LLVM Generation

Lecture #03

Charles Averill

Introduction to Compiler Design
The University of Texas at Dallas

Fall 2023

Recap of Class Format

- "How should I utilize the lectures to help build the compiler?" - I suggest taking notes in the form of comments in your own codebase that highlight the ideas behind the changes we're making in class.
- "I'm worried that my implementation might be too similar to ECCO" - I recognize that if you're studying the code in the lectures, your compiler will look very similar, and the logic and structure will likely be nearly identical. This is okay! I just don't want you to copy and paste from the ECCO codebase. If I say something is an "implementation detail", I really don't mind if you copy it straight from ECCO, because it's not important to the concepts of compiler design.
- "I'm worried the code I write now will interfere with the code I write later, or that I'll have to do large refactoring" - This is a good worry to have, because I guarantee it will happen! This is the consequence of starting out with a small project and expanding it to support more features. We will do our best to properly think ahead, but there will be many occasions where you might have to refactor old code. This builds understanding of the compiler and all of its interlaced parts.

My Philosophy on Teaching

I learned compiler design by reading ACWJ on my free time, and building Purple using its principles. This allowed me to process the information at my own speed, and make improvements where I saw fit. I greatly enjoyed this unstructured approach to learning. But not everybody does!

I would love it if many people took the same path I did, and I would also love it if many people were interested in just listening to me lecture about how I built my compilers. The reality is that the majority of people would not like either of these approaches. This course's structure – lecturing about the big changes to our compiler over time and leaving the rest up to y'all – is a combo of these two approaches.

I was speaking about my first week teaching this course with a professor of mine, and he told me that when an instructor teaches a course for the first time, they usually learn more than the students of the course! So far, this is absolutely true. And I'm having a blast!

The Current State of the Compiler

We can parse binary expressions and print them out with a toy interpreter we've designed.

```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/parse_test_1
2 + 3 * 5 - 8 / 3
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/parse_test_1
-----RUN-----
15
```

```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/parse_test_2
13 -6+  4*
5
      +
08 / 3
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/parse_test_2
-----RUN-----
29
```

You didn't sign up for Practical Interpreter Design. Let's write a compiler!

The Goal

Our first goal for ECCO is to compile our input files (containing binary expressions) down to LLVM. Then, we will pass the LLVM into `clang`, the LLVM compiler, which will give us a binary.

That's not cheating, it's what LLVM is designed for! LLVM's strength is that it is a "pseudo-assembly language", so it abstracts features of popular CPU architectures. That way, when we compile to LLVM, the `clang` compiler can take our LLVM and convert it to x86, ARM, MIPS, etc. while handling stuff like register allocation for us! That's so cool!

We will not learn LLVM all at once. Over the course of the semester we will learn LLVM as the complexity of our compiler grows. However, there are a few core tenets of the language that we'll go over now. If you aren't familiar with assembly code or basic computer architecture, please watch Lecture 0.

LLVM Basics

- There are no registers in LLVM. There are infinite "virtual registers". When virtual registers are defined, they can only be declared with '=' once. After that, you can update their value with the `store` instruction. Our compiler will name virtual registers with numbers (although they could be named with letters and some punctuation).
- LLVM is strongly-typed, so every instruction will force you to give it the type of your argument. Integer types are defined with bit widths, e.g. `i1, i4, i8, i32, i64`. You can actually have any bit width here, up to $2^{32} - 1$
- LLVM lets you define functions and hook into the C standard library. This helps a lot, as we won't have to explicitly deal with stack memory when calling functions (although we will talk about what's happening when the time comes)

Let's look at some LLVM (Program Preamble)

```
; ModuleID = 'examples/test1'
source_filename = "examples/test1"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-↩
    f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@print_int_fstring = private unnamed_addr constant
    [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
```

- Comments start with ;
- `target datalayout` and `target triple` are pieces of information we give to clang to describe the target machine we want to compile to. These describe things like the memory architecture, CPU, and OS

Let's look at some LLVM (Program Preamble)

```
; ModuleID = 'examples/test1'
source_filename = "examples/test1"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-↩
    f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@print_int_fstring = private unnamed_addr constant
    [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
```

