Introducing LLVM Intermediate Representation

In this article by **Bruno Cardoso Lopez** and **Rafael Auler**, the authors of Getting Started with LLVM Core Libraries, we will look into some basic concepts of the LLVM intermediate representation (IR).

(For more resources related to this topic, see here.)

LLVM **IR** is the backbone that connects frontends and backends, allowing LLVM to parse multiple source languages and generate code to multiple targets. Frontends produce the IR, while backends consume it. The IR is also the point where the majority of LLVM target-independent optimizations takes place.

Overview

The choice of the compiler IR is a very important decision. It determines how much information the optimizations will have to make the code run faster. On one hand, a very high-level IR allows optimizers to extract the original source code intent with ease. On the other hand, a low-level IR allows the compiler to generate code tuned for a particular hardware more easily. The more information you have about the target machine, the more opportunities you have to explore machine idiosyncrasies. Moreover, the task at lower levels must be done with care. As the compiler translates the program to a representation that is closer to machine instructions, it becomes increasingly difficult to map program fragments to the original source code. Furthermore, if the compiler design is exaggerated using a representation that represents a specific target machine very closely, it becomes awkward to generate code for other machines that have different constructs.

This design trade-off has led to different choices among compilers. Some compilers, for instance, do not support code generation for multiple targets and focus on only one machine architecture. This enables them to use specialized IRs throughout their entire pipeline that make the compiler efficient with respect to a single architecture, which is the case of the Intel C++ Compiler (*icc*). However, writing compilers that generate code for a single architecture is an expensive solution if you aim to support multiple targets. In these cases, it is unfeasible to write a different compiler for each architecture, and it is best to design a single compiler that performs well on a variety of targets, which is the goal of compilers such as GCC and LLVM.

For these projects, called *retargetable compilers*, there are substantially more challenges to coordinate the code generation for multiple targets. The key to minimizing the effort to build a retargetable compiler lies in using a common IR, the point where different backends share the same understanding about the source program to translate it to a divergent set of machines. Using a common IR, it is possible to share a set of target-independent optimizations among multiple backends, but this puts pressure on the designer to raise the level of the common IR to not overrepresent a single machine. Since working at higher levels precludes the compiler from exploring target-specific trickery, a good retargetable compiler also employs other IRs to perform optimizations at different, lower levels.

The LLVM project started with an IR that operated at a lower level than the Java bytecode, thus, the initial acronym was Low Level Virtual Machine. The idea was to explore low-level optimization opportunities and employ link-time optimizations. The link-time optimizations were made possible by writing the IR to disk, as in a bytecode. The bytecode allows the user to amalgamate multiple modules in the same file and then apply interprocedural optimizations. In this way, the optimizations will act on multiple compilation units as if they were in the same module.

LLVM, nowadays, is neither a Java competitor nor a virtual machine, and it has other intermediate representations to achieve efficiency. For example, besides the LLVM IR, which is the common IR where target-independent optimizations work, each backend may apply target-dependent optimizations when the program is represented with the *MachineFunction* and *MachineInstr* classes. These classes represent the program using target-machine instructions.

On the other hand, the *Function* and *Instruction* classes are, by far, the most important ones because they represent the common IR that is shared across multiple targets. This intermediate representation is mostly target-independent (but not entirely) and the *official* LLVM intermediate representation. To avoid confusion, while LLVM has other levels to represent a program, which technically makes them IRs as well, we do not refer to them as LLVM IRs; however, we reserve this name for the official, common intermediate representation by the *Instruction* class, among others. This terminology is also adopted by the LLVM documentation.

The LLVM project started as a set of tools that orbit around the LLVM IR, which justifies the maturity of the optimizers and the number of optimizers that act at this level. This IR has three equivalent forms:

- An in-memory representation (the *Instruction* class, among others)
- An on-disk representation that is encoded in a space-efficient form (the bitcode files)
- An on-disk representation in a human-readable text form (the LLVM assembly files)

LLVM provides tools and libraries that allow you to manipulate and handle the IR in all forms. Hence, these tools can transform the IR back and forth, from memory to disk as well as apply optimizations, as illustrated in the following diagram:

Understanding the LLVM IR target dependency

The LLVM IR is designed to be as target-independent as possible, but it still conveys some target-specific aspects. Most people blame the C/C++ language for its inherent, target-dependent nature. To understand this, consider that when you use standard C headers in a Linux system, for instance, your program implicitly imports some header files from the *bits* Linux headers folder. This folder contains target-dependent header files, including macro definitions that constrain some entities to have a particular type that matches what the **syscalls** of this kernel-machine expect. Afterwards, when the frontend parses your source code, it needs to also use different sizes for *int*, for example, depending on the intended target machine where this code will run.

