Lab 3

ECE 479K — Compiler
Spring Semester 2024

Handed out: Mar. 7, 2024 due: Apr. 9, 2024, 11:59 pm

In this lab you will implement a loop identification analysis and a loop-invariant code motion optimization, and a bonus sparse conditional constant propagation optimization, all in LLVM 15. For this lab you may work in teams of two (Section 1), and the grading will be primarily based on a report (Section 3).

1 Working in Teams

In this lab you may work in teams of two. Both members of a team will get the same grade, *with no exceptions*. If you work in a team, you *must* use pair programming and both of you should take turns being the driver (who types) and the observer (who watches, advises, and plans ahead). You should switch roles at a relatively fine grain, e.g. at different parts of a pass rather than not switching roles for an entire pass. Make sure that you are both familiar with the algorithmic concepts and infrastructure, enough that you could do the entire lab on your own. Your report should describe clearly how you worked together on this project, including both the programming and testing of the passes. You don't need to physically be together for pair programming, you can do it over ZOOM or some editors (e.g. VSCode) also support live collaboration.

2 What To Do for This Lab

You must implement the following passes:

- **Loop Analysis:** This is an analysis pass which identifies all of the *natural loops* in each function. The algorithm for this is provided in lecture 13 (there is a very nice and concise writeup with algorithm form at https://pages.cs.wisc.edu/~fischer/cs701.f14/finding.loops.html). In addition, to identifying the natural loops of a function this analysis will record information about the loops so that you will be able to use it in other passes.
- **LICM:** This is a Loop-Invariant Code Motion optimization pass. The algorithm for this is provided in lecture 17. This will pass will consume the results of your Loop Analysis, using that information to identify the loops and basic blocks contained within them.

You can choose to implement the following pass as a bonus part for this lab:

- **SCCP:** This is a pass that implements the Sparse Conditional Constant Propagation optimization. The algorithm for this is provided in lecture 17.

More details on what you are expected to implement are provided in Section 4.

For this lab, we provide a code skeleton which can be used to build these passes as out-of-tree LLVM passes. Included in the skeleton are:

- `README`: Provides instructions for building and running your passes.

- `CMakeLists.txt`: Configures the build system for your passes. You should not need to make any modifications to the existing parts (although you are allowed to), but feel free to add more cmake targets for automated testing.
- `UnitLoopInfo.h, UnitLoopInfo.cpp`: You should implement your Loop Analysis pass in these files as well as implement the `UnitLoopInfo` class which provides the results of your loop analysis to your LICM pass. The provided code includes an empty definition of the `UnitLoopInfo` class and defines the `UnitLoopAnalysis` analysis and also acquires the function's `DominatorTree` which you are allowed to use in your analysis.
- `UnitLICM.h, UnitLICM.cpp`: You should implement your LICM pass in these files. The provided code defines the `UnitLICM` class as an optimization pass operating on functions and acquires the results of your loop analysis pass.
- `UnitSCCP.h, UnitSCCP.cpp`: If you choose to, you should implement your SCCP pass in these files. The provided code defines the `UnitSCCP` class as a function optimization pass.
- `RegisterPasses.cpp`: Provides boilerplate code needed to register the passes with LLVM so they can be run.

When compiling your passes, we recommend you to do an out-of-source build to keep built files away from source code files:

```
mkdir build; cd build; cmake ../;
```

Here is a suggested order in which to approach the problem.

1. Double check your LLVM installation. If you are using the container (the same container you used for lab 1 & 2) we provided, LLVM has been installed at a global location (`/usr/local/{bin|include|lib}`).

   If you wish to build LLVM on your own and run it natively on your own machine, you can contact the TAs for help.

2. Extract the hand-out source code. Follow the directions in the `README` file for configuring the `CMakeLists.txt` file is using your own build of LLVM. Also, follow the directions in `README` to ensure that you can successfully build the passes.

3. Read the class notes mentioned above for each pass and make sure that you understand the algorithm that you will be implementing. To help solidify your understanding of each algorithm, we encourage you to write simple test cases (in C, Cool, or LLVM) which could be used to demonstrate the various features of the analysis or optimizations; this will also help with testing of your passes.

