Project 2

CS 510

Priyam Biswas

Introduction

The objective of this project is to generate a call graph from input program in LLVM. A call graph is a control flow graph showing the calling relationships between different functions in a program. LLVM is a compiler framework designed for compile-time, link-time and runtime optimization.

In this project, I have generated call graph from a given input program by implementing a LLVM pass. The pass gives output in two ways: a) in the console b) generates a dot file of the callgraph.

Pass Implementation:

I have implemented a module pass and modified it to use it as a sanitizer so that it can be easily used with **'-fsanitize'** option. For generating the call graph, both direct and indirect function call has been considered. Implementation details of each type is described below:

i) **Direct Call**

It is easy to check whether a function has been called directly. First, I have checked for a call instruction within the function, after getting a call instruction if the *getCalledFunction()* of the instruction is not *null* then, the function has been called directly.

ii) **Indirect Call**

If the *getCalledFunction()* of a call instruction returns *null*, then, we definitely know that a function has been called indirectly. For indirect function call, I considered two cases:

a. **via Function Pointer**

For checking, whether an indirect function call has been performed via a function pointer, I have analyzed the control flow of the program. First, I checked whether the *getCalledValue()* of the call instruction is a load instruction, then trace back the operands of the load instruction to retrieve the name of function.

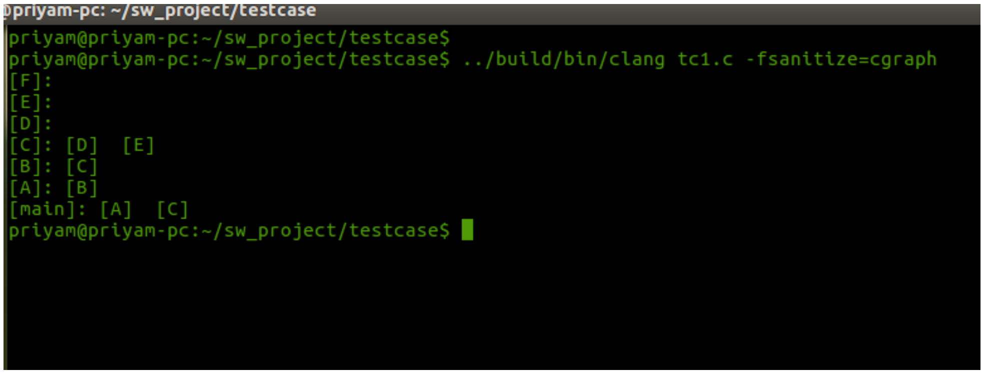
b. **via Struct Field**

This case was a bit complicated compared to the other two above cases. Similar to the previous case (a), first, I checked the *getCalledValue()* of the call instruction is a load instruction and then again checked whether the operands of the load instruction is a *GetElementPointer (gep)* instruction, because fields of struct are accessed via *gep* instruction. Then I have to trace back operands of *gep* instruction to get the function name.

Evaluation

To evaluate that my pass is working correctly, I tested the given three test cases as well as my own test cases. The result of the given three test cases is represented below:

Test Case 1:



```
priyam@priyam-pc:~/sw_project/testcase$
priyam@priyam-pc:~/sw_project/testcase$ ../build/bin/clang tc1.c -fsanitize=cgraph
[F]:
[E]:
[D]:
[C]: [D]  [E]
[B]: [C]
[A]: [B]
[main]: [A]  [C]
priyam@priyam-pc:~/sw_project/testcase$ 
```

