Introduction to Data-flow analysis

Last Time
– Register allocation for expression trees

Today
– Control flow graphs
– 3-address code
– Register allocation using liveness analysis

Reading for this week
– Ch. 6.2-6.2.2, 3-address code
– 8-8.1.2, 8.1.4, intro to program analysis
– 8.4, 8.8, register allocation via data-flow analysis

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 + expr temp (4)
– 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)

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 * 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           a<9
6          return c

No
Yes
**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 $(1\rightarrow 2)$ and again from $(4\rightarrow 5\rightarrow 2)$
- $a$ is dead from $(2\rightarrow 3\rightarrow 4)$

Live range of $b$
- $b$ is live from $(2\rightarrow 3\rightarrow 4)$

Live range of $c$
- $c$ is live from $(\text{entry}\rightarrow 1\rightarrow 2\rightarrow 3\rightarrow 4\rightarrow 5\rightarrow 2, 5\rightarrow 6)$

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
- $\text{def}_\text{node}[v] = \text{set of CFG nodes that define variable } v$
- $\text{def}[n] = \text{set of variables that are defined at node } n$

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

**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 $\text{use}_\text{node}[v]$), and
  2. that path does not go through any def of $v$ (no nodes in $\text{def}_\text{node}[v]$)
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 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

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

Solving the Data-flow Equations

Algorithm

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

This is iteractive data-flow analysis (for liveness analysis)
Example

<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>c</td>
<td>ac</td>
<td>c</td>
<td>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>ac</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>c</td>
<td>bc</td>
<td>b</td>
<td>bc</td>
<td>ac</td>
<td>bc</td>
<td>ac</td>
<td>bc</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>ac</td>
<td>ac</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>ac</td>
<td>a</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

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

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

Liveness Analysis in the MeggyJava compiler

Currently ...
- Parse into AST
- Allocate space on stack for locals and parameters and space in heap for member variables
- Use stack for expression evaluation
- Generate AVR code from AST

To perform data-flow analysis ...
- Need intermediate representation like 3-address code
- Use temporaries/symbolic registers for expression results
- Indicate uses and defs of temporaries and locals and parameters in each 3-address code instruction
- Create a control-flow graph with each 3-address code instruction as a node
A Low-Level IR: 3-address code

3-address code
- Linear representation
- Typically language-independent and nearly corresponds to machine instructions
- Each var is assumed to have a base + offset
- Assumes infinite temps (t#), or symbolic registers, are available

Example operations
- Copy \( x = z, t1 = t2 \)
- Indexed copy \( x = y[i], y[i] = x, t1 = y[i] \)
- Unary op \( x = \text{op} z \)
- Binary op \( x = v \text{ op } z, t1 = t2 \text{ op } t3 \)
- Address of \( p = \& v \)
- Load \( x = *p \)
- Store \( *p = x, \)
- Pass param \( \text{param } t0 \)
- Call \( t1 = \text{call } f, 1 \)
- Branch \( \text{goto } L1 \)
- Cbranch \( \text{if } (x==y) \text{ goto } L1 \)

Register Allocation

Problem
- Assign an unbounded number of symbolic registers, or temporaries, 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
Interprocedural

Granularity of Allocation

What is allocated to registers?
– Variables
– Live ranges/Webs (i.e., du-chains with common uses)
– Values (i.e., definitions; same as variables with SSA)

Variables: 2 (x & y)
Live Ranges/Web: 3 (s_1 \rightarrow s_3, 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_2))
**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

---

**Interference Graph Example (Variables)**

```
a := ...
b := ...
c := ...
... a ...
d := ...
... d ...
a := ...
... e ...
... a ...
... e ...
... b ...
```

```c
... c ...
a := ...
... d ...
... d ...
e := ...
... a ...
e := ...
... a ...
... e ...
... b ...
```
Computing the Interference Graph

Use results of live variable analysis

\[
\text{for each symbolic-register/temporary/var } t_i \text{ do } \\
\quad \text{for each symbolic-register/temporary/var } t_j \text{ (} j < i \text{) do } \\
\quad \quad \text{for each } \text{def } \in \{\text{definitions of } t_i\} \text{ do } \\
\quad \quad \quad \text{if } (t_j \text{ is live out at def}) \text{ then } \\
\quad \quad \quad \quad E \leftarrow E \cup (t_i, t_j)
\]

Options
– treat all instructions the same
– treat MOVE instructions special
– which is better?

Allocating Registers Using the Interference Graph

\textit{K-coloring}
– Color graph nodes using up to } k \text{ colors
– Adjacent nodes must have different colors

Allocating to } k \text{ registers \textit{\textbf{finding a }k\text{-coloring of the interference graph}}
– Adjacent nodes must be allocated to distinct registers

But. . .
– Optimal graph coloring is NP-complete
  – Optimal register allocation is NP-complete, too (must approximate)
– What if we can’t } k\text{-color a graph? (must spill)}
Register Allocation: Spilling

If we can’t find a k-coloring of the interference graph
– Spill variables (nodes) until the graph is colorable

Choosing variables to spill
– Choose arbitrarily or
– Choose least frequently accessed variables
– Break ties by choosing nodes with the most conflicts in the interference graph
– Yes, these are heuristics!

Spilling (Original CFG and Interference Graph)
Spilling (After spilling b)

\[
\begin{align*}
a & := \ldots \\
b & := \ldots \\
\ast(Y+4) & := b \\
c & := \ldots \\
\ldots a & \ldots \\
d & := \ldots \\
\end{align*}
\]

Simple Greedy Algorithm for Register Allocation

\[
\text{for each } n \in N \text{ do} \\
\quad \{ \text{select } n \text{ in decreasing order of weight} \} \\
\quad \text{if } n \text{ can be colored then} \\
\quad \quad \{ \text{reserve a register for } n \} \\
\quad \quad \text{do it} \\
\quad \quad \text{else} \\
\quad \quad \text{Remove } n \text{ (and its edges) from graph} \\
\quad \quad \{ \text{allocate } n \text{ to stack (spill)} \}
\]

\[\text{(After spilling b)}\]

\[
\begin{align*}
a & := \ldots \\
r24 & := \ldots \\
\ast(Y+4) & := r24 \\
c & := \ldots \\
\ldots a & \ldots \\
d & := \ldots \\
\end{align*}
\]
Example

Attempt to 3-color this graph ( , , )

What if you use a different order?

Example

Attempt to 2-color this graph ( , )
Concepts

Liveness
- Used in register allocation
- Generating liveness
- Flow and direction
- Data-flow equations and analysis

3-address code and Control flow graphs

Register allocation
- scope of allocation
- granularity: what is being allocated to a register
- order that allocation units are visited in matters in all heuristic algorithms

Global approach: greedy coloring

Liveness in the MiniJava compiler