Create a working compiler with the LLVM framework, Part 1

Build a custom compiler with LLVM and its intermediate representation

Arpan Sen

June 19, 2012
(First published June 05, 2012)

The LLVM compiler infrastructure provides a powerful way to optimize your applications regardless of the programming language you use. Learn the basics of the LLVM in this first article of a two-part series. Building a custom compiler just got easier!
19 Jun 2012 - *Added links to Part 2 of this article to sidebars in the introduction and Conclusion, and in Resources*.

The LLVM (formerly the Low Level Virtual Machine) is an extremely powerful compiler infrastructure framework designed for compile-time, link-time, and run time optimizations of programs written in your favorite programming language. LLVM works on several different platforms, and its primary claim to fame is generating code that runs fast.

Other articles in this series

View more articles in the Create a working compiler with the LLVM framework series.

The LLVM framework is built around a well-documented intermediate representation (IR) of code. This article—the first in a two-part series—delves into the basics of the LLVM IR and some of its subtleties. From there, you will build a code generator that can automate the work of generating the LLVM IR for you. Having an LLVM IR generator means that all you need is a front end for your favorite language to plug into, and you have a full flow (front-end parser + IR generator + LLVM back end). Creating a custom compiler just got simplified.

Getting started with the LLVM

Before you start, you must have the LLVM compiled on your development computer (see Related topics for a link). The examples in this article are based on LLVM version 3.0. The two most important tools for post-build and installation of LLVM code are `llc` and `lli`.

llc and lli

Because LLVM is a virtual machine (VM), it likely should have its own intermediate byte code representation, right? Ultimately, you need to compile LLVM byte code into your platform-specific assembly language. Then you can run the assembly code through a native assembler and linker to generate executables, shared libraries, and so on. You use `llc` to convert LLVM byte code to platform-specific assembly code (see Related topics for a link to more information about this tool). For directly executing portions of LLVM byte code, don't wait until the native executable crashes to figure out that you have a bug or two in your program. This is where `lli` comes in handy, as it can directly execute the byte code. `lli` performs this feat either through an interpreter or by using a just-in-time (JIT) compiler under the hood. See Related topics for a link to more information about `lli`.

llvm-gcc

llvm-gcc is a modified version of the GNU Compiler Collection (gcc) that can generate LLVM byte code when run with the `-S -emit-llvm` options. You can then use `lli` to execute this generated byte code (also known as *LLVM assembly*). For more information about llvm-gcc, see Related topics. If you don't have llvm-gcc preinstalled on your system, you should be able to build it from sources; see Related topics for a link to the step-by-step guide.

Hello World with LLVM

To better understand LLVM, you have to learn LLVM IR and its idiosyncrasies. This process akin to learning yet another programming language. But if you have been through `c` and `c++` and their quirks, there shouldn't be much to deter you in the LLVM IR. Listing 1 shows your first program, which prints "Hello World" in the console output. To compile this code, you use llvm-gcc.

Listing 1. The familiar-looking Hello World program

```
#include <stdio.h>
int main( )
{
  printf("Hello World!\n");
}
```

To compile the code, enter this command:

```
Tintin.local# llvm-gcc helloworld.cpp -S -emit-llvm
```

After compilation, llvm-gcc generates the file helloworld.s, which you can execute using `lli` to print the message to console. The `lli` usage is:

```
Tintin.local# lli helloworld.s
Hello, World
```

Now, take a first look at the LLVM assembly. Listing 2 shows the code.

Listing 2. LLVM byte code for the Hello World program

```
@.str = private constant [13 x i8] c"Hello World!\00", align 1 ;

define i32 @main() ssp {
entry:
  %retval = alloca i32
  %0 = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  %1 = call i32 @puts(i8* getelementptr inbounds ([13 x i8]* @.str, i64 0, i64 0))
  store i32 0, i32* %0, align 4
  %2 = load i32* %0, align 4
  store i32 %2, i32* %retval, align 4
  br label %return
return:
  %retval1 = load i32* %retval
  ret i32 %retval1
}

declare i32 @puts(i8*)
```

Understanding the LLVM IR

The LLVM comes with a detailed assembly language representation (see Related topics for a link). Here are the must-knows before you try to craft your own version of the Hello World program discussed earlier:

