B547/I533 Lab 4: Call Graph Generation (LLVM)

Project Description

Call graph is a directed graph that represents calling relationships between functions in a program. Specifically, each node represents a function, and each edge (f, g) indicates that function f calls function g [1].

In this project, you are asked to familiarize yourself with the LLVM source code and then compose a program analysis *pass* for LLVM. This analysis will produce the call graph of input program.

Using LLVM

We will be using the Low Level Virtual Machine (LLVM) compiler infrastructure [4] for this project. We assume you have access to an x86 based machine (preferably running Linux, although Windows and Mac OS X should work as well).

In this project, you should use LLVM version 10.0.0 on Ubuntu 20.04. You can download, install, and build the LLVM source code from [4]. Follow the instructions on the website [3] for your particular machine configuration.

Read the official documentation [5] carefully, specially the following pages:

1. The LLVM Programmer's Manual
    - https://llvm.org/docs/ProgrammersManual.html
2. Writing an LLVM Pass tutorial
    - https://llvm.org/docs/WritingAnLLVMPass.html

Project Requirements. LLVM is already able to generate call graphs. However, this feature is limited to direct function calls which is not able to detect calls by function pointers.

In this project, you are expected to write an analysis pass for LLVM which can detect simple indirect function calls. We are considering two cases of indirect function calls:

1. Assigning a function address to a function pointer and then invoke the function pointer.

2. Assigning a function address to a field of a C structure (which is a function pointer), and then invoking that function pointer field. (Your analysis needs to support this case even when we have a pointer access to such structure, i.e both structure1.f and structure2->f need to be captured.)

Implementation Notes:

1. Analysis needs to be interprocedural. Therefore, you can extend the intraprocedural form directly. Note that intraprocedural can use a FunctionPass, and interprocedural requires a ModulePass.
2. Tracking the variables that contain *function pointers* can be done through *data flow analysis* and a standard *worklist algorithm*
3. In order to determine what functions are called through a function pointer, you need to track the potential functions associated with any particular variable.
4. The called function is one of the arguments of the *Call* and *Invoke* instructions.
5. When accessing a field or array element, the GEP instruction is used to identify which field or array index should be accessed [2].

Example Consider the following test case:

```
void F() {   }
void E() {   }
void D() {   }
void C() { D(); E(); }
void B() { C(); }
void A() { B(); }

int main() {

        void (*p)();
        A();
        p = &C;
        (*p)();

        struct s{
                void (*q)();
                int value;
        } s1;
        s1.q = &F;
        s1.q();
```

The execution and invocation of your pass is going to be like the following line:

```
clang -Xclang -load -Xclang <path to so>/CGraph.so <example.c>
```

Notes:

1. "<path to so>" is the relative or absolute path to the generated dynamic library.
2. "<example.c>" is the name of the input test case.

The output of your analysis should be as following:

$main: \ A \ C \ F$
$A: \ B$
$C: \ E \ D$
...

Expected Output Format You are expected to print the function names of the input program, followed by all functions which are called within that function. The expected output format would be a set of rows as following:
$func_i: \ func_{j1} \ func_{j2} \ func_{j3} \ldots$
Where $func_{j1}, func_{j2}, func_{j3}, \ldots$ are called within $func_i$.

Limitations

1. You are not required to address extra level of indirections (e.g., s->p1->p2()).
2. The exact static call graph generation is undecidable. So, we limit out test cases to those that the function pointer addresses are available at compile time.

Further Instructions
This project is not trivial. Please start early.

Due Date and Turn in

The project is due on May 2. Please turn in your project through Canvas. The project file should be named **Lab4_fullname.tar.gz**, where fullname is your name. It includes the following files:

- Your source code: CGraph.cpp.
- Makefile or CMakeLists.txt.

There will be several test cases in Canvas. Make sure your codes pass these test cases and your codes should work on Ubuntu 20.04.

Please contact TA Chaoqi Zhang (cz42@iu.edu) for any questions.

References

[1] Call Graph, Wikipedia. http://en.wikipedia.org/wiki/Call_graph.

[2] GEP, Get Element Pointer Instruction. http://llvm.org/docs/GetElementPtr.html .

[3] LLVM Getting Started. http://llvm.org/docs/GettingStarted.html.

[4] LLVM homepage. http://llvm.org/.

[5] LLVM Official Documentation. http://llvm.org/docs/.