Topic 15: Static Single Assignment

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer
Def-Use Chains, Use-Def Chains

Many optimizations need to find all use-sites of a definition, and/or all def-sites of a use:

- constant propagation needs the site of the unique reaching def
- copy propagation, common subexpression elimination,…

Data structures supporting these lookups:

- **def-use** chain: for each definition $d$ of variable $r$, store the use sites of $r$ that $d$ reaches

- **use-def** chain: for each use site $u$ of variable $r$, store the def-sites of $r$ that reach $u$

$N$ definitions, $M$ uses: $2\times N \times M$ relationships
Use-Def Chains, Def-Use Chains

6: \[ r4 = 10 \]
7: \[ r1 = r1 + r4 \]
8: \[ M[r3] = r1 \]

1: \[ r1 = 5 \]
2: \[ r3 = 1 \]
3: \[ \text{branch } r3 > r1, \ 6: \]
4: \[ r3 = r3 + 1 \]
5: \[ \text{goto } 3: \]

Add the def-use relationships…
Use-Def Chains, Def-Use Chains

And these are just the def-use relationships…
Static Single Assignment (SSA):

- improvement on def-use chains
- each register has only one definition in program
- for each use $u$ of $r$, only one definition of $r$ reaches $u$

\[ r1 = 5 \]
\[ r1 = r1 + 1 \]

\[ r2 = r1 + 1 \]
\[ r3 = r1 - 1 \]

How can this be achieved?
Static Single Assignment (SSA):

- improvement on def-use chains
- each register has only one definition in program
- for each use $u$ of $r$, only one definition of $r$ reaches $u$

\[
\begin{align*}
  r1 &= 5 \\
  r1' &= r1 + 1 \\
  r2 &= r1' + 1 \\
  r3 &= r1' - 1
\end{align*}
\]

Rename variables consistently between defs and uses.
Why SSA?

Static Single Assignment Advantages:

- Dataflow analysis and code optimization made simpler.
  - Variables have only one definition - no ambiguity.
  - Dominator information is encoded in the assignments.
- Less space required to represent def-use chains. For each variable, space is proportional to uses * defs. Distinguishing different defs makes use lists shorter and more precise: less overlap.
- Eliminates unnecessary relationships:

  \[
  \text{for } i = 1 \text{ to } N \text{ do } A[i] = 0 \\
  \text{for } i = 1 \text{ to } M \text{ do } B[i] = 1
  \]

  - No reason why both loops should be forced to use same register to hold index register.
  - SSA renames second i to new register which may lead to better register allocation/optimization.

(Dynamic Single Assignment is also proposed in the literature.)
Conversion to SSA Code

Easy to convert basic blocks into SSA form:

- Each definition modified to define brand-new register, instead of redefining old one.
- Each use of register modified to use most recently defined version.

\[
\begin{align*}
  r1 &= r3 + r4 \\
  r2 &= r1 - 1 \\
  r1 &= r4 + r2 \\
  r2 &= r5 * 4 \\
  r1 &= r1 + r2
\end{align*}
\]
Conversion to SSA Code

Easy to convert basic blocks into SSA form:

- Each definition modified to define brand-new register, instead of redefining old one.
- Each use of register modified to use most recently defined version.

\[
\begin{align*}
    r1 &= r3 + r4 \\
    r2 &= r1 - 1 \\
    r1' &= r4 + r2 \\
    r2' &= r5 \times 4 \\
    r1'' &= r1' + r2'
\end{align*}
\]

Control flow introduces problems.
Conversion to SSA Form

\[ r1 = 5 \]

\[ r2 = r1 + 1 \]

\[ r3 = r2 + 1 \]  \[ r3 = r2 - 1 \]

\[ r4 = r3 \times 4 \]
Use $\phi$ functions.
Conversion to SSA Form

\[
\begin{align*}
\text{r1} &= 5 \\
\text{r2} &= \text{r1} + 1 \\
\text{r3} &= \text{r2} + 1 \\
\text{r3}' &= \text{r2} - 1 \\
\text{r3}'' &= \Phi (\text{r3}, \text{r3}') \\
\text{r4} &= \text{r3}'' \times \text{r1}
\end{align*}
\]

\[
\text{r3}'' = \phi(\text{r3}, \text{r3}'):
\begin{align*}
- \text{r3}'' &= \text{r3} & \text{if control enters from left} \\
- \text{r3}'' &= \text{r3}' & \text{if control enters from right}
\end{align*}
\]
Conversion to SSA Form

- $\phi$-functions enable the use of r3 to be reached by exactly one definition of r3.
- Can implement $\phi$-functions as set of move operations on each incoming edge.

- For analysis & optimization: no implementation necessary: $\Phi$ just used as notation
- Left side of $\Phi$-function constitutes a definition; variables in RHS are uses
- Ordering of argument positions corresponds to (arbitrary) order of incoming control flow arcs, but left implicit (could name positions using the labels of predecessor basic blocks…)

- Elimination of $\Phi$-functions/translation out-of-SSA: insert move instructions; often coalesced during register allocation
- Typically, basic blocks have several $\Phi$-functions – all near the top, with identical ordering of incomings arcs from control flow predecessors
Conversion to SSA Form

Naïve insertion:
add a $\Phi$-function for each register at each node with $\geq 2$ predecessors

\[
r1 = 5
\]
\[
r2 = r1 + 1
\]
\[
r3 = r2 + 1
\]
\[
r3' = r3 = r2 - 1
\]
\[
r3'' = \Phi(r3, r3')
\]
\[
r2' = \Phi(r2, r2)
\]
\[
r1' = \Phi(r1, r1)
\]
\[
r4 = r3 * r1
\]

Can we do better?

Trivial $\Phi$-functions – should clearly be avoided!
Path-Convergence Criterion: Insert a $\phi$-function for a register $r$ at node $z$ of the flow graph if ALL of the following are true:

1. There is a block $x$ containing a definition of $r$.
2. There is a block $y \neq x$ containing a definition of $r$.
3. There is a non-empty path $P_{xz}$ of edges from $x$ to $z$.
4. There is a non-empty path $P_{yz}$ of edges from $y$ to $z$.
5. Paths $P_{xz}$ and $P_{yz}$ do not have any node in common other than $z$.
6. The node $z$ does not appear within both $P_{xz}$ and $P_{yz}$ prior to the end, though it may appear in one or the other. (eg if $y=z$)

Assume CFG entry node contains implicit definition of each register:

- $r$ = actual parameter value
- $r$ = undefined

$\phi$-functions are counted as definitions.
Conversion to SSA Form

Solve path-convergence iteratively:

WHILE (there are nodes $x, y, z$ satisfying conditions 1-6) &&
      ($z$ does not contain a $phi$-function for $r$) DO:
        insert $r = \phi(r, r, \ldots, r)$ (one per predecessor) at node $z$.

- Costly to compute. (3 nested loops, for $x, y, z$)
- Since definitions dominate uses, use domination to simplify computation.
Conversion to SSA Form

Solve path-convergence iteratively:

WHILE (there are nodes x, y, z satisfying conditions 1-6) &&
    (z does not contain a phi-function for r) DO:
    insert $r = \phi(r, r, ..., r)$ (one per predecessor) at node z.

- Costly to compute. (3 nested loops, for x, y, z)
- Since definitions dominate uses, use domination to simplify computation.

Use Dominance Frontier...

Remember dominance: node x dominates node w if every path from entry to w goes through x. (In particular, every node dominates itself)
Definitions:

- $x$ strictly dominates $w$ if $x$ dominates $w$ and $x \neq w$.
- Dominance frontier of node $x$ is set of all nodes $w$ such that $x$ dominates a predecessor of $w$, but does not strictly dominate $w$. 

DF(5) = ?
Definitions:

- *x strictly dominates w* if *x dominates w* and *x ≠ w*.
- *dominance frontier* of node *x* is set of all nodes *w* such that *x* dominates a predecessor of *w*, but does not strictly dominate *w*.

\[ DF(5) = \{4, 5, 10, 11\} \]
Dominance Frontier Criterion:

Whenever node $x$ contains a definition of a register $r$, insert a $\Phi$-function for $r$ in all nodes $z \in DF(x)$.

Iterated Dominance Frontier Criterion:

Apply dominance frontier condition repeatedly, to account for the fact that $\Phi$-functions constitute definitions themselves.

Suppose 5 contains a definition of $r$. 
**Dominance Frontier Criterion:**
Whenever node $x$ contains a definition of a register $r$, insert a $\Phi$-function for $r$ in all nodes $z \in \text{DF}(x)$.

**Iterated Dominance Frontier Criterion:**
Apply dominance frontier condition repeatedly, to account for the fact that $\Phi$-functions constitute definitions themselves.

Suppose 5 contains a definition of $r$. Insert $\Phi$-functions for $r$ in red blocks.

But not here.
Dominance Frontier Computation

- Use dominator tree
- $DF[n]$: dominance frontier of $n$
- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

Alternative formulation: $DF_{local}[n] = \text{successors } s \text{ of } n \text{ with } \text{idom}[s] \neq \text{n}$. 

