Assignment 2: Iterative Dataflow Analysis Framework

Due Date: Mar. 8th, Total Marks: 100 pts

CSCD70H3 S (Winter) Compiler Optimization
Department of Computer Science, UTSC

ABSTRACT

In class, we discussed many interesting data flow 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 (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 will implement such an iterative data flow analysis framework in LLVM, and use it to implement a forward data flow analysis (Reaching Definitions) and a backward data flow analysis (Liveness). Although Liveness and Reaching Definitions implementations are available in some form in LLVM, they are not of the iterative flavor, and 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 Reaching Definitions analysis.

1 POLICY

1.1 Collaboration

You will work in groups of two people to solve the problems for this assignment (you can also choose to work individually). **Please turn in a single writeup per group, indicating the names and UTORid of all group members.**

1.2 Submission

Please include the following items in an archive labeled with the UTORid of all group members (`a2-utorid1(-utorid2).tar.gz`), and submit the resulting file via *Markus* (`https://markus.utsc.utoronto.ca/cscd70w18/`). Please make sure that when this archive is extracted, the files appear as follows:

```
./a2-utorid1(-utorid2)/writeup.pdf

./a2-utorid1(-utorid2)/Dataflow/inc/Framework.hpp
./a2-utorid1(-utorid2)/Dataflow/src/AvailableExpr.cpp
./a2-utorid1(-utorid2)/Dataflow/src/Liveness.cpp

./a2-utorid1(-utorid2)/Dataflow/Makefile
./a2-utorid1(-utorid2)/Dataflow/tests/
```

[1]

- A report named **writeup.pdf** that briefly describes the implementation of the framework and both passes, and has answers to the theoretical questions in this assignment.
- Well-commented source code for your **iterative framework** and passes (**AvailableExpr** and **Liveness**), and associated **`Makefile`** (please write your `Makefile` in such a way that all passes can be built, integrated (and run) using the command `make all`).
- A subfolder named **`tests`** that contains all the microbenchmarks that were used for verification of your code.

[1] You are free to organize your files (e.g. you can have two source files `AvailableExpr.cpp` and `AvailableExprSupport.cpp` for the Available Expressions problem), but please write your makefile accordingly.

2 PROBLEM STATEMENT

2.1 Iterative Framework [40 pts]

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 the analysis supplies the following**:

(1) Domain (including the Semi-Lattice)
(2) Direction (Forwards/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. For example, direction could reasonably be represented as a boolean, while function pointers may seem more appropriate for representing transfer functions.

Please note that it will be worth your time putting more thoughts and efforts on this part because you will be using this framework in the upcoming Assignment 3.

Hint: If you are not sure where to get started, please first proceed to the next section and start working on the Liveness and Available Expressions problem, because you can have a better idea of what those dataflow-analysis problems share in common.

2.2 Dataflow Analysis [40 pts]

2.2.1 Liveness [20 pts]. Upon convergence, your Liveness pass should report all variables that are **live** at each program point. 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:

```
Instruction * inst = ...

for (auto iter  = inst->op_begin();
          iter != inst->op_end(); ++iter)
{
    Value * val = *iter;

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

You should carefully consider how your analysis passes are affected by the 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.

The fact that you are working on code in SSA form will have ramifications on how your passes are implemented. Think carefully about what this means to your implementation and briefly explain this in your assignment report.

2.2.2 Available Expressions [20 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 by 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 op y` is equal to expression `y op x` under the condition that the operator `op` is commutative.

3 THEORY QUESTIONS

3.1 Loop Invariant Code Motion [10 pts]

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.

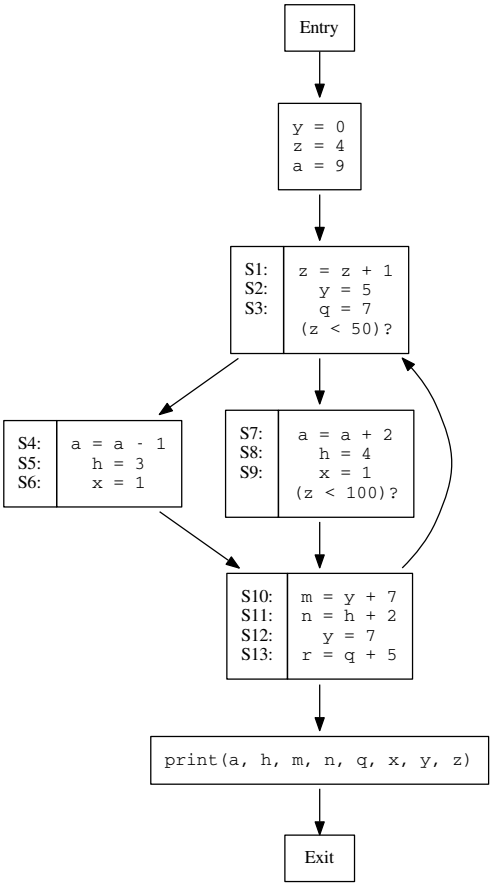

Figure 1: CFG for Analysis

3.2 Lazy Code Motion [10 pts]

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

(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, provide **anticipated expressions** for each basic block.
(2) Now, provide **available expressions** for each basic block, and indicate the **earliest basic block** for each expression, if applicable.
(3) Next, provide **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. Provide the finished control-flow graph, and label each basic block with its instruction(s).

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

    return e;
}
```

Listing 1: Source Code for Analysis

APPENDICES

Appendix A Framework Instantiation

For your reference, given below is how those two problems should fit into the general framework that you have developed.

