Control-Flow Static Analysis

Dragon Book: Chapter 8, Section 8.4, Chapter 9, Section 9.6
Outline

• Program representation: three-address code
• Control-Flow Graphs (CFGs)
• Dominators and post-dominators in CFGs
• Loops in CFGs
“Intermediate” Program Representations: ASTs and Three-Address Code

• AST is a high-level IR
  – Close to the source language
  – Suitable for tasks such as type checking

• Three-address code is a lower-level IR
  – Closer to the target language (i.e., assembly code), but machine-independent
  – Suitable for tasks such as code generation/optimization

• Basic ideas
  – A small number of simple instructions: e.g. \( x = y \text{ op } z \)
  – A number of compiler-generated temporary variables
    \[ a = b + c + d; \] in source code \( \rightarrow t = b + c; a = t + d; \)
  – Simple flow of control – conditional and unconditional jumps to labeled statements (no while-do, switch, …)
Addresses and Instructions

• “Address”: a program variable, a constant, or a compiler-generated temporary variable

• Instructions
  – \( x = y \, \text{op} \, z \): binary operator \( \text{op} \)
  – \( x = \text{op} \, y \): unary operator \( \text{op} \)
  – \( x = y \): copy instruction
  – Flow-of-control (more later ...)
  – Each instruction contains at most three “addresses”
    • Thus, three-address code

• This looks very similar to the assembly language we discussed in the code generation examples
Examples of Three-Address Code

\( x = y; \) in the source code produces one three-address instruction

Left: a pointer to the symbol table entry for \( x \)
Right: a pointer to the symbol table entry for \( y \)
For convenience, we will write this as \( x = y \)

\( x = - y; \) produces \( t1 = - y; x = t1; \)
\( x = y + z; \) produces \( t1 = y + z; x = t1; \)
\( x = y + z + w; \) produces \( t1 = y + z; t2 = t1 + w; x = t2; \)
\( x = y + - z; \) produces \( t1 = - z; t2 = y + t1; x = t2; \)
More Complex Expressions & Assignments

• All binary & unary operators are handled similarly
• We run into more interesting issues with
  – Expressions that have side effects
  – Arrays
• Example: in C, we can write \( x = y = z + z \): maybe it should be translated to \( t1 = z + z; y = t1; x = y \)?
  – How should we translate \( x = y = z++ + w \)? How about \( a[v = x++] = y = z++ + w \)? Or \( i = i++ + 1 \)? Or \( a[i++] = i \)?
  – Not discussed in this course; some details in CSE 5343
Flow of Control - Statements

Example: \( \text{if (} x < 100 \text{ || } x > 200 \text{ && } x \neq y \text{)} x = 0; \)
\( \text{if (} x < 100 \text{)} \text{goto L2;} \)
\( \text{if (} !(x > 200)) \text{goto L1;} \)
\( \text{if (} !(x \neq y)) \text{goto L1;} \)
\( \text{L2: } x = 0; \)
\( \text{L1: } \ldots \)

Instructions

– \text{goto L:} unconditional jump to the three-address instruction with label L
– \text{if (} x \text{ relop } y \text{) goto L:} x \text{ and } y \text{ are variables, temporaries, or constants; relop } \in \{ <, \leq, =, \neq, >, \geq \}
Control-Flow Graphs

• Control-flow graph (CFG) for a procedure/method
  – A node is a basic block: a single-entry-single-exit sequence of three-address instructions
  – An edge represents the potential flow of control from one basic block to another

• Uses of a control-flow graph
  – Inside a basic block: local code optimizations; done as part of the code generation phase
  – Across basic blocks: global code optimizations; done as part of the code optimization phase
  – Other aspects of code generation: e.g., global register allocation
Control-Flow Analysis

• Part 1: Constructing a CFG
• Part 2: Finding dominators and post-dominators
• Part 3: Finding loops in a CFG
  – What exactly is a loop? Cannot simply say “whatever CFG subgraph is generated by \textit{while}, \textit{do-while}, and \textit{for} statements” – need a general graph-theoretic definition
Part 1: Constructing a CFG

• Nodes: basic blocks; edges: possible control flow

• **Basic block**: maximal sequence of consecutive three-address instructions such that
  – The flow of control can enter only through the first instruction (i.e., no jumps to the middle of the block)
  – Can exit only at the last instruction (i.e., no jumps out of the middle of the block)

