10 Liveness Analysis

live: of continuing or current interest

Webster's Dictionary

The front end of the compiler translates programs into an intermediate language with an unbounded number of temporaries. This program must run on a machine with a bounded number of registers. Two temporaries a and b can fit into the same register, if a and b are never "in use" at the same time. Thus, many temporaries can fit in few registers; if they don't all fit, the excess temporaries can be kept in memory.

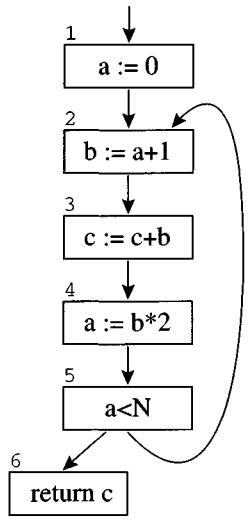
Therefore, the compiler needs to analyze the intermediate-representation program to determine which temporaries are in use at the same time. We say a variable is *live* if it holds a value that may be needed in the future, so this analysis is called *liveness* analysis.

To perform analyses on a program, it is often useful to make a *control-flow graph*. Each statement in the program is a node in the flow graph; if statement x can be followed by statement y, there is an edge from x to y. Graph 10.1 shows the flow graph for a simple loop.

Let us consider the liveness of each variable (Figure 10.2). A variable is live if its current value will be used in the future, so we analyze liveness by working from the future to the past. Variable b is used in statement 4, so b is live on the $3 \to 4$ edge. Since statement 3 does not assign into b, then b is also live on the $2 \to 3$ edge. Statement 2 assigns into b. That means that the contents of b on the $1 \to 2$ edge are not needed by anyone; b is dead on this edge. So the *live range* of b is $\{2 \to 3,\ 3 \to 4\}$.

The variable a is an interesting case. It's live from $1 \to 2$, and again from $4 \to 5 \to 2$, but not from $2 \to 3 \to 4$. Although a has a perfectly

$a \leftarrow 0$
$L_1 : b \leftarrow a + 1$
$c \leftarrow c + b$
$a \leftarrow b * 2$
if $a < N$ goto L_1
return c



GRAPH 10.1. Control-flow graph of a program.

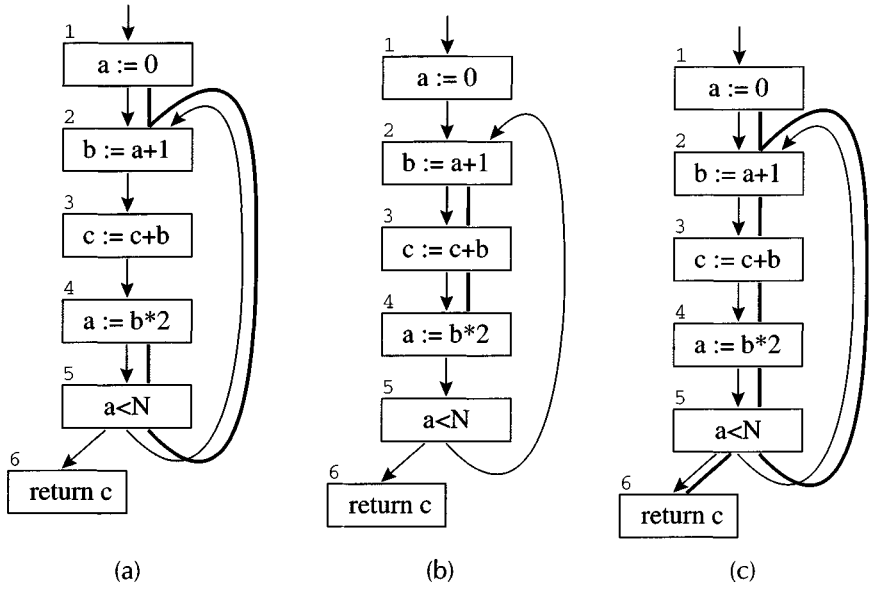


FIGURE 10.2. Liveness of variables a, b, c.

well defined value at node 3, that value will not be needed again before a is assigned a new value.

