Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

Lab 5: Liveness and Dataflow Analysis

(Adapted from Prof. Andrew Tolmach's earlier version)

In this lab we will introduce the concept of *variable liveness* and the framework of *dataflow analysis*. We'll show how to formulate and solve liveness analysis as a dataflow problem.

Start by downloading and unzipping `lab5.zip` from D2L in the usual way.

Liveness

Liveness tries to capture variables' dynamic usage behavior with compile-time approximation. Liveness information is essential for optimizing variables' register allocation.

First, the definition of liveness. A variable is *live* at a program execution point if its current value *might* be needed later in the execution: in other words, if the variable might be read again before it is updated.

One convenient way to record the live variables of a program is to list which variables are live immediately before and after each instruction. (We say these variables are "live in" and "live out" of the instruction.)

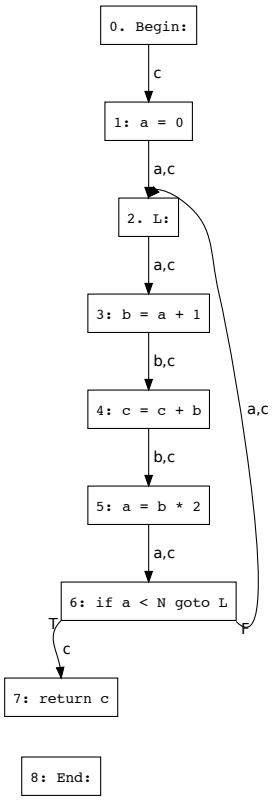
Example 0

Here is an example, using our familiar IR format. (This is also in file `example0.ir`.)

```
program:                      live-out:    live-in:
0. Begin:                         c            c
1.    a = 0                     a   c            c
2. L:                           a   c          a   c
3.    b = a + 1                   b c          a   c
4.    c = c + b                   b c            b c
5.    a = b * 2                 a   c            b c
6.    if a < 1000 goto L        a   c          a   c
7.    return c                  (none)           c
8. End:                         (none)       (none)
```

It is often easier to understand liveness if we use the *control flow graph (CFG)* of the program. Here we use the *fine-grained* CFG. Such CFG contains a node for each instruction (including labels) and a directed edge wherever one instruction can be executed immediately after another. (It is more conventional to define CFGs where the nodes are *basic blocks*, as shown in lecture. But using instructions is simpler, although it makes the CFG bigger.)

Here is the CFG for this program, where we have labeled each edge with the set of variables live along it.



Note that if a node has multiple in-edges, they are all labeled with the same live set (which is the node's live-in set). If a node has multiple out-edges, its live-out set is the *union* of the labels on those edges.

Using the liveness labels on edges, we can trace the lifetime of each variable. For example, variable `b` is live from its point of definition in instruction 3 until its last point of use in instruction 5. Variable `a` is live from its initial definition in instruction 1 to its use in instruction 3, and then again from its redefinition in instruction 5 until its use in instruction 3 (going around the loop); however, it is *not* live after instruction 3, because its value there will not be used again before it is redefined in instruction 5. Variable `c` is live throughout, even before instruction 1; this is an indication that is used before being intitialized (perhaps because of a program error).

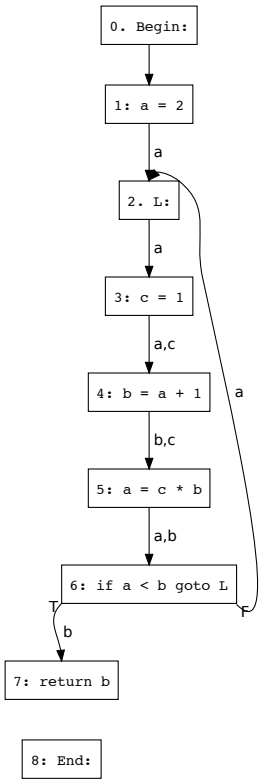
For programs this small, it is not hard to compute liveness informally. At worst, we can ask ourselves the question: for each variable mentioned in the program and each CFG edge, will the variable's current value be used again on some path starting with that edge?

Exercise: Example 1

Examine the program in file `example1.ir`. Draw a CFG for this program and annotate each edge with the live variables, using informal reasoning.

Here's a solution:

```
program:                        live-out:    live-in:
 0. Begin:                       (none)       (none)
 1.     a = 2                    a            (none)
 2. L:                           a            a
 3.     c = 1                    a,c          a
 4.     b = a + 1                b,c          a,c
 5.     a = c * b                a,b          b,c
 6.     if a < b goto L          a,b          a,b
 7.     return b                 (none)       b
 8. End:
```



Looking at the solution, notice that `a` and `b` are both live-out of instruction 6. That's because we're assuming that every conditional branch might or might not be taken. A sufficiently clever analysis could notice that the branch can never actually be taken (why not?), so only `b` is "really" live-out from that instruction. But even the cleverest analysis will be unable to guess control flow in all cases, so the liveness analysis is necessarily a conservative approximation: some variables will be considered live even though they aren't really.

Dataflow Analysis

Informal reasoning is fine for simple examples done by hand, but if we want to automate the liveness calculation, we need a systematic approach. There is a general family of analysis techniques called *dataflow analysis* that provides the necessary tools. Dataflow analysis is useful for computing a wide variety of properties over programs that have loops; liveness is just one of its applications.

