Analysis and Transformation

• Analysis:
  – Control Flow Analysis
  – Dataflow Analysis

• Transformation:
  – Register Allocation
  – Optimization
    * Machine dependent/independent
    * Local/Global/Interprocedural
    * Acyclic/Cyclic
  – Scheduling
Dataflow Analysis Motivation

Constant Propagation and Dead Code Elimination:

\[ r_2 = r_1 + 5 \]
\[ r_1 = 4 \]
\[ r_2 = r_1 + 5 \]
\[ r_2 = 9 \]

Needs dominator, liveness, and reaching definition information.
Dataflow Analysis Motivation

Register Allocation:

- Infinite number of registers (virtual registers) must be mapped to a limited number of real registers.

- Pseudo-assembly must be examined by *live variable analysis* to determine which virtual registers contain values which may be used later.

- Virtual registers which are not simultaneously *live* may be mapped onto the same real register.

```
1  r2 = r1 + 1

2  r3 = M[r2]

3  r4 = r3 + 4

4  LOAD r5 = M[r2 + r4]
```
Dataflow Analysis

Three types we will cover:

- Live Variable
  - Live range for register allocation
  - Scheduling
  - Dead code elimination

- Reaching Definitions
  - Constant propagation
  - Constant folding
  - Copy propagation

- Available expressions
  - Common subexpression elimination
Iterative Dataflow Analysis Framework

• These dataflow analyses are all very similar → define a framework.

• Specify:
  – Two set definitions - $A[n]$ and $B[n]$
  – A transfer function - $f (A, B, IN/OUT)$
  – A confluence operator - $\lor$.
  – A direction - FORWARD or REVERSE.

• For forward analyses:
  
  $IN[n] = \lor_{p \in PRED[n]} OUT[p]$
  
  $OUT[n] = f (A, B, IN)$

• For reverse analyses:
  
  $OUT[n] = \lor_{s \in SUCC[n]} IN[s]$
  
  $IN[n] = f (A, B, OUT)$
Definitions

Control Flow Definitions:

- CFG node has *out-edges* leading to *successor nodes*.
- CFG node has *in-edges* coming from *predecessor nodes*.
- For each CFG node $n$, $PRED[n] = \text{set of all predecessors of } n$.
- For each CFG node $n$, $SUCCE[n] = \text{set of all successors of } n$. 
Iterative Dataflow Analysis Framework

- Iterative dataflow analysis equations are applied in an iterative fashion until $IN$ and $OUT$ sets do not change.

- Typically done in (FORWARD or REVERSE) topological sort order of CFG for efficiency.

- $IN$ and $OUT$ sets initialized to $\emptyset$.

For each node $n$ {
    $IN[n] = OUT[n] = \{}$;
}

Repeat {
    For each node $n$ in forward/reverse topological order {
        $IN'[n] = IN[n]$;
        $OUT'[n] = OUT[n]$;
        $IN[n], OUT[n] = (Equations)$;
    }
} until $IN'[n] = IN[n]$ and $OUT'[n] = OUT[n]$ for all $n$. 
Definitions

Liveness Definitions:

• A source (RHS) register $t$ is a use of $t$.

• A destination (LHS) register $t$ is a definition of $t$.

• A register $t$ is live on edge $e$ if there exists a path from $e$ to a use of $t$ that does not go through a definition of $t$.

• Register $t$ is live-in at CFG node $n$ if $t$ is live on any in-edge of $n$.

• Register $t$ is live-out at CFG node $n$ if $t$ is live on any out-edge of $n$. 
Live Variable Analysis

Live Variable Analysis Equation:

- Set definition \((A[n])\): \(USE[n]\) - the set of registers that \(n\) uses.
- Set definition \((B[n])\): \(DEF[n]\) - the set of registers that \(n\) defines.
- Transfer function \((f(A, B, OUT))\): \(USE[n] \cup (OUT[n] - DEF[n])\)
- Confluence operator \((\lor)\): \(\lor\)
- Direction: REVERSE

\[
OUT[n] = \bigcup_{s \in SUCC[n]} IN[s] \\
IN[n] = USE[n] \cup (OUT[n] - DEF[n])
\]
Live Variable Analysis Example

1: \( r_1 = 0 \)
2: \( r_2 = r_1 + 1 \)
3: \( r_3 = r_3 + r_2 \)
4: \( r_1 = r_2 * 2 \)
5: branch \( r_1 < 10, \text{L1} \)

