15-745, Spring 2013
Homework Assignment 2

Assigned: Thursday, January 31
Due: Thursday, February 14, 9:00AM

Introduction

In class, we have seen several examples of dataflow analyses. These analyses, like liveness, reaching definitions, and available expressions, compute different properties of a program and do so by traversing the control-flow graph in different ways. However, as we have discussed, they may all be brought together under a common framework for performing iterative dataflow (as in Slide 23 from Lecture Notes 4).

In this assignment, you will implement an iterative dataflow analysis framework that works in LLVM. To demonstrate the functionality of your framework, you will use it to run the liveness and reaching definitions analyses.

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

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

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

`/afs/cs.cmu.edu/academic/class/15745-s13/public`

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

1 Iterative Dataflow Analysis Framework

In this section, you will build an abstraction for performing iterative dataflow analysis. Your abstraction should solve any unidirectional dataflow analysis problem given the following parameters:

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 should be represented as bit vectors so that the semi-lattice and set operations (union, intersection) are easy to implement. Careful thought should be given to how the analysis parameters are represented. Do not perform redundant computation when possible: use a worklist and somehow offer the capacity to summarize the operation of the transfer function over basic blocks (so your framework doesn't *always* step through every instruction in an active block while trying to reach convergence). **We may ask you to reuse this framework in future assignments, so it is important that you do not overspecialize your implementation to the analyses required for this homework.**

In your writeup, please describe the design of your framework. Do you make any additional assumptions about the structure of a dataflow problem or of the C++ types involved in its implementation? How did you apply it to either of the two analyses in the next section?

2 Dataflow Analyses

You will now use your iterative dataflow analysis framework to implement *liveness* and *reaching definitions*. Each analysis should perform computation at program points. Program points are assumed to lie *between* instructions (not in the middle of instructions).

Liveness On convergence, your *liveness* pass should report all variables that are "live" at each program point to standard error. Please call this pass "`cd-liveness`".

For the purpose of this assignment, a variable is an `alloca` instruction that:

- is defined in the function's entry block
- is only used by non-volatile `LoadInst`s
- is only used by non-volatile `StoreInst`s as the pointer operand
- is not used by any other instructions than these

In terms of the *liveness* analysis, the only use sites are load instructions and the only def sites are store instructions. This kind of analysis is useful when trying to minimize the size of a stack frame: if two stack variables are not simultaneously live, they may be aliased to the same stack slot.

When printing out your live sets at program points, please use the `print` method provided by the `Value` class and separate set elements with semicolons. (It's fine to have trailing commas.)

Here is an acceptable output format:

```
entry:

Live: {}
  %i = alloca i32, align 4
Live: {}
  %x = alloca i32, align 4
Live: {}
  store i32 1, i32* %i, align 4
Live: {  %i = alloca i32, align 4;}
  %0 = load i32* %i, align 4
Live: {}
  store i32 %0, i32* %x, align 4
Live: {  %x = alloca i32, align 4;}
  %1 = load i32* %x, align 4
Live: {}
  ret i32 %1
Live: {}
```

Reaching Definitions On convergence, your *reaching definitions* pass should report all the definition sites that "reach" each program point to standard error. Please call this pass "`cd-reaching-definitions`".

For this assignment, a *definition* of a variable is a `StoreInst` with that variable as its pointer operand.

Please use the same format as you did for *liveness*, but prefix your sets with "`Reach:`" instead of "`Live:`":

```
entry:

Reach: {}
  %i = alloca i32, align 4
Reach: {}
  %x = alloca i32, align 4
Reach: {}
  store i32 1, i32* %i, align 4
Reach: {  store i32 1, i32* %i, align 4;}
  %0 = load i32* %i, align 4
Reach: {  store i32 1, i32* %i, align 4;}
  store i32 %0, i32* %x, align 4
Reach: {  store i32 1, i32* %i, align 4;  store i32 %0, i32* %x, align 4;}
  store i32 2, i32* %i, align 4
Reach: {  store i32 2, i32* %i, align 4;  store i32 %0, i32* %x, align 4;}
  %1 = load i32* %x, align 4
Reach: {  store i32 2, i32* %i, align 4;  store i32 %0, i32* %x, align 4;}
  ret i32 %1
Reach: {  store i32 2, i32* %i, align 4;  store i32 %0, i32* %x, align 4;}
```

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 `w=x+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.