- Comments in LLVM assembly begin with a semicolon (`;`) and continue to the end of the line.
- Global identifiers begin with the at (`@`) character. All function names and global variables must begin with `@`, as well.
- Local identifiers in the LLVM begin with a percent symbol (`%`). The typical regular expression for identifiers is `[%@][a-zA-Z$._][a-zA-Z$._0-9]*`.
- The LLVM has a strong type system, and the same is counted among its most important features. The LLVM defines an integer type as `iN`, where *N* is the number of bits the integer will occupy. You can specify any bit width between 1 and 223- 1.
- You declare a vector or array type as `[no. of elements X size of each element]`. For the string "Hello World!" this makes the type `[13 x i8]`, assuming that each character is 1 byte and factoring in 1 extra byte for the NULL character.
- You declare a global string constant for the hello-world string as follows: `@hello = constant [13 x i8] c"Hello World!\00"`. Use the `constant` keyword to declare a constant followed by the type and the value. The type has already been discussed, so let's look at the value: You begin by using `c` followed by the entire string in double quotation marks, including `\0` and ending with `0`. Unfortunately, the LLVM documentation does not provide any explanation of why a string needs to be declared with the `c` prefix and include both a NULL character and 0 at the end. See Related topics for a link to the grammar file, if you're interested in exploring more LLVM quirks.
- The LLVM lets you declare and define functions. Instead of going through the entire feature list of an LLVM function, I concentrate on the bare bones. Begin with the `define` keyword followed by the return type, and then the function name. A simple definition of `main` that returns a 32-bit integer similar to: `define i32 @main() { ; some LLVM assembly code that returns i32 }`.

- Function declarations, like definitions, have a lot of meat to them. Here's the simplest declaration of a `puts` method, which is the LLVM equivalent of `printf`: `declare i32 puts(i8*)`. You begin the declaration with the `declare` keyword followed by the return type, the function name, and an optional list of arguments to the function. The declaration must be in the global scope.
- Each function ends with a return statement. There are two forms of return statement: `ret <type> <value>` or `ret void`. For your simple main routine, `ret i32 0` suffices.
- Use `call <function return type> <function name> <optional function arguments>` to call a function. Note that each function argument must be preceded by its type. A function test that returns an integer of 6 bits and accepts an integer of 36 bits has the syntax: `call i6 @test( i36 %arg1 )`.

That's it for a start. You need to define a main routine, a constant to hold the string, and a declaration of the `puts` method that handles the actual printing. Listing 3 shows the first attempt.

Listing 3. First attempt to create a hand-crafted Hello World program

```
declare  i32 @puts(i8*)
@global_str = constant [13 x i8] c"Hello World!\00"
define i32 @main {
  call i32 @puts( [13 x i8] @global_str )
  ret i32 0
}
```

And here's the log from `lli`:

```
lli: test.s:5:29: error: global variable reference must have pointer type
  call i32 @puts( [13 x i8] @global_str )
                            ^
```

Oops, that didn't work as expected. What just happened? The LLVM, as mentioned earlier, has a powerful type system. Because `puts` was expecting a pointer to `i8` and you passed a vector of `i8`, `lli` was quick to point out the error. The obvious fix to this problem, coming from a `C` programming background, is typecasting. And that brings you to the LLVM instruction `getelementptr`. Note that you must modify the `puts` call in Listing 3 to something like `call i32 @puts(i8* %t)`, where `%t` is of type `i8*` and is the result of the typecast from `[13 x i8] to i8*`. (See Related topics for a link to a detailed description of `getelementptr`.) Before going any farther, Listing 4 provides the code that works.

Listing 4. Using getelementptr to correctly typecast to a pointer

```
declare i32 @puts (i8*)
@global_str = constant [13 x i8] c"Hello World!\00"

define i32 @main() {
  %temp = getelementptr [13 x i8]*  @global_str, i64 0, i64 0
  call i32 @puts(i8* %temp)
  ret i32 0
}
```

The first argument to `getelementptr` is the pointer to the global string variable. The first index, `i64 0`, is required to step over the pointer to the global variable. Because the first argument to the

`getelementptr` instruction must always be a value of type `pointer`, the first index steps through that pointer. A value of 0 means 0 elements offset from that pointer. My development computer is running 64-bit Linux®, so the pointer is 8 bytes. The second index, `i64 0`, is used to select the 0th element of the string, which is supplied as the argument to `puts`.

