Recap

**Last Time**
- Algorithm to compute dominators
- Did you understand it?

**Exercise**
- Can we start with the empty set and grow the set of dominators?

---

### 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

---

January 28, 2015

Control Flow Analysis
Introduction to Data-flow Analysis

Last Time
– Control flow analysis

Today
– Introduce iterative data-flow analysis
  – Liveness analysis
  – Introduce other useful concepts

Data-flow Analysis

Idea
– Data-flow analysis derives information about the dynamic behavior of a program by only examining the static code

Example
– How many variables does this code have?
– How many registers do we need for these variables?
– Easy bound: 3

```
1   a := 0
2   L1: b := a + 1
3   c := c + b
4   a := b * 2
5   if a < 9 goto L1
6   return c
```
Data-flow Analysis

Idea
– Data-flow analysis derives information about the dynamic behavior of a program by only examining the static code

Example
– Better answer is found by considering the dynamic requirements of the program

```
1  a := 0
2  L1: b := a + 1
3  c := c + b
4  a := b * 2
5  if a < 9 goto L1
6  return c
```

February 2, 2015

Liveness Analysis

Definition
– A variable is live at a particular point in the program if its value at that point will be used in the future (dead, otherwise).
\[ \therefore \] To compute liveness at a given point, we need to look into the future

Example
– Is b live on line 2?
– Is b live on line 4?

```
1  a := 0
2  L1: b := a + 1
3  c := c + b
4  a := b * 2
5  if a < 9 goto L1
6  return c
```
Motivation for Liveness Analysis

Register Allocation

- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (i.e., never simultaneously live).
  \[ \therefore \] Register allocation uses liveness information

Liveness by Example

What is the live range of \( b \)?

- Variable \( b \) is read in statement 4, so \( b \) is live on the \((3 \rightarrow 4)\) edge
- Since statement 3 does not assign into \( b \), \( b \) is also live on the \((2 \rightarrow 3)\) edge
- Statement 2 assigns \( b \), so any value of \( b \) on the \((1 \rightarrow 2)\) and \((5 \rightarrow 2)\) edges are not needed, so \( b \) is dead along these edges

\[ b \)'s live range is \((2 \rightarrow 3 \rightarrow 4)\]
Exercise: Liveness by Example

Live range of a
- a is live from (1→2) and again from (4→5→2)
- a is dead from (2→3→4)

Live range of b
- b is live from (2→3→4)

Live range of c
- c is live from (entry→1→2→3→4→5→2, 5→6)
  a and b are never simultaneously live, so they can share a register

Control Flow Graphs (CFGs)

Simplification
- For now, we will use CFGs in which nodes represent program statements rather than basic blocks

Example
1  a := 0
2  b := a + 1
3  c := c + b
4  a := b * 2
5  if a < 9 goto L1
6  return c

1  a := 0
2  b := a + 1
3  c := c + b
4  a := b * 2
5  if a < 9 goto L1
6  return c
Terminology

Flow Graph Terms
- A CFG node has **out-edges** that lead to **successor** nodes and **in-edges** that come from **predecessor** nodes.
- \( \text{pred}[n] \) is the set of predecessors of node \( n \).
- \( \text{succ}[n] \) is the set of successors of node \( n \).

Examples
- Out-edges of node 5: \((5 \rightarrow 6)\) and \((5 \rightarrow 2)\)
- \( \text{succ}[5] = \{2, 6\} \)
- \( \text{pred}[5] = \{4\} \)
- \( \text{pred}[2] = \{1, 5\} \)

Defs and Uses

Def (or definition)
- An **assignment** of a value to a variable
- \( \text{def}[v] \) = set of CFG nodes that define variable \( v \)
- \( \text{def}[n] \) = set of variables that are defined at node \( n \)

Use
- A **read** of a variable’s value
- \( \text{use}[v] \) = set of CFG nodes that use variable \( v \)
- \( \text{use}[n] \) = set of variables that are used at node \( n \)
More precise definition of liveness

- A variable \( v \) is live on a CFG edge if
  
  1. \( \exists \) a directed path from that edge to a use of \( v \) (node in \( \text{use}[v] \)), and
  
  2. that path does not go through any def of \( v \) (no nodes in \( \text{def}[v] \))

The Flow of Liveness

Data-flow

- Liveness of variables is a property that flows through the edges of the CFG

Direction of Flow

- Liveness flows \textbf{backwards} through the CFG, because the behavior at future nodes determines liveness at a given node
  
  - Consider \textbf{a}
  
  - Consider \textbf{b}  

- Later, we’ll see other properties that flow \textbf{forward}
Two More Definitions

- A variable is **live-out** at a node if it is live just after the node’s computation.
- A variable is **live-in** at a node if it is live just before the node’s computation.

