Machine Problem 4

CS 426 — Compiler Construction
Fall Semester 2019

Handed out: December 2, 2019. Due: Friday, December 13, 2019, 5PM

This assignment deals with implementing two optimizations for LLVM. You will run your passes both on the LLVM code generated from COOL programs by your compiler, and on LLVM code for C programs. Your passes should be completely language-independent.

1 Getting Started

To prepare for this project, read the basic parts of the following documents on the LLVM Webpage quickly (you can refer back to them later, as you work):

- *Getting Started Guide* (in *Getting Started/Tutorials*)
- *Writing an LLVM pass* (in *User Guides*). This document includes a simple example that shows you how to write, compile, and run a trivial LLVM pass (search for "Hello").
- Use *LLVM Programmers' Manual* (in *Getting Started/Tutorials*) as a reference (but read the sections on *General Information* and on *Helpful Hints for Common Operations* for some useful tips).

2 What You Must Do

The two optimizations to be implemented are the following:

LICM: Loop-Invariant Code Motion

ADCE: Aggressive Dead Code Elimination

Detailed pseudo-code for each algorithm is given to you in Appendices of this handout. The basic steps to follow are:

1. Download and install LLVM-9.0.0 from source. Full instructions for this step are in *mp4/src/README.txt*. You should have enough disk space, but contact me if you run out.
2. The LLVM passes defined in LICM.cpp and ADCE.cpp (mp4-licm and mp4-adce) should be compiled and linked into the shared library `LLVMMP4.so` and placed in the `$LLVMOBJROOT/lib/` subdirectory. See *mp4/src/README.txt*.
3. Read the pseudo-code for your pass in the Appendix and make sure you understand all or most of it.
4. Use the skeleton files we have provided to get you started. The ADCE optimization will be an LLVM function pass and the LICM optimization will be a loop pass. For good examples of non-trivial LLVM passes, read the following from LLVM source:

lib/Transforms/Scalar/LoopUnrollPass.cpp

lib/Transforms/Scalar/SimplifyCFGPass.cpp

5. Test your individual pass as much as you can, both on your own test cases and on ones that we give you. Also test it by running it using part of the same sequence of LLVM passes specified in Section 4 (*Sequence of Passes*) below. You can use `opt` to run any passes you want, including LLVM's built-in versions of `LICM` and `ADCE`.

   Note: The built-in versions of ADCE and LICM have more features than what you are required to implement. For example, LICM will hoist memory operations (which requires alias analysis) and will move operations after a loop, not only before: you don't need to do any of these things. Similarly, ADCE will update the dominator tree and will remove entire loops when they are dead, and you don't need to do these. For other functionality, however, your passes should behave similarly, and you can use the built-in versions as a reference solution for your tests (we will not be giving you a separate solution).

6. Then test your two passes together, using the full sequence specified below. Use the test cases we give you for this step. *These are the only test cases we will give you for this MP. There also won't be any automated testing during handin.*

Turn in your two passes and a short README file, as described in the last Section (before the Appendices) below.

3 Testing Your Pass

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 C, study the LLVM code to understand the input you're actually getting. Write them in LLVM if necessary to get very specific features.

Eventually, you should test your passes individually, and also in the full test sequence described below. You can run simple test cases provided in the directory

mp4/test

You can also run your pass on any compiled COOL program and compare its output to LLVM's built-in implementations of `ADCE` and `LICM`.

4 Sequence of Passes

You should run the following sequence of passes for full testing:

```
opt -load $LLVMOBJROOT/lib/LLVMMP4.so -mem2reg -instcombine -simplifycfg -mp4-adce -inline
-globaldce -sroa -early-cse -sccp -jump-threading -correlated-propagation -simplifycfg -instcombine
-simplifycfg -reassociate -scalar-evolution -mp4-licm -mp4-adce -simplifycfg -instcombine -globaldce
```

We encourage you to play around with adding and removing options to improve the effectiveness of both mp4-adce and mp4-licm. We will also test your mp4-adce pass with a simpler sequence that does not include the mp4-licm pass or the standard adce pass.

Brief explanations of a few of the extra passes:

- `-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).
- `-sccp`: Sparse conditional constant propagation. This simultaneously (and iteratively) propagates constant values and uses them to eliminate infeasible branches, repeating this process until it finds no more of either. SCCP may find more constants and infeasible branches if it is run after ADCE.
- `-adce`: The LLVM version of ADCE is more powerful than yours or than SCCP because it is the only one that can delete empty loops. This should clean up all empty basic blocks fully.
- `-simplifycfg -deadargelim`: Branch folding, followed by cleanup of unused function arguments.