Creating a custom LLVM IR code generator

Understanding the LLVM IR is fine, but you need an automated code-generation system that dumps LLVM assembly. Thankfully, LLVM does have enough application programming interface (API) support to see you through this effort (see Related topics for a link to the programmer's manual). Look for the file LLVMContext.h on your development computer; if that file is missing, something is probably wrong with the way you installed the LLVM.

Now, let's create a program that generates LLVM IR for the Hello World program discussed earlier. The program won't deal with the entire LLVM API here, but the code samples that follow should prove that a fair bit of the LLVM API is intuitive and easy to use.

Linking against LLVM code

The LLVM comes with a nifty tool called `llvm-config` (see Related topics). Run `llvm-config –cxxflags` to get the compile flags that need to be passed to g++, `llvm-config –ldflags` for the linker options, and `llvm-config –libs` to link against the right LLVM libraries. In the sample in Listing 5, all the options need to be passed to g++.

Listing 5. Using llvm-config to build code with the LLVM API

```
tintin# llvm-config --cxxflags --ldflags --libs \
-I/usr/include  -DNDEBUG -D_GNU_SOURCE \
-D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS \
-D__STDC_LIMIT_MACROS -O3  -fno-exceptions -fno-rtti -fno-common \
-Woverloaded-virtual -Wcast-qual \
-L/usr/lib  -lpthread -lm \
-lLLVMXCoreCodeGen -lLLVMTableGen -lLLVMSystemZCodeGen \
-lLLVMSparcCodeGen -lLLVMPTXCodeGen \
-lLLVMPowerPCCodeGen -lLLVMMSP430CodeGen -lLLVMMipsCodeGen \
-lLLVMMCJIT -lLLVMRuntimeDyld \
-lLLVMObject -lLLVMMCDisassembler -lLLVMXCoreDesc -lLLVMXCoreInfo \
-lLLVMSystemZDesc -lLLVMSystemZInfo \
-lLLVMSparcDesc -lLLVMSparcInfo -lLLVMPowerPCDesc -lLLVMPowerPCInfo \
-lLLVMPowerPCAsmPrinter \
-lLLVMPTXDesc -lLLVMPTXInfo -lLLVMPTXAsmPrinter -lLLVMMipsDesc \
-lLLVMMipsInfo -lLLVMMipsAsmPrinter \
-lLLVMMSP430Desc -lLLVMMSP430Info -lLLVMMSP430AsmPrinter \
-lLLVMMBlazeDisassembler -lLLVMMBlazeAsmParser \
-lLLVMMBlazeCodeGen -lLLVMMBlazeDesc -lLLVMMBlazeAsmPrinter \
-lLLVMMBlazeInfo -lLLVMLinker -lLLVMipo \
-lLLVMInterpreter -lLLVMInstrumentation -lLLVMJIT -lLLVMExecutionEngine \
-lLLVMDebugInfo -lLLVMCppBackend \
-lLLVMCppBackendInfo -lLLVMCellSPUCodeGen -lLLVMCellSPUDesc \
-lLLVMCellSPUInfo -lLLVMCBackend \
-lLLVMCBackendInfo -lLLVMBlackfinCodeGen -lLLVMBlackfinDesc \
-lLLVMBlackfinInfo -lLLVMBitWriter \
-lLLVMX86Disassembler -lLLVMX86AsmParser -lLLVMX86CodeGen \
-lLLVMX86Desc -lLLVMX86AsmPrinter -lLLVMX86Utils \
-lLLVMX86Info -lLLVMAsmParser -lLLVMARMDisassembler -lLLVMARMAsmParser \
-lLLVMARMCodeGen -lLLVMARMDesc \
-lLLVMARMAsmPrinter -lLLVMARMInfo -lLLVMArchive -lLLVMBitReader \
-lLLVMAlphaCodeGen -lLLVMSelectionDAG \
-lLLVMAsmPrinter -lLLVMMCParser -lLLVMCodeGen -lLLVMScalarOpts \
```

```
-lLLVMInstCombine -lLLVMTransformUtils \
-lLLVMipa -lLLVMAnalysis -lLLVMTarget -lLLVMCore -lLLVMAlphaDesc \
-lLLVMAlphaInfo -lLLVMMC -lLLVMSupport
```

