LLVM IR

Hosein Ghahramanzadeh

LLVM IR

- LLVM Intermediate Representation
- Better than assembly code
  - Independent of the architecture
    - No headaches reading the x86 code
    - Proper optimizations already implemented
  - Type safety
  - More human readable

LLVM IR Types

- We will discuss only a few of the LLVM IR types, full documentation can be found here.
  - `void` Does not represent a value and has the size 0.
  - `iN` An integer that is N bits long.
    - N can be anywhere from 1 bit to 2^{23} - 1.
    - `i1` A 1 bit integer
    - `i8` An 8 bit integer
    - `i32` A 32 bit integer
    - `i1000` A 1000 bit integer.
    - Is it unsigned?

LLVM IR Types

- Floating point types.
  - `half` A 16 bit floating point .
  - `float` A 32 bit floating point.
  - `double` A 64 bit floating point.
  - More specific floating point types here.
- Array type `[<count> x <type>]` (and yes that is an ‘x’)
  - `[14 x i32]` An array of 14 elements of type `i32`.
  - `[1 x float]` An array of one element of type `float`.

LLVM IR Types

- Function types
  - `i32 (i32)` A function that takes a `i32` and returns a `i32`.
  - `double (float, ...)` A vararg function that takes at least a floating point and returns a `double`.
  - Further details can be found here.
- Pointer types
  - `i32*` A pointer to a 32 bit integer.
  - `i32 (i32)*` A pointer to a function that takes a `i32` and returns a `i32`.
  - `[14 x i32]*` A pointer to an array of 14 elements of type `i32`.
- Label type, represents a code label.
  - `here:` Label at the code.
  - Labels must be placed after a terminator instruction or at the starting of a block(will be explained further later).

LLVM IR Identifiers

- There are two kinds of identifiers
  - Local identifier names start with %
    - Are only available in the scope.
  - Global identifier names start with @
    - Can be functions or global variables
- Identifiers help handle stack.
  - No need to implement stack.
- Identifiers should fit in the following regex.
  - `[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*`
- Unnamed value are represented by unsigned integer names.
  - `%1`, `@2`

LLVM IR Functions

- Functions can be defined and declared.
  - The definition specifies what the function does.
  - The declaration specifies what the function signature is.
- A function definition is made of basic execution blocks.
  - These blocks define the control flow graph.
  - Each block may start with a label.**
- Simple function definition
  - ```
    define <type> @<name> (<type> <name>, …)
    {
    }
    ```
  - Argument list can be empty.
- Read further here.
    ```

LLVM IR Instructions

- Again as with the case of the types, we are not going to discuss all the LLVM IR instructions, a full documentation can be found here.
  - `ret <type> <value>` Return a value from inside a function.
    - `ret void` Return for void return type functions.
    - `ret i32 0` Return zero in a function that returns 32 bit integers.
  - `br i1 <cond>, label <iftrue>, label <iffalse>` Conditional branch instruction.
    `br label <dest>` Unconditional branch instruction.
    - Infinite loop:
      ```
      here:
      br label %here
      ```
    - Infinite loop (since the condition is a constant 1, it is always true, and branches to %here):
      ```
      here:
      br i1 1, label %here, label %there
      there:
      ```

LLVM IR Instructions

- Read further on flow control instructions here.
- Binary operations
  - Require two operands of the same type.
  - `<result> = add <type> <op1>, <op2>` Adds two operands and the yields the result with type ty (for integers).
  - `<result> = sub <type> <op1>, <op2>` Subtracts op1 from op2 (for integers).
  - `<result> = mul <type> <op1>, <op2>` Multiplies op1 and op2 (for integers).
  - `<result> = udiv <type> <op1>, <op2>` Divedes op1 by op2 (for unsigned integer).
  - `<result> = sdiv <type> <op1>, <op2>` Divedes op1 by op2 (for signed integer).
  - Read further here.
- Memory access instruction
  - You can read further here.

LLVM IR Calling a Functions

- Call instructions calls a function
  - `%<name> = call @<name> (<name>, …)`
  - Arguments are passed inside the parentheses.
  - Read further on call instruction here.
- Compare instructions
  - These instructions compare two values and yield `i1`.
  - `%<result> = icmp <cond> <type> <op1>, <op2>` Compares two operands with type, <type>.
    - The compare condition is specified by `<cond>`.
    - The result is 1 on condition being correct, and 0 on condition being false.
    - Read more here.

Entry Point

- The execution of the program starts from the entry point.
- Each executable should have an entry point.
- Main function is the entry point on LLVM IR.
  - ```
    define i32 @main() {
        ret i32 0
    }
    ```
    ```