Therefore, both library headers and C types are already target-dependent, which makes it challenging to generate an IR that can later be translated to a different target. If you consider only the target-dependent, C standard library headers, the parsed AST for a given compilation unit is already target-dependent, even before the translation to the LLVM IR. Furthermore, the frontend generates IR code using type sizes, calling conventions, and special library calls that match the ones defined by each target ABI. Still, the LLVM IR is quite versatile and is able to cope with distinct targets in an abstract way.

Exercising basic tools to manipulate the IR formats

We mention that the LLVM IR can be stored on disk in two formats: bitcode and assembly text. We will now learn how to use them. Consider the *sum.c* source code:

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

To make Clang generate the bitcode, you can use the following command:

```
$ clang sum.c -emit-llvm -c -o sum.bc
```

To generate the assembly representation, you can use the following command:

```
$ clang sum.c -emit-llvm -S -c -o sum.ll
```

You can also assemble the LLVM IR assembly text, which will create a bitcode:

```
$ llvm-as sum.ll -o sum.bc
```

To convert from bitcode to IR assembly, which is the opposite, you can use the disassembler:

```
$ llvm-dis sum.bc -o sum.ll
```

The *llvm-extract* tool allows the extraction of IR functions, globals, and also the deletion of globals from the IR module. For instance, extract the *sum* function from *sum.bc* with the following command:

```
$ llvm-extract -func=sum sum.bc -o sum-fn.bc
```

Nothing changes between *sum.bc* and *sum-fn.bc* in this particular example since *sum* is already the sole function in this module.

Introducing the LLVM IR language syntax

Observe the LLVM IR assembly file, *sum.ll*:

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:
16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:
128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.7.0"

define i32 @sum(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32* %a.addr, align 4
  %1 = load i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}

attributes #0 = { nounwind ssp uwtable ... }
```

The contents of an entire LLVM file, either assembly or bitcode, are said to define an LLVM module. The module is the LLVM IR top-level data structure. Each module contains a sequence of functions, which contains a sequence of basic blocks that contain a sequence of instructions. The module also contains peripheral entities to support this model, such as global variables, the target data layout, and external function prototypes as well as data structure declarations.

LLVM local values are the analogs of the registers in the assembly language and can have any name that starts with the *%* symbol. Thus, *%add = add nsw i32 %0, %1* will add the local value *%0* to *%1* and put the result in the new local value, *%add*. You are free to give any name to the values, but if you are short on creativity, you can just use numbers. In this short example, we can already see how LLVM expresses its fundamental properties:

- It uses the **Static Single Assignment** (**SSA**) form. Note that there is no value that is reassigned; each value has only a single assignment that defines it. Each use of a value can immediately be traced back to the sole instruction responsible for its definition. This has an immense value to simplify optimizations, owing to the trivial use-def chains that the SSA form creates, that is, the list of definitions that reaches a user. If LLVM had not used the SSA form, we would need to run a separate data flow analysis to compute the use-def chains, which are mandatory for classical optimizations such as constant propagation and common subexpression elimination.
- Code is organized as three-address instructions. Data processing instructions have two source operands and place the result in a distinct destination operand.

- It has an infinite number of registers. Note how LLVM local values can be any name that starts with the *%* symbol, including numbers that start at zero, such as *%0*, *%1*, and so on, that have no restriction on the maximum number of distinct values.

The *target datalayout* construct contains information about endianness and type sizes for *target triple* that is described in *target host*. Some optimizations depend on knowing the specific data layout of the target to transform the code correctly. Observe how the layout declaration is done:

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:
16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:
128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.7.0"
```

We can extract the following facts from this string:

