Control Flow Analysis & Def-Use

Hwansoo Han
Control Flow Graph

- What is CFG?
  - Represents program structure for internal use of compilers
  - Used in various program analyses
  - Generated from AST or a sequential list of statements

- Basic block
  - To build CFG, first partition a program into basic blocks
  - Longest possible sequence of straight-line code
    - No branch-in/out in the middle of basic block
    - Single entry at the beginning (label, join-point)
    - Single exit at the end (can be branch or non-branch)
Build a CFG

- Given N, a set of basic blocks, a CFG is built by creating E, a set of edges that connect the blocks in the graph.

- Create a directed edge \((N_i, N_j)\) in E if \(N_i\) and \(N_j\) satisfy either condition:
  1. There is a condition or unconditional jump from \(N_i\) to \(N_j\).
  2. \(N_j\) is an immediate successor of \(N_i\) in the original order of the program and the last statement of \(N_i\) is not unconditional jump.

- If a program starts/exits at multiple nodes, a unique node is created for the Start/Exit node with an edge between each original entry/exit node.
CFG for Bubble Sort Code

ti := n − 2

goto L1 on ti < 0

tj := 0

goto L2 on tj > i

t1 := tj
t2 := 4 * t1
t3 := a[t2]
t4 := tj + 1
t5 := t4
t6 := 4 * t5
t7 := a[t6]
goto L3 on t3 ≤ t7

L1: ...

L2: ...

L3: ...

t8 := tj
t9 := 4 * t8
tx := a[t9]
t10 := tj + 1
t11 := t10
t12 := 4 * t11
t13 := a[t12]
t14 := tj
t15 := 4 * t14
a[t15] := t13
t16 := tj + 1
t17 := t16
t18 := 4 * t17
a[t18] := tx

ti := ti − 1

goto L5

tj := tj + 1

goto L4
Unreachable Code Elimination

- Graph search algorithms can be directly used for several optimization techniques

- *Unreachable code elimination*
  - An instruction is *unreachable* when there is no path from the entry
  - “If an instruction is not reachable from S, the block containing the instruction will not be executed; therefore it can be safely removed from the code”
  - An algorithm based DFS (or BFS)
    1. Use DFS to traverse the CFG and mark instructions that are visited in the traversal
    2. Remove all unmarked instructions
Unreachable Code Elimination (cont’d)

- Basic blocks are *unreachable* when there is no path from the entry
  - Orange block is unreachable
  - The declaration of `z` may also be removed.

```c
int test(int n)
{
    int x, y, z, w;
    if (n < 2)
        x = 3;
    else
        x = 0;
    goto l;
    z = x + 5;
    y = z * 3;
    l: w = x + 10;
    y = w * 2;
    return y;
}
```
Edge Classification

- Depth-First-Spanning-Tree on CFG
  - $G_T = \langle N, E_T \rangle$, $E_T$: edges in DFST
  - Edge classification is useful for several compiler analyses

- The four types of edges in $E$
  1. Tree edges: all edges $\in E_T$
  2. Forward edges: edges $\not\in E_T$ and a node to a direct descendant
  3. Back edges: edges $\not\in E_T$ and a node to one of its ancestor (possibly to itself) in $G_T$
  4. Cross edges: edges $\not\in E_T$ and neither node is an ancestor of the other
Dominance Relation

- **Dominance**
  - A binary relation indicates which basic block always precedes which blocks in the execution path
  - Useful for various control/data flow analysis techniques (identifying loops, def-use chains, etc...)

