Programming Project 1 – Printing LLVM Instructions and Removing Unreachable Basic Blocks

Due date: March 23rd, 2014 at midnight PST

Description: In this first compiler project, you will be asked to develop two simple compiler passes that traverse the LLVM internal data structures and/or carry out simple transformations. We now describe each of these two tasks with a series of illustrative examples that will help you understand the goal of the project as well as develop a work plan.

For each of these project passes you will need to write one or more LLVM passes. All passes will be invoked by command-line flags recognized by the LLVM opt command (see the installation notes). Everyone will write the following two passes (explained in more detail below):

1. A `printCode` pass, implemented in a file called printCode.cpp. Your `printCode` pass will print the LLVM assembly code for each function in a useful format (defined below).
2. An `deadBlocks` pass, implemented in a file called deadBlocks.cpp. Your `deadBlocks` pass will find and remove unreachable basic blocks in each function (as defined below).

PrintCode Pass: To learn about writing an optimization pass in LLVM, you are asked to write one that just pretty-prints out the bytecode. There are several different types of passes, depending on their scope. For example, there are passes that operate on a single basic block, a single function, the call graph, etc. More info on the different types of passes is available here: http://llvm.org/docs/WritingAnLLVMPass.html.

If you have followed the installation suggestions and notes, all of the pass code will reside in the `llvm/cs565/lib` directory. This will mean all of the custom passes will be in one dynamic library, and then any number of these passes can be ran in the same invocation of the `opt` command. We have included a skeleton of this at the web site (see file llvm-start.zip) where there's only one pass, a `FunctionPass` that's declared in `passes.h` and implemented in `prettyPrint.cpp`.

You can build the library with just `make`, and then you can run a test (that operates on the previously built test1.bc) with `make testPrint`. The skeleton code just print out the name of the main function, but you will add code that iterates through each basic block, and then all the instructions in each basic block, and outputs them to `stderr`.

Note that if you're on Linux, you may find that opt fails to load `cs565opt.dylib`. That's because Linux dynamic libraries usually have a `.so` extension. To fix this problem, simply change the LIB_EXT definition in the Makefile.

There are a few additional things to point out:

1. The instructions suggest making a static variable in `runOnFunction` that tracks the instruction number in the file. This might not be a good idea, as the LLVM passes are designed to be modular and scalable to multiple cores. So you should heed what the LLVM Pass docs say and have no data persist beyond the invocation of a single `runOnFunction` call. Just make a non-static variable that tracks the instruction number for the specific function.
2. There's pretty much no documentation for the LLVM classes other than auto-generated **`doxygen`**. So you have to figure out what functions to call by either looking at that or the source files directly. This is where the Xcode project becomes helpful, also, because at least you'll have some code completion.
3. Iterating through a function/block is pretty simple. For example, the `Function` class has a `begin()` function that returns an iterator that points to the first basic block. The LLVM iterators operate much like STL iterators.
4. For the map the key should be the pointer to the instruction, and the value should be the instruction number.
5. It is useful to know what the difference is between the three main categories of operand: instruction, named, and unnamed. Most operands will be either instructions or unnamed. Here's my understanding of it:
    a. An instruction operand means that the result of the instruction in question is being used as an operand. This makes it easy to figure out dependencies between instructions.
    b. A named operand is one that has to have a specific name, such as a label or function name.
    c. An "unnamed" operand means it's either a constant value or a temporary variable that isn't important enough to have a name assigned to it.
6. `runOnFunction` just returns false because it's not modifying the IR. For other passes, it'll return true if the IR is modified.

But there's one other change to make, which is to remove most of the XXXs. While XXX is fine for temporary variables, it would be more useful if constant integers were printed with their actual value. So before you output "XXX", try to `dyn_cast` the operand to a `ConstantInt`. If it succeeds, you can then output the actual integral value. When you implement this, the output of **`testPrint`** will be something like:

```
FUNCTION main

BASIC BLOCK entry
%1:    alloca 1
%2:    alloca 1
%3:    alloca 1
%4:    store  0 %1
%5:    store  0 %3
```

```
%6:    store  0 %2
%7:    br     for.cond

BASIC BLOCK for.cond
%8:    load   %2
%9:    icmp   %8 10
%10:   br     %9 for.end for.body

BASIC BLOCK for.body
%11:   load   %3
%12:   load   %2
%13:   load   %2
%14:   mul    %12 %13
%15:   add    %11 %14
%16:   store  %15 %3
%17:   br     for.inc

BASIC BLOCK for.inc
%18:   load   %2
%19:   add    %18 1
%20:   store  %19 %2
%21:   br     for.cond

BASIC BLOCK for.end
%22:   load   %3
%23:   call   XXX %22 printf
%24:   load   %1
%25:   ret    %24
```

In case you're curious, the `getAnalysisUsage` function in `prettyPrint` is what tells LLVM that this pass "preserves" everything, which means no aspect of the bytecode is modified. Normally, this function is used to specify which dependencies a pass has so that it can ensure any dependencies are executed prior to the pass in question, and also so it can specify which passes are invalidated by this one.

