Assignment 1 Introduction to LLVM

Due Date: **Jan. 27th (Friday)**, 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 the 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 they have all necessary software components installed. 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 and to perform local optimizations.

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 (see also Section 5.**Q1**).

1.2 Submission

Please submit all your files to GitHub Classroom (here is the link to accept the assignment). Make sure that your submission includes the followings:

```
writeup.pdf

FunctionInfo/lib/FunctionInfo.cpp
FunctionInfo/CMakeLists.txt
FunctionInfo/test/...

LocalOpts/lib/AlgebraicIdentity.cpp
LocalOpts/lib/StrengthReduction.cpp
LocalOpts/lib/MultiInstOpt.cpp
LocalOpts/CMakeLists.txt
LocalOpts/test/...
```

- A report named `writeup.pdf` that has answers to the theoretical questions, and **optioanlly** describes the implementation details of your passes.
- Well-commented source code for your passes, together with a build file `CMakeLists.txt`. Please make sure 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 Example: Creating a Pass

The source file `FunctionInfo/lib/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`, make sure that you can build this pass with the command:

  ```
  cd FunctionInfo
  make
  ```

- Compile the testing source code `test/Loop.c` to an *optimized* LLVM bytecode object `Loop.bc` using `clang`, which is LLVM's frontend for the C language family:

  ```
  export LLVM_VERSION=16
  export PATH=${PATH}:/usr/lib/llvm-${LLVM_VERSION}/bin
  clang -O2 -emit-llvm -c ./test/Loop.c \
      -o ./test/Loop.bc
  ```

  and inspect the generated bytecode using `llvm-dis`:

  ```
  llvm-dis ./test/Loop.bc -o=./test/Loop.ll
  ```

  This will create a readable disassembly listing in `Loop.ll` of the `Loop.bc` bytecode.
- Run the dummy pass `FunctionInfo` on the bytecode using `opt` with the command:

  ```
  opt -load-pass-plugin=./libFunctionInfo.so \
      -passes=function-info ./test/Loop.bc \
      -o ./test/LoopFunctionInfo.bc
  ```

  Note the use of flag `-passes=function-info` to enable this pass (see if you can locate the declaration of this flag in the source file `FunctionInfo/lib/FunctionInfo.cpp`).
- If everything goes well,

  ```
  CSCD70 Function Information Pass
  ```

  should be printed to `stdout`. This can be verified with the `FileCheck` directive in the comments of `Loop.c`:

  ```
  // SAMPLE: CSCD70 Function Information Pass
  ```

  The directive specifies our expected output, which can be used to check against the real output from `opt`:

  ```
  opt -load-pass-plugin=./libFunctionInfo.so \
      -passes=function-info ./test/Loop.bc \
      -disable-output[1] \
  | $(llvm-config --bindir)[2]/FileCheck \
      --check-prefix=SAMPLE[3] \
      ./test/Loop.c
  ```

1. Do not generate any output files, because in this specific example we are only concerned about the outputs from the print statements, which will be passed to the `FileCheck` command using pipes ('`|`' in the next line).
2. Since `FileCheck` is not added to the default executable search path, we have to specify its absolute path using `llvm-config --bindir`.
3. If the option `--check-prefix` is neglected, `FileCheck` will use `CHECK` as the default directive.

- There is a provided `CMakeLists.txt` with the starter code that automatically goes through all the above process (see if you can locate the equivalent execution commands in the `RUN` directive of `Loop.c`):

```
rm test/Loop.ll[1]
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
make test[2]
```

1. Remove the previously generated `Loop.ll` because otherwise it will be treated as another test case by the tester.
2. When debugging, you can use the command `ctest -V` instead to inspect the output from the test cases.

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 ("`N+*`" in the case of variadic arguments (where N is the number of non-variadic ones), e.g., function

   ```
   int printf(const char *format, ...);
   ```

   should print "`1+*`".
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.

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. Although there are many of them, we will keep things simple in this section and focus only on the algebraic optimizations, the scope of which is a single ~~basic block~~function. Specifically, you will implement the following local optimizations:

1. Algebraic Identity

$$x + 0 = 0 + x, x \times 1 = 1 \times x \Rightarrow x$$

2. Strength Reduction

$$4 \times x = x \times 4 \Rightarrow \text{ or } x \ll 2$$

3. Multi-Instruction Optimization

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

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. For example, you should build **unoptimized** LLVM bytecode from the test cases with the commands:

```
clang -O0 -Xclang -disable-O0-optnone[1] -emit-llvm \
    -c Test.c
opt -passes=mem2reg Test.bc -o TestM2R.bc[2]
```

1. If you do not add the `-Xclang -disable-O0-optnone` option, further optimizations such as `mem2reg` will be disabled.
2. The `mem2reg` optimization pass promotes the variables from memory to registers. This greatly simplifies the bytecode. You can ignore this optimization pass and check what the bytecode looks like.