See errata list of MCIML
Dominance Frontier Computation

- Use dominator tree
- \( DF[n] \): dominance frontier of \( n \)
- \( DF_{local}[n] \): successors of \( n \) in CFG that are not strictly dominated by \( n \)
- \( DF_{up}[c] \): nodes in dominance frontier of \( c \) that are not strictly dominated by \( c \)'s immediate dominator

\[
DF[n] = DF_{local}[n] \cup (\bigcup_{c \in \text{children}[n]} DF_{up}[c])
\]

- where \( \text{children}[n] \) are the nodes whose idom is \( n \).
- Work bottom up in dominator tree.
  Leaf \( p \) satisfies \( DF[p] = DF_{local}[p] \) since \( \text{children}[p] = \emptyset \).

Alternative formulation: \( DF_{local}[n] = \) successors \( s \) of \( n \) with \( \text{idom}[s] \neq n \).
Dominator Analysis (slide 22 from “Control Flow”)

- If \( d \) dominates each of the \( p_i \), then \( d \) dominates \( n \).
- If \( d \) dominates \( n \), then \( d \) dominates each of the \( p_i \).
- \( \text{Dom}[n] = \) set of nodes that dominate node \( n \).
- \( N = \) set of all nodes.
- Computation:
  
  1. \( \text{Dom}[s_0] = \{s_0\} \).
  2. \text{for } n \in N - \{s_0\} \text{ do } \text{Dom}[n] = N
  3. \text{while (changes to any Dom}[n] \text{ occur) do }
  4. \text{for } n \in N - \{s_0\} \text{ do }
  5. \text{Dom}[n] = \{n\} \cup (\cap_{p \in \text{pred}[n]} \text{Dom}[p]) \).

starting point: \( n \) dominated by all nodes

nodes that dominate all predecessors of \( n \)
SSA Example

<table>
<thead>
<tr>
<th>Node</th>
<th>$DOM[n]$</th>
<th>$IDOM[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

set of nodes that dominate n
SSA Example

set of nodes that dominate n

<table>
<thead>
<tr>
<th>Node</th>
<th>$DOM[n]$</th>
<th>$IDOM[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3, 4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1, 2, 3, 4, 5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1, 2, 3, 4, 6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 3, 4, 5, 7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1, 2, 3, 4, 5, 7, 8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1, 2, 3, 4, 5, 9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1, 2, 3, 4, 5, 9, 10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 3, 4, 5, 11</td>
<td></td>
</tr>
</tbody>
</table>
SSA Example

1: \[ r1 = 1 \]
2: \[ r2 = 1 \]
3: \[ r3 = 0 \]
4: \[ \text{branch } r3 < 100 \]
5: \[ \text{branch } r2 < 20 \]
6: \[ \text{return } r2 \]
7: \[ r2 = r1 \]
8: \[ r3 = r3 + 1 \]
9: \[ r2 = r3 \]
10: \[ r3 = r3 + 2 \]

set of nodes that dominate \( n \)

<table>
<thead>
<tr>
<th>Node</th>
<th>( DOM[n] )</th>
<th>( IDOM[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
<td>1, 2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3, 4</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>5</td>
<td>1, 2, 3, 4, 5</td>
<td>1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>6</td>
<td>1, 2, 3, 4, 6</td>
<td>1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 3, 4, 5, 7</td>
<td>1, 2, 3, 4, 5, 7, 8</td>
</tr>
<tr>
<td>8</td>
<td>1, 2, 3, 4, 5, 7, 8</td>
<td>1, 2, 3, 4, 5, 7, 8</td>
</tr>
<tr>
<td>9</td>
<td>1, 2, 3, 4, 5, 9</td>
<td>1, 2, 3, 4, 5, 9</td>
</tr>
<tr>
<td>10</td>
<td>1, 2, 3, 4, 5, 9, 10</td>
<td>1, 2, 3, 4, 5, 9, 10</td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 3, 4, 5, 11</td>
<td>1, 2, 3, 4, 5, 11</td>
</tr>
</tbody>
</table>

- Every node \( n (n \neq s_0) \) has exactly one immediate dominator \( IDom[n] \).
- \( IDom[n] \neq n \)
- \( IDom[n] \) dominates \( n \)
- \( IDom[n] \) does not dominate any other dominator of \( n \).

Hence: last dominator of \( n \) on any path from \( s_0 \) to \( n \) is \( IDom[n] \).
SSA Example

set of nodes that dominate n

<table>
<thead>
<tr>
<th>Node</th>
<th>$DOM[n]$</th>
<th>$IDOM[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3, 4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1, 2, 3, 4, 5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1, 2, 3, 4, 6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 3, 4, 5, 7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>1, 2, 3, 4, 5, 7, 8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1, 2, 3, 4, 5, 9</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>1, 2, 3, 4, 5, 9, 10</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 3, 4, 5, 11</td>
<td>5</td>
</tr>
</tbody>
</table>

- Every node $n$ ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
- $IDom[n] \neq n$
- $IDom[n]$ dominates $n$
- $IDom[n]$ does not dominate any other dominator of $n$. Hence: last dominator of $n$ on any path from $s_0$ to $n$ is $IDom[n]$. 

1: $r1 = 1$
2: $r2 = 1$
3: $r3 = 0$
4: 
branch $r3 < 100$
5: 
branch $r2 < 20$
6: return $r2$
7: $r2 = r1$
8: $r3 = r3 + 1$
9: 
$r2 = r3$
10: 
$r3 = r3 + 2$
11: 
SSA Example

Dominator Tree

set of nodes that dominate n

<table>
<thead>
<tr>
<th>Node</th>
<th>DOM[n]</th>
<th>IDOM[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3, 4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1, 2, 3, 4, 5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1, 2, 3, 4, 6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 3, 4, 5, 7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>1, 2, 3, 4, 5, 7, 8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1, 2, 3, 4, 5, 9</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>1, 2, 3, 4, 5, 9, 10</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 3, 4, 5, 11</td>
<td>5</td>
</tr>
</tbody>
</table>

- Every node \( n \) (\( n \neq s_0 \)) has exactly one immediate dominator \( IDom[n] \).
- \( IDom[n] \neq n \)
- \( IDom[n] \) dominates \( n \)
- \( IDom[n] \) does not dominate any other dominator of \( n \).

Hence, last dominator of \( n \) on any path from \( s_0 \) to \( n \) is \( IDom[n] \).
SSA Example

\[ \text{DF}_{\text{local}}[n] = \text{successors of } n \text{ with } \text{idom}[s] \neq n. \]

1: \( r1 = 1 \)
2: \( r2 = 1 \)
3: \( r3 = 0 \)
4: branch \( r3 < 100 \)
5: branch \( r2 < 20 \)
6: return \( r2 \)
7: \( r2 = r1 \)
8: \( r3 = r3 + 1 \)
9: \( r2 = r3 \)
10: \( r3 = r3 + 2 \)
11: 

<table>
<thead>
<tr>
<th>Node</th>
<th>( \text{IDOM}[n] )</th>
<th>( \text{DF}_{\text{local}}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
**SSA Example**

\[ \text{DF}_{\text{local}}[n] = \text{successors } s \text{ of } n \text{ with idom}[s] \not= n. \]

<table>
<thead>
<tr>
<th>Node</th>
<th>IDOM ([n])</th>
<th>DF_{local}[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
**SSA Example**

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\cup_{c \in \text{children}[n]} DF_{up}[c])$$

- where $\text{children}[n]$ are the nodes whose idom is $n$.

- Work bottom up in dominator tree.

---

<table>
<thead>
<tr>
<th>Node</th>
<th>$IDOM[n]$</th>
<th>$DF_{local}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>$U_{c(n)}$</th>
<th>$DF_{up}[c]$</th>
<th>$DF[n]$</th>
<th>$DF_{up}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>{}</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>{}</td>
<td></td>
<td>{}</td>
</tr>
</tbody>
</table>
**SSA Example**

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\bigcup_{c \in \text{children}[n]} DF_{up}[c])$$

- where $\text{children}[n]$ are the nodes whose idom is $n$.
- Work bottom up in dominator tree.


**Dominator Tree**

<table>
<thead>
<tr>
<th>Node</th>
<th>$IDOM[n]$</th>
<th>$DF_{local}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>$U_c(n)$</th>
<th>$DF[n]$</th>
<th>$DF_{up}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{}</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>{}</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>{}</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>{}</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$'s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\cup_{c \in \text{children}[n]} DF_{up}[c])$$

- where $\text{children}[n]$ are the nodes whose idom is $n$.
- Work bottom up in dominator tree.


