Lab 7: Liveness, Dataflow Analysis, and Register Allocation via Graph Coloring

CS 322 Languages and Compiler Design II, Spring Term 2014
Department of Computer Science, Portland State University

In this lab we will show how to formulate and solve liveness analysis as a dataflow problem, and then see how to use liveness information to formulate and solve the register allocation problem using graph coloring, which will be central to homework 3.

Start by downloading and unzipping `lab7.zip` from D2L in the usual way.

1 Liveness

Recall from lecture 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.

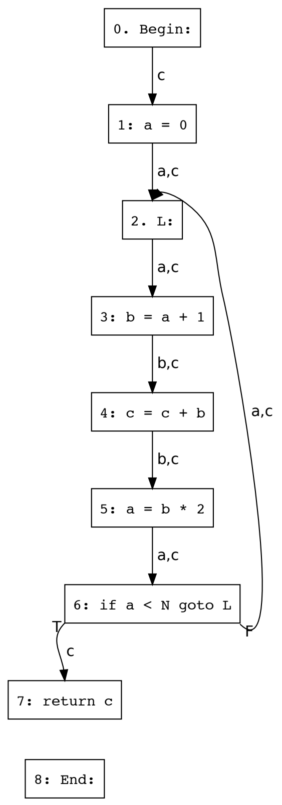
One convenient way to record the live variables of a program is to list which variables are live immediately before and after each instruction. (We say these variables are "live in" and "live out" of the instruction.)

Example 0 Here is the example from lecture, 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. The 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 draw CFG's where the nodes are *basic blocks*, i.e. sequences of straight-line code with no internal jumps. 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).

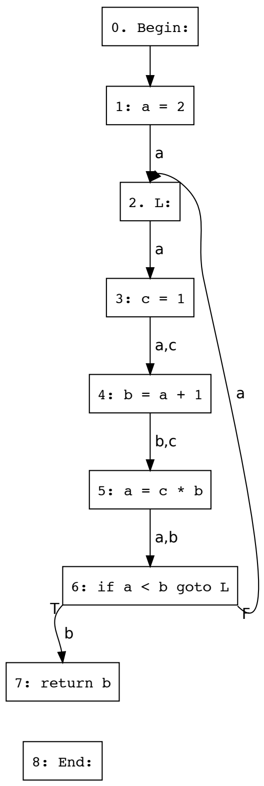
For programs this small, it is not hard to compute liveness informally. At worst, we can ask ourselves the question: for each variable mentioned in the program and each CFG edge, will the variable's current value be used again on some path starting with that edge?

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.

2 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 $\mathit{out}[8], \mathit{in}[8], \mathit{out}[7], \mathit{in}[7]$, 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 $\mathit{in}[n]$ must be unioned into the $\mathit{out}[n-1]$ 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 $\mathit{out}[6]$ to reflect the most recent approximation of $\mathit{in}[2]$. 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.

Applying this algorithm by hand quickly becomes very tedious. Of course, the whole point of defining the algorithm was so that we can implement it within a compiler.

A simple implementation of a liveness analyzer is provided in file `Liveness.java`. To build it, type `javac Liveness.java`; this should also compile the other support files (notably `IR.java`) that it depends on. To run the analyzer on file `foo.ir`, type

```
java -ea Liveness foo.ir
```

and the computed live sets and live ranges will be printed to standard output. Note that the IR code bodies you feed to the generator *must* start and end with labels; otherwise an assertion violation will be raised (the `-ea` flag enables assertion checking in the Java runtime engine.)

This implementation isn't particularly efficient; real compilers often devote significant effort to implementating dataflow analyses using specialized data structures (such as bit vectors to represent sets) to speed things up.

By the way, the order of entries in the live sets and ranges is not meaningful; it is governed by the behavior of the underlying hash tables in the underlying implementation, so is essentially arbitrary.

Exercise: Example 3 Use the `Liveness` program to compute live sets for file `example3.ir`. Confirm that the computed results match your intuition about the live sets. (The best way to do this is to draw a CFG first.)

3 Register Allocation via Graph Coloring

Now that we have liveness information available, we can perform precise register assignment. Recall that the goal of register assignment is to assign a *unique* physical machine register to each IR variable and temporary.

As described in lecture, one way to do this is to translate the assignment problem into a *graph coloring* problem. This has the following steps (for each procedure):

• Construct a *register interference graph*. This graph has a node for each IR variable used in the procedure, and an edge between every two nodes that are simultaneously live (more precisely, that appear together in some live-out set).

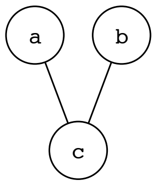
• We try to assign a physical register to each node, such that no two nodes connected by an edge are assigned the same register.

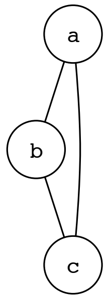
• If we succeed, we get a register assignment that avoids any need to spill registers to memory. If we fail, spilling will be necessary.

