CSE 504: Compiler Design

Static Single Assignment (SSA) Form

Pradipta De
pradipta.de@sunykorea.ac.kr
Current Topic

- Iterative Data Flow Analysis
- LiveOut sets
- Static Single Assignment (SSA) Form
Motivation

• Many dataflow problems require
  – The location where a defined variable is used
  – The location where a used variable is defined

• Can be captured using a def-use chain
  – Data structure that keeps
    • A list of pointers to all the use sites of variables that are defined there
    • A list of pointers to all definition sites of the variables defined there

• With N uses and M definitions, space and time complexity will be O(N.M) for def-use chains
SSA Form

- Each variable is assigned exactly one definition in the program text
  - Analysis becomes simpler with only one definition
  - SSA form needs space linear to size of program
  - Simplifies unrelated use of same variable

```plaintext
for i ← 1 to N do A[i] ← 0
for i ← 1 to M do s ← s + B[i]
```

Although i is used in both loops, the intermediate code temporary variable can rename it to different names while maintaining program correctness.
Why do we need SSA form?

Convert Straight-line code to use new variables

When two control paths merge, then how to assign new variable name?

Introduce a notation $\Phi$ that represents that two variables merge at some point
Although \( c_1 \leftarrow c_2 + b_2 \) is recomputed every time in the loop, the assignment statement does not change \( \Rightarrow \) the assignment is “static” (not dynamic)

Hence the name: **Static** Single Assignment (SSA) form
Convert a Program to SSA form

- Add $\phi$ functions
- Rename all definitions and the use of the variables using subscripts

Program:

```
i ← 1
j ← 1
k ← 0
while k < 100
  if j < 20
    j ← i
    k ← k + 1
  else
    j ← k
    k ← k + 2
return j
```

CFG:

```
1. i ← 1
   j ← 1
   k ← 0
2. if k < 100
3. if j < 20
4. return j
5. j ← i
   k ← k + 1
6. j ← k
   k ← k + 2
```

$\phi$ functions inserted:

```
1. i ← 1
   j ← 1
   k ← 0
2. if k < 100
3. if j < 20
4. return j
5. j ← i
   k ← k + 1
6. j ← k
   k ← k + 2
```

Variables Renamed:

```
1. i_1 ← 1
   j_1 ← 1
   k_1 ← 0
2. if k_2 < 100
3. if j_2 < 20
4. return j_2
5. j_3 ← i_1
   k_3 ← k_2 + 1
6. j_5 ← k_2
   k_5 ← k_2 + 2
```

CSE 504: Compiler Design
How to insert $\phi$ functions

• Add a $\phi$ function for every variable at join points
  – Unnecessary if a variable is reached by the same definition along both edges

• There should be a $\phi$ function for a variable $a$ at node $z$ of the flow graph, when all of the following conditions are true
  – 1. There is a block $x$ containing a definition of $a$,
  – 2. There is a block $y$ (with $y = x$) containing a definition of $a$,
  – 3. There is a nonempty path $P_{xz}$ of edges from $x$ to $z$,
  – 4. There is a nonempty 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 $z$, and
  – 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
Simple method to add $\phi$ function

\[
\text{while there are nodes } x, y, z \text{ satisfying conditions 1–5 and } z \text{ does not contain a } \phi\text{-function for } a \\
\text{do insert } a \leftarrow \phi(a, a, \ldots, a) \text{ at node } Z
\]

The $\Phi$ function has as many ‘a’ arguments as there are predecessors of node $z$

The algorithm must examine every triple of nodes $<x, y, z>$ ➔ not efficient
The **dominance frontier** of a node $x$ is the set of all nodes $w$ such that $x$ dominates a predecessor of $w$, but does not strictly dominate $w$.

$x$ strictly dominates $w$ if $x$ dominates $w$ and $x \neq w$.

5 dominates all the nodes in the grey area (NOTE: d dominates n if every path from start node to n passes through d)

Dominance frontier of 5 is \{4, 5, 12, 13\}

Whenever node $x$ contains a definition of some variable $a$, then any node $z$ in the dominance frontier of $x$ needs a $\varphi$-function for $a$

Any node in the dominance frontier of $n$ is also a point of convergence of non intersecting paths, one from $n$ and one from the root node
Computing Dominance Frontier

Definitions:
1. Given a node \( n \) in a flow graph, the set of nodes that strictly dominate \( n \) is given by \((\text{Dom}(n) - n)\). The node in that set that is closest to \( n \) is called \( n \)'s immediate dominator, denoted \( \text{IDom}(n) \).
2. Dominator Tree: contains all nodes of the flow graph, but there is an edge from \( m \) to \( n \), if \( m \) is \( \text{DOM}(n) \).

