How to Build and Run an LLVM Pass

The following instructions are for you to install LLVM and prepare the template LLVM Pass in your verifier account. If you have any trouble, please post a question on Noah BBS.

Build LLVM

1. Download the homework#4 material from course homepage (http://swtv.kaist.ac.kr/  
courses/cs453-fall14/hw4-dist.tar.gz.), and extract the files from hw4-dist.tar.gz at your home directory (e.g., tar –xvf hw4-dist.tar.gz).

Then, you will see there are thee directories llvm, runtime, and test. llvm contains the full package of LLVM 3.4 with Clang. We do not use the latest LLVM (ver. 3.5) as the other programs in verifierX are not compatible with it. The sample and the template LLVM Passes are located under llvm/lib/Transforms/IntWrite, CCov. runtime contains the runtime module working with the sample LLVM Pass IntWrite; test has the target programs example.c and grep\_1.2\_v0.

2. Build LLVM with the following sequence of commands:

|  |
| --- |
| $ **cd llvm**  llvm$ **./configure**  llvm$ **make –j4** |

Then, you will see that directory llvm/Debug+Asserts or llvm/Release+Asserts was built[[1]](#footnote-1). Usually, it takes several minutes for the compilation; however, you do not need to re-compile the whole LLVM afterward.

3. Add the following environment variables.

|  |
| --- |
| **export LLVM\_PATH=${PWD}/llvm**  **export LLVM\_LIB\_PATH=${LLVM\_PATH}/Debug+Asserts/lib**  **export LLVM\_BIN\_PATH=${LLVM\_PATH}/Debug+Asserts/bin**  **export PATH=${PATH}:${LLVM\_BIN\_PATH}** |

Now you can execute clang anywhere. We will use clang to run LLVM Passes.

Build LLVM Passes

4. The sample Pass IntWrite and the template pass CCov contain their own Makefile. For example, you can build IntWrite with by following commands:

|  |
| --- |
| llvm$ **cd lib/Transforms/IntWrite**  llvm/lib/Transforms/IntWrite$ **make** |

Then, IntWrite Pass is compiled as a static library and stored in the LLVM library directory   
(i.e., llvm/Debug+Asserts/lib/IntWrite.so)

5. You need to compile the runtime module as well. The runtime module would be compiled as an object file, and then given to the linker together with an instrumented target program. For example, you can compile intwrite.c, the runtime module of IntWrite as follows:

|  |
| --- |
| $ **cd runtime**  runtime$ **clang –c intwrite.c** |

Note that clang is compatible with gcc, and clang++ with g++.

Run LLVM Passes

6. You can instrument a target program with an LLVM Pass as you compile the target program. You can configure clang to run a given LLVM Pass in the middle of the compiling process, such that the produced binary gets modified. If the LLVM Pass inserts a new function declaration with its definition, you should link the object file with the definition.

For example, you can build test/example.c with IntWrite instrumentation as follows:

|  |
| --- |
| $ **cd test**  test$ **clang –g –Xclang –load -Xclang ${LLVM\_LIB\_PATH}/IntWrite.so example.c ../runtime/intwrite.o** |

Note that –g option was used for IntWrite to utilize the debugging information (e.g. line numbers).

As you execute a.out, you can see that log file is produced by the probe executions.

|  |
| --- |
| test$ **./a.out**  test$ **cat log**  Line 0: Variable retval <- 0  Line 15: Variable a <- 0  Line 23: Variable a <- 4  Line 27: Variable i <- 0  …  Line 47: Variable a <- 197 |

Similarly, you can run IntWrite for grep.c as follows:

|  |
| --- |
| $ **cd test/grep**  test/grep$ **clang –g –Xclang –load -Xclang ${LLVM\_LIB\_PATH}/IntWrite.so grep.c ../../runtime/intwrite.o** |

1. It depends on execution environment. Later on, I present only the Debug+Asserts case for simplicity. [↑](#footnote-ref-1)