• Advantages of using basic blocks
  – Reduces the cost and complexity of compile-time analysis
  – Intra-BB optimizations are relatively easy
CFG Construction

• Given: the entire sequence of instructions
• First, find the **leaders** (starting instructions of all basic blocks)
  – The first instruction
  – The target of any conditional/unconditional jump
  – Any instruction that immediately follows a conditional or unconditional jump

• Next, find the **basic blocks**: for each leader, its basic block contains itself and all instructions up to (but not including) the next leader
Example

1. $i = 1$
2. $j = 1$
3. $t1 = 10 \times i$
4. $t2 = t1 + j$
5. $t3 = 8 \times t2$
6. $t4 = t3 - 88$
7. $a[t4] = 0.0$
8. $j = j + 1$
9. if ($j \leq 10$) goto (3)
10. $i = i + 1$
11. if ($i \leq 10$) goto (2)
12. $i = 1$
13. $t5 = i - 1$
14. $t6 = 88 \times t5$
15. $a[t6] = 1.0$
16. $i = i + 1$
17. if ($i \leq 10$) goto (13)

Note: this example sets array elements $a[i][j]$ to 0.0, for $1 \leq i, j \leq 10$ (instructions 1-11). It then sets $a[i][i]$ to 1.0, for $1 \leq i \leq 10$ (instructions 12-17). The array accesses in instructions 7 and 15 are done with offsets from the beginning of the array.
ENTRY

B1
 i = 1

B2
 j = 1

B3
 t1 = 10 * i
 t2 = t1 + j
 t3 = 8 * t2
 t4 = t3 - 88
 a[t4] = 0.0
 j = j + 1
 if (j <= 10) goto B3

B4
 i = i + 1
 if (i <= 10) goto B2

B5
 i = 1

B6
 t5 = i - 1
 t6 = 88 * t5
 a[t6] = 1.0
 i = i + 1
 if (i <= 10) goto B6

EXIT

Artificial ENTRY and EXIT nodes are often added for convenience.

There is an edge from B_p to B_q if it is possible for the first instruction of B_q to be executed immediately after the last instruction of B_p.
Single Exit Node

• Single-exit CFG
  – If there are multiple exits (e.g., multiple return statements), redirect them to the artificial EXIT node
  – Use an artificial return variable `ret`
    – `return expr;` becomes `ret = expr; goto exit;`

• It gets ugly with exceptions (e.g., Java exceptions)

• Common properties (we will always assume them in this class)
  – Every node is reachable from the entry node
  – The exit node is reachable from every node
    • Not always true: e.g., a server thread could be `while(true) ...`
Practical Considerations

• The usual data structures for graphs can be used
  – The graphs are sparse (i.e., have relatively few edges), so an adjacency list representation is the usual choice
  • Number of edges is at most 2 * number of nodes

• Nodes are basic blocks; edges are between basic blocks, not between instructions
  – Inside each node, some additional data structures for the sequence of instructions in the block (e.g., a linked list of instructions)
  – Often convenient to maintain both a list of successors (i.e., outgoing edges) and a list of predecessors (i.e., incoming edges) for each basic block
Part 2: Dominance

• A CFG node \( d \) dominates another node \( n \) if every path from ENTRY to \( n \) goes through \( d \)
  – Implicit assumption: every node is reachable from ENTRY (i.e., there is no dead code)
  – A dominance relation \( \text{dom} \subseteq \text{Nodes} \times \text{Nodes}: d \text{ dom } n \)
  – The relation is trivially reflexive: \( d \text{ dom } d \)

• Node \( m \) is the immediate dominator of \( n \) if
  – \( m \neq n \)
  – \( m \text{ dom } n \)
  – For any \( d \neq n \) such \( d \text{ dom } n \), we have \( d \text{ dom } m \)

• Every node has a unique immediate dominator
  – Except ENTRY, which is dominated only by itself
ENTRY

1

2

ENTRY dom n for any n
1 dom n for any n except ENTRY
2 does not dominate any other node
3 dom 3, 4, 5, 6, 7, 8, 9, 10, EXIT
4 dom 4, 5, 6, 7, 8, 9, 10, EXIT
5 does not dominate any other node
6 does not dominate any other node
7 dom 7, 8, 9, 10, EXIT
8 dom 8, 9, 10, EXIT
9 does not dominate any other node
10 dom 10, EXIT