<table>
<thead>
<tr>
<th>( B )</th>
<th>( B_0 )</th>
<th>( B_1 )</th>
<th>( B_2 )</th>
<th>( B_3 )</th>
<th>( B_4 )</th>
<th>( B_5 )</th>
<th>( B_6 )</th>
<th>( B_7 )</th>
<th>( B_8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{DOM} )</td>
<td>( {0} )</td>
<td>( {0,1} )</td>
<td>( {0,1,2} )</td>
<td>( {0,1,3} )</td>
<td>( {0,1,3,4} )</td>
<td>( {0,1,5} )</td>
<td>( {0,1,5,6} )</td>
<td>( {0,1,5,7} )</td>
<td>( {0,1,5,8} )</td>
</tr>
<tr>
<td>( \text{IDOM} )</td>
<td>—</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Given a node \( n \) in the dominator tree, \( \text{IDom}(n) \) is just its parent in the tree.

Algorithm to compute Dominance Frontier uses the Dominator Tree
Efficiently Inserting $\phi$ functions

- A definition of $x$ in block $b$ forces a $\phi$-function at every node in $DF(b)$
- $\phi$-function is a new definition of $x$ $\Rightarrow$ it may force the insertion of additional $\phi$-function
- A variable that is live within a single block can never have a live $\phi$-function
Renaming

- After $\varphi$-functions are placed, walk the dominator tree and rename different definitions
- In straight-line program,
  - Rename all definitions of a
  - Rename each use of a variable, say $x$, to the most recent definition of $x$
- In control-flow with branches and joins,
  - Each use of $x$ is renamed to the closest definition $d$ of $x$ that is above $x$ in the dominator tree.
- The algorithm for renaming works by,
  - Traversing the dominator tree and remembers for each variable the most recently defined version of each variable
  - It uses a separate stack data structure for each variable
  - It takes time proportional to the size of the program
Convert to SSA Form using DF

\[ i \leftarrow 1 \]
\[ j \leftarrow 1 \]
\[ k \leftarrow 0 \]

while \( k < 100 \)
  if \( j < 20 \)
    \[ j \leftarrow i \]
    \[ k \leftarrow k + 1 \]
  else
    \[ j \leftarrow k \]
    \[ k \leftarrow k + 2 \]
return \( j \)

<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>{2}</td>
</tr>
<tr>
<td>3</td>
<td>{2}</td>
</tr>
<tr>
<td>4</td>
<td>{}</td>
</tr>
<tr>
<td>5</td>
<td>{7}</td>
</tr>
<tr>
<td>6</td>
<td>{7}</td>
</tr>
<tr>
<td>7</td>
<td>{2}</td>
</tr>
</tbody>
</table>
Application of SSA Form in Optimization

- **Dead Code Elimination**
  - A variable is live at its site of definition if and only if its list of uses is not empty.
  - Delete variable $v$ with no use

- **Simple Constant Propagation**
  - If there is a statement of the form $v \leftarrow c$ for some constant $c$, then replace any use of $v$ with $c$.
  - Any $\varphi$-function of the form $v \leftarrow \varphi(c_1,c_2,\ldots,c_n)$, where all $c_i$s are equal, can be replaced by $v \leftarrow c$. 
Simple Constant Propagation

\[ W \leftarrow \text{a list of all statements in the SSA program} \]

\[ \text{while } W \text{ is not empty} \]

\[ \begin{align*}
\text{remove some statement } S \text{ from } W \\
\text{if } S \text{ is } v \leftarrow \phi(c, c, \ldots, c) \text{ for some constant } c \\
\text{replace } S \text{ by } v \leftarrow c \\
\text{if } S \text{ is } v \leftarrow c \text{ for some constant } c \\
\text{delete } S \text{ from the program} \\
\text{for each statement } T \text{ that uses } v \\
\text{substitute } c \text{ for } v \text{ in } T \\
W \leftarrow W \cup \{T\}
\end{align*} \]

More transformations can be incorporated in this algorithm:

- **Copy Propagation**: Replace a single argument \( \phi \)-function with the LHS value
- **Constant Folding**: For a statement \( x \leftarrow a \cdot b \), where \( a \) and \( b \) are constants, evaluate \( c \leftarrow a \cdot b \) and replace \( x \) with \( c \)
- **Constant Conditions**: If conditional branch conditions are constants, then replace code with labeled jumps.
- **Unreachable Code**: Deleting a predecessor may cause a block to be unreachable \( \Rightarrow \) delete all statements in the unreachable block
Summary

• Basic notion of SSA
  – Combine dataflow and control flow analysis using a single structure
• Techniques to convert program to SSA form
• Simple optimizations using SSA form