Homework #2

15-745 Spring 2017

Assigned: Wednesday, February 1
Due: Wednesday, February 22, 11:59PM

Abstract

In class, we discussed many interesting dataflow analyses such as Liveness, Reaching Definitions, and Available Expressions. Although these analyses may differ by computing different program properties and performing analysis in opposing 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 dataflow analysis. In this assignment, you and your partner will implement such an iterative dataflow analysis framework in LLVM, and use it to implement a forward dataflow analysis (Available Expressions) and a backward dataflow 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.

1 Introduction

1.1 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.

1.2 Logistics

All clarifications (if any) to this assignment will be posted on the class discussion board on Piazza. Any revisions will be uploaded to the class web page.

1.3 Submission

Please include the following items in an archive labeled with the Andrew ID of one member in your group (e.g., `bovik.tar.gz`), and submit the resulting file to Blackboard. Ensure that when this archive is extracted, the files appear as follows:

```
./bovik/README
./bovik/Dataflow/Available.cpp
./bovik/Dataflow/AvailableSupport.cpp
./bovik/Dataflow/AvailableSupport.h
./bovik/Dataflow/Dataflow.cpp
./bovik/Dataflow/Dataflow.h
```

```
./bovik/Dataflow/Liveness.cpp
./bovik/Dataflow/Makefile
./bovik/writeup.pdf
./bovik/tests/
```

- A report that briefly describes the implementations of both passes, and answers the homework questions, named `writeup.pdf`, containing the Andrew ID's of all members in your group.
- Well-commented source code for your passes (`Liveness` and `AvailableExpressions`), and associated `Makefile`s.
- A `README` file describing how to build and run your passes.
- Any tests used for verification of your code.

2 Dataflow Analysis

2.1 Iterative Framework

A well-written iterative data flow analysis framework significantly reduces the burden of implementing new dataflow passes; the developer only needs to write pass specific components such as the meet operator, transfer function, analysis direction, etc. In particular, the framework should be capable of solving any unidirectional dataflow analysis as long as the following are defined:

1. Domain, including the semi-lattice
2. Direction (forwards or backwards)
3. Transfer function
4. Meet operation
5. Boundary condition (entry or exit)
6. Initial interior points (in or out)

To simplify the design process, the domain of values can be represented as bitvectors so that the semi-lattice and set operations (union, intersection) are performant and easy to implement. You are not required to use bitvectors, 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.

In particular, note that it will be worth your while to do a good job on this assignment because you will be reusing this framework in Assignment 3.

2.2 Analysis Passes

You will now use your iterative dataflow framework to implement Liveness and Available Expressions. As explained below in more detail, each analysis should perform computation at program points, which were defined in class to lie *between* instructions (not in the middle of instructions).

You may assume that the input received by your pass has already been transformed by the `mem2reg` pass.

2.2.1 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**.

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 expected output of running the available expressions pass on the bytecode from Fig. 5 shown 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. This means that you do not need to worry about commutative expressions. For example, you can consider `x + y` and `y + x` to be separate expressions.

Note: LLVM works hard to make sure that you can compare pointers (say two `Value *`'s) with the expected results. However, unless you do additional work, you will not be able to directly compare two `Expression` pointers. You can, however, compare the `Expression` objects themselves.

2.2.2 Liveness

Upon convergence, your liveness pass should report all variables that are "live" at each program point. Please call your pass **liveness**.

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

```
1 for (User::op_iterator OI = insn->op_begin(); OI != insn->op_end(); ++OI) {
2     Value *val = *OI;
3     if (isa<Instruction>(val) || isa<Argument>(val)) {
4         // val is used by insn
5     }
6 }
```

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 ϕ 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 bytecode from Fig. 3 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.

3 Homework Questions

3.1 Loop Invariant Code Motion (Purple Dragon Book 9.5.1)

Suppose that you are given the code shown below in Figure 1.

(a) List the loop invariant instructions.

(b) Indicate if each loop invariant instruction can be moved to the loop preheader, and give a brief justification.

