15-411 Compiler Design, Fall 2015
Lab 6 - LLVM

Rob Simmons, Will Crichton, Grant Della Silva, Matt Bryant, Anshu Bansal

Due 11:00pm, Thursday, December 10, 2015
Papers due 11:00pm, Tuesday, December 15, 2015

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. This writeup describes the option of re-targeting the compiler to generate LLVM code; other writeups detail other options. The language L4 does not change for this lab and remains the same as in Labs 4 and 5.

2 Requirements

You are required to hand in two separate items:

1. The working compiler and runtime system, and
2. a term paper describing and critically evaluating your project.

2.1 Compilers

Your compilers should treat the language L4 as in Labs 4 and 5. You are required to support safe and unsafe memory semantics, as well as `-O0` and `-O2`, for LLVM. Note that safe memory semantics is technically a valid implementation of unsafe memory semantics; therefore, if you have trouble getting the exception semantics of L4 working in a manner that corresponds directly to x86-64, use the safe semantics as a starting point, and try to remove as much of the overhead as you can.

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.

2.2 Testing

Your writeup is expected to analyze the performance of your code. If you port the cycle counting benchmarking tools developed for Lab 5, then porting and applying these tools will be your responsibility and should be described in the term paper.

2.3 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 an 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 an language extension that would be otherwise difficult to handle.
- Implement your own LLVM optimization pass to do an optimization that is relevant to C0 and that is otherwise not well supported.
- Implement a LLVM backend that emits C0?

2.4 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. This should provide an overview of your implementation and briefly summarize the results you obtained.
2. Comparison. Compare compilation to LLVM, followed by native code generate 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?

3 Deadlines and Deliverables

All your code should be placed in subdirectories of the `lab6llvm` directory as before. The autograder will run regression tests against your own tests and the same tests it used in Lab 5, but these will not directly contribute to your grade. We will grade you based on the code and `README` file(s) you have checked in at the deadline.

Compiler Files (due 11:00pm on Thu Dec 10)

As for all labs, the files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader.

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.

In order for you to be able to provide a runtime system or library functions, the compiler expects a file `runllvm.c` at the top-level of your compiler directory and compile and link this against your file when compiled in `--llvm` mode. The command

```
% make clean
```

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

Term Paper (due 11:00 on Tue Dec 15)

Submit your term paper in PDF form via Autolab 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!**

4 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.

- 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 convert into SSA form. See Chapter 7.3 of the LLVM Tutorial.
- 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.