- The target is an *x86_64* processor with *macOSX 10.7.0*. It is a little-endian target, which is denoted by the first letter in the layout (a lowercase *e*). Big-endian targets need to use an uppercase *E*.
- The information provided about types is in the format *type:<size>:<abi>:<preferred>*. In the preceding example, *p:64:64:64* represents a pointer that is 64 bits wide in *size*, with the *abi* and *preferred* alignments set to the 64-bit boundary. The ABI alignment specifies the minimum required alignment for a type, while the preferred alignment specifies a potentially larger value, if this will be beneficial. The 32-bit integer types *i32:32:32* are 32 bits wide in *size*, 32-bit *abi* and *preferred* alignment, and so on.

The function declaration closely follows the C syntax:

```
define i32 @sum(i32 %a, i32 %b) #0 {
```

This function returns a value of the type *i32* and has two *i32* arguments, *%a* and *%b*. Local identifiers always need the *%* prefix, whereas global identifiers use *@*. LLVM supports a wide range of types, but the most important ones are the following:

- Arbitrary-sized integers in the *iN* form; common examples are *i32*, *i64*, and *i128*.
- Floating-point types, such as the 32-bit single precision *float* and 64-bit double precision *double*.
- Vectors types in the format *<<# elements> x <elementtype>>*. A vector with four *i32* elements is written as *<4 x i32>*.

The *#0* tag in the function declaration maps to a set of function attributes, also very similar to the ones used in C/C++ functions and methods. The set of attributes is defined at the end of the file:

```
attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false"
"no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false"
"use-soft-float"="false" }
```

For instance, *nounwind* marks a function or method as not throwing exceptions, and *ssp* tells the code generator to use a *stack smash protector* in an attempt to increase the security of this code against attacks.

The function body is explicitly divided into **basic blocks** (**BBs**), and a label is used to start a new BB. A label relates to a basic block in the same way that a value identifier relates to an instruction. If a label declaration is omitted, the LLVM assembler automatically generates one using its own naming scheme. A basic block is a sequence of instructions with a single entry point at its first instruction, and a single exit point at its last instruction. In this way, when the code jumps to the label that corresponds to a basic block, we know that it will execute all of the instructions in this basic block until the last instruction, which will change the control flow by jumping to another basic block. Basic blocks and their associated labels need to adhere to the following conditions:

- Each BB needs to end with a terminator instruction, one that jumps to other BBs or returns from the function
- The first BB, called the entry BB, is special in an LLVM function and must not be the target of any

branch instructions

Our LLVM file, *sum.ll*, has only one BB because it has no jumps, loops, or calls. The function start is marked with the *entry* label, and it ends with the return instruction, *ret*:

```
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32* %a.addr, align 4
  %1 = load i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
```

The *alloca* instruction reserves space on the stack frame of the current function. The amount of space is determined by element type size, and it respects a specified alignment. The first instruction, *%a.addr = alloca i32, align 4*, allocates a 4-byte stack element, which respects a 4-byte alignment. A pointer to the stack element is stored in the local identifier, *%a.addr*. The *alloca* instruction is commonly used to represent local (automatic) variables.

The *%a* and *%b* arguments are stored in the stack locations *%a.addr* and *%b.addr* by means of *store* instructions. The values are loaded back from the same memory locations by *load* instructions, and they are used in the addition, *%add = add nsw i32 %0, %1*. Finally, the addition result, *%add*, is returned by the function. The *nsw* flag specifies that this add operation has “no signed wrap”, which indicates instructions that are known to have no overflow, allowing for some optimizations. If you are interested in the history behind the *nsw* flag, a worthwhile read is the LLVMdev post at http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html by Dan Gohman.

In fact, the *load* and *store* instructions are redundant, and the function arguments can be used directly in the *add* instruction. Clang uses *-O0* (no optimizations) by default, and the unnecessary loads and stores are not removed. If we compile with *-O1* instead, the outcome is a much simpler code, which is reproduced here:

```
define i32 @sum(i32 %a, i32 %b) ... {
entry:
  %add = add nsw i32 %b, %a
  ret i32 %add
}
...
```

Using the LLVM assembly directly is very handy when writing small examples to test target backends and as a means to learn basic LLVM concepts. However, a library is the recommended interface for frontend writers to build the LLVM IR, which is the subject of our next section. You can find a complete reference to the LLVM IR assembly syntax at http://llvm.org/docs/LangRef.html.

