COS320: Compiling Techniques

Zak Kincaid

April 11, 2024
Static Single Assignment form
Each %uid appears on the left-hand-side of at most one assignment in a CFG

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

→

```plaintext
if (x_0 < 0) {
    y_1 := y_0 - x_0;
} else {
    y_2 := y_0 + x_0;
}
y_3 := \phi(y_1, y_2)
return y_3
```

- 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 $x$
  - 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 $x$
  - 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 $x$
  - 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:

\[
\text{while some } \%x \text{ has no uses do}
\]
\[
\quad \text{Remove definition of } \%x \text{ from CFG;}
\]
\[
\quad \text{SSA conversion } \Rightarrow \text{ more assignments are eliminated}
\]

\[
\begin{align*}
x & := 0 \\
x & := 1 \\
\text{return } 2 \times x
\end{align*}
\]

\(^1\)does not eliminate dead stores
Dead assignment elimination

Simple algorithm for eliminating assignment\(^1\) instructions that are never used:

\[
\text{while some } \%x \text{ has no uses do}
\]
- Remove definition of \(\%x\) from CFG;

- SSA conversion $$\Rightarrow$$ more assignments are eliminated

\[
\begin{align*}
\text{x := 0} \\
\text{x := 1} \\
\text{return 2 * x}
\end{align*}
\]

\[
\begin{align*}
\text{x}_0 := 0 \\
\text{x}_1 := 1 \\
\text{return 2 * x}_1
\end{align*}
\]

\(^1\)does not eliminate dead stores
Dead assignment elimination

Simple algorithm for eliminating assignment\(^1\) instructions that are never used:

\[
\text{while some } \% x \text{ has no uses do}
\]
\[
\text{ Remove definition of } \% x \text{ from CFG;}
\]

- SSA conversion $\Rightarrow$ more assignments are eliminated

\[
\begin{align*}
&x := 0 \\
&x := 1 \\
&\text{return } 2 \times x
\end{align*}
\]

\[
\begin{align*}
&x_0 := 0 \\
&x_1 := 1 \\
&\text{return } 2 \times x_1
\end{align*}
\]

\(^1\)does not eliminate dead stores
Dead assignment elimination

Simple algorithm for eliminating assignment\(^1\) instructions that are never used:

```plaintext
while some \( \%x \) has no uses do
    Remove definition of \( \%x \) from CFG;

• SSA conversion \( \Rightarrow \) more assignments are eliminated
```

\[
\begin{align*}
    x &:= 0 \\
    x &:= 1 \\
    \text{return } 2 \times x
\end{align*}
\]

\[
\begin{align*}
    x_1 &:= 1 \\
    \text{return } 2 \times x_1
\end{align*}
\]

\(^1\)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 $\text{IN}, \text{OUT}$ is *conservative* if
  1. $\text{IN}[s]$ assigns each variable $\top$
  2. For each node $bb \in N$,
     \[ \text{OUT}[bb] \equiv \text{post}_{CP}(bb, \text{IN}[bb]) \]
  3. For each edge $src \rightarrow dst \in E$,
     \[ \text{IN}[dst] \equiv \text{OUT}[src] \]
(Dense) constant propagation performance

- Memory requirements: $\Theta(|N| \cdot |Var|)$
  - Constant environment has size $\Theta(|Var|)$, need to track $\Theta(1)$ per node
- Time requirements: $\Theta(|E| \cdot |Var|) = \Theta(|N| \cdot |Var|)$
  - Processing a single node takes $\Theta(1)$ time
  - Each edge is processed $\Theta(|Var|)$ times
    - **Height** of the abstract domain (length of longest strictly ascending sequence): $|Var| + 1$
- 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 $\text{rhs}(\%x)$ to be the right hand side of the unique assignment to $\%x$
  - Define $\text{succ}(\%x) = \{ \%y : \text{rhs}(\%y) \text{ reads } \%x \}$
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 $Uid \rightarrow \mathbb{Z} \cup \{\top, \bot\}$ 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 $\%x = \phi(\%y, \%z)$, $scp(\%x) = scp(\%y) \cup scp(\%z)$
Worklist algorithm