Naive Unreachable Basic Blocks Removal: Now let's make a pass that actually optimizes the code to some degree. For this pass, we're going to inspect the CFG to find any blocks that are unreachable from the entry block.

We call this "naive" removal because it's not going to modify edges in the CFG. So it's not smart enough to realize that something like the interior of this conditional is unreachable:

```
if (0)
{
    /* Unreachable code */
}
```

Rather, it's only going to remove unreachable blocks in pathologically bad code like test2.c:

```
#include <stdio.h>
int main() {
    int n = 5;
    goto label5;
label0:
    printf("This code is unreachable\n");
    n = 0;
label1:
    n *= 2;
    goto label4;
label2:
    printf("This code is unreachable\n");
    n = 0;
label3:
    printf("This code is reachable from label2, but not from entry.\n");
    n = 0;
label4:
    n *= 5;
    goto end;
label5:
    n *= 3;
    goto label1;
end:
    printf("Expected result: 5*3*2*5=150\n");
    printf("%d", n);
}
```

However, this optimization is not as simple as checking to see if a particular block has no predecessors. In the above example, label0 and label2 clearly have no predecessors so they should be removed. However, label3 does have a predecessor (label2), but since label2 has no predecessors that means label3 is also unreachable. To solve such cases, we have to perform a DFS starting from the entry block to determine all of the reachable blocks. Then once the DFS completes, any basic blocks, which weren't visited, should be culled.

To make a new pass, first add a declaration of `CFGNaive` to passes.h, which will be fairly similar to the declaration of `prettyPrint`. Then copy `prettyPrint.cpp` into `cfgNaive.cpp`, replace identifiers as appropriate, empty out `runOnFunction` other than the return statement, and change the `RegisterPass` call so it registers the `cfgNaive` command. Also clear out `getAnalysisUsage` since this pass does change data. This technically will invalidate all passes executed prior to `cfgNaive`, but that's okay for testing purposes.

You also need to add a test rule to the Makefile in `cs565/lib` as for example:

```
TEST_NAIVE = $(LEVEL)/../bin/opt -load
$(LEVEL)/../lib/$(LIBRARYNAME)$(LIB_EXT) -cfgNaive <
testNaive:
      $(TEST_PRINT) ../tests/test2.bc > /dev/null
      $(TEST_NAIVE) ../tests/test2.bc > ../tests/test2.opt.bc
      $(TEST_PRINT) ../tests/test2.opt.bc > /dev/null
```

This will first pretty-print test2.bc, then run the `cfgNaive` pass to create test2.opt.bc, and then pretty-print this optimized file.

In terms of the DFS, there's a depth-first iterator defined in `llvm/ADT/DepthFirstIterator.h`. So you don't have to implement the DFS. But make sure you also include `llvm/Support/CFG.h` as it declares the necessary traits to allow `BasicBlock` to be treated as a node in a graph. You'll also get a taste of exactly how much template metaprogramming LLVM uses when you try to figure out the proper way to construct the `df_ext_iter`. Unreachable blocks can be removed using `eraseFromParent`. Don't forget that `runOnFunction` must return true if the function was modified.

To test out your implementation, make the lib and then make `testNaive`. The latter part of the output should look something like this:

```
(commands...)
label0 is unreachable
label2 is unreachable
label3 is unreachable
(commands...)
FUNCTION main

BASIC BLOCK entry
%1:    alloca 1
%2:    alloca 1
%3:    store  0 %1
%4:    store  5 %2
%5:    br     label5

BASIC BLOCK label1
%6:    load   %2
%7:    mul    %6 2
%8:    store  %7 %2
%9:    br     label4
```

```
BASIC BLOCK label4
%10:   load   %2
%11:   mul    %10 5
%12:   store  %11 %2
%13:   br     end

BASIC BLOCK label5
%14:   load   %2
%15:   mul    %14 3
%16:   store  %15 %2
%17:   br     label1

BASIC BLOCK end
%18:   call   XXX printf
%19:   load   %2
%20:   call   XXX %19 printf
%21:   load   %1
%22:   ret    %21
```

Notice that the unreachable blocks (labels 0, 2, and 3) have been removed from the IR. You can test this to make sure it still works by generating the assembly/executable for `test2.opt.bc`.