Assignment 1: Introduction to LLVM

Due Date: **Jan. 31st (Thursday)**, Total Marks: 100 pts

CSCD70 Compiler Optimization
Department of Computer Science
University of Toronto

ABSTRACT

Welcome to CSCD70 Compiler Optimization. We will be using the Low Level Virtual Machine (LLVM) Compiler infrastructure from University of Illinois Urbana-Champaign (UIUC) for our programming assignments. While LLVM is currently supported on a number of hardware platforms, we expect the assignments to be completed on the undergraduate workstations, since that is where they will be graded. The objective of this first assignment is to introduce you to LLVM and some ways that it can be used to make your programs run faster. In particular, you will be using LLVM to analyze code to output interesting properties about your program (Section 3.1) and to perform local optimizations (Section 3.2).

1 POLICY

1.1 Collaboration

You will work in groups of *two* for the assignments in this course. Please turn in a single writeup per group, indicating the names and UTORid of both group members.

1.2 Submission

Please include all your files in an archive labeled with the UTORid of both group members, and email the resulting file to bojian@cs.toronto.edu. Make sure that when this archive is extracted, the files appear as follows:

```
./a1-utorid1-utorid2/writeup.pdf

./a1-utorid1-utorid2/FunctionInfo/FunctionInfo.cpp
./a1-utorid1-utorid2/FunctionInfo/Makefile
./a1-utorid1-utorid2/FunctionInfo/tests/...

./a1-utorid1-utorid2/LocalOpts/LocalOpts.cpp
./a1-utorid1-utorid2/LocalOpts/Makefile
./a1-utorid1-utorid2/LocalOpts/tests/...
```

[1]

• A report named *writeup.pdf* that briefly describes the implementation of both passes, and has answers to the theoretical questions in this assignment.

• Well-commented source code for your passes (*FunctionInfo* and *LocalOpts*), 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`).

• Two subfolders named *tests* that contain all the microbenchmarks used for verification of your code.

[1] You do not have to implement all your local optimization passes in one file.

2 EXAMPLE: CREATING A PASS

The source file `FunctionInfo/FunctionInfo.cpp` that is provided with this assignment contains a dummy LLVM pass for analyzing the functions in a program. Currently it only prints out:

```
CSCD70 Functions Information Pass
```

In the next section, you will extend this file to print out more interesting information. For now, we will use this pass to demonstrate how to build and run LLVM passes on programs.

• Using the provided `Makefile` `Build.mk`, make sure that you can build this pass with the command:

```
make -f Build.mk all
```

• Compile the source code `tests/loop.c` to an optimized LLVM bytecode object (`loop.bc`) as follows:

```
clang-6.0 -O2 -emit-llvm -c tests/loop.c
    -o tests/loop.bc
```

(`clang` is the LLVM's frontend for the C language family), and inspect the `loop.bc` generated bytecode using `llvm-dis` with the command:

```
llvm-dis-6.0 ./tests/loop.bc -o=./tests/loop.ll
```

This will create a disassembly listing in `loop.ll` of the `loop.bc` bytecode.

• Now try running the dummy `FunctionInfo` pass on the bytecode. To do this, use the `opt` command as follows:

```
opt-6.0 -load FunctionInfo/FunctionInfo.so
    -function-info loop.bc -o loop-opt.bc
```

Note the use of flag `-function-info` to enable this pass (see if you can locate the declaration of this flag).

• If all goes well, `CSCD70 Function Information Pass` should be printed to `stdout`.

• We also have another makefile `Optimize.mk` that automatically goes through all the above process. Upon entering

```
make -f Optimize.mk all
```

You should be able to see the same output from the command line. For the programming assignments throughout this

course, we strongly recommend that you use `Optimize.mk` as your primary `Makefile`.

