Assignment 2

ECE/CS 5544 Spring 2023

Assigned: Tuesday, Feb 14

Part 1 Due: Wednesday, March 1, 11:59PM

Part 2 Due: TBA

Late Submission Penalty Policy: 1-day - 5% and 2-days - 10%

Abstract

In class, we discussed many interesting dataflow analyses such as Live Variables (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 for solving bit-vector dataflow problems, and use it to implement a forward dataflow analysis (Available Expressions) and a backward dataflow analysis (Liveness).

1 Introduction

1.1 Policy

You will work in groups of two people to solve the problems for this assignment. You can create your group in the people section of the canvas.

1.2 Submission

Both members should upload the solutions. Please include the following items in an archive labeled with your pid(partner1)_pid(partner2) (e.g., pid_pid.tar.gz), and submit the resulting file to Canvas. Ensure that when this archive is extracted, the files appear as follows:

```
./pid/README
./pid/Dataflow/available.cpp
./pid/Dataflow/available-support.cpp
./pid/Dataflow/available-support.h
./pid/Dataflow/dataflow.cpp
./pid/Dataflow/dataflow.h
./pid/Dataflow/liveness.cpp
./pid/Dataflow/Makefile
./pid/Dataflow/reaching.cpp
./pid/writeup.pdf
./pid/tests/
```

- A report that briefly describes the implementation of your dataflow framework and passes, along with the solutions to the non-programming questions, named `writeup.pdf`, containing both the group members' pid.
- Well-commented source code for your passes and associated Makefiles.

- A `README` file describing how to build and run your passes.
- Any tests used for verification of your code.

2 Part 1: 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:

- Domain, including the semi-lattice
- Direction (forwards or backwards)
- Transfer function
- Meet Operation
- Boundary condition (entry or exit)
- Initial interior points (in or out)

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 efficient and 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, the direction could reasonably be represented as a boolean, while function pointers may seem more appropriate for representing transfer functions.

2.2 Analysis Passes

You will now use your iterative dataflow framework to implement Available Expressions, Reaching Definitions, and Liveness. As explained below in more detail, each analysis should perform computation at the entry point and the exit point of a basic block. You may assume that the input received by your pass has already been transformed by the `mem2reg` pass.

2.2.1 Available Expression

Upon convergence, your Available Expressions pass should report all the binary expressions that are `available` at entry and exit points of basic blocks. 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.

We will consider two expressions as equal if the instructions that calculate these expressions 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.

Your program you should print out the following information for each basic block (BB):

1. `BB Name`
2. `gen [BB]`
3. `kill [BB]`
4. `IN [BB]`
5. `OUT [BB]`

2.2.2 Reaching Definitions

Upon convergence, your reaching definitions pass should report all definitions reaching the entry and exit points of basic blocks. Please call your pass `reaching`.

Your program you should print out the following information for each basic block (BB):

1. `BB Name`
2. `gen [BB]`
3. `kill [BB]`
4. `IN [BB]`
5. `OUT [BB]`

2.2.3 Liveness

Upon convergence, your liveness pass should report all variables that are "live" at the entry and exit points of basic blocks. Please call your pass `liveness`.

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.

Your program should print out the following information for each basic block (BB):

1. `BB Name`
2. `def [BB]`
3. `use [BB]`
4. `IN [BB]`
5. `OUT [BB]`

2.3 Skeleton Code

We are providing some code for convenience and to get you started. You can find this source code in Canvas.

3 Part 2: Non-programming Homework Questions

3.1 Loop Invariant Code Motion (Purple Dragon Book 9.5.1)

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

1. List the loop invariant instructions.
2. Indicate if each loop invariant instruction can be moved to the loop preheader, and give a brief justification.

[Note]: There will be no definitions reaching before the Entry, every definitions are contained in the code.

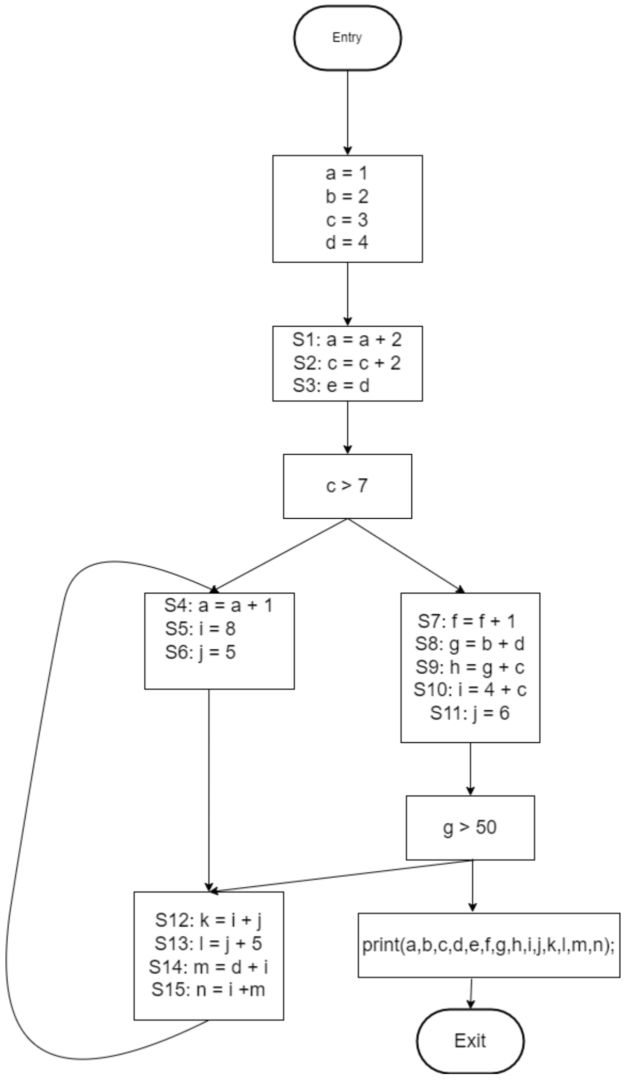

Figure 1: Labeled control-flow graph for question 3.1

3.2 Lazy Code Motion (Purple Dragon Book 9.5.5)

Suppose that you are optimizing the code shown in Table 1.

Table 1: Code listing for question 3.2

```
int funfoo ( a , b , c ) {
 if( c > 4 ) {
L1:
  g = a - b;
  if ( g > 8 ) {
   b = b + 1;
   goto L1;
  }
 }else {
   while ( a < 4 ) {
    a = a + 1;
    d = a - b;
   }
 }
 f = a - b;
 return f;
}
```

1. 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 (Algorithm 9.36), determine the *anticipated expressions* for each basic block.
2. Now, determine the *available expressions* for each basic block, and indicate the *earliest* basic block for each expression, if applicable.
3. Determine the *postponable expressions* and *used* expressions for each basic block, and indicate the *latest* basic block for each expression, if applicable.
4. Complete the final pass of lazy code motion by inserting and replacing expressions. Build the finished control-flow graph, and label each basic block with its instruction(s).

4 Reference

This assignment is inspired by CMU 15-745 course's assignment and is reused here with permission. The original assignment can be found here.