Control-Flow Analysis and Loop Detection

Last time
- Speeding up data-flow analysis

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

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,

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

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

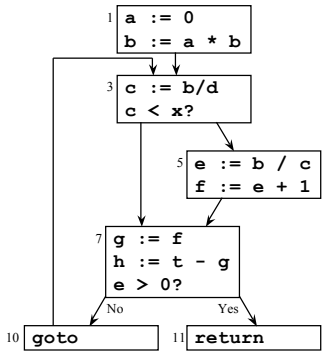
Other options
- Control dependences in program dependence graph (PDG) [Ferrante87]
- Dependences on explicit state in value dependence graph (VDG) [Weise 94]

What Is Control-Flow Analysis?

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

Example

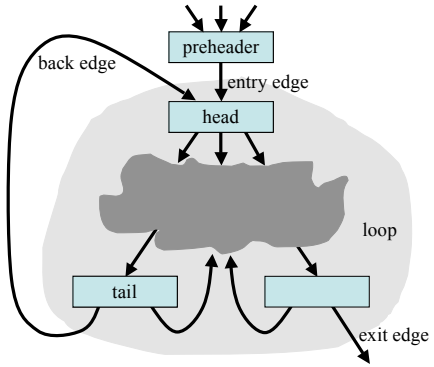
```
1       a := 0
2       b := a * b
3  L1:  c := b/d
4       if c < x goto L2
5       e := b / c
6       f := e + 1
7  L2:  g := f
8       h := t - g
9       if e > 0 goto L3
10 goto L1
11 L3:  return
```



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

Picturing Loop Terminology



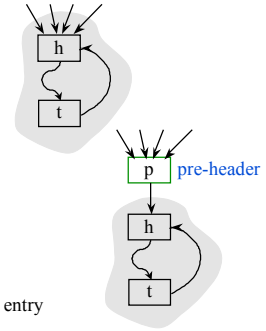
The Value of Preheader Nodes

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

Dominator Terminology

Dominators

d **dom** i if all paths from entry to node i include d

Strict dominators

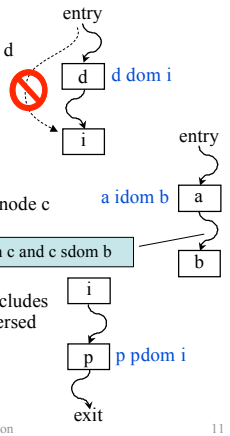
d **sdom** i if d dom i and d ≠ i

Immediate dominators

a **idom** b if a **sdom** b and there does not exist a node c such that c ≠ a, c ≠ b, a dom c, and c dom b

Post dominators

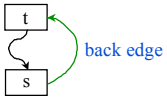
p **pdom** i if every possible path from i to exit includes p (p dom i in the flow graph whose arcs are reversed and entry and exit are interchanged)



Identifying Natural Loops with Dominators

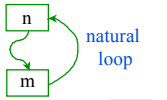
Back edges

A **back edge** of a natural loop is one whose target dominates its source



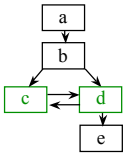
Natural loop

The **natural loop** of a back edge (m→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

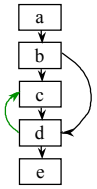


Example

This loop has two entry points, c and d



The target, c, of the edge (d→c) does not dominate its source, d, so (d→c) does not define a natural loop



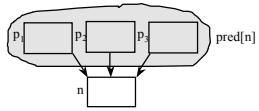
Computing Dominators

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

```
Dom[s] = {s}
for each n ∈ N – {s}
    Dom[n] = N
repeat
    change = false
    for each n ∈ N – {s}
        D = {n} ∪ (∩_{p∈pred(n)} Dom[p])
        if D ≠ Dom[n]
            change = true
            Dom[n] = D
until !change
```

Key Idea

If a node dominates all predecessors of node n, then it also dominates node n

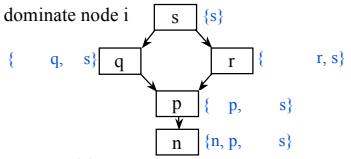


$x \in Dom(p_1) \wedge x \in Dom(p_2) \wedge x \in Dom(p_3) \Rightarrow x \in Dom(n)$

Computing Dominators (example)

Input: Set of nodes N and an entry node s
Output: Dom[i] = set of all nodes that dominate node i

Dom[s] = {s}
for each n ∈ N – {s}
  Dom[n] = N
repeat
  change = false
  for each n ∈ N – {s}
    D = {n} ∪ ($\cap_{p \in pred(n)}$ Dom[p])
    if D ≠ Dom[n]
      change = true
      Dom[n] = D
until !change



Initially
  Dom[s] = {s}
  Dom[q] = {n, p, q, r, s}. . .
Finally
  Dom[q] = {q, s}
  Dom[r] = {r, s}
  Dom[p] = {p, s}
  Dom[n] = {n, p, s}

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.

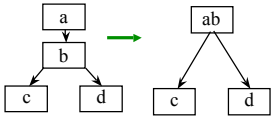
Structured control-flow constructs give rise to reducible CFGs

Value of reducibility
- 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

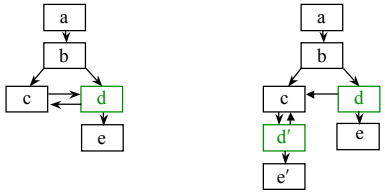
T1 transformation
- remove self-cycles

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



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

Concepts

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

Next Time

Assignments

- Project 2 due: the writeup is IMPORTANT

Reading

- Ch 18.2

Lecture

- Loop invariant code motion