<table>
<thead>
<tr>
<th>n</th>
<th>$U_{c(n)}$</th>
<th>DF[n]</th>
<th>DF_up[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{}</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>{}</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>{}</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>{}</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>{}</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Dominator Tree
### SSA Example

- **$DF_{local}[n]$:** successors of $n$ in CFG that are not strictly dominated by $n$
- **$DF_{up}[c]$:** nodes in dominance frontier of $c$ that are not strictly dominated by $c$’s immediate dominator

$$DF[n] = DF_{local}[n] \cup \left( \cup_{c \in children[n]} DF_{up}[c] \right)$$

- Where $children[n]$ are the nodes whose idom is $n$.
- Work bottom up in dominator tree.
  - Leaf $p$ satisfies $DF[p] = DF_{local}[p]$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$U_{c(n)}$</th>
<th>$DF_{up}[c]$</th>
<th>$DF[n]$</th>
<th>$DF_{up}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>{</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>{</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>{</td>
<td>4</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

### Dominator Tree

<table>
<thead>
<tr>
<th>Node</th>
<th>IDOM_[n]</th>
<th>$DF_{local}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

---

### Notes

- $DF$ denotes the set of nodes in the dominance frontier of a node.
- $DF_{local}$ includes successors that are not strictly dominated.
- $DF_{up}$ includes nodes in the dominance frontier of the node’s immediate dominator.
- The table represents the computation of $DF[n]$ for each node.
- The dominator tree is used to traverse the nodes bottom-up to compute $DF[n]$.
SSA Example

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$’s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\cup_{c \in \text{children}[n]} DF_{up}[c])$$

- where $\text{children}[n]$ are the nodes whose idom is $n$.
- Work bottom up in dominator tree.

Leaf $p$ satisfies $DF[p] = DF_{local}[p]$

<table>
<thead>
<tr>
<th>n</th>
<th>$U_{c(n)}$</th>
<th>DF</th>
<th>DF$_{up}$</th>
<th>DF$_{up}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{}</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>{}</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>{}</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>{}</td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Dominator Tree

```
   1
 /|
| 2 -- 3
 /|
| 4   5
|
| 6
|
| 7
| 8
|
| 9
|
| 10
|
| 11
```

---

**Notes:**
- $DF_{local}[n]$ is the set of successors of $n$ that are not strictly dominated by $n$.
- $DF_{up}[c]$ is the set of nodes in the dominance frontier of $c$ that are not strictly dominated by $c$’s immediate dominator.
- $DF[n]$ is the union of $DF_{local}[n]$ and the union of $DF_{up}[c]$ for all $c$ in the children of $n$.
- Work bottom up in the dominator tree, starting from leaf nodes and moving up to the root.

---
**SSA Example**

- $DF_{local}[n]$: successors of $n$ in CFG that are not strictly dominated by $n$
- $DF_{up}[c]$: nodes in dominance frontier of $c$ that are not strictly dominated by $c$’s immediate dominator

$$DF[n] = DF_{local}[n] \cup (\cup_{c \in \text{children}[n]} DF_{up}[c])$$

- where $children[n]$ are the nodes whose idom is $n$.
- Work bottom up in dominator tree.


### Dominator Tree

<table>
<thead>
<tr>
<th>Node</th>
<th>$IDOM[n]$</th>
<th>$DF_{local}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th>$n$</th>
<th>$U_{c(n)}$</th>
<th>$DF_{up}[c]$</th>
<th>$DF[n]$</th>
<th>$DF_{up}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>:</td>
<td>--</td>
<td>--</td>
<td>:</td>
</tr>
<tr>
<td>3</td>
<td>:</td>
<td>--</td>
<td>--</td>
<td>:</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{}</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>{}</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>{}</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>{}</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
**SSA Example**

**Insert \( \phi \)-functions:**

1: \( r1 = 1 \)

2: \( r2 = 1 \)

3: \( r3 = 0 \)

4: branch \( r3 < 100 \)

5: branch \( r2 < 20 \)

6: return \( r2 \)

7: \( r2 = r1 \)

8: \( r3 = r3 + 1 \)

9: \( r2 = r3 \)

10: \( r3 = r3 + 2 \)

11: 

**Dominance Frontier Criterion:**

Whenever node \( x \) contains a **definition** of a register \( r \), insert a \( \Phi \)-function for \( r \) in all nodes \( z \in DF(x) \).

<table>
<thead>
<tr>
<th>n</th>
<th>DF[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{}</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>{}</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>
**SSA Example**

**Insert \( \phi \)-functions:**

1: \( r_1 = 1 \)

2: \( r_2 = 1 \)

3: \( r_3 = 0 \)

4: \( r_2 = \Phi(r_2, r_2) \)
\( r_3 = \Phi(r_3, r_3) \)

(Second round)

branch \( r_3 < 100 \)

5: branch \( r_2 < 20 \)

6: return \( r_2 \)

7: \( r_2 = r_1 \)

8: \( r_3 = r_3 + 1 \)

9: \( r_2 = r_3 \)

10: \( r_3 = r_3 + 2 \)

11: \( r_2 = \Phi(r_2, r_2) \)
\( r_3 = \Phi(r_3, r_3) \)

(First round)

**Dominance Frontier Criterion:**

Whenever node \( x \) contains a definition of a register \( r \), insert a \( \Phi \)-function for \( r \) in all nodes \( z \) \( \in \text{DF}(x) \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{DF}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ }</td>
</tr>
<tr>
<td>2</td>
<td>{ }</td>
</tr>
<tr>
<td>3</td>
<td>{ }</td>
</tr>
<tr>
<td>4</td>
<td>{ }</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>{ }</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>
Rename Variables:

1. traverse dominator tree, renaming different definitions of $r$ to $r_1, r_2, r_3...$
2. rename each regular use of $r$ to most recent definition of $r$
3. rename $\phi$-function arguments with each incoming edge’s unique definition
SSA Example

Rename Variables:

1: \( r1 = 1 \)
2: \( r2 = 1 \)
3: \( r3 = 0 \)
4: \( r2' = \Phi(r2, r2''') \)
   \( r3' = \Phi(r3, r3''') \)
   \( \text{branch } r3' < 100 \)
5: \( \text{branch } r2' < 20 \)
6: \( \text{return } r2' \)
7: \( r2'' = r1 \)
8: \( r3'' = r3' + 1 \)
9: \( r2''' := r3' \)
10: \( r3''' = r3' + 2 \)
11: \( r2''''' = \Phi(r2'', r2''') \)
   \( r3''''' = \Phi(r3'', r3''') \)

Dominator Tree
Alternative construction methods for SSA

Lengauer-Tarjan: efficient computation of dominance tree
- near linear time
- uses depth-first spanning tree
- see MCIML, Section 19.2

- Starts from “crude” placement of $\Phi$-functions: in every block, for every variable
  - then iteratively eliminates unnecessary $\Phi$-functions
  - For reducible CFG

- avoids computation of dominance or iterated DF
- works directly on AST (avoids CFG)
Static Single Assignment

Static Single Assignment Advantages:

- Less space required to represent def-use chains. For each variable, space is proportional to uses * defs.

- Eliminates unnecessary relationships:
  
  \[
  \text{for } i = 1 \text{ to } N \text{ do } A[i] = 0 \\
  \text{for } i = 1 \text{ to } M \text{ do } B[i] = 1
  \]

  – No reason why both loops should be forced to use same register to hold index register.
  – SSA renames second i to new register which may lead to better register allocation.

- SSA form make certain optimizations quick and easy → dominance property.
  – Variables have only one definition - no ambiguity.
  – Dominator information is encoded in the assignments.
Dominance property of SSA form: definitions dominate uses

- If $x$ is $i^{th}$ argument of $\phi$-function in node $n$, then definition of $x$ dominates $i^{th}$ predecessor of $n$.
- If $x$ is used in non-$\phi$ statement in node $n$, then definition of $x$ dominates $n$. 
SSA Dead Code Elimination

Given $d: \mathit{t} = x \ op \ y$

- $\mathit{t}$ is live at end of node $d$ if there exists path from end of $d$ to use of $\mathit{t}$ that does not go through definition of $\mathit{t}$.

- if program not in SSA form, need to perform liveness analysis to determine if $\mathit{t}$ live at end of $d$.

