

# *COS320: Compiling Techniques*

Zak Kincaid

April 16, 2020

*Static Single Assignment form*

## SSA

- Each %uid appears on the left-hand-side of at most one assignment in a CFG

<pre>if (x &lt; 0) {     y := y - x; } else {     y := y + x; } return y</pre>	$\rightarrow$	<pre>if (x<sub>0</sub> &lt; 0) {     y<sub>1</sub> := y<sub>0</sub> - x<sub>0</sub>; } else {     y<sub>2</sub> := y<sub>0</sub> + x<sub>0</sub>; } y<sub>3</sub> := <math>\phi</math>(y<sub>1</sub>, y<sub>2</sub>) return y<sub>3</sub></pre>
--	---------------	---

- Recall:  $y_3 := \phi(y_1, y_2)$  picks either  $y_1$  or  $y_2$  (whichever one corresponds to the branch that is actually taken) and stores it in  $y_3$
- Well-formedness condition: uids must be defined before they are used.

# Register allocation

- SSA form reduces register pressure
  - Each variable  $x$  is replaced by potentially many “subscripted” variables  $x_1, x_2, x_3, \dots$ 
    - (At least) one for each definition of  $x$
  - Each  $x_i$  can potentially be stored in a different memory location

# Register allocation

- SSA form reduces register pressure
  - Each variable  $x$  is replaced by potentially many “subscripted” variables  $x_1, x_2, x_3, \dots$ 
    - (At least) one for each definition of  $x$
  - Each  $x_i$  can potentially be stored in a different memory location
- Interference graphs for SSA programs are *chordal* (every cycle contains a chord)
  - Chordal graphs can be colored optimally in polytime
  - (But optimal translation out of SSA form is intractable)

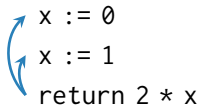
## Dead assignment elimination

Simple algorithm for eliminating assignment<sup>1</sup> instructions that are never used:

**while some  $\%x$  has no uses do**

    | Remove definition of  $\%x$  from CFG;

- SSA conversion  $\Rightarrow$  more assignments are eliminated



```
x := 0
x := 1
return 2 * x
```

---

<sup>1</sup>does *not* eliminate dead *stores*

## Dead assignment elimination

Simple algorithm for eliminating assignment<sup>1</sup> instructions that are never used:

**while some  $\%x$  has no uses do**

    | Remove definition of  $\%x$  from CFG;

- SSA conversion  $\Rightarrow$  more assignments are eliminated



---

<sup>1</sup>does *not* eliminate dead *stores*

## Dead assignment elimination

Simple algorithm for eliminating assignment<sup>1</sup> instructions that are never used:

**while some  $\%x$  has no uses do**

    | Remove definition of  $\%x$  from CFG;

- SSA conversion  $\Rightarrow$  more assignments are eliminated



---

<sup>1</sup>does *not* eliminate dead *stores*



## Dead assignment elimination

Simple algorithm for eliminating assignment<sup>1</sup> instructions that are never used:

**while some  $\%x$  has no uses do**

    | Remove definition of  $\%x$  from CFG;

- SSA conversion  $\Rightarrow$  more assignments are eliminated



---

<sup>1</sup>does *not* eliminate dead *stores*

## Recall: constant propagation

- The goal of constant propagation: determine at each instruction  $I$  a *constant environment*
  - A **constant environment** is a symbol table mapping each variable  $x$  to one of:
    - an integer  $n$  (indicating that  $x$ 's value is  $n$  whenever the program is at  $I$ )
    - $\top$  (indicating that  $x$  might take more than one value at  $I$ )
    - $\perp$  (indicating that  $x$  may take no values at run-time –  $I$  is unreachable)
- Say that the assignment **IN**, **OUT** is **conservative** if

① **IN** $[s]$  assigns each variable  $\top$

② For each node  $bb \in N$ ,

$$\mathbf{OUT}[bb] \sqsupseteq \text{post}_{CP}(bb, \mathbf{IN}[bb])$$

③ For each edge  $src \rightarrow dst \in E$ ,

$$\mathbf{IN}[dst] \sqsupseteq \mathbf{OUT}[src]$$

## (Dense) constant propagation performance

- Memory requirements:  $O(|N| \cdot |Var|)$ 
  - Constant environment has size  $O(|Var|)$ , need to track  $O(1)$  per node
- Time requirements:  $O(|N| \cdot |Var|)$ 
  - Processing a single node takes  $O(1)$  time
  - Each node is processed  $O(|Var|)$  times
    - Height of the abstract domain (length of longest strictly ascending sequence):  $3|Var|$
- Can we do better?

