15-411 Compiler Design: Lab 6 - LLVM
Fall 2012

Instructor: Andre Platzer
TAs: Alex Crichton and Ian Gillis

Compilers due: 11:59pm, Tuesday, December 4, 2011
Term Paper due: 11:59pm, Thursday, December 6, 2011

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. This writeup describes the option of retargeting the compiler to generate LLVM code; other writeups detail the option of implementing garbage collection or optimizing the generated code. 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.

3 Testing

You are not required to hand in new tests. We will use a subset of the tests from the previous labs to test your compiler.

However, if you wish to use LLVM to optimize your code, consult the handout for `lab6opt` for guidelines on how to test your compiler, and what information should be added to the term paper.

4 Compilers

Your compilers should treat the language *L4* as in Labs 4 and 5. You are required to support safe and unsafe memory semantics. 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*`.ll`, which is in the LLVM human-readable assembly language. The driver will use LLVM commands to generate from this the file *name*`.s`, in the x86-64 assembly language.

If you would like to apply LLVM optimizations yourself, your compiler may generate a *name*`.bc` file using `llvm-as` and run optimizations on it. (See the LLVM documentation.) In this case, the driver will ignore any existing *name*`.ll` file and use the *name*`.bc` file instead.

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

5 Deadlines and Deliverables

Project Proposal (due 11:59pm on Mon Nov 26)

Email the course staff at `15411@symbolaris.com` letting us know whether you elect to do this project for lab 6.

Compiler Files (due 11:59pm on Tue Dec 4)

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 l4c
```

should generate the appropriate files so that

```
% bin/l4c --safe --llvm <args>
% bin/l4c --unsafe --llvm <args>
% bin/l4c --safe --x86_64 <args>
% bin/l4c --unsafe --x86_64 <args>
```

will run your *L4* compiler in safe and unsafe modes, generating LLVM or direct x86-64 native code, respectively. For backwards compatibility, the default is `--unsafe --x86_64`. The driver will only use the `--llvm` flag to run your code.

In order for you to be able to provide a runtime system or library functions, the compiler expects a file `l4lib.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.

The compiler should be handed in to svn, under `lab6llvm`

Term Paper (due 11:59 on Thu Dec 6)

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. The recommended outline varies depending on your project. Submit a file `<team>-llvm.pdf` via email to the course staff at `15411@symbolaris.com`.

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? If you are applying optimizations at the LLVM level, describes those 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?

6 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 assembly code. Just looking at the assembly code that your compiler produces will give you useful insights into what you may need to change.

- The intermediate form accepted by LLVM must be in SSA form. However, it is possible to allocate all variables on the stack and use a script provided with LLVM in order to convert into SSA form. See Chapter 4.7 of the LLVM Tutorial.

- LLVM, like C, leaves the result for certain operations undefined (e.g., division by 0), so you may need to turn them into function calls.