The LLVM compiler framework

Introduction

Daniele Cattaneo

Politecnico di Milano

2021-03-09

These slides are based on material originally written by Michele Scandale, Ettore Speziale and Stefano Cherubin for the “Code Transformation and Optimization” course.

Contents

Compilers and compilers

You might have already have had experience working in a **toy compiler**...

Toy Compiler	**Production-Quality Compiler**
► small codebase	► huge codebase
► easy to modify	► hard to modify
► limited capabilities	► produces high-quality code

Initially, working with a production-quality compiler might seem **hard**...

...however it provides a huge set of tools that toy compilers **miss**!

Why LLVM

Why is this course focused on LLVM?

- Key technology in the **industry**
  - AMD, Apple, Google, Intel, NVIDIA...
- Biggest platform for **research** about compilers
- **Modular** and **hackable**

Initially started as a small research project at Urbana-Champaign.

Now it has grown to a huge size...

GCC vs LLVM

LLVM [1] is Open Source

If you are familiar with Linux you might have used **GCC** [2]...

GCC is older than LLVM

$\Rightarrow$ GCC produces better code
$\Rightarrow$ LLVM is generally faster
$\Rightarrow$ LLVM is more modular and *clean*

Contents

Compiler pipeline

Typically a compiler is a **pipeline**.

Advantages of the pipeline model:

- **simplicity** – read something, produce something
- **locality** – no superfluous state

Complexity lies on **chaining** together stages.

Frontends and Drivers

The external interface to LLVM is provided by the **compiler driver**.

The **compiler driver** is the program that:

- Provides the interface to the user
- Performs setup of the frontend and LLVM itself.

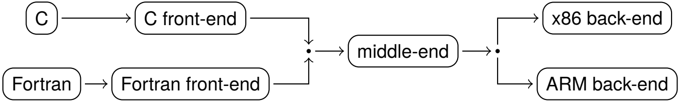
The driver invokes the **first stage of the pipeline**, the **frontend**.

Example

Clang[3] is the frontend for the C language family.
The driver of *Clang* is the `clang` executable (compatible with GCC).

Compiler pipeline

High-level pipeline structure of a compiler:



There are three main components:

Front-end **translates** a source file into the intermediate representation

Middle-end **analyzes** the intermediate representation, **optimizes** it

Back-end **generates** target machine assembly from the intermediate representation

The LLVM compiler pipeline

We will focus on the *middle-end*.
Same concepts are valid also for {front,back}-end.

- The *front-end* is completely language-specific
  - There are no special facilities in LLVM for parsing and AST generation
  - This is slowly changing: **MLIR**
- The *back-end* uses its own machine-specific IR
  - As opposed to the frontend, in LLVM there are generalized facilities for implementing a backend
  - Complex topic, not enough time...
  - Most optimizations happen in the middle-end anyway

The LLVM compiler pipeline

The lingua franca of LLVM is its **Intermediate Representation** called

LLVM-IR

The LLVM-IR is:

- **Produced** by the **front-end**
- **Transformed and optimized** by the **middle-end**
- **Consumed** by the **back-end**

Understanding LLVM-IR is the key to hacking within LLVM.

Remember...

LLVM is a **compiler construction framework**
It operates on the **LLVM-IR** language.

$\Downarrow$

Using LLVM *by itself* does not make much sense!
Writing LLVM-IR by hand is unfeasible.

The Middle End

The middle-end is not a monolithic block,
but it is a pipeline in and of itself.

LLVM-IR	the **language** used in the middle-end
Pass	a **pipeline stage** a Pass may have **dependencies** on other Passes.
Pass Manager	component that **schedules** passes according to their **dependencies** and **executes** them *(builds the pipeline)*

Our focus: **writing a pass**

First insights

A compiler is **complex**:

- passes are the **elementary unit of work**
- Pass Manager must be **advised** about pass chaining
- pipeline shapes are **not fixed** – they can change from one compiler execution to another

  e.g. optimized/not optimized builds, compiler options, ...

A word of warning!

Compilers must be **conservative**:

$\Downarrow$

All passes **must preserve the program semantics**

$\Downarrow$

Compiler passes must be designed **very carefully**!

Contents

Classical Algorithm Design

In algorithm design, a good approach is the following:

1. study the problem
2. make some example
3. identify the **common case**
4. derive the algorithm for the common case
5. add handling for **corner cases**
6. improve performance by **optimizing the common case**

Weakness of the approach:

- **corner cases**:
  a *correct* algorithm **must** consider *all the corner cases*!

Compiler Algorithm Design

Corner cases are difficult to handle, but they cannot be ignored

Compiler algorithms must be **proven** to preserve
program semantic **at all times**

As an aid, a *standard methodology* is employed.