3 PROBLEM STATEMENT

3.1 Function Information [40 pts]

Your job now is to extend the dummy `FunctionInfo` pass from the previous section to learn interesting properties about the functions in a program. Your pass should report the following information about all functions that appear in a program:

(1) Name
(2) Number of Arguments (`*` if applicable)
(3) Number of Direct Call Sites in the same LLVM module (i.e. locations where this function is *explicitly* called, ignoring function pointers).
(4) Number of Basic Blocks
(5) Number of Instructions

The expected output of running `FunctionInfo` on the optimized bytecode is shown in Table 1. Note that although the source code for `loop.c` has a call to `g_incr` in `loop`, this call is optimized away in the LLVM bytecode. *When reporting the number of calls, please count the number that appear in the bytecode, even if it does NOT match the number of calls in the original source code.*

It is recommended that you debug your pass with more complex source files, as you can imagine your grade on this assignment is directly related to the quality of your test cases.

Table 1: Expected `FunctionInfo` Output for `loop.c`

Name	# Args	# Calls	# Blocks	# Insts
`g_incr`	1	0	1	4
`loop`	3	0	3	10

3.2 Local Optimizations [40 pts]

Now that you are familiar with LLVM passes, it is time to write a pass for making programs faster. You will implement optimizations that have been covered in class. While there are many types of optimizations, we will keep things simple in this section and focus only on the algebraic optimizations, the scope of which is a single basic block. Specifically, you will implement the following local optimizations:

(1) Algebraic Identity

$$x+0=0+x \Rightarrow x$$

(2) Strength Reductions

$$2\times x = x\times 2 \Rightarrow x+x \text{ or } x \ll 1$$

(3) Multi-Inst Optimization

$$a=b+1, c=a-1 \Rightarrow a=b+1, c=b$$

This is a somewhat open-ended question. *Please handle at least the above cases, as well as one more in each category that you come up with, for integer targets.*

You should create a new LLVM pass (or multiple passes) following the steps in Section 3.1. Because this will be a transformation pass rather than an analysis pass, there will be some small differences from the setup of the `FunctionInfo` pass. Please provide an appropriate `Makefile` at `LocalOpts` and write it in such a way that all the pass(es) can be built and run with the command `make all`).

To better test your pass(es), you should build *unoptimized* LLVM bytecode from the test cases with the commands:

```
clang-6.0 -O0 -Xclang -disable-O0-optnone
    -emit-llvm -c mytest.c
opt-6.0 -mem2reg mytest.bc -o mytest-m2r.bc
```

[2]

(you may assume that all input to your pass will first go through the `mem2reg` pass as shown above).

In addition to transforming the bytecode, your pass should also print to standard output a summary of the optimizations it performed. There is no canonical format for this output, but you should at least try to categorize and count the transformations your pass applies, e.g.,

```
Transformations applied:
    Algebraic Identity: 2
    Strength Reduction: 3
    Multi-Inst Optimization: 1
```

4 THEORETICAL QUESTIONS

4.1 Control Flow Graph (CFG) [5 pts]

Consider the following code and answer the questions below:

(1) Identify the leader instruction and their corresponding basic blocks. Draw the CFG.

(2) Identify the back-edge(s) in the CFG drawn in Question (1). Write them down using the form $T \rightarrow H$, where T is the basic block at the tail of the edge and H is the basic block at the head of the edge.

```
S1: x = y + z
S2: if (y < 100) goto S5
S3: x = x + 1
S4: z = z + 1
S5: if (x < 100) goto S3
S6: y = y + 1
S7: if (y <  50) goto S1
S8: print (x, y, z)
```

[2] If you do not add the `-Xclang -disable-O0-optnone` option, then further optimizations such as `mem2reg` will be disabled. The `mem2reg` optimization pass promotes the variables from memory to registers. You can try to ignore the second command and check how the bytecode looks like.

```
S9: return
```

4.2 Natural Loops [5 pts]

Find and describe the natural loop(s) in the following code. For full marks, be sure to show (1) basic blocks (2) CFG (3) dominator tree (4) back-edges (head and tail) (5) basic blocks that comprise the natural loop for each back-edge. Be sure to give your basic blocks clear labels that match those in the original code:

```
    x, y = ...
    goto L4
L1: y = x * x
    if (x < 50) goto L2
    y = x + y
    goto L3
L2: y = x - y
    x = x + 1
L3: print y
    if (y < 10) goto L1
    if (x <= 0) goto L5
L4: x = x / 2
    goto L1
L5: return y
```

4.3 Available Expressions [10 pts]

An expression $x \oplus y$ is *available* at a point p if every path from the entry node to p evaluates $x \oplus y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y. For the *available expressions* dataflow analysis we say that a block *kills* expression $x \oplus y$ if it assigns (or may assign) x or y and does not subsequently recompute $x \oplus y$. A block *generates* expression $x \oplus y$ if it definitely evaluates $x \oplus y$ and does not subsequently define x or y.

