defined in B_n}
    Compute DF+(S)
```

Inserting ϕ-functions

```
foreach variable v {
    S = Entry ∪ {B_n | v defined in B_n}
    Compute DF+(S)
    foreach n in DF+(S) {
```

Inserting ϕ-functions

```
foreach variable v {
    S = Entry ∪ {B_n | v defined in B_n}
    Compute DF+(S)
    foreach n in DF+(S) {
        insert φ-function for v at the start of B_n
    }
}
```

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment ($x = \ldots$) in n

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment $(x = \ldots)$ in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment ($x = \ldots$) in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment $(x = \ldots)$ in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment ($x = \ldots$) in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i
    - $i = i + 1$

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment $(x = \ldots)$ in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i
    - $i = i + 1$
- For the successors of n

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment $(x = \ldots)$ in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i
    - $i = i + 1$
- For the successors of n
  - Rename ϕ operands through SUCC edge index

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment $(x = \ldots)$ in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i
    - $i = i + 1$
- For the successors of n
  - Rename ϕ operands through SUCC edge index
- Recursively rename all child nodes in the dominator tree

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment ($x = \ldots$) in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i
    - $i = i + 1$
- For the successors of n
  - Rename ϕ operands through SUCC edge index
- Recursively rename all child nodes in the dominator tree
- For each assignment ($x = \ldots$) in n

Renaming Variables (Pseudo Code)

- Rename from the *Entry* node recursively
  - For each variable x, maintain a rename stack of $x \mapsto x_{\text{version}}$ mapping
- For node n
  - For each assignment $(x = \ldots)$ in n
    - If non-ϕ assignment, rename any use of x with the Top mapping of x from the rename stack
    - Push the mapping $x \mapsto x_i$ on the rename stack
    - Replace lhs of the assignment by x_i
    - $i = i + 1$
- For the successors of n
  - Rename ϕ operands through SUCC edge index
- Recursively rename all child nodes in the dominator tree
- For each assignment $(x = \ldots)$ in n
  - Pop $x \mapsto \ldots$ from the rename stack