The variable c is live on entry to this program. Perhaps it is a formal parameter. If it is a local variable, then liveness analysis has detected an uninitialized variable; the compiler could print a warning message for the programmer.

Once all the live ranges are computed, we can see that only two registers are needed to hold a, b, and c, since a and b are never live at the same time. Register 1 can hold both a and b, and register 2 can hold c.

10.1 SOLUTION OF DATAFLOW EQUATIONS

Liveness of variables "flows" around the edges of the control-flow graph; determining the live range of each variable is an example of a *dataflow* problem. Chapter 17 will discuss several other kinds of dataflow problems.

Flow graph terminology. A flow-graph node has *out-edges* that lead to *successor* nodes, and *in-edges* that come from *predecessor* nodes. The set $pred[n]$ is all the predecessors of node n, and $succ[n]$ is the set of successors.

In Graph 10.1 the out-edges of node 5 are $5 \to 6$ and $5 \to 2$, and $succ[5] = \{2, 6\}$. The in-edges of 2 are $5 \to 2$ and $1 \to 2$, and $pred[2] = \{1, 5\}$.

Uses and defs. An assignment to a variable or temporary *defines* that variable. An occurrence of a variable on the right-hand side of an assignment (or in other expressions) *uses* the variable. We can speak of the *def* of a variable as the set of graph nodes that define it; or the *def* of a graph node as the set of variables that it defines; and similarly for the *use* of a variable or graph node. In Graph 10.1, $def(3) = \{c\}$, $use(3) = \{b, c\}$.

Liveness. 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 of the in-edges of that node; it is *live-out* at a node if it is live on any of the out-edges of the node.

CALCULATION OF LIVENESS

Liveness information (*live-in* and *live-out*) can be calculated from *use* and *def* as follows:

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

EQUATIONS 10.3. Dataflow equations for liveness analysis.

for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
repeat
  for each n
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

ALGORITHM 10.4. Computation of liveness by iteration.

1. If a variable is in $use[n]$, then it is *live-in* at node n. That is, if a statement uses a variable, the variable is live on entry to that statement.
2. If a variable is *live-in* at a node n, then it is *live-out* at all nodes m in $pred[n]$.
3. If a variable is *live-out* at node n, and not in $def[n]$, then the variable is also *live-in* at n. That is, if someone needs the value of a at the end of statement n, and n does not provide that value, then a's value is needed even on entry to n.

These three statements can be written as Equations 10.3 on sets of variables. The live-in sets are an array $in[n]$ indexed by node, and the live-out sets are an array $out[n]$. That is, $in[n]$ is all the variables in $use[n]$, plus all the variables in $out[n]$ and not in $def[n]$. And $out[n]$ is the union of the live-in sets of all successors of n.

Algorithm 10.4 finds a solution to these equations by iteration. As usual, we initialize $in[n]$ and $out[n]$ to the the empty set $\{\}$, for all n, then repeatedly treat the equations as assignment statements until a fixed point is reached.

Table 10.5 shows the results of running the algorithm on Graph 10.1. The columns 1st, 2nd, etc. are the values of in and out on successive iterations of the **repeat** loop. Since the 7th column is the same as the 6th, the algorithm terminates.

We can speed the convergence of this algorithm significantly by ordering the nodes properly. Suppose there is an edge $3 \rightarrow 4$ in the graph. Since $in[4]$

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

TABLE 10.5. Liveness calculation following forward control-flow edges.

			1st		2nd		3rd	
	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

TABLE 10.6. Liveness calculation following reverse control-flow edges.

is computed from *out*[4], and *out*[3] is computed from *in*[4], and so on, we should compute the in and out sets in the order *out*[4] → *in*[4] → *out*[3] → *in*[3]. But in Table 10.5, just the opposite order is used in each iteration! We have waited as long as possible (in each iteration) to make use of information gained from the previous iteration.

