CS6215: Installing and Running LLVM

1 Installing LLVM

The LLVM compiler infrastructure is a collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends (source: Wikipedia). LLVM enable us to compile from different programming languages to different target architectures using the same tool. In this course, we will be using LLVM to implement some of the analyses presented in the course.

In this article, we will first explain the steps to install and run LLVM. Next, we will present the steps to run a simple analysis on a C program.

It is recommended that you install LLVM on Ubuntu 16 and above for this course. If you have difficulty installing LLVM or you do not have access to an Ubuntu machine email Rasool (`rasool@comp.nus.edu.sg`, make sure your subject line begins with "CS5218"). We will provide you an account for accessing an Ubuntu server which has LLVM pre-installed on it.

Use the following steps to install LLVM on Ubuntu 16 or Ubuntu 18:

- Install LLVM on Ubuntu 16:
  ```
  sudo apt-get install llvm3.5-dev clang-3.5
  ```
- Install LLVM on Ubuntu 18:
  ```
  sudo apt-get install llvm3.9-dev clang-3.9
  ```

2 LLVM Intermediate Representation

An Intermediate representation (IR) is the data structure or code used internally by a compiler to represent the source code. LLVM IR is a low-level intermediate representation used by the LLVM compiler framework. You can think of LLVM IR as a platform-independent assembly language with an infinite number of local registers.

When developing compilers there are huge benefits with compiling your source language to an intermediate representation (IR) instead of compiling directly to a target architecture (e.g. x86). As many optimization techniques are general (e.g. dead code elimination, constant propagation), these optimization passes may be performed directly at the IR level and thus shared between all target architectures.

Several different LLVM frontends have been developed which compile the source from different programming languages to LLVM IR. The frontend compiler `clang` generates LLVM IR from C. The following instructions generate the LLVM IR for a sample C program `test1.c`:

- Generating human-readable LLVM IR (change clang-3.5 to clang-3.9 in Ubuntu 18):
  ```
  clang-3.5 -emit-llvm -S -o test1.ll test1.c
  ```

- Generating binary LLVM IR:
  `clang-3.5 -emit-llvm -c -o test1.bc test1.c`

You can open `test1.ll` and have a look at the LLVM IR instructions generated. On the other hand, `test1.bc` file is a binary file which represents the same information but is not human readable. You can find a brief explanation on LLVM IR instructions in the LLVM tutorial slides (`LLVM_tutorial.pdf` slides 1-26).

3 Generating the Control Flow Graph by LLVM

The control flow graph (CFG) of a program can be generated from the LLVM IR using the following instruction (change opt-3.5 to opt-3.9 in Ubuntu 18):

```
opt-3.5 -dot-cfg test1.ll
```

or

```
opt-3.5 -dot-cfg test1.bc
```

This instruction generates a file named `cfg.main.dot`. This file contains the information of the graph representing the CFG of `test1.c`.

Run the following instructions to rename `cfg.main.dot` file to `test1.dot`:

```
mv cfg.main.dot test1.dot
```

Next, run the following instruction. The script in "allfigs2pdf" generates a PDF file `text1.pdf` which contains the CFG of `test1.c`:

```
allfigs2pdf
```

Finally, run the above instructions to generate the LLVM IR and CFG for `test2.c` and `test3.c`. Try to compare the generated LLVM IR and CFG of the three programs. Note, how the CFG is different in test3.c which contains a loop.

4 Running LLVM Pass

In this section, we will review the steps to compile and run an LLVM pass. The LLVM Pass Framework is an important part of the LLVM system, because LLVM passes are where most of the interesting parts of the compiler exist. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations.

There are 6 types of LLVM passes where we will be using only one type in this course (FunctionPass, which is performed on functions). LLVM contains many prebuilt LLVM passes such as "dead code elimination" pass. Users can also write their own passes. In the assignments in this course we will be developing LLVM passes which perform different analyses on the source code.

We start by showing you a sample pass, RandomPath, which randomly prints one of the paths in a source program. The source code of the pass is in the file `RandomPath.cpp`. For compiling the RandomPath LLVM pass run the following command:

```
   clang++-3.5 -o RandomPath RandomPath.cpp 'llvm-config-3.5 --cxxflags' 'llvm-config-3.5
--ldflags' 'llvm-config-3.5 --libs' -lpthread -lncurses -ldl
```

Note: Please make sure the instruction is pasted correctly to the terminal. If you saw any error messages, try copying it from the file `instruction.txt`.

Running this command will generate an executable file `RandomPath`. Note that we have used `clang++-3.5` which compiles C++ to LLVM IR and finally to an executable file.

Next, you would be able to run the LLVM Pass on the generated LLVM IR of a source program:

```
RandomPath test1.bc
```

or

```
RandomPath test1.ll
```

The LLVM pass will print a sequence of basic blocks which represent a path chosen randomly from the function. The first basic block is the entrance basic block in a function and the last basic block is the exit basic block.

Next, run the above instructions to print a random path for `test2.c` and `test3.c`. Finally, try to compare the different paths printed by this LLVM pass. Since `test3.c` contains a loop, if you run the pass multiple times, it might print the basic blocks in the loop several times. It is even possible that the pass continues to print basic blocks in the loop infinitely.

Try to find how you can fix this bug in the code of the LLVM pass!