Exercise on Translation to LLVM

CS 426 — Compiler Construction
Fall Semester 2019

This handout gives you a simple exercise to help you learn the LLVM code representation and tools, before writing the intermediate code generator from COOL to LLVM for MP2. LLVM (formerly, Low Level Virtual Machine, although the acronym is now deprecated) is a compiler infrastructure for writing static compilers, just-in-time compilers, and numerous compiler-based tools. It is based on a language-independent *and* machine-independent, SSA-based, mid-level intermediate representation called LLVM IR. The LLVM IR is *persistent*, i.e., it is designed to be used both as a compiler internal representation and as an offline (bitcode) representation for shipping programs. (See Lattner and Adve, CGO 2004, available at the Papers link on the course Web site, for a research paper on LLVM IR.)

1 Documentation

You can find all the documentation for LLVM at `http://llvm.org/releases/9.0.0/docs/index.html` Some manuals you will find useful are:

1. *LLVM Language Reference Manual*: LLVM instructions and types, and some examples of their usage. Read this while doing this exercise.
2. *LLVM Command Guide*: Web page with links to online man pages for the LLVM tools. Read the man pages for *llvm-as*, *llvm-dis* and *lli* before doing this exercise. Other tools you may find useful later on include *llvm-bcanalyzer*, *opt*, *llvm-extract* and (when writing optimization passes) *bugpoint*. Also see *http://llvm.org/releases/3.9.0/docs/GettingStarted.html#getting-started-with-llvm*.

2 Setting up your environment

Do the following to set up your environment on the EWS Linux machines. You should do this in your .cshrc or .bashrc file so the environment is correctly set up every time you log in.

1. Add to your LD_LIBRARY_PATH environment variable the directory

   `/class/cs426/GCC/lib64`.

   This provides access to newer versions of certain libraries, which are needed by LLVM. You can use the tool 'ldd' to check if any dynamically-loaded libraries are missing in your environment, e.g., with 'ldd /bin/ls'.

2. Add the following directory to your PATH environment variable:

   /class/cs426/llvm/llvm-9.0.0.obj/bin

This step lets you invoke the LLVM tools, as well as Clang, the LLVM-based C/C++ compiler. You should use Clang to compile C/C++ code as well as X86-64 assembly code for MP2 and all subsequent MPs. The provided makefiles will be set up to do this automatically.

Next, test your environment by doing the following:

1. Download the file *llvm-examples.tar.gz* from the Project page and extract the directory *llvm-examples/*. It includes the following C example programs:

function_declaration.c	global.c	printf.c	fib.c
getelementprt.c	ternary.c	vector.c	

2. Compile the programs to the LLVM IR. Use the command:

   make all

   If you have set your path correctly, you will see a *.ll* file for every C example program, except *fib.c*.

3. Try running *printf.ll*. Use the command:

   llvm-as < printf .ll | lli

   The output should be:

   Hello World!
   Once again, hello! This was the 2nd time.

3 Translation Exercise

Translate *llvm-examples/fib.c* into LLVM by hand. **Don't use the Clang frontend.** Your code should comply with the following rules:

- Be precise. Don't just write a similar program.
- Use chained branches and `phi` nodes to implement the nested select expressions.
- Use getelementptr for indexing arrays.
- Use the c- and ll-files from the first question as examples of how to write function declarations, global variables, printf, and getelementptr.

Test your program as before, but passing an integer argument to lli this time (fib.c will segfault if you run it without arguments, for simplicity):

llvm-as < fib .ll | lli - N

(where *N* is some integer constant). Compare the results you get with the output of `fib.c` compiled with clang or gcc directly, for different values of *N*.

Some tips:

- Omit attributes on globals, functions and function arguments: they are optional and meant for compiler consumption. Examples of attributes you can omit are `align, nounwind, uwtable, inbounds, attributes, dbg`, and the #N attributes on functions.
- *After* you have tested your hand-written `fib.ll` successfully, compare it with the output of `clang -O3 -S -emit-llvm fib.c`, to see how clang chose to generate LLVM IR for this simple function.
- Also compare with the Clang output when using -O0 instead of -O3: how does the generated LLVM IR differ when it is *not* optimized much? (Clang does some minimal code simplifications, like constant folding, at the AST level, so this is not completely unoptimized, but close.)