Paul Smith

How to get started with the LLVM C API

January 20, 2015 by Paul Smith

I enjoy making toy programming languages to better understand how compilers (and, ultimately, the underlying machine) work and to experiment with techniques that aren't in my repertoire. LLVM is great because I can tinker, and then wire it up as the backend to have it generate fast code that runs on most platforms. If I just wanted to see my code execute, I could get away with a simple hand-rolled interpreter, but having access to LLVM's JIT, suite of optimizations, and platform support is like having a superpower — your little toy can perform impressively well. Plus, LLVM is the foundation of things like Emscripten and Rust, so I like developing intuition about how new technologies I'm interested in are implemented.

I'm going to show how to use the LLVM API to programmatically construct a function that you can invoke like any other and have it execute directly in the machine language of your platform.

In this example, I'm going to use the C API, because it is available in the LLVM distribution, along with a C++ API, and so is the simplest way to get started. There are bindings to the LLVM API in other languages — Python, OCaml, Go, Rust — but the concepts behind using LLVM to generate code are the same across the wrapper APIs.

This example sort of skips to the middle phase of compiler construction. Assume the frontend (lexer, parser, type-checker) has built an AST and we're now walking it to emit the intermediate representation of the code for the backend to take and optimize and spit out machine code.

In this case, we'll just type out the straight-line procedural code for a simple function that would normally be dynamically cobbled together in a AST walker function, calling the LLVM API when it encounters certain nodes in the tree.

For the example, we'll build a simple adder function, which takes two integers as arguments and returns their sum, the equivalent of, in C:

```
int sum(int a, int b) {
    return a + b;
}
```

To be clear about what we are doing here: we are using LLVM to dynamically build an in-memory representation of this function, using its API to set up things like function entry and exit, return and parameter types, and the actual integer add instruction. Once this in-memory representation is complete, we can instruct LLVM to jump to it and execute it with arguments we supply, just as if it was a executable we had compiled from a language like C.

Click here to view the final code.

Modules

The first step is to create a module. A module is a collection of the global variables, functions, external references, and other data in LLVM. Modules aren't quite like, say, modules in Python, in that they don't provide separate namespaces. But they are the top-level container for all things built in LLVM, so we start by creating one.

```
LLVMModuleRef mod = LLVMModuleCreateWithName("my_module");
```

The string `"my_module"` passed to the module factory function is an identifier of your choosing.

Note that as you're navigating the LLVM C API documentation, different aspects are grouped together under different header includes. Most of what I'm detailing here, such as modules and functions, is contained in the `Core.h` header, but I'll include others as we move along.

Types

Next, I create the `sum` function and add it to the module. A function consists of:

- its type (return type),
- a vector of its parameter types, and
- a set of basic blocks.

I'll get to basic blocks in a moment. First, we'll handle the type and parameter types of the function — its prototype, in C terms — and add it to the module.

```
LLVMTypeRef param_types[] = { LLVMInt32Type(), LLVMInt32Type() };
LLVMTypeRef ret_type = LLVMFunctionType(LLVMInt32Type(), param_types, 2, 0);
```

```
LLVMValueRef sum = LLVMAddFunction(mod, "sum", ret_type);
```

LLVM types correspond to the types that are native to the platforms we're targeting, such as integers and floats of fixed bit width, pointers, structs, and arrays. (There's no platform-dependent `int` type like in C, where the actual size of the integer, 32- or 64-bit, depends on the underlying machine architecture.)

LLVM types have constructors, and follow the form "LLVM*TYPE*Type()". In our example, both the arguments passed to the sum function and the function's type itself are 32-bit integers, so we use `LLVMInt32Type()` for each.

The arguments to `LLVMFunctionType()` are, in order;

1. the function's type (return type),
2. the function's parameter type vector (the arity of the function should match the number of types in the array), and
3. the function's arity, or parameter count,
4. a boolean whether the function is variadic, or accepts a variable number of arguments.

Notice that the function type constructor returns a type reference. This reinforces the notion that what we did here is the LLVM equivalent of declaring a function prototype in C.

The third line in here adds the function type to the module, and gives it the name `sum`. We get a value reference in return, which can be thought of as a concrete location in the code (ultimately, memory) upon which to add the function's body, which we do below.

