CS590 Project 2: Call Graph Generation for LLVM

Project Description

A call graph is a directed graph that represents calling relationships between functions in a computer 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 write a program analysis *pass* for LLVM. This analysis will produce the call graph of input program.

Installing and Using LLVM

We will be using the Low Level Virtual Machine (LLVM) compiler infrastructure [5] developed at the University of Illinois Urbana-Champaign 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).

First download, install, and build the latest LLVM source code from [5]. Follow the instructions on the website [4] for your particular machine configuration. Note that in order use a debugger on the LLVM binaries you will need to pass `--enable-debug-runtime --disable-optimized` to the configure script. Moreover, make the LLVM in debug mode.

Read the official documentation [6] carefully, specially the following pages:

1. The LLVM Programmers Manual (http://llvm.org/docs/ProgrammersManual.html)
2. Writing an LLVM Pass tutorial (http://llvm.org/docs/WritingAnLLVMPass.html)

Project Requirements LLVM is already able to generate the call graphs of the programs. 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 for LLVM which can detect the 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 in the bitcode.

4. The called function is one of the arguments of the *Call* and *Invoke* instructions. The `CallSite` class in `include/llvm/Support/` would be very helpful to manage the call sites.

5. When accessing a field or array element, the `GEP` instruction is used to identify which field or array index should be accessed [3].

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:
`opt -load <path to so>/CGraph.so -cgraph < inputBitcode.bc > /dev/null`

Notes:

1. `"<path to so>"` is the relative or absolute path to the generated dynamic library. By default, it should be located in `../../../Debug+Asserts/lib/`
2. `"cgraph"` is the name of your pass which is registered in LLVM.
3. `"inputBitcode"` is the name of the input test case. You can generate your own bitcodes using the following command:
   `clang -O0 -c -emit-llvm <source files>`

For more information on how to invoke a pass, please consult the official document [7].

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:
$[func_i] : [func_{j1}], [func_{j2}], [func_{j3}], ...$
Where $func_{j1}, func_{j2}, func_{j3}, ...$ are called within $func_i$.

Producing a graphical representation of the call graph similar to the current feature of LLVM [2] will be considered as an extra credit.

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. So, it is highly recommended that you start early.

Grading

Your grade will be mainly based on the correctness of your implementation. We will also use a mixture of face-to-face demonstration, and your writeup (1 or 2-pages) of the project in grading. Your writeup should include the technical approach and implementation details.

Submission

Please send your code package together with instructions about how to run your analysis to Guanhong Tao (taog@purdue.edu).

References

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

[2] Generating Call Graph from C++ code, Stack overflow. `http://stackoverflow.com/questions/5373714/generate-calling-graph-for-c%-code`.

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

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

[5] LLVM homepage. `http://llvm.org/`.

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

[7] Runnign a pass with opt. `http://llvm.org/docs/WritingAnLLVMPass.html#running-a-pass-with-opt`.