Introducing the LLVM IR in-memory model

The in-memory representation closely models the LLVM language syntax that we just presented. The header files for the C++ classes that represent the IR are located at *include/llvm/IR*. The following is a list of the most important classes:

- The *Module* class aggregates all of the data used in the entire translation unit, which is a synonym for "module" in LLVM terminology. It declares the *Module::iterator* typedef as an easy way to iterate across the functions inside this module. You can obtain these iterators via the *begin()* and *end()* methods. View its full interface at http://llvm.org/docs/doxygen/html/classllvm_1_1Module.html.
- The *Function* class contains all objects related to a function definition or declaration. In the case of a declaration (use the *isDeclaration()* method to check whether it is a declaration), it contains only the function prototype. In both cases, it contains a list of the function parameters accessible via the *getArgumentList()* method or the pair of *arg_begin()* and *arg_end()*. You can iterate through them using the

Function::arg_iterator typedef. If your *Function* object represents a function definition, and you iterate through its contents via the *for (Function::iterator i = function.begin(), e = function.end(); i != e; ++i)* idiom, you will iterate across its basic blocks. View its full interface at http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html.

- The *BasicBlock* class encapsulates a sequence of LLVM instructions, accessible via the *begin()/end()* idiom. You can directly access its last instruction using the *getTerminator()* method, and you also have a few helper methods to navigate the CFG, such as accessing predecessor basic blocks via *getSinglePredecessor()*, when the basic block has a single predecessor. However, if it does not have a single predecessor, you need to work out the list of predecessors yourself, which is also not difficult if you iterate through basic blocks and check the target of their terminator instructions. View its full interface at http://llvm.org/docs/doxygen/html/classllvm_1_1BasicBlock.html.
- The *Instruction* class represents an atom of computation in the LLVM IR, a single instruction. It has some methods to access high-level predicates, such as *isAssociative()*, *isCommutative()*, *isIdempotent()*, or *isTerminator()*, but its exact functionality can be retrieved with *getOpcode()*, which returns a member of the *llvm::Instruction* enumeration, which represents the LLVM IR opcodes. You can access its operands via the *op_begin()* and *op_end()* pair of methods, which are inherited from the *User* superclass that we will present shortly. View its full interface at http://llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html.

We have still not presented the most powerful aspect of the LLVM IR (enabled by the SSA form): the *Value* and *User* interfaces; these allow you to easily navigate the use-def and def-use chains. In the LLVM in-memory IR, a class that inherits from *Value* means that it defines a result that can be used by others, whereas a subclass of *User* means that this entity uses one or more *Value* interfaces. *Function* and *Instruction* are subclasses of both *Value* and *User*, while *BasicBlock* is a subclass of just *Value*. To understand this, let's analyze these two classes in depth:

- The *Value* class defines the *use_begin()* and *use_end()* methods to allow you to iterate through *User*s, offering an easy way to access its def-use chain. For every *Value* class, you can also access its name through the *getName()* method. This models the fact that any LLVM value can have a distinct identifier associated with it. For example, *%add1* can identify the result of an add instruction, *BB1* can identify a basic block, and *myfunc* can identify a function. *Value* also has a powerful method called *replaceAllUsesWith(Value *)*, which navigates through all of the users of this value and replaces it with some other value. This is a good example of how the SSA form allows you to easily substitute instructions and write fast optimizations. You can view the full interface at http://llvm.org/docs/doxygen/html/classllvm_1_1Value.html.
- The *User* class has the *op_begin()* and *op_end()* methods that allows you to quickly access all of the *Value* interfaces that it uses. Note that this represents the use-def chain. You can also use a helper method called *replaceUsesOfWith(Value *From, Value *To)* to replace any of its used values. You can view the full interface at http://llvm.org/docs/doxygen/html/classllvm_1_1User.html.

Summary

In this article, we acquainted ourselves with the concepts and components related to the LLVM intermediate representation.