Hello LLVM-C!

Filip van der Meeren

As I child I loved the dutch version of the 99 bottles of beer song. Every programming language I learned up until now have had its 99 bottles of beer song implementation. C, Objective-C, C++, Python, Java, even SQL had and still has its own implementation of 99 bottles of beer. All stored in my personal svn.

After a while in a certain language I started extending the language from within the language. Python has this really nice feature where you can create code dynamically and then push it through its runtime so it ends up as your really written code.

But for a few languages I failed to implement such a feature without hacks, workarounds etcetera. Those languages were C / Objective-C / C++, until I came across LLVM...

Exploring

LLVM[i] is quite large, you should really check out the entire website. But finally it all comes down to one thing, the LLVM Core. More exactly, the LLVM-C interface to the core[ii]. And that is the part where we are going to focus.

LLVM-C has a structure you must follow, just like C has one. In C you have source files, to organize the code in manageable chunks, there counterpart in the core is a module.

Before you start programming with LLVM, you should realize that LLVM uses a Static single assignment form (SSA). This comes down to one little thing: you can assign something to a variable only once. If you want to assign something multiple times to the same variable, you will have to allocate the space for it on the heap (malloc, calloc, realloc) or stack (alloca). Luckily the core has some easy methods to invoke to make this possible.

Hello World

Now lets start with a much used example (don't know why). You guessed it, we are going to write the LLVM version of "Hello World".

This is the C version that we will try to equal:

```
// C Headers
#include <stdio.h>

void
sayHelloWorld(void) {
    puts("Hello LLVM-C world!");
}
```

I will assume a few things in this tutorial, the first is that you know how to use C (if you don't know how to read/write C, congratulations for getting this far without quitting). And the second is that you know how to compile the LLVM libraries and how to use your own compiler and linker. And I will also expect you are putting the following include statement before using the code below.

```
#include <llvm-c/Core.h>
```

Module

As I have said before, LLVM uses modules to represent source files. So, lets create a module:

```
LLVMModuleRef module;
module = LLVMModuleCreateWithName("Say hello to LLVM");
```

That wasn't to hard now was it?

Puts

Now we are going to use "puts", in C we have the include that takes care of the fact that the compiler/linker must know the definition of a method before it can use it. LLVM requires the same.

```
int puts(const char *s);
```

is what we want to create. Lets start with the return type, what is `int` on a system. Is it a 16 bit integer, a 32 bit integer? It might be more or less. Who knows, for all we know it might be 17 bits on a system. You never know what the compiler translates this to. So lets be sure and retrieve the llvm-type:

```
LLVMTypeRef returnType = LLVMIntType(sizeof(int) * CHAR_BIT);
```

The statement creates a LLVMTypeRef that represents an integer with X bits, these types are comparable with the types in C. There are a few quickies made for much the used (LLVMInt1Type(), LLVMInt8Type(), LLVMInt16Type(), LLVMInt32Type(), LLVMInt64Type()), but they can all be created with the one above.

For those who don't know what `CHAR_BIT` is, it is defined in <limits.h> and it contains the number of bits in one char. Just a reminder, sizeof() returns the number of bytes for any given type.

We now have a return type, lets create the entire functiontype.

```
LLVMTypeRef putsArguments[] = {
    LLVMPointerType(LLVMInt8Type(), 0),
};

LLVMTypeRef functionType;
functionType = LLVMFunctionType(
    /* Return type */ LLVMIntType(sizeof(int) * CHAR_BIT),
    /* Parameters  */ putsArguments,
    /* Param count */ 1,
    /* Var Args ?  */ false);
```

The code is mostly explanatory except for one small detail, LLVMPointerType. You are able to create a pointer to any type by providing the type as the first argument and an address space as the second parameter. You can ignore this and just provide 0 (zero). This is the default value that is valid on most (dare I say all?) systems.

All that is left is creating the function... We have a type, all that is left is providing a name.

```
LLVMValueRef function;
function = LLVMAddFunction(module, "puts", functionType);
```

Admiring your work

If you want to see the result of your hard work or just check what it looks like in the LLVM assembly language, just invoke the following line:

```
LLVMDumpModule(module);
```

This prints all of your LLVM assembly onto the stderr.