4. Start to familiarize yourself with the LLVM infrastructure. Some good places to start are
   - Getting Started with the LLVM System
   - Writing an LLVM Pass
   - LLVM Programmers' Manual

5. As another way of familiarizing yourself some with LLVM, we find that the loop unroll pass and simplify CFG pass provide good examples of non-trivial LLVM passes.

6. Implement the passes, we recommend starting with the Loop Analysis pass and then proceeding to the LICM pass, and possibly SCCP passes. You may also find it easier to start partially implementing each pass and then adding additional features until the passes are complete.

7. Alongside implementing the passes you should test each; for your loop analysis pass you may wish to add debug prints to allow you to view the results and for the optimizations you should use test cases you write and those we provide. You should experiment with different sequences of LLVM passes (see more below).

   Note: The passes are standard and have built-in versions in LLVM. However, do not rely on these too much and do not copy from them. For example, the built-in LICM will move operations after a loop, not only before, a feature you are not required to implement. Therefore, in comparing with the results of the built-in passes keep in mind what you are and are not expected to implement.

   Tip: For initial testing, use many tiny or small test cases covering all the possible individual cases you can think of, instead of one or a few larger tests. *Write these small tests before you start writing your pass — this is invaluable for getting an understanding of what you're trying to do.* If you write these in COOl or C, study the LLVM code to understand the input you're actually getting. Write them in LLVM if necessary to get very specific features.

2.1 Helpful Built-In Passes

As we have discussed in class, the order of passes can have a significant impact on the efficacy of various optimizations. We recommend running the following pass sequence:

```
clang <program>.cc -c -O0 -Xclang -disable-O0-optnone -emit-llvm -S -o - | opt -load-pass-plugin=./build/libLab3.so -passes="function(mem2reg,instcombine,simplifycfg,adce),inline,globaldce,function(sroa,early-cse,unit-sccp,jump-threading,correlated-propagation,simplifycfg,instcombine,simplifycfg,reassociate,unit-licm,adce,simplifycfg,instcombine),globaldce" -S -o <output>
```

Important: `clang -O0` not only applies no optimization on the program but also sticks attributes onto the LLVM IR to prevent basically all optimizations.

- Passing `-Xclang -disable-O0-optnone` removes `optnone` so that your passes can take effect. You should *always* do this when compiling test cases with `clang`.
- There is also `noinline` which is impossible to turn off at `-O0`; `inline,globaldce` will probably do nothing in this case. If you absolutely need function inlining to happen, you are allowed to add `__attribute__((always_inline))` to functions in your test cases. See this for more details.
- Do not use `-O1` or higher with `clang` as that triggers a lot of optimizations, including those that you should implement.

Brief explanation of this sequence:

- The `function(...)` in the passes string is for distinguishing function passes over a single function, like `mem2reg`, from module passes over an entire LLVM module, like `globaldce`, when both are present. If you are using only function passes, you do not need to specify `function(...)` explicitly.
- Note that LLVM requires there not be any spaces between between pass names. (Remember to remove all extraneous newlines when you copy the command above from this PDF.)
- `unit-licm` is the registration names of your pass; see `RegisterPasses.cpp`.
- `instcombine`: Performs many instruction folding, elimination, or simplification transformations.
- `inline, globaldce`: Inlines small functions, and then eliminates unused ones.
- `sroa`: Scalar replacement of aggregates. This iteratively expands struct-type allocas into individual allocas of each of the fields of the struct, until no more allocas can be scalar-replaced. For fields of primitive types (e.g., i32), it promotes those to SSA-form virtual registers (subsuming mem2reg).

- `adce`: Advanced dead code elimination, removes unnecessary (dead) code and can even delete empty loops. This should clean up all empty basic blocks fully.
- `simplifycfg, deadargelim`: Branch folding, followed by cleanup of unused function arguments.

