CS510'15 Project #2 Call Graph Generation using LLVM (Due: Feb-26, 2015)

February 6, 2015

1 Goal

In this project, you will learn to write an LLVM pass that will generate a call graph and find all the allocation sites in a C program. Your pass and analysis will be *interprocedural.* Meaning it will operate across the whole program, not just one function. Your pass and analysis will also be *flow-sensitive.* Meaning it should not include paths that can be statically determined to be unreachable.

2 Call Graphs

A call graph is a directed graph that represents calling relationships between functions in a program. Each node represents a function and each edge (f, g) indicates that function f calls function g. Keep in mind that function calls can be recursive.

3 Flow-sensitive Static Analysis

Your analysis will be *flow-sensitive.* Note that whether a given condition is true or false is undecidable in the general case. When your analysis encounters a statically undecidable condition it should consider all branches from that condition to be possible. However, when your analysis reaches a statically decidable condition it should only take the appropriate path. Consider the two following examples:

```c
#include <stdio.h>
int main(int argc, char** argv) {
  int x = 1;
  if (x == 1) {
    printf("hello\n");
  } else {
    printf("bye\n");
  }
  return 0;
}
```

Listing 1: Statically Decidable

```c
#include <stdio.h>
int main(int argc, char** argv) {
  if (argc == 1) {
    printf("hello\n");
  } else {
    printf("bye\n");
  }
  return 0;
}
```

Listing 2: Statically Undecidable

In Listing 1 we can statically determine the program will always take the ”if” branch. In Listing 2 we cannot statically determine which branch will be taken because we don’t statically know the value for `argc`. Keep in mind that constants can be passed as parameters (requires a global analysis).

4 Function Pointers

Your call graph must include both direct function calls and calls through function pointers. In general, statically determining which function a function pointer points to is undecidable. Keep in mind that function pointers can be passed as parameters (requires a global analysis). For this project we limit our test cases to three statically decidable cases:

4.1 Case 1: direct assignment

```c
1 void foo() { }
2
3 int main(int argc, char** argv) {
4   void (*p)();
5   p = &foo;
6   ...
7   p();
8   return 0;
9 }
```

Note that "..." on line 6 represents arbitrary code. The only restriction is that p is not modified. You of course need to correctly handle cases like:

```c
1  void foo() { }
2  void bar() { }
3
4  int main(int argc, char** argv) {
5    void (*p)();
6    p = &foo;
7    ...
8    p = &bar;
9    p();
10   return 0;
11 }
```

Because you can statically know that `p` points to `bar` when it is called.

4.2 Case 2: function pointer members of structs

For example:

```c
1  typedef struct {
2    void (*p)();
3    int n;
4  } apple;
5
6  void foo() { }
7
8  int main(int argc, char** argv) {
9    apple a;
10   a.p = &foo;
11   ...
12   a.p();
13   return 0;
14 }
```

4.3 Case 3: function pointer from pointers to structs

For example:

```c
typedef struct {
  void (*p)();
  int n;
} apple;

void foo() { }

int main(int argc, char** argv) {
  apple *a = malloc(sizeof(apple));;
  a->p = &foo;
  ...
  a->p();
}
```

Keep in mind that in all three cases the ”...” can represent any arbitrary code that doesn’t reassign the function pointer value. We make no guarantee that each function pointer will only be assigned once. However, none of our test cases will include more than one direction, i.e., `a->b->c` will not appear in our test cases.

4.4 Call Graph Output Format

Your pass should print the call graph output to `stderr`. The format is:

```
[func1]:[callee1],[callee2],...,[calleeN]
[func2]:[callee1],[callee2],...,[calleeN]
...
[funcN]:[callee1],[callee2],...,[calleeN]
```

Note that we don’t require a specific ordering for the callees or the callers. Specifically you could reorder the rows and the output is still correct, or for each row you could reorder the callees and the output is still correct.

For example this program:

```c
void a() {
  b();
}

void b() { }

void c() {
  a();
  b();
}

int main(int argc, char** argv)) {
  c();
  return 0;
}
```

Would yield the following output:

```
[main]:[c]
[a]:[b]
[b]:
[c]:[a],[b]
```

5 Memory Allocations

Your pass will identify all the memory allocations in the program. For each memory allocation, you will record the containing function, the type of the allocated variable, and the amount of memory allocated (in byte) if statically determined. If the amount of memory allocated is statically undecidable then output "unknown" in the size field for that allocation.

We're analyzing C programs so all memory will be allocated with `malloc`, `calloc`, `realloc`, and `alloca` (please explain why `alloca` is special and in what form).