- if program is in SSA form:
SSA Dead Code Elimination

Given $d: \tau = x \; \text{op} \; y$

- $\tau$ is live at end of node $d$ if there exists path from end of $d$ to use of $\tau$ that does not go through definition of $\tau$.
- If program not in SSA form, need to perform liveness analysis to determine if $\tau$ live at end of $d$.
- If program is in SSA form:
  - Cannot be another definition of $\tau$
  - If there exists use of $\tau$, then path from end of $d$ to use exists, since definitions dominate uses.
    * Every use has a unique definition
    * $\tau$ is live at end of node $d$ if $\tau$ is used at least once
SSA Dead Code Elimination

Algorithm:

WHILE (for each temporary \( t \) with no uses \&\&
statement defining \( t \) has no other side-effects) DO
delete statement definition \( t \)

```
a \leftarrow 0
b \leftarrow a + 1
c \leftarrow c + b
a \leftarrow b * 2
if a < N
return c
```

```
a1 \leftarrow 0
a3 \leftarrow \Phi(a1, a2)
b1 \leftarrow \Phi(b0, b2)
c2 \leftarrow \Phi(c0, c1)
b2 \leftarrow a3 + 1
c1 \leftarrow b2
a2 \leftarrow b2 * 2
if a2 < N
return c1
```

```
a1 \leftarrow 0
a3 \leftarrow \Phi(a1, a2)
b1 \leftarrow \Phi(b0, b2)
c2 \leftarrow \Phi(c0, c1)
b2 \leftarrow a3 + 1
c1 \leftarrow c2 + b2
a2 \leftarrow b2 * 2
if a2 < N
return c1
```

Typo in MCIML…
Given $d: \ t = c$, $c$ is constant

Given $u: \ x = t \ op \ b$

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of $t$ in $u$ may be replaced by $c$ if $d$ reaches $u$ and no other definition of $t$ reaches $u$

- if program is in SSA form:
SSA Simple Constant Propagation

Given $d: t = c$, $c$ is constant
Given $u: x = t \ \text{op} \ b$

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of $t$ in $u$ may be replaced by $c$ if $d$ reaches $u$ and no other definition of $t$ reaches $u$

- if program is in SSA form:
  - $d$ reaches $u$, since definitions dominate uses, and no other definition of $t$ exists on path from $d$ to $u$
  - $d$ is only definition of $t$ that reaches $u$, since it is the only definition of $t$.
    * any use of $t$ can be replaced by $c$
    * any $\phi$-function of form $v = \phi(c_1, c_2, \ldots, c_n)$, where $c_i = c$, can be replaced by $v = c$

eliminate branches whose outcome is constant

Similarly: copy propagation, constant folding, constant condition, elimination of unreachable code
SSA Simple Constant Propagation

\[
\begin{align*}
&i_1 \leftarrow 1 \\
&j_1 \leftarrow 1 \\
&k_1 \leftarrow 0 \\
&j_2 \leftarrow \Phi(j_4, j_1) \\
&k_2 \leftarrow \Phi(k_4, k_1) \\
&\text{if } k_2 < 100 \\
&\text{if } j_2 < 20 \\
&j_3 \leftarrow i_1 \\
&k_3 \leftarrow k_2 + 1 \\
&j_4 \leftarrow \Phi(j_3, j_5) \\
&k_4 \leftarrow \Phi(k_3, k_5) \\
&j_5 \leftarrow k_2 \\
&k_5 \leftarrow k_2 + 2 \\
&\text{return } j_2
\end{align*}
\]
SSA Simple Constant Propagation

\[ i_1 \leftarrow 1 \]
\[ j_1 \leftarrow 1 \]
\[ k_1 \leftarrow 0 \]

\[ j_2 \leftarrow \Phi(j_4, j_1) \]
\[ k_2 \leftarrow \Phi(k_4, k_1) \]
if \( k_2 < 100 \)

if \( j_2 < 20 \)
return \( j_2 \)

\[ j_3 \leftarrow i_1 \]
\[ k_3 \leftarrow k_2 + 1 \]

\[ j_4 \leftarrow \Phi(j_3, j_5) \]
\[ k_4 \leftarrow \Phi(k_3, k_5) \]

\[ j_5 \leftarrow k_2 \]
\[ k_5 \leftarrow k_2 + 2 \]

\[ j_2 \leftarrow \Phi(j_4, 1) \]
\[ k_2 \leftarrow \Phi(k_4, 0) \]
if \( k_2 < 100 \)

if \( j_2 < 20 \)
return \( j_2 \)

\[ j_3 \leftarrow 1 \]
\[ k_3 \leftarrow k_2 + 1 \]

\[ j_4 \leftarrow \Phi(j_3, j_5) \]
\[ k_4 \leftarrow \Phi(k_3, k_5) \]

\[ j_5 \leftarrow k_2 \]
\[ k_5 \leftarrow k_2 + 2 \]
SSA Conditional Constant Propagation

1: \[ r1 = 1 \]
2: \[ r2 = 1 \]
3: \[ r3 = 0 \]
4: \[ r2' = \Phi(r2, r2''') \]
   \[ r3' = \Phi(r3, r3''') \]
   branch \[ r3' < 100 \]
5: \[ \text{branch } r2' < 20 \]
6: \[ \text{return } r2' \]
7: \[ r2'' = r1 \]
8: \[ r3''' = r3' + 1 \]
9: \[ r2''' = r3' \]
10: \[ r3'''' = r3' + 2 \]
11: \[ r2''''' = \Phi(r2'', r2''') \]
   \[ r3''''' = \Phi(r3'', r3''') \]

- \( r2 \) always has value of 1
- nodes 9, 10 never executed
- “simple” constant propagation algorithms assumes (through reaching definitions analysis) nodes 9, 10 may be executed.
- cannot optimize use of \( r2 \) in node 5 since definitions 7 and 9 both reach 5.
Much smarter than “simple” constant propagation:

- Does not assume a node can execute until evidence exists that it can be.
- Does not assume register is non-constant unless evidence exists that it is.
Much smarter than “simple” constant propagation:

- Does not assume a node can execute until evidence exists that it can be.
- Does not assume register is non-constant unless evidence exists that it is.

Track run-time value of each register \( r \) using lattice of values:

- \( V[r] = \bot \) (bottom): compiler has seen no evidence that any assignment to \( r \) is ever executed.
- \( V[r] = 4 \): compiler has seen evidence that an assignment \( r = 4 \) is executed, but has seen no evidence that \( r \) is ever assigned to another value.
- \( V[r] = \top \) (top): compiler has seen evidence that \( r \) will have, at various times, two different values, or some value that is not predictable at compile-time.
Much smarter than “simple” constant propagation:

- Does not assume a node can execute until evidence exists that it can be.
- Does not assume register is non-constant unless evidence exists that it is.

Track run-time value of each register $r$ using lattice of values:

- $V[r] = \bot$ (bottom): compiler has seen no evidence that any assignment to $r$ is ever executed.
- $V[r] = 4$: compiler has seen evidence that an assignment $r = 4$ is executed, but has seen no evidence that $r$ is ever assigned to another value.
- $V[r] = \top$ (top): compiler has seen evidence that $r$ will have, at various times, two different values, or some value that is not predictable at compile-time.

Also:

- all registers start at bottom of lattice
- new information can only move registers up in lattice
SSA Conditional Constant Propagation

Track executability of each node in $\mathcal{N}$:

- $E[N] = \text{false}$: compiler has seen no evidence that node $N$ can ever be executed.
- $E[N] = \text{true}$: compiler has seen evidence that node $N$ can be executed.

Initially:

- $V[r] = \perp$, for all registers $r$
- $E[s_0] = \text{true}$, $s_0$ is CFG start node
- $E[N] = \text{false}$, for all CFG nodes $N \neq s_0$
SSA Conditional Constant Propagation

Algorithm: apply following conditions until no more changes occur to $E$ or $V$ values:

1. Given: register $r$ with no definition (formal parameter, uninitialized).
   Action: $V[r] = \top$

2. Given: executable node $B$ with only one successor $C$
   Action: $E[C] = \text{true}$

3. Given: executable assignment $r = x \circ \circ y$, $V[x] = c_1$ and $V[y] = c_2$
   Action: $V[r] = c_1 \circ \circ c_2$  In particular, use this rule for $r = c$.

4. Given: executable assignment $r = x \circ \circ y$, $V[x] = \top$ or $V[y] = \top$
   Action: $V[r] = \top$

5. Given: executable assignment $r = \phi(x_1, x_2, \ldots, x_n)$, $V[x_i] = c_1$, $V[x_j] = c_2$, and predecessors $i$ and $j$ are executable
   Action: $V[r] = \top$

6. Given: executable assignment $r = M[\ldots]$ or $r = f(\ldots)$:
   Action: $V[r] = \top$
7. Given: executable assignment \( r = \Phi (x_1, ..., x_n) \) where \( V[x_i] = T \) for some \( i \) such that the \( i^{th} \) predecessor is executable:
   Action: \( V[r] = T \)

8. Given: executable assignment \( r = \Phi (x_1, ..., x_n) \) where
   -- \( V[x_i] = c_i \) for some \( i \) where the \( i^{th} \) predecessor is executable, and
   -- for each \( j \neq i \), either the \( j^{th} \) predecessor is not executable or \( V[x_j] \in \{ \bot, c_i \} \):
   Action: \( V[r] = c_i \)

9. Given: executable branch \( \text{br } x \text{ bop } y, L1 \text{ (else } L2) \) where \( V[x] = T \) or \( V[y] = T \)
   Action: \( E[L1] = \text{true} \) and \( E[L2] = \text{true} \)

10. Given: executable branch \( \text{br } x \text{ bop } y, L1 \text{ (else } L2) \) where \( V[x] = c_1 \) and \( V[y] = c_2 \)
    Action: \( E[L1] = \text{true} \) or \( E[L2] = \text{true} \) depending on \( c_1 \text{ bop } c_2 \)

Iterate until no update possible.
Given $V$, $E$ values, program can be optimized as follows:

- if $E[B] = \text{false}$, delete node $B$ from CFG.
- if $V[r] = c$, replace each use of $r$ by $c$, delete assignment to $r$. 
SSA Conditional Constant Propagation: example

