

# Assignment 2 Dataflow Analysis

Due Date: **Mar. 3rd**, Total Marks: 100 pts

CSCD70 Compiler Optimizations

Department of Computer Science

University of Toronto

## Abstract

In class, we discussed interesting dataflow analyses such as reaching definitions, liveness, 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 (i.e., forward, backward), they share some common properties such as iterative algorithms, transfer functions, and meet operators. These commonalities make it worthwhile to develop a generic framework that can be parameterized appropriately for solving a specific dataflow analysis. In this assignment, you will implement such an iterative dataflow analysis framework in LLVM, and use it to implement forward and backward dataflow analyses. The objective of this assignment is to create a generic framework for solving iterative dataflow analysis problems.

## 1 Policy

### 1.1 Collaboration

You will work in groups of **two** for the assignments in this course. Please turn in a single submission per group.

### 1.2 Submission


Please submit all your files to GitHub Classroom. Make sure that your submission includes the followings:

writeup.pdf

```
include/Domain/Base.h
include/Domain/Expression.h
include/Domain/Variable.h
include/DFA/Flow/Framework.h
include/DFA/Flow/ForwardAnalysis.h
include/DFA/Flow/BackwardAnalysis.h
include/DFA/MeetOp.h
include/Utility.h
```

```
lib/1-AvailExprs.cpp
lib/2-Liveness.cpp
lib/3-SCCP.cpp
lib/4-LCM/1-AnticipatedExprs.cpp
lib/4-LCM/2-WBAvailExprs.cpp
lib/4-LCM/3-EarliestPlacement.cpp
lib/4-LCM/4-PostponeableExprs.cpp
lib/4-LCM/5-LatestPlacement.cpp
lib/4-LCM/6-UsedExprs.cpp
lib/4-LCM/LCM.h
lib/DFA/Domain/Expression.cpp
lib/DFA/Domain/Variable.cpp
lib/DFA.h
lib/DFA.cpp
lib/CMakeLists.txt
```

```
CMakeLists.txt
test/*
```

- A report named `writeup.pdf` that has answers to the **programming** (highlighted in  using Section 2) and theoretical questions, and **optionally** describes the implementation details of your passes.
- Well-commented source code for your framework and passes (Available Expressions, Liveness, SCCP, and LCM), together with a build file `CMakeLists.txt` (please write your `CMakeLists.txt` in such a way that all passes can be built, integrated, and tested using the command

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
make test
```

- Two subfolders named `test` that include all the microbenchmarks used for the verification of your code.

## 2 Problem Statement

### 2.1 Iterative Framework

A well-written iterative dataflow analysis framework significantly reduces the burden of implementing new dataflow passes. The developer only writes details such as the meet operator, the transfer function, the analysis direction, etc. Specifically, the framework should solve any unidirectional dataflow analysis as long the analysis supplies the following:

1. Domain
2. Direction (Forward/Backward)
3. Meet Operation
4. Transfer Function

### 2.2 Dataflow Analysis [50 pts]

**2.2.1 Available Expressions [10 pts].** Upon convergence, your *Available Expressions* pass should report all the binary expressions that are *available* at each program point. For this assignment, we are only concerned with expressions represented using an instance of `BinaryOperator`. Analyzing comparison instructions and unary instructions such as negation is not required.

We will consider two expressions *equal* if the instructions that calculate these expressions share the same opcode, left-hand-side and right-hand-side operand. In addition to this, the expression  $x \oplus y$  is equal to expression  $y \oplus x$  under the condition that the operator  $\oplus$  is *commutative*.

**2.2.2 Liveness [20 pts].** Upon convergence, your *Liveness* pass should report all variables that are *live* at each program point. For this assignment, you only need to 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:

