Interference graphs

Liveness in the MiniJava compiler
Interference

- The set of temporaries a, b, c, ... must be allocated to registers r1, r2, ..., rk.

- *Interference* prevents a and b from being allocated to the same register.

- Overlapping live ranges is the most common type of interference.

- If a and b are both live at the same program point, they cannot be allocated to the same register.
# Interference Representations

**Matrix:** x's indicate interference

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**Graph:** edges connect variables that interfere

\[ \begin{align*}
    a & \quad b \\
    \quad c
\end{align*} \]
Artificial Interference

\[
t \leftarrow s \\
x \leftarrow s + 1 \\
y \leftarrow t - 2
\]

- Given our usual liveness analysis, what are the live ranges of \( s \) and \( t \)?

- Do we need separate registers for \( s \) and \( t \)?

- To avoid creating an artificial interference between \( s \) and \( t \), we should not add an edge \((t, s)\) to the graph.
Adding Interference Edges

1. At any non-move instruction that defines variable a 
   (where live-out variables are \( b_1, ..., b_j \)) add interference 
   edges \((a, b_1), ..., (a, b_j)\).

2. At a move instruction \( a \leftarrow c \) (where variables \( b_1, ..., b_j \) 
   are live-out) add interference edges \((a, b_1), ..., (a, b_j)\) for 
   any \( b_i \) that is not the same as \( c \).
Example

\[ \text{in}[1] = \{c\}, \text{out}[1] = \{a, c\} \]

\[ \text{in}[2] = \{a, c\}, \text{out}[2] = \{b, c\} \]

\[ \text{in}[3] = \{b, c\}, \text{out}[3] = \{b, c\} \]

\[ \text{in}[4] = \{b, c\}, \text{out}[4] = \{a, c\} \]

\[ \text{in}[5] = \{a, c\}, \text{out}[5] = \{a, c\} \]

\[ \text{in}[6] = \{c\}, \text{out}[6] = \{} \]

a := c

b := a + 1

c := c + b

a := b * 2

a < N

return c
Example

Interference Graph:

```
a := c
b := a + 1
c := c + b
a := b * 2
a < N
return c
```
Liveness Analysis for MiniJava

- In the MiniJava compiler, liveness analysis is done in two stages.
  1. Analyze the control flow of the `Assem` program, producing a control flow graph.
  2. Analyze the liveness of variables in the flow graph, producing an interference graph.

- The text provides a `Graph` abstract data type to represent both types of graphs.
Graph

- `g.newNode()` creates new node in graph `g`.
- `g.addEdge(n, m)` adds directed edge `n` to `m` to `g`.
- `m` will be in `n.succ()` and `n` will be in `m.pred()`.
- List `m.adj() = m.succ() U m.pred()` is useful for undirected graphs.
- Each graph node represents something (an instruction or a temporary). Use a hashtable to associate each node with the appropriate information.
public abstract class FlowGraph extends Graph {
    public abstract TempList def(Node node);
    public abstract TempList use(Node node);
    public abstract boolean isMove(Node node);
    public void show(PrintStream out);
}

- Each node represents an instruction.
- def/use(n) indicates temporaries defined/used at n.
- isMove(n) is later used to determine if n may be omitted (possible if def and use are same).
public class AssemFlowGraph extends FlowGraph {
    public Assem.Instr instr(Node node);
    public AssemFlowGraph(Assem.InstrList il);
}

- Constructor takes instruction list, returns flow graph.
- Recall the jump field of the Assem.Instr class, used to create control-flow edges.
- use/def information obtained from src/dst fields of Assem.Instr.
abstract class InterferenceGraph extends Graph{
  public abstract Node tnode(Temp temp);
  public abstract Temp gtemp(Node node);
  public abstract MoveList moves();
  public int spillCost(Node node);
}

• tnode() and gtemp() relate Node's and Temp's.

• moves() gives the MOVEs associated with this graph.

• spillcost(n) gives estimate of how many extra
  instructions needed if n kept in memory instead of register.
Liveness

public class Liveness extends InterferenceGraph {
    public Liveness(FlowGraph flow);
}

• The constructor produces an interference graph from the
  given control flow graph.

• It is useful to maintain a hashtable liveMap that maps
  each Node n to the TempList of temporaries live at the
  exit of n.
Constructing Interference Graph

At each node $n$ of the control flow graph such that $d \in \text{def}(n)$ and $\text{liveMap} = \{t_1, t_2, \ldots\}$, add edges $(d, t_1), (d, t_2), \ldots$ to the interference graph.

```java
NodeList nl = flow.nodes();
while (nl != null) {
    TempList liveouts = liveMap.get(nl.head);
    while (liveouts != null) {
        TempList defs = flow.def(nl.head);
        while (defs != null) {
            //add edge (defs.head, liveouts.head)
            defs = defs.tail;
        }
        liveouts = liveouts.tail;
    }
    nl = nl.tail;
}
```