1: \[ r1 = 1 \]
2: \[ r2 = 1 \]
3: \[ r3 = 0 \]
4: \[ r2' = \Phi(r2, r2''', r3''') \]
   \[ r3' = \Phi(r3, r3''', r3''') \]
   branch \( r3' < 100 \)
5: branch \( r2' < 20 \)
6: return \( r2' \)
7: \[ r2''' = r1 \]
8: \[ r3''' = r3' + 1 \]
9: \[ r2''''' = r3'''' = r3'' \]
10: \[ r3''''' = r3' + 2 \]
11: \[ r2''' = \Phi(r2'', r2''', r3''') \]
   \[ r3''' = \Phi(r3'', r3'''', r3''') \]

<table>
<thead>
<tr>
<th>( N )</th>
<th>( E[N] )</th>
<th>( r )</th>
<th>( V[r] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t</td>
<td>( r1 )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>( r2 )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>( r2' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>( r2'' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>( r2''' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>( r2''''' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td>( r3 )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>( r3' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>( r3'' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>( r3''' )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>( r3''''' )</td>
<td>( \perp )</td>
</tr>
</tbody>
</table>
SSA Conditional Constant Propagation: example

1: \[ r1 = 1 \]
2: \[ r2 = 1 \]
3: \[ r3 = 0 \]
4: \[ r2' = \Phi(r2, r2''\ldots) \]
   \[ r3' = \Phi(r3, r3''\ldots) \]
   branch \( r3' < 100 \)

5: branch \( r2' < 20 \)
6: \[ return r2' \]
7: \[ r2'' = r1 \]
8: \[ r3'' = r3' + 1 \]
9: \[ r2''' = r3' \]
10: \[ r3''' = r3' + 2 \]

11: \[ r2'''' = \Phi(r2'', r2'''\ldots) \]
    \[ r3'''' = \Phi(r3'', r3'''\ldots) \]

<table>
<thead>
<tr>
<th>( N )</th>
<th>( E[N] )</th>
<th>( r )</th>
<th>( V[r] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>( r1 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>( r2 )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>( r2' )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>( r2'' )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>( r2''' )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>( r2'''' )</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>( r3 )</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>( r3' )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>( r3'' )</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>( r3''' )</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>( r3'''' )</td>
<td></td>
</tr>
</tbody>
</table>
SSA Conditional Constant Propagation: example

3: \[ r3 = 0 \]

4: \[ r3' = \Phi(r3, r3''') \]
   branch \( r3' < 100 \)

6: \[ \text{return 1} \]

8: \[ r3''' = r3' + 1 \]

11: \[ r3'''' = \Phi(r3'', r3''') \]

Next: eliminate \( \Phi \)-functions: easy in this case - map all versions of \( r3 \) to \( r3 \)
SSA Conditional Constant Propagation: example

3: \[ r3 = 0 \]

4: branch \( r3 < 100 \)

6: return 1

8: \[ r3 = r3 + 1 \]
Translating out of SSA: elimination of Φ-functions

Intuitive interpretation of Φ-functions suggests insertion of move instructions at the end of immediate control flow predecessors

\[ z \leftarrow \Phi(x_1, x_2, \ldots, x_n) \]
\[ u \leftarrow z \times 2 \]
\[ \ldots \]
Translating out of SSA: elimination of $\Phi$-functions

Intuitive interpretation of $\Phi$-functions suggests insertion of move instructions at the end of immediate control flow predecessors

$z \gets \Phi(x_1, x_2, \ldots, x_n)$
$u \gets z \times 2$

Then rely on register allocator to coalesce / eliminate moves when possible.
Translating out of SSA -- issue 1

\[
\begin{align*}
x_1 & \leftarrow \ldots \\
& \vdots \\
z & \leftarrow \Phi(x_1, x_2) \\
u & \leftarrow z \times 2 \\
& \ldots \\
v & \leftarrow \Phi(y_1, y_2) \\
k & \leftarrow v \times 3 \\
& \ldots \\
y_2 & \leftarrow \ldots \\
& \vdots \\
y_1 & \leftarrow \ldots \\
x_2 & \leftarrow \ldots \\
x_1 & \leftarrow \ldots \\
y_2 & \leftarrow \ldots \\
y_1 & \leftarrow \ldots \\
x_2 & \leftarrow \ldots \\
y_1 & \leftarrow \ldots \\
x_2 & \leftarrow \ldots \\
\end{align*}
\]
Translating out of SSA -- issue I

Move instructions pile up in blocks with multiple successors – they’re not dead.
Translating out of SSA -- issue I

Solution: place move instructions “in the CFG edge”, in a new basic block, whenever predecessor block has several successors.
“Edge-split SSA form”: each CFG edge is either its source block’s only out-edge or its sink block’s only in-edge.
Easy to achieve during SSA construction: add empty blocks.
More motivation for edge splitting: “lost copy” problem

Let's consider the following code snippet:

```plaintext
x ← 1
y ← x
x ← x + 1
if p
  return y

x ← 1
x2 ← Φ(x1, x3)
y ← x2
x3 ← x2 + 1
if p
  return y
```

The diagram illustrates the process of SSA (Single-Assignment form) construction and elimination, as well as copy propagation. This diagram helps in understanding the transformation of the given code from the original to SSA form and back after optimization.
More motivation for edge splitting: “lost copy” problem

x \leftarrow 1

y \leftarrow x
x \leftarrow x + 1
if p

return y

x_1 \leftarrow 1

x_2 \leftarrow \Phi(x_1, x_3)
y \leftarrow x_2
x_3 \leftarrow x_2 + 1
if p

return y

SSA constr.

Copy prop y

SSA elim

More motivation for edge splitting: “lost copy” problem

x_1 \leftarrow 1

x_2 \leftarrow \Phi(x_1, x_3)
y \leftarrow x_2
x_3 \leftarrow x_2 + 1
if p

return x_2

SSA constr.

Copy prop y

SSA elim
More motivation for edge splitting: “lost copy” problem

```plaintext
x ← 1
y ← x
x ← x + 1
if p
return y
```

```plaintext
x_1 ← 1
x_2 ← \Phi(x_1, x_3)
y ← x_2
x_3 ← x_2 + 1
if p
return y
```

```plaintext
x_1 ← 1
x_2 ← \Phi(x_1, x_3)
y ← x_2
x_3 ← x_2 + 1
if p
return x_2
```

More motivation for edge splitting: “lost copy” problem

SSA constr. → Copy prop y → SSA elim
More motivation for edge splitting: “lost copy” problem

Incorrect result: copy propagation + \( \Phi \)-elimination incompatible.
More motivation for edge splitting: “lost copy” problem

Edge split makes copy propagation + Φ-elimination compatible.
More motivation for edge splitting: “lost copy” problem

Root cause: copy propagation (and other transformations) potentially alter liveness ranges, so that the ranges of different SSA-versions $x_i$ of a source-program variable $x$ are not any longer distinct.

After SSA construction, different “versions” $x_i$ of a source-program variable $x$ are “first-class citizens”, unrelated to each other or to $x$. 
Translating out of SSA -- issue II: “swap problem”

SSA constr. + edge split

Copy folding

SSA elim
Translating out of SSA -- issue II: “swap problem”

\[ a \leftarrow \ldots \]
\[ b \leftarrow \ldots \]
\[ x \leftarrow a \]
\[ a \leftarrow b \]
\[ b \leftarrow x \]
\[ \text{if } p \]
\[ \text{return } a-b \]

SSA constr. + edge split

\[ a_1 \leftarrow \ldots \]
\[ b_1 \leftarrow \ldots \]
\[ a_2 \leftarrow \Phi(a_1, a_3) \]
\[ b_2 \leftarrow \Phi(b_1, b_3) \]
\[ x \leftarrow a_2 \]
\[ a_3 \leftarrow b_2 \]
\[ b_3 \leftarrow x \]
\[ \text{if } p \]
\[ \text{return } a_3-b_3 \]

Copy folding

\[ a_1 \leftarrow \ldots \]
\[ b_1 \leftarrow \ldots \]
\[ a_2 \leftarrow \Phi(a_1, b_2) \]
\[ b_2 \leftarrow \Phi(b_1, a_2) \]
\[ \text{if } p \]
\[ \text{return } b_2-a_2 \]

SSA elim
Translating out of SSA -- issue II: “swap problem”

```
Translating out of SSA -- issue II: “swap problem”

a \leftarrow \ldots
b \leftarrow \ldots

x \leftarrow a
a \leftarrow b
b \leftarrow x
if p

\text{return } a-b

\text{a_1} \leftarrow \ldots
\text{b_1} \leftarrow \ldots

\text{a_2} \leftarrow \Phi(\text{a_1}, \text{a_3})
\text{b_2} \leftarrow \Phi(\text{b_1}, \text{b_3})
x \leftarrow \text{a_2}
a_3 \leftarrow \text{b_2}
b_3 \leftarrow x
if p

\text{return } a_3-b_3

\text{a_1} \leftarrow \ldots
\text{b_1} \leftarrow \ldots
\text{a_2} \leftarrow \Phi(\Phi(\text{a_1}, \text{b_2}), \text{b_1})
\text{b_2} \leftarrow \Phi(\text{b_1}, \text{a_2})
if p

\text{return } b_2-a_2

\text{a_1} \leftarrow \ldots
\text{b_1} \leftarrow \ldots
\text{a_2} \leftarrow \text{a_1}
\text{b_2} \leftarrow \text{b_1}

\text{if p}

\text{a_2} \leftarrow b_2
\text{b_2} \leftarrow a_2

\text{return } b_2-a_2

SSA constr.
+ edge split

Copy folding

SSA elim
```
Translating out of SSA -- issue II: “swap problem”