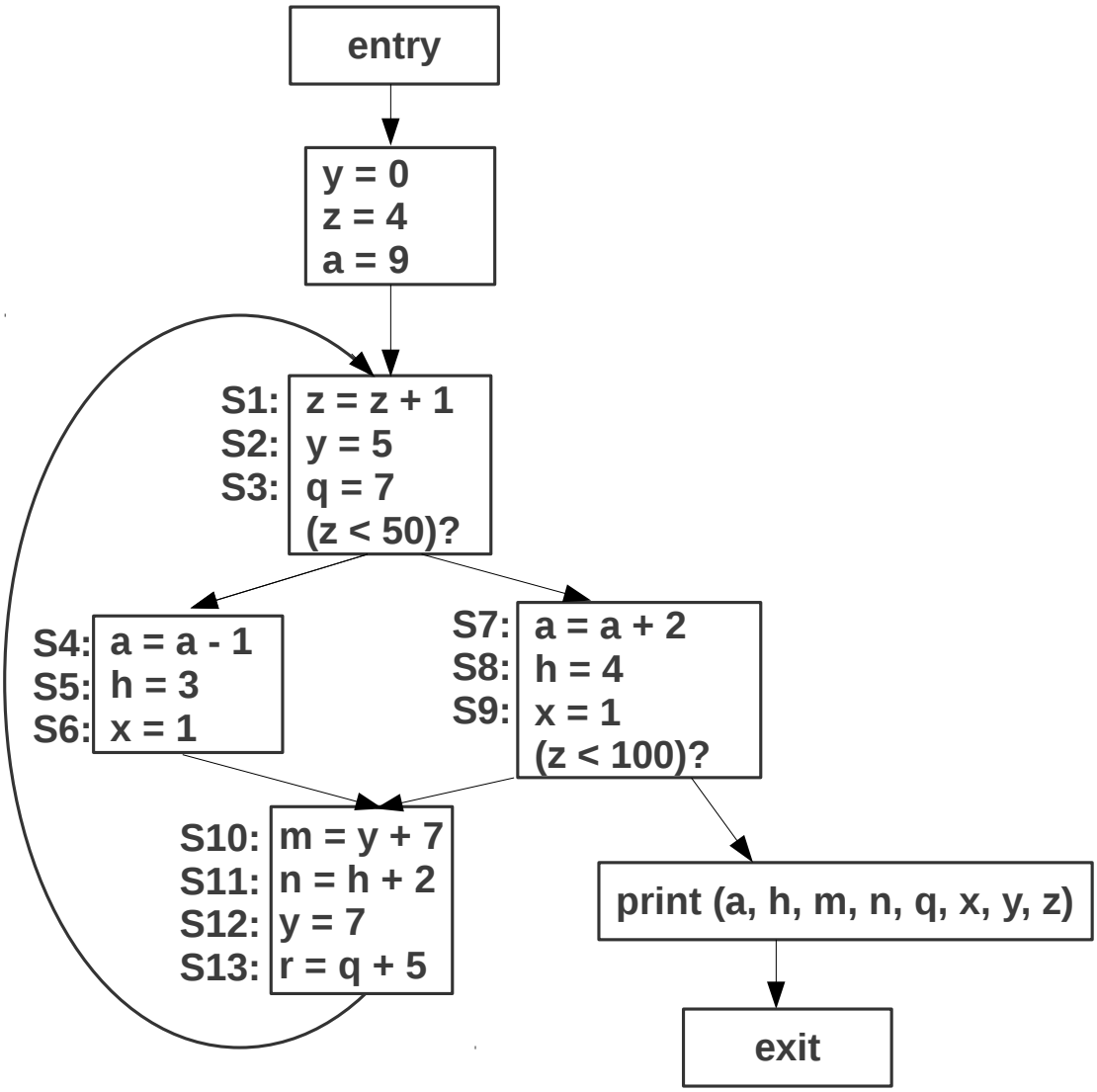

Figure 1: Labeled control-flow graph for question 3.1.

