Programming Project 1-Identifying Basic Blocks and Call-Graph Analysis

Due date: Oct. 16th, 2015 at midnight PST

Description: This project focuses on the popular LLVM compilation infrastructure. The LLVM Project is a collection of modular and reusable compiler and tool-chain technologies. It is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. Clang is a compiler that is written specifically for C, Objective-C and C++.

There are three phases in this project.

a) Installation of LLVM and Clang and the required prerequisites.
b) Running an existing pass in LLVM;
c) Writing your own analysis pass;

We will now explain each of these phases in more detail.

1. Installation on Mac and Windows

The instructions on how to install LLVM and Clang can be found http://clang.llvm.org/get_started.html. A more detailed set of instruction is provided in the LLVMInstructions-MacOSX10.9.pdf file for the Mac OSX 10.9 Operating Systems - Mavericks. and the LLVMInstructions-Windows.pdf file respectively After installing and building llvm and clang, try to compile a hello world program to see if you get it installed correctly. A few examples are provided using the templates in the 'cs565-start.zip' archive.

2. Executing Call-Graph Pass

Optimization features of LLVM are implemented as passes that traverse some portion of a program and thus of its intermediate representation to either collect information or transform the program. We have analysis passes, transform passes and utility passes. A list of all LLVM passes is available in http://llvm.org/docs/Passes.html .

In this section you are required to use the "-dot-callgraph" pass on an input file. This pass will print out the CFG in a DOT format. DOT is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use.

Graphviz is an open-source software that is used to visualize graphs. http://www.graphviz.org/. Use this software to represent the dot file that was the output of the pass.

3. Writing your own Analysis Pass

In this phase you have to write a semantic analysis pass that traverses the LLVM internal data structures and perform some simple analysis. You can use one or more of the LLVM passes in order to write your pass. All passes will be invoked by command-line flags recognized by the LLVM opt command. The basic steps to write a new pass this can be found http://llvm.org/docs/WritingAnLLVMPass.html.

Your semantic analysis pass will check that function calls to user-defined functions in the code comply in number and type with the corresponding function declarations with the following simplifying constraints:

1. You should include in your call graph analysis predefined library functions with variable number of args such as printf but ignore the type checking and argument/parameter agreement checks for these functions;
2. Check that all user-defined functions are indeed declared in the input and check that the types of the parameters agree in number and type with those of the call instruction.
3. In cases of discrepancies of number and/or type we should report an error indicating the offending parameters, the expected type and the observed type. In the case of numeric types such as integer versus floating point you should report a warning instead as conversions can be defined. That is, a floating point can be observed when an integer is expected and vice versa. For a mixing of other numeric and non-numeric types just report a descriptive error.
4. The format of printing the errors you found should be as follows:

   Function 'X' call on line 'Y': argument type mismatch. Expected 'int' but argument is of type 'char'.
   Function 'Z' call on line 'T': expected '3' arguments but '2' are/is present.

In addition to this type analysis you will also conduct a call-graph analysis where you will compute the number of times a given user-defined function is statically referenced in the code. This will give a later compiler analysis and transformation of which functions are more "intense" as they may be invoked more often (as you can imagine there are no guarantees that the more referenced function is the one being more often invoked at run-time). This statistical counting is to be performed across the entire program beginning with the main function.

```
List of function calls:
Foo(int,char,int) : 4
Bar(int) : 3
```

4. Turn-In Instructions:

You need to turn in your pass source files and corresponding auxiliary functions. Please make sure you isolate all you code in specific folders and include whatever Makefile the TA needs to have to compile and test your code. In the end you will be responsible for helping the TA to validate your input and output. Should this approach become infeasible to validate your programming project we might include a live demonstration with along side your computer at the convenience of the TA during her office hours. This can only be done with files you have provided on the project's deadline.