	Liveness	**Available Expressions**
Domain	Sets of Variables	Sets of Expressions
Directions	Backwards $\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S,\text{succ}(B)}\text{IN}[S]$	Forwards $\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P,\text{pred}(B)}\text{OUT}[P]$
Transfer Function	$\text{use}_B \cup (x - \text{def}_B)$	$\text{gen}_B \cup (x - \text{kill}_B)$
Boundary	$\text{IN}[\text{exit}] = \emptyset$	$\text{OUT}[\text{entry}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cap$
Initial	$\text{in}[B] = \emptyset$	$\text{OUT}[B] = \mathbb{U}$

Appendix B Example on Liveness

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

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

Listing 2: Example Source Code for Analysis

```
define i32 @sum(i32, i32) #0 {
  br label %3

; <label>:3:                                      ; preds = %7, %2
  %.01 = phi i32 [ 1, %2 ], [ %6, %7 ]
  %.0 = phi i32 [ %0, %2 ], [ %8, %7 ]
  %4 = icmp slt i32 %.0, %1
  br i1 %4, label %5, label %9

; <label>:5:                                      ; preds = %3
  %6 = mul nsw i32 %.01, %.0
  br label %7

; <label>:7:                                      ; preds = %5
  %8 = add nsw i32 %.0, 1
  br label %3

; <label>:9:                                      ; preds = %3
  ret i32 %.01
}
```

Listing 3: LLVM Bytecode after `llvm-dis`

Table 1: Expected Output according to the Bytecode above

	%0, %1
`br label %3`	
	-
`%.01 = phi i32 [ 1, %2 ], [ %6, %7 ]`	
	-
`%.0 = phi i32 [ %0, %2 ], [ %8, %7 ]`	
	%.01, %0, %.0, %1
`%4 = icmp slt i32 %.0, %1`	
	%.01, %0, %.0, %1, %4
`br i1 %4, label %5, label %9`	
	%.01, %0, %.0, %1
`%6 = mul nsw i32 %.01, %.0`	
	%0, %6, %.0, %1
`br label %7`	
	%0, %6, %.0, %1
`%8 = add nsw i32 %.0, 1`	
	%0, %6, %8, %1
`br label %3`	
	%.01
`ret i32 %.01`	

Appendix C Example on Available Expression

```
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;
        e = c * b;
    }
    else
    {
        f = b + a;
        e = c * b;
    }
    b = a - c;
    d = b + f;

    return 0;
}
```

Listing 4: Example Source Code for Analysis

```
define i32 @main(i32, i8**) #0 {
  %3 = add nsw i32 %0, 50
  %4 = add nsw i32 %3, 96
  %5 = icmp slt i32 50, %3
  br i1 %5, label %6, label %9

; <label>:6:                                      ; preds = %2
  %7 = sub nsw i32 %3, 50
  %8 = mul nsw i32 96, %3
  br label %12

; <label>:9:                                      ; preds = %2
  %10 = add nsw i32 %3, 50
  %11 = mul nsw i32 96, %3
  br label %12

; <label>:12:                                     ; preds = %9, %6
  %.0 = phi i32 [ %7, %6 ], [ %10, %9 ]
  %13 = sub nsw i32 50, 96
  %14 = add nsw i32 %13, %.0
  ret i32 0
}
```

Listing 5: LLVM Bytecode after `llvm-dis`

Table 2: Expected Output according to the Bytecode above

	∅
`%3 = add nsw i32 %0, 50`	
	[add %0, 50]
`%4 = add nsw i32 %3, 96`	
	[add %0, 50], [add %3, 96]
`%5 = icmp slt i32 50, %3`	
	[add %0, 50], [add %3, 96]
`br i1 %5, label %6, label %9`	
	[add %0, 50], [add %3, 96]
`%7 = sub nsw i32 %3, 50`	
	[add %0, 50], [add %3, 96], [sub %3, 50]
`%8 = mul nsw i32 96, %3`	
	[add %0, 50], [add %3, 96], [sub %3, 50], [mul 96, %3]
`br label %12`	
	[add %0, 50], [add %3, 96], [sub %3, 50], [mul 96, %3]
`%10 = add nsw i32 %3, 50`	
	[add %0, 50], [add %3, 96], [add %3, 50]
`%11 = mul nsw i32 96, %3`	
	[add %0, 50], [add %3, 96], [add %3, 50], [mul 96, %3]
`br label %12`	
	[add %0, 50], [add %3, 96], [add %3, 50], [mul 96, %3]
`%.0 = phi i32 [ %7, %6 ], [ %10, %9 ]`	
	[add %0, 50], [add %3, 96], [mul 96, %3]
`%13 = sub nsw i32 50, 96`	
	[add %0, 50], [add %3, 96], [mul 96, %3], [sub 50, 96]
`%14 = add nsw i32 %13, %.0`	
	[add %0, 50], [add %3, 96], [mul 96, %3] [sub 50, 96], [add %13, %.0]
`ret i32 0`	
	[add %0, 50], [add %3, 96], [mul 96, %3] [sub 50, 96], [add %13, %.0]