5 Implementation Tips and Guidelines

1. *Do NOT look at the* `lib/Transforms/Scalar` *directory in the LLVM code, since it includes all the passes assigned in this MP.* Exception to rule - it is OK to look at *LoopUnrollPass.cpp* and *SimplifyCFGPass.cpp* as mention in `section 2`.
2. Read through the basic parts of the documents *Writing an LLVM Pass* and *LLVM Programmer's Guide* before you start.
3. Where appropriate, use the `InstVisitor` template to implement code that must traverse a function, handling different instruction types.
4. If you find you are writing code for doing something basic within LLVM, it is quite likely that the code for this exists already.
5. Try using the C++ Standard Template Library, if you haven't already. It is powerful and widely used within LLVM, and can save you a lot of low-level implementation effort. More importantly, it can enable you to use effective data structures quickly (e.g.,, sets, maps, lists, and vectors). See The LLVM Programmers Manual for an excellent guide to choosing appropriate data structures.

6 Helpful LLVM APIs

Below are a list of some of the LLVM APIs that you might find helpful in this MP:

- Function:
  - `inst_begin`

  - `inst_end`

- Instruction:
  - `mayWriteToMemory`
  - `mayReadFromMemory`
  - `isa<TerminatorInst>`
  - `mayHaveSideEffects`
  - `isVolatile`
  - `clone`
  - `dropAllReferences` (before deleting an instruction or basic block)
  - `eraseFromParent` (removes and frees memory)

- ValueTracking:
  - `isSafeToSpeculativelyExecute`

- Other useful classes:
  - `LoopPass`
  - `LoopInfo`
  - `Loop`
  - `DominatorTree`
  - `BasicBlock`

7 What and how to hand in

You have to hand in `LICM.cpp`, `ADCE.cpp`, and a README file to explain in short what to find where in your code. The handin script will be released similar to the previous Machine Problems. Note, again, that there will no automated testing during handin but we will build and test your passes later, so you should be sure to test that your passes compile and run correctly for all tests on EWS, against LLVM 9.0.0.

Appendix A: Algorithm for ADCE

Implementation: Implement this algorithm as an LLVM function pass.

Input: An LLVM function F.

Output: The function, with provably unused computations eliminated

Preconditions: The `getAnalysisUsage()` method of your pass should look like this:

```
virtual void getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesCFG();                      // CFG is not modified in ADCE
}
```

Algorithm: The goal of this algorithm is to identify as many unused computations as possible. It is called "aggressive" because it assumes a computation is dead unless proven otherwise. This allows it to prove instructions dead even if they are in a cycle in the def-use graph. Note that such a cycle will always include a Phi in SSA form, i.e., you must delete completely dead Phis. (The algorithm is designed so you do not have to make *any changes* to live Phis.)

This algorithm incidentally also finds basic blocks with no path reaching it from the entry node of the CFG. However, infeasible paths due to branches that are always or never taken are *not* discovered here, so you should use the full ADCE pass in LLVM (opt -adce) to eliminate those after your pass is done.

```
ADCE(F)
{
    // Initial pass to mark trivially live and trivially dead instructions
    // Perform this pass in depth-first order on the CFG so that we never
    // visit blocks that are unreachable: those are trivially dead.
    // Use df_ext_iterator<BasicBlock*> in order to remember those blocks;
    // This is defined in llvm/include/llvm/ADT/DepthFirstIterator.h.
    //
    LiveSet = emptySet;
    for (each BB in F in depth-first order)     // use df_ext_iterator<BasicBlock*>
        for (each instruction I in BB)
            if (isTriviallyLive(I))             // see below
                markLive(I, LiveSet, WorkList);
            else if (I.use_empty())             // trivially dead
                remove I from BB;

    // Worklist to find new live instructions
    while (WorkList is not empty) {
        I = get instruction at head of work list;
        if (basic block containing I is reachable)
            for (all operands op of I)
                if (operand op is an instruction)
                    markLive(op, LiveSet, WorkList);
    }
```

```
        // Delete all instructions not in LiveSet.  Since you may be deleting
        // multiple instructions that may be in a def-use cycle, you must call
        // I.dropAllReferences() on all of them before deleting any of them
        // because you cannot delete a Value that has users.
        for (each BB in F in any order)             // use F.begin(), F.end()
            if (BB is reachable)
                for (each non-live instruction I in BB)
                    I.dropAllReferences();
        for (each BB in F in any order)
            if (BB is reachable)
                for (each non-live instruction I in BB)
                    erase I from BB;
    }
```

markLive:

```
        markLive(I, LiveSet, WorkList) {
            if (I is not in LiveSet) {
                insert I in LiveSet;
                append I at end of WorkList;
            }
        }
```

isTriviallyLive(I) :

