Control-Flow Analysis and Loop Detection

Last time
- Speeding up data-flow analysis

Today
- Control-flow analysis
- Loops
- Identifying loops using dominators
- Reducibility

Why study control flow analysis?

Finding Loops
- most computation time is spent in loops
- to optimize them, we need to find them

Loop Optimizations
- Loop-invariant code hoisting
- Induction variable elimination
- Array bounds check removal
- Loop unrolling
- Parallelization
- ...

Identifying structured control flow
- can be used to speed up data-flow analysis

Context

Data-flow
- Flow of data values from defs to uses
- Could alternatively be represented as a data dependence

Control-flow
- Sequencing of operations
- Could alternatively be represented as a control dependence
- e.g., Evaluation of then-code and else-code depends on if-test,

Representing Control-Flow

High-level representation
- Control flow is implicit in an AST

Low-level representation:
- Use a Control-flow graph
  - Nodes represent statements
  - Edges represent explicit flow of control

Other options
- Control dependences in program dependence graph (PDG) [Ferrante87]
- Dependences on explicit state in value dependence graph (VDG) [Weise 94]
What Is Control-Flow Analysis?

Control-flow analysis discovers the flow of control within a procedure (e.g., builds a CFG, identifies loops)

Example

1. \( a := 0 \)
2. \( b := a * b \)
3. \( \text{L1: } c := b/d \)
4. \( \text{if } c < x \text{ goto L2} \)
5. \( e := b / c \)
6. \( f := e + 1 \)
7. \( \text{L2: } g := f \)
8. \( h := t - g \)
9. \( \text{if } e > 0 \text{ goto L3} \)
10. \( \text{goto L1} \)
11. \( \text{L3: return} \)

Loop Concepts

Loop: Strongly connected component of CFG with a single entry point (header)
Loop entry edge: Source not in loop & target in loop
Loop exit edge: Source in loop & target not in loop
Loop header node: Target of loop entry edge
Natural loop: Nodes with path to backedge without going through header.
Back edge: Target is loop header & source is in the loop
Loop tail node: Source of back edge
Loop preheader node: Single node that’s source of the loop entry edge
Nested loop: Loop whose header is inside another loop

The Value of Preheader Nodes

Not all loops have preheaders
- Sometimes it is useful to create them

Without preheader node
- There can be multiple entry edges

With single preheader node
- There is only one entry edge

Useful when moving code outside the loop
- Don’t have to replicate code for multiple entry edges
Identifying Loops

Why?
- Most execution time spent in loops, so optimizing loops will often give most benefit

Many approaches
- Interval analysis
- Exploit the natural hierarchical structure of programs
- Decompose the program into nested regions called intervals
- Structural analysis: a generalization of interval analysis
- Identify dominators to discover loops

We’ll look at the dominator-based approach

Identifying Natural Loops with Dominators

Back edges
- A back edge of a natural loop is one whose target dominates its source

Natural loop
- The natural loop of a back edge \((m \rightarrow n)\), where \(n\) dominates \(m\), is the set of nodes \(x\) such that \(n\) dominates \(x\) and there is a path from \(x\) to \(m\) not containing \(n\)

Example
- This loop has two entry points, \(c\) and \(d\)
- The target, \(c\), of the edge \((d \rightarrow c)\) does not dominate its source, \(d\), so \((d \rightarrow c)\) does not define a natural loop

Dominator Terminology

Dominator Terminology

Dominator Terminology

Computing Dominators

Input: Set of nodes \(N\) (in CFG) and an entry node \(s\)
Output: \(\text{Dom}[i] = \{\text{set of all nodes that dominate node } i\}\)

\[\text{Dom}[s] = \{s\}\]

for each \(n \in N - \{s\}\)
\[\text{Dom}[n] = N\]
repeat
change = false
for each \(n \in N - \{s\}\)
\[D = [n] \cup \left(\bigcup_{p \in \text{pred}[n]} \text{Dom}[p]\right)\]
if \(D = \text{Dom}[n]\)
change = true
\[\text{Dom}[n] = D\]
until change

x \in \text{Dom}(p)^+ \land x \in \text{Dom}(p)^+ \land x \in \text{Dom}(p) \Rightarrow x \in \text{Dom}(n)\]
### Computing Dominators (example)

**Input:** Set of nodes $N$ and an entry node $s$

**Output:** $\text{Dom}[i]$ = set of all nodes that dominate node $i$

- $\text{Dom}[s] = \{s\}$
- for each $n \in N - \{s\}$
  - $\text{Dom}[n] = N$
- repeat
  - change = false
  - for each $n \in N - \{s\}$
    - $D = \{n\} \cup (\bigcap_{p \in \text{pred}(n)} \text{Dom}[p])$
    - if $D \neq \text{Dom}[n]$
      - change = true
      - $\text{Dom}[n] = D$
  - until !change

<table>
<thead>
<tr>
<th>Initially</th>
<th>Finally</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Dom}[s] = {s}$</td>
<td>$\text{Dom}[q] = {q, s}$</td>
</tr>
<tr>
<td>$\text{Dom}[q] = {s}$</td>
<td>$\text{Dom}[r] = {r, s}$</td>
</tr>
<tr>
<td>$\text{Dom}[p] = {p, s}$</td>
<td>$\text{Dom}[n] = {n, p, s}$</td>
</tr>
</tbody>
</table>

### Reducibility

**Definition**

- A CFG is **reducible** (well-structured) if we can partition its edges into two disjoint sets, the forward edges and the back edges, such that
  - The forward edges form an acyclic graph in which every node can be reached from the entry node
  - The back edges consist only of edges whose targets dominate their sources

- A CFG is **reducible** if it can be converted into a single node using T1 and T2 transformations.

**Structured control-flow constructs give rise to reducible CFGs**

- Dominance useful in identifying loops
- Simplifies code transformations (every loop has a single header)
- Permits interval analysis and it is easy to calculate the CFG depth

### T1 and T2 transformations

**T1 transformation**
- remove self-cycles

| $a \rightarrow a$ |

**T2 transformation**
- if node $n$ has a unique predecessor $p$, then remove $n$ and make all the successors for $n$ be successors for $p$

| $a \rightarrow \{b, c\}$ | $a \rightarrow \{b, c\}$ |

### Handling Irreducible CFG's

**Node splitting**
- Can turn irreducible CFGs into reducible CFGs
Why Go To All This Trouble?

Modern languages provide structured control flow
- Shouldn’t the compiler remember this information rather than throw it away and then re-compute it?

Answers?
- We may want to work on the binary code in which case such information is unavailable
- Most modern languages still provide a `goto` statement
- Languages typically provide multiple types of loops. This analysis lets us treat them all uniformly
- We may want a compiler with multiple front ends for multiple languages; rather than translate each language to a CFG, translate each language to a canonical LIR, and translate that representation once to a CFG

Next Time

Assignments
- Project 2 due: the writeup is IMPORTANT

Reading
- Ch 18.2

Lecture
- Loop invariant code motion

Concepts

Control-flow analysis
Control-flow graph (CFG)
Loop terminology
Identifying loops
Dominator
Reducibility