

## Control-Flow Analysis and Loop Detection

---

### Last time

- PRE

### Today

- Control-flow analysis
- Loops
- Identifying loops using dominators
- Reducibility
- Using loop identification to identify induction variables

## 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

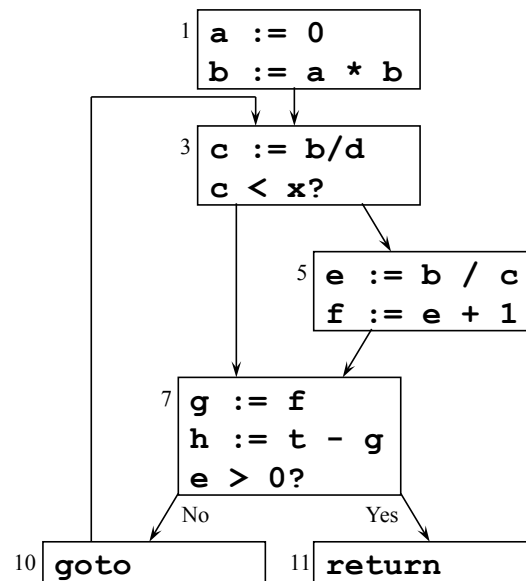
- Control dependences in program dependence graph (PDG) [Ferrante87]
- Dependences on explicit state in value dependence graph (VDG) [Weise 94]

## 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 subgraph 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. Dominates all nodes in loop.

**Back edge:** Target is loop header & source is in the loop

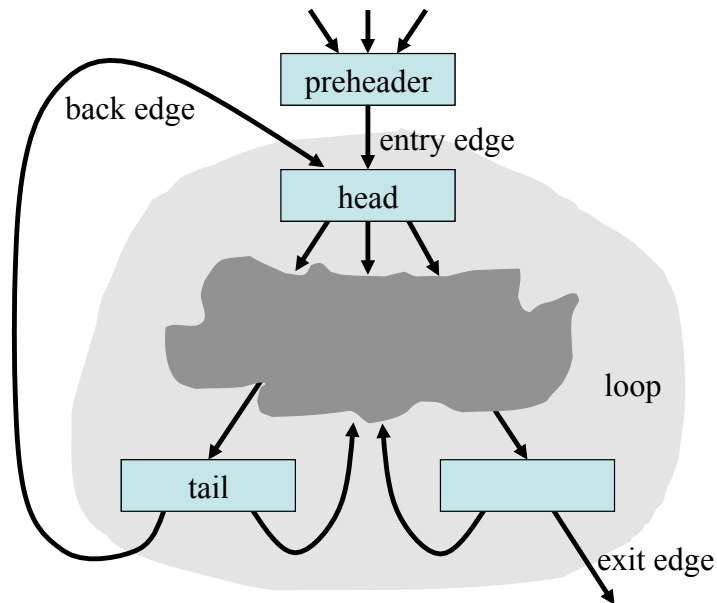
**Natural loop:** Associated with each back edge. Nodes dominated by header and with path to back edge without going through header

**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

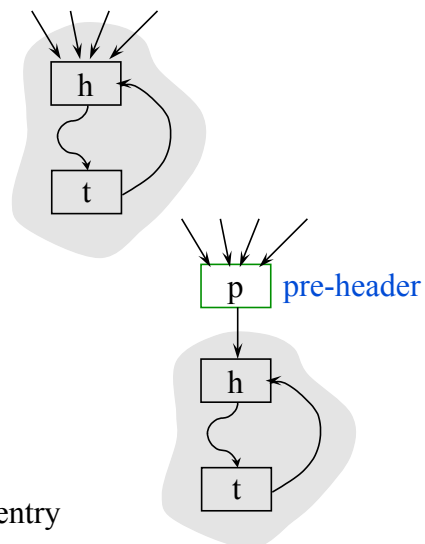
- There can be multiple entry edges

### 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

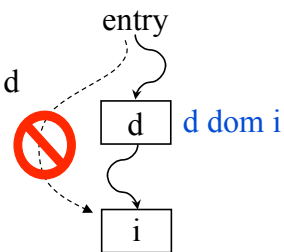
- Interval analysis
  - Exploit the natural hierarchical structure of programs
  - Decompose the program into nested regions called intervals
- Structural analysis: a generalization of interval analysis
- Identify **dominators** to discover loops

