Introduction to Data-flow analysis

Last Time
- LULESH intro
- Typed, 3-address code
- Basic blocks and control flow graphs
- LLVM Pass architecture
- Data dependencies, DU chains, and SSA

Today
- CFG and SSA example
- Liveness analysis
- Register allocation
int main() {
    int x, y, z;
    x = 0; y = 1, z = 2;
    while (x<42) {
        if (x>0) {
            y++; x++; z++; continue;
        }
        y = z*y;
    }
    return 0;
}
Control Flow Graph Review

while kond:
    %0 = load i32* %x, align 4
    %cmp = icmp slt i32 %0, 42
    br i1 %cmp,
    label %while.body,
    label %while.end

while.body:
    %1 = load i32* %x, align 4
    %cmp1 = icmp sgt i32 %1, 0
    br i1 %cmp1,
    label %if.then,
    label %if.end

if.then:
    ...
    br label %while.cond

if.end:
    ...
    store i32 %mul, i32* %y, align 4
    br label %while.cond

while.end:
    ret i32 0
while.cond:
  %y.0 = phi i32 [1,%entry], [%inc,%if.then], [%mul, %if.end]
  %x.0 = phi i32 [0,%entry ], [%inc2,%if.then ], [%x.0, %if.end]
  %z.0 = phi i32 [2,%entry ], [%inc3,%if.then ], [%z.0, %if.end]
  %cmp = icmp slt i32 %x.0, 42  %cmp = icmp slt i32 %0, 42
  br i1 %cmp,
    label %while.body,
    label %while.end

while.body:
  %cmp1 = icmp sgt i32 %x.0, 0
    ; previously
    ; %1 = load i32* %x, align 4
    ; %cmp1 = icmp sgt i32 %1, 0
  br i1 %cmp1,
    label %if.then,
    label %if.end
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

```plaintext
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).
  \[ \therefore \text{To compute liveness at a given point, we need to look into the future} \]

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).
  \[ \therefore \text{Register allocation uses liveness information} \]
Control Flow Graphs (CFGs)

Definition

– A **CFG** is a graph whose nodes represent program statements and whose directed edges represent control flow

Example

1. \( a := 0 \)
2. \( L1: b := a + 1 \)
3. \( c := c + b \)
4. \( a := b \times 2 \)
5. \( \text{if } a < 9 \text{ goto } L1 \)
6. \( \text{return } c \)
**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 all predecessors of node \( n \)
- \( \text{succ}[n] \) is the set of all successors of node \( n \)

**Examples**
- Out-edges of node 5: \((5 \rightarrow 6)\) and \((5 \rightarrow 2)\)
- \( \text{succ}[5] = \{2, 6\} \)
- \( \text{pred}[5] = \{4\} \)
- \( \text{pred}[2] = \{1, 5\} \)
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 \((2 \rightarrow 3 \rightarrow 4)\)
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
Uses and Defs

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

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

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**

```
1  a := 0
2  b := a + 1
3  c := c + b
4  a := b * 2
5  a < 9?
6  return c
```
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** of that node’s out-edges
  – A variable is **live-in** at a node if it is live on **any** of that node’s 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

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

Algorithm

\[
\text{for each node } n \text{ in CFG} \\
\quad \text{in}[n] = \emptyset; \quad \text{out}[n] = \emptyset \\
\text{repeat} \\
\quad \text{for each node } n \text{ in CFG} \\
\quad \quad \text{in}'[n] = \text{in}[n] \\
\quad \quad \text{out}'[n] = \text{out}[n] \\
\quad \quad \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
\quad \quad \text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s] \\
\text{until } \text{in}'[n] = \text{in}[n] \text{ and out}'[n] = \text{out}[n] \text{ for all } n
\]

\{ initialize solutions \}

\{ save current results \}

\{ solve data-flow equations \}

\{ test for convergence \}

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

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

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

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

1. \(a := 0\)
2. \(b := a + 1\)
3. \(c := c + b\)
4. \(a := b \times 2\)
5. \(a < 9?\)
6. \text{return } c
Do Liveness Example from Old Midterm
Register Allocation in LLVM

Have a reading assignment for Monday about this.

What register allocation does
  – Takes code out of SSA.
  – Replaces virtual registers with physical registers.

As of LLVM 3.0, Basic and Greedy
  – Used to be linear scan but there were efficiency and maintenance issues
  – Basic algorithm orders live ranges by size
  – Allocates large live ranges first
  – Let’s talk about live ranges and the general register allocation problem
Register Allocation

Problem

– Assign an unbounded number of symbolic registers to a fixed number of architectural registers
– Simultaneously live data must be assigned to different architectural registers

Goal

– Minimize overhead of accessing data
  – Memory operations (loads & stores)
  – Register moves
Scope of Register Allocation

Expression
Local
Loop
Global (over the control-flow graph for a function)
Interprocedural
Granularity of Allocation

**What is allocated to registers?**

- Variables
- Webs (i.e., du-chains with common uses)
- Values (i.e., definitions; same as variables with SSA)

```
\begin{align*}
\text{Variables:} & \quad 2 \ (x \ & y) \\
\text{Webs:} & \quad 3 \ (s_1 \rightarrow s_2, s_4; \\
& \quad s_2 \rightarrow s_3; \\
& \quad s_3, s_5 \rightarrow s_6) \\
\text{Values:} & \quad 4 \ (s_1, s_2, s_3, s_5, \phi (s_3, s_5))
\end{align*}
```

**What are the tradeoffs?**

*Each allocation unit is given a symbolic register name (e.g., t1, t2, etc.)*
Global Register Allocation by Graph Coloring

Idea [Cocke 71], First allocator [Chaitin 81]

1. Construct **interference graph** $G=(N,E)$
   - Represents notion of “simultaneously live”
   - Nodes are units of allocation (e.g., variables, live ranges, values)
   - $\exists$ edge $(n_1,n_2) \in E$ if $n_1$ and $n_2$ are simultaneously live
   - Symmetric (not reflexive nor transitive)

2. Find **k-coloring** of $G$ (for $k$ registers)
   - Adjacent nodes can’t have same color

3. **Allocate** the same register to all allocation units of the same color
   - Adjacent nodes must be allocated to distinct registers

\[ \begin{align*}
    t_2 \\
    t_1 & \quad t_3 
\end{align*} \]
Granularity of Allocation (Draw 3 Interference Graphs)

Variables: \(3 (x & y)\)

Webs: \(3 (s_1 \rightarrow s_2, s_4; s_2 \rightarrow s_3; s_3, s_5 \rightarrow s_6)\)

Values: \(4 (s_1, s_2, s_3, s_5, \phi (s_3, s_5))\)
Next Time

Reading
- Slide set and article about register allocation in LLVM, see progress page

Lecture
- Improvements to graph coloring register allocators
- Register allocation across procedure calls

Assignments
- PA1 is due Monday!!