Table 10.6 shows the computation, in which each **for** loop iterates from 6 to 1 (approximately following the *reversed* direction of the flow-graph arrows), and in each iteration the out sets are computed before the in sets. By the end of the second iteration, the fixed point has been found; the third iteration just confirms this.

When solving dataflow equations by iteration, the order of computation should follow the "flow." Since liveness flows *backward* along control-flow arrows, and from "out" to "in," so should the computation.

Ordering the nodes can be done easily by depth-first search, as shown in Section 17.4.

Basic blocks. Flow-graph nodes that have only one predecessor and one successor are not very interesting. Such nodes can be merged with their predecessors and successors; what results is a graph with many fewer nodes, where each node represents a basic block. The algorithms that operate on flow graphs, such as liveness analysis, go much faster on the smaller graphs. Chapter 17 explains how to adjust the dataflow equations to use basic blocks. In this chapter we keep things simple.

One variable at a time. Instead of doing dataflow "in parallel" using set equations, it can be just as practical to compute dataflow for one variable at a time as information about that variable is needed. For liveness, this would mean repeating the dataflow traversal once for each temporary. Starting from each *use* site of a temporary t, and tracing backward (following *predecessor* edges of the flow graph) using depth-first search, we note the liveness of t at each flow-graph node. The search stops at any definition of the temporary. Although this might seem expensive, many temporaries have very short live ranges, so the searches terminate quickly and do not traverse the entire flow graph for most variables.

REPRESENTATION OF SETS

There are at least two good ways to represent sets for dataflow equations: as arrays of bits or as sorted lists of variables.

If there are N variables in the program, the bit-array representation uses N bits for each set. Calculating the union of two sets is done by *or*-ing the corresponding bits at each position. Since computers can represent K bits per word (with $K = 32$ typical), one set-union operation takes N/K operations.

A set can also be represented as a linked list of its members, sorted by any totally ordered key (such as variable name). Calculating the union is done by merging the lists (discarding duplicates). This takes time proportional to the size of the sets being unioned.

Clearly, when the sets are sparse (fewer than N/K elements, on the average), the sorted-list representation is asymptotically faster; when the sets are dense, the bit-array representation is better.

TIME COMPLEXITY

How fast is iterative dataflow analysis?

A program of size N has at most N nodes in the flow graph, and at most N variables. Thus, each live-in set (or live-out set) has at most N elements;

			X		Y		Z	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

TABLE 10.7. X and Y are solutions to the liveness equations; Z is not a solution.

each set-union operation to compute live-in (or live-out) takes $O(N)$ time.

The **for** loop computes a constant number of set operations per flow-graph node; there are $O(N)$ nodes; thus, the **for** loop takes $O(N^2)$ time.

Each iteration of the **repeat** loop can only make each in or out set larger, never smaller. This is because the in and out sets are *monotonic* with respect to each other. That is, in the equation $in[n] = use[n] \cup (out[n] - def[n])$, a larger $out[n]$ can only make $in[n]$ larger. Similarly, in $out[n] = \bigcup_{s \in succ[n]} in[s]$, a larger $in[s]$ can only make $out[n]$ larger.

Each iteration must add something to the sets; but the sets cannot keep growing infinitely; at most every set can contain every variable. Thus, the sum of the sizes of all in and out sets is $2N^2$, which is the most that the repeat loop can iterate.

Thus, the worst-case run time of this algorithm is $O(N^4)$. Ordering the nodes using depth-first search (Algorithm 17.5, page 390) usually brings the number of **repeat**-loop iterations to two or three, and the live-sets are often sparse, so the algorithm runs between $O(N)$ and $O(N^2)$ in practice.

Section 17.4 discusses more sophisticated ways of solving dataflow equations quickly.

LEAST FIXED POINTS

Table 10.7 illustrates two solutions (and a nonsolution!) to the Equations 10.3; assume there is another program variable d not used in this fragment of the program.

