

## 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

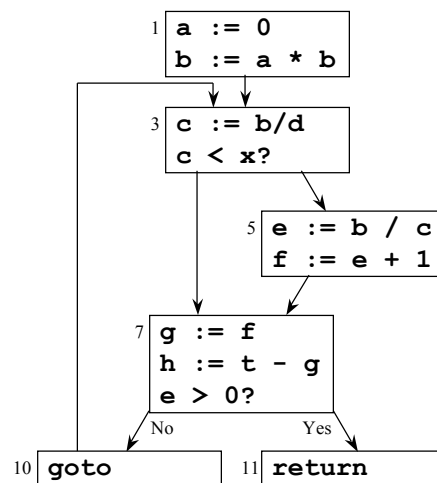
- Control dependences in program dependence graph (PDG) [Ferrante87]
- Dependences on explicit state in value dependence graph (VDG) [Weise 94]

## What Is Control-Flow Analysis?

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

### Example

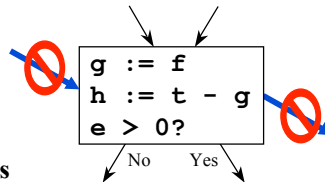
```
1      a := 0
2      b := a * b
3  L1:  c := b/d
4      if c < x goto L2
5      e := b / c
6      f := e + 1
7  L2:  g := f
8      h := t - g
9      if e > 0 goto L3
10     goto L1
11  L3:  return
```



## 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 ( $\text{instr}[i] = i^{\text{th}}$  instruction)

**Output:** Set of leaders & list of basic blocks  
( $\text{block}[x]$  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  $\cup$  set of potential targets of instr[i]
foreach x  $\in$  leaders
  block[x] = { x }
  i = x+1 // Fill out x's basic block
  while i  $\leq$  n and i  $\notin$  leaders
    block[x] = block[x]  $\cup$  { i }
    i = i + 1
```

## Building Basic Blocks Example

### Example

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

### Leaders?

– {1, 3, 5, 7, 10, 11}

### Blocks?

– {1, 2}

– {3, 4}

– {5, 6}

– {7, 8, 9}

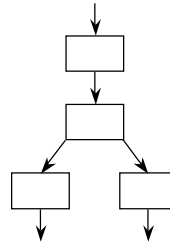
– {10}

– {11}

## Extended Basic Blocks

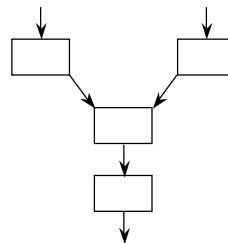
### Extended basic blocks

- 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[ $i$ ]

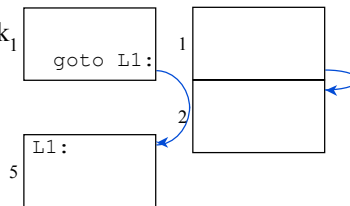
**if** instr  $x$  is a branch

**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  $i+1$



CS553 Lecture

Control-Flow Analysis

10

## 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?

CS553 Lecture

Control-Flow Analysis

11

## Challenges

---

**When is CFG construction more complex?**

**Languages with branch delay slots**

– *e.g.*, Sparc

