UE19CS351 - Compiler Design

Session: Jan 2022 – May 2022

MANUAL

Semester:	VI
Problem Statement :	Introduction to LLVM IR
Course Anchor :	Prof. Preet Kanwal
Teaching Assistant :	Anirudh H M

Table of Contents

1 Intro

- **LLVM intermediate representation code** (LLVM IR) is a SSA-based universal intermediate code representation, used for platform-independent optimization and code generation by the LLVM Compiler Infrastructure

- LLVM intermediate code may be represented in one of three ways, each of which is used for different

  purposes

  1. As an in-memory representation of code for general compilers (**in-memory data structures**)

  2. As an on-disk representation of code for JIT compilers (**LLVM bitcode**)

  3. As a human-readable assembly language (**LLVM human-readable assembly**)

- The following terms will be used in this document -

  Bitcode The LLVM IR code, represented in machine readable format

  LLVM IR The LLVM IR code, represented in human readable format

  Machine code Platform-specific code which can be executed by the CPU

- This document is a brief description of the LLVM language. The formal language description can be found at https://llvm.org/docs/LangRef.html

2 Syntax elements

These are the textual elements of IR code

2.1 Comments

In LLVM IR, comments begin with `;` and go up to the end of the line

2.2 Identifiers

- There are two types of identifiers - global and local

  Global identifiers Begin with `@` character prefix

  Local identifiers Begin with `%` character prefix

  Prefixes are used to avoid clashes between identifier names and reserved words

- There are three different identifier formats -

  Named values String of characters with scope prefix (global or local). Names must adhere to regex `[-a-zA Z$._][-a-zA-Z$._0-9]*`. Ex: `@myvar`

  Unnamed values Unsigned numeric value with local scope prefix (cannot be global). These must be in increasing order, starting from 1. Ex: `%1`

  Constants Depends on the token. See constants

- Unnamed temporaries are created automatically during IR code generation when the result of an operation is not assigned to a named value. They are numbered sequentially, using a per-function incrementing counter starting from 0 (i.e, first temp variable in function is `%1`, second is `%2` and so on)

3 Semantic elements

These are logical elements of the IR representation, which are used to generate machine code

3.1 Global variables

- Define regions of memory allocated at compile time
- Must be initialized with a value, unless the variable is defined in another module
- Ex: `@G = external global i32 0`
  - `@G` : Name of global variable, prefixed with `@`
  - `external global` : Qualifiers
  - `i32` : Data type (32 bit integer)
  - `0` : Value of variable

3.2 Functions

- **Function declaration** consist of the `declare` keyword followed by
  - Return type
  - Function name
  - List of arguments (possibly empty)

  Along with these, several other optional qualifiers may be specified
- **Function definition** consist of the `define` keyword followed by
  - Return type
  - Function name
  - List of arguments (possibly empty)
  - Function body as a list of **basic blocks**, enclosed by curly braces

  Along with these, several other optional qualifiers may be specified (before the function body)

• Function body consists of a list of basic blocks, forming the **CFG** for the function. Each basic block contains a list of instructions and ends with a terminator instruction (branch or function return)

• Ex:

```
declare i32 @foo(i32)

define i32 @foo(i32 arg) {
  ; Convert [13 x i8]* to i8*...
  ; getelementptr is a LLVM keyword
  %.str = constant [13 x i8] c"hello world\0A\00"
  %cast210 = getelementptr [13 x i8], [13 x i8]* %.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  ; puts is declared externally
  call i32 @puts(i8* %cast210)
  ret i32 arg
}
```

3.3 Aliases

• Aliases create a new symbol for an existing position in the symbol table
• Aliases can be created for either a global value or a constant expression

• Ex: `@aliasname = alias i32, i32* @aliasee`

– `@aliasname` : Name of the alias

– `i32` : Type of aliasee (value to which alias is being created)

– `@aliasee` : Name of the aliasee

3.4 Constants

The basic constants available in LLVM are -

Boolean constants `true` and `false`

Integer constants Standard integers (positive and negative)

Floating-point constants Standard floating point values in decimal notation (123.456), exponential notation (1.23456e+2) or IEEE hexadecimal notation

Null pointer constant `null` is a null pointer constant

3.5 Types

LLVM has a strong type system which helps in performing optimizations on the intermediate code. Some of the important ones are -

Void Has no value or size. Represented by the `void` keyword

Function Represents a function signature, consisting of a return type and a list of parameter types. For example, `i32 (i8, i8, float)` is a function type that takes two 8 bit integers and a float, and returns a 32 bit integer

Integer Arbitrary width integer, represented as `i<n>`, where `n` is the number of bits in the representation

Float Floating point value. Some variants are `half` (16 bit float), `float` (32 bit float) and `double` (64 bit float)

Pointer Used to store references to memory locations. It may be specified as `<type>*` (pointer to memory storing value of `type`), or as `ptr`, which is an opaque pointer (type unspecified)

Aggregate types Derived types that contain multiple elements of simple types. Includes arrays, structures etc

3.6 Instructions

The LLVM instruction set consists of several classes of instructions. The classes, and some important instructions from each class, are described below -

Terminators Instruction that ends a basic block. They return a `void` value, and are used for control flow. Ex: `ret`, `br`, `switch`, `resume`

Unary Take a single operand. The only unary instruction is `fneg`

Binary Take two operands of the same type, execute an operation and produce a single value, which has the same type as the operands. Ex: `add`, `sub`, `mul`, `udiv`

Bitwise binary Same as binary, but perform bitwise operations on the operands. Ex: `shl`, `shr`, `and`, `or`

Vector Operations on vectors that are target-independent (support all platforms). Ex: `extractelement`, `insertelement`, `shufflevector`

Aggregate Operations on aggregate types. Ex: `extractvalue`, `insertvalue`

Memory access and allocation Operations to read, write and allocate memory. Ex: `alloca`, `load`, `store`
Note: LLVM manages all memory allocation and management using instructions, and does not represent memory locations in SSA form

Conversion Perform bit-level transformations to convert values from one type to another. Ex: `trunc`, `zext`, `sext`, `ptrtoint`

Miscellaneous Instructions that do not fall into any of the previous categories. Ex: `icmp`, `fcmp`, `phi`

3.7 Intrinsic functions

- Intrinsic functions are functions that are built into the compiler. These functions are implemented in an optimized way by the compiler
- Intrinsic functions may be introduced into code in two ways -
    1. If an intrinsic function is directly called in the source code, it is emitted as-is in the LLVM IR (requires compiler frontend support)
    2. Certain operations may be replaced by intrinsic functions during the LLVM optimization process
- All intrinsic functions are named with a prefix of `llvm.`

Note: Almost all new functionality to the LLVM compiler backend are implemented as intrinsic functions, and are converted to instructions only if needed .

4 Module structure

- Programs are composed of `Module`'s. A `Module` is formed from a single translation unit from the input programs

- Each module consists of
  - Functions
  - Global variables
  - Symbol table entries

  Of these, functions and global variables are considered as **global values**, which are represented by a pointer to a memory location

- Modules are combined by the **LLVM linker**, which merges global value definitions and symbol table entries