In solution Y, the variable d is carried uselessly around the loop. But in fact, Y satisfies Equations 10.3 just as X does. What does this mean? Is d live or not?

The answer is that any solution to the dataflow equations is a *conservative approximation*. If the value of variable a will truly be needed in some execution of the program when execution reaches node n of the flow graph, then we can be assured that a is live-out at node n in any solution of the equations. But the converse is not true; we might calculate that d is live-out, but that doesn't mean that its value will really be used.

Is this acceptable? We can answer that question by asking what use will be made of the dataflow information. In the case of liveness analysis, if a variable is *thought to be live* then we will make sure to have its value in a register. A conservative approximation of liveness is one that may erroneously believe a variable is live, but will never erroneously believe it is dead. The consequence of a conservative approximation is that the compiled code might use more registers than it really needs; but it will compute the right answer.

Consider instead the live-in sets Z, which fail to satisfy the dataflow equations. Using this Z we think that b and c are never live at the same time, and we would assign them to the same register. The resulting program would use an optimal number of registers but *compute the wrong answer*.

A dataflow equation used for compiler optimization should be set up so that any solution to it provides conservative information to the optimizer; imprecise information may lead to suboptimal but never incorrect programs.

Theorem. Equations 10.3 have more than one solution.

Proof. X and Y are both solutions.

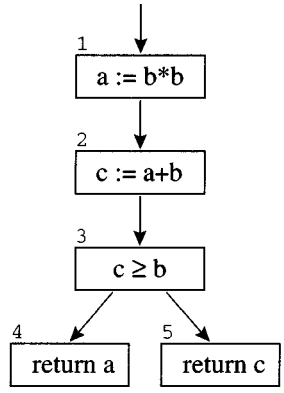
Theorem. All solutions to Equations 10.3 contain solution X. That is, if $in_X[n]$ and $in_Y[n]$ are the live-in sets for some node n in solutions X and Y, then $in_X[n] \subseteq in_Y[n]$.

Proof. See Exercise 10.2.

We say that X is the *least solution* to Equations 10.3. Clearly, since a bigger solution will lead to using more registers (producing suboptimal code), we want to use the least solution. Fortunately, Algorithm 10.4 always computes the least fixed point.

STATIC VS. DYNAMIC LIVENESS

A variable is live "if its value will be used in the future." In Graph 10.8, we know that $b \times b$ must be nonnegative, so that the test $c \geq b$ will be true. Thus,



GRAPH 10.8. Standard static dataflow analysis will not take advantage of the fact that node 4 can never be reached.

node 4 will never be reached, and a's value will not be used after node 2; a is not live-out of node 2.

But Equations 10.3 say that a is live-in to node 4, and therefore live-out of nodes 3 and 2. The equations are ignorant of which way the conditional branch will go. "Smarter" equations would permit a and c to be assigned the same register.

Although we can prove here that $b*b \geq 0$, and we could have the compiler look for arithmetic identities, no compiler can ever fully understand how all the control flow in every program will work. This is a fundamental mathematical theorem, derivable from the halting problem.

Theorem. There is no program H that takes as 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 that there were such a program H; then we could arrive at a contradiction as follows. From the program H, construct the function F,

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

By the definition of H, if $F(F)$ halts, then $H(F, F)$ is true; so the **then** clause is taken; so the **while** loop executes forever; 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. The program $F(F)$ halts if it doesn't halt, and doesn't halt if

it halts: a contradiction. Thus there can be no program H that tests whether another program halts (and always halts itself).

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

Proof. From H' we could construct H. In some program that we want to test for halting, just let L be the end of the program, and replace all instances of the **halt** command with **goto** L.

Conservative approximation. This theorem does not mean that we can *never* tell if a given label is reached or not, just that there is not a general algorithm that can *always* tell. We could improve our liveness analysis with some special-case algorithms that, in some cases, calculate more information about run-time control flow. But any such algorithm will come up against many cases where it simply cannot tell exactly what will happen at run time.