### We'll focus on the dominator-based approach

## Dominator Terminology

### Dominators

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



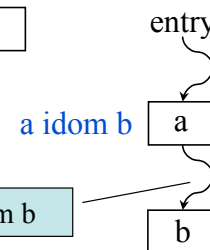
### Strict dominators

$d$  **sdom**  $i$  if  $d$  **dom**  $i$  and  $d \neq i$

### Immediate dominators

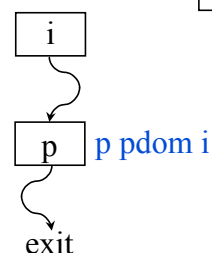
$a$  **idom**  $b$  if  $a$  **sdom**  $b$  and there does not exist a node  $c$  such that  $c \neq a$ ,  $c \neq b$ ,  $a$  **dom**  $c$ , and  $c$  **dom**  $b$

not  $\exists c, a$  **sdom**  $c$  and  $c$  **sdom**  $b$



### Post dominators

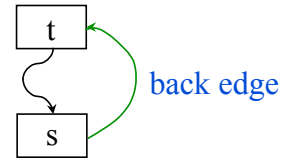
$p$  **pdom**  $i$  if every possible path from  $i$  to exit includes  $p$  ( $p$  **dom**  $i$  in the flow graph whose arcs are reversed and entry and exit are interchanged)



## Identifying Natural Loops with Dominators

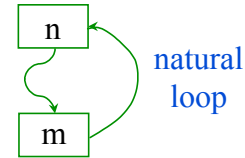
### Back edges

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



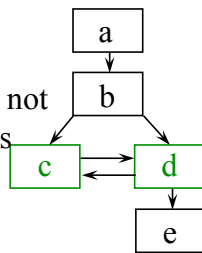
### Natural loop

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

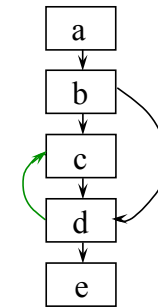


### Example

SCC with  $c$  and  $d$  not a loop because has two entry points



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



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

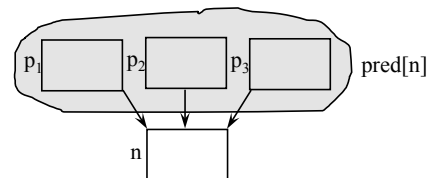
change = true

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

**until** !change

### Key Idea

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



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

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

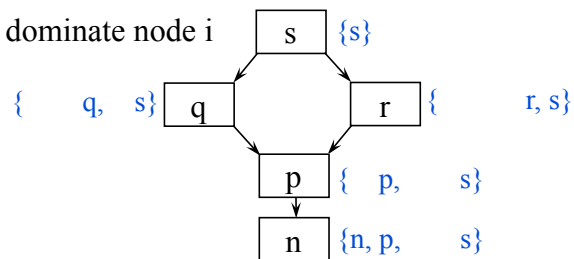
$D = \{n\} \cup (\bigcap_{p \in \text{pred}(n)} \text{Dom}[p])$

**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

### Definitions

- 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

---

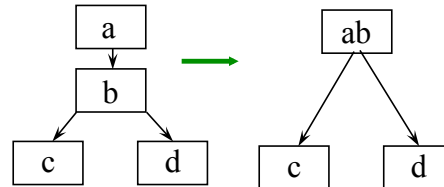
### T1 transformation

- 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  $p$

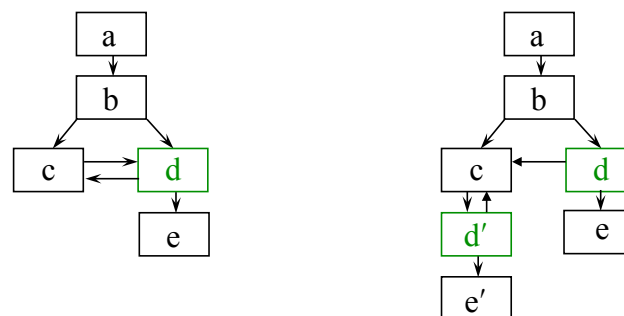


## Handling Irreducible CFG's

---

### 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

## Induction variables

---

### Induction variable identification

- Induction variables
  - Variables whose values form an arithmetic progression

### Why bother?

- Useful for strength reduction, induction variable elimination, loop transformations, and automatic parallelization

