CS 745, Spring 2015
Homework Assignment 2

Assigned: Thursday, January 29
Due: Thursday, February 12, 9:00AM
Updated: Tuesday, February 3 (version 3)

Introduction

In class, we discussed many interesting data flow analyses such as Liveness, Reaching Definitions, and Available Expressions. Although these analyses are different in certain ways, for example they compute different program properties and analyze the program in different directions (forwards, backwards), they share some common properties such as iterative algorithms, transfer functions, and meet operators. These commonalities make it worthwhile to write a generic framework that can be parameterized appropriately for solving a specific data flow analysis. In this assignment, you and your partner will implement such an iterative data flow analysis framework in LLVM, and use it to implement a forward data flow analysis (Available Expressions) and a backward data flow analysis (Liveness). Although Liveness and Available Expressions implementations are available in some form in LLVM, they are not of the iterative flavor. The objective of this assignment is to create a generic framework for solving iterative bit-vector dataflow analysis problems, and use it to implement Liveness and Available Expressions analysis.

Policy

You will work in groups of two people to solve the problems for this assignment. Turn in a single writeup per group, indicating all group members.

Logistics

Any clarifications and revisions to the assignment will be posted on Piazza.

In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15745-s15/public`

and *ASSTDIR* refers to the subdirectory *HOMEDIR*/`asst/asst2`.

1 Iterative Data Flow Analysis Framework

A well-written iterative data flow analysis framework significantly reduces the burden of implementing new data flow passes; the developer only writes pass specific details such as the meet operator, transfer function, analysis direction, etc. In particular, the framework should solve any unidirectional data flow analysis as long as the analysis supplies the following:

1. Domain, including the semi-lattice

2. Direction (forwards or backwards)
3. Transfer function
4. Meet operation
5. Boundary condition
6. Initial interior points (Top)

To simplify the design process, the domain of values can be represented as bit vectors so that the semi-lattice and set operations (union, intersection) are easy to implement. You are not required to use bit vectors, but doing so is recommended. Careful thought should be given to how the analysis parameters are represented. For example, *direction* could reasonably be represented as a boolean, while function pointers may seem more appropriate for representing transfer functions.

It will be worth your while to do a good job on this assignment because you will be reusing this framework in Assignment 3.

2 Data Flow Analyses

You will now use your iterative data flow analysis framework to implement Liveness and Available Expressions. As explained below in more detail, each analysis should perform computation at program points. As defined in class, program points are assumed to lie *between* instructions (not in the middle of instructions). Please assume that `mem2reg` pass is always run on code your passes receive.

Liveness Upon convergence, your Liveness pass should report all variables that are "live" at each program point. A useful debugging strategy might be to use results of the LLVM Liveness pass as a reference. Please call this pass "liveness".

For this assignment, we will only track the liveness of instruction-defined values and function arguments. That is, when determining what values are used by an instruction, you will use code like this:

```
User::op_iterator OI, OE;
for (OI = insn->op_begin(), OE = insn->op_end(); OI != OE; ++OI)
{
  Value *val = *OI;
  if (isa<Instruction>(val) || isa<Argument>(val)) {
    // val is used by insn
  }
}
```

You should carefully consider how your analysis passes are affected by ϕ instructions. For example, your passes should not output results for the program point preceding a *phi* instruction since they are pseudo instructions which will not appear in the executable. To guide you in formatting the output of your passes, the expected output of running Liveness analysis on the

```
int sum (int a, int b)
{
 int i;
 int res = 1;

 for (i = a; i < b; i++)
 {
    res *= i;
 }
 return res;
}
```

(a)

```
define i32 @sum(i32 %a, i32 %b) nounwind readnone ssp {
entry:
  %0 = icmp slt i32 %a, %b
  br i1 %0, label %bb.nph, label %bb2

bb.nph: ; preds = %entry
  %tmp = sub i32 %b, %a
  br label %bb

bb:     ; preds = %bb, %bb.nph
  %indvar = phi i32 [ 0, %bb.nph ], [ %indvar.next, %bb ]
  %res.05 = phi i32 [ 1, %bb.nph ], [ %1, %bb ]
  %i.04 = add i32 %indvar, %a
  %1 = mul nsw i32 %res.05, %i.04
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp eq i32 %indvar.next, %tmp
  br i1 %exitcond, label %bb2, label %bb

bb2:    ; preds = %bb, %entry
  %res.0.lcssa = phi i32 [ 1, %entry ], [ %1, %bb ]
  ret i32 %res.0.lcssa
```

(b)

Figure 1: (a) Simple loop code, and (b) corresponding optimized (-O) LLVM bytecode.

bytecode from Figure 1(b) is shown in Table 1. Note that you do not need to match the output exactly in terms of commas, spacing, etc.

The fact that you will be working on code in SSA form means that computed values are never destroyed. This will have ramifications for how your passes are implemented. Think carefully about what this means to your implementation.