Because of this inherent limitation of program analysis, no compiler can really tell if a variable's value is truly needed – whether the variable is truly live. Instead, we have to make do with a conservative approximation. We assume that any conditional branch goes both ways. Thus, we have a dynamic condition and its static approximation:

Dynamic liveness A variable a is dynamically live at node n if some execution of the program goes from n to a use of a without going through any definition of a.

Static liveness A variable a is statically live at node n if there is some path of control-flow edges from n to some use of a that does not go through a definition of a.

Clearly, if a is dynamically live it is also statically live. An optimizing compiler must allocate registers, and do other optimizations, on the basis of static liveness, because (in general) dynamic liveness cannot be computed.

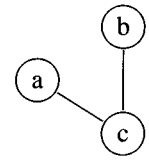
INTERFERENCE GRAPHS

Liveness information is used for several kinds of optimization in a compiler. For some optimizations, we need to know exactly which variables are live at each node in the flow graph.

One of the most important applications of liveness analysis is for register allocation: we have a set of temporaries $a, b, c, \ldots$ that must be allocated to

	a	b	c
a			x
b			x
c	x	x	

(a) Matrix



(b) Graph

FIGURE 10.9. Representations of interference.

registers $r_1, \ldots, r_k$. A condition that prevents a and b being allocated to the same register is called an *interference*.

The most common kind of interference is caused by overlapping live ranges: when a and b are both live at the same program point, then they cannot be put in the same register. But there are some other causes of interference: for example, when a must be generated by an instruction that cannot address register r_1, then a and r_1 interfere.

Interference information can be expressed as a matrix; Figure 10.9a has an **x** marking interferences of the variables in Graph 10.1. The interference matrix can also be expressed as an undirected graph (Figure 10.9b), with a node for each variable, and edges connecting variables that interfere.

Special treatment of MOVE instructions. In static liveness analysis, we can give MOVE instructions special consideration. It is important not to create artifical interferences between the source and destination of a move. Consider the program:

$t \leftarrow s$ (*copy*)

$\vdots$

$x \leftarrow \ldots s \ldots$ (*use of s*)

$\vdots$

$y \leftarrow \ldots t \ldots$ (*use of t*)

After the copy instruction both s and t are live, and normally we would make an interference edge (s, t) since t is being defined at a point where s is live. But we do not need separate registers for s and t, since they contain the same value. The solution is just not to add an interference edge (t, s) in this case. Of course, if there is a later (nonmove) definition of t while s is still live, that will create the interference edge (t, s).

Therefore, the way to add interference edges for each new definition is:

```
signature GRAPH =
sig
    type graph
    type node

    val nodes: graph -> node list
    val succ: node -> node list
    val pred: node -> node list
    val adj: node -> node list
    val eq: node*node -> bool

    val newGraph: unit -> graph
    val newNode : graph -> node
    exception GraphEdge
    val mk_edge: {from: node, to: node} -> unit
    val rm_edge: {from: node, to: node} -> unit

    structure Table : TABLE
    sharing type Table.key = node

    val nodename: node->string   (* for debugging *)
end
```

PROGRAM 10.10. The `Graph` abstract data type.

1. At any nonmove instruction that *defines* a variable a, where the *live-out* variables are $b_1, \ldots, b_j$, add interference edges $(a, b_1), \ldots, (a, b_j)$.
2. At a move instruction $a \leftarrow c$, where variables $b_1, \ldots, b_j$ are *live-out*, add interference edges $(a, b_1), \ldots, (a, b_j)$ for any b_i that is *not* the same as c.

10.2 LIVENESS IN THE Tiger COMPILER

The flow analysis for the Tiger compiler is done in two stages: first, the control flow of the `Assem` program is analyzed, producing a control-flow graph; then, the liveness of variables in the control-flow graph is analyzed, producing an interference graph.

GRAPHS

To represent both kinds of graphs, let's make a `Graph` abstract data type (Program 10.10).