### Simple approach

- Search for statements of the form,  $i = i + c$
- Examine reaching definitions to make sure there are no other defs of  $i$  in the loop
- Does not catch all induction variables. Examples?

## Example Induction Variables

---

```
s = 0;
for (i=0; i<N; i++)
    s += a[i];
```

## Induction Variable Identification

---

### Types of Induction Variables

- **Basic** induction variables (eg. loop index)
  - Variables that are defined once in a loop by a statement of the form,  $i=i+c$  (or  $i=i-c$ ), where  $c$  is a constant integer or loop invariant
- **Derived** induction variables
  - Variables that are defined once in a loop as a linear function of another induction variable
    - $k = j + c_1$  or
    - $k = c_2 * j$  where  $c_1$  and  $c_2$  are loop invariant

## Induction Variable Triples

---

**Each induction variable  $k$  is associated with a triple  $(i, c_1, c_2)$**

- $i$  is a basic induction variable
- $c_1$  and  $c_2$  are constants such that  $k = c_1 + c_2 * i$  when  $k$  is defined
- $k$  belongs to the family of  $i$

**Basic induction variables**

- their triple is  $(i, 0, 1)$
- $i = 0 + 1*i$  when  $i$  is defined

## Algorithm for Identifying Loop Invariant Code

---

**Input:** A loop  $L$  consisting of basic blocks. Each basic block contains a sequence of 3-address instructions. We assume reaching definitions have been computed.

**Output:** The set of instructions that compute the same value each time through the loop

**Informal Algorithm:**

1. Mark “invariant” those statements whose operands are either
  - Constant
  - Have all reaching definitions outside of  $L$
2. Repeat until a fixed point is reached: mark “invariant” those unmarked statements whose operands are either
  - Constant
  - Have all reaching definitions outside of  $L$
  - Have exactly one reaching definition and that definition is in the set marked “invariant”

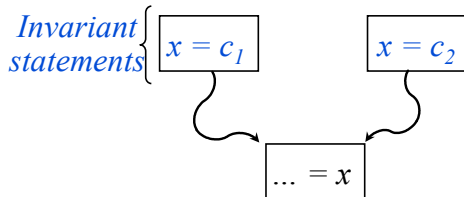
Is this last condition too strict?

## Algorithm for Identifying Loop Invariant Code (cont)

---

### Is the Last Condition Too Strict?

- No
- If there is more than one reaching definition for an operand, then neither one dominates the operand
- If neither one dominates the operand, then the value can vary depending on the control path taken, so the value is not loop invariant



## Algorithm for Identifying Induction Variables

---

**Input:** A loop  $L$  consisting of 3-address instructions, reaching defs, and loop-invariant information.

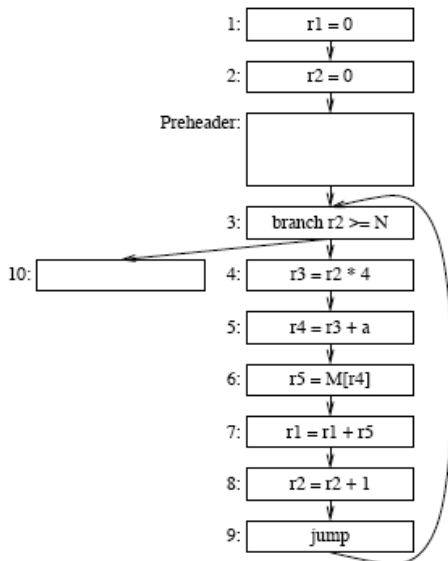
**Output:** A set of induction variables, each with an associated triple.