- **Given** $G=(N,E,s)$, and two nodes $d, n \in N$,
  - $d$ dominates $n$ – written as $d \text{ dom } n$, if every path from the entry node $s$ to $n$ goes through $d$
  - $d$ strictly (or properly) dominates $n$, if $d \text{ dom } n$ and $d \neq n$
  - $d$ immediately dominates $n$, if $d$ is the last dominator of $n$ on any path from the entry node $s$ to $n$
    - $d$ properly dominates $n$ and any other node $k$, which properly dominates $n$, also properly dominates $d$
Lemmas for Dominance Relation

1. The initial node $s$ of a flow graph $G = (N,E,s)$ dominates all nodes of $G$ reachable from $s$.
2. Node $n$ dominates itself, $n \text{ dom } n$
3. The dominance relation of $G$ is a partial ordering.
   a. Reflexive: $n \text{ dom } n$
   b. Transitive: $n \text{ dom } m \land m \text{ dom } k \Rightarrow n \text{ dom } k$
   c. Anti-symmetric: $n \text{ dom } m \land m \text{ dom } n \Rightarrow n = m$
4. The dominators of a node forms a chain.
   $\Rightarrow$ The partial ordering property enables the compiler to build a chain of ordering of dominance relations between nodes in a certain subset of $N$. 
Algorithm to Find All Dominators

dom(n) : set of nodes that dominates n

\[
dom(n) = \{n\} \cup \bigcap_{p \in \text{pred}(n)} \dom(p)
\]

```python
find_all_doms(G):
    dom(entry) = \{entry\}
    for all n in N-\{entry\}
        dom(n) = N
        while any dom(n) changes
            for all n in N-\{entry\}
                for all predecessors p of n
                    dom(n) = dom(n) \cap dom(p)
                    dom(n) = dom(n) \cup \{n\}
            end
    end
```

<table>
<thead>
<tr>
<th>CFG</th>
<th>dom(1)</th>
<th>dom(2)</th>
<th>dom(3)</th>
<th>dom(4)</th>
<th>dom(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{1}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
</tr>
<tr>
<td>2</td>
<td>{1}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
</tr>
<tr>
<td>3</td>
<td>{1}</td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4,5}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>4</td>
<td>{1}</td>
<td>{1}</td>
<td>{1,2,4}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>5</td>
<td>{1}</td>
<td>{1,2,4}</td>
<td>{1,3,4}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
</tbody>
</table>
Dominator Tree

- Dominator tree, $T = (N, E_D, s)$, can be created from CFG, $G = (N, E, s)$
  - $E_D$ is a set of edges $(n, m)$ such that $n \ idom m$
  - $s$ is the root of the tree.

- In a dominator tree
  - A node dominates itself and its descendents (all the nodes in its subtree)
  - A node immediately dominates its child nodes

<table>
<thead>
<tr>
<th>CFG</th>
<th>Dominator Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4 (dom)</td>
</tr>
<tr>
<td>3</td>
<td>4 (dom)</td>
</tr>
<tr>
<td>5</td>
<td>1 (dom)</td>
</tr>
</tbody>
</table>

- $dom(1) = \{1\}$
- $dom(2) = \{1, 2, 4\}$
- $dom(3) = \{1, 3, 4\}$
- $dom(4) = \{1, 4\}$
- $dom(5) = \{1, 4, 5\}$

- $idom(1) = none$
- $idom(2) = 4$
- $idom(3) = 4$
- $idom(4) = 1$
- $idom(5) = 4$
Dominance can be used in an optimization technique such as common subexpression elimination (CSE)

\[ \text{dom}(n_1) \cap \text{dom}(n_2) = d \]
Some compiler optimizations (e.g., prefetching) try to move operations to the earliest point to reduce pipeline stalls.

The operation may be moved to the immediate dominator, unless the move does not violate the original data dependence.