```
ba loop
sub %11, %12, %13
```

**Languages where branch targets may be unknown**

– *e.g.*, Executable code

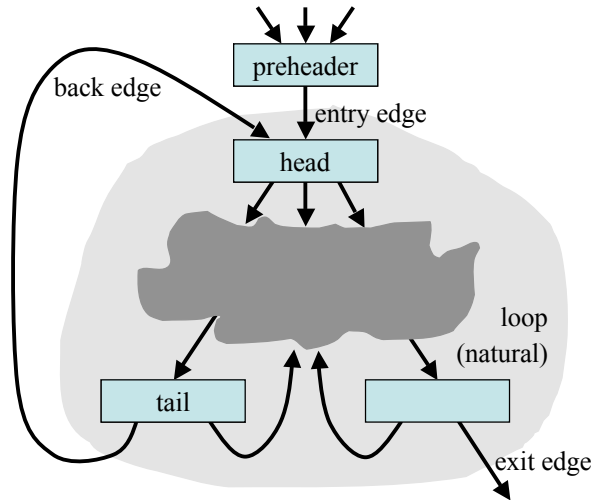
```
ld $8, 104($7)
jmp $8
```

## Loop Concepts

---

<b>Loop:</b>	Strongly connected component of CFG
<b>Loop entry edge:</b>	Source not in loop & target in loop
<b>Loop exit edge:</b>	Source in loop & target not in loop
<b>Loop header node:</b>	Target of loop entry edge
<b>Natural loop:</b>	Loop with only a single loop header
<b>Back edge:</b>	Target is loop header & source is in the loop
<b>Loop tail node:</b>	Source of back edge
<b>Loop preheader node:</b>	Single node that's source of the loop entry edge
<b>Nested loop:</b>	Loop whose header is inside another loop
<b>Reducible flow graph:</b>	CFG whose loops are all natural loops

## Picturing Loop Terminology



CS553 Lecture

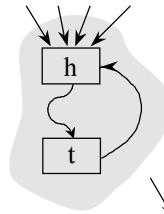
Control-Flow Analysis

14

## The Value of Preheader Nodes

### Not all loops have preheaders

- Sometimes it is useful to create them

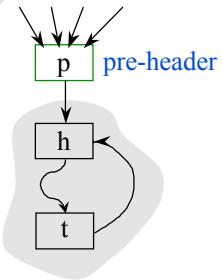


### Without preheader node

- 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

CS553 Lecture

Control-Flow Analysis

15

## Identifying Loops

### Why?

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

### Many approaches

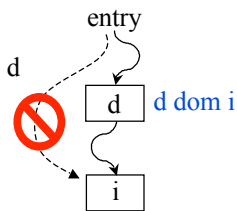
- Interval analysis
  - Exploit the natural hierarchical structure of programs
  - Decompose the program into nested regions called intervals
- Structural analysis: a generalization of interval analysis
- Identify **dominators** to discover loops

### We'll look at the dominator-based approach

## Dominator Terminology

### Dominators

$d \text{ dom } i$  if all paths from entry to node  $i$  include  $d$



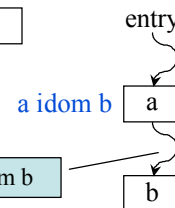
### Strict dominators

$d \text{ sdom } i$  if  $d \text{ dom } i$  and  $d \neq i$

### Immediate dominators

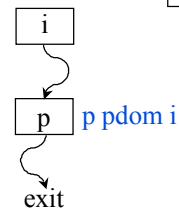
$a \text{ idom } b$  if  $a \text{ sdom } b$  and there does not exist a node  $c$  such that  $c \neq a$ ,  $c \neq b$ ,  $a \text{ dom } c$ , and  $c \text{ dom } b$

not  $\exists c, a \text{ sdom } c$  and  $c \text{ sdom } b$



### Post dominators

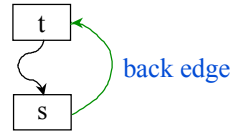
$p \text{ pdom } i$  if every possible path from  $i$  to exit includes  $p$  ( $p \text{ dom } i$  in the flow graph whose arcs are reversed and entry and exit are interchanged)



## Identifying Natural Loops with Dominators

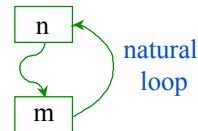
### Back edges

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



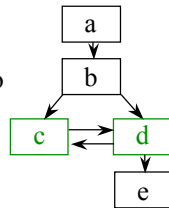
### Natural loop

The **natural loop** of a back edge ( $m \rightarrow n$ ), 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  $n$

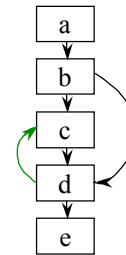


### Example

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



CS553 Lecture

Control-Flow Analysis

18

## Computing Dominators

**Input:** Set of nodes  $N$  (in CFG) 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

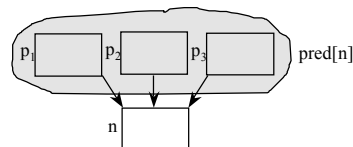
$\text{Dom}[n] = D$

**until** !change

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

### Key Idea

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



CS553 Lecture

Control-Flow Analysis

19

## 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**  $n \in N - \{s\}$

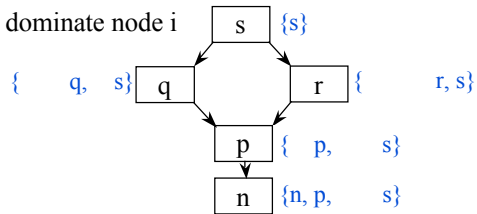
$D = \{n\} \cup (\bigcap_{p \in \text{pred}(n)} \text{Dom}[p])$

**if**  $D \neq \text{Dom}[n]$

change = true

$\text{Dom}[n] = D$

**until** !change



**Initially**

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

$\text{Dom}[q] = \{n, p, q, r, s\} \dots$

**Finally**

$\text{Dom}[q] = \{q, s\}$

$\text{Dom}[r] = \{r, s\}$

$\text{Dom}[p] = \{p, s\}$

$\text{Dom}[n] = \{n, p, s\}$

## 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  $\Leftrightarrow$  irreducibility

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

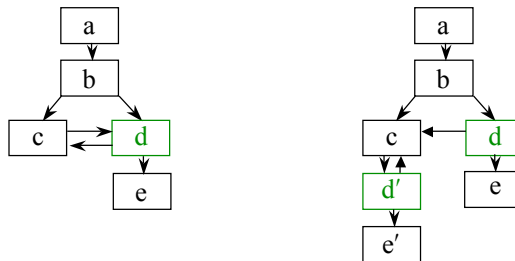
### Value of reducibility

- 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