Available Expressions Upon convergence, your Available Expressions pass should report all the binary expressions that are "available" each program point. Please call this pass "available".

Please note the following details and simplifications:

- For this assignment, we are only concerned with expressions represented by an instance of `BinaryOperator`. Analyzing comparison instructions and unary instructions (e.g., negation) is not required. See the example output in Table 2.
- We will consider two expressions equal if the instructions that calculate these expression share the same opcode, first operand, and second operand. To make reasoning about equivalent expressions easier, we have provided an `Expression` class that performs some of the comparison (and pretty printing) logic for you.

  Side note about comparisons in LLVM: LLVM works hard to make sure that you can compare pointers (say two Value *s) with the expected results. Unless you do additional work, however, you will not be able to compare two `Expression` pointers. You can, however, compare the `Expression` objects themselves.
- Do not worry about commutative expressions. You may, for example, consider `x+y` and `y+x` to be separate expressions.

	define i32 @sum(i32 %a, i32 %b) nounwind readnone ssp {
	entry:
{%a,%b}	
	%0 = icmp slt i32 %a, %b
{%a,%b,%0}	
	br i1 %0, label %bb.nph, label %bb2
	bb.nph: ; preds = %entry
{%a,%b}	
	%tmp = sub i32 %b, %a
{%a,%tmp}	
	br label %bb
	bb: ; preds = %bb, %bb.nph
	%indvar = phi i32 [0, %bb.nph], [%indvar.next, %bb]
	%res.05 = phi i32 [1, %bb.nph], [%1, %bb]
{%a,%tmp,%indvar,%res.05}	
	%i.04 = add i32 %indvar, %a
{%a,%tmp,%indvar,%res.05,%i.04}	
	%1 = mul nsw i32 %res.05, %i.04
{%a,%tmp,%indvar,%1}	
	%indvar.next = add i32 %indvar, 1
{%a,%tmp,%1,%indvar.next}	
	%exitcond = icmp eq i32 %indvar.next, %tmp
{%a,%tmp,%1,%indvar.next,%exitcond}	
	br i1 %exitcond, label %bb2, label %bb
	bb2: ; preds = %bb, %entry
	%res.0.lcssa = phi i32 [1, %entry], [%1, %bb]
{%res.0.lcssa}	
	ret i32 %res.0.lcssa
{}	
	}

Table 1: Output of Liveness on the bitcode in Figure 1(b).

	entry:
{}	
	%add = add nsw i32 %argc, 50
{%argc+50}	
	%add1 = add nsw i32 %add, 96
{%argc+50, %add+96}	
	%cmp = icmp slt i32 50, %add
{%argc+50, %add+96}	
	br i1 %cmp, label %if.then, label %if.else
{%argc+50, %add+96}	
	if.then:
	%sub = sub nsw i32 %add, 50
{%argc+50, %add+96, %add-50}	
	%mul = mul nsw i32 96, %add
{%argc+50, %add+96, %add-50, 96*%add}	
	br label %if.end
{%argc+50, %add+96, %add-50, 96*%add}	
	if.else:
	%add2 = add nsw i32 %add, 50
{%argc+50, %add+96, %add+50}	
	%mul3 = mul nsw i32 96, %add
{%argc+50, %add+96, %add+50, 96*%add}	
	br label %if.end
{%argc+50, %add+96, %add+50, 96*%add}	
	if.end:
	%f.0 = phi i32 [%sub, %if.then], [%add2, %if.else]
	%sub4 = sub nsw i32 50, 96
{%argc+50, %add+96, 96*%add, 50-96}	
	%add5 = add nsw i32 %sub4, %f.0
{%sub4+%f.0, %argc+50, %add+96, 96*%add, 50-96}	
	ret i32 0
{%sub4+%f.0, %argc+50, %add+96, 96*%add, 50-96}	

Table 2: Output of Available Expressions on bitcode for source in Figure 2.

```
int main(int argc, char * argv[]) {
  int a,b,c,d,e,f;
  a = 50;
  b = argc+a;
  c = 96;
  e = b + c;
  if (a < b) {
    f = b-a;
    d = c*b;
  }
  else {
    f = b+a;
    e = c*b;
  }
  b = a-c;
  d = b+f;
  return 0;
}
```

Figure 2: Source code for Available Expressions Example

3 Questions

3.1 Lazy Code Motion

Suppose you were processing the program illustrated by the pseudo-code in Listing 1. Assume that x, y, and z are initialized prior to the code reaching the first statement in the code, and are not constants.

1. Build the CFG for this code, indicating which instructions from the original code will be in each basic block. You may indicate the instructions using the line number for that line of code in parentheses (for example (1) for "d=c+3" on the first line below). Also indicate which expressions are anticipated on each edge, based upon the algorithm described in class.
2. Show the CFG after the Early Placement pass. You may apply constant folding at this time.
3. Show the CFG after the Lazy Code Motion and Cleanup passes.

