15-411 Compiler Design: Lab 5
Fall 2008

Instructor: Frank Pfenning
TAs: Ruy Ley-Wild and Miguel Silva

Test Programs Due: 11:59pm, Tuesday, November 10, 2009
Compilers Due: 11:59pm, Tuesday, November 17, 2009
Papers Due: 11:59pm, Thursday, November 19, 2009

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L4* and also implement a first set of basic optimizations. The language itself is unchanged from Lab 4, but its dynamic semantics is now *safe* in that certain operations such as attempting to access an array out of bounds must result in an exception.

2 Requirements

As for the earlier labs, you are required to hand in test programs as well as a complete working compiler that translates *L4* source programs into correct target programs written in x86-64 assembly language. In addition you have to submit a PDF file via email to the instructor (`fp@cs`), which describes and evaluates the optimizations you implemented.

3 Safety Requirements

The behavior of *L4* was purposely unspecified in certain situations. When the Lab 5 compiler is called with the `--unsafe` switch this behavior remains undefined. This is also the default, for backward compatibility.

When the Lab 5 compiler is called with the `--safe` switch, the unspecified behavior is now explicated as outlined below.

Memory-related exceptions must be signaled as the exception 11 (`SIGSEGV`). We refer to this as *illegal memory reference*.

Pointers

As in Lab 4, derefencing the null pointer is an *illegal memory reference*. Since the page at address 0 is read/write protected by the operating system, attempting to read from 0 will yield the appropriate exception.

Structs

When the address of a struct is computed as 0, an *illegal memory reference* must be reported. This may happen when computing `*p` for a pointer `p` declared with `var p : s*` for a struct `s`. In this case we must signal an exception instead of basing address calculations for fields on 0, because with a sufficiently large offset we may skip past the read/write protected pages at the beginning of memory.

Arrays

If an array A was allocated with `new(`τ`[`n`])` for a non-negative n, any attempt to access $A[i]$ where $i < 0$ or $i \geq n$ must generate an *illegal memory reference.*

Similarly to structs, when the address of an array is computed as 0, an *illegal memory reference* must be reported.

Allocation

When allocation fails, an *illegal memory reference* must be reported. Allocation with `new(`τ`[`n`])` for $n < 0$ must fail; other allocations may or may not fail depending on current resource constraints. Your test cases should use a moderate amount of memory so that one would not expect allocation to fail due to resource constraints. In any case, `new` is not allowed to return the null pointer.

4 Array Layout

In order to be able to call C libraries, your code must strictly adhere to the convention specified in the Application Binary Interface (ABI) for the x86-64 and C. This is exactly as in Lab 4 (data alignment, struct layout, calling conventions, stack pointer alignment), except that special considerations are necessary when compiling to safe code.

In order for your code to be able to check whether array access is in bounds, it must store with each array the number of elements of the array. This should be done as follows: if the data elements in an array start at address a, then the (non-negative) integer n recording the number of elements in the array is stored at $a-8, a-7, a-6, a-5$. The bytes at $a-4, \ldots a-1$ are padding, to simplify alignment (the strictest alignment requirement in our language for data is 0 mod 8, and `calloc` returns memory aligned in this manner).

This layout will allow the implementation to correctly call C functions with array addresses in arguments, whether we are using safe or unsafe mode. Arrays generated by C cannot be returned to *L4* in safe mode because their size is in general unknown. Special wrapper code is needed in this case, which may be different for different libraries.

5 Optimizations

In addition to the safe compilation option, you are also required to implement and describe some basic optimizations. Choose 3 from the following menu.

- **Instruction selection.** You may optimize instruction selection to generate more compact or faster code. This includes generating good code for conditions (e.g., avoiding `set` instructions) or loops (e.g., enabling good branch prediction or aligning jump targets), and other improvements on your code.

- **Constant propagation and folding.** Implement constant propagation together with constant folding and eliminating constant conditional branches.
- **Dead code elimination.** Implement dead code elimination using the analysis described in Lecture 5.
- **Eliminating register moves.** Explore techniques for eliminating register moves such as improved instruction selection, copy propagation, register coalescing, and peephole optimization. We suggest coalescing registers in a single pass after register allocation as suggested by Pereira and Palsberg and the notes to Lecture 3.
- **Common subexpression elimination.** Implement common subexpression elimination, with or without type-based alias analysis to avoid redundant loads from memory.
- **Other optimizations.** If there is a particular optimization you would like to implement that is not in the above list you may wait until the next lab or contact the instructor with a proposal.