LLVM modules, contexts, and more

An LLVM module class is the top-level container for all other LLVM IR objects. An LLVM module class can contain a list of global variables, functions, other modules on which this module depends, symbol tables, and more. Here's the constructor for an LLVM module:

```
explicit Module(StringRef ModuleID, LLVMContext& C);
```

You must begin your program by creating an LLVM module. The first argument is the name of the module and can be any dummy string. The second argument is something called `LLVMContext`. The `LLVMContext` class is somewhat opaque, but it's enough to understand that it provides a context in which variables and so on are created. This class becomes important in the context of multiple threads, where you might want to create a local context per thread, and each thread runs completely independently of any other's context. For now, use the default global context handle that the LLVM provides. Here's the code to create a module:

```
llvm::LLVMContext& context = llvm::getGlobalContext();

llvm::Module* module = new llvm::Module("top", context);
```

The next important class to learn is the one that actually provides the API to create LLVM instructions and insert them into basic blocks: the `IRBuilder` class. `IRBuilder` comes with a lot of bells and whistles, but I chose the simplest possible way to construct one—by passing the global context to it with the code:

```
llvm::LLVMContext& context = llvm::getGlobalContext();

llvm::Module* module = new llvm::Module("top", context);

llvm::IRBuilder<> builder(context);
```

When the LLVM object model is ready, you can dump its contents by calling the module's `dump` method. Listing 6 shows the code.

Listing 6. Creating a dummy module

```
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Support/IRBuilder.h"

int main()
{
  llvm::LLVMContext& context = llvm::getGlobalContext();
  llvm::Module* module = new llvm::Module("top", context);
  llvm::IRBuilder<> builder(context);

  module->dump( );
}
```

After running the code in Listing 6, this prints to the console:

```
; ModuleID = 'top'
```

You need to create the `main` method next. LLVM provides the classes `llvm::Function` to create a function and `llvm::FunctionType` to associate a return type for the function. Also, remember that the `main` method must be a part of the module. Listing 7 shows the code.

Listing 7. Adding the main method to the top module

```
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Support/IRBuilder.h"

int main()
{
  llvm::LLVMContext& context = llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("top", context);
  llvm::IRBuilder<> builder(context);

  llvm::FunctionType *funcType =
      llvm::FunctionType::get(builder.getInt32Ty(), false);
  llvm::Function *mainFunc =
      llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);

  module->dump( );
}
```

Note that you wanted `main` to return `void`, which is why you called `builder.getVoidTy()`; if `main` returned `i32`, the call would be `builder.getInt32Ty()`. After compiling and running the code in Listing 7, the result is:

```
; ModuleID = 'top'
declare void @main()
```

You have not yet defined the set of instructions that `main` is supposed to execute. For that, you must define a basic block and associate it with the `main` method. A *basic block* is a collection of instructions in the LLVM IR that has the option of defining a label (akin to `c` labels) as part of its constructor. The `builder.setInsertPoint` tells the LLVM engine where to insert the instructions next. Listing 8 shows the code.

Listing 8. Adding a basic block to main

```
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Support/IRBuilder.h"

int main()
{
  llvm::LLVMContext& context = llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("top", context);
  llvm::IRBuilder<> builder(context);

  llvm::FunctionType *funcType =
      llvm::FunctionType::get(builder.getInt32Ty(), false);
  llvm::Function *mainFunc =
      llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);

  llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
  builder.SetInsertPoint(entry);

  module->dump( );
}
```

Here's the output of Listing 8. Note that because the basic block for `main` is now defined, the LLVM dump now treats `main` as a method definition, not a declaration. Cool stuff!

```
; ModuleID = 'top'
define void @main() {
entrypoint:
}
```

Now, add the global hello-world string to the code. Listing 9 shows the code.

Listing 9. Adding the global string to the LLVM module

```
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Support/IRBuilder.h"

int main()
{
  llvm::LLVMContext& context = llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("top", context);
  llvm::IRBuilder<> builder(context);

  llvm::FunctionType *funcType =
      llvm::FunctionType::get(builder.getVoidTy(), false);
  llvm::Function *mainFunc =
      llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);

  llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
  builder.SetInsertPoint(entry);

  llvm::Value *helloWorld = builder.CreateGlobalStringPtr("hello world!\n");

  module->dump( );
}
```

In this output of Listing 9, note how the LLVM engine dumps the string:

```
; ModuleID = 'top'
@0 = internal unnamed_addr constant [14 x i8] c"hello world!\0A\00"
define void @main() {
entrypoint:
}
```

All you need now is to declare the `puts` method and make a call to it. To declare the `puts` method, you must create the appropriate `FunctionType*`. From your original Hello World code, you know that `puts` returns `i32` and accepts `i8*` as the input argument. Listing 10 shows the code to create the right type for `puts`.