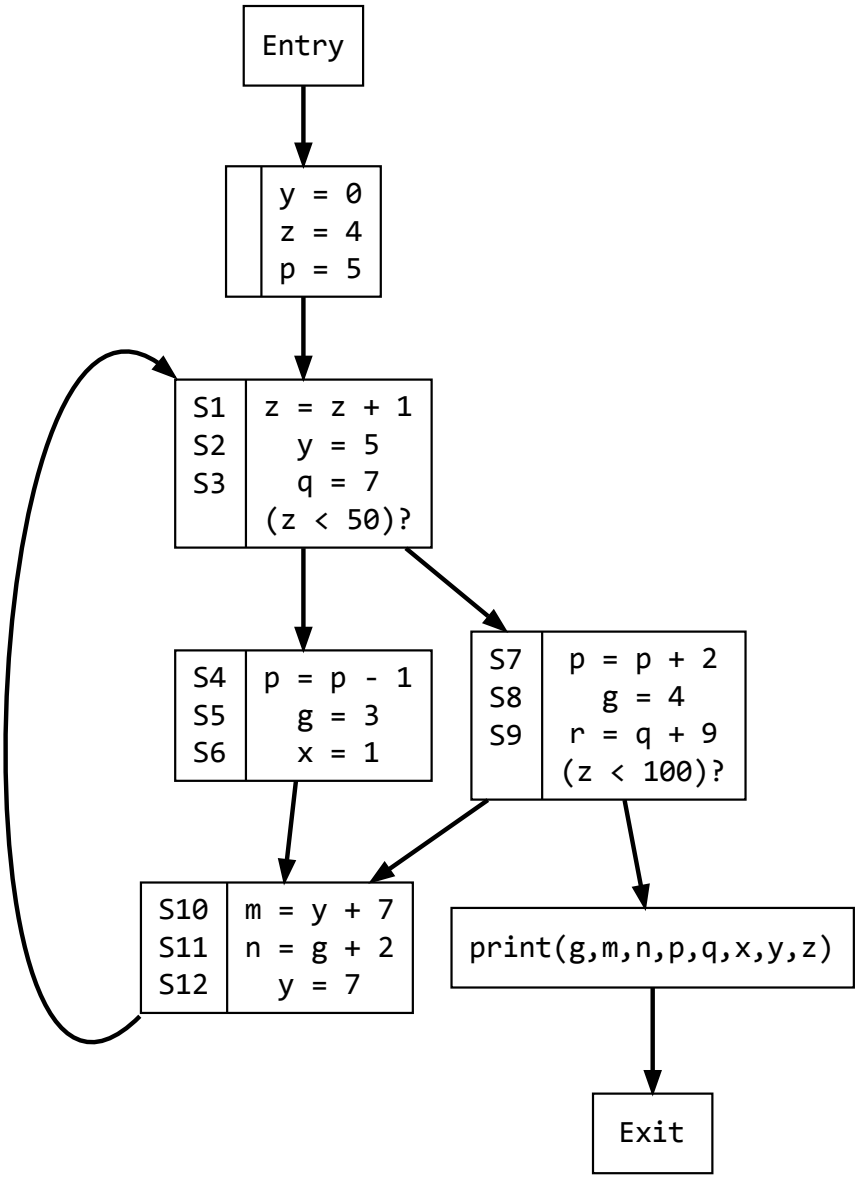
All CFGs may be drawn either via a computer or by hand. (If you draw the CFG by hand, you will still need to legibly include it in your writeup PDF.)

```
 1 w=x+3;
 2 c=y>5;
 3 if (c) {
 4    z=z+x;
 5    w=4;
 6 } else {
 7    y=y-5;
 8    z=z+x;
 9 }
10 x=x+3;
11 return z;
```

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

Hard-copy submission:

1. A report that briefly describes the design and implementation of your framework and passes, and how you tested them. 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. A listing of your source code.
3. Your answers to the questions in part 3.

Electronic submission:

- A PDF of your writeup report and answers to the questions, named `writeup.pdf`.
- The source code for your framework and passes, the associated `Makefile`, your test cases, and a `README` describing how to build and run them (especially if you for some reason diverge significantly from the boilerplate code).

  Place all of these files in a directory with the same name as the Andrew ID of one of your group members. Archive this directory and name the file with the same Andrew ID (`bovik.tar.gz`):

  ```
  tar czvf bovik.tar.gz bovik
  ```

  When the file is extracted with `tar xf`, we expect to see these required files in these locations:

  ```
  ./bovik/README
  ./bovik/ClassicalDataflow/liveness.cpp
  ./bovik/ClassicalDataflow/reaching-definitions.cpp
  ./bovik/ClassicalDataflow/dataflow.cpp
  ./bovik/ClassicalDataflow/dataflow.h
  ./bovik/ClassicalDataflow/Makefile
  ./bovik/writeup.pdf
  ```

  It is fine if there are other files included; please also include any additional tests you used for verification.

  If you need to update your submission but are not able to overwrite the old file, you may append a version number (separated by a dash) to the Andrew ID in the filename: `bovik-2.tar.gz`

  Copy the tar.gz file into the directory

  ```
  /afs/cs.cmu.edu/academic/class/15745-s13/public/asst/asst2/handin
  ```

  Include as comments near the beginning of your source files the identities of all members of your group. Please don't forget to comment your code.