## Sparse constant propagation

- Idea: SSA connects variable *definitions* directly to their *uses*
  - Don't need to store the value of *every* variable at *every* program point
  - Don't need to propagate changes through irrelevant blocks

## Sparse constant propagation

- Idea: SSA connects variable *definitions* directly to their *uses*
  - Don't need to store the value of *every* variable at *every* program point
  - Don't need to propagate changes through irrelevant blocks
- Can think of SSA as a graph, where edges correspond to *data flow* rather than *control flow*
  - Define  $rhs(\%x)$  to be the right hand side of the **unique** assignment to  $\%x$
  - Define  $succ(\%x) = \{\%y : rhs(\%y) \text{ reads } \%x\}$

## Sparse constant propagation

- Idea: SSA connects variable *definitions* directly to their *uses*
  - Don't need to store the value of *every* variable at *every* program point
  - Don't need to propagate changes through irrelevant blocks
- Can think of SSA as a graph, where edges correspond to *data flow* rather than *control flow*
  - Define  $rhs(\%x)$  to be the right hand side of the **unique** assignment to  $\%x$
  - Define  $succ(\%x) = \{\%y : rhs(\%y) \text{ reads } \%x\}$
- Local specification for constant propagation:
  - $scp$  is the smallest function  $Uid \rightarrow \mathbb{Z} \cup \{\top, \perp\}$  such that
    - If  $G$  contains no assignments to  $\%x$ , then  $scp(\%x) = \top$
    - For each instruction  $\%x = e$ ,  $scp(\%x) = eval(e, scp)$

## Worklist algorithm

$$scp(\%x) = \begin{cases} \perp & \text{if } \%x \text{ has an assignment} \\ \top & \text{otherwise} \end{cases}$$

$work \leftarrow \{\%x \in Uid : \%x \text{ is defined}\};$

**while**  $work \neq \emptyset$  **do**

    Pick some  $\%x$  from  $work$ ;

$work \leftarrow work \setminus \{\%x\}$  ;

**if**  $rhs(\%x) = \phi(\%y, \%z)$  **then**

        |  $v \leftarrow scp(\%y) \sqcup scp(\%z)$

**else**

        |  $v \leftarrow eval(rhs(\%x), scp)$

**if**  $v \neq scp(\%x)$  **then**

        |  $scp(\%x) \leftarrow v$ ;

        |  $work \leftarrow work \cup succ(\%x)$

## Computational complexity of constant propagation

	Dense	Sparse
Memory	$O( N  \cdot  Var )$	$O( N ) = O( Var )$
Time	$O( N  \cdot  Var )$	$O( N ) = O( Var )$

- However, observe that we only find constants for uids, not stack slots.
  - Again, advantageous to use uids to represent variable whenever possible



## *Computing SSA*

## (High-level) SSA conversion

- Replace each definition  $x = e$  with a  $x_i = e$  for some unique subscript  $i$
- Replace each *use* of a variable  $y$  with  $y_i$ , where the  $i$ th definition of  $y$  is the unique reaching definition

## (High-level) SSA conversion

- Replace each definition  $x = e$  with a  $x_i = e$  for some unique subscript  $i$
- Replace each use of a variable  $y$  with  $y_i$ , where the  $i$ th definition of  $y$  is the unique reaching definition
- If multiple definitions reach a single use, then they must be merged using a  $\phi$  (phi) statement

```
y := 0;
while (x >= 0) {
  x := x - 1;
  y := y + x;
}
return y
```

→

```
y0 := 0;
while (true) {
  x2 =  $\phi$ (x0, x1)
  y2 =  $\phi$ (y0, y1)
  if (x2 < 0) break;
  x1 := x2 - 1;
  y1 := y2 + x1;
}
return y2
```

## Placing $\phi$ statements

- Easy, inefficient solution: place a  $\phi$  statement for each variable location at each *join point*
  - A *join point* is a node in the CFG with more than one predecessor

---

<sup>2</sup>The entry node of the CFG is considered to be an implicit definition of every variable

