Outline

1. Liveness Analysis
### Liveness Analysis

**IR after instruction selection**
- abstract assembly code
- operates on unbounded number of temporaries

**Next goal**
- register allocation
Register allocation

- instruction operands in registers
- bounded number of registers $\Rightarrow$ limited resource
- questions to be addressed
  - how many registers are needed at every program point?
  - what to do if fewer registers are available than needed?
- optimal allocation is NP-complete
How many registers are needed?

Concept: Live range
The live range of a temporary spans all instructions that may be executed between its definition and one of its uses.

Concept: Liveness
A temporary is live at some instruction if its value may be used in the future.

Answers
- At any given instruction, all live temporaries may be needed.
- Temporaries that are not needed at the same time may share a register.
What if fewer registers are available than needed?

**Concept: Spill**

Spilling a temporary means
- allocate it in a stack frame
- insert store instruction right after its definition
- insert load instruction before every use

**Consequences of spilling**
- shortens the live range of a temporary
- increases the size of a stack frame
- accessing the temporary becomes more expensive
Roadmap

1. control-flow graph
2. liveness analysis
3. interference graph
Graphical representation of control flow in a program

**CFG of a program**

- **Nodes**: entry, exit, and each occurrence of a statement in program
- **Edges**: an edge from $n$ to $n'$ represents a potential control transfer from (the end of) $n$ to (the beginning of) $n'$

**Terminology**

- **Out-edges** from $n$ lead to successor nodes, $\text{succ}[n]$
- **In-edges** to $n$ come from predecessor nodes, $\text{pred}[n]$
Example CFG

\[ a \leftarrow 0 \]

\[ L_1 : \]
\[ b \leftarrow a + 1 \]
\[ c \leftarrow c + b \]
\[ a \leftarrow b \times 2 \]
\[ \text{if } a < N \text{ goto } L_1 \]
\[ \text{return } c \]
Definitions and uses

Consider a CFG

- A variable $v$ gets **defined** by node $n$, if the statement at $n$ assigns to $v$.
- A variable $v$ gets **used** by node $n$, if $v$ occurs in an expression at $n$, i.e., it reads from $v$.

- **def**[$n$] set of variables defined by $n$
- **use**[$n$] set of variables used by $n$
- **def**[$n$] and **use**[$n$] are fixed by program/CFG
### Example def-use

<table>
<thead>
<tr>
<th></th>
<th>def[n]</th>
<th>use[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \leftarrow 0 )</td>
<td>{a}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( L_1: b \leftarrow a + 1 )</td>
<td>{b}</td>
<td>{a}</td>
</tr>
<tr>
<td></td>
<td>{c}</td>
<td>{c, b}</td>
</tr>
<tr>
<td></td>
<td>{a}</td>
<td>{b}</td>
</tr>
<tr>
<td>if ( a &lt; N ) goto ( L_1 )</td>
<td>\emptyset</td>
<td>{a}</td>
</tr>
<tr>
<td>return ( c )</td>
<td>\emptyset</td>
<td>{c}</td>
</tr>
</tbody>
</table>
Liveness

**Definition**
Variable $v$ is live on edge $e$ if there is an execution path from $e$ to a use of $v$ that does not pass through any definition of $v$.

**Liveness Analysis**
A data flow analysis that computes the variables that may be live at each edge of a control flow graph.

**Definition for analysis**
Variable $v$ is live on edge $e$ if there is a directed path from $e$ to a use of $v$ that does not pass through any definition of $v$. 
More on liveness

Liveness at node \( n \)

- \( v \) is **live-in** at \( n \) if \( v \) is live on any in-edge of \( n \)
  - \( \text{in}[n] \) variables live-in at \( n \)

- \( v \) is **live-out** at \( n \) if \( v \) is live on any out-edge of \( n \)
  - \( \text{out}[n] \) variables live-out at \( n \)
Liveness analysis

Computation rules for liveness

1. $v \in \text{use}[n]$ implies $v$ live-in at $n$
2. $v$ live-in at $n$ implies $v$ live-out at all $m \in \text{pred}[n]$
3. $v$ live-out at $n$ and $v \notin \text{def}[n]$ implies $v$ live-in at $n$

