CS302 Spr'99 Lecture Notes
Lecture 10a

Control-flow Graphs

To assign registers on a per-procedure vasis, need to perform liveness analysis on entire procedure, not just basic blocks.

To analyze the properties of entire procedures with multiple basic blocks, 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 *pred*$[n]$ for the set of predecessors of node n, and *succ*$[n]$ for the set of successors.

(Can also build control-flow graphs where each node is a **basic block**.)

Example routine:

```
    a = 0
 L: b = a + 1
    c = c + b
    a = b * 2
    if a < N goto L
    return c
```


Control-flow graph

```
           |
           |
        1  V
       |-------|
       | a = 0 |
       |-------|
           |
           |   |------|
           |   |      |
        2  V   V      |
       |-----------|  |
       | b = a + 1 |  |
       |-----------|  |
           |          |
           |          |
        3  V          |
       |-----------|  |
       | c = c + b |  |
       |-----------|  |
           |          |
           |          |
        4  V          |
       |-----------|  |
       | a = b * 2 |  |
       |-----------|  |
           |          |
           |          |
        5  V          |
       |-------|      |
       | a < N |      |
       |-------|      |
         |  T |       |
       F |    |-------|
         |
  6      V
 |----------|
 | return c |
 |----------|
```

Liveness Analysis using Dataflow Analysis

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 two registers suffice to hold a,b,c.

Dataflow Equations

We can do liveness analysis (and many other analyses) via **dataflow** analysis.

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

- $\mathit{def}[v]$ = set of graph nodes that define v
- $\mathit{use}[v]$ = set of graph nodes that use v

Similarly, for any node n, define

- $\mathit{def}[n]$ = set of variables defined by node n;
- $\mathit{use}[v]$ = set of variables used by node n.

Setting up Equations

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:

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

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

We want the **least fixed point** of these equations: the smallest *in* and *out* sets such that the equations hold.

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.

Adding additional variables to the sets is safe, as long as the sets still obey the equations, but inaccurately suggests that more live variables exist than actually do.

Solution

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:

			1st		2nd		3rd	
node	*use*	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Implementation issues:

- Algorithm always terminates, because each iteration must enlarge at least one set, but sets are limited in size (by total number o variables).
- Time complexity is $O(N^4)$ worst-case, but between $O(N)$ and $O(N^2)$ in practice.
- Typically do analysis using entire basic blocks as nodes.
- Can compute liveness for all variables in parallel (as here) or independently for each variable, on demand.
- Sets can be represented as bit vectors or linked lists; best choice depends on set density.

Static vs. Dynamic Liveness

Consider the following graph:

Is `a` live-out at node 2? It depends on whether control flow ever reaches node 4.

A smart compiler could answer no.

A smarter compiler could answer similar questions about more complicated programs.

But **no** compiler can ever **always** answer such questions correctly.

This is a consequence of the **uncomputability** of the **Halting Problem**.

So we must be content with **static** liveness, which talks about paths of control-flow edges, and is just a **conservative** approximation of **dynamic liveness**, which talks about actual execution paths.

Halting Problem

Theorem There is no program H that takes an input any program P and input X, and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite-loops.

Proof Suppose there were such an H. From it, construct the function

$F(Y) =$ `if` $H(Y,Y)$ `then (while true do ()) else 1`

Now consider $F(F)$.

• If $F(F)$ halts, then, by the definition of H, $H(F,F)$ is true, so the `then` clause executes, so $F(F)$ does not halt.

• But,if $F(F)$ loops forever, then $H(F,F)$ is false, so the `else` clause is taken, so $F(F)$ halts.

Hence $F(F)$ halts if any only if it doesn't halt.

Since we've reached a contradiction, the initial assumption is wrong: there can be no such H.

Corollary No program $H'(P,X,L)$ can tell, for any program P, input X, and label L within P, whether L is ever reached on an execution of P on X.

Proof If we had H', we could construct H. Consider a program transformation T that, from any program P constructs a new program by putting a label L at the end of the program, and changing every `halt` to `goto` L. Then $H(P,X) = H'(T(P),X,L)$.