3.2 Lazy Code Motion (Purple Dragon Book 9.5.5)

Suppose that you are optimizing the code shown below in Listing 1.

(a) Build the control-flow graph for this code, indicating which instructions from the original code will be in each basic block. Using the algorithm described in class and in the textbook, provide anticipated expressions for each basic block.

(b) Now, provide available expressions for each basic block, and indicate the earliest basic block for each expression, if applicable.

(c) Next, provide postponable expressions and used expressions for each basic block, and indicate the latest basic block for each expression, if applicable.

(d) Complete the final pass of lazy code motion by inserting and replacing expressions. Provide the finished control-flow graph, and label each basic block with its instruction(s).

```
1  fun foo(x, y, z) {
2      d = z + 3;
3      f = 10;
4      if (y > 5) {
5          e = x + d;
6          z = 4;
7      } else {
8          y = y - 9;
9      }
10     e = x + d;
11     return e;
12 }
```

Listing 1: Source code for question 3.2.

4 Appendix

4.1 Liveness: Example Input

```
1 int sum (int a, int b) {
2     int res = 1;
3
4     for (int i = a; i < b; i++) {
5         res *= i;
6     }
7     return res;
8 }
```

Listing 2: Example input code.

```
1  define i32 @sum( i32 %a , i32 %b) nounwind readnone ssp {
2  entry:
3      %0 = icmp slt i32 %a , %b
4      br i1 %0, label %bb.nph , label %bb2
5
6  bb.nph: ; preds = %entry
7      %tmp = sub i32 %b , %a
8      br label %bb
9
10 bb: ; preds = %bb , %bb.nph
11     %indvar = phi i32 [ 0 , %bb.nph ] , [ %indvar.next , %bb ]
12     %res.05 = phi i32 [ 1 , %bb.nph ] , [ %1, %bb ]
13     %i.04 = add i32 %indvar , %a
14     %1 = mul nsw i32 %res.05 , %i.04
15     %indvar.next = add i32 %indvar, 1
16     %exit cond = icmp eq i32 %indvar.next , %tmp
17     br i1 %exitcond , label %bb2 , label %bb
18
19 bb2: ; preds = %bb , %entry
20     %res.0.lcssa = phi i32 [ 1 , %entry ] , [ %1, %bb ]
21     ret i32 %res.0.lcssa
```

Listing 3: Example simplified input bitcode.

4.2 Liveness: Example Output

	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: Result of the liveness pass on the example input from Listing 2.

4.3 Available Expressions: Example Input

```
1  int main(int argc, char *argv[]) {
2      int a, b, c, d, e, f;
3      a = 50;
4      b = argc + a;
5      c = 96;
6      e = b + c;
7      if (a < b) {
8          f = b - a;
9          d = c * b;
10     }
11     else {
12         f = b + a;
13         e = c * b;
14     }
15     b = a - c;
16     d = b + f;
17     return 0;
18 }
```

Listing 4: Example input code.

```
1  entry:
2      %add = add nsw i32 %argc , 50
3      %add1 = add nsw i32 %add , 96
4      %cmp = icmp slt i32 50 , %add
5      br i1 %cmp , label %if.then , label% if.else
6
7  if.then:
8      %sub = sub nsw i32 %add , 50
9      %mul = mul nsw i32 96 , %add
10     br label %if.end
11
12 if.else:
13     %add2 = add nsw i32 %add , 50
14     %mul3 = mul nsw i32 96 , %add
15     br label %if.end
16
17 if.end:
18     %f.0 = phi i32 [ %sub , %if.then ] , [ %add2 , %if.else ]
19     %sub4 = sub nsw i32 50 , 96
20     %add5 = add nsw i32 %sub4 , %f.0
21     ret i32 0
```

Listing 5: Example simplified input bitcode.

4.4 Available Expressions: Example Output

	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: Result of the available expressions pass on the example input from Listing 4.