COS320: Compiling Techniques

Zak Kincaid

April 26, 2019

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:
  - 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.
  - If $\%x$ is used in a non-ϕ statement in block n, then the definition of $\%x$ must dominate n
  - Essentially: no using uninitialized uids. More on dominance later.

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

SSA admits a very simple algorithm for eliminating assignment instructions that are never used:

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

    Remove definition of $\%x$ from CFG;

- Note: does *not* eliminate dead *stores*

Recall: constant propagation

- Let $G = (N, E, s)$ be a control flow graph.
- *cp* is the *smallest*[1] function such that
  - $cp(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$
  - For each $p \to n \in E$, $post(p, cp(p)) \leq cp(n)$

$\mathit{cp}(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$\mathit{cp}(n) = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ for all other nodes;
work $\leftarrow N \setminus \{s\}$; /* Set of nodes that may violate spec */
while *work* $\neq \emptyset$ **do**
  Pick some n from work;
  work $\leftarrow$ *work* $\setminus \{n\}$;
  $C \leftarrow \bigsqcup_{p \in \mathit{pred}(n)} \mathit{post}(p, \mathit{cp}(p))$;
  if $C \neq \mathit{cp}(n)$ **then**
    $\mathit{cp}(n) \leftarrow C$;
    work $\leftarrow$ *work* $\cup\ \mathit{succ}(n)$

[1] Pointwise order: $f \leq g$ if for all nodes n and all variables x, $f(n)(x) \preceq g(n)(x)$

(Dense) constant propagation performance

- Memory requirements: $O(|N| \cdot |\mathit{Var}|)$
- Height of the abstract domain (length of longest strictly ascending sequence): $|\mathit{Var}|$
- Time requirements: $O(|N| \cdot |\mathit{Var}|)$
- 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
- Define *rhs*$(\%x)$ to be the right hand side of the unique assignment to $\%x$
- Define *succ*$(\%x) = \{\%y :$ *rhs*$(\%y)$ *reads* $\%x\}$

- *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***$)$

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

work $\leftarrow \{\%x \in$ *Uid* $: \%x$ 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*$(\%x)$

	Dense	Sparse
Memory	$O(\lvert N\rvert \cdot \lvert \mathit{Var}\rvert)$	$O(\lvert N\rvert) = O(\lvert \mathit{Var}\rvert)$
Time	$O(\lvert N\rvert \cdot \lvert \mathit{Var}\rvert)$	$O(\lvert N\rvert) = O(\lvert \mathit{Var}\rvert)$

However, observe that we only find constants for uids, not stack slots.

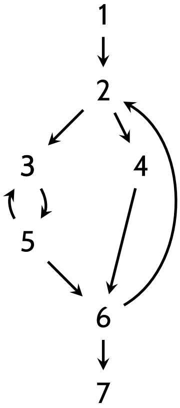
- Again: advantageous to use uids to represent variable whenever possible

Dominance

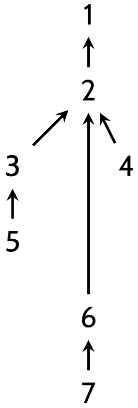
- Let $G = (N, E, s)$ be a control flow graph
- We say that a node $d \in N$ *dominates* a node $n \in N$ if every path from s to n contains d
  - Every node dominates itself
  - d *strictly dominates* n if d is not n
  - d *immediately dominates* n if d strictly dominates n and bud does not strictly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

Control Flow Graph



Dominator tree



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)

SSA construction

- In SSA, each *use* of a variable must be linked to a single corresponding definition
- If multiple definitions reach a single use, then they must be merged using a ϕ (phi) node

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

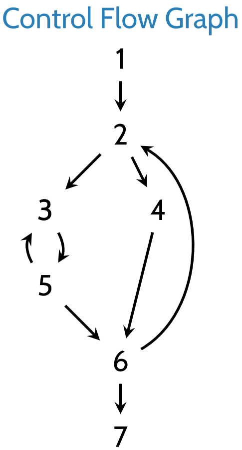
$\longrightarrow$

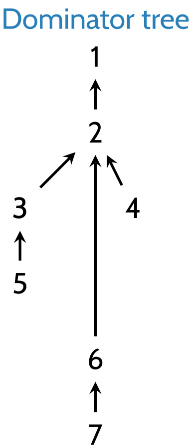
```
y₀ := 0;
while (true) {
  x₂ = ϕ(x₀, x₁)
  y₂ = ϕ(y₀, y₁)
  if (x₂ < 0) break;
  x₁ := x₂ - 1;
  y₁ := y₂ + x₁;
}
return y₂
```

- Easy, inefficient solution: place a ϕ 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 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 *dominance frontiers*

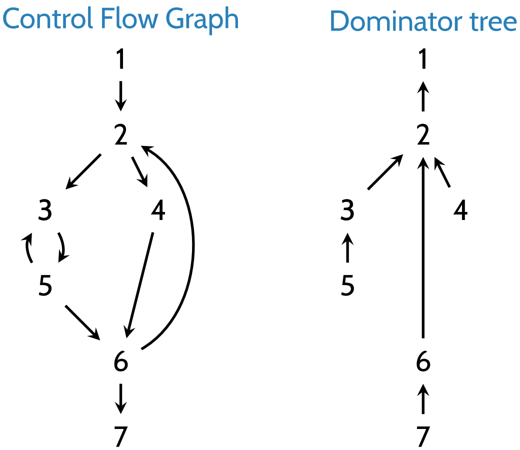
[2] The entry node of the CFG is considered to be an implicit definition of every variable

- 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 dominate 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 $\%x$.



