Assignment #1

15-745 Spring 2018

Assigned: Friday, January 19
Due: Monday, February 5, 11:59PM

Abstract

Welcome to the Spring 2018 edition of Optimizing Compilers (15-745). 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 recommended 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 2.3.1) and to perform local optimizations (Section 2.3.2).

1 Introduction

1.1 Policy

You will work in groups of two people to solve the problems for this assignment. Turn in a single writeup per group, indicating all group members. The course Canvas has a way to indicate your group for the assignment. This can be done via the 'People' tab on the course page. Please do so.

1.2 Logistics

All clarifications (if any) to this assignment will be posted on the class discussion board on Piazza. Any revisions will be uploaded to the class web page.

1.3 Submission

Please include the following items in an archive labeled with the Andrew ID of one member in your group (e.g., `bovik.tar.gz`), and submit the resulting file to Canvas. Ensure that when this archive is extracted, the files appear as follows:

```
./bovik/README
./bovik/FunctionInfo/FunctionInfo.cpp
./bovik/FunctionInfo/Makefile
./bovik/LocalOpts/LocalOpts.cpp
./bovik/LocalOpts/Makefile
./bovik/writeup.pdf
./bovik/tests/
```

- A report that briefly describes the implementations of both passes, and answers the homework questions, named `writeup.pdf`, containing the Andrew ID's of all members in your group.
- Well-commented source code for your passes (`FunctionInfo` and `LocalOpts`), and associated `Makefile`s.
- A `README` file describing how to build and run your passes.
- Any tests used for verification of your code.

2 LLVM Project

2.1 Obtaining the System Image

To ensure that all assignments are graded in the same environment with LLVM 5.0.1, we are distributing a virtual machine based on 32-bit Lubuntu 16.04 (Xenial Xerus). 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 `http://www.virtualbox.org`. The provided image was built with version 5.2.4. On some machines, you may need to enable virtualization extensions in the BIOS (reboot, press F12 to get the boot menu, choose "System Setup", then "Virtualization Support", then make sure that the box is checked). The virtual machine image is available over AFS at `/afs/cs.cmu.edu/academic/class/15745-s18/www/vm-images/15745-S18Lubuntu.ova`, or over HTTP at here.

In the same directory, there is a checksum file named `15745-S18Lubuntu.ova.sha1` that can be used to verify integrity of the downloaded image, as follows: `sha1sum -c 15745-S18Lubuntu.ova`. To import the virtual machine image into VirtualBox, either double-click the image file, or launch VirtualBox, select the "File" menu, and click "Import Appliance".

The machine name is `15745-S18Lubuntu`, and an account has been created with username `user` and password `user`. The LLVM binaries are located in `/home/user/llvm/llvm-5.0.1.install` (or `LLVM_ROOT`) and the source files are located in `/home/user/llvm/llvm-5.0.1.src`. `LLVM_ROOT/bin` has also been added to the `PATH`.

For ease of use, we recommend keeping your source code in the filesystem of the host, and sharing that directory with the virtual machine using the VirtualBox "Shared Folder" feature. This enables both the host and virtual machine to access the same directory simultaneously, and can be set up in the VirtualBox GUI under "Settings", then "Shared Folders". You can automatically mount this shared directory within the virtual machine by enabling the auto-mount option, which will mount the shared folder at `/media/sf_foldername`, where `foldername` is the name of the shared folder.

Look through the LLVM 5.0.1 Documentation, Programmer's Manual and Writing an LLVM Pass Tutorial. You may also find the primary LLVM Documentation and Doxygen pages to be useful, but keep in mind that they correspond to the latest source code in the LLVM SVN repository, not version 5.0.1 that we use. While there should not be any significant changes that affect this class, keep in mind that the two may diverge due to changes in the LLVM API. For ease of reference, we are making the Doxygen documentation for LLVM 5.0.1 available on the course website here.