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

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

\begin{algorithm}
\[\]
while work \neq \emptyset do
\[\]
\hspace{1em} \text{Pick some } \%x \text{ from work;}
\hspace{1em} \text{work} \leftarrow \text{work} \setminus \{\%x\};
\hspace{1em} \text{if } rhs(\%x) = \phi(\%y, \%z) \text{ then}
\hspace{2.5em} v \leftarrow scp(\%y) \sqcup scp(\%z)
\hspace{1em} \text{else}
\hspace{2.5em} v \leftarrow \text{eval}(rhs(\%x), scp)
\hspace{1em} \text{if } v \neq scp(\%x) \text{ then}
\hspace{3.5em} scp(\%x) \leftarrow v,
\hspace{3.5em} \text{work} \leftarrow \text{work} \cup \text{succ}(\%x)
\]\end{algorithm}
Computational complexity of constant propagation

<table>
<thead>
<tr>
<th></th>
<th>Dense</th>
<th>Sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>$\Theta(</td>
<td>N</td>
</tr>
<tr>
<td>Time</td>
<td>$\Theta(</td>
<td>N</td>
</tr>
</tbody>
</table>

- *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

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

```plaintext
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 $\phi$ 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

$\phi$ statements can be placed 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. 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. The entry node of the CFG is considered to be an implicit definition of every variable.

$^2$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 $x$
  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)
Dominance and SSA

- SSA well-formedness criteria
  - If $x$ is used in a non-$\phi$ statement in block $n$, then the definition of $x$ must dominate $n$.
  - 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 $n$. 
Dominator analysis

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

• 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.
  \[
  DF(n) = \{ m : (\exists p \in Pred(m). n \in dom(p)) \land (m = n \lor n \notin dom(m)) \}\]

• Whenever a node $n$ contains a definition of some uid $\%x$, then any node $m$ in the dominance frontier of $n$ needs a $\phi$ function for $\%x$. 
\[ DF(1) = \emptyset \]
• $DF(1) = \emptyset$
• $DF(2) = \{2\}$
• $DF(1) = \emptyset$
• $DF(2) = \{2\}$
• $DF(3) = \{3, 6\}$
\[ \begin{align*}
\text{DF}(1) &= \emptyset \\
\text{DF}(2) &= \{2\} \\
\text{DF}(3) &= \{3, 6\} \\
\text{DF}(4) &= \{6\} \\
\text{DF}(5) &= \{3, 6\} \\
\text{DF}(6) &= \{2\} \\
\end{align*} \]
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 \( \%_0x \).
- \textbf{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.

```
4: x_4 = ...
5: x_5 = ...
8: x_8 = \phi(x_4, x_5)
6: x_6 = ...
7: x_7 = ...
9: x_9 = \phi(x_6, x_7)
```
Dominance frontier is not enough!

- Whenever a node $n$ contains a definition of some uid $%_0 x$, then any node $m$ in the dominance frontier of $n$ needs a $\phi$ statement for $%_0 x$.
- *But*, that is not the only place where $\phi$ statements are needed
Placing $\phi$ 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
  - $IDF_0(M) = DF(M)$
  - $IDF_{i+1}(M) = IDF_i(M) \cup IDF(IDF_i(M))$

Finally, we can characterize $\phi$ statement placement: Insert a $\phi$ statement for $x$ at every node in $IDF(Def(x))$.
Placing $\phi$ 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 $\phi$ 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 $\%x = \phi(\%x_1, \ldots, \%x_k)$ in block $n$, $n$ must have exactly $k$ predecessors $p_1, \ldots, p_k$.
- Insert a new block along each edge $p_i \rightarrow n$ that 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 $%x = \phi(%x_1, \ldots, %x_k)$ in block $n$, $n$ must have exactly $k$ predecessors $p_1, \ldots, p_k$.
- Insert a new block along each edge $p_i \rightarrow n$ 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 appearance next week when we talk about loop optimizations