Basic blocks

The next step is to add a basic block to the function. Basic blocks are parts of code that only have one entry and exit point - in other words, there is no other way execution can go than by single stepping through a list of instructions. No if/else, while, loops, or jumps of any kind. Basic blocks are the key to modeling control flow and creating optimizations later on, so LLVM has first-class support for adding these to our in-progress module.

```
LLVMBasicBlockRef entry = LLVMAppendBasicBlock(sum, "entry");
```

Note the "append" in the name of the function: it's helpful to think of what we're doing as growing a running tally of chunks of code, and so our basic block is appended relative to the function we added to the module previously.

Instruction builders

This notion of a running tally fits with the instruction builder, which is how we add instructions to our function's one and only basic block.

```
LLVMBuilderRef builder = LLVMCreateBuilder();
LLVMPositionBuilderAtEnd(builder, entry);
```

Similar to appending the basic block to the function, we're positioning the builder to start writing instructions where we left off with the entry to the basic block.

LLVM IR

Sidebar: LLVM's main stock-in-trade is the LLVM intermediate representation, or IR. I've seen it referred to as a midway point between assembly and C. The LLVM IR is a very strictly defined language that is meant to facilitate the optimizations and platform portability that LLVM is known for. If you look at IR, you can see how individual instructions can be translated into the loads, stores, and jumps of the ultimate assembly that will be generated. The IR has 3 representations:

- as an in-memory set of objects, which is what we're using in this example,
- as a textual language like assembly,
- as a string of bytes in a compact binary encoding, called bitcode.

You may see clang or other tools emit LLVM IR as text or bitcode.

Back to our example. Now comes the crux of our function, the actual instructions to add the two integers passed in as arguments and return them to the caller.

```
LLVMValueRef tmp = LLVMBuildAdd(builder, LLVMGetParam(sum, 0), LLVMGetParam(sum, 1), "tmp");
LLVMBuildRet(builder, tmp);
```

`LLVMBuildAdd()` takes a reference to the builder, the two integers to add, and a name to give the result. (The name is required due to LLVM IR's restriction that all instructions produce intermediate results. This can further be simplified or optimized away by LLVM later, but while generating IR, we follow its strictures.) Since the numbers we wish to add are the arguments that were supplied to the function by the caller, we can retrieve them in the form of the function's parameters using `LLVMGetParam()`: the second argument to is the index of the parameter we seek from the function.

We call `LLVMBuildRet()` to generate the return statement and arrange for the temporary result of the add instruction to be the value returned.

Analysis & execution

That concludes the building instructions phase of creating our function; the module is now complete. The next phase of the example is setting it up for execution.

First, let's verify the module. This will ensure that our module was correctly built and will abort if we missed or mixed up any steps.

```
char *error = NULL;
LLVMVerifyModule(mod, LLVMAbortProcessAction, &error);
LLVMDisposeMessage(error);
```

LLVM provides either a JIT or an interpreter to execute the IR we've built. It will create a JIT if it can for the target platform, and fall back to an interpreter otherwise. In any case, the thing that will run our code is called the *execution engine*.

```
LLVMExecutionEngineRef engine;
error = NULL;
LLVMLinkInJIT();
```

```
LLVMInitializeNativeTarget();
if (LLVMCreateExecutionEngineForModule(&engine, mod, &error) != 0) {
    fprintf(stderr, "failed to create execution engine\n");
    abort();
}
if (error) {
    fprintf(stderr, "error: %s\n", error);
    LLVMDisposeMessage(error);
    exit(EXIT_FAILURE);
}
```

We could hard-code some integers to be summed, but it's easy enough to have our program receive them from the command line.

```
if (argc < 3) {
    fprintf(stderr, "usage: %s x y\n", argv[0]);
    exit(EXIT_FAILURE);
}
long long x = strtoll(argv[1], NULL, 10);
long long y = strtoll(argv[2], NULL, 10);
```

Now that we have two integers in the representation of our host language, we need to transform them into the analogous representation in LLVM. LLVM provides factory functions that convert values into the types we need to pass to our function:

```
LLVMGenericValueRef args[] = {
    LLVMCreateGenericValueOfInt(LLVMInt32Type(), x, 0),
    LLVMCreateGenericValueOfInt(LLVMInt32Type(), y, 0)
};
```

Now for the moment of truth: we can call our (JIT'd) function!

```
LLVMGenericValueRef res = LLVMRunFunction(engine, sum, 2, args);
```

We have a result, but it's still in LLVM-land. We recover it to a C type, the reverse operation from above, and print the sum:

```
printf("%d\n", (int)LLVMGenericValueToInt(res, 0));
```

And there we have it. We've programmatically constructed a function from the ground up, and had it run directly in machine code native to our platform. There is much more to LLVM, including control flow (eg., implementing if/else) and optimization passes, but we've covered the basics that would be in any LLVM-IR-to-code program.

