Introduction to Data-flow Analysis

Last Time
- Control flow analysis
- BT discussion

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 registers do we need for the program on the right?
- Easy bound: the number of variables used (3)
- 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
```
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).
∴ To compute liveness at a given point, we need to look into the future

Motivation: 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).
∴ Register allocation uses liveness information

Control Flow Graphs (CFGs)

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

Example
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
Liveness by Example

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

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

\( b \)'s live range is (2 → 3 → 4)

Liveness by Example (cont)

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)

Variables \( a \) and \( b \) are never simultaneously live, so they can share a register
**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 all predecessors of node \( n \)
- \( \text{succ}[n] \) is the set of all successors of node \( n \)

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

---

**Uses and Defs**

**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 **backwards** through the CFG, because the behavior at future nodes determines liveness at a given node

– Consider a
– Consider b
– Later, we’ll see other properties that flow **forward**

Liveness at Nodes

We have liveness on **edges**
– How do we talk about liveness at nodes?

Two More Definitions

– A variable is **live-out** at a node if it is live on **any** out-edges

– A variable is **live-in** at a node if it is live on **any** in-edges
Computing Liveness

Rules for computing liveness

1. Generate liveness:
   If a variable is in use[n], 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 pred[n].

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

Data-flow equations

\[
\begin{align*}
\text{(1)} & \quad \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
\text{(2)} & \quad \text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s] \\
\text{(3)} & \quad \text{in}[n] = \text{use}[n]
\end{align*}
\]

Solving the Data-flow Equations

Algorithm

\[
\begin{align*}
\text{for each node } n \text{ in CFG} & \quad \text{initialize solutions} \\
\text{repeat} & \\
\text{for each node } n \text{ in CFG} & \quad \text{save current results} \\
\text{in}'[n] = \text{in}[n] & \quad \text{solve data-flow equations} \\
\text{out}'[n] = \text{out}[n] & \\
\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) & \\
\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s] & \quad \text{test for convergence} \\
\text{until } \text{in}'[n] = \text{in}[n] \text{ and out}'[n] = \text{out}[n] \text{ for all } n &
\end{align*}
\]

This is iterative data-flow analysis (for liveness analysis)
**Example**

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

**Data-flow Equations for Liveness**

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

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

**Example (cont)**

**Data-flow Equations for Liveness**

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

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

**Improving Performance**

Consider the (3→4) edge in the graph:

- out[4] is used to compute in[4]
- in[4] is used to compute out[3]...

So we should compute the sets in the order: out[4], in[4], out[3], 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>a</td>
<td>ac</td>
<td>bc</td>
<td>ac</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>c</td>
<td>bc</td>
<td>bc</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td>ac</td>
<td>bc</td>
<td>ac</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>ac</td>
<td>c</td>
<td>ac</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] = ∅; out[n] = ∅ \{ Initialize solutions \}
repeat
  for each node n in CFG in reverse toposort order
    in'[n] = in[n] \{ Save current results \}
    out'[n] = out[n]
    out[n] = \bigcup_{s \in suc[n]} \{ in[s] \}
    in[n] = use[n] \cup (out[n] - def[n]) \{ Solve data-flow equations \}
  until in'[n] = in[n] and out'[n] = out[n] for all n "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: $O(N^4)$
Typical case: 2 to 3 iterations with good ordering & sparse sets
$\Rightarrow O(N)$ to $O(N^2)$

More Performance Considerations

Basic blocks
- Decrease the size of the CFG by merging nodes that have a single predecessor and a single successor into basic blocks
  (requires local analysis before and after global analysis)

One variable at a time
- Instead of computing data-flow information for all variables at once using sets, compute a (simplified) analysis for each variable separately

Representation of sets
- For dense sets, use a bit vector representation
- For sparse sets, use a sorted list (e.g., linked list)
### Conservative Approximation

<table>
<thead>
<tr>
<th>node #</th>
<th>use def</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>in</td>
<td>out</td>
<td>in</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>c</td>
<td>ac</td>
<td>ed</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>ac</td>
<td>bc</td>
<td>aed</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>c</td>
<td>bc</td>
<td>bdc</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td>ac</td>
<td>bdc</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>ac</td>
<td>ac</td>
<td>aed</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

**Solution X**
- Our solution as computed on previous slides

```plaintext
1. a := 0
2. b := a + 1
3. c := c + b
4. a := b * 2
5. a < 9?
6. return c
```

### Conservative Approximation (cont)

**Solution Y**
- Carries variable d uselessly around the loop
- Does Y solve the equations?
- Is d live?
- Does Y lead to a correct program?

**Imprecise conservative solutions** ⇒ **sub-optimal but correct programs**
Conservative Approximation (cont)

<table>
<thead>
<tr>
<th>node</th>
<th>use def</th>
<th>X in</th>
<th>Y in</th>
<th>Z out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>c ac</td>
<td>ed ac</td>
<td>c ac</td>
</tr>
<tr>
<td>2</td>
<td>a b</td>
<td>ac bc</td>
<td>ac</td>
<td>ac b</td>
</tr>
<tr>
<td>3</td>
<td>bc c</td>
<td>bc</td>
<td>bc</td>
<td>b c</td>
</tr>
<tr>
<td>4</td>
<td>b a</td>
<td>bc ac</td>
<td>bc ac</td>
<td>bc ac</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>ac ac</td>
<td>ac</td>
<td>ac ac</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>c</td>
<td>ac</td>
<td>c</td>
</tr>
</tbody>
</table>

Solution Z

- Does not identify c as live in all cases
- Does Z solve the equations?
- Does Z lead to a correct program?

Non-conservative solutions \(\Rightarrow\) incorrect programs

The Need for Approximations

Static vs. Dynamic Liveness

- In the following graph, \(b \ast b\) is always non-negative, so \(c \geq b\) is always true and \(a\)'s value will never be used after node 2

Rule (2) for computing liveness

- Since \(a\) is live-in at node 4, it is live-out at nodes 3 and 2
- This rule ignores actual control flow

No compiler can statically know all a program’s dynamic properties!
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

Conservative approximation
- Static versus dynamic liveness

Next Time

Lecture
- Generalizing data-flow analysis