The function `newGraph` creates an empty directed graph; `newNode(g)` makes a new node within a graph g. A directed edge from n to m is created

by `mk_edge(n,m)`; after that, m will be found in the list `succ(n)` and n will be in `pred(m)`. When working with undirected graphs, the function `adj` is useful: `adj`(m) = `succ`(m) ∪ `pred`(m).

To delete an edge, use `rm_edge`. To test whether m and n are the same node, use `eq(m,n)`.

When using a graph in an algorithm, we want each node to represent something (an instruction in a program, for example). To make mappings from nodes to the things they are supposed to represent, we use a `table`. The following idiom associates information x with node n in a mapping `mytable`.

```
Graph.Table.enter(mytable,n,x)
```

CONTROL-FLOW GRAPHS

The `Flow` structure manages control-flow graphs. Each instruction (or basic block) is represented by a node in the flow graph. If instruction m can be followed by instruction n (either by a jump or by falling through), then there will be an edge (m, n) in the graph.

```
structure Flow :
sig
    structure Graph
    datatype flowgraph =
      FGRAPH of {control: Graph.graph,
                 def: Temp.temp list Graph.Table.table,
                 use: Temp.temp list Graph.Table.table,
                 ismove: bool Graph.Table.table}
end
```

A flow graph has four components:

`control` a directed graph wherein each node represents an instruction (or, perhaps, a basic block);

`def` a table of the temporaries defined at each node (destination registers of the instruction);

`use` a table of the temporaries used at each node (source registers of the instruction);

`ismove` tells whether each instruction is a MOVE instruction, which could be deleted if the `def` and `use` are identical.

The `MakeGraph` module turns a list of `Assem` instructions into a flow graph.

```
structure MakeGraph:
sig
   val instrs2graph: Assem.instr list ->
                  Flow.flowgraph * Flow.Graph.node list
end
```

The function `instrs2graph` takes a list of instructions and returns a flow graph, along with a list of nodes that corresponds exactly to the instructions. In making the flow graph, the `jump` fields of the `instrs` are used in creating control-flow edges, and the `use` and `def` information (obtained from the `src` and `dst` fields of the `instrs`) is attached to the nodes by means of the `use` and `def` tables of the `flowgraph`.

Information associated with the nodes. For a flow graph, we want to associate some *use* and *def* information with each node in the graph. Then the liveness-analysis algorithm will also want to remember *live-in* and *live-out* information at each node. We could make room in the `node` record to store all of this information. This would work well and would be quite efficient. However, it may not be very modular. Eventually we may want to do other analyses on flow graphs, which remember other kinds of information about each node. We may not want to modify the data structure (which is a widely used interface) for each new analysis.

Instead of storing the information *in* the nodes, a more modular approach is to say that a graph is a graph, and that a flow graph is a graph along with separately packaged auxiliary information (tables, or functions mapping nodes to whatever). Similarly, a dataflow algorithm on a graph does not need to modify dataflow information *in* the nodes, but modifies its own privately held mappings.

There may be a trade-off here between efficiency and modularity, since it may be faster to keep the information *in* the nodes, accessibly by a simple pointer-traversal instead of a hash-table or search-tree lookup.

LIVENESS ANALYSIS

The `Liveness` module of the compiler has a function `interferenceGraph` that takes a flow graph and returns two kinds of information: an interference graph (`igraph`) and a table mapping each flow-graph node to the set of temps that are *live-out* at that node.

```
structure Liveness:
sig
    datatype igraph =
        IGRAPH of {graph: IGraph.graph,
                   tnode: Temp.temp -> IGraph.node,
                   gtemp: IGraph.node -> Temp.temp,
                   moves: (IGraph.node * IGraph.node) list}

    val interferenceGraph :
           Flow.flowgraph ->
              igraph * (Flow.Graph.node -> Temp.temp list)

    val show : outstream * igraph -> unit
end
```

The components of an `igraph` are

`graph` the interference graph;
`tnode` a mapping from temporaries of the `Assem` program to graph nodes;
`gtemp` the inverse mapping, from graph nodes back to temporaries;
`moves` a list of move instructions. This is a hint for the register allocator; if the "move instruction" (m, n) is on this list, it would be nice to assign m and n the same register if possible.