An instruction is trivially live if either:

- it may have side effects (use `Instruction::mayHaveSideEffects()`), or
- it is a terminator (including `ret` and `unwind`), a volatile `load`, or a `store, call, unwind, or free`. As a short-cut, `I.mayWriteToMemory()` will catch many of these instructions, other than terminators.

Appendix B: Algorithm for LICM

Implementation: Implement this algorithm as an LLVM loop pass.

Input: An LLVM loop L.

Output: The loop, with loop-invariant computations hoisted out of that loop and its inner loops as far as possible.

Preconditions: The loop simplification pass should have been performed on the function containing the loop. This ensures that every loop has a preheader. It does *not* ensure that a "while" loop is converted into a do-while loop nested inside an IF; you should not assume that. You will also need information about natural loops and about dominators. (Another pass that can make LICM more effective is *reaassociation*, but that is not required for the algorithm and is not included here; you should run it yourself before mp4-licm). The complete `getAnalysisUsage()` method for your pass should look like this:

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addPreserved<DominatorTreeWrapperPass>();
    AU.addPreserved<LoopInfoWrapperPass>();
    getLoopAnalysisUsage(AU);
}
```

Algorithm: The goal of this algorithm is to hoist as many loop-invariant computations out of loops as possible. We focus only on register-to-register LLVM computations, i.e., those that do not read or write memory. We do not try to move out computations that have no uses within the loop: these could be moved after the loop (at all loop exits), but that requires patching up SSA form.

We say it is safe to hoist a computation out of a loop only if it is executed at least once in the original loop (when the loop is entered); this condition is checked using dominator information. The full list of criteria are given below.

```
LICM(L)
{
    // Each Loop object also gives you a preheader block for the loop.
    for (each basic block BB dominated by loop header,
                        in preorder on dominator tree) {
        if (BB is immediately within L) {  // not in an inner loop or outside L
            for (each instruction I in BB) {
                if (isLoopInvariant(I) && safeToHoist(I))
                    move I to pre-header basic block;
            }
        }
    }
}
```

isLoopInvariant(I) :

An instruction is loop-invariant if *both* of the following bulleted points are true:

- It is one of the following LLVM instructions or instruction classes:
  `binary operator, shift, select, cast, getelementptr`.
  All other LLVM instructions are treated as not loop-invariant. Examples of instructions you are not moving are terminators, `phi, load, store, call, invoke, malloc, free, alloca, vanext, vaarg`.
- Every operand of the instruction is either (a) constant or (b) computed outside the loop. You can use the `Loop::contains()` method to check (b).

safeToHoist(I) :

An instruction is safe to hoist if *either* of the following is true:

- It has no side effects (use `isSafeToSpeculativelyExecute(Instruction *)`, you can find it in llvm/Analysis/ValueTracking.h).
- The basic block containing the instruction dominates all exit blocks for the loop. The exit blocks are the targets of exits from the loop, i.e., they are *outside the loop.* Use Loop::getExitBlocks() to get the exit blocks, and use the dominator tree to check for dominance.

Comments:

We recommend you implement ADCE first.

Comparing this algorithm with the lecture notes:

- We are using the relaxed conditions: 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.
- You are only moving SSA operations, so checking D1 (see lecture notes) is enough. Moving stores to memory require checking conditions D2 and D3. We are *not* checking for exit blocks where a value is dead: that requires a separate dataflow analysis for live variables.

Note some of the implications of these safety conditions:

- You may hoist a non-trapping instruction even out of a while loop, i.e., even if it does not dominate all exit blocks.
- You will hoist a trapping instruction only if it dominates all exit blocks. This guarantees that you will not cause a trap unless it would also have been caused by the original program.
- You may potentially reorder two trapping instructions if you hoist one of them out of the loop and not the other. This means the algorithm is safe for C, Fortran, C++ (the front-end ICG pass would have to make sure exceptions are not caught), and COOL (there are no caught exceptions), but not for Java or C#.