(you may assume that all inputs to your pass will first go through the `mem2reg` pass as shown above). Alternatively, you could also write microbenchmarks to test your code, as is shown in the provided `Foo.ll`.

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 at the head.

```
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)
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 program point p if every path from the entry 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. Based on the definitions above, answer the questions below:

(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.

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	$\text{OUT}[\text{entry}] = \emptyset$

(2) Perform available expressions analysis on the CFG 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.

Table 3. Solution Format

BB	GEN	KILL	IN	OUT
1				
2				
...				

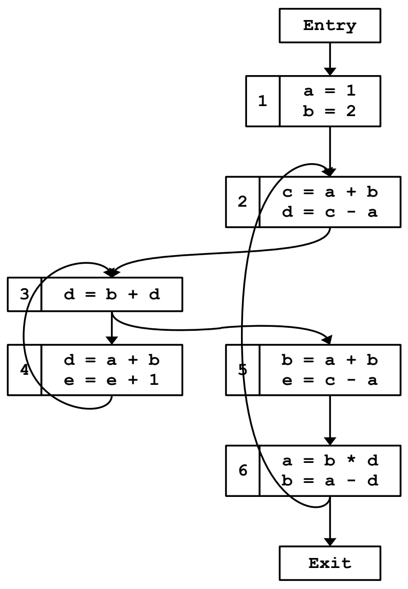

Figure 1. CFG to analyze

5 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 [Logistics] Can I work in groups of one?

A. Yes. You can. However, please note that working in groups of one would not give you any advantage in terms of grading. For people working in groups of two, although it is up to you to distribute the work, both of you are responsible for knowing all the assignment materials as they will be tested in the exams.

Q2 [Section 3] Are we allowed to include headers other than those that are provided by LLVM (e.g., STL)?

A. Yes. You can. However, we strongly doubt whether you need libraries beyond those of LLVM and STL in this course.

Q3 [Section 3] How could we inspect the generated assembly of the testing source files (e.g., `Fibonacci.c`)?

A. The generated file is located inside the `build` folder, i.e., `build/test/Fibonacci.c.ll`.

Q4 [Section 3] How do we know what each type of instruction does?

A. For most instructions you can directly infer from their names. Please refer to the LLVM Language Reference Manual for more detailed information.

Q5 [Section 3] Do we need to write our own test cases? What is your expectation for the test cases?

A. You have to complete the `CHECK` directives on the provided test cases. You do not have to add new test cases. Your first priority is to make sure that your optimization passes function correctly on the provided test cases.
However, please feel free to add test cases to the test suite for the corner cases you have in mind during development.

Cases	**Need to handle?**	**Comments**
Floating-point numbers	✗	
Negative numbers	✗	
$a \times 0 = 0$	✗	
$a - 0 = 0, a/1 = 1$	✓	
$a/8 = a \gg 3$	✓	Please be generic and handle for all powers of 2, same with multiplications.
$a \times 7 = a \ll 3 - a$	✗	
$a = t - b, c = t - a \longrightarrow a = t - b, c = b$	✗	
$a = b + t, c = a - t \longrightarrow a = b + t, c = b$ (t is not a constant)	✓	
Uses in a different basic block	✓	

Table 4. Cases to cover in the Local Optimizations (Section 3.2)

To facilitate the process, we have provided a tool named `C_to_LLVM_IR` in the GitHub repository.

Q6 [Section 3.2] For the *Strength Reduction* optimization pass, do we need to handle the cases that are listed in Table 4?

A. Please refer to the table for detailed comments.

Q7 [Section 3.2] Suppose that we get one of the local optimizations wrong, would this affect our grades of other passes?

A. No. Each pass is graded separately.

Q8 [Section 4.3] What counts as an expression?

A. In the case of *available expressions* analysis, we are only concerned about binary expressions.

Q9 [Section 4.3] Is an expression in the *kill* set of a basic block if that expression never goes into that basic block (e.g., $a + b$ in BB_1)?

A. Both solutions are fine, because they output the same available expressions. However, for succinctness, it is better not to include expressions that do not go into a basic block.

Q10 [Section 4.3] Consider $i = i + 1$, should we treat expression $i + 1$ as *available* after the statement?

A. 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 $i + 1$.