The function `show` just prints out – for debugging purposes – a list of nodes in the interference graph, and for each node, a list of nodes adjacent to it.

In the implementation of the `Liveness` module, it is useful to maintain a data structure that remembers what is live at the exit of each flow-graph node:

```
type liveSet = unit Temp.Table.table * temp list
type liveMap = liveSet Flow.Graph.Table.table
```

Given a flow-graph node n, the set of live temporaries at that node can be looked up in a global `liveMap`.This set has a redundant representation: the `table` is useful for efficient membership tests, and the `list` is useful for enumerating all the live variables in the set.

Having calculated a complete `liveMap`, we can now construct an interference graph. At each flow node n where there is a newly defined temporary $d \in def(n)$, and where temporaries $\{t_1, t_2, \ldots\}$ are in the `liveMap`, we just add interference edges $(d, t_1), (d, t_2), \ldots$. For MOVEs, these edges will be safe but suboptimal; pages 221–222 describe a better treatment.

What if a newly defined temporary is not live just after its definition? This would be the case if a variable is defined but never used. It would seem that there's no need to put it in a register at all; thus it would not interfere with any

other temporaries. But if the defining instruction is going to execute (perhaps it is necessary for some other side effect of the instruction), then it *will* write to some register, and that register had better not contain any other live variable. Thus, zero-length live ranges *do* interfere with any live ranges that overlap them.

PROGRAM CONSTRUCTING FLOW GRAPHS

Implement the `MakeGraph` module that turns a list of `Assem` instructions into a flow graph. Use the `Graph` structure provided in `$TIGER/chap10`.

PROGRAM LIVENESS

Implement the `Liveness` module. Use either the set-equation algorithm with the array-of-boolean or sorted-list-of-temporaries representation of sets, or the one-variable-at-a-time method.

EXERCISES

10.1 Perform flow analysis on the program of Exercise 8.6:

a. Draw the control-flow graph.

b. Calculate live-in and live-out at each statement.

c. Construct the register interference graph.

****10.2** Prove that Equations 10.3 have a least fixed point and that Algorithm 10.4 always computes it.

Hint: We know the algorithm refuses to terminate until it has a fixed point. The questions are whether (a) it must eventually terminate, and (b) the fixed point it computes is smaller than all other fixed points. For (a) show that the sets can only get bigger. For (b) show by induction that at any time, the *in* and *out* sets are subsets of those in any possible fixed point. This is clearly true initially, when *in* and *out* are both empty; show that each step of the algorithm preserves the invariant.

***10.3** Analyze the asymptotic complexity of the one-variable-at-a-time method of computing dataflow information.

***10.4** Analyze the worst-case asymptotic complexity of making an interference graph, for a program of size N (with at most N variables and at most N control-flow nodes). Assume the dataflow analysis is already done and that *use*, *def*, and *live-out* information for each node can be queried in constant time. What representation of graph adjacency matrices should be used for efficiency?

10.5 The DEC Alpha architecture places the following restrictions on floating-point instructions, for programs that wish to recover from arithmetic exceptions:

1. Within a basic block (actually, in any sequence of instructions not separated by a *trap-barrier* instruction), no two instructions should write to the same destination register.
2. A source register of an instruction cannot be the same as the destination register of that instruction or any later instruction in the basic block.

$r_1 + r_5 \rightarrow r_4$	$r_1 + r_5 \rightarrow r_4$	$r_1 + r_5 \rightarrow r_3$	$r_1 + r_5 \rightarrow r_4$
$r_3 \times r_2 \rightarrow r_4$	$r_4 \times r_2 \rightarrow r_1$	$r_4 \times r_2 \rightarrow r_4$	$r_4 \times r_2 \rightarrow r_6$
violates rule 1.	*violates rule 2.*	*violates rule 2.*	*OK*

Show how to express these restrictions in the register interference graph.