LLVM Study Note

Henry He 2023

What's LLVM?

> LLVM is a set of compiler and tool-chain technologies that can be used to develop a front-end for any programming language and a back-end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes. The name LLVM originally stood for **Low Level Virtual Machine,** though the project has expanded and the name is no longer officially an initialization.

This note mainly refers to the book LLVM IR Tutorial.

Data Flow

data representation

We are interested in data representation at the assembly language level.

Generally, there are three types of data: **global data / stack data / heap data**.

But heap data can not exist independently, there must be a reference for them.

So in the program, the data we can use directly is actually those in the **global data section / stack / register**.

register and stack

Operating data in register is much more efficient than in stack.

Sometimes the register count is not enough or we need to process memory address, except in these cases, using register is a much better choice.

That's why LLVM introduced the concept of virtual registers. In LLVM, we can use virtual registers as if unlimitedly. If real register count is not enough, LLVM will put the data in the stack.

Here is an example of register usage in LLVM IR: ('%' is important)

```
%variable_name = add i32 1, 2
```

If we want to use data in stack, `alloca` is all you need.

example:

```
%stack_data = alloca i32
```

use of data

LLVM IR view both **global** variable and **stack** variable as pointer `ptr`.

So the usage like this is wrong:

```
@global_variable = global i32 0
%1 = add i32 1, @global_variable ; Error!
```

To operate `ptr` value, we have to use `load` / `store` command like this:

```
%1 = load i32, ptr @global_variable
%2 = add i32 1, %1 ; Success!
```

SSA

LLVM IR obeys to SSA (**Static Single Assignment**) strategy strictly, which can make its optimization easier.

types

The basic data type in LLVM IR includes: `void` / `iN` / `ptr`

One thing to note is that in LLVM, the symbol of integer (`iN`) is distinguished by instruction rather than type.

For example:

```
%1 = udiv i8 -6, 2 ; Get (256 - 6) / 2 = 125

%2 = sdiv i8 -6, 2 ; Get (-6) / 2 = -3
```

To declare an array, you may do like this:

```
%1 = alloca [4 x i32]

@global_array = global [4 x i32] [i32 0, i32 1, i32 2, i32 3]
; with initialization
```

Structure type is also easy to use:

```
%MyStruct = type { i32, i8 }

@global_structure = global %MyStruct { i32 1, i8 0 }
             ; or = global { i32, i8 } { i32 1, i8 0 }
```

If we want to get data from a pointer (maybe array or structure) , LLVM provides `getelementptr` command. Here is an example:

```
%my_y_ptr = getelementptr %MyStruct, ptr %my_structs_ptr,  i64 2,  i32 1
;                pointing data type,       operating ptr, offset,  index
; corresponding C code: int *my_y_ptr = my_structs_ptr[2].y;
```

`getelementptr` view all `ptr` as an array pointer, corresponding to C feature: array_name == &array_name[0].

Control Flow

control statement

To implement control statements (like branch&loop) in most high-level languages, four things are crucial:

1. label
2. unconditional jump
3. compare instruction
4. conditional jump

In LLVM IR, label used like this:

```
start:
    %i_value = load i32, ptr %i
```

compare instruction:

```
%comparison_result = icmp uge i32 %a, %b
; uge means for "unsigned greater than"
; common cmp condition: eq, ne / ugt , uge , ult , ule / sgt , sge , slt , sle
```

jump:

```
br i1 %comparison_result, label %A, label %B
; conditional jump, if true then A, else B
br label %start
; unconditional jump
```

basic block

> A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) for the function. Each basic block may optionally start with a label, contains a list of instructions, and **ends with a terminator instruction** (such as a branch or function return). If an explicit label name is not provided, a block is assigned an implicit numbered label.

So here is an example LLVM code for loop statement:

```
    %i = alloca i32                                 ;int i = ...
    store i32 0, ptr %i                             ; ... = 0
    br label %start
start:
    %i_value = load i32, ptr %i
    %comparison_result = icmp slt i32 %i_value, 4   ; Test if i < 4
    br i1 %comparison_result, label %A, label %B
A:
    ; Do something A
    %1 = add i32 %i_value, 1                        ; ... = i + 1
    store i32 %1, ptr %i                            ; i = ...
    br label %start
```

```
B:
    ; Do something B
```

select and phi

Rust has a very convenient feature: view `if...else` as an expression, so does LLVM:

```
%result = icmp sgt i32 %x, 0    ; x > 0
%y = select i1 %result, i32 1, i32 2
; if result true then y = 1, else y = 2
```

`select` can help avoid a lot of trouble caused by SSA.

`phi` is just a stronger version of `select`, like `match` expression in Rust, determining option through control flow.

Here is the example code using `phi` :

```
define void @foo(i32 %x) {
    %result = icmp sgt i32 %x, 0
    br i1 %result, label %btrue, label %bfalse
btrue:
    br label %end
bfalse:
    br label %end
end:
    %y = phi i32 [1, %btrue], [2, %bfalse]
    ;       type [val, label], ...
    ; Do something with %y
    ret void
}
```

function

LLVM IR supports function very well, almost indistinguishable from high-level languages.

Example code:

```
define i32 @foo(i32 %a) {
    ; ...
    ret i32 0
}

define void @bar() {
    %1 = call i32 @foo(i32 1)
    ; ...
}
```