(1) Table 2 shows the definitions of available expressions dataflow analysis, with the **Meet Operator** entry left unspecified deliberately. Please answer what it should be and explain. (*Hint: Your explanation should be based on the definitions we have provided.*)

(2) Perform available expressions analysis on the code in Figure 1. For each basic block, list the *final* GEN, KILL, IN and OUT sets. Your answer should be what the sets are *upon convergence*, and in the format shown in Table 3.

5 FAQ

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

Table 2: Available Expressions Dataflow Analysis

Domain	Sets of Expressions
Direction	Forward
Transfer Function	$f_B := \text{gen}_B \cup (x - \text{kill}_B)$
Meet Operator	$\wedge := \_$
OUT Equation	$\text{OUT}[B] = f_B(\text{IN}[B])$
IN Equation	$\text{IN}[B] = \wedge_{p \in \text{pred}(B)} \text{OUT}[p]$
Initial Condition	$\text{OUT}[B] = \mathbb{U}$
Boundary Condition	$\text{OUT}[\text{entry}] = \emptyset$

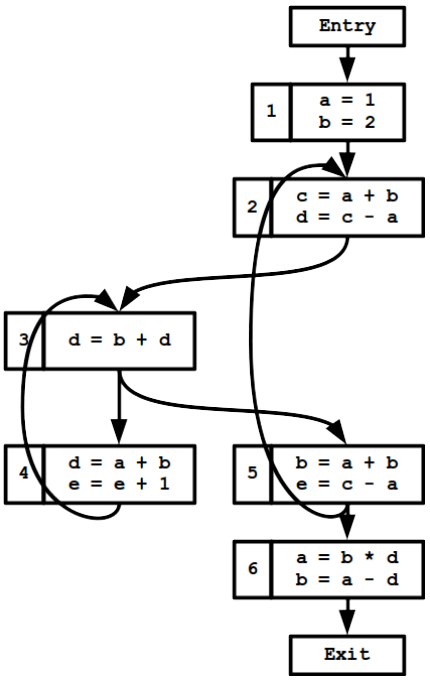

Figure 1: Code for Analysis

Table 3: Solution Format

BB	GEN	KILL	IN	OUT
1				
2				
...				

Q: Can I work in group of one? A: Yes. You can, but we want to remind you that working in group of one would not give you any advantage in terms of grading.

For people working in group of two, although it is up to you to decide how to distribute the work evenly between the group members, *both of you are responsible for knowing all the assignment materials as they will be tested in the exams.*

Q: Are we allowed to include headers other than those that are provided by LLVM (e.g., STL, Boost)? Yes. You can, but we strongly doubt whether this is truly necessary to use libraries beyond those of LLVM and STL in this course.

Q: How do we know what each type of instruction does? For most instructions you can directly infer from their names. Please refer to the LLVM Language Reference Manual for more detailed information.

Q: Do we need to develop our own test cases? What is your expectation on test cases? A: Yes. You have to write your own test cases and *provide explanations on why they justify the correctness of your programs*.

By "*complex source files*", we are not saying that the test cases have to be long, as it makes it hard for both you and us to verify the correctness of your programs. What we really mean is that *the test cases should cover all the corner cases you have in mind when you develop your optimization passes*.

Q: Suppose that we have the statement $i = i + 1$. Should we treat expression $i + 1$ as *available* after the statement? No. It is not. The reason is because – Suppose that we have statement $j = i + 1$ right after $i = i + 1$, clearly we cannot obtain the value of j directly from the previously computed $i + 1$.