Compiler algorithms are built combining **three** kinds of passes:

Analysis, Optimization, Normalization

Compiler Algorithm Design

Corner cases are difficult to handle, but they cannot be ignored

Compiler algorithms must be **proven** to preserve
program semantic **at all times**

As an aid, a *standard methodology* is employed.

Compiler algorithms are built combining **three** kinds of passes:

Analysis, Optimization, Normalization

We now consider a simple example: *loop hoisting*.

Loop Hoisting

It is a transformation that:

- looks for statements (inside a loop) not depending on the loop state
- move them outside the loop body

Loop Hoisting – Before

```
do {
  a += i;
  b = c;
  i++;
} while (i < k);
```

Loop Hoisting – After

```
b = c;
do {
  a += i;
  i++;
} while (i < k);
```

Loop Hoisting

The general idea:

- move “good” statement outside of the loop

This **pass** modifies the code, thus it is an **optimization pass**.
It needs to know:

- which pieces of code are loops
- which statements are “good” statements

This information is computed by the the **analysis** passes:

- detecting loops in the program
- detecting loop-independent statements

The loop hoisting pass declares which analyses it needs:

- pipeline automatically built: **analysis** $\rightarrow$ **optimization**

Loop Hoisting

The **proof** is trivial:

- the transformation shall preserve program semantics
- the analyses shall be correct

Analysis passes are usually built starting from other analyses already implemented inside the compiler, or are already present in LLVM

- often, no proof is necessary for the analyses

However...

You also have to prove that the combination of analysis + transformation is correct!

”Beware of bugs in the above code;
I have only proved it correct, not tried it.”
— Knuth

The importance of normalization

We have spoken about loops, but which kind of loop?

`do-while`? **`while`**? **`for`**?

Almost all loops are different forms of the **same exact thing**

$\Downarrow$

We can convert a lot of loops to a loop of another kind!

To account for the various kinds of loops, we choose a **normal** kind of loop, and then we write a **normalization** pass.

Usually, **`do-while`** loops are chosen to be the *normal* loops.
Sometimes, normalization is also called **canonicalization**.

The more loops we recognize, the higher the potential **optimization impact**!

A Methodology

You have to:

1. analyze the problem
2. make some examples
3. detect the common case
4. determine the **input conditions**
5. determine which **analyses** you need
6. design the **optimization** pass
7. proof its **correctness**
8. improve algorithm perfomance on the common case
9. improve the effectiveness of the algorithm by adding **normalization passes**

Ignore corner cases!

Something is missing...

Corner Cases!

Why?

1. It makes no sense to optimize code that is seldom executed
2. Your optimization will be based on **properties of the code that are true only in the common case you are considering**
   - If the code does not fit the common case, it shall stay as-is
   - Otherwise you **risk breaking program semantics**!

Contents

LLVM-IR is like a box of chocolates

LLVM-IR comes in 3 different flavours:

assembly	on-disk human-readable format (file extension: `.ll`)
bitcode	on-disk machine-oriented binary format (file extension: `.bc`)
in-memory	in-memory binary format (used during compilation process)

All formats have the same expressiveness!

Using the driver to produce LLVM-IR

We can generate LLVM-IR assembly using the `clang` driver:

```
clang -emit-llvm -S -o out.ll in.c
```

If you want to generate bitcode instead:

```
clang -emit-llvm -o out.bc in.c
```

The compiler driver can also generate native code starting from LLVM-IR assembly

(Like compiling an assembly file with GCC)

Playing with LLVM Passes

Run one or more passes on the LLVM-IR on-demand by using `opt`:

- Syntax is like `clang` (supports even `-O1`, `-O2`...)
- One command line argument per pass to run
- Order of execution is the same as the argument order
  - Different order, different results! (**phase/stage ordering**)

Some useful passes for debugging (they do not transform anything):

print CFG	`opt -view-cfg input.ll`
print dominator tree	`opt -view-dom input.ll`
print current IR	`opt -print-module input.ll`

Example

- Run *mem2reg*, then view the CFG:
  - `opt -mem2reg -view-cfg input.ll`

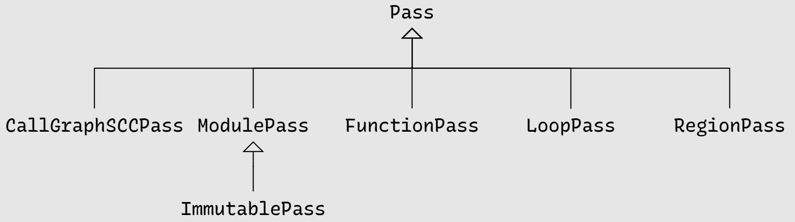
Pass Hierarchy

LLVM provides a lot of passes...

- Try `opt -help`!