Immediate dominators:
1 → ENTRY  2 → 1
3 → 1  4 → 3
5 → 4  6 → 4
7 → 4  8 → 7
9 → 8  10 → 8
EXIT → 10
A Few Observations

• Dominance is a **transitive** relation: \( a \ dom \ b \) and \( b \ dom \ c \) means \( a \ dom \ c \)

• Dominance is an **anti-symmetric** relation: \( a \ dom \ b \) and \( b \ dom \ a \) means that \( a \) and \( b \) must be the same
  – Reflexive, anti-symmetric, transitive: **partial order**

• If \( a \) and \( b \) are two dominators of some \( n \), either \( a \ dom \ b \) or \( b \ dom \ a \)
  – Therefore, \( dom \) is a **total order** for \( n \)’s dominator set
  – Corollary: for any acyclic path from ENTRY to \( n \), all dominators of \( n \) appear along the path, always in the same order; the last one is the immediate dominator
The parent of \( n \) is its immediate dominator.

The path from \( n \) to the root contains all and only dominators of \( n \).


Post-Dominance

• A CFG node $d$ **post-dominates** another node $n$ if every path from $n$ to EXIT goes through $d$
  ─ Implicit assumption: EXIT is reachable from every node
  ─ A relation $pdom \subseteq \text{Nodes} \times \text{Nodes}$: $d \ pdom \ n$
  ─ The relation is trivially reflexive: $d \ pdom \ d$

• Node $m$ is the **immediate post-dominator** of $n$ if
  ─ $m \neq n$; $m \ pdom \ n$; $\forall d \neq n$. $d \ pdom \ n \Rightarrow d \ pdom \ m$
  ─ Every $n$ has a unique immediate post-dominator

• Post-dominance on a CFG is equivalent to dominance on the reverse CFG (all edges reversed)

• **Post-dominator tree**: the parent of $n$ is its immediate post-dominator; root is EXIT
ENTRY does not post-dominate any other \( n \)

1 \( pdom \) ENTRY, 1, 9

2 does not post-dominate any other \( n \)

3 \( pdom \) ENTRY, 1, 2, 3, 9

4 \( pdom \) ENTRY, 1, 2, 3, 4, 9

5 does not post-dominate any other \( n \)

6 does not post-dominate any other \( n \)

7 \( pdom \) ENTRY, 1, 2, 3, 4, 5, 6, 7, 9

8 \( pdom \) ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9

9 does not post-dominate any other \( n \)

10 \( pdom \) ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

EXIT \( pdom \) \( n \) for any \( n \)

Immediate post-dominators:

ENTRY \( \rightarrow \) 1 
1 \( \rightarrow \) 3
2 \( \rightarrow \) 3 
3 \( \rightarrow \) 4
4 \( \rightarrow \) 7 
5 \( \rightarrow \) 7
6 \( \rightarrow \) 7 
7 \( \rightarrow \) 8
8 \( \rightarrow \) 10 
9 \( \rightarrow \) 1
10 \( \rightarrow \) EXIT
The parent of \( n \) is its immediate post-dominator.

The path from \( n \) to the root contains all and only post-dominators of \( n \).

Constructing the post-dominator tree: use any algorithm for constructing the dominator tree; just “pretend” that the edges are reversed.
Part 3: Loops in CFGs

- **Cycle**: sequence of edges that starts and ends at the same node
  - Example: 1 → 2 → 3 → 4 → 5

- **Strongly-connected (induced) subgraph**: each node in the subgraph is reachable from every other node in the subgraph
  - Example: 2, 3, 4, 5

- **Loop**: informally, a strongly-connected subgraph with a single entry point
  - Not a loop:
Back Edges and Natural Loops

- Back edge: a CFG edge \((n,h)\) where \(h\) dominates \(n\)
  - Easy to see that \(n\) and \(h\) belong to the same SCC

- Natural loop for a back edge \((n,h)\)
  - The set of all nodes \(m\) that can reach node \(n\) without
    going through node \(h\) (trivially, this set includes \(h\))
  - Easy to see that \(h\) dominates all such nodes \(m\)
  - Node \(h\) is the header of the natural loop

