Lab 0: Introduction to LLVM

Spring Semester 2020

Due: 20 January, at 8:00 a.m. Eastern Time

Objective

This lab involves running and extending LLVM, a popular compiler framework for a large class of programming languages, that will be used to implement all the labs in this course. You will setup the LLVM framework in the provided course VM and implement an LLVM pass that will perform simple analytics on an input program. Specifically, your pass will compute the number of functions and instructions in the input C program.

Resources

- Tools used in the course:
  - http://cmake.org/cmake-tutorial/
  - http://www.gnu.org/software/make/manual/html_node/Simple-Makefile.html
  - http://releases.llvm.org/8.0.0/tools/clang/docs/UsersManual.html
  - http://releases.llvm.org/8.0.0/docs/CommandGuide/index.html

- LLVM Programming:
  - http://releases.llvm.org/8.0.0/docs/index.html
  - http://releases.llvm.org/8.0.0/docs/WritingAnLLVMPass.html
  - http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#basic-inspection-and-traversal-routines

Setup

The LLVM framework is pre-installed on the course VM and the skeleton code for Lab 0 is located under `/home/cs6340/lab0/`. We will refer to this top-level directory for Lab 0 simply as `lab0` when describing file locations for the lab.

Throughout the labs, we will use `CMake`, a modern tool for managing the build process. If you are unfamiliar with `CMake`, you are strongly advised to read the CMake tutorial first (please pay attention to Step 1 and Step 2.) Running `cmake` produces a Makefile that you might be more familiar with. If not, read the Makefile tutorial before proceeding further. Once a Makefile is generated, you need only call `make` to rebuild your project after editing the source files.

The following commands set up the lab:

```
$ cd ~/lab0
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Among the files generated, you should now see `PrereqPass.so` in the current directory. `PrereqPass.so` is built from `lab0/src/PrereqPass.cpp` which you will modify shortly.

`clang` is a C language compiler that uses LLVM and serves as a drop-in replacement for the `gcc` compiler. If you know how to use `gcc`, you should be fine with `clang`. Otherwise, you should scan the user manual to familiarize yourself with some of the `clang` command line options.

While we typically think of building static and dynamic analysis tools for C programs, our actual LLVM passes must run on LLVM IR -- think of it as LLVM assembly language. If we want to run our PrereqPass on some C program, we first compile that C program down to LLVM IR:


```
$ cd ~/lab0/test
$ clang -emit-llvm -S -fno-discard-value-names -c simple0.c
```


Some of these options may be unfamiliar to you: `-S` instructs clang to perform preprocessing and compilation steps only, `-emit-llvm` instructs the compiler to generate LLVM IR (which will be saved to `simple0.ll`), and `-fno-discard-value-names` prevents clang from discarding names in the generated LLVM.

`opt` is a tool from LLVM that performs analyses and optimizations on LLVM IR. We will use `opt` to run our custom LLVM pass on the compiled C code:

```
$ cd ~
$ opt -load lab0/build/PrereqPass.so -Prereqs -disable-output
lab0/test/simple0.ll
```

Note that `-load` loads our LLVM pass library, but `-Prereqs` actually instructs opt to run the pass on `simple0.ll`. (Libraries can and often do contain multiple LLVM compiler passes.) You should consult the documentation for opt and understand the potential ways to use the tool; it may help you build and debug your solutions.

You should see the sample output from the above command as follows:

```
Analytics of Module lab0/test/simple0.ll
  # Functions    : 0
  # Instructions : 0
```

Lab Instructions

LLVM passes are subprocesses of the LLVM framework. They usually perform transformations, optimizations, or analyses on programs. We have templated a pass for you to modify; in particular, you will need to edit the `runOnModule` function in the pass file (`lab0/src/Prereqs.cpp`) so that it correctly prints the number of functions and instructions for an input C program.

You will have to learn how to utilize many classes in the LLVM API. Some of the more general ones that you will use and reuse throughout the labs in this course are `Module`, `Function`, `BasicBlock`, and `Instruction`. You should get in the habit of scanning some of the LLVM class documentation, as each contains many useful methods that you might find helpful in implementing some of the labs. We also follow the LLVM coding standards for code (variables, types, classes etc.). At the very least, you should consult the relevant section, but the standard serves as a good programming guideline.

Now, returning to the task at hand, think of the `runOnModule` function as the main entry point to your compiler pass. Inside a `Module`, you can find all program `Functions`. In LLVM, a function consists of one or more `BasicBlocks` that contain `Instructions`. Traversing over these program elements is common when working with LLVM and you should expect to get familiar with various traversal techniques.

In short, you should break down the lab into the following tasks:

1. Find all functions in a `Module`.
2. For each `Function`, count its instructions using one of the traversal techniques in the documentation.
3. Update `NumOfFunctions` and `NumOfInstructions` accordingly.

LLVM Pass Structure. At the end of `Prereqs.cpp` you will notice some additional utility code:

```
char Prereqs::ID = 1;
static RegisterPass<Prereqs> X("Prereqs", "Prereqs", false, false);
```

The code hooks our `Prereqs` class into the LLVM framework so we can use it via the `opt` command. We register our `Prereqs` pass via `RegisterPass<Prereqs>`, and instruct LLVM to identify it on the command line as "Prereqs" with the arguments to `X`. It will help if you familiarize yourself with some of the basic code required to setup a new LLVM pass end-to-end.

Example Input and Output

Your pass should run on any C program that compiles to LLVM IR. As we demonstrated in the Setup section, we will compile some C programs to LLVM IR and then run your pass on them using `opt`:

```
$ cd ~
$ opt -load lab0/build/PrereqPass.so -Prereqs -disable-output
lab0/test/simple0.ll
```


If completed correctly, you should see the sample output as follows:

```
Analytics of Module lab0/test/simple0.ll
  # Functions    : 1
  # Instructions : 22
```


Items to Submit

We expect your submission to conform to the standards specified below. To ensure that your submission is correct, you should run the provided file validator. You should not expect submissions that have an improper folder structure, incorrect file names, or in other ways do not conform to the specification to score full credit. The file validator will check folder structure and names match what is expected for this lab, but won't (and isn't intended to) catch everything.

The command to run the tool is: `python3 zip_validator.py lab0 lab0.zip`

Submit the following files in a single compressed file (`.zip` format) named `lab0.zip`. For full credit, there must not be any subfolders or extra files contained within your zip file.

- Submit your modified version of the file `Prereqs.cpp`

Upload your zip file to Canvas. Make sure the spelling of the filenames and the extensions are exactly as indicated above. Misspellings in filenames may result in a deduction to your grade.