- The `@` prefix declares a global variable. We're using it here to define a format string for printing integers. `[4 x i8]` is describing the data type (four characters, a.k.a. a string). `%d` should be familiar if you've written fstrings. `\0A` is the newline character and `\00` is the null-terminator character

Let's look at some LLVM (Program Preamble)

```
; ModuleID = 'examples/test1'
source_filename = "examples/test1"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-↩
    f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@print_int_fstring = private unnamed_addr constant
    [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
```

- Finally we define our main function. The `define` keyword is obvious enough. You don't need to worry about `dso_local` yet. `i32` says our function should return an integer that fits into 32 bits. `#0` is a way to apply various attributes to `main` that we will see later on.

Let's look at some LLVM (Program Guts)

$$2 + 3 \times 5$$

```
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 2, i32* %3, align 4
    store i32 3, i32* %2, align 4
    store i32 5, i32* %1, align 4
    %4 = load i32, i32* %2, align 4
    %5 = load i32, i32* %1, align 4
    %6 = mul nsw i32 %4, %5
    %7 = load i32, i32* %3, align 4
    %8 = add nsw i32 %7, %6
    call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x    ↩
        i8], [4 x i8]* @print_int_fstring , i32 0, i32 0), i32 %8)
    ret i32 0
}
```

Let's look at some LLVM (Loading Constants)

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 2, i32* %3
store i32 3, i32* %2
store i32 5, i32* %1
%4 = load i32, i32* %2
%5 = load i32, i32* %1
```

The first thing we need to do when compiling binary expressions is allocating space for each number we want to compute. The `alloca` instruction allocates space for data **on the stack** given a data type, and ensures the address will be a multiple of 4 (we might talk more about stack alignment later, but even if you get this wrong, `clang` can fix your alignment errors usually). `alloca` returns a pointer, which we store into registers 1, 2, 3 (local variables are denoted with %).

Let's look at some LLVM (Loading Constants)

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 2, i32* %3
store i32 3, i32* %2
store i32 5, i32* %1
%4 = load i32, i32* %2
%5 = load i32, i32* %1
```

Next, we can `store` values into the space we just allocated. Again, we give it our datatype `i32`, then we tell the instruction that we're storing it into an i32 pointer that exists at a certain register.

Finally, we `load` the values from pointers into virtual registers. This allows us to perform computations on the data. This seems very verbose for just storing values, but it's a good starting point for more complex behavior later.

Let's look at some LLVM (Arithmetic)

```
%6 = mul nsw i32 %4, %5
%7 = load i32, i32* %3
%8 = add nsw i32 %7, %6
```

We're now going to multiply the contents of registers 4 and 5, which contain the values 3 and 5, respectively. The `nsw` keyword stands for "no signed wrap". When we use this keyword, we enable some error checking for integer overflow. If overflow occurs, the output of the `mul` instruction becomes a "poison value". Poison values have an interesting decision; they do not evoke errors when passed into most instructions, but indicate that something has gone wrong. Generally, they prevent undefined behavior from occurring. The output of `mul` is stored into register 6, then added to the value loaded from register 3. The value of the binary expression is now present in register 8.

Let's look at some LLVM (Printing)

```
call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 ↩
    x i8]* @print_int_fstring , i32 0, i32 0), i32 %8)
ret i32 0
```

We're now going to hook into the C standard library to call `printf`. We do this with the `call` instruction. The first two arguments, `i32 (i8*, ...)`, provide the output and input types of the function to call. We then provide the function's name and its arguments, also annotated with type signatures. Don't worry about `getelementptr inbounds`, that's just more undefined behavior prevention.

Finally, we `return 0` to indicate that our `main` function executed successfully.

Recap

So we've looked at an LLVM program. Our goal is to traverse our AST and generate programs like this. Let's take a look at the changes we need to make to ECCO so that we can accomplish this.

If we take a look at `ecco/ecco.py` we can see that all of our interpreter code has been replaced by a call to `generate_llvm()` in `ecco/generation/translate.py`.