Incorrect result: copy folding + \( \Phi \)-elimination incompatible.

\[ \begin{align*}
    a & \leftarrow \ldots \\
    b & \leftarrow \ldots \\
\end{align*} \]

\[ \begin{align*}
    x & \leftarrow a \\
    a & \leftarrow b \\
    b & \leftarrow x \\
    \text{if } p
\end{align*} \]

\begin{align*}
    a_1 & \leftarrow \ldots \\
    b_1 & \leftarrow \ldots \\
\end{align*}

\[ \begin{align*}
    a_2 & \leftarrow \Phi(a_1, a_3) \\
    b_2 & \leftarrow \Phi(b_1, b_3) \\
    x & \leftarrow a_2 \\
    a_3 & \leftarrow b_2 \\
    b_3 & \leftarrow x \\
    \text{if } p
\end{align*} \]

\[ \begin{align*}
    a & \leftarrow b \\
    b & \leftarrow x \\
    \text{if } p
\end{align*} \]

\begin{align*}
    a_1 & \leftarrow \ldots \\
    b_1 & \leftarrow \ldots \\
\end{align*}

\[ \begin{align*}
    a_2 & \leftarrow \Phi(a_1, b_2) \\
    b_2 & \leftarrow \Phi(b_1, a_2) \\
    \text{if } p
\end{align*} \]

\[ \begin{align*}
    a & \leftarrow b \\
    b & \leftarrow \ldots \\
    \text{if } p
\end{align*} \]

\begin{align*}
    a_1 & \leftarrow \ldots \\
    b_1 & \leftarrow \ldots \\
\end{align*}

\[ \begin{align*}
    a_2 & \leftarrow b_2 \\
    b_2 & \leftarrow a_2 \\
\end{align*} \]

\[ \text{return } a - b \]

\[ \text{return } a_3 - b_3 \]

\[ \text{return } b_2 - a_2 \]

\[ \text{return } b_2 - a_2 \]

\( \text{SSA constr.} \quad \text{+ edge split} \)

\( \text{Copy folding} \)

\( \text{SSA elim} \)

p true: correct result

p false: a and b are identified in first loop iteration, so \( b_2 = a_2 \) holds upon loop exit, so return value is 0.
Translating out of SSA -- issue II: “swap problem”

Root cause: the moves should “execute in parallel”, ie \textbf{first} read their RHS, then assign to the LHS variables in parallel!

\(\Phi\)-functions in a basic block should be considered a single \(\Phi\)-block, of concurrent assignment, so that the relative order of \(\Phi\)-functions is irrelevant:

\[
\begin{align*}
\Phi(a_1, b_2) & \quad \text{if } p \\
\Phi(b_1, a_2) & \\
\text{return } b_2-a_2
\end{align*}
\]

\[
\begin{align*}
(a_2) & \quad \Phi(a_1, b_2) \\
(b_2) & \quad \Phi(b_1, a_2)
\end{align*}
\]
Translating out of SSA -- issue II: “swap problem”

The $\Phi$-functions in a basic block should be considered concurrent – as a single $\Phi$-block:

$$\left( \begin{array}{c} a_2 \\ b_2 \end{array} \right) \leftarrow \Phi \left( \begin{array}{c} a_1, b_2 \\ b_1, a_2 \end{array} \right)$$

And replacement of $\Phi$ by moves should respect this interpretation.

Conceptual intermediate step: unary $\Phi$-blocks at the end of the CFG predecessors / in the incoming CFG edges.

If $p$:

- $a_1 \leftarrow \ldots$
- $b_1 \leftarrow \ldots$

$$\left( \begin{array}{c} a_2 \\ b_2 \end{array} \right) \leftarrow \Phi \left( \begin{array}{c} a_1, b_2 \\ b_1, a_2 \end{array} \right)$$

If $p$:

- $a_1 \leftarrow \ldots$
- $b_1 \leftarrow \ldots$

$$\left( \begin{array}{c} a_2 \\ b_2 \end{array} \right) \leftarrow \Phi \left( \begin{array}{c} a_1 \\ b_1 \end{array} \right)$$

Return $b_2-a_2$
Translating out of SSA -- issue II: “swap problem”

Then, concurrent elimination of unary \( \Phi \)-blocks.

\[
\begin{align*}
    a_1 & \leftarrow \ldots \\
    b_1 & \leftarrow \ldots \\
    (a_2) & \leftarrow \Phi(a_1) \\
    (b_2) & \leftarrow \Phi(b_1)
\end{align*}
\]

\[\text{if } p \]

\[
\begin{align*}
    \text{return } b_2 - a_2
\end{align*}
\]

no problem here

\[
\begin{align*}
    a_1 & \leftarrow \ldots \\
    b_1 & \leftarrow \ldots \\
    a_2 & \leftarrow a_1 \\
    b_2 & \leftarrow b_1
\end{align*}
\]
Translating out of SSA -- issue II: “swap problem”

Then, concurrent elimination of unary Φ-blocks.

Then, concurrent elimination of unary Φ-blocks.

\[
\begin{align*}
&\text{if } p \\
&\text{return } b_2 - a_2
\end{align*}
\]

\[
\begin{align*}
&a_1 \leftarrow \ldots \\
&b_1 \leftarrow \ldots \\
&\begin{pmatrix} a_2 \\ b_2 \end{pmatrix} \leftarrow \Phi \begin{pmatrix} a_1 \\ b_1 \end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
&a_1 \leftarrow \ldots \\
&b_1 \leftarrow \ldots \\
&a_2 \leftarrow a_1 \\
&b_2 \leftarrow b_1
\end{align*}
\]

no problem here

but here, have cyclic dependency

horizontal : left-to-right
Translating out of SSA -- issue II: “swap problem”

Then, concurrent elimination of unary $\Phi$-blocks.

If $p$

\[
\begin{align*}
\text{return } b_2 - a_2 \\
\text{if } p \\
(a_2) &\leftarrow \Phi_{(b_1)}^{(a_1)} \\
(b_2) &\leftarrow \Phi_{(a_2)}^{(b_2)} \\
\end{align*}
\]

Breaking dependence cycle into sequence of move instructions requires an additional variable.

\[
\begin{align*}
&\quad a_1 \leftarrow \ldots \\
&\quad b_1 \leftarrow \ldots \\
&\quad a_2 \leftarrow a_1 \\
&\quad b_2 \leftarrow b_1 \\
\text{no problem here} \\
\text{but here, have cyclic dependency} \\
&\quad k \leftarrow a_2 \\
&\quad a_2 \leftarrow b_2 \\
&\quad b_2 \leftarrow k
\end{align*}
\]
Resulting code has correct behavior, for \( p=\text{true} \) and \( p=\text{false} \).
Translating out of SSA -- issue II: “swap problem”

In general, the variables in a (unary) $\Phi$-block can form multiple (non-overlapping) cycles, of different length.

The cycles can be broken in succession, so the single additional variable/register $k$ can be reused!

New (implicit) sanity condition of SSA: LHS variables should be distinct!

Variables may occur repeatedly in RHS – but only participate in one cycle.

The moves not involved in a cycle (like $e \leftarrow a$) are emitted first.
Translating out of SSA -- discussion

Some care is needed to avoid lost copies and the swap problem, but basic principle – manifest the intuitive meaning of \( \Phi \)-functions by locally inserting copy instructions “in the incoming edges” – works fine.

Alternative: globally identify groups of variables that can be unified
- first guess - the original variables: works fine, until aggressive optimizations yield overlapping liveness ranges etc.

- \( \Phi \)-congruence classes (Sreedhar et al., *Translating out of static single assignment form*. 6th Static Analysis Symposium, LNCS 1694, Springer, 1999)

Insertion of moves, effect on liveness ranges, etc suggest exploration of interaction between SSA and register allocation
Interference graphs of SSA programs are **chordal** graphs.

Any cycle of > 3 vertices has a *chord*, i.e. an edge that is not part of the cycle but connects two of its vertices.

Key properties of chordal graphs:

1. their chromatic number is equal to the size of the largest clique
2. they can be optimally colored in **quadratic** time (w.r.t. number of nodes)

**Also:** the largest clique in the interference graph of an SSA program P is locally manifest in P: there is at least one instruction \( i_p \) where all members of the clique are live.

Can hence traverse program and obtain required number of colors – and know which variables to spill/coalesce in case we don’t have this many registers.

Resulting approach to register allocation:

1. Spill
2. Color
3. Coalesce
4. SSA-destruction

No need for iteration!

Don’t merge nodes in G, but share reg for variables in a \( \Phi \)-node.