$\Rightarrow$ liveness information is propagated backwards
Liveness analysis

Computation rules for liveness

1. $v \in \text{use}[n]$ implies $v$ live-in at $n$
2. $v$ live-in at $n$ implies $v$ live-out at all $m \in \text{pred}[n]$
3. $v$ live-out at $n$ and $v \notin \text{def}[n]$ implies $v$ live-in at $n$

$\Rightarrow$ liveness information is propagated backwards

Inequations from computation rules

\[
\begin{align*}
\text{in}[n] & \supseteq \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\
\text{out}[n] & \supseteq \bigcup_{m \in \text{succ}[n]} \text{in}[m]
\end{align*}
\]

- rule 1
- rule 2
- rule 3
Liveness analysis

- Each solution of the inequations is valid liveness information
- Wanted: least solution that does not contain spurious information
- computed by fixpoint iteration
  - treat inequations (from right to left) as functions
  - update the left-hand in[n] and out[n] until no further change happens
- result is a fixpoint because afterwards

\[
\begin{align*}
in[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\
\text{out}[n] &= \bigcup_{m \in \text{succ}[n]} \text{in}[m]
\end{align*}
\]
for all node $n$ do
    $in^0[n] \leftarrow \emptyset$
    $out^0[n] \leftarrow \emptyset$
end for

$i = 0$

repeat
    $i \leftarrow i + 1$
    for all node $n$ do
        $in^i[n] \leftarrow use[n] \cup (out^{i-1}[n] \setminus def[n])$
        $out^i[n] \leftarrow \bigcup_{s \in succ[n]} in^{i-1}[s]$
    end for
until $\forall n, in^i[n] = in^{i-1}[n] \land out^i[n] = out^{i-1}[n]$
Each loop iteration increases \textbf{in}[n] and/or \textbf{out}[n]

Liveness flows backwards along control-flow arcs

The inner loop should visit nodes in reverse flow order as much as possible

Speedup: compress nodes to basic blocks
### Correctness

#### Monotone

\[ \text{in}^{i+1}[n] \supseteq \text{in}^i[n] \quad \text{out}^{i+1}[n] \supseteq \text{out}^i[n] \]

#### Bounded

\[ \text{in}^i[n] \subseteq \text{use}[n] \cup (\text{out}^i[n] \setminus \text{def}[n]) \]

\[ \text{out}^i[n] \subseteq \bigcup_{s \in \text{succ}[n]} \text{in}^i[s] \]
Example analysis, 1st iteration

<table>
<thead>
<tr>
<th></th>
<th>(\text{def}[n])</th>
<th>(\text{use}[n])</th>
<th>(\text{in}^1[n])</th>
<th>(\text{out}^1[n])</th>
<th>(\text{in}^2[n])</th>
<th>(\text{out}^2[n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \leftarrow 0)</td>
<td>{a}</td>
<td>\emptyset</td>
<td>{c}</td>
<td>{c, a}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>(L_1: b \leftarrow a + 1)</td>
<td>{b}</td>
<td>{a}</td>
<td>{c, a}</td>
<td>{c, b}</td>
<td>{c, a}</td>
<td>{c, b}</td>
</tr>
<tr>
<td>(c \leftarrow c + b)</td>
<td>{c}</td>
<td>{c, b}</td>
<td>{c, b}</td>
<td>{c, b}</td>
<td>{c, a}</td>
<td>{c, a}</td>
</tr>
<tr>
<td>(a \leftarrow b \times 2)</td>
<td>{a}</td>
<td>{b}</td>
<td>{c, b}</td>
<td>{c, a}</td>
<td>{c, a}</td>
<td>{c, a}</td>
</tr>
<tr>
<td>if (a &lt; N) goto (L_1)</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{c, a}</td>
<td>{c}</td>
<td>{c}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>return (c)</td>
<td>\emptyset</td>
<td>{c}</td>
<td>{c}</td>
<td>\emptyset</td>
<td>{c}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Example analysis, 2nd iteration