We start by formalizing the liveness problem using the following definitions:

Suppose we have a CFG with node identifiers n. Write

- $succ[n]$ for the set of successors of node n.

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

A node **uses** a variable if its corresponding instruction mentions that variable in an expression (e.g., on the right-hand side of an assignment, in a conditional, etc.).

Now, for any graph node n, define

- $def[n]$ = set of variables defined by node n;

- $use[n]$ = set of variables used by node n.

Then the live *in* and *out* sets for a node are defined by the following equations:

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

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

It should not be too hard to see why these equations must be obeyed by the live sets. But it is much less obvious that there is an algorithm for computing the sets from the equations. The equations are recursive! It is not immediately clear whether they have a solution, much less a unique one.

In fact, these equations do always have at least one solution, and they have a unique *smallest* solution (i.e. one in which the live-in and live-out sets are as small as possible). This solution can be calculated as the *least fixed point* of the equations, using an iterative approximation algorithm.

The iterative calculation works like this:

- We start by approximating all the sets (for every node) by the empty set, (i.e., assuming that nothing is live anywhere).
- On each iteration, we calculate new approximations for *in* and *out* at each node, by applying the *in* and *out* equations to our most recent previous approximations. This will have the effect of forcing some variables into the sets (because otherwise the equations would be violated), so the sets gradually get larger.
- If we complete an iteration without *any* of the sets changing from the previous iteration, then we have reached a fixed point. No further iterations will change the sets any more. The sets will be a valid solution to the equations, and they are as small as possible (because we've only added variables to them when we had to).

Note that the algorithm must always terminate, because each iteration (except the last one) must increase the size of some set, but the size of each set is bounded by the number of variables in the program.

To illustrate, consider again the calculation of live sets for Example 0.

We start with just the *succ*, *use*, and *def* information. Note that we have now completely abstracted away from the actual program text, and are left with a pure math problem involving sets of names.

Initially, we approximate all *in* and *out* sets by the empty set. Although we can consider the nodes in any order and still get a correct answer, it turns out that if we take the nodes in roughly reverse order, we will need fewer iterations to reach the least fixed point, so we list them in this order.

				initial	
node	*succ*	*use*	*def*	*out*	*in*
8	-	-	-	-	-
7	-	c	-	-	-
6	2,7	a	-	-	-
5	6	b	a	-	-
4	5	bc	c	-	-
3	4	a	b	-	-
2	3	-	-	-	-
1	2	-	a	-	-
0	1	-	-	-	-

In the first iteration, we improve this approximation by applying the equations to each node in turn. We calculate in the order $out[8], in[8], out[7], in[7]$, and so on. We use the *most recent* approximation for each set on the right-hand side of the equation. This might be the approximation from the previous iteration, but often if will be an approximation from an earlier node of *this* iteration. In particular, if node n is a

successor of node $n-1$ (as is often the case), then $in[n]$ must be unioned into the $out[n-1]$ and we use the approximation we've just made of the former.

				initial		1st	
node	*succ*	*use*	*def*	*out*	*in*	*out*	*in*
8	-	-	-	-	-	-	-
7	-	c	-	-	-	-	c
6	2,7	a	-	-	-	c	ac
5	6	b	a	-	-	ac	bc
4	5	bc	c	-	-	bc	bc
3	4	a	b	-	-	bc	ac
2	3	-	-	-	-	ac	ac
1	2	-	a	-	-	ac	c
0	1	-	-	-	-	c	c

Thanks to this ordering trick, we already have an almost complete solution at this stage. The only problem is that we have not updated $out[6]$ to reflect the most recent approximation of $in[2]$. This will be fixed by the second iteration:

				1st		2nd	
node	*succ*	*use*	*def*	*out*	*in*	*out*	*in*
8	-	-	-	-	-	-	-
7	-	c	-	-	c	-	c
6	2,7	a	-	c	ac	ac	ac
5	6	b	a	ac	bc	ac	bc
4	5	bc	c	bc	bc	bc	bc
3	4	a	b	bc	ac	bc	ac
2	3	-	-	ac	ac	ac	ac
1	2	-	a	ac	c	ac	c
0	1	-	-	c	c	c	c

This is in fact the final answer. But to know that, we have to perform another iteration to check that nothing changes, confirming that we have really reached the fixed point:

				2nd		3rd	
node	*succ*	*use*	*def*	*out*	*in*	*out*	*in*
8	-	-	-	-	-	-	-
7	-	c	-	-	c	-	c
6	2,7	a	-	ac	ac	ac	ac
5	6	b	a	ac	bc	ac	bc
4	5	bc	c	bc	bc	bc	bc
3	4	a	b	bc	ac	bc	ac
2	3	-	-	ac	ac	ac	ac
1	2	-	a	ac	c	ac	c
0	1	-	-	c	c	c	c

Exercise: Example 2

Consider the program in file `example2.ir`. Write down the `succ`, *use*, and *def* sets for each node. Then use the iterative algorithm to compute live-in and live-out sets at each node.

Solution: The sole variable (`x`) lives in the live-out sets of nodes 2,3,4,7,8 and the live-in sets of nodes 2,3,4,5,8.