COS320: Compiling Techniques

Zak Kincaid

January 29, 2026

Static Single Assignment form

SSA

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

```
if (x < 0) {
  y := y - x;
} else {
  y := y + x;
}
return y
```

$\longrightarrow$

```
if (x₀ < 0) {
  y₁ := y₀ - x₀;
} else {
  y₂ := y₀ + x₀;
}
y₃ := ϕ(y₁, y₂)
return y₃
```

  - 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.
  - Formal definition to follow!

Register allocation

- SSA form reduces register pressure
  - Each variable x is replaced by potentially many "subscripted" variables x_1, x_2, x_3,...
    - (At least) one for each definition of 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,...
    - (At least) one for each definition of 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,...
    - (At least) one for each definition of 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[1] 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
```

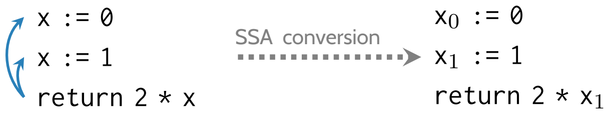
[1] does *not* eliminate dead *stores*

Dead assignment elimination

Simple algorithm for eliminating assignment[1] 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
```

```
x0 := 0
x1 := 1
return 2 * x1
```

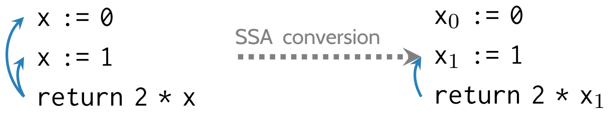
[1] does *not* eliminate dead *stores*

Dead assignment elimination

Simple algorithm for eliminating assignment[1] 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
```

```
x0 := 0
x1 := 1
return 2 * x1
```

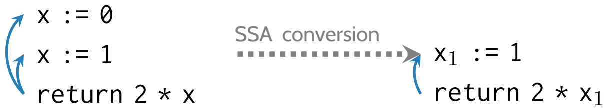
[1] does *not* eliminate dead *stores*

Dead assignment elimination

Simple algorithm for eliminating assignment[1] 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
```

```
x₁ := 1
return 2 * x₁
```

[1]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)
    - $\bot$ (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* $\in N$,
     $$\textbf{OUT}[\textit{bb}] \sqsupseteq \textit{post}_{CP}(\textit{bb}, \textbf{IN}[\textit{bb}])$$
  3. For each edge *src* $\to$ *dst* $\in E$,
     $$\textbf{IN}[\textit{dst}] \sqsupseteq \textbf{OUT}[\textit{src}]$$

(Dense) constant propagation performance

- Memory requirements: $\Theta(|N| \cdot |\mathit{Var}|)$
  - Constant environment has size $\Theta(|\mathit{Var}|)$, need to track $\Theta(1)$ per node
- Time requirements: $\Theta(|E| \cdot |\mathit{Var}|) = \Theta(|N| \cdot |\mathit{Var}|)$
  - Processing a single node takes $\Theta(1)$ time
  - Each edge is processed $\Theta(|\mathit{Var}|)$ times
    - Height of the abstract domain (length of longest strictly ascending sequence): $|\mathit{Var}| + 1$
- 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*$(\%y)$ *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*$(\%y)$ ***reads*** $\%x\}$
- Local specification for constant propagation:
  - *scp* is the smallest function *Uid* $\rightarrow \mathbb{Z} \cup \{\top, \bot\}$ such that
    - If G contains no assignments to *%x*, then *scp*$(\%x) = \top$
    - For each instruction *%x = e*, *scp*$(\%x) = eval(e,$ *scp*$)$
    - For each instruction *%x =ϕ(%y,%z)*, *scp*$(\%x) =$ *scp*$(\%y) \sqcup$ *scp*$(\%z)$

Worklist algorithm

```
$scp(\%x) = \begin{cases} \bot & \text{if } \%x \text{ has an assignment} \\ \top & \text{otherwise} \end{cases}$
$work \leftarrow \{\%x \in Uid : \%x \text{ is defined}\}$;
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	$\Theta(\lvert N\rvert \cdot \lvert \mathit{Var}\rvert)$	$\Theta(\lvert N\rvert) = \Theta(\lvert \mathit{Var}\rvert)$
Time	$\Theta(\lvert N\rvert \cdot \lvert \mathit{Var}\rvert)$	$\Theta(\lvert N\rvert) = \Theta(\lvert \mathit{Var}\rvert)$

- *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 ith 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 ith definition of y is the unique reaching definition
- If multiple definitions reach a single use, then they must be merged using a ϕ (phi) statement

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

$\rightarrow$

