Project 3: Traversing Functions, Basic Blocks, Instructions and Loops within a Module

ECE 466/566 Fall 2015

ECE 466 students may work in teams of 2,
ECE 566 students must work individually.

~~**Due: October 15, 11:59pm**~~
Due: October 19, 11:59pm

You are encouraged to comment directly on this document rather than posting questions on Piazza about the specs, as comments make it easier to understand context.

Objectives

- Implement a library that will calculate vital statistics for an LLVM module.
- Gain experience traversing the global definitions in a module, the basic blocks in a function, and the instructions in a basic block.
- Use dominance to identify loops.

Specification

Your compiler should take in a LLVM bitcode file and print vital statistics to an output file as comma-separated values. You should print the following for each LLVM module analyzed:

- Total number of functions (defined only)
- Total number of global variables defined and initialized in the module
- Total number of basic blocks (across all functions)
- Total number of instructions (across all functions)
- Total number of loads
- Total number of stores
- Total number of conditional branches
- Total number of function calls
- Total number of instructions that have at least one source operand defined within the same basic block.
- Total number of instructions that have only global value (LLVMIsAGlobalValue) or constant value (LLVMIsAConstant) operands. The instruction may have a mix of global values or constant values, but it may not have values of any other class.
- Total number of loops (number of unique headers with a back edge)

You must implement your code as a separate library that is called by your p3 tool. A basic stub library implemented in C or C++ will be provided for you along with a tool that calls it. You will need to implement the functionality within the library.

You can find the stub function inside one of these files:

- For C++: ECE566Projects/projects/lib/summary/summary.cpp
- For C: ECE566Projects/projects/lib/summary/summary.c

They will be called by the p3 tool, in one of these directories:

- For C++: ECE566Projects/projects/tools/p3cpp
- For C: ECE566Projects/projects/tools/p3

Implementation Details

1. Provide an implementation for the function, Summarize, that calculates the properties itemized above. A prototype is already available for you in the projects directory.
   a. The code you implement should be added to the summary library, not the p3 tool, so that it can be re-used in later projects.
   b. Your code should be added to `projects/lib/summary/summary.c` for C or `projects/lib/summary/summary.cpp` for C++. A stub version of the `Summarize` function has already been implemented for you in each of the libraries. You only need to extend it to compute all the required quantities listed earlier.
   c. You may not change the name of the `Summarize` function.
   d. The `Summarize` function takes three arguments in this order: A reference to the Module (.bc file), a keyword, and a filename. You must save your statistics to the filename provided using Comma Separated Values (CSV). I have already implemented a `print_csv_file` function that I recommend you use since it works properly with the `wolfbench/stats.py` script that accumulates all of your output. You may also use `pretty_print_stats` to look at the information on the terminal as it's dumped to file. Final version of the project should not have a `pretty_print_stats` call.

2. You may need to update your repository to get the latest version of code. Then rebuild everything:
   a. `cd path/to/ECE566Projects`
   b. `git pull`
      i. If this command fails, it's because you have modified files. You can either commit them or stash them (but not both). A commit will keep your changes in the local directory, but *stash will remove your changes and save them* elsewhere. Pick the best one for your case:
         1. `git commit -a -m"some changes I made blah blah blah"`
            Or...
         2. `git stash`

Now, go back and re-execute `git pull`.

c. `cd path/to/ECE566Projects/projects/build`

d. Note: If your build directory is already configured for the language you want, then you can skip this step and go straight to building (step e). Otherwise, follow the instructions for the language you prefer. For C, remove the build directory and re-run the cmake configure command inside the build directory to choose C (and I think the default language is C++ for p2):

   i.
   ```
   export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake
   ```
   ii.
   ```
   cmake -DUSE_C=1 -DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..
   ```

For C++, remove the previous build directory and re-run the cmake configure command inside the build directory to choose C++ (I think for C++ you can skip this part):

   iii.
   ```
   export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake
   ```
   iv.
   ```
   cmake -DUSE_CPP=1 -DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..
   ```

e. `cmake --build .`

3. Assuming that wolfbench, p3-test, and projects are all in the same parent directory, you can configure the project as shown below.
   a. For C:
      i.
      ```
      ../wolfbench/configure --enable-customtool=`cd ../projects/build/tools/p3/; pwd`/p3
      ```
      For C++ (select the 566 or 466 option):
      ii.
      ```
      ../wolfbench/configure --enable-customtool=`cd ../projects/build/tools/p3cpp/; pwd`/p3
      ```
   b. Then, build the test code:
      i. `make all test`
   c. If you want the output to be less verbose, run it this way:
      i. `make -s all test`
   d. If you encounter a bug, you can run your tool in a debugger this way:
      i. `make clean`
      ii. `make DEBUG=1`

      This will launch gdb on your tool with one of the input files. You can set breakpoints directly in your implementation files.