Figure1: Console output of TestCase1

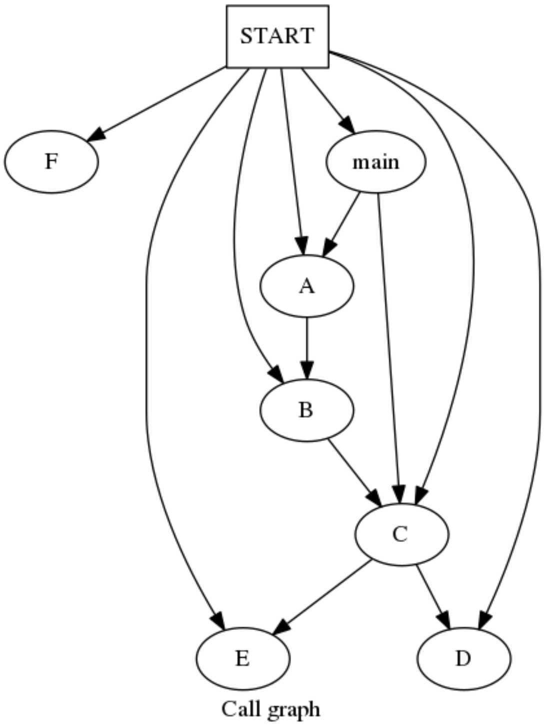
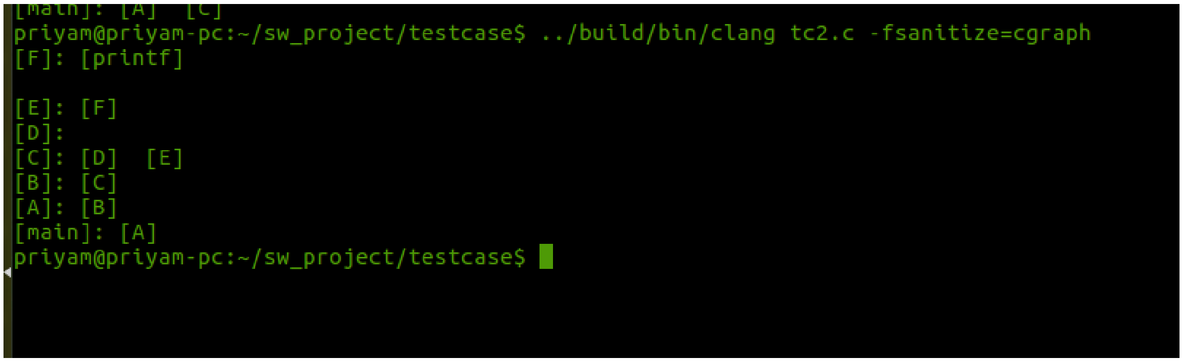


Figure2: Call graph of TestCase1

Test Case 2:



```
[main]: [A]  [C]
priyam@priyam-pc:~/sw_project/testcase$ ../build/bin/clang tc2.c -fsanitize=cgraph
[F]: [printf]

[E]: [F]
[D]:
[C]: [D]  [E]
[B]: [C]
[A]: [B]
[main]: [A]
priyam@priyam-pc:~/sw_project/testcase$
```

Figure3: Console output of TestCase2

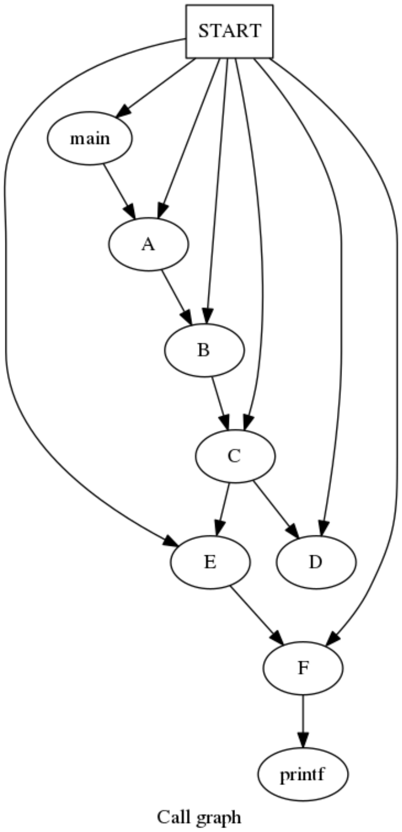
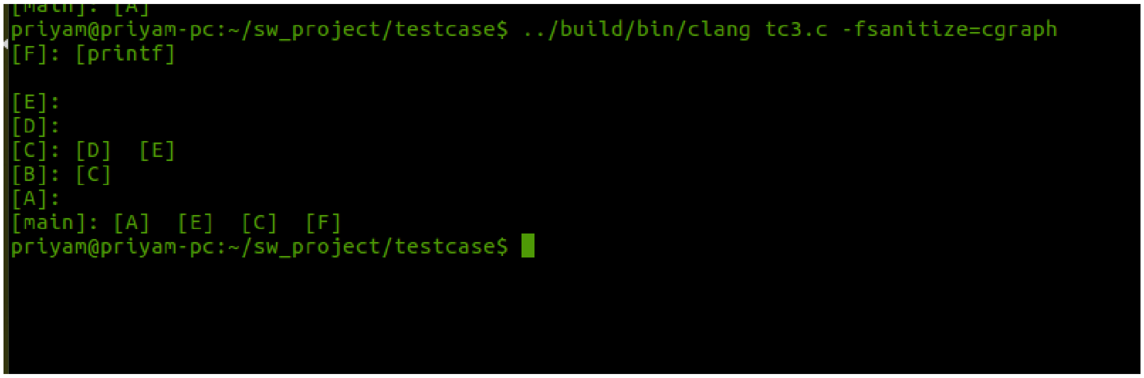


Figure4: Call Graph of TestCase2

Test Case 3:



```
[main]: [A]
priyam@priyam-pc:~/sw_project/testcase$ ../build/bin/clang tc3.c -fsanitize=cgraph
[F]: [printf]

[E]:
[D]:
[C]: [D]  [E]
[B]: [C]
[A]:
[main]: [A]  [E]  [C]  [F]
priyam@priyam-pc:~/sw_project/testcase$ 
```

