2.4 Control flow graph (CFG)

CFG a graph to represent one function/method

- nodes represent basic blocks
- edges represent possible control transfers

one special node: START: designated start node
(first node of any path through the control flow graph)
Construct control flow graph

1) walk (recursively) over the AST (abstract syntax tree) and form basic blocks as needed
2) process assembly language program
   (watch out for targets of jumps and branches)
3) rework structure from basic compiler design

CFG relates to SSA:
where do we insert $\phi$ nodes?
- where
- which arguments must be used?
Dominance relationship captures properties of paths and basic blocks.

basic block \( A \) dominates basic block \( B \) iff \( A \) is on every path from START to \( B \).

**Dominance tree (DT):** immediate dominance relationship

- nodes: basic block (same b.b. as in the CFG)
- edges: immediate dominance
  \( A \) dom \( B \): \( A \) dominates \( B \) and \( A \) immediate dominator of \( B \): there is no b.b. \( C \) such that \( A \) dom \( C \) and \( C \) dom \( B \)
Given a CFG, there are two approaches to insert φ nodes:

1. Analyze program (look at dominator tree)
   (3.0) (works for all programs)

2. Syntax-directed approach
   (look at patterns in the CFG)
   (2.5) (looks for some...)

   finite set of nodes
Example CFG

\[ x = \]

\[ x = \]

\[ = x \]
Turn program into SSA form

- option 1
  - insert a φ node into blocks that read a variable (X)

Consequence: φ node needs to allow a number of arguments. Number is equal to the number of paths that reach block and may affect X.

Here: 4 paths

Problems:
- all paths must be considered
- unbounded number of loops
1

2

\begin{align*}
x_3 &= \phi(i, z) \\
x_7 &= \phi(x_3, x_3) \\
x_5 &= \phi(x_3, x_4) \\
\end{align*}

\text{1) option 2}

\begin{itemize}
  \item Insert \(\phi\) nodes as early as possible even if there is no read/use of variable \(x\) in a given block
  \item Combines multiple paths
  \item Number of arguments of \(\phi\) node is a function of the max. number of predecessors
\end{itemize}
We use option 2 as the number of arguments for $\phi$ nodes can be bounded.

Now, if the programming language restricts the kind of CFG that can be built, then we can use a set of patterns to insert the $\phi$ nodes.

Restriction: for well-structured programs there exist a limited set of patterns in the CFG that we must deal with.

- programs without goto
- ...
Patterns in well-structured programming language

if - then

if - then - else

while loop

*: φ node model
Well-structured programs may include more sophisticated constructs.
2.5 Syntax-directed insertion of ϕ nodes

Task: given a program (CFG), transform program into SSA form
- insert ϕ nodes
- use correct versions of variables as arguments

For now: we consider only scalar variables
  (method-local)
  - no pointers
  - no arrays
  - no fields in objects
First step: augment the symbol table

For all scalar variables we keep:
- current version (integer version number)
- next version (int to be used for assignment)

2.5.1 Straight line code (a.k.a. basic blocks)

- process block from first statement to last statement, one statement at a time

1. If a variable $V$ appears on the right hand side, use the current version found in symbol table $V_i$:
- replace $V$ with $V_i$
(2) If a variable appears on the left hand side (i.e., it's modified) use \( v_{\text{next}} \) as version of newly created instance, update \( v_{\text{current}} \), increment \( v_{\text{next}} \).
Example

\[ s_1 \]
\[ x_1 = 6 \]

\[ s_2 \]
\[ y_1 = x_1 \]

\[ s_3 \]
\[ x_2 = 9 \]

\[ s_4 \]
\[ z_1 = y_1 \]

\[ s_5 \]
\[ y_2 = z_1 \]

<table>
<thead>
<tr>
<th>symbol</th>
<th>table</th>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s1</th>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s2</th>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s3</th>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s4</th>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s5</th>
<th>c</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
2.5.2 If-then-else

- - -

Binit

Bthen " |

Belse "|

B continue

Binit has been processed
* S.T. is up to date for all variables
(i) Process B then

1) copy the symbol table (call copy ST')

2) process & then-block statements

   - Single b.b.: use 2.5.1
   - Other construct: use 2.5.x

If we find a variable $v$ that has an assignment in $B$ then:

   - create a $\phi$ node in $B$
     continue

   $\phi$ node argument is set by $v$. Next in $B$ then

   $v = \phi(v_{\text{then-block}}, \text{open})$
(If there is already a \( \Phi \) node: update arguments)

(ii) process \( B \) else

1) reset symbol table \( S.T. \) to \( S.T.' \) (as read before (i))

2) copy the "next" field of all entries from \( S.T. \) \( B \) then into \( S.T. \)

3) process statements

- insert \( \Phi \) nodes (as for \( B \) then)
  - if there is none
  - update open in existing \( \Phi \) nodes

\( \Phi \) clean up