One thing to mention is that ACWJ, the project I originally based Purple and ECCO on, compiles to x86. LLVM and x86 are fairly different in the structure of their programs, so our code generation stage is going to be different than ACWJ. One primary difference you would see at the end of this lecture would be that ACWJ has to manage register usage between x86's r8, r9, r10, and r11 registers, while LLVM has infinite virtual registers.

LLVM Generation (`ecco/generation/translate.py`)

First, we have our preamble generation function, `llvm_preamble()`. The preamble is mostly static for now, so this isn't too complicated.

The next step is to allocate the stack space for all of our constants. We *could* do this every time we need a new constant, but we can make it easier to optimize if we perform a stack usage detection algorithm. Let's take a look at `determine_binary_expression_stack_allocation()`.

Stack Allocation

We're going to recursively traverse our AST to determine the necessary stack allocation statements required to store the constants in our binary expressions.

I've defined a new `LLVMStackEntry` class in `ecco/generation/llvmstackentry.py` that stores a register number and the byte width of our numbers. We only have one kind of number right now (`i32`) but this class will help later on when we add some more number types (we will eventually add shorts, chars, ints, and longs). Our stack allocation algorithm will return a list of LLVMStackEntries.

At each step, if the root node has any children, we will recursively call `determine_binary_expression_stack_allocation()` on each child and append their results together. Otherwise, we must be at a terminal node (so far, these can only be `i32` constants), so we'll store its data into an LLVMStackEntry, and return it inside of a length-1 list.

Continuing our LLVM Generation

Back in `generate_llvm()`, we pass the output of `determine_binary_expression_stack_allocation(root)` into an LLVM generation function called `llvm_stack_allocation()` that traverses a list of LLVMStackEntries and generates the LLVM `alloca` statements we saw in our example program.

After this, we call `ast_to_llvm()`. This is the analogue to the tiny interpreter function we wrote in the last lecture. The `left_vr` and `right_vr` LLVMValues are going to store the virtual register numbers of the outputs of sub-ASTs.

For example, in the AST for the expresion `2 + 3`, the corresponding `left_vr` and `right_vr` register numbers for the `+` node will be the register numbers that the constants `2` and `3` were stored in.

Continuing our LLVM Generation

After recursively computing the LLVM of our subtrees (or not, if the current node is a leaf node), we'll check whether the current node is a binary arithmetic operator, or a terminal node representing a constant.

In the case of a binary operator, we will want to ensure that `left_vr` and `right_vr` are loaded. Remember that when we `alloca` and then `store` a constant into a register, we still have to `load` it into a new register before we can use it. However, when we assign the output of an arithmetic operation to a virtual register, we can assume that it is already loaded. Let's take a look at the `llvm_ensure_registers_loaded()` function.

After we know our `left_vr` and `right_vr` have loaded register values, we can generate our arithmetic statements in `llvm_binary_arithmetic()`, which abstracts the logic of checking for various operators.

In the case of a constant, we will simply store the constant into one of our allocated registers.

Finishing our LLVM Generation

Finally, we will call `llvm_print_int()`, which prints the contents of a virtual register using `printf` and the format string we defined in the preamble.

After we've generated our actual program contents, we need to generate a postamble with `llvm_postamble()` that closes our `main` function with a `ret i32 0`, hooks into C by defining a reference to `printf`, then defines the attributes I mentioned a while ago.

And that's it! We can drop arithmetic statements into our compiler now and compile them to valid LLVM. We can compile and run this LLVM by running `clang programname.ll -o programname && ./programname` (you should add this to your `scripts`).

Summary

This was a really complicated lecture. Likely the hardest we will have all semester! The next hardest will be adding functions, which we'll handle a few lectures from now. After that, it should be smooth sailing.

Until then, please come to office hours if you have any trouble or are feeling overwhelmed by the workload. Again, now that we have this groundwork out of the way, it becomes much easier to expand our compiler. Lecturing is only half of my job here, I'd like to make sure all of you feel comfortable with, and understand well the compiler updates.

Remember that to earn the PCD certification, there are a set number of required features your compiler must have. After implementing what we talked about from this lecture, we are roughly 40% done with required topics! After this, the way we introduce things will open the door for you to choose your own syntax for many common language features. Get excited and start dreaming up ideas!