**Algorithm:**

1. For each stmt in  $L$  that matches the pattern  $i = i + c$  or  $i = i - c$  create the triple  $(i, 0, 1)$ .
2. Derived induction variables: For each stmt of  $L$ ,
  - If the stmt is of the form  $k = j + c_1$  or  $k = j * c_2$
  - and  $j$  is an induction variable with the triple  $(x, p, q)$
  - and  $c_1$  and  $c_2$  are loop invariant
  - and  $k$  is only defined once in the loop
  - and if  $j$  is a derived induction variable belonging to the family of  $i$  then
    - the only def of  $j$  that reaches  $k$  must be in  $L$
    - and no def of  $i$  must occur on any path between the definition of  $j$  and  $k$
  - then create the triple  $(x, p + c_1, q)$  for  $k = j + c_1$  or  $(x, p * c_2, q * c_2)$  for  $k = j * c_2$

## Example: Induction Variable Detection

---



Picture from Prof David Walker's CS320 slides

## Algorithm for Strength Reduction

---

**Input:** A loop  $L$  consisting of 3-address instructions and induction variable triples.

**Output:** A modified loop with a new preheader.

**Algorithm:**

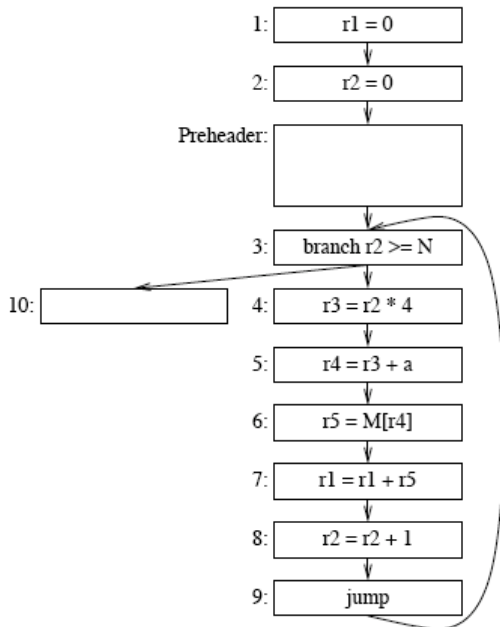
1. For each derived induction variable  $j$  with triple  $(i, p, q)$ 
  - create a new  $j'$
  - after each definition of  $i$  in  $L$ , where  $i = i + c$  insert  $j' = j' + t$
  - put computation  $t = q * c$  in preheader
  - initialize  $j'$  at the end of the preheader to  $j' = p + q * i$
  - replace the definition of  $j$  with  $j = j'$

**Note:**

- $j'$  also has triple  $(i, p, q)$
- multiplication has been moved out of the loop

## Example: Strength Reduction

---



Picture from Prof David Walker's CS320 slides

## Algorithm for Induction Variable Elimination

---

**Input:** A loop  $L$  consisting of 3-address instructions, reaching definitions, loop-invariant information, and live-variable information.

**Output:** A revised loop.

**Algorithm:**

1. Apply copy propagation followed by dead code elimination to eliminate copies introduced by strength-reduction.
2. Remove any induction variable definitions where the induction variable is only used and defined within that definition. (useless vars)
3. For each induction variable  $i$  (almost useless vars)
  - If only uses are to compute other induction variables in its family and in conditional branches, then mark as eliminated
    - Use a triple  $(j, c, d)$  in family associated with variable  $k$
    - Modify each conditional involving  $i$  so that  $k$  is used instead, uses relationships set up with triples
  - Delete all assignments to the eliminated induction variable

## Concepts

---

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

#### **Induction variable detection and elimination require loop identification**

- Induction variable detection uses
  - strength reduction and induction variable elimination
  - data dependence analysis, which can then be used for parallelization
- Strength reduction
  - removes multiplications
  - the definition for some derived induction variables no longer depend directly on a basic induction variable
- Induction variable elimination
  - removes unnecessary induction variables

## Next Time

---

### **Lecture**

- SSA