- Trivial algorithm to find the natural loop of \((n,h)\)
  - Mark \(h\) as visited
  - Perform depth-first search (or breadth-first) starting
    from \(n\), but follow the CFG edges in reverse direction
  - All and only visited nodes are in the natural loop
Immediate dominators:

\[
\begin{align*}
1 & \rightarrow \text{ENTRY} \\
4 & \rightarrow 3 \\
7 & \rightarrow 4 \\
10 & \rightarrow 8 \\
3 & \rightarrow 1 \\
5 & \rightarrow 4 \\
8 & \rightarrow 7 \\
9 & \rightarrow 8 \\
\end{align*}
\]

Back edges: \(4 \rightarrow 3, 7 \rightarrow 4, 8 \rightarrow 3, 9 \rightarrow 1, 10 \rightarrow 7\)

Loop(\(10 \rightarrow 7\)) = \{ 7, 8, 10 \}

Loop(\(7 \rightarrow 4\)) = \{ 4, 5, 6, 7, 8, 10 \}

\textit{Note: Loop}(\(10 \rightarrow 7\)) \subseteq \textit{Loop}(\(7 \rightarrow 4\))

Loop(\(4 \rightarrow 3\)) = \{ 3, 4, 5, 6, 7, 8, 10 \}

\textit{Note: Loop}(\(7 \rightarrow 4\)) \subseteq \textit{Loop}(\(4 \rightarrow 3\))

Loop(\(8 \rightarrow 3\)) = \{ 3, 4, 5, 6, 7, 8, 10 \}

\textit{Note: Loop}(\(8 \rightarrow 3\)) = \textit{Loop}(\(4 \rightarrow 3\))

Loop(\(9 \rightarrow 1\)) = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \}

\textit{Note: Loop}(\(4 \rightarrow 3\)) \subseteq \textit{Loop}(\(9 \rightarrow 1\))
Loops in the CFG

• Find all back edges; each target \( h \) of at least one back edge defines a loop \( L \) with \( \text{header}(L) = h \)

• \( \text{body}(L) \) is the union of the natural loops of all back edges whose target is \( \text{header}(L) \)
  – Note that \( \text{header}(L) \in \text{body}(L) \)

• Example: this is a single loop with header node 1

• For two CFG loops \( L_1 \) and \( L_2 \)
  – \( \text{header}(L_1) \) is different from \( \text{header}(L_2) \)
  – \( \text{body}(L_1) \) and \( \text{body}(L_2) \) are either disjoint, or one is a proper subset of the other (nesting – inner/outer)
Use Scenario: Loop-Invariant Code Motion

Motivation: avoid redundancy

\[ a = \ldots \]

\[ b = \ldots \]

\[ c = \ldots \]

\textit{start loop}

\[ d = a + b \]

Both instructions are \textit{loop-invariant}; let’s move them out

\[ e = c + d \]

\textit{end loop}
Code Transformation

• First, create a **preheader** for the loop

- Original CFG

- Modified CFG

• Next, move loop-invariant instructions into the preheader (but only if correctness conditions are satisfied)

• Need control flow analysis to identify loops and loop headers
One of Several Correctness Conditions

- The basic block that contains the loop-invariant instruction **must dominate all loop exit nodes**
  - i.e., all nodes that are sources of loop-exit edges: source node is in the loop, target node is not
  - This means that it is impossible to exit the loop before the instruction is executed

[Diagram]

- Node 6 is a **loop exit node**; 3 dominates 6, but 4 and 5 do not dominate 6
- Any loop-invariant instructions in 4 and 5 cannot be moved into a preheader
May Need an Enabling Pre-Transformation
• CFGs for \textbf{while} and \textbf{for} loops will not work
• Consider \texttt{while(y<0) \{ a = 1+2; y++; \}}

\begin{itemize}
  \item L1: if (y<0) goto L2;
  \item goto L3;
  \item L2: a = 1+2;
  \item y = y + 1;
  \item goto L1;
  \item L3: ...
\end{itemize}

\begin{itemize}
  \item \texttt{a = 1+2} does not dominate the exit node B1
  \item loop header is now B3 and \texttt{a = 1+2} dominates the exit node B5
\end{itemize}