Checking your work

If you are doubtful that what you create is valid LLVM assembly. You should validate your module. This is by no means a requirement, but it is nice to know even in production code.

```
char * errMsg = NULL;
if(LLVMVerifyModule(module, LLVMAbortProcessAction, &errMsg) == 0)
{
    /* valid module! */
}
```

If the module isn't value it will have a message attached to it, the message will be stored into the char* variable you provided. This has to be freed after you no longer need it by a call to LLVMDisposeMessage(char*).

You can change the LLVMAbortProcessAction (guess what that option will do ;-)) with LLVMPrintMessageAction (prints to stderr) and LLVMReturnStatusAction that just returns without doing any extras. Any message can be retrieved by checking the errMsg.

The same method also exists for checking single functions:

```
LLVMVerifyFunction(function, LLVMPrintMessageAction);
```

It also returns 0 (zero) after a valid function. It isn't that useful, except for debugging purposes. But it is nice to know.

Hello LLVM method

A prototype alone isn't that useful. Lets create a function that calls it...

```
LLVMTypeRef functionType;
LLVMValueRef function;
functionType = LLVMFunctionType(/* Return type */ LLVMVoidType(),
                                /* Parameters  */ NULL,
                                /* Param count */ 0,
                                /* Var Args ?  */ false);

// Add it to the module, with a name.
function = LLVMAddFunction(module, "sayHelloWorld", functionType);
```

All we are missing now is a body. I hope you have created C code with a lot of labels and goto statements. Because every branch in your code will become a block (don't worry much about this that is for the following chapter).

Lets give our function its first block:

```
LLVMBasicBlockRef block = LLVMAppendBasicBlock(function, "aName");
```

The block can have a name, or you can leave it blank, you can give a function a thousand blocks with the same name. LLVM will make sure that each of them gets a postfix appended to avoid name-collisions.

Create a builder (that can optionally be reused for multiple blocks) to give the blocks some content.

```
LLVMBuilderRef builder = LLVMCreateBuilder();
/* using the builder */
LLVMDisposeBuilder(builder);
```

During the builders existence you can place it at the beginning of a block or the end.

```
LLVMPositionBuilderAtEnd(builder, block);
```

And whenever you are positioned in a block, you can invoke LLVMBuildXXX methods. So lets do this:

```
LLVMValueRef callArguments[] = {
    LLVMBuildGlobalStringPtr(builder, "Hello LLVM-C world!", ""),
};

LLVMBuildCall(builder, putsFunction, callArguments, 1, "");
LLVMBuildRetVoid(builder);
```

What does the code above? It creates a call to the putsFunction prototype you have defined earlier. LLVMs JIT will resolve any method that you haven't provided.

Each block has to have a branch to another block or an return. But that will become much clearer in the next chapter.

There you are, your entire module is up! And now just get it running.

Running your first hello

I will explain the following in more detail in the chapters that will follow. But just to provide you with some satisfaction from what you have accomplished so far. Lets run your dynamically created function.

```
#include <llvm-c/ExecutionEngine.h>

LLVMInitializeNativeTarget();
LLVMLinkInJIT();

LLVMExecutionEngineRef engine;
void (*myHello)(void);
if(LLVMCreateJITCompilerForModule(&myEngine,
                                  module, 0, NULL) == 0) {

    myHello = (void (*)(void))LLVMGetPointerToGlobal(engine,
 sayHelloFunctionValueRef);
    myHello();
}
```

Wow, that chuck was impressive. Now what did I do ? I created a Just In Time compiler, something that Java and dotNet do: compile the function from the minute it is used, not a second earlier. Then I ask the engine for a pointer to my method. LLVM compiles my method and returns a pointer to it.

All you then have to do, is invoke your freshly created method!

Have fun experimenting with LLVM-C ! Keep checking these pages, because the next chapter will create 99 bottles of beer in LLVM.

[i] http://llvm.org/

[ii] If you want to use C++, I suggest you take a look at the C++ tutorials. The C skills are transferable, but it is still a huge difference.