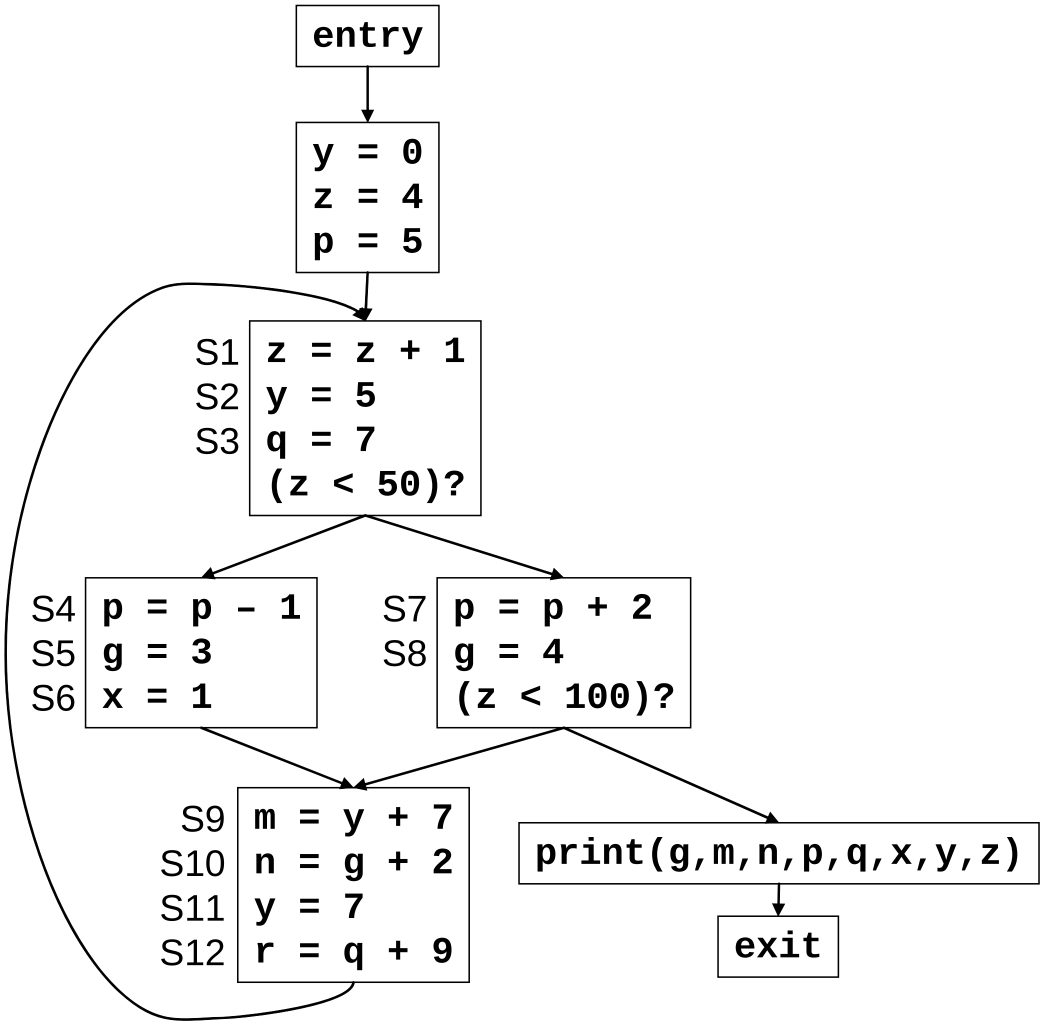
All CFGs may be drawn either via a computer or by hand, but they must be legible in the writeup.

```
1  d=c+3;
2  if (b>5) {
3      e=a+d;
4      c=4;
5  } else {
6      b=b-9;
7  }
8  e=a+d;
9  return e;
```

Listing 1: Source code for question 3.1

3.2 LICM: Loop Invariant Code Motion

For the following code, clearly (i) list the loop invariant instructions, and (ii) clearly indicate why each may or may not be moved to the pre-header by a loop invariant code motion pass.



4 Hand In

Electronic submission: The source code for your framework and passes, the associated `Makefiles`, your test cases, and a `README` describing how to build and run them. Do this by creating a tar file with the last name of at least one of your group members in the filename, and copying this tar file into the directory

```
ASSTDIR/handin
```

Include as comments near the beginning of your source files the identities of all members of your group. Also remember to do a good job of commenting your code.

Hard-copy submission:

1. A report that briefly describes the design and implementation of your framework and passes, and how you tested it. In particular, describe the interface of your framework clearly, so that someone else (e.g., the grader) could write a pass that will work with it.
2. Your answers to the questions in part 3.
3. A listing of your source code.