Control-Flow Analysis

Last time
- Introduction to data-flow analysis (liveness)

Today
- Control-flow analysis
  - Building basic blocks
  - Building control-flow graphs (CFGs)
  - Loops

Context

Data-flow
- Flow of data values from defs to uses

Control-flow
- Sequencing of operations
- *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

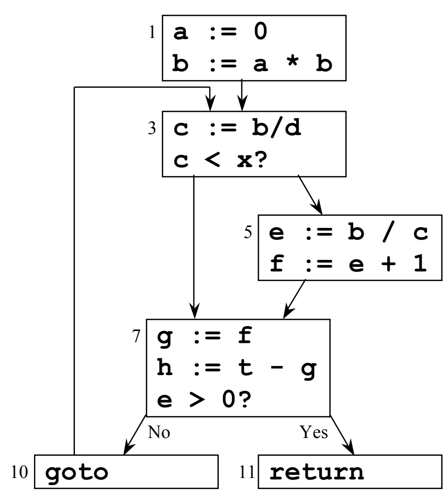
Other options
- Control dependences in program dependence graph (PDG) [Ferrante87]
- Dependences on explicit state in value dependence graph (VDG) [Weise 94]

What Is Control-Flow Analysis?

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

Example

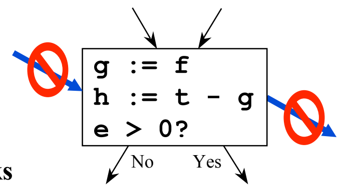
```
        a := 0
        b := a * b
L1:     c := b/d
        if c < x goto L2
        e := b / c
        f := e + 1
L2:     g := f
        h := t - g
        if e > 0 goto L3
goto L1
L3:  return
```



Basic Blocks

Definition

– A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



Building basic blocks

– Identify **leaders**
  – The first instruction in a procedure, or
  – The target of any branch, or
  – An instruction immediately following a branch (implicit target)

– Gobble all subsequent instructions until the next leader

Algorithm for Building Basic Blocks

Input: List of n instructions (instr[i] = i^{th} instruction)

Output: Set of leaders & list of basic blocks
(block[x] is block with leader x)

```
leaders = {1}                    // First instruction is a leader
for i = 1 to n                   // Find all leaders
    if instr[i] is a branch
        leaders = leaders ∪ set of potential targets of instr[i]
foreach x ∈ leaders
    block[x] = { x }
    i = x+1                      // Fill out x's basic block
    while i ≤ n and i ∉ leaders
        block[x] = block[x] ∪ { i }
        i = i + 1
```

Building Basic Blocks Example

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
```

Leaders?

– {1, 3, 5, 7, 10, 11}

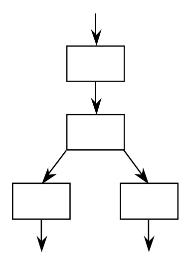
Blocks?

– {1, 2}
– {3, 4}
– {5, 6}
– {7, 8, 9}
– {10}
– {11}

Extended Basic Blocks

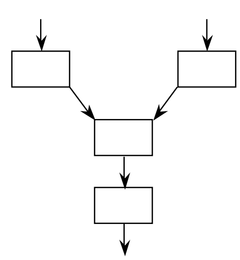
Extended basic blocks

– A maximal sequence of instructions that has no merge points in it (except perhaps in the leader)
– Single entry, multiple exits



How are these useful?

– Increases the scope of any local analysis or transformation that "flows forwards" (*e.g.,* copy propagation, register renaming, instruction scheduling)



Reverse extended basic blocks

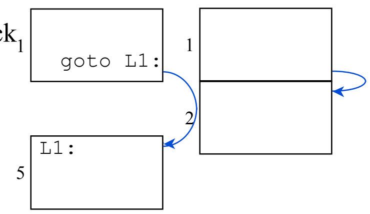
– Useful for "backward flow" problems

Building a CFG from Basic Blocks

Construction

– Each CFG node represents a basic block
– There is an edge from node i to j if
  – Last statement of block i branches to the first statement of j, or
  – Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)

Input: A list of m basic blocks (block)
Output: A CFG where each node is a basic block



for i = 1 **to** m
    x = last instruction of block[i]
    if instr x is a branch
        for each target (to block j) of instr x
            create an edge from block i to block j
    if instr x is not an unconditional branch
        create an edge from block i to block i+1

Details

Multiple edges between two nodes

```
            . . .
      if (a<b) goto L2
L2:  . . .
```

– Combine these edges into one edge

Unreachable code

```
      . . .
      goto L1
L0:   a = 10
L1:   . . .
```

– Perform DFS from entry node
– Mark each basic block as it is visited
– Unmarked blocks are unreachable

Unreachable code vs. dead code?

Challenges

When is CFG construction more complex?

Languages with branch delay slots

- *e.g.*, Sparc