```

for (auto Iter = Inst->op_begin();
     Iter != Inst->op_end(); ++Iter) {
    Value *V = *Iter;

    if (isa<Instruction>(V) || isa<Argument>(V)) {
        ...
    }
}

```

The fact that there are  $\phi$  instructions has ramifications on how your passes are implemented. Think carefully about what this implies and **explain**  $\not\Leftarrow$  this in your writeup.

**2.2.3 Sparse Conditional Constant Propagation (SCCP) [20 pts].** Upon convergence, your *SCCP* pass should report all variables that are *constant* at each program point. To simplify the implementation, we will only track the *instruction-defined values* and *function arguments*. Furthermore, you will only have to handle instructions in the provided test case *SCCP.ll*.

When developing the *SCCP* pass, you will notice that it is using a different value type from the previous two passes. **Describe**  $\not\Leftarrow$  your value type and transfer function design in the writeup.

### 2.3 Lazy Code Motion [40 pts]

Now that you have completed two examples on the forward analysis and one example on the backward analysis, consider the Lazy Code Motion (LCM) problem that we described in class, which consists of two forward and two backward dataflow analyses. Your job in this subsection is to leverage the framework to determine the *latest placement* in LCM. This requires the four uni-directional analyses, all of which you will have to inherit from the framework.

The implementation of LCM is somewhat similar to that of the passes in the previous subsections: Similar to the case of Available Expressions, the domain of analysis in LCM is restricted to expressions represented using an instance of `BinaryOperator`; and similar to the case of Liveness, the presence of  $\phi$  instructions has ramifications on your passes.

The provided benchmark file *LCM.ll* is directly copied from the problem statement in Question 3.1 (therefore, it is recommended that you complete Question 3.1 first before approaching this problem). Check your program outputs with your answer in Question 3.1 after the completion of both parts. Do they corroborate each other? **Show**  $\not\Leftarrow$  your observation in the writeup.

## 3 Theoretical Questions

### 3.1 Lazy Code Motion [10 pts]

Consider the code in Listing 1.

(2) Build the CFG for this code and break the critical edges. Using the algorithm described in class, provide *anticipated expressions* for each basic block.

(3) Provide *will-be-available expressions* for each basic block, and indicate the *earliest placement* for each expression, if applicable.

(4) Provide *postponable expressions* and *used expressions* for each basic block, and indicate the *latest placement* for each expression, if applicable.

(5) Complete the final pass of lazy code motion by inserting and replacing expressions. Provide the finalized CFG, and label each basic block with its instruction(s). Answer why it should have better performance compared with the raw CFG.

```

int foo(a, b, c) {
    if (a > 5) {
        g = b + c;
        print(g);
    } else {
        while (b < 5) {
            b = b + 1;
            d = b + c;
            print(d);
        }
    }
    e = b + c;
    return e;
}

```

Listing 1. Source Code for Analysis

## 4 FAQ

Given below is the questions asked during previous offerings of the class. If you do not think they fully answer your question, please open a new thread on Piazza.

**Q1** [Section 2.2] Q: In class, we always dealt with the DFA by basic blocks (i.e., the transfer function operates on an entire basic block). But it seems from the starter code that we are invoking the transfer function on each instruction. How does this work?

**A.** Remember that we mentioned in class that the transfer function of the entire basic blocks is just the composite of the transfer function of all the instructions within that basic block, i.e.,

$$f_{bb} = f_{i_n} \circ f_{i_{n-1}} \circ \dots \circ f_{i_1}^1, i_{1..n-1,n} \in bb$$

You can prove that the above equation is mathematically correct, but hopefully, it makes sense by intuition.

**Q2** [Section 2.2] How does the domain work? What is the relationship between the domain and the instruction-domain mapping? Specifically, suppose that we have  $N$  expressions in our program text, then for available expressions we just have a bitvector of length  $N$ ? If that is the case, then how do we know which bit in the bit vector is associated with a particular expression?

**A.** Your understanding is right. Suppose that you have  $N$  domain elements, then your bitvector length should be  $N$  as well. The reason is that there shall be a one-to-one correspondence between the domain elements and the bitvector indices.

As an example, let us suppose that we have three expressions in our program text, namely  $\{a + b, a - b, a \times b\}$ . When we say that the output of instruction  $i$  is  $\{001\}$ , that '1' can be  $a + b$ ,  $a - b$ , or  $a \times b$ . What really matters is that *the mapping between bitvector indices and domain elements must be consistent throughout the entire analysis*, and that is the reason why we have domain, which plays the role of the metadata that describes the bitvectors.

<sup>1</sup>The order of composition depends on the direction of DFA. We use forward analysis as an example here.