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a ← 0</td>
<td>{a}</td>
<td>∅</td>
<td>{c}</td>
<td>{c, a}</td>
<td>{c}</td>
<td>{c, a}</td>
</tr>
<tr>
<td>L_1 : b ← a + 1</td>
<td>{b}</td>
<td>{a}</td>
<td>{c, a}</td>
<td>{c, b}</td>
<td>{c, a}</td>
<td>{c, b}</td>
</tr>
<tr>
<td>c ← c + b</td>
<td>{c}</td>
<td>{c, b}</td>
<td>{c, b}</td>
<td>{c, b}</td>
<td>{c, b}</td>
<td>{c, b}</td>
</tr>
<tr>
<td>a ← b × 2</td>
<td>{a}</td>
<td>{b}</td>
<td>{c, b}</td>
<td>{c, a}</td>
<td>{c, b}</td>
<td>{c, a}</td>
</tr>
<tr>
<td>if a &lt; N goto L_1</td>
<td>∅</td>
<td>{a}</td>
<td>{c, a}</td>
<td>{c}</td>
<td>{c, a}</td>
<td>{c, a}</td>
</tr>
<tr>
<td>return c</td>
<td>∅</td>
<td>{c}</td>
<td>{c}</td>
<td>∅</td>
<td>{c}</td>
<td>∅</td>
</tr>
</tbody>
</table>
def \[n\] use \[n\] \hspace{1cm} \text{in}^1[n] \hspace{1cm} \text{out}^1[n] \hspace{1cm} \text{in}^2[n] \hspace{1cm} \text{out}^2[n] \\
\hline
a & \leftarrow 0 & \{a\} & \emptyset & \{c\} & \{c, a\} & \{c\} & \{c, a\} \\
L_1: & b & \leftarrow a + 1 & \{b\} & \{a\} & \{c, a\} & \{c, b\} & \{c, a\} & \{c, b\} \\
c & \leftarrow c + b & \{c\} & \{c, b\} & \{c, b\} & \{c, b\} & \{c, b\} & \{c, b\} \\
a & \leftarrow b \times 2 & \{a\} & \{b\} & \{c, b\} & \{c, a\} & \{c, b\} & \{c, a\} \\
\text{if } a < N \text{ goto } L_1 & \emptyset & \{a\} & \{c, a\} & \{c\} & \{c, a\} & \{c, a\} & \{c, a\} \\
\text{return } c & \emptyset & \{c\} & \{c\} & \emptyset & \{c\} & \emptyset
\hline

**Fixpoint reached**

- maximum number of live variables = 2
- 2 registers sufficient
For input program of size $N$

- $\leq N$ nodes in CFG
  - $\Rightarrow \leq N$ variables
  - $\Rightarrow \leq N$ elements per $\text{in}[n]$ and $\text{out}[n]$  
  - $\Rightarrow O(N)$ time per set operation

- for-loop performs constant number of set operations per node  
  - $\Rightarrow O(N^2)$ time for the loop

- the repeat loop cannot decrease any set sizes of all in and out sets $\leq 2N^2$  
  - $\Rightarrow$ repeat loop terminates after $\leq 2N^2$ iterations

- $\Rightarrow$ overall worst-case complexity $O(N^4)$

- in practice only few iterations when ordering is observed
Least fixpoints

- Technically, the algorithm computes the least fixpoint / least solution of the inequations.
- Any fixpoint/solution is a conservative approximation that tacitly assumes further uses of variables.
- The least fixpoint only considers manifest uses in the CFG.
- It is always safe to assume a variable is live.
- It is unsafe to assume a variable is dead.
Suppose that $\text{in}[n]$ and $\text{out}[n]$ solve the liveness inequations.

**Interference graph**

The interference graph is an undirected graph with

- nodes the variables of the CFG
- an edge $\{v, v'\}$ if exists node $n$ in the CFG such that $\{v, v'\} \subseteq \text{in}[n]$
Suppose that $\text{in}[n]$ and $\text{out}[n]$ solve the liveness inequations.

**Interference graph**

The interference graph is an undirected graph with
- nodes the variables of the CFG
- an edge $\{v, v'\}$ if exists node $n$ in the CFG such that $\{v, v'\} \subseteq \text{in}[n]$

**Interference graph for example**

```
  a
 / \
 b   c
```
Approach to register allocation

- Find a coloring of the interference graph with \( n \) colors where \( n \) is the number of available registers.
- Difficulties:
  - include spilling
  - efficiency

2-colored interference graph for example