2.2 Creating a Pass

Create a directory named `FunctionInfo` and copy `FunctionInfo.cpp` (provided with the assignment) into the new directory. `FunctionInfo.cpp` contains a dummy LLVM pass for analyzing the functions in a program. Currently it prints out "15745 Function Information Pass". In the next

Command	Options	Meaning
`clang`		Compile high-level source code, such as C
	`-Xclang <option>`	Pass `option` to the clang compiler
	`-disable-O0-optnone`	Do not generate the optnone function attribute which prevents further passes from running.
	`-O`	Set level of optimization performed by `clang` (to the default)
	`-O0`	Direct `clang` not to perform any optimization
	`-emit-llvm`	Output an LLVM bytecode object
	`-c`	Output object code, do not fully compile
`llvm-dis`		Generate disassembly of LLVM bytecode
`opt`		Run LLVM passes
	`-load`	Load the pass found in the corresponding file (file path must be specified)
	`-{passname}`	Load the pass with this name
	`-mem2reg`	Load the mem2reg pass, which simplifies LLVM bytecode
	`-o`	Specify the output filename
	`-S`	Print the disassembly of the output LLVM bytecode

Table 1: A simplified summary of some useful `LLVM` commands and arguments.

section, you will extend `FunctionInfo.cpp` 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.

Next, copy the `loop.c` source code (shown in Figure 1(a)) from `FunctionInfo/loop.c` into your local `FunctionInfo` directory. Compile it to an optimized LLVM bytecode object (`loop.bc`) as follows: `clang -O -emit-llvm -c loop.c` (`clang` is the LLVM project's frontend for the C language family.)

Then, inspect the `loop.bc` generated bytecode using `llvm-dis` with the command `llvm-dis loop.bc`. This will create a disassembly listing in `loop.ll` of the `loop.bc` bytecode. These should appear similar to Figure 1(b).

Now, try running the dummy `FunctionInfo` pass on the bytecode. To do this, use the `opt` command as follows: `opt -load path/to/FunctionInfo.so -function-info loop.bc -o out`. Note the use of the command line flag "`-function-info`" to enable this pass. (See if you can locate the declaration of this flag in `FunctionInfo.cpp`). Note that you must provide the correct path to FunctionInfo.so, and may need to use "`./`" if you are in the same directory.

If all goes well, "15745 Function Information Pass" should be printed to stderr.

2.3 Analysis Passes

2.3.1 Function Information [20 pts]

Program analysis is an important prerequisite to applying correct optimizations: we want to improve code, not break it. For example, before the optimizer can remove some piece of code to make a 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 now 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

```c
int g;
int g_incr (int c)
{
  g += c;
  return g;
}
int loop (int a, int b, int c)
{
  int i;
  int ret = 0;
  for (i = a; i < b; i++) {
   g_incr (c);
  }
  return ret + g;
}
```

(a)

```llvm
@g = common local_unnamed_addr global i32 0, align 4

; Function Attrs: norecurse nounwind
define i32 @g_incr(i32 %c) local_unnamed_addr #0 {
entry:
  %0 = load i32, i32* @g, align 4
  %add = add nsw i32 %0, %c
  store i32 %add, i32* @g, align 4
  ret i32 %add
}

; Function Attrs: norecurse nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) local_unnamed_addr #0 {
entry:
  %cmp4 = icmp sgt i32 %b, %a
  %0 = load i32, i32* @g, align 4
  br i1 %cmp4, label %for.body.lr.ph, label %for.end

for.body.lr.ph:                         ; preds = %entry
  %1 = sub i32 %b, %a
  %2 = mul i32 %1, %c
  %3 = add i32 %0, %2
  store i32 %3, i32* @g, align 4
  br label %for.end

for.end:                                ; preds = %for.body.lr.ph, %entry
  %.lcssa = phi i32 [ %3, %for.body.lr.ph ],
                    [ %0, %entry ]
  ret i32 %.lcssa
}
```

(b)

Figure 1: (a) A simple loop source code, and (b) its LLVM bytecode.

about all functions that appear in a program:

1. Name.
2. Number of arguments (or `*` if variable).
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.

To assist you in writing this pass, the expected output of running `FunctionInfo` on the optimized bytecode (Figure 1(b)) is shown in Table 2. As you can see, the output in Table 2 is not that interesting, since `loop.c` is a fairly trivial piece of code. Note, however, that llvm optimized the call to `g_incr` in `loop`. 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.

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

Table 2: Expected FunctionInfo output for the optimized bytecode of `loop.c`

2.3.2 Local Optimizations (Purple Dragon Book 8.5.4) [20 pts]

Now that you are an expert at writing LLVM passes, it is time to write a pass for making programs faster. You will implement optimizations on basic blocks as discussed in class. More details on local optimizations are available in Chapter 8.5 of the Purple Dragon Book. While there are many types of local optimizations, we will keep things quite simple in this section and focus only on the algebraic optimizations discussed in Section 8.5.4 of the book. Specifically, you will implement the following local optimizations:

1. Algebraic identities: e.g, `x + 0 = 0 + x => x`
2. Constant folding: e.g, `2 * 4 => 8`
3. Strength reductions: e.g, `2 * x = x * 2 => (x + x) or (x « 1)`

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 (scalar) integer types.