In ordinary programs, iteration was needed since spilling/coalescing was not guaranteed to reduce the number of colors needed. For SSA, this is guaranteed, if we spill/coalesce variables live at \( i_p \).
SSA and register allocation: Hack et al.'s result

Remember: interference graph of an SSA program P

- interference graph: G=(V, E) where
  - nodes V: program variables
  - edges E: \((v, w) \in E\) if there is a program point at which \(v\) and \(w\) are both live
- SSA: each use of a variable \(v\) is dominated by the (unique) definition \(D_v\) of \(v\)

Lemma 1: if \(v\) and \(w\) interfere, either \(D_v\) dominates \(D_w\), or \(D_w\) dominates \(D_v\).

Idea: Let \(i\) be the instruction at which \(v\) and \(w\) both live. Thus, there are paths \(i \rightarrow U_v\) and \(i \rightarrow U_w\) to some uses of \(v\) and \(w\). As \(U_v\) is dominated by \(D_v\), there is a path \(D_v \rightarrow i\). Similarly, there is a path from \(D_w\) to \(i\). Hence, entry \(\rightarrow D_v \rightarrow i \rightarrow U_w\) must contain \(D_w\), and entry \(\rightarrow D_w \rightarrow i \rightarrow U_v\) must contain \(D_v\). From this obtain claim…
Lemma 1: if v and w interfere, either $D_v$ dominates $D_w$, or $D_w$ dominates $D_v$.

Lemma 2: if v and w interfere and $D_v$ dominates $D_w$, then v is live at $D_v$. 
Lemma 1: if \( v \) and \( w \) interfere, either \( D_v \) dominates \( D_w \), or \( D_w \) dominates \( D_v \).

Lemma 2: if \( v \) and \( w \) interfere and \( D_v \) dominates \( D_w \), then \( v \) is live at \( D_v \).

Theorem 1: Let \( C = \{c_1, \ldots, c_n\} \) be a clique in \( G \), ie \((c_i, c_j) \in E\) forall \( i \neq j \). Then, there is a label in \( P \) where \( c_1, \ldots, c_n \) are all live.

Proof:

- by Lemma 1, the nodes \( c_1, \ldots, c_n \) are totally ordered by the dominance relationship: \( c_{\sigma(1)}, \ldots, c_{\sigma(n)} \) for some permutation \( \sigma \) of \( \{1, \ldots, n\} \)
- as dominance is transitive, all \( c_{\sigma(i)} \) dominate \( c_{\sigma(n)} \)
- by Lemma 2, all \( c_{\sigma(i)} \) are hence all live at \( c_{\sigma(n)} \).
SSA and register allocation: Hack et al.’s result

• we color nodes by stack-based simplify-select (cf Kempe).
• suppose we can simplify nodes in a **perfect elimination order**: when a node is removed, its remaining neighbors form a clique
• then, when we reinsert the node, we again have a clique
• the size of the latter clique is bound by $\omega(G)$, the size of G’ largest clique
we color nodes by stack-based simplify-select (cf Kempe).

suppose we can simplify nodes in a **perfect elimination order**: when a node is removed, its remaining neighbors form a clique

then, when we reinsert the node, we again have a clique

the size of the latter clique is bound by \( \omega(G) \), the size of \( G' \) largest clique

**Theorem 2**: \( G \) admits simplification by a PEO.

(admitting simplification by PEO is equivalent to being chordal)
SSA and register allocation: Hack et al.’s result

- we color nodes by stack-based simplify-select (cf Kempe).
- suppose we can simplify nodes in a **perfect elimination order**: when a node is removed, its remaining neighbors form a clique
- then, when we reinsert the node, we again have a clique
- the size of the latter clique is bound by $\omega(G)$, the size of G’ largest clique

**Theorem 2**: G admits simplification by a PEO.

(admitting simplification by PEO is equivalent to being chordal)

**Theorem 3**: Chordal graphs are **perfect**: max colors needed = size of the largest clique

Thus, we can color G (using a PEO) using $\omega(G)$ many colors, and P contains an instruction where $\omega(G)$ variables are live (and no instruction with more).

Thus: can traverse P, search for largest **local** live-set, and obtain #registers.
SSA and functional programming

SSA:

• each variable has a unique site of definition; different uses of the same source-program variable name are disambiguated
• the def-site dominates all uses
• in straight-line code, each variable is assigned to only once
SSA and functional programming

SSA:
- each variable has a unique site of definition; different uses of the same source-program variable name are disambiguated
- the def-site dominates all uses
- in straight-line code, each variable is assigned to only once

Functional code:
- each name has a unique site of binding: let x = e₁ in e₂; different uses of the same name are kept apart by the language definition, or can be explicitly disambiguated by α-renaming
- the binding-site determines a scope that contains all uses
- in straight-line code, the value to which a name is bound is never changes
SSA and functional programming

SSA:
• each variable has a unique site of definition; different uses of the same source-program variable name are disambiguated
• the def-site dominates all uses
• in straight-line code, each variable is assigned to only once

Functional code:
• each name has a unique site of binding: let \( x = e_1 \) in \( e_2 \); different uses of the same name are kept apart by the language definition, or can be explicitly disambiguated by \( \alpha \)-renaming
• the binding-site determines a scope that contains all uses
• in straight-line code, the value to which a name is bound never changes – and in a recursive function, we’re in different stack frames (but see details on stack frames in later lecture).

SSA and functional programming - correspondences

<table>
<thead>
<tr>
<th>Functional concept</th>
<th>Imperative/SSA concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable binding in let</td>
<td>assignment (point of definition)</td>
</tr>
<tr>
<td>$\alpha$-renaming</td>
<td>variable renaming</td>
</tr>
<tr>
<td>unique association of binding occurrences to uses</td>
<td>unique association of defs to uses</td>
</tr>
<tr>
<td>formal parameter of continuation/local function</td>
<td>$\phi$-function (point of definition)</td>
</tr>
<tr>
<td>lexical scope of bound variable</td>
<td>dominance region</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional concept</th>
<th>Imperative/SSA concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>subterm relationship</td>
<td>control flow successor relationship</td>
</tr>
<tr>
<td>arity of function $f_i$</td>
<td>number of $\phi$-functions at beginning of $b_i$</td>
</tr>
<tr>
<td>distinctness of formal parameters of $f_i$</td>
<td>distinctness of LHS-variables in the $\phi$-block of $b_i$</td>
</tr>
<tr>
<td>number of call sites of function $f_i$</td>
<td>arity of $\phi$-functions in block $b_i$</td>
</tr>
<tr>
<td>parameter lifting/dropping</td>
<td>addition/removal of $\phi$-function</td>
</tr>
<tr>
<td>block floating/sinking</td>
<td>reordering according to dominator tree structure</td>
</tr>
<tr>
<td>potential nesting structure</td>
<td>dominator tree</td>
</tr>
<tr>
<td>nesting level</td>
<td>maximal level index in dominator tree</td>
</tr>
</tbody>
</table>

- construction of SSA can be recast as transformation of a corresponding functional program; destruction, too
- latent structural properties of SSA often explicit in FP view
- correctness arguments for SSA analyses & transformations transfer to/from functional view
SSA construction in functional style

1:

\[
\begin{align*}
v & \leftarrow 1 \\
z & \leftarrow 8 \\
y & \leftarrow 4
\end{align*}
\]

2:

\[
\begin{align*}
x & \leftarrow 5 + y \\
y & \leftarrow x \times z \\
x & \leftarrow x - 1 \\
& \quad \text{if } x = 0
\end{align*}
\]

Step 1:
convert into functional form

3:

\[
\begin{align*}
w & \leftarrow y + v \\
& \quad \text{return } w
\end{align*}
\]

- one function per basic block
- all functions mutually (tail-)recursive
- entry point: top-level initial function call
- function bodies: let-bindings for basic instructions (ANF)
- liveness analysis yields formal parameter and argument lists
SSA construction in functional style

1: \[ \begin{align*} v & \leftarrow 1 \\ z & \leftarrow 8 \\ y & \leftarrow 4 \end{align*} \]

2: \[ \begin{align*} x & \leftarrow 5 + y \\ y & \leftarrow x \times z \\ x & \leftarrow x - 1 \\ \text{if } x = 0 \end{align*} \]

3: \[ \begin{align*} w & \leftarrow y + v \\ \text{return } w \end{align*} \]

let fun \( f_1() \) = let val \( v = 1 \)
val \( z = 8 \)
val \( y = 4 \)
in \( f_2(v, z, y) \) end
and \( f_2(v, z, y) = \) let val \( x = 5 + y \)
val \( y = x \times z \)
val \( x = x - 1 \)
in if \( x = 0 \) then \( f_3(y, v) \)
else \( f_2(v, z, y) \) end
and \( f_3(y, v) = \) let val \( w = y + v \)
in \( w \) end
in \( f_1() \) end;

- one function per basic block
- all functions mutually (tail-)recursive
- entry point: top-level initial function call
- function bodies: let-bindings for basic instructions (ANF)
- liveness analysis yields formal parameter and argument lists
SSA construction in functional style

