CS6380: Assignment #4
Loop Invariant Code Motion Pass
Due Friday October 7th, 2016 at 11:59 PM

The aim of this assignment is to write a transformation pass to optimize loops by hoisting loop-invariant code out of the loop.

1 Loop Inavriant Code Motion

1.1 Preconditions

The loop simplification pass should have been performed on the function where the loop is present. This ensures that every loop has a preheader. You will also need information about natural loops and dominators. You could also run mem2reg pass to reduce memory accesses.

1.2 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(Loop L){
   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 are true:

1. 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. In particular, you are not moving these instructions - `terminators, phi, load, store, call, invoke, malloc, free, alloca, vanext, vaarg.`
2. Every operand of the instruction is either (a) constant or (b) computed outside the loop.

`safeToHoist(I):`
An instruction is safe to hoist if either of the following is true:

1. It has no side effects (exceptions/traps). You can use `isSafeToSpeculativelyExecute()` (you can find it in `llvm/Analysis/ValueTracking.h`).

2. 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.

1.3 Comment on Safety Conditions

You are hoisting out an excepting expression (only) if it dominates all exit blocks. This guarantees that you will not cause a trap unless it would have been caused by the original program.

2 Implementation Guidelines

Create a new directory BasicLICM in `lib/Transforms/` folder of the LLVM source tree. All your code should be in this directory as this directory is part of your submission. Register your pass as `basic-licm` so that we can run the pass using `opt -load $LLVM_BUILD/lib/BasicLICM.so -basic-licm input.ll`

3 Testing

You are supposed to write test cases to test your pass. They should vary in complexity from simple to more complex ones. You need to submit 5 test cases.

You should generate the LLVM bitcode file for your test cases using -O0; otherwise LLVM's built-in version of LICM would have optimized your code. You should test your pass individually as well as in combination with other passes. For example, here is a sequence of passes you can test with:

```
opt -load $OBJ ROOT/lib/BasicLICM.so -simplifycfg -instcombine -inline -globaldce -instcombine
-simplifycfg -adce -scalarrepl -mem2reg -adce -sccp -adce -basic-licm -verify -instcombine -dce
-simplifycfg -deadargelim -globaldce input.ll
```

You can compare the behavior of your pass with that of LLVM's built-in version of LICM (replace -basic-licm with -licm in your opt command). Note that the built-in version of LICM is more complex than what you are required to implement. We can use any program and any sequence of passes for testing your pass. Your program must be robust (no crashes or undefined behavior), precise (correct output), and optimized (faster execution time). So do test your code thoroughly.

4 Submission

Your submission should be an archive containing the BasicLICM directory, a test-cases directory and a Readme file. The Readme should mention all the materials that you have read/used for this assignment including LLVM documentation and source files. Mention the status of your submission, in case some part of it is incomplete. You can also include feedback, like what was challenging and what was trivial.