LLVM/clang Tutorial for Project - Part 1

Passes

For this assignment you will need to extend the `FunctionPass` class. In particular, methods you might need to override include

- `virtual bool doInitialization(Module &)` which is inherited from `Pass`. Override this method to access the current module and extract the context with:

```
LLVMContext &context = mod.getContext();
```

- `virtual bool runOnFunction(Function &F)`

Iterators

Use a `Function::iterator` to iterate over the **basic blocks** of a function:

```
for (Function::iterator B = F.begin(), BE = F.end(); B != BE; ++B) {
// Here B is a pointer to a basic block
  ...
}
```

Use a `BasicBlock::iterator` to iterate over the **instructions** in a basic block:

```
for (BasicBlock::iterator I = B->begin(), IE = B->end(); I != IE; ++I) {
  // Here I is a pointer to an instruction
  ...
}
```

Global Variables

Creation

Global variables define regions of memory allocated at compilation time instead of run-time. So this is a useful option if you think you'll need to allocate data that won't be changing throughout the execution of your program (e.g. a static array).

You can create a global variable with the following constructor:

```
GlobalVariable::GlobalVariable(Module &M,
                               Type * Ty,
                               bool isConstant,
                               LinkageTypes Linkage,
                               Constant * Initializer,
                               const Twine & Name = "")
```

For example, let's say you want to allocate the static array `[0,1]`. You would do that by calling

```
GlobalVariable* v = new GlobalVariable(
    *mod,
    ArrayTy,
    true,
    GlobalValue::InternalLinkage,
    ConstantDataArray::get(*ctx, *(new ArrayRef<uint32_t>({0,1}))),
    "my_array");
```

where `ArrayTy` is the LLVM representation of the type of an array with two elements

```
ArrayType* arrayTy = ArrayType::get(IntegerType::get(F.getContext(), 32), 2);
```

The above allocates a constant array that can be seen in the instrumented file as a line of the form

```
@my_array.NNN = internal constant [2 x i32] [i32 0, i32 1]
```

Reference

To obtain a reference to the above array you will find the `getElementPtr` (http://releases.llvm.org/5.0.1/docs/LangRef.html#getelementptr-instruction) instruction particularly useful.

To understand the particulars of this instruction

- I recommend reading this additional documentation (http://releases-origin.llvm.org/5.0.1/docs/GetElementPtr.html) carefully.
- In particular, this paragraph (http://releases-origin.llvm.org/5.0.1/docs/GetElementPtr.html#why-is-the-extra-0-index-required) helped me understand how to compute the right index for the first element of a statically allocated global array.
- You should also try writing programs that you contain some global variable reference and inspect the produced code, after compiling to LLVM IR.

Alternatively, you might want to look into the `bitcast` (http://releases.llvm.org/5.0.1/docs/LangRef.html#bitcast-to-instruction) instruction, that converts its argument to a specified type without changing any bits.

For example, to convert a pointer `v` to the type `int*`, we can call `IBuilder::CreatePointerCast`

```
Value* castV = Builder.CreatePointerCast(v, Type::getInt32PtrTy(context));
```

The `Builder` class is introduced in the last section of this tutorial.

Looking up Functions

Use `Module::getOrInsertFunction` to look up the specified function in the module's symbol table. Note that `getOrInsertFunction` will return a cast of the existing function if the function already existed with a different prototype.

For example, to look up a function `foo` that accepts two `i32` arguments and returns `void`, we would issue the call:

```
Constant *fooFunc = Mod.getOrInsertFunction(
    "foo",                            // name of function
    Type::getVoidTy(context),         // return type
    Type::getInt32Ty(context),        // first parameter type
    Type::getInt32Ty(context),        // second parameter type
  );
```

Creating a Call

Use the `IRBuilder` to create instructions and insert them into a basic block.

For example we can define a builder as

```
IRBuilder<> Builder(&*blk->getFirstInsertionPt());
```

Suppose we want to call function `foo` from above with arguments `0` and `1`. First we create a vector `args` to hold the arguments to the call

```
std::vector<Value*> args;
args.push_back(zero);
args.push_back(one);
```

where `zero` is the representation for the number `0`

```
Constant * zero = ConstantInt::get(IntegerType::get(F.getContext(),32), 0);
```

Finally, we create the actual call

```
Builder.CreateCall(fooFunc, args);
```

This will place a call to `foo` **before** the basic block that `blk` points to.

How to Build, Link and Run using LLVM/Clang

This might be useful when working with the LLVM/Clang toolchain. Note that this is not the only way to proceed, but just one that has been tried and seems to work.

How to build a bitcode file (.bc)

Assume an source file called HelloWorld.cpp. To build an LLVM bitcode representation run the following:

```
clang++ -c -O0 -emit-llvm HelloWorld.cpp -o HelloWorld.bc
```

The contents of the folder will now be:

```
$ ls
HelloWorld.bc  HelloWorld.cpp
```

How to build an object file (.o) from a bitcode file

Once we have the bitcode file we can use llc to build an object file:

```
$ llc -filetype=obj HelloWorld.bc -o HelloWorld.o
```

The contents of the folder will now be:

```
$ ls
HelloWorld.bc  HelloWorld.cpp  HelloWorld.o
```

How to build an executable file

We can use clang++ again to produce an executable file:

```
$ clang++ HelloWorld.o -o HelloWorld
```

We can run this with:

```
./HelloWorld
Hello world!
```

How to link multiple bitcode files

Assume we have a main file `main.cpp` that calls to a library `lib.cpp`. We can translate these files to bitcode like before and get a `main.bc` and a `lib.bc`:

```
$ clang++ -c -O0 -emit-llvm lib.cpp -o lib.bc
$ clang++ -c -O0 -emit-llvm main.cpp -o main.bc
```

However, before creating an object file from these two bitcode files we first need to link them into a single bitcode file. We can use llvm-link to do this:

```
$ llvm-link lib.bc main.bc -o final.bc
```

This produces a `final.bc` file that contains all the bitcode we need to create an object file, then compile it to binary and run it:

```
$ llc -filetype=obj final.bc -o final.o
$ clang++ final.o -o final
$ ./final
...
```

Keep in mind that for this to work `main.cpp` needs to declare functions it is calling that are defined in `lib.cpp`.