Liveness at Nodes (cont)

**Live-out**

- A variable is **live-out** at a node if it is live on any of that node’s out-edges.

**Live-in**

- How do we know if a variable is **live-in** at a node?
Computing Liveness

Rules for computing liveness

1. Generate liveness:
   If a variable is in use\([n]\), then it is live-in at node \(n\).

2. Push liveness across edges:
   If a variable is live-in at a node \(n\), then it is live-out at all nodes in \(\text{pred}[n]\).

3. Push liveness across nodes:
   If a variable is live-out at node \(n\) and not in \(\text{def}[n]\), then the variable is also live-in at \(n\).

Data-flow equations

1. \(\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])\)
2. \(\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]\)
3. \(\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]\)

Solving the Data-flow Equations

Algorithm

for each node \(n\) in CFG
\(\text{in}[n] = \emptyset; \text{out}[n] = \emptyset\) \{ initialize solutions \}
repeat
  for each node \(n\) in CFG
   \(\text{in}'[n] = \text{in}[n]\)
   \(\text{out}'[n] = \text{out}[n]\)
   \(\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])\)
   \(\text{out}[n] = \bigcup_{s \in \text{succ}[s]} \text{in}[s]\) \{ solve data-flow equations \}
until \(\text{in}'[n] = \text{in}[n]\) and \(\text{out}'[n] = \text{out}[n]\) for all \(n\) \{ test for convergence \}

This is iterative data-flow analysis (for liveness analysis)
Exercise: Compute the First Iteration

<table>
<thead>
<tr>
<th>node #</th>
<th>use</th>
<th>def</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>bc</td>
<td>b</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>c</td>
<td>bc</td>
<td>b</td>
<td>bc</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

Data-flow Equations for Liveness

\[
\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

\[
\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]
\]

Example (cont)

Data-flow Equations for Liveness

\[
\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

\[
\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]
\]

Improving Performance

Consider the (3 \rightarrow 4) edge in the graph:

\text{out}[4] is used to compute \text{in}[4];

\text{in}[4] is used to compute \text{out}[3] . . .

So we should compute the sets in the order: \text{out}[4], \text{in}[4], \text{out}[3], \text{in}[3], . . .

The order of computation should follow the direction of flow
Iterating Through the Flow Graph Backwards

<table>
<thead>
<tr>
<th>node</th>
<th>use</th>
<th>def</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>be</td>
<td>be</td>
<td>be</td>
<td>be</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>bc</td>
<td>bc</td>
<td>bc</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>e</td>
<td>ac</td>
<td>c</td>
<td>ac</td>
</tr>
</tbody>
</table>

Converges much faster!

Solving the Data-flow Equations (reprise)

**Algorithm**

```
for each node n in CFG
  in[n] = \emptyset; out[n] = \emptyset

repeat
  for each node n in CFG in reverse toposort order
    in'[n] = in[n]
    out'[n] = out[n]
    out[n] = \bigcup_{s \text{ a use(n)}} in[s]
    in[n] = use[n] \cup (out[n] - def[n])

until in'[n] = in[n] and out'[n] = out[n] for all n
```

- **initialize solutions**
- **save current results**
- **solve data-flow equations**
- **test for convergence**
Time Complexity

Consider a program of size $N$
- Has $N$ nodes in the flow graph and at most $N$ variables
- Each live-in or live-out set has at most $N$ elements
- Each set-union operation takes $O(N)$ time
- The for loop body
  - constant # of set operations per node
  - $O(N)$ nodes $\Rightarrow O(N^2)$ time for the loop
- Each iteration of the repeat loop can only make the set larger
- Each set can contain at most $N$ variables $\Rightarrow 2N^2$ iterations

Worst case: $O(N^3)$
Typical case: 2 to 3 iterations with good ordering & sparse sets $\Rightarrow O(N)$ to $O(N^2)$

Concepts

Liveness
- Use in register allocation
- Generating liveness
- Flow and direction
- Data-flow equations and analysis
- Complexity
- Improving performance (basic blocks, single variable, bit sets)

Control flow graphs
- Predecessors and successors

Defs and uses
Next Time

**Lecture**
- Generalizing data-flow analysis

**Assignment 2**
- Now available
- Due February 13
- Please start early