Figure5: Console output of TestCase3

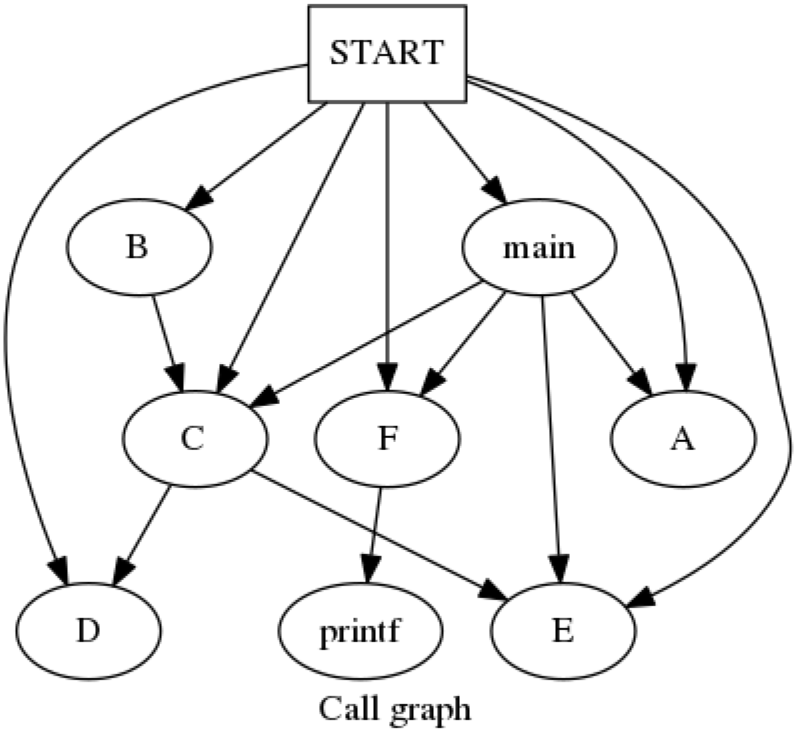


Figure6: Call Graph of TestCase3

STEPS:

To build llvm, we have to follow this steps:

Step1: First, we need to create the build directory in the directory project2, where there are two other directories name llvm and testcase

```
mkdir build && cd build
```

Step2: Then from the build directory, we have to run **cmake**, there is a script in the project2 directory , we can simply run this script, but before that, we need to check whether we have execution permission to that script

```
chmod +x cmake_script.sh
cd build
../cmake_script.sh
```

or we can just give the command of cmake instead of running the script:

```
cmake -GNinja \
-DCMAKE_BUILD_TYPE=Debug \
-DCMAKE_C_COMPILER=clang \
-DCMAKE_CXX_COMPILER=clang++ \
-DLLVM_ENABLE_ASSERTIONS=ON \
-DLLVM_BUILD_TESTS=OFF \
-DLLVM_BUILD_EXAMPLES=OFF \
-DLLVM_INCLUDE_TESTS=OFF \
-DLLVM_INCLUDE_EXAMPLES=OFF \
-DBUILD_SHARED_LIBS=on \
-DLLVM_TARGETS_TO_BUILD="X86" \
-DCMAKE_C_FLAGS="-fstandalone-debug -fuse-ld=gold" \
-DCMAKE_CXX_FLAGS="-fstandalone-debug -fuse-ld=gold" \
 ../llvm
```

Step 3: For building, we are needed to have ninja, if ninja is not installed, we have to remove the -GNinja option from the cmake script

```
ninja
```

Step 4: Next, we have to compile test case with my pass, the general command for this is

```
/path/to/clang   /path/to/testcase/tc1.c    -fsanitize=cgraph
```

For example, to check testcase tc1.c within the testcase directory, we have to give the following command:

```
../build/bin/clang tc1.c -fsanitize=cgraph
```

Step 5: The previous command will output in the console, it will also generate a dot file of the program, to check the dot file, we can convert it from dot to png using the following command. For this command we need to install **graphviz**.

```
dot -Tpng cgraph.dot > cgraph.png
```

Files Needed to change:

To implement the pass, I have created a pass name “CGRAPH.cpp” and put it in the “llvm/lib/Transforms/Instrumentation/” directory. Apart from this, the following files were edited :

- /llvm/tools/clang/include/clang/Basic/Sanitizers.def
- /llvm/include/llvm/Transforms/Instrumentation.h
- /llvm/include/llvm/InitializePasses.h
- /llvm/lib/Transforms/Instrumentation/CMakeLists.txt

Conclusion

This project helped me to better understand how LLVM works and how to generate call graph and the relationships between different functions.