```
$y_0$ := 0;
while (true) {
  $x_2$ = $\phi$($x_0$, $x_1$)
  $y_2$ = $\phi$($y_0$, $y_1$)
  if ($x_2$ < 0) break;
  $x_1$ := $x_2$ - 1;
  $y_1$ := $y_2$ + $x_1$;
}
return $y_2$
```

Placing ϕ statements

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

[2]The entry node of the CFG is considered to be an implicit definition of every variable

Placing ϕ statements

- Easy, inefficient solution: place a ϕ statement for each variable locaction at each *join point*
  - A *join point* is a node in the CFG with more than one predecessor
- Better solution: place a ϕ 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[2]
  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*

[2]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 Haase diagram of the dominated-by order)

Control Flow Graph

$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[d, bend left] & & 4 \arrow[ddl] \\
5 \arrow[u, bend left] \arrow[dr] & & \\
 & 6 \arrow[uuu, bend right=60] \arrow[d] & \\
 & 7 &
\end{tikzcd}$$

Dominator tree

$$\begin{tikzcd}
 & 1 & \\
 & 2 \arrow[u] & \\
3 \arrow[ur] & & 4 \arrow[ul] \\
5 \arrow[u] & & \\
 & 6 \arrow[uuu] & \\
 & 7 \arrow[u] &
\end{tikzcd}$$

Dominance and SSA

- SSA well-formedness criteria
    - If $\%x$ is used in a non-ϕ statement in block n, then the definition of $\%x$ must dominate n
    - If $\%x$ is the ith argument of a ϕ function in a block n, then the definition of $\%x$ must dominate the ith predecessor of 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 $p \to n \in E$, *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 $p \to n \in E$, *dom*$(n) \subseteq \{n\} \cup$ *dom*(p)
- Can be solved using dataflow analysis techniques
  - In practice: nearly linear time algorithm due to Lengauer & Tarjan

- Recall: If $\%x$ is the ith argument of a ϕ function in a block n, then the definition of $\%x$ must dominate the ith predecessor of n.
- 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 $\%x$, then any node m in the dominance frontier of n needs a ϕ function for $\%x$.

Control Flow Graph

$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[d, bend left] & & 4 \arrow[ddl] \\
5 \arrow[u, bend left] \arrow[dr] & & \\
 & 6 \arrow[uuu, bend right=70] \arrow[d] & \\
 & 7 &
\end{tikzcd}$$

Dominator tree

$$\begin{tikzcd}
 & 1 & \\
 & 2 \arrow[u] & \\
3 \arrow[ur] & & 4 \arrow[ul] \\
5 \arrow[u] & & \\
 & 6 \arrow[uuu] & \\
 & 7 \arrow[u] &
\end{tikzcd}$$

- $DF(1) = \emptyset$

Control Flow Graph

$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[d, bend left] & & 4 \arrow[ddl] \\
5 \arrow[u, bend left] \arrow[dr] & & \\
 & 6 \arrow[uuu, bend right=60] \arrow[d] & \\
 & 7 &
\end{tikzcd}$$

Dominator tree

$$\begin{tikzcd}
 & 1 & \\
 & 2 \arrow[u] & \\
3 \arrow[ur] & & 4 \arrow[ul] \\
5 \arrow[u] & & \\
 & 6 \arrow[uuu] & \\
 & 7 \arrow[u] &
\end{tikzcd}$$

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

Control Flow Graph

$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[d, bend left] & & 4 \arrow[ddl] \\
5 \arrow[u, bend left] \arrow[dr] & & \\
 & 6 \arrow[d] \arrow[uuu, bend right=60] & \\
 & 7 &
\end{tikzcd}$$