## Placing $\phi$ statements

- Easy, inefficient solution: place a  $\phi$  statement for each variable location at each *join point*
  - A *join point* is a node in the CFG with more than one predecessor
- Better solution: place a  $\phi$  statement for variable  $x$  at location  $n$  exactly when the following **path convergence criterion** holds: there exist a pair of non-empty paths  $P_1, P_2$  ending at  $n$  such that
  - 1 The start node of both  $P_1$  and  $P_2$  defines  $x$ <sup>2</sup>
  - 2 The only node shared by  $P_1$  and  $P_2$  is  $n$

---

<sup>2</sup>The entry node of the CFG is considered to be an implicit definition of every variable

## Placing $\phi$ statements

- Easy, inefficient solution: place a  $\phi$  statement for each variable location at each *join point*
  - A *join point* is a node in the CFG with more than one predecessor
- Better solution: place a  $\phi$  statement for variable  $x$  at location  $n$  exactly when the following **path convergence criterion** holds: there exist a pair of non-empty paths  $P_1, P_2$  ending at  $n$  such that
  - 1 The start node of both  $P_1$  and  $P_2$  defines  $x$ <sup>2</sup>
  - 2 The only node shared by  $P_1$  and  $P_2$  is  $n$
- The path convergence criterion can be implemented using the concept of *iterated dominance frontiers*

---

<sup>2</sup>The entry node of the CFG is considered to be an implicit definition of every variable

# Dominance

- Let  $G = (N, E, s)$  be a control flow graph
- We say that a node  $d \in N$  **dominates** a node  $n \in N$  if every path from  $s$  to  $n$  contains  $d$ 
  - Every node dominates itself
  - $d$  **strictly dominates**  $n$  if  $d$  is not  $n$
  - $d$  **immediately dominates**  $n$  if  $d$  strictly dominates  $n$  and but does not strictly dominate any strict dominator of  $n$ .

# Dominance

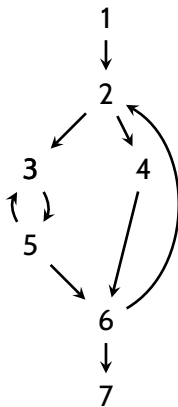
- Let  $G = (N, E, s)$  be a control flow graph
- We say that a node  $d \in N$  **dominates** a node  $n \in N$  if every path from  $s$  to  $n$  contains  $d$ 
  - Every node dominates itself
  - $d$  **strictly dominates**  $n$  if  $d$  is not  $n$
  - $d$  **immediately dominates**  $n$  if  $d$  strictly dominates  $n$  and but does not strictly dominate any strict dominator of  $n$ .
- Observe: dominance is a partial order on  $N$ 
  - Every node dominates itself (reflexive)
  - If  $n_1$  dominates  $n_2$  and  $n_2$  dominates  $n_3$  then  $n_1$  dominates  $n_3$  (transitive)
  - If  $n_1$  dominates  $n_2$  and  $n_2$  dominates  $n_1$  then  $n_1$  must be  $n_2$  (anti-symmetric)



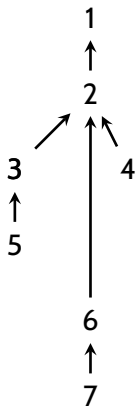
If we draw an edge from every node to its immediate dominator, we get a data structure called the *dominator tree*.

- (Essentially the Hasse diagram of the dominated-by order)

Control Flow Graph



Dominator tree



# Dominance and SSA

- SSA well-formedness criteria

- If  $\%x$  is the  $i$ th argument of a  $\phi$  function in a block  $n$ , then the definition of  $\%x$  must dominate the  $i$ th predecessor of  $n$ .
- If  $\%x$  is used in a non- $\phi$  statement in block  $n$ , then the definition of  $\%x$  must dominate  $n$

## Dominator analysis

- Let  $G = (N, E, s)$  be a control flow graph.
- Define  $dom$  to be a function mapping each node  $n \in N$  to the set  $dom(n) \subseteq N$  of nodes that dominate it

## Dominator analysis

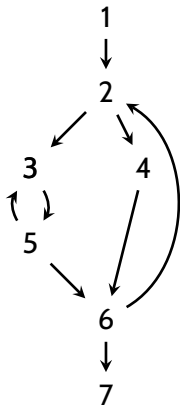
- Let  $G = (N, E, s)$  be a control flow graph.
- Define  $dom$  to be a function mapping each node  $n \in N$  to the set  $dom(n) \subseteq N$  of nodes that dominate it
- *Local specification:*  $dom$  is the largest (equiv. least in superset order) function such that
  - $dom(s) = \{s\}$
  - For each  $p \rightarrow n \in E$ ,  $dom(n) \subseteq \{n\} \cup dom(p)$