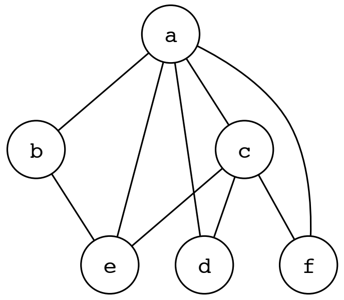
If we think of the different physical registers as colors, then the assignment process is like coloring the nodes of a graph such that no two connected nodes have the same color. This is closely related to the practical problem of coloring the countries on a map such that no two countries with a common border share the same color. (There is a famous mathematical result that says four colors suffice to color any map. Unfortunately, this doesn't imply that four registers are sufficient for any procedure. Why not?)

3.1 Register interference graphs

As examples, here are the interference graphs for Examples 0, 1, and 3.







It is trivial to assign registers for Example 0: only two are needed, with `c` living in one register, say `%r0` and both `a` and `b` living in the other, say `%r1`. Substituting these assignments in for the variables in the original code gives:

```
0. Begin:
1.    %r1 = 0
2. L:
3.    %r1 = %r1 + 1
4.    %r0 = %r0 + %r1
5.    %r1 = %r1 * 2
6.    if %r1 < 1000 goto L
7.    return %r0
8. End:
```

Example 1 obviously requires three registers. If we assign, say `a` to `%r0`, `b` to `%r1` and `c` to `%r2`, the resulting code after substitution is:

```
 0. Begin:
 1.     %r0 = 2
 2. L:
 3.     %r2 = 1
```

```
        %r1 = %r0 + 1
        %r0 = %r2 * %r1
        if %r0 < %r1 goto L
        return %r1
```

Note that Example 1 requires three registers even though no more than two variables are ever live-out at any point! This may seem paradoxical at first. The reason is that we are insisting that each variable be assigned a *unique* register for the entirety of the procedure. It is a worthwhile exercise to try to use just two registers for this example, e.g., using the greedy algorithm described in lecture. Where do things go wrong?

Also note that the Example 1 graph contains 3-clique. A *clique* is a subgraph in which every node is connected to every other node. It should be easy to see that a graph containing k-clique always requires at least k distinct colors.

It is a little less obvious how many registers are needed for Example 3. At least three must be needed (why?) but will they be enough? The answer is yes: if we assign `a` to `%r0`, then `b` and `c` can be assigned to `%r1`, and `d`,`e`, and `f` to `%r2`.

Notice that node `a` is connected to each of the other five nodes; we say that the *degree*(`a`) = 5. But this doesn't make it particularly hard to color the graph.

Exercise Write an example IR program whose interference graph has a node of degree 5, but can be colored with just two colors. Design your program so that it can be easily extended to induce a graph node with arbitrarily high degree, still requiring just two colors. Test your program with `Liveness`.

(Solution: `example4.ir`)

Exercise Write an example IR program whose interference graph requires five colors. Design your program so that it can be easily extended to induce a graph requiring an arbitrarily large number of colors. Test your program with `Liveness`.

(Solution: `example5.ir`)

3.2 Finding a Coloring

So far we have been finding colorings by inspection, but again, of course, we really want to write an algorithm that a compiler can implement to perform this task. If our machine architecture gives us k registers to play with, we want to find a k-coloring, i.e., one that uses no more than k colors. More generally, we might want to find a coloring that uses the minimum number of colors.

Unfortunately, even determining whether a graph has a k-coloring or not is an NP-complete problem, meaning that, as far as anyone currently knows, it requires $O(2^n)$ time, where n is the number of nodes in the graph. Exponential-time algorithms are usually not considered practical inside compilers!

However, there turn out to be simple and fast *heuristic* methods that will usually find a k-coloring if one exists. These methods aren't guaranteed to work, but they usually do a good job on graphs produced from real programs. One simple heuristic procedure to obtain a k-coloring works like this:

1. If the graph is empty, it is trivial to color (with zero colors!)
2. Otherwise, choose a node n from the graph such that *degree*$(n) < k$. If there are no such nodes, the procedure fails.

3. Remove node n and all its incident edges from the graph.
4. Recursively invoke the coloring procedure on the remaining graph. If it succeeds, there must be a k-coloring for that graph.
5. Now add back node n and its edges, and assign it a color. We are guaranteed to be able to do this, because it has fewer than k neighbors, so even if they all have different colors, there will still be at least one color left we can use.

In practice, this procedure is usually implemented using an explicit stack data structure and two loops, rather than via a recursive function. When a node is removed from the graph in step 2, it is pushed onto the stack, together with a list of its neighbors. Steps 1–4 become a loop that terminates when the graph is empty. Step 5 becomes a loop that iterates, popping each node from the stack in turn and assigning it color that doesn't conflict with those of its neighbors (which are guaranteed to have been colored already).

Also, rather than failing at step 2 if we can't find a node with *degree*$(n) < k$, we can just pick (any) node of lowest degree and push it on the stack. When it is popped off, we might get lucky and be able to color it even though it has k or more neighbors, as long as they don't all have distinct colors.

Exercise Walk through how this procedure can be used to find a 3-coloring for the graph of Example 3.