We encourage you to play around with adding and removing options to improve the effectiveness of your passes, but you *should never use the LLVM built-in passes in place of yours*. For testing we also encourage you to test each separately and using simpler pass sequences.

A pass you may also find useful is `loop-simplify` which convert all loops to a "loop simplify form". If you find this useful for your Loop Analysis or LICM pass, you should make your pass depend on it.

3 Report

Your report should be about (and no more than) 5 pages long, 11-on-14 pt. (i.e., 11pt. font with 14 pt. line spacing) with 1 in. margins, formatted as a PDF. It should include:

1. A brief description of all algorithms implemented (in 1.5 pages or less total) *in your own words*. Use one or more diagrams or examples to make it easier to understand.
2. A brief description of the status of your code: what works, what doesn't, and especially any key features that you would have liked to implement but could not (if any).
3. If you worked in a team, a clear summary of how you worked together on the project. In particular, state which student did the typing and which was the observer for the different parts of the passes. *This should describe what actually happened, not the initial plan!*
4. A table for each pass which reports the statistics described in Section 4 for each benchmark. We will provide a set of benchmarks close to due date. Each benchmark should be in a separate row and each statistic a separate column of the table.
5. A discussion of the experimental results, including at least one example of the optimization counted by each statistic as well as identifying at least one missed opportunity for optimization by each pass and describe why the pass fails to perform that optimization. If you cannot find such a missed opportunity in our benchmarks suite you may use a test case of your own, in which case provide a path in your hand-in to view the test case.
6. A brief Conclusion section, summarizing what you did and the results you obtained. Never write a technical document without one!
7. References. Never write a technical document without them.

4 Optimization Requirements

Your implementations should be based on the algorithms presented in the lecture notes, with a few specific requirements that may differ.

For LICM:

- You should use the relaxed condition: moving a non-excepting expression is ok even if it does not dominate all exits, i.e., it may lengthen some path on which it was never executed before. Any instruction that has side-effects should only be hoisted if it dominates all exits.
- Your algorithm should handle all unary, binary, and bitwise operations as well as bitcasts, `icmp` and `fcmp` instructions, `select` instructions, and `getelementptr` instructions. In addition to these SSA operations, *you should handle load and store instructions, using LLVM's Alias Analysis to determine*

whether hoisting is permissible. You do not need to include other instructions, such as terminators, `phi`, `call`, `invoke`, `malloc`, `free`, or `alloca`.

- Your algorithm only needs to hoist instructions to be before the loop, not after. In particular hoisting some stores would require them to be placed after the loop: you are only required to hoist stores to before the loop.
- In your report you **must include the following statistics for your pass**
  - The number of hoisted store instructions.
  - The number of hoisted load instructions.
  - The number of *computational* instructions that are hoisted. This statistic excludes the memory operations included in the above two categories and should also exclude casting instructions and `getelementptr` instructions (these are excluded since these instructions may not have any performance impact).

For SCCP:

- Constant propagation should handle all unary, binary, and bitwise operations as well as bitcasts, `icmp` and `fcmp` instructions, `select` instructions, and `phi` instructions. You may ignore all other instructions.
- You should simplify only conditional branch (`br`) instructions when the condition is proven constant. You may ignore all other terminators.
- Your code should define the `CONST` lattice and use it, following the description of SCCP in the lecture slides.
- In your report you **must include the following statistics for your pass**
  - The number of instructions removed.
  - The number of basic blocks that are made unreachable.
  - The number of instructions replaced with (simpler) instructions.