The operation can be repeatedly moved to the next immediate dominator, if data dependence is preserved.
Postdominance Relation

- Flow graph $G=(N,E,s)$
  - $u$ *postdominates* $v$, if every path from $v$ to the exit node goes through $u$
  - $u$ *immediately postdominates* node $v$, if $u$ is the first postdominator on the every path from $v$ to the exit node
    - $u$ properly postdominates $v$ and any other node $w$, which properly postdominates $v$, also properly postdominates $u$

- Postdominance is a dual of dominance.
Loops, Strongly Connected Components

- Back edges form loops!
  - Loop entry: a node one of whose predecessors is not in the loop
  - Loop exit: a node one of whose successors is not in the loop

- Natural loop
  - For a back edge \((m \rightarrow n)\), \(n\) dominates \(m\) (i.e. one entry node)
  - Most loops from high level construct are natural loops

- Strongly connected component (SCC)
  - A subgraph of \(G\), \(G_{scc} = (N_{scc}, E_{scc})\)
  - For any \(u, v \in N_{scc}\), a path from \(u\) to \(v\) always exists and vice versa.
Natural Loop Example

**Dominator Tree**

- **Entry**
  - B1
    - B2
      - B3
      - B4
      - Exit
    - B5
      - Exit
  - B2 dominates B4

**CFG**

- **Entry**
  - B1
    - B2
      - B3
      - B4
        - B5
          - Exit
    - B3
      - Exit
  - B2 doesn’t dominate B3
  - B2, B3 is **NOT** a natural loop
CFA Summary

- Control Flow Graph
  - Basic blocks
  - Edges – control flow

- Dominance relation
  - Dominator tree
  - Postdominance

- Loop
  - Natural loops
  - Strongly connected components
Many optimization algorithms need to know …

“the relationship between definitions of variables and the point where they are actually used later in the flow of control.”

DU(Def-Use) and UD(Use-Def) chains are a representation of reaching definition information.

- A DU chain connects a definition of a variable to all the uses the definition may reach.
- A UD chain connects a use of a variable to all the definitions that may reach to it.

They are implemented with linked lists.

They can be obtained directly from DFI of the problem.

Thus, they can be constructed by solving the reaching definitions data flow problem.
Example: UD chains
Example: DU chains
Webs

- DU chains are used to build a web which is the maximal union of intersecting DU-chains for a variable
- Webs are useful for register allocation
  - A single register can be allocated for each web
Static Single Assignment (SSA) form

- **Multiple Assignment Form**
  - Ordinary imperative languages
  - No restriction on # of assignments of values to a variable
  - A variable = a location

- **Single Assignment Form**
  - A variable = a value
  - Only a single assignment to a variable within the whole code.
  - Compilers statically generate SA form as an intermediate form for data flow analysis → *static single assignment* (SSA) form
  - SSA is a compact representation for Def-Use information
  - Used in real compilers
SSA (cont’d)

- SSA form needs code translation between MA and SA
  - However, the translation time is quite fast ($= O(n) \rightarrow$ read the literature “Efficiently computing static single assignment form and the control dependence graph by R. Cytron, et al. ACM Transactions on Programming Languages and Systems (TOPLAS) October 1991”).

- DU chains are harder to incrementally update than SSA
  - DU-chain provides reaching definition information with linked list, while SSA form provides the same information itself without additional data structure.
Incremental Update

**DU chains**

\[
x := 4 \\
y := y + 3 \\
goto L \text{ on } x < 5 \\
z := y * 2 \\
x := y + 9 \\
L: z := y * 5
\]

**constant propagation**

\[
x := 4 \\
y := y + 3 \\
goto L \text{ on } 4 < 5 \\
z := y * 2 \\
x := y + 9 \\
L: z := y * 5
\]

**need extra operation to avoid dangling pointers**

\[
x := 4 \\
y := y + 3 \\
L: z := y * 5
\]

**dead code elimination**

**SSA forms**

\[
x_1 := 4 \\
y_2 := y_1 + 3 \\
goto L \text{ on } x_1 < 5 \\
z_1 := y_2 * 2 \\
x_2 := y_2 + 9 \\
L: x_3 := \phi(x_1,x_2) \\
z_2 := \phi(z_0,z_1) \\
z_3 := y_2 * 5
\]