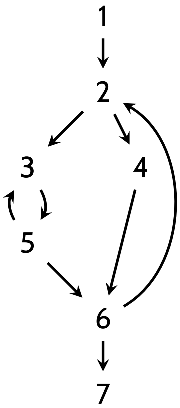


- $DF(1) = \emptyset$

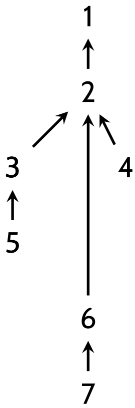


- $DF(1) = \emptyset$
- $DF(2) = \{2\}$

Control Flow Graph

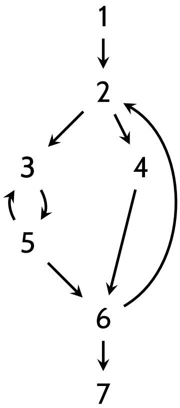


Dominator tree

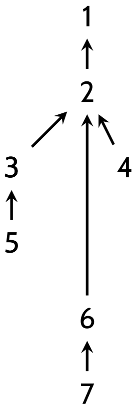


- $DF(1) = \emptyset$
- $DF(2) = \{2\}$
- $DF(3) = \{3, 6\}$

Control Flow Graph



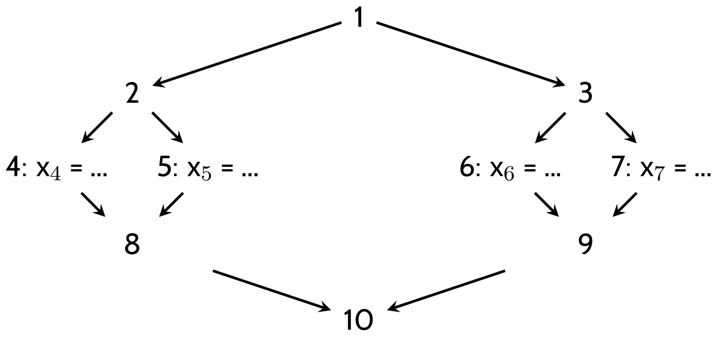
Dominator tree



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

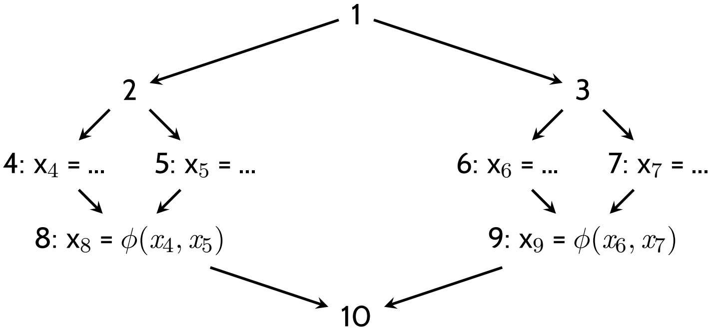
Dominance frontier is not enough!

- 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



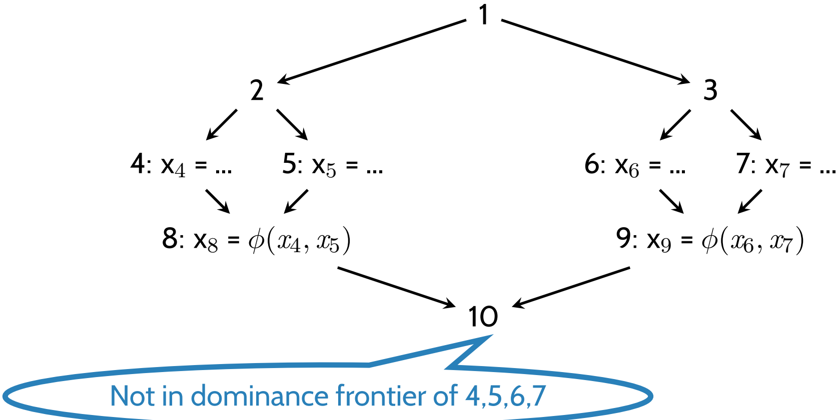
Dominance frontier is not enough!

- 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



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



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*$(M) = \bigcup IDF_i(M)$, 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
- Insert a ϕ statement for x at every node in $IDF($***Def***$(x))$

Transforming out of SSA

- The ϕ statement is not executable, so it must be removed in order to generate code
- For each ϕ statement $\%x = \phi(\%x_1, ..., \$x_k)$ in block n, n must have exactly k predecessors $p_1, ...p_k$
- Insert a new block along each edge $p_i \to n$ 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