```
ba loop
sub %l1, %l2, %l3
```

Languages where branch targets may be unknown

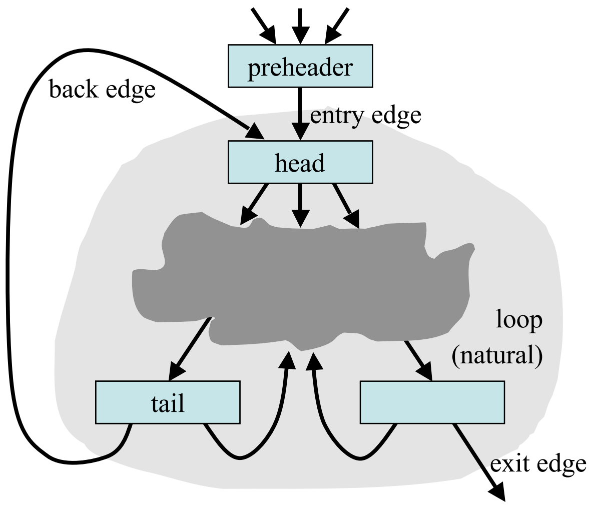
- *e.g.*, Executable code

```
ld $8, 104($7)
jmp $8
```


Loop Concepts

Loop:	Strongly connected component of CFG
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:	Loop with only a single loop 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
Reducible flow graph:	CFG whose loops are all natural loops

Picturing Loop Terminology



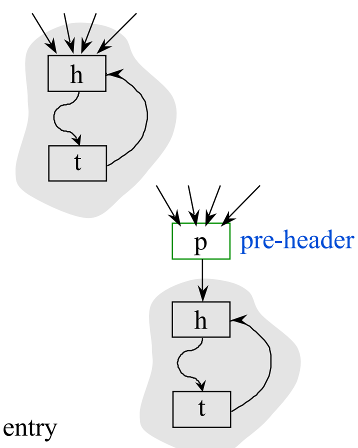
The Value of Preheader Nodes

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

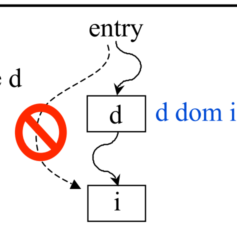
– Interval analysis
  – Exploit the natural hierarchical structure of programs
  – Decompose the program into nested regions called intervals
– Structural analysis: a generalization of interval analysis
– Identify **dominators** to discover loops

We'll look at the dominator-based approach

Dominator Terminology

Dominators

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

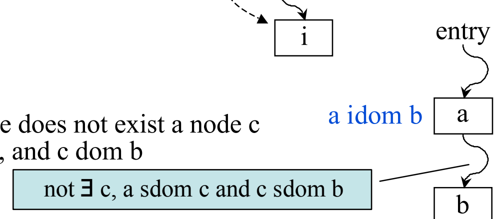


Strict dominators

d **sdom** i if d dom i and d ≠ i

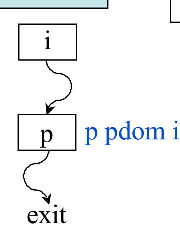
Immediate dominators

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

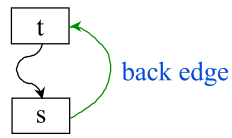
p **pdom** i if every possible path from i to exit includes p (p dom i in the flow graph whose arcs are reversed and entry and exit are interchanged)



Identifying Natural Loops with Dominators

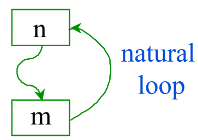
Back edges

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



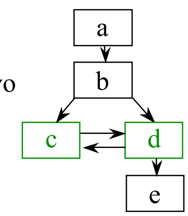
Natural loop

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

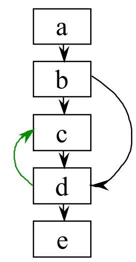


Example

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



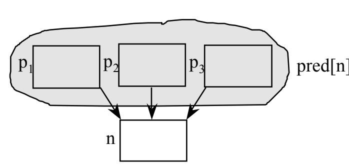
Computing Dominators

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}
        $D = \{n\} \cup (\bigcap_{p \in pred(n)} Dom[p])$
        if D ≠ Dom[n]
            change = true
            Dom[n] = D
until !change

Key Idea

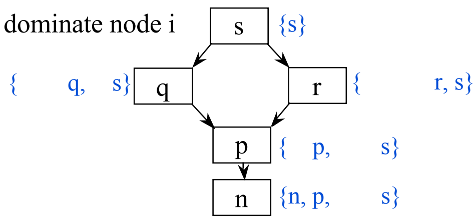
If a node dominates all predecessors of node n, then it also dominates node n



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

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
- Non-natural loops ⇔ irreducibility

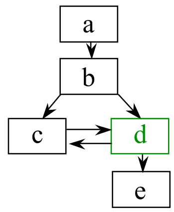
Structured control-flow constructs give rise to reducible CFGs

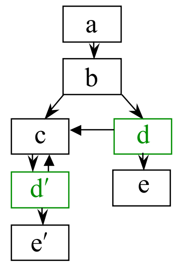
Value of reducibility
- Dominance useful in identifying loops
- Simplifies code transformations (every loop has a single header)
- Permits interval analysis

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

Basic blocks

- Computing basic blocks
- Extended basic blocks

Control-flow graph (CFG)

Loop terminology

Identifying loops

Dominators

Reducibility

Next Time

Assignments

- Project 1 due: the writeup is IMPORTANT

Reading

- Muchnick Ch. 8.2-8.4

Lecture

- Generalizing data-flow analysis