Compiling

In order to compile our program, we need to reference the LLVM includes and link its libraries. Even though we've written a C program, the linking step requires the C++ linker. (LLVM is a C++ project, and the C API is a wrapper thereof.)

```
$ cc `llvm-config --cflags` -c sum.c
$ c++ `llvm-config --cxxflags --ldflags --libs core executionengine jit interpreter analysis native bitwriter --system-libs` sum.o -o sum
$ ./sum 42 99
141
```

Bitcode

One final thing. I mentioned previously that LLVM IR has three representations, including bitcode. Once you have a completed module, you can emit bitcode and write it out to a file.

```
if (LLVMWriteBitcodeToFile(mod, "sum.bc") != 0) {
    fprintf(stderr, "error writing bitcode to file, skipping\n");
}
```

From there, you can use tools to manipulate it, like `llvm-dis` to disassemble the bitcode into the textual LLVM IR assembly language.

```
$ llvm-dis sum.bc
$ cat sum.ll
; ModuleID = 'sum.bc'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"

define i32 @sum(i32, i32) {
entry:
  %tmp = add i32 %0, %1
  ret i32 %tmp
}
```

Source code of example

Here is the complete source of the program from above:

```
/**
 * LLVM equivalent of:
 *
 * int sum(int a, int b) {
 *     return a + b;
 * }
 */

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

```
#include <llvm-c/Analysis.h>
#include <llvm-c/BitWriter.h>

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char const *argv[]) {
    LLVMModuleRef mod = LLVMModuleCreateWithName("my_module");

    LLVMTypeRef param_types[] = { LLVMInt32Type(), LLVMInt32Type() };
    LLVMTypeRef ret_type = LLVMFunctionType(LLVMInt32Type(), param_types, 2, 0);
    LLVMValueRef sum = LLVMAddFunction(mod, "sum", ret_type);

    LLVMBasicBlockRef entry = LLVMAppendBasicBlock(sum, "entry");

    LLVMBuilderRef builder = LLVMCreateBuilder();
    LLVMPositionBuilderAtEnd(builder, entry);
    LLVMValueRef tmp = LLVMBuildAdd(builder, LLVMGetParam(sum, 0), LLVMGetParam(sum, 1), "tmp");
    LLVMBuildRet(builder, tmp);

    char *error = NULL;
    LLVMVerifyModule(mod, LLVMAbortProcessAction, &error);
    LLVMDisposeMessage(error);

    LLVMExecutionEngineRef engine;
    error = NULL;
    LLVMLinkInJIT();
    LLVMInitializeNativeTarget();
    if (LLVMCreateExecutionEngineForModule(&engine, mod, &error) != 0) {
        fprintf(stderr, "failed to create execution engine\n");
        abort();
    }
    if (error) {
        fprintf(stderr, "error: %s\n", error);
        LLVMDisposeMessage(error);
        exit(EXIT_FAILURE);
    }

    if (argc < 3) {
        fprintf(stderr, "usage: %s x y\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    long long x = strtoll(argv[1], NULL, 10);
    long long y = strtoll(argv[2], NULL, 10);

    LLVMGenericValueRef args[] = {
        LLVMCreateGenericValueOfInt(LLVMInt32Type(), x, 0),
        LLVMCreateGenericValueOfInt(LLVMInt32Type(), y, 0)
    };
    LLVMGenericValueRef res = LLVMRunFunction(engine, sum, 2, args);
    printf("%d\n", (int)LLVMGenericValueToInt(res, 0));

    // Write out bitcode to file
    if (LLVMWriteBitcodeToFile(mod, "sum.bc") != 0) {
        fprintf(stderr, "error writing bitcode to file, skipping\n");
    }

    LLVMDisposeBuilder(builder);
    LLVMDisposeExecutionEngine(engine);
}
```

See the GitHub repo for the Makefile and details on how to build the example on your machine.

Recent posts

- T-shirt retirement
- Chicago wards & precincts shapefiles in 2015

About

Paul Smith is a software engineer. More …

Elsewhere

- Threads
- Mastodon
- Bluesky
- Flickr