2.4 Implementation Details

You should create a new LLVM pass in a file named `LocalOpts/LocalOpts.cpp` following the steps in Section 2.2. 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`. (Note that it is possible to implement more than one pass in the same directory or file, but we're trying to keep things clean.)

To better test your pass, you should build unoptimized LLVM bytecode from the test cases using the following commands: `clang -Xclang -disable-O0-optnone -O0 -emit-llvm -c infile.c -o infile.bc; opt -mem2reg infile.bc -o infile-m2r.bc`.

You may assume that all input to your pass will first go through `mem2reg` pass as shown above.

Then, we should be able to run your local optimization pass in the following way, from the location of the shared library: `opt -load ./LocalOpts.so -local-opts infile-m2r.bc -o out`

In addition to transforming the bytecode, your pass should also print to standard out 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:

```
Transformations applied:
  Algebraic identities: 2
  Constant folding: 1
  Strength reduction: 3
```

We will provide toy source files with unrealistic amounts of local optimization opportunities for you to debug your pass in: `tests`. In addition to using these test inputs, we recommend that you test your pass on more realistic programs.

3 Homework Questions

3.1 CFG Basics [20 pts]

For the code provided below, (i) find (maximal) basic blocks, and (ii) build the CFG (Control Flow Graph). Be sure to give your basic blocks clear labels that correspond to the original code, and include "entry" and "exit" blocks. You may assume that the `print` function always returns, and that the code is not optimized.

```
    x = 50
    y = 8
    z = 234
    if (x < z) { goto L1 }
    y = 89
    goto L2
L1: z = 65
    return z
L2: y = x + 1
    if (z < x) { goto L3 }
    x = 25
L3: y = x + z
    switch (y) { 334: goto L4 | default: goto L5 }
L4: print("failure")
L5: y = 65
    return y
```

3.2 Available Expressions (Purple Dragon Book 9.2.6) [20 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.

Based on this definition and the corresponding dataflow analysis description (See Table 3 from New Dragon Book 9.2.7) perform Available Expressions analysis on the code in Figure 2.

Domain	**Direction**	**Transfer Function**	**Boundary**
Sets of expressions	Forwards	$gen_B \cup (x - kill_B)$	$OUT[entry] = \emptyset$
Meet ($\wedge$)	**OUT Equation**	**IN Equation**	**Initial**
$\cap$	$OUT[B] = f_B(IN[B])$	$IN[B] = \bigwedge_{P,pred(B)} OUT[P]$	$OUT[B] = \mathbb{U}$

Table 3: Definnition of available expressions dataflow analysis.

For each basic block, list the `GEN`, `KILL`, and final `IN` and `OUT` sets, after the available expressions analysis is performed, as shown below. You may ignore expressions inside conditional statements (e.g., $i > 42$).

BB	**GEN**	**KILL**	**IN**	**OUT**
1				
2				
3				
4				
5				

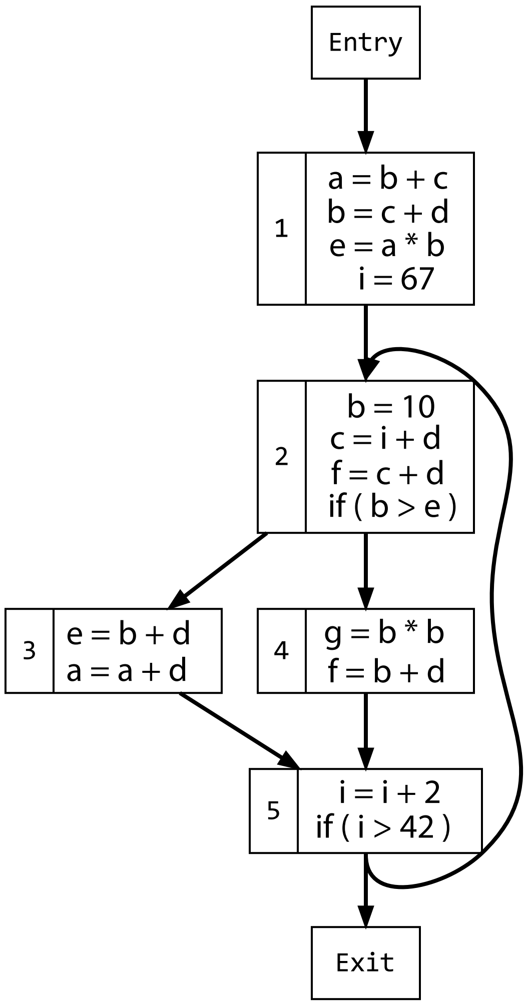

Figure 2: Code for available expressions dataflow analysis.