5.1 Memory Allocations Output Format

```
[func1]:[type1,size1],...,[typeN,sizeN]
...
[funcN]:[typeN,sizeN],...,[typeN,sizeN]
```

For example consider the following program:

```c
struct apple {
  int x;
  double a;
};

void bar() { }

void foo() {
  double *y = malloc(42);
  struct apple *a = malloc(sizeof(struct apple));
  bar();
}

int main(int argc, char** argv) {
  int *x = malloc(100);
  foo();
  return 0;
}
```

The output for this program would be:

```
[main]:[int*,100]
[foo]:[double*,42],[%struct.apple*, 16]
[bar]:
```

Note that `%struct.apple*` in LLVM is equivalent to `struct apple*` in C. Your pass should output the LLVM types so you don't have to do string manipulation.

6 Building LLVM

For this project, you will need to build LLVM from source. There are many guides on the web for building LLVM.

At a minimum you must read and understand `http://llvm.org/docs/GettingStarted.html`. This page gives a high level summary of the LLVM project. Note this page explains how to build using configure, but you're required to build with cmake for this project.

6.1 Building with cmake

The TA strongly suggests you get started by building the unmodified LLVM source to make sure your build environment works. The TA has added the LLVM source to each of your repositories, so all you need to do is clone it, cmake, and build. Specifically, run the following commands:

```
git clone <your repository repo>
cd <your repository name>
cd project2
mkdir build
cd build
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ \
   -DLLVM_INCLUDE_TESTS=OFF -DLLVM_TARGETS_TO_BUILD=X86 \
   ../llvm
make -j8
```

Listing 3: Build Commands

If this worked, go ahead and read the documentation in the next section then start working on your pass. Note that this requires `clang` and `clang++` to be installed on your system. We also recommend you install ld gold on your machine. If you're on a recent Ubuntu run:

```
sudo apt-get install binutils-gold binutils-dev
cd /usr/bin
sudo rm ld
sudo ln -s ld.gold ld
```

7 Writing your pass

It is essential that you read `http://llvm.org/docs/WritingAnLLVMPass.html`.
In your repository there is a directory called `project2/llvm/lib/Transforms/CallGraphPass`.
There is a file in this directory called "CallGraphPass.cpp" with stub code. Implement your entire project in this file.

8 Extra Credit

8.1 Tracking non-local variables

You will receive extra credit if your analysis can produce the correct output on more advanced test cases. All previous examples dealt with local variables – variables whose reaching definition and use are in the same function. However real programs often assign a value to a variable in one function and use it in another.

For a function pointer example consider:

```c
void bar() { };

void foo(void (*fn)()) {
  fn();
}

int main(int argc, char** argv) {
  void (*p)();
  p = &bar;
  foo(p);
  return 0;
}
```

The correct output is:

```
[main]:[foo]
[foo]:[bar]
[bar]:
```

The same idea applies for finding unreachable paths. Consider:

```c
void bar() {};
void baz() {};

void foo(int x) {
  if (x == 0) {
    bar()
  } else {
    baz();
  }
}

int main(int argc, char** argv) {
  int y = 0;
  foo(y);
  return 0;
}
```

The correct output is:

```
[main]:[foo]
[foo]:[bar]
[bar]:
[baz]:
```

Beware that this analysis is quiet tricky for more complicated programs. That's why it's extra credit. For example, in this program:

```c
void bar() {};
void baz() {};
void buz(int x) {
  foo(x);
}

void foo(int x) {
  if (x == 0) {
    bar()
  } else {
    baz();
  }
}

int main(int argc, char** argv) {
  int y = 0;
  foo(y);
  buz(argc);
  return 0;
}
```

It is not statically decidable that `foo` will never call `baz`, so the correct output is:

```
[main]:[foo]
[foo]:[bar],[baz]
[bar]:
[baz]:
[buz]:[foo]
```

8.2 DOT file output

For extra credit, have your pass produce a DOT file (`http://en.wikipedia.org/wiki/DOT_%28graph_description_language%29`) called "CallGraph.dot" in the current working directory that represents the call graph of the program you're analyzing. DOT files can be used to generate graph visualizations.

9 How to submit

Like project1, you will submit your code in BitBucket. All your code should be in the directory `project2/llvm/lib/Transforms/CallGraph/Pass`.

10 Grading

Your code needs to compile on Ubuntu 14.04 in order to be graded. This will not be an issue if you followed the instruction to only add code to CallGraph-

Pass.cpp.

To build your code, the TA will use the exact same set of commands you used to build your code in Listing 3. It is your responsibility to make sure your code builds using those exact commands.

To run the test cases, the TA will first build your code, then from the build directory run:

```
bin/opt -load lib/LLVMCallGraphPass.so -CallGraphPass < test.bc > /
    dev/null
```