4. At the end of make's run, you will see a summary of all the statistics generated. Note, I am not providing you the correct final outputs, so you should investigate your answers and make sure they are correct. In addition, you will probably notice some interesting properties in the data you collect. You are **allowed to compare your output** (**not your code**) with other students in the class. If you want to tally the stats across all applications, you can do so using the wolfbench/stats.py script:
    a. `cd p3-test`
    b. ``path/to/wolfbench/stats.py `find . -name '*.stats'` ``

LLVM API

The required LLVM API is smaller than previous assignments since you are not building IR. But, you will need to make use of the following:

```
LLVM-C/Core/Values/Modules
LLVM-C/Core/Values/BasicBlock
LLVM-C/Core/Values/Instructions
LLVM-C/Core/Values/General APIs
LLVM-C/Core/Values/User Value
LLVM-C/Core/Values/Constants/Global Values/Global Variables
```

Extra Files: Computing Dominance

To assist with some of the project's requirements, I've added a C library that provides dominator information. You will find it here:

```
projects/libs/dominance
```

The header file is here:

```
projects/include/dominance.h
```

C projects can call this library directly using the provided header file. C++ projects can either call it or extract the useful code and integrate it into their SummaryCpp library.

To include dominance.h in a library or tool, all you need to do is this:

```
#include "dominance.h"
```

The makefiles already provide the necessary search paths during compiler-compile time so you don't need to specify the full relative path in the include directive.

Visiting Instructions in a Module

The Summarize function takes an LLVMModuleRef as an argument. Using that reference, you can iterate over each function in the module like this:

```
LLVMValueRef  fn_iter; // iterator
for (fn_iter = LLVMGetFirstFunction(Module); fn_iter!=NULL;
     fn_iter = LLVMGetNextFunction(fn_iter))
{
     // fn_iter points to a function
     LLVMBasicBlockRef bb_iter; /* points to each basic block
                              one at a time */
     for (bb_iter = LLVMGetFirstBasicBlock(fn_iter);
          bb_iter != NULL; bb_iter = LLVMGetNextBasicBlock(bb_iter))
     {

        LLVMValueRef inst_iter; /* points to each instruction */
        for(inst_iter = LLVMGetFirstInstruction(bb_iter);
            inst_iter != NULL;
            inst_iter = LLVMGetNextInstruction(inst_iter))
        {
             // get the basic block of this instruction
             LLVMBasicBlockRef ref =
                  LLVMGetInstructionParent(inst_iter);
        }
     }
}
```

Analyzing Instructions

To determine the type of an instruction, use either `LLVMIsA*` or `LLVMGetInstructionOpcode`. Using this call, you can differentiate loads, stores, branches, and call instructions.

To examine an operand of an instruction, use `LLVMGetNumOperands (LLVMValueRef)` to determine how many operands it has. It's return value will tell you how many there are. Then, you can obtain a pointer to a specific operand using `LLVMGetOperand (LLVMValueRef, int pos)`.

Since the destination block of a branch is just an operand to the branch instruction, you can find back edges by getting the operand of a branch and testing if the destination dominates the branch's block.

Getting Help

History shows that my specs are sometimes incomplete or incorrect. Therefore, please start early. If you run into problems, please post a question on Piazza.

Grading

ECE 566 students must work individually, but ECE 466 students are allowed and *encouraged* to form groups of two. Only one student needs to submit in ECE 466, but both names must appear in the comments of all your code.

Uploading instructions

Upload your summary.c or Summary.cpp file. Also, please add a note to your submission in the Notes field indicating which language you used. We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

The assignment is out of 100 points total. If you make no attempt and submit the provided code without meaningful changes (i.e. white space and comments do not count), you earn 10 points. Otherwise, assigned as follows:

ECE 566

10 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications

35 points: Assigned based on how closely your results for Wolfbench match the instructor's solution. There is some tolerance built in. For example, if your number is off by one on a large count, that difference will be ignored. Off by one on a small count, however, might be meaningful.

15 points: Fraction of secret tests that pass (these may overlap with provided tests)

ECE 466

20 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications

40 points: Assigned based on how closely your results for Wolfbench match the instructor’s solution. There is some tolerance built in. For example, if your number is off by one on a large count, that difference will be ignored. Off by one on a small count, however, might be meaningful.