3.3 Faint Analysis [20 pts]

You have been hired to help develop a software analysis package that will detect *faint* expressions, which are useful for performing Dead Code Elimination (*DCE*). The idea behind DCE is that an assignment of the form "`x = t`" can be eliminated if its LHS variable `x` is not live (i.e dead) at the program point P immediately following the assignment. One of the limitations of DCE is that it cannot directly eliminate the assignment "`x = x + 1`" in the two examples shown below:

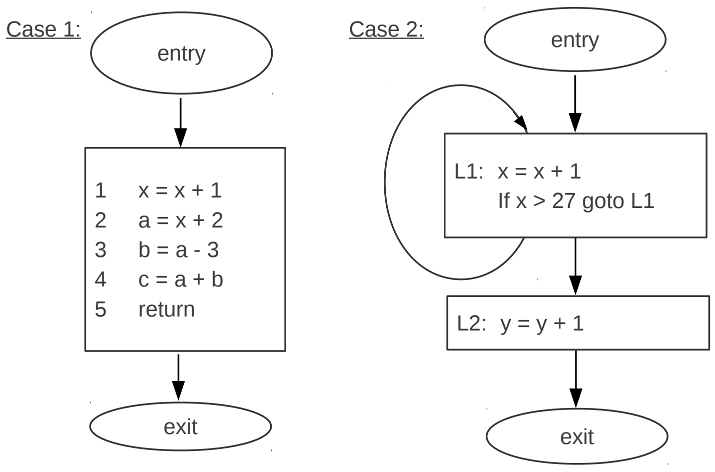

Figure 3: Code for faint expressions dataflow analysis.

In the first case, `x` is not dead after the "`x = x + 1`" assignment (instruction 1) because it is used in instruction 2. Instruction 2 is also not dead because its LHS variable "`a`" is used in instructions 3 and 4. However, instruction 4 is in fact dead. If we applied DCE repeatedly to this code, we could eventually eliminate instruction 1. However, it would be more desirable to eliminate "`x = x + 1`" in a single data flow pass.

In the second case, the LHS of "`x = x + 1`" is not dead because it is used by its own RHS due to the cycle in the flow graph. However, since the ultimate value of `x` is never used, this instruction could in fact be safely eliminated from the loop body. Nevertheless, for the purposes of this assignment, the conditional expression "`if x > 27 goto L1`" can be treated as a side-effect of `x`, causing it to be live and thus not faint.

We say that the LHS variable `x` in an assignment "`x = t`" is *faint* if along every path following the assignment, `x` is either dead or is only used by an instruction whose LHS variable is also faint.

Your mission in this assignment is to design a new dataflow analysis pass specifically for determining whether the LHS variable of an expression is faint, in both of these cases. Your analysis should be as simple as possible (i.e., it should not gather unnecessary information), and as fast as possible. Your analysis will be plugged into a generic dataflow framework (e.g., Purple Dragon Book 9.2 - 9.3). For the purposes of this assignment, you should assume that the return values of functions are not faint. Though not precise, you should also assume that arguments to function calls are not faint, to keep things simple.

1. Define the set of elements that your analysis operates on.
2. Define the direction of your analysis.
3. Define the transfer function. Make sure to define any sets that your transfer function uses.

4. Define the meet operator, and give the equation that uses this operator.
5. What are the value(s) that `ENTRY` and/or `EXIT` is initialized to?
6. What are the value(s) that the **IN** and/or **OUT** sets are initialized to?
7. What impact, if any, does the order of basic block traversal have on the correctness of your analysis? What order would you implement and why?
8. Will your analysis converge? Please explain in a few sentences; no proof is necessary.
9. In pseudo-code, give an algorithm that performs the dataflow analysis and removes faint expressions, using your definitions from the previous subproblems.