## Dominator analysis

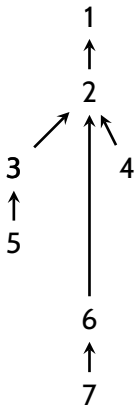
- Let  $G = (N, E, s)$  be a control flow graph.
- Define  $dom$  to be a function mapping each node  $n \in N$  to the set  $dom(n) \subseteq N$  of nodes that dominate it
- *Local specification*:  $dom$  is the largest (equiv. least in superset order) function such that
  - $dom(s) = \{s\}$
  - For each  $p \rightarrow n \in E$ ,  $dom(n) \subseteq \{n\} \cup dom(p)$
- Can be solved using dataflow analysis techniques
  - In practice: nearly linear time algorithm due to Lengauer & Tarjan

- The *dominance frontier* of a node  $n$  is the set of all nodes  $m$  such that  $n$  dominates a predecessor of  $m$ , but does not strictly dominate  $m$  itself.
  - $DF(n) = \{m : (\exists p \in Pred(m). n \in dom(p)) \wedge (m = n \vee n \notin dom(m))\}$
- Whenever a node  $n$  contains a definition of some uid  $\%_0x$ , then any node  $m$  in the dominance frontier of  $n$  needs a  $\phi$  function for  $\%_0x$ .

Control Flow Graph

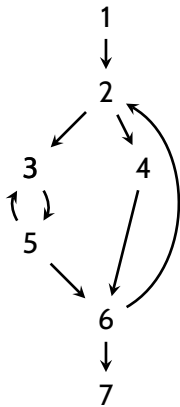


Dominator tree

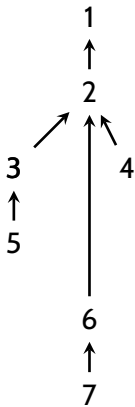


- $DF(1) = \emptyset$

Control Flow Graph



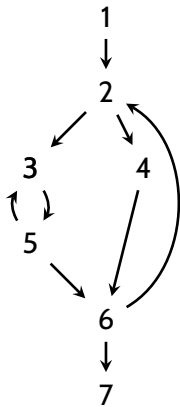
Dominator tree



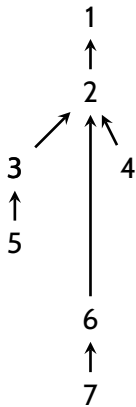
- $DF(1) = \emptyset$
- $DF(2) = \{2\}$



Control Flow Graph

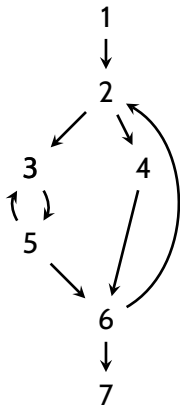


Dominator tree



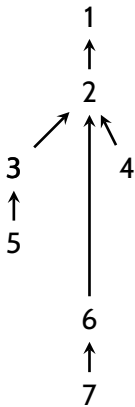
- $DF(1) = \emptyset$
- $DF(2) = \{2\}$
- $DF(3) = \{3, 6\}$

Control Flow Graph



- $DF(1) = \emptyset$
- $DF(2) = \{2\}$
- $DF(3) = \{3, 6\}$

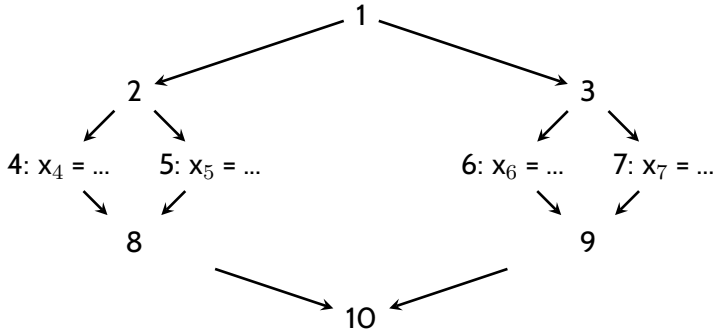
Dominator tree



- $DF(4) = \{6\}$
- $DF(5) = \{3, 6\}$
- $DF(6) = \{2\}$

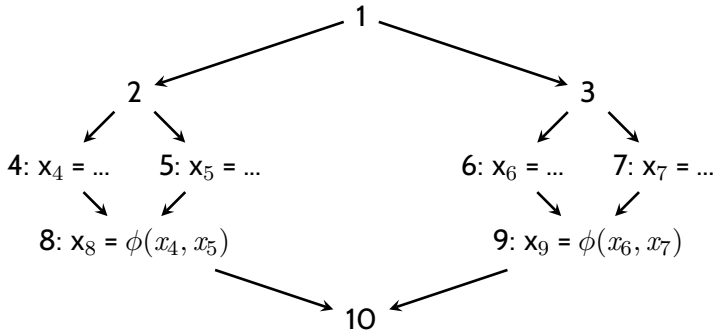
## Dominance frontier is not enough!