6: return \( r_3 \)

<table>
<thead>
<tr>
<th>Node</th>
<th>USE</th>
<th>DEF</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Live Variable Application 1: Register Allocation

Register Allocation:

1. Perform live variable analysis.

2. Build *interference graph*.

3. Color interference graph with real registers.
Interference Graph

- Node $t$ corresponds to virtual register $t$.
- Edge $\langle t_i, t_j \rangle$ exists if registers $t_i, t_j$ have overlapping live ranges.
- For some node $n$, if $DEF[n] = \{a\}$ and $OUT[n] = \{b_1, b_2, ... b_k\}$, then add interference edges: $\langle a, b_1 \rangle, \langle a, b_2 \rangle, \langle a, b_k \rangle$

Interference Graph For Example:

<table>
<thead>
<tr>
<th>Node</th>
<th>$DEF$</th>
<th>$OUT$</th>
<th>$IN$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1</td>
<td>r1,r3</td>
<td>r3</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2,r3</td>
<td>r1,r3</td>
</tr>
<tr>
<td>3</td>
<td>r3</td>
<td>r2,r3</td>
<td>r2,r3</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
<td>r1,r3</td>
<td>r2,r3</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>r1, r3</td>
<td>r1,r3</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td></td>
<td>r3</td>
</tr>
</tbody>
</table>

Virtual registers r1 and r2 may be mapped to same real registers.
Live Variable Application 2: Dead Code Elimination

- Given statement $s$ with a definition and no side-effects:
  
  \[
  r_1 = r_2 + r_3, \quad r_1 = M[r_2], \quad \text{or} \quad r_1 = r_2
  \]

  If $r_1$ is *not* live at the end of $s$, then the $s$ is *dead*

- Dead statements can be deleted.

- Given statement $s$ without a definition or side-effects:
  
  \[
  r_1 = \text{call FUN\_NAME}, \quad M[r_1] = r_2
  \]

  Even if $r_1$ is not live at the end of $s$, it is not dead.

Example:

\[
\begin{align*}
  & r_1 = r_2 + 1 \\
  & r_2 = r_2 + 2 \\
  & r_1 = r_2 + 3 \\
  & M[r_1] = r_2
\end{align*}
\]
Reaching Definition Analysis

Determines whether definition of register $t$ directly affects use of $t$ at some point in program.

**Reaching Definition Definitions:**

- *unambiguous* - instruction explicitly defines register $t$.
- *ambiguous* - instruction may or may not define register $t$.
  - Global variables in a function call.
  - No ambiguous definitions in tiger since all globals are stored in memory.
- Definition of $d$ (of $t$) *reaches* statement $u$ if a path of CFG edges exists from $d$ to $u$ that does not pass through an unambiguous definition of $t$.
- One unambiguous and many ambiguous definitions of $t$ may reach $u$ on a single path.
Reaching Definition Analysis

Reaching Definition Analysis Equation:

- Set definition ($B[n]$): $KILL[n]$ - the set of definition id’s that $n$ kills.
  - $defs(t)$ - set of all definition id’s of register $t$.
- Transfer function ($f(A, B, IN)$): $GEN[n] \cup (IN[n] - KILL[n])$
- Confluence operator ($\lor$): $\cup$
- Direction: FORWARD

$$IN[n] = \bigcup_{p \in \text{PRE}[n]} OUT[p]$$

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$$
Reaching Definition Analysis Example

1: \( r_1 = 5 \)
2: \( r_3 = 1 \)
3: \( \text{branch } r_3 > r_1, 6: \)
4: \( r_3 = r_3 + 1 \)
5: \( \text{goto } 3: \)
6: \( r_4 = 10 \)
7: \( r_1 = r_1 + r_4 \)
8: \( M[r_3] = r_1 \)

<table>
<thead>
<tr>
<th>Node</th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definition Application 1: Constant Propagation

- Given Statement $d$: $a = c$ where $a$ is constant
- Given Statement $u$: $t = a \ op \ b$
- If statement $d$ reach $u$ and no other definition of $a$ reaches $u$, then replace $u$ by $t = c \ op \ b$.

Statements 1 and 6 are dead.
Constant Folding

- Given Statement \( d: t = a \ op \ b \)
- If \( a \) and \( b \) are constant, compute \( c \) as \( a \ op \ b \), replace \( d \) by \( t = c \)