5 Implementation Tips and Guidelines

1. Do NOT use any code from the `lib/Transforms/Scalar` or `lib/Analysis` directories in LLVM, since these include all the passes assigned in this lab – except `LoopUnrollPass.cpp` and `SimplifyCFGPass.cpp` as mentioned in Section 2. Especially, do NOT look at OR USE LLVM's built-in `LoopInfo` or `LoopAnalysis` passes in any of your passes. Additionally do NOT look at OR USE `include/Analysis/ValueLattice.h` or `include/Analysis/ValueLatticeUtils.h`; you must define your own `CONST` lattice class for SCCP.
2. If you find you are writing code for doing something basic within LLVM, it is quite likely that the code for this exists already.
3. Try using the LLVM built-in data structures in `llvm/include/llvm/ADT/` – they offer a similar API to the C++ STL interface you're likely familiar with, there's a wider range of data structures to choose from, and they're often more efficient than STL in LLVM-specific scenarios. See LLVM Programmers Manual for how to choose and use them.
4. Feel free to try out and use modern C++ features whenever appropriate. We enabled C++20 standards in the handout `CMakeLists.txt`, and the `gcc-11.4` in your container fully support C++20 features. (Except C++ modules, which you cannot use regardless due to LLVM.)

6 Helpful LLVM APIs

Below are a list of some of the LLVM classes and functions that you may find helpful in this project:

- `Function`

- `BasicBlock`
- `Instruction`
  - `mayReadOrWriteMemory()`
  - `mayHaveSideEffects()`
  - `isVolatile()`
  - `operands()` (returns a list of the values used by an instruction)
  - `users()` (returns a list of values using the value of this instruction)
  - `isSafeToSpeculativelyExecute()` (not a member function)
  - `replaceAllUsesWith()` (replaces all uses of an instruction with another value)
  - `eraseFromParent()` (removes an instruction)
- `DominatorTree`
- `AliasAnalysis`
  - `isNoAlias()` (tests whether memory operated on by two instructions may alias)
- The `STATISTIC` macro (described in detail in the Programmer's Manual) makes it easy to count actions within a pass.
- The `LLVM_DEBUG` macro can be used to wrap debugging code and it will only run in a Debug build with the `-debug` flag passed to `opt`.

Additionally, to iterate through all instructions in a `llvm::Function`, you can (but don't have to) iterate every instruction in every basic block; it's more concise to write

```
for (auto &inst: instructions(func)) { }
```

The function `inst_range llvm::instructions(Function *F)` is given in `include/llvm/IR/InstIterator.h`.

7 What and How to Hand In

Submission will be made on Gradescope. You submission should contain:

- A lab report in PDF.
- All new source files you have created along with all of the files that were provided to you in the skeleton.
- The unit tests you wrote to test your algorithms. Provide a `CMakeLists.txt`, ideally with a target called `test-all` which runs your passes on all your test cases, producing all the `ll` files and binaries. If your `CMakeLists.txt` or `Makefile` is less automated than this, you should provide clear instructions to compile and run the tests. If any tests were only used for certain algorithms, make that clear. Also list any tests that are known to not pass (quite common in compiler projects).

you're allowed to submit multiple times, but we will only look at your last submission.

If you used LLM-based code-generating tool in your code, follow the same rules as usual:

- If you used a prompted code generator, such as GPT-4, annotate every block of code that was assisted by the code generator (even if you further edited the code generator's output). You should label and number each such code block with comments (e.g., `/* LLM Block 1 */`) and clearly indicate the start/ end line of the block. Comment at the end of the file on (1) the tool you used (GPT-3.5, GPT-4, etc.) and (2) the full prompt you gave to the tool, for each code block.
- If you used an unprompted code generator, such as GitHub Copilot, comment at the beginning of each file the name of the tool you used, and indicate any large block of code (≥ 5 lines) you got from the tool.

- We will manually check these comments and may try to reproduce some of the code generation results with your prompts.

8 Grading

You will be graded on the effectiveness of the passes you write, on your test cases, on your experiment results, and on the contents of the report.

- 30% for the status of your work, in particular, the extent to which your passes have been completed and tested.
- 10% for the quality of your code implementing the passes. Factors we will look at at (i) appropriateness of data structures and algorithms, (ii) modularity, (iii) reuse, (iv) documentation, and (v) your unit tests.
- 30% for the experimental results.
- 20% for the presentation in the report, including the description of the algorithms and the discussion of the experimental results.
- 10% for the effort you put into this project, as shown by the code, experiments, and report. This should be a free 10% for you, if you each work diligently on the project!
- 25% overall bonus if you implement SCCP.