If you have already implemented some of these optimizations, you may revisit and describe them, perhaps improve them further.

Your compiler must take a new option, `-O`*n* (dash capital-O), where `-O0` means no optimizations, `-O1` performs some optimizations, and `-O2` performs the most aggressive optimizations. Part of our evaluation may be based on the performance improvements you achieve.

You are required to write a short paper of 3–5 pages, to be submitted via email to the instructor as a PDF file. This paper should describe each optimization, how you implemented it, any heuristics you developed for its application, and whether you found it to be effective both on contrived examples and the provided benchmark suite.

6 Regression Testing

As you are implementing optimizations, it is extremely important to carry out regression testing to make sure your compiler remains correct. We are providing directories `lab5/regression/tests*` in the subversion repository, assembled from tests for earlier labs. This may or may not be checked by Autolab upon hand-in, so you should make sure to apply this frequently as you proceed with optimizations. If regression testing is not performed automatically, we will apply it by hand during instructor evaluation.

7 Project Requirements

For this project, you are required to hand in test cases, a complete working compiler for *L4* that produces correct target programs written in Intel x86-64 assembly language, and a description and assessment of your optimizations. When we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Test Files

Test files should have extension `.l4` and start with one of the following lines

`#test return` *i*	program must execute correctly and return *i*
`#test exception` *n*	program must compile but raise runtime exception *n*
`#test error`	program must fail to compile due to an *L4* source error

followed by the program text. If an exception number is missing, any exception is accepted. Defined exceptions are at least `SIGFPE` (8), `SIGSEGV` (11), and `SIGALRM` (14). These are raised by division by 0 or division overflow (8), illegal memory references (11), or by a time-out (14). All test files should be collected into a directory `test/` (containing no other files) and submitted via the Autolab server.

Your test programs should be in two categories: those that test `--safe` and those that test your optimizations.

Compiler Files

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 description of your code and algorithms used at the beginning of this file.

Issuing the shell command

```
% make l4c
```

should generate the appropriate files so that

```
% bin/l4c <args>
```

will run your *L4* compiler. The command

```
% make clean
```

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

Using the Subversion Repository

The recommended method for handout and handin is the course subversion repository.

The handout files for this course can be checked out from our subversion repository via

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab5` subdirectory. Or, if you have checked out `15411-f09/<team>` directory before, you can issue the command `svn update` in that directory.

After first adding (with `svn add` or `svn copy` from a previous lab) and committing your handin directory (with `svn commit`) to the repository you can hand in your tests or compiler by selecting

```
S3 - Autograde your code in svn repository
```

from the Autolab server menu. It will perform one of

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>/lab5/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>/lab5/compiler
```

to obtain the files directories to autograde, depending on whether you are handing in your test files or your compiler.

If you are submitting multiple versions, please remember to commit your changes to the repository before asking the Autolab server to grade them! And please do not include an compiled files or binaries in the repository!

What to Turn In

Hand-in on the Autolab server:

- At least 10 test cases. The directory `tests/` should only contain your test files and be submitted via subversion. Since the language has not changed from Lab 4, test cases should primarily be there to (1) verify that the the safety requirements are adhered to, and (2) be somewhat interesting from the point of view of checking your optimizations. Keep in mind, however, that all compilers must be able to compile and run them within the time limit even with `-O0`. The server will test your test files with `-O0 --safe` and notify you if there is a discrepancy between your answer and the outcome of the reference implementation. You may hand in as many times as you like before the deadline without penalty. If you feel the reference implementation is in error, please notify the instructors. The compiled binary for each test case should run within 1 second with the reference compiler on the lab machines; we will use a 5 second limit for testing compilers.

  Test cases are due **11:59pm on Tue Nov 10, 2009**.

- The complete compiler. The directory `compiler/` should contain only the sources for your compiler and be submitted via subversion as described above. The Autolab server will build your compiler, run it on all extant test files, link the resulting assembly files against our runtime system (if compilation is successful), execute the binaries (each with a 5 second time limit), and finally compare the actual with the expected results. It will also test the `--safe`, `--unsafe` and `-O`*n* flags. You may hand in as many times as you like before the deadline without penalty.

  Compilers are due **11:59pm on Tue Nov 17, 2009**.

- The description of the optimizations. The PDF file of 3–5 pages should describe your optimizations and assess how well they worked in improving the code, over individual tests and the benchmark suite. It should be emailed to the instructor, `fp@cs`.

  Papers are due **11:59 on Thu Nov 19, 2009**.