Assignment 1: Introduction to LLVM

Due Date: Feb. 1st, Total Mark: 100 pts

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

ABSTRACT

Welcome to CSCD70H3 S (Winter) 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 x86 machines, since that is where they will be graded. We strongly recommend that assignments be done in the Linux VM that we provide.

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 use LLVM to analyze code to output interesting properties about your program (Section 4.1) and to perform local optimizations (Section 4.2).

1 POLICY

1.1 Collaboration

You will work in groups of two people to solve the problems for this assignment. **Please turn in a single writeup per group, indicating the names and UTORid of both group members.**

1.2 Submission

Please include the following items in an archive labeled with the UTORid of both group members (`a1-utorid1-utorid2.tar.gz`), and email the resulting file to **bojian@cs.toronto.edu**. Please 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 and integrated using the command `make all`).
- Two subfolders named **tests** that contain all the microbenchmarks that were used for verification of your code.

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

2 SETUP: THE LLVM COMPILER

To ensure that all assignments are graded in the same environment with LLVM 5.0.1, we will be distributing a virtual machine based on 32-bit Ubuntu 16.04 (Xenial). **You must ensure that all of your code works in this image, but you are not required to do all of your development in it.**

The VirtualBox software is available on several platforms from `https://www.virtualbox.org/`. The provided image was built with version 5.2. On some machines, you may need to enable virtualization extensions in the BIOS. The virtual machine image is available from `/cmshome/pekhimen/cscd70w18_space`. The machine name is `CSCD70`, and an account has been created with **user name `cscd70` and password `llvm`**. The LLVM source files are located at `/Workspace/LLVM/`.

3 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 the dummy LLVM pass to demonstrate how to build and run LLVM passes on programs. Using the provided `Makefile`, make sure that you can `make` this dummy pass.

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

```
clang -O -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 ./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 -load FunctionInfo/FunctionInfo.so
       -function-info loop.bc -o loop-finfo.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 might need to clean the previous output files using the command `make clean`), you should be able to see the same output from the command line. It is up to you to decide whether to keep the pass building and IR optimization together or separate.

4 PROBLEM STATEMENT

4.1 Function Information [40 pts]

Program analysis is an important prerequisite to applying optimizations, we want to improve code, not break it. For example, before the optimizer can remove some piece of code to make program run faster, it must examine other parts of the program to determine whether the code is truly redundant. A compiler pass is the standard mechanism for analyzing and optimizing programs.

You will 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

As an example, the expected output of running `FunctionInfo` on the optimized bytecode is shown in Table 1. As you can see, the output in Table 1 is not that interesting, since `loop.c` is a fairly trivial piece of code. Note, however, 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, even when using the `-O0` flag. **When reporting the number of calls, please count the number that appear in the bytecode, even if this 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 grading will be done with complex programs. Feel free to hand in your additional testing source files in a separate directory together with your source code.

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

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

4.2 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 on basic blocks as discussed in class. While there are many types of local optimizations, we will keep things quite simple in this section and focus only on the algebraic optimizations. Specifically, you will implement the following local optimizations:

(1) Algebraic Identities

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

(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 in the file `LocalOpts/LocalOpts.cpp` following the steps in Section 4.1. Because this will be an optimization pass rather than an analysis pass, there will be some small differences from the set up of the `FunctionInfo` pass. Provide an appropriate `Makefile` at `LocalOpts/Makefile` (again, please try to write your `Makefile` in such a way that the pass can be built using the command `make all`).

To better test your pass, you should build **unoptimized** LLVM bytecode from the test cases using the following commands:

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

(you may assume that all input to your pass will first go through `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 Identities: 2
        Strength Reduction: 3
        Multi-Inst. Optimization: 1
```

5 THEORETICAL QUESTIONS

5.1 Control Flow Graph (CFG) [5 pts]

Consider the following code and answer the questions below:

```
S1: x = x + 1
S2: if (y < 10) goto S5
S3: k = k + 1
S4: x = x + 1
S5: if (x < 100) goto S4
S6: y = y + 1
S7: x = 0
S8: if (y < 100) goto S1
S9: print x
S10: print y
S11: return
```

(1) Identify the **leader instruction** and their corresponding **basic blocks**. Draw the **CFG**.
(2) Identify the **back-edge(s)** in the CFG drawn in the part 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.

5.2 Natural Loops [5 pts]

Find the describe the natural loops 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 = 100
    y = 0
    go to L4
L1: y = x * y
    if (x < 50) go to L2
    y = x - y
    go to L3
L2: y = x + y
L3: print y
    if (y < 1000) go to L1
    if (x <= 0) go to L5
L4: x = x - 1
    go to L1
L5: return y
```

5.3 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 schema 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.

Based on this definition and the corresponding dataflow analysis description in Table 2, perform available expressions analysis on the code in Figure 1.

Table 2: Definitions of Available Expressions

Domain	Sets of Expressions
Directions	Forwards
Transfer Function	$\text{gen}_B \cup (x - \text{kill}_B)$
Boundary	OUT[entry] = 0
Meet ($\wedge$)	$\cap$
OUT Equation	$\text{OUT}[B] = f_B(\text{IN}[B])$
IN Equation	$\text{IN}[B] = \wedge_{P,\text{pred}(B)}\text{OUT}[P]$
Initial	$\text{OUT}[B] = \mathbb{U}$

For each basic block, list the **GEN**, **KILL**, and final **IN** and **OUT** sets, after the available expressions analysis is performed, as is shown below in Table 3. You may ignore expressions inside conditional statements.

Table 3: Solution Format

BB	GEN	KILL	IN	OUT
1				
2				
...				

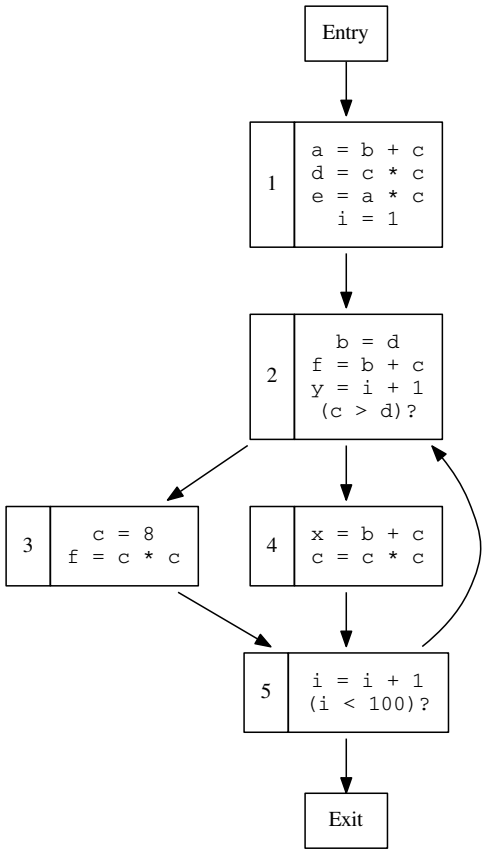

Figure 1: Code for Analysis