15-411 Compiler Design, Spring 2023
Lab 6 - LLVM

Jan and co.

Due 11:59pm, Thursday, May 4th, 2023

1 Introduction

The main goal of Lab 6 is to explore advanced aspects of compilation, and for the LLVM option, that means re-targeting your compiler to generate LLVM code. The source language you will compile is still L4, which remains unchanged from Labs 4 and 5.

2 Deliverables

2.1 Compiler

When generating code for the LLVM, given file *name*`.l4`, your compiler should generate: *name*`.l4.ll`, which is in the LLVM human-readable assembly language. The driver will use LLVM commands, without additional optimization, to generate the x86-64 assembly file *name*`.s` from this file. You may want your compiler to instead generate a *name*`.l4.bc` file using `llvm-as`. (See the LLVM documentation.) In this case, the driver will ignore any existing *name*`.l4.ll` file and use the *name*`.l4.bc` file instead.

After all is said and done, your compiler must support *both* LLVM and x86-64 as backends.

Issuing the shell command

```
% make
```

should generate the appropriate files so that

```
% bin/c0c --llvm <args>
% bin/c0c --unsafe --llvm <args>
% bin/c0c <args>
% bin/c0c --unsafe <args>
```

will run your compiler in safe and unsafe modes, generating LLVM or direct x86-64 native code, respectively. You should also still support the optimization flags `-O0` and `-O1`, for LLVM.

The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader. In particular, since there are likely to be many different projects undertaken, do introduce the project at the very top of the README.

2.2 Term Paper

You need to describe your implemented compiler and critically evaluate it in a term paper of about 5-10 pages. You may use more space if you need it. Your paper should follow this outline.

1. Introduction: Provide an overview of your implementation and briefly summarize the results you obtained.
2. Comparison: Compare your LLVM compilation pipeline with direct native code generation. How does the structure of your compiler differ? How does the generated code differ? Describe optimizations and their rationale.
3. Analysis: Critically evaluate the results of your compiler via LLVM, which could include size and speed of the generated code. Also provide an evaluation of LLVM: how well did it serve your purpose? What might be improved?

As you will be expected to analyze the performance of your code, you may find it convenient to port the cycle counting benchmarking tools developed for Lab 5. Doing so will be your responsibility and should be described in the term paper.

Submit your term paper in PDF form via Gradescope before the stated deadline. Early submissions are much appreciated since it lessens the grading load of the course staff near the end of the semester.

You may not use any late days on the term paper describing your implementation of Lab 6!

3 Regarding mem2reg

By default, the intermediate form accepted by LLVM must be in SSA form. However, it is possible to allocate all variables on the stack and use the mem2reg optimization pass of LLVM to automatically convert into SSA form. As this can drastically simplify this L6 option and allow you to avoid implementing SSA form within your compiler, here are a few things to keep in mind before you do this

- You **must** state in your term paper whether or not you used mem2reg. We will consider it an Academic Integrity Violation if you claim that you implemented SSA and did not use mem2reg, but we learn that you did in fact use mem2reg.
- If you do use mem2reg and do not have something else that is significantly interesting (see section 4 Something Extra), you should expect to get one letter grade lower for L6.

4 Something Extra

Beyond the basic tasks of emitting LLVM, using LLVM to apply optimizations, and investigating the performance of these optimizations, an excellent final project will include a little something extra. Exactly what your "something extra" is is up to you, but it should represent a different direction, rather than just further investigating LLVM optimizations.

The strength of LLVM is its ability to act as common intermediate language, so the obvious approach for doing something extra with LLVM is to re-target your compiler to something besides `x86-64` assembly. Whatever you do, make sure your `README` describes any flags or commands necessary for cross-compilation or compilation on another system.

- Using LLVM, additionally support compiling to 32-bit `x86` assembly. This would require creativity, because the current implementation of 8-byte floating point values assumes they can fit in a pointer. Taking this approach would allow for a relatively honest performance comparison of 32-bit and 64-bit code.
- LLVM can target many architectures, such as ARM, MIPS, and RISC-V. And there are lots of LLVM-based projects, such as emscripten, which targets JavaScript! It's okay that performance analysis will be less straightforward if you take this approach.
- Use LLVM to support a language extension that would be otherwise difficult to handle.
- Implement your own LLVM optimization pass to do an optimization that is relevant to L4 and that is otherwise not well supported.

5 Notes and Hints

- Apply regression testing. It is very easy to get caught up in writing a back end for a new target. Please make sure your native code compiler continues to work correctly!
- Read the LLVM code. Just looking at the LLVM code that your compiler produces will give you useful insights into what you may need to change.
- Study LLVM code generated by other compilers. Particularly, on the Andrew Linux machines you can run

  ```
  clang -S -emit-llvm -O2 <file>.c
  ```

  to generate `<file>.ll` in the LLVM assembly language.
- LLVM, like C, may leave the result for certain operations undefined (e.g., division by 0). For this reason, optimization will not be as simple as just applying the `-O2` flag to LLVM's optimizations. Make sure to use appropriate keyword modifiers, and/or apply only select optimizations in order to ensure correct behavior.