Dominator tree

$$\begin{tikzcd}
 & 1 & \\
 & 2 \arrow[u] & \\
3 \arrow[ur] & & 4 \arrow[ul] \\
5 \arrow[u] & & \\
 & 6 \arrow[uuu] & \\
 & 7 \arrow[u] &
\end{tikzcd}$$

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

Control Flow Graph

$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[d, bend left] & & 4 \arrow[ddl] \\
5 \arrow[u, bend left] \arrow[dr] & & \\
 & 6 \arrow[uuu, bend right=60] \arrow[d] & \\
 & 7 &
\end{tikzcd}$$

Dominator tree

$$\begin{tikzcd}
 & 1 & \\
 & 2 \arrow[u] & \\
3 \arrow[ur] & & 4 \arrow[ul] \\
5 \arrow[u] & & \\
 & 6 \arrow[uuu] & \\
 & 7 \arrow[u] &
\end{tikzcd}$$

- $DF(1) = \emptyset$
- $DF(2) = \{2\}$
- $DF(3) = \{3, 6\}$
- $DF(4) = \{6\}$
- $DF(5) = \{3, 6\}$
- $DF(6) = \{2\}$

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 ϕ statement for $\%x$.
- *But*, that is not the only place where ϕ statements are needed

$$\begin{tikzcd}
 & & & 1 \arrow[dll] \arrow[drr] & & & \\
 & 2 \arrow[dl] \arrow[dr] & & & & 3 \arrow[dl] \arrow[dr] & \\
4\text{: x}_4 = \ldots \arrow[dr] & & 5\text{: x}_5 = \ldots \arrow[dl] & & 6\text{: x}_6 = \ldots \arrow[dr] & & 7\text{: x}_7 = \ldots \arrow[dl] \\
 & 8 \arrow[drr] & & & & 9 \arrow[dll] & \\
 & & & 10 & & &
\end{tikzcd}$$

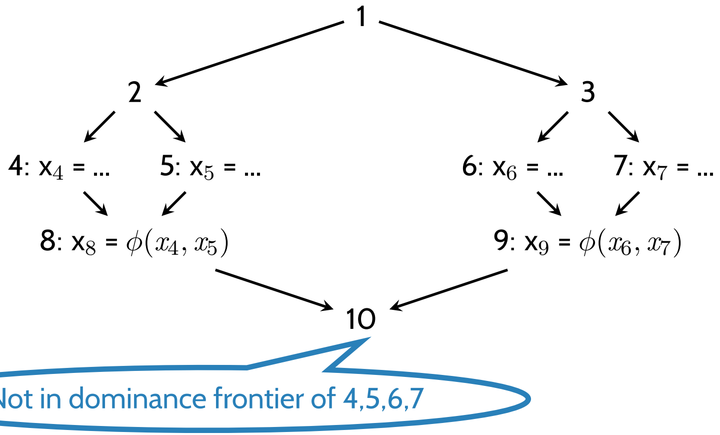
Dominance frontier is not enough!

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

$$\begin{tikzcd}
 & & & 1 \arrow[dll] \arrow[drr] & & & \\
 & 2 \arrow[dl] \arrow[dr] & & & & 3 \arrow[dl] \arrow[dr] & \\
4\text{: x}_4 = \ldots \arrow[dr] & & 5\text{: x}_5 = \ldots \arrow[dl] & & 6\text{: x}_6 = \ldots \arrow[dr] & & 7\text{: x}_7 = \ldots \arrow[dl] \\
 & 8\text{: x}_8 = \phi(x_4, x_5) \arrow[drr] & & & & 9\text{: x}_9 = \phi(x_6, x_7) \arrow[dll] & \\
 & & & 10 & & &
\end{tikzcd}$$

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 ϕ statement for $\%x$.
- *But*, that is not the only place where ϕ statements are needed


1
2
3
4: x$_4$ = ...
5: x$_5$ = ...
6: x$_6$ = ...
7: x$_7$ = ...
8: x$_8$ = $\phi(x_4, x_5)$
9: x$_9$ = $\phi(x_6, x_7)$
10
Not in dominance frontier of 4,5,6,7

Placing ϕ statements

- 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
  - $IDF_0(M) = DF(M)$
  - $IDF_{i+1}(M) = IDF_i(M) \cup IDF(IDF_i(M))$

Placing ϕ statements

- 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 ϕ statement placement:

$$\boxed{\text{Insert a } \phi \text{ statement for } x \text{ at every node in } IDF(\textit{\textbf{Def}}(x))}$$

Transforming out of SSA

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

Transforming out of SSA

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

Transforming out of SSA

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

SSA overview

- SSA can make analysis and optimization
  - simpler
  - more efficient
  - more accurate
- at the cost of
  - having to compute SSA / maintain SSA invariants
  - complicating code generation
- Most imperative compilers use SSA: LLVM, gcc, HotSpot, mono, v8, spidermonkey, go, ...
- *Dominance* is the key idea needed to efficiently transform into SSA
  - Will also make an appearence next week when we talk about loop optimizations