CS577 Modern Language Processors
Spring 2018
Lecture Liveness via Dataflow Analysis
To assign registers effectively for a procedure, we need to look at the uses of each variable across expressions and statements.

To see how long to keep a given variable (or temporary) in a register, need to know the range of instructions for which the variable is live.

A variable or temporary is live immediately following an instruction ("live-out") if its current value will be needed in the future (i.e., it will be used again, and it won’t be changed before that use).

Example:

```plaintext
! temps live after instruction:

mov 3, t2     ! t2
mov t2, t3    ! t2 t3
add t3, 4, t4 ! t2 t4
add t2, t4, t4 ! t4
mov t4, r0    ! (nothing)
```

It’s easy to calculate liveness for a consecutive series of instructions without branches, just by working backwards.
But if a value can stay in a register over a jump, things get harder, e.g.:

```
1    mov 0, t1       ! t1   t3
2 L1: add t1, 1, t2   ! t2 t3
3    add t3, t2, t3   ! t2 t3
4    mul t2, 2, t1    ! t1 t3
5    cmp t1, 1000     ! t1 t3
6    jl L1            ! t1 t3
7    mov t3, r0       ! (nothing)
```

To calculate liveness in general requires **data flow analysis** and the result is only a **conservative approximation** to true liveness.

The **live range** of a variable is the set of instructions which leave it live (for which it is “live out”). E.g. here live range of `t1` is `{1, 4, 5, 6}`.

Basic idea: If two variables have overlapping live ranges, they cannot occupy the same physical register.

So in both examples, at least 2 physical registers are required to allocate all temporaries without temporarily **spilling** values to memory.
CONTROL-FLOW GRAPHS, REVISITED

To compute liveness and other properties of an entire procedure, we use a control-flow graph.

In simplest form, control flow graph has one node per statement, and an edge from \( n_1 \) to \( n_2 \) if control can ever flow directly from statement 1 to statement 2.

We write \( \text{pred}[n] \) for the set of predecessors of node \( n \), and \( \text{succ}[n] \) for the set of successors.

(In practice, usually build control-flow graphs where each node is a basic block, rather than a single statement.)

Example....
\[
a = 0 \\
L: \ b = a + 1 \\
\ c = c + b \\
\ a = b \times 2 \\
\text{if } a < N \text{ goto } L \\
\text{return } c
\]
**Liveness Analysis on the CFG**

Working from the future to the past, we can determine the edges over which each variable is live.

In the example:

- \( b \) is live on \( 2 \rightarrow 3 \) and on \( 3 \rightarrow 4 \).
- \( a \) is live from on \( 1 \rightarrow 2 \), on \( 4 \rightarrow 5 \), and on \( 5 \rightarrow 2 \) (but not on \( 2 \rightarrow 3 \rightarrow 4 \)).
- \( c \) is live throughout (including on entry \( \rightarrow 1 \)).

Can see that at least two registers are needed to hold \( a, b, c \).

In fact, two registers are also sufficient in this case, but that doesn’t always hold.
We can do liveness analysis (and many other analyses) via dataflow analysis over the control flow graph.

For simplicity, continue to assume that the CFG contains one node per statement.

A node **defines** a variable if its corresponding statement assigns to it.

A node **uses** a variable if its corresponding statement mentions that variable in an expression (e.g., on the rhs of assignment).

For any variable \(v\), define

- \(\text{def}[v]\) = set of graph nodes that define \(v\)
- \(\text{use}[v]\) = set of graph nodes that use \(v\)

Similarly, for any node \(n\), define

- \(\text{def}[n]\) = set of variables defined by node \(n\);
- \(\text{use}[v]\) = set of variables used by node \(n\).
A variable is **live** on an edge if there is a directed path from that edge to a **use** of the variable that does not go through any **def**.

A variable is **live-in** at a node if it is live on any in-edge of that node; it is **live-out** if it is live on any out-edge.

Then the following equations hold:

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

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

We want the **least fixed point** of these equations: the smallest \( in \) and \( out \) sets such that the equations hold.
SOLVING FIXED-POINT EQUATIONS

We can find this solution by iteration:

• Start with empty sets
• Use equations to add variables to sets, one node at a time.
• Repeat until sets don’t change any more.

Implementation issues:

• Algorithm always terminates, because each iteration must enlarge at least one set, but sets are limited in size (by total number of variables).
• Time complexity is $O(N^4)$ worst-case, but between $O(N)$ and $O(N^2)$ in practice.
• Sets can be represented as bit vectors or linked lists; best choice depends on set density.

Example revisited...
a = 0
L: b = a + 1
c = c + b
a = b * 2
if a < N goto L
return c
EXAMPLE REVISITED

For correctness, order in which we take nodes doesn’t matter, but it turns out to be fastest to take them in roughly reverse order:

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

Normally, entire basic blocks are used as the nodes of the CFG. Most dataflow algorithms reduce to very simple form within a block.

For example, for liveness, it is easy to compute the in and out sets of each block by a simple backward pass over its instructions.