- Whenever a node  $n$  contains a definition of some uid  $\%x$ , then any node  $m$  in the dominance frontier of  $n$  needs a  $\phi$  statement for  $\%x$ .
- *But*, that is not the only place where  $\phi$  statements are needed



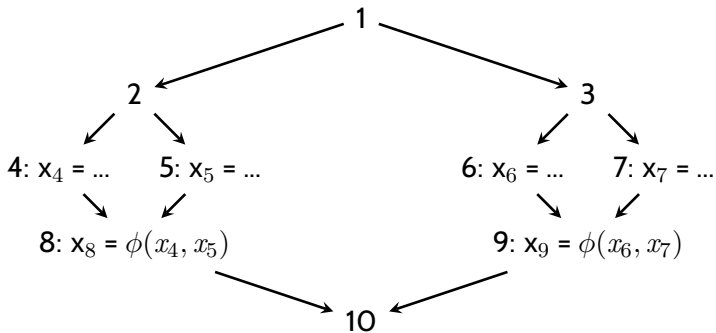
## Dominance frontier is not enough!

- Whenever a node  $n$  contains a definition of some uid  $\%_0x$ , then any node  $m$  in the dominance frontier of  $n$  needs a  $\phi$  statement for  $\%_0x$ .
- *But*, that is not the only place where  $\phi$  statements are needed



## Dominance frontier is not enough!

- Whenever a node  $n$  contains a definition of some uid  $\%x$ , then any node  $m$  in the dominance frontier of  $n$  needs a  $\phi$  statement for  $\%x$ .
- *But*, that is not the only place where  $\phi$  statements are needed



Not in dominance frontier of 4,5,6,7

## SSA construction

- Extend dominance frontier to sets of nodes by letting  $DF(M) = \bigcup_{m \in M} DF(m)$
- Define the *iterated dominance frontier*  $IDF(M) = \bigcup_i IDF_i(M)$ , where
  - $IDF_0(M) = DF(M)$
  - $IDF_{i+1}(M) = IDF_i(M) \cup IDF(IDF_i(M))$

## SSA construction

- Extend dominance frontier to sets of nodes by letting  $DF(M) = \bigcup_{m \in M} DF(m)$
- Define the *iterated dominance frontier*  $IDF(M) = \bigcup_i IDF_i(M)$ , where
  - $IDF_0(M) = DF(M)$
  - $IDF_{i+1}(M) = IDF_i(M) \cup IDF(IDF_i(M))$
- For any node  $x$ , let  $Def(x)$  be the set of nodes that define  $x$
- Finally, we can characterize  $\phi$  statement placement

Insert a  $\phi$  statement for  $x$  at every node in  $IDF(Def(x))$

## Transforming out of SSA

- The  $\phi$  statement is not executable, so it must be removed in order to generate code



## Transforming out of SSA

- The  $\phi$  statement is not executable, so it must be removed in order to generate code
- For each  $\phi$  statement  $\%x = \phi(\%x_1, \dots, \%x_k)$  in block  $n$ ,  $n$  must have exactly  $k$  predecessors  $p_1, \dots, p_k$
- Insert a new block along each edge  $p_i \rightarrow n$  which executes  $\%x = \%x_i$  (program no longer satisfies SSA property!)

## Transforming out of SSA

- The  $\phi$  statement is not executable, so it must be removed in order to generate code
- For each  $\phi$  statement  $\%x = \phi(\%x_1, \dots, \%x_k)$  in block  $n$ ,  $n$  must have exactly  $k$  predecessors  $p_1, \dots, p_k$
- Insert a new block along each edge  $p_i \rightarrow n$  which executes  $\%x = \%x_i$  (program no longer satisfies SSA property!)
- Using a graph coalescing register allocator, often possible to eliminate the resulting move instructions