**constant propagation**

\[
x_1 := 4 \\
y_2 := y_1 + 3 \\
goto L \text{ on } 4 < 5 \\
z_1 := y_2 * 2 \\
x_2 := y_2 + 9 \\
L: x_3 := \phi(x_1,x_2) \\
z_2 := \phi(z_0,z_1) \\
z_3 := y_2 * 5
\]

**dead code elimination**

\[
x_1 := 4 \\
y_2 := y_1 + 3 \\
L: x_3 := x_1 \\
z_2 := z_0 \\
z_3 := y_2 * 5
\]
Translating MA form into SSA form

- To convert an ordinary program to an SSA form
  - A pseudo-instruction \( v=\phi(v_1,v_2,\ldots,v_n) \) is added to a join node where multiple distinct definitions of \( v \) reach
  - Variables are renamed to remove multiple assignments

```plaintext
if (A < B) then
  A := 7;
else
  A := 0;
endif
B := A;
```

```plaintext
if (A < B) then
  A1 := 7;
else
  A2 := 0;
endif
A3 := \phi(A1,A2)
B := A3;
```
SSA Loop Example

- $\phi$ explicitly inserted at merge point of values

```
Read(N)
    I := 1
L2: if (I>N) goto L3
    A(I) := A(I) +1
    I := I + 1
    goto L2
L3: Print( A(N) )
```

```
Read(N)
    I1 := 1
L2: I3 = $\phi$(I1, I2)
    if (I3>N) goto L3
    A(I3) := A(I3) +1
    I2 := I3 + 1
    goto L2
L3: Print( A(N) )
```
Dominance Relation (review)

- Dominance is a binary relation
  - If basic block A always precedes basic block B in the execution path on CFG, A dominates B.

- Given $G=(N, E, s)$, and two nodes $n$ and $m$ in $N$,
  - $n$ dominates $m$: $n \in \text{dom}(m)$
    if every path from $s$ to $m$ contains $n$.
  - $n$ strictly (or properly) dominates $m$: $n \in \text{sdom}(m)$
    if $n \in \text{dom}(m)$ and $n \neq m$.
  - $n$ immediately dominates $m$: $n = \text{idom}(m)$
    if $n$ properly dominates $m$ and any other node $k$ that properly dominates $m$ also properly dominates $n$. 
Dominance Frontier

- Dominance frontier of a node $x$

$$\text{DF}(x) = \{y | \exists z \in \text{Pred}(y) \text{ s.t. } x = \text{dom}(z) \text{ and } x \neq \text{sdom}(y)\}$$
Dominance Frontier Examples
Translating SSA From

- **Iterated dominance frontier**
  \[ DF^+(S) = \lim_{i \to \infty} DF^i(S) \]
  where \( DF^1(S) = DF(S), \ DF^{i+1}(S) = DF(S \cup DF^i(S)) \)

- **Nodes that require \( \phi \) for a variable \( x \)**
  Let \( S = \{ \text{nodes with assignment to } x \} \cup \{ \text{Entry} \} \)
  \( DF^+(S) \) is the set of nodes that need \( \phi \)

- **Worklist algorithm**
  for each variable \( x \)
  \[ \text{worklist} = \{ \text{all nodes that have assignment to } x \} \quad // \text{initialize} \]
  for each \( y \in \text{worklist} \)
  for each \( z \in DF(y) \) and \( \phi \) not in \( z \)
  place \( \phi \) and add \( z \) on \text{worklist}
Renaming Variables (SSA)

- Stack algorithm

  // Initialization
  Keep separate stack and index for each variable
  Set each index to zero

  // Traversal of Dominator Tree, starting from Entry
  Visit(Node)
  
  RHS : rename with index from variable’s Top-Of-Stack
  LHS : rename with variable’s index; push (index); index++
  if $\phi$ exists in Succ(Node) in CFG:
    rename variable in $\phi$ with Top-Of-Stack index
  For each child node N in Dominator Tree
  Visit(N) // recursive calls
  pop stack for each LHS assignment in Node
Example: SSA translation

- Place $\phi$ for variable, $v$ and $w$
- Rename variables