1: $v \leftarrow 1$
   $z \leftarrow 8$
   $y \leftarrow 4$

2: $x \leftarrow 5 + y$
   $y \leftarrow x \times z$
   $x \leftarrow x - 1$
   if $x = 0$ then
   
   \[ \text{Step 1} \]
   
   convert into functional form

3: $w \leftarrow y + v$
   return $w$

- all functions \textit{closed}
- variables not globally unique, but uses have unique defs (scope)

let fun $f_1() = let\ val\ v = 1$
        val z = 8
        val y = 4
        in $f_2(v, z, y)$ end
and $f_2(v, z, y) = let\ val\ x = 5 + y$
        val y = x \times z
        val x = x - 1
        in if $x=0$ then $f_3(y, v)$
        else $f_2(v, z, y)$ end
and $f_3(y, v) = let\ val\ w = y + v$
        in $w$ end
in $f_1()$ end;

- one function per basic block
- all functions mutually (tail-)recursive
- entry point: top-level initial function call
- function bodies: let-bindings for basic instructions (ANF)
- liveness analysis yields formal parameter and argument lists
SSA construction in functional style

let fun \( f_1() \) = let val \( v = 1 \)
    val \( z = 8 \)
    val \( y = 4 \)
    in \( f_2(v, z, y) \) end
and \( f_2(v, z, y) \) = let val \( x = 5 + y \)
    val \( y = x * z \)
    val \( x = x - 1 \)
    in if \( x=0 \) then \( f_3(y, v) \)
      else \( f_2(v, z, y) \) end
and \( f_3(y, v) \) = let val \( w = y + v \)
    in \( w \) end
in \( f_1() \) end;

• as functions are closed, can rename each function definition individually
let fun f₁() = let val v₁ = 1
    val z₁ = 8
    val y₁ = 4
in f₂(v₁, z₁, y₁) end
and f₂(v, z, y) = let val x = 5 + y
    val y = x * z
    val x = x – 1
in if x=0 then f₃(y, v)
    else f₂(v, z, y) end
and f₃(y, v) = let val w = y + v
in w end
in f₁() end;

• as functions are closed, can rename each function definition individually
SSA construction in functional style

1:
\[
\begin{align*}
v_1 & \leftarrow 1 \\
z_1 & \leftarrow 8 \\
y_1 & \leftarrow 4
\end{align*}
\]

2:
\[
\begin{align*}
v_2 & \leftarrow \Phi(v_1, v_2) \\
z_2 & \leftarrow \Phi(z_1, z_2) \\
y_2 & \leftarrow \Phi(y_1, y_3) \\
x_1 & \leftarrow 5 + y_2 \\
y_3 & \leftarrow x_1 \ast z_2 \\
x_2 & \leftarrow x_1 - 1 \\
\text{if } x_2 = 0 \text{ then } f_3(y_3, v_2) \text{ end}
\end{align*}
\]

\[
\text{interpret back in imperative form}
\]

3:
\[
\begin{align*}
y_4 & \leftarrow \Phi(y_3) \\
v_3 & \leftarrow \Phi(v_2) \\
w_1 & \leftarrow y_4 + v_3 \\
\text{return } w_1
\end{align*}
\]

let fun f_1() = let val v_1 = 1  \\
val z_1 = 8  \\
val y_1 = 4  \\
in f_2(v_1, z_1, y_1) end  \\
and f_2(v_2, z_2, y_2) = let val x_1 = 5 + y_2  \\
val y_3 = x_1 \ast z_2  \\
val x_2 = x_1 - 1  \\
in if x_2 = 0 then f_3(y_3, v_2) \text{ end}  \\
else f_2(v_2, z_2, y_3) end  \\
and f_3(y_4, v_3) = let val w_1 = y_4 + v_3  \\
in w_1 end  \\
in f_1() \text{ end;}

- each formal parameter of a function definition is the LHS of a Φ-function. Arguments are the function arguments at calls
- arity of functions, distinctness of LHS variables etc all ok
- resulting code “pruned SSA”
- which functional prog avoids the unnecessary Φ-functions?

“unnecessary”: all call sites provide identical arguments
Removing unnecessary arguments: \(\lambda\)-dropping

- transformation of functional programs to eliminate formal parameters
- can be performed before or after names are made unique - former option more instructive
- (inverse operation: \(\lambda\)-lifting)
- 2 phases: block sinking and parameter dropping

remove parameters
modify nesting structure of function definitions
Removing unnecessary arguments: block sinking

Observation: if
- all calls to $g$ are in body of $f$ (or $g$), and
- $g$ is closed (all free variables of body are parameters)

then the definition of $g$ can be moved inside the definition of $f$

let fun ...
and $f(...)$ = let ... in $g(...)$ end
and $g(...)$ = let ...in
  if ... then $g(...)$ else $h(...)$ end
and $h(...)$ = ...(*no call to $g$*)
in ... end;

Note: $g$ is allowed to
- make recursive calls
- make calls to “host function” $f$
- make calls to other functions, like $h$

Placing $g$ near the end of $f$’s body is advantageous for next step…
Block sinking: example

let fun \( f_1() = \) let val \( v = 1 \)
val \( z = 8 \)
val \( y = 4 \)
in \( f_2(v, z, y) \) end 

and \( f_2(v, z, y) = \) let val \( x = 5 + y \)
val \( y = x * z \)
val \( x = x - 1 \)
in if \( x = 0 \) then \( f_3(y, v) \)
else \( f_2(v, z, y) \) end 

and \( f_3(y, v) = \) let val \( w = y + v \)
in \( w \) end 

in \( f_1() \) end;

(in fact, insert \( f_3 \) “in the edge” ie only in the then-branch – cf edge split form)
Block sinking makes dominance structure explicit: $f_2 = \text{idom}(f_3)$, and $f_1 = \text{idom}(f_2)$
let fun $f_1() = \begin{cases} \text{let val } v = 1 \\ \text{val } z = 8 \\ \text{val } y = 4 \\ \text{in let fun } f_2(v, z, y) = \\ \quad \text{let val } x = 5 + y \\ \quad \text{val } y = x * z \\ \quad \text{val } x = x - 1 \\ \quad \text{in if } x=0 \\ \quad \text{then let fun } f_3(y, v) = \\ \quad \quad \text{let val } w = y + v \\ \quad \quad \text{in } w \text{ end} \\ \quad \text{in } f_3(y, v) \text{ end} \\ \quad \text{else } f_2(v, z, y) \text{ end} \\ \text{in } f_2(v, z, y) \text{ end} \end{cases}$ in \quad f_1() \text{ end;}

Parameters $y$ and $v$ of $f_3$:

tightest scope for $y$ (ie the def of) surrounding the call to $f_3$ is also the tightest scope surrounding the function definition $f_3$. Can hence remove parameter $y$ – and similarly parameter $v$. 
let fun f₁() = let val v = 1
    val z = 8
    val y = 4
    in let fun f₂(v, z, y) = 
        let val x = 5 + y
            val y = x * z
            val x = x – 1
        in if x=0
            then let fun f₃() = …
                in f₃() end
            else f₂(v, z, y) end
        in f₂(v, z, y) end
    in f₂(v, z, y) end

Similarly, the external call to f₂ from within the body of f₁ would allow to remove all three parameters from f₂.
let fun f₁() = let val v = 1
    val z = 8
    val y = 4
    in let fun f₂(v, z, y) =
        let val x = 5 + y
            val y = x * z
            val x = x – 1
        in if x = 0
            then let fun f₃() = …
                in f₃() end
            else f₂(v, z, y) end
        in f₂(v, z, y) end
    in f₁() end;

Similarly, the external call to f₂ from within the body of f₁ would allow to remove all three parameters from f₂.

Recursive call of f₂:
• **admits** the removal of parameters v and z, since thedefs associated with the uses at the call site are thedefs in the formal parameter list
• does not **admit** the removal of parameters y, since the def associated with the use of y at the call site is **not** the def in the formal parameter list
Parameter dropping IV

let fun $f_1() = \begin{array}{l}
  \text{let val } v = 1 \\
  \text{val } z = 8 \\
  \text{val } y = 4 \\
  \text{in let fun } f_2(y) = \\
  \hspace{1em} \text{let } x = 5 + y \\
  \hspace{1em} \text{val } y = x \times z \\
  \hspace{1em} \text{val } x = x - 1 \\
  \hspace{1em} \text{in if } x = 0 \hspace{1em} \text{then let fun } f_3() = \\
  \hspace{2em} \text{let } w = y + v \\
  \hspace{2em} \text{in } w \text{ end} \\
  \hspace{1em} \text{else } f_2(y) \text{ end} \\
  \text{in } f_2(y) \text{ end} \\
  \text{end} \\
\end{array} ;$

make names distinct

read as SSA program

Superfluous $\Phi$-functions avoided.
SSA discipline shares many properties with tail-recursive, first-order fragment of functional languages

- transfer of analysis/optimization algorithms
- suitable intermediate format for compiling functional and imperative languages

- function calls not in tail position: calls to imperative functions/methods/procedures
- alternative functional representation of control flow: continuations