Listing 10. Code to declare the puts method

```
  std::vector<llvm::Type *> putsArgs;
  putsArgs.push_back(builder.getInt8Ty()->getPointerTo());
  llvm::ArrayRef<llvm::Type*>  argsRef(putsArgs);

  llvm::FunctionType *putsType =
    llvm::FunctionType::get(builder.getInt32Ty(), argsRef, false);
  llvm::Constant *putsFunc = module->getOrInsertFunction("puts", putsType);
```

The first argument to `FunctionType::get` is the return type; the second argument is an `LLVM::ArrayRef` structure, and the last `false` indicates that no variable number of arguments follows. The `ArrayRef` structure is similar to a vector, except that it does not contain any underlying data and is primarily used to wrap data blocks like arrays and vectors. With this change, the output appears in Listing 11.

Listing 11. Declaring the puts method

```
; ModuleID = 'top'
@0 = internal unnamed_addr constant [14 x i8] c"hello world!\0A\00"
define void @main() {
entrypoint:
}
declare i32 @puts(i8*)
```

All that remains is to call the `puts` method inside `main` and return from `main`. The LLVM API takes care of the casting and all the rest: All you need to call `puts` is to invoke `builder.CreateCall`. Finally, to create the return statement, call `builder.CreateRetVoid`. Listing 12 provides the complete working code.

Listing 12. The complete code to print Hello World

```
#include "llvm/ADT/ArrayRef.h"
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/Function.h"
#include "llvm/BasicBlock.h"
#include "llvm/Support/IRBuilder.h"
#include <vector>
#include <string>

int main()
{
  llvm::LLVMContext & context = llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("asdf", context);
```

```
  llvm::IRBuilder<> builder(context);

  llvm::FunctionType *funcType = llvm::FunctionType::get(builder.getVoidTy(), false);
  llvm::Function *mainFunc =
    llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);
  llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
  builder.SetInsertPoint(entry);

  llvm::Value *helloWorld = builder.CreateGlobalStringPtr("hello world!\n");

  std::vector<llvm::Type *> putsArgs;
  putsArgs.push_back(builder.getInt8Ty()->getPointerTo());
  llvm::ArrayRef<llvm::Type*>  argsRef(putsArgs);

  llvm::FunctionType *putsType =
    llvm::FunctionType::get(builder.getInt32Ty(), argsRef, false);
  llvm::Constant *putsFunc = module->getOrInsertFunction("puts", putsType);

  builder.CreateCall(putsFunc, helloWorld);
  builder.CreateRetVoid();
  module->dump();
}
```

Conclusion

Other articles in this series

View more articles in the Create a working compiler with the LLVM framework series.

In this initial study of LLVM, you learned about LLVM tools like `lli` and `llvm-config`, dug into LLVM intermediate code, and used the LLVM API to generate the intermediate code for you. The second and final part of this series will explore yet another task you can use the LLVM for—adding an extra compilation pass with minimum effort.

Related topics

- Move beyond the basics of the LLVM in Create a working compiler with the LLVM framework, Part 2: Use clang to preprocess C/C++ code (Arpan Sen, developerWorks, June 2012). Put your compiler to work as you use the clang API to preprocess `C/C++` code as the LLVM compiler series continues.
- Take the official LLVM Tutorial for a great introduction to LLVM.
- See Chris Latner's chapter in *The Architecture of Open Source Applications* for more information on the development of LLVM.
- Learn more about two important LLVM tools: `llc` and `lli`.
- Read more about the LLVM assembly language in the LLVM Language Reference Manual.
- Check out its grammar file, Log of /llvm/trunk/utils/llvm.grm, for more information about the global string constant in LLVM.
- Learn more about the `getelementptr` instruction in "The Often Misunderstood GEP Instruction" document.
- Dig into the LLVM Programmer's Manual, an indispensable resource for the LLVM API.
- Read about the `llvm-config` tool for printing LLVM compilation options.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.
- Follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.