For performance reasons there are different kind of passes:

LLVM Passes



LLVM Passes

Each kind of pass visits particular elements of a module:

ImmutablePass	compiler configuration – never run
CallGraphSCCPass	post-order visit of CallGraph SCCs
ModulePass	visit the whole module
FunctionPass	visit functions
LoopPass	post-order visit of loop nests
RegionPass	visit a custom-defined region of code

Specializations come with restrictions:

- e.g. a **FunctionPass** cannot add or delete functions
- refer to “Writing a LLVM Pass” [4] for documentation on features and limitations of each kind of pass

Recap

- The **user** invokes the **compiler** via the **driver**
- The **compiler** is made of three **stages** (front-end, middle-end, back-end)
- The **middle-end** is made of **passes**

If you want things done, you want to work on a **pass**.

- Edit an existing pass
- Create a new pass

To design a pass, you must follow the principle of **conservativeness**.

Next step: how to actually code a pass!

Contents

How passes work

A **pass** is a **subroutine** that programmatically transforms a piece of code.

The code of the pass operates on the LLVM-IR using a set of **object-oriented APIs**.

Let's examine the LLVM-IR [5] more closely, first by looking at its **human-readable** form.

LLVM-IR

```
define i32 @fact(i32 %n) {
entry:
  %retval = alloca i32, align 4
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32, i32* %n.addr, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 1, i32* %retval, align 4
  br label %return

if.end:
  %1 = load i32, i32* %n.addr, align 4
  %2 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call i32 @fact(i32 %sub)
  %mul = mul nsw i32 %1, %call
  store i32 %mul, i32* %retval, align 4
  br label %return

return:
  %3 = load i32, i32* %retval, align 4
  ret i32 %3
}
```

LLVM-IR

LLVM-IR looks a lot like a RISC assembly language:

- Few instructions, all perfectly orthogonal
  - There are infinite registers
  - There are no special-purpose registers
  - No implicit flags register
- Basic block boundaries are denoted by **labels**
- Only **`load`** and **`store`** access memory

There are also a few CISC-like **high level instructions**:

- Reserve memory on the stack – **`alloca`**
- Function call – **`call`**
  - The calling convention is abstracted away
  - There is an implicit call stack
- Pointer arithmetics – **`getelementptr`**
- …

LLVM-IR

In reality LLVM-IR is much more high-level than assembly.

- The topmost object of a LLVM-IR program is the **Module**.
- **Modules** contain a list of **Globals**.
  - Globals can be either **Functions** or **Global Variables**.
  - A global can be a **Forward declaration**.
- **Functions** contain a list of **Basic Blocks** + **Arguments**.
- **Basic Blocks** are a list of **Instructions**.

The in-memory representation

All these parts will correspond directly to **C++ objects**.
The abundance of lists guarantees low overhead and scalability to very large programs.

LLVM-IR

LLVM-IR is **strongly typed**:

- e.g. you cannot assign a floating point value to an integer register without an explicit cast

Almost everything is **typed**:

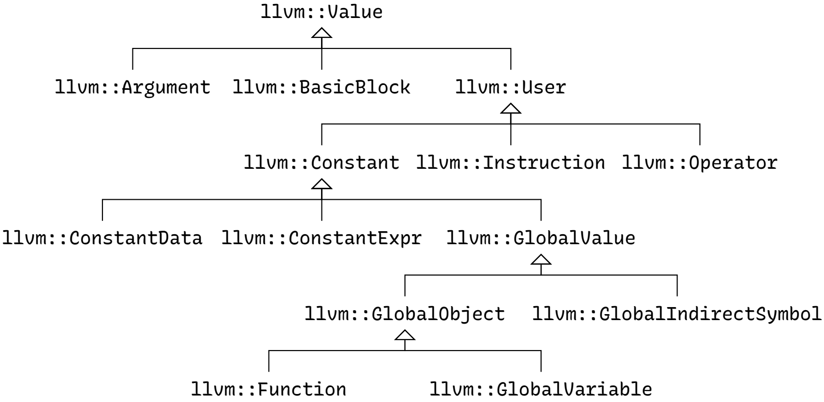
functions	`@fact`	$\longrightarrow$	`i32 (i32)`
registers	`%3 = icmp eq i32 %2, 0`	$\longrightarrow$	`i1`
globals var.	`@var = common global i32 0`	$\longrightarrow$	`i32`

These objects that have a type are called (somewhat confusingly) **LLVM Values**.

The in-memory representation

`llvm::Value` is the **base class** of almost all interesting LLVM-IR objects!

LLVM-IR


llvm::Value
llvm::Argument
llvm::BasicBlock
llvm::User
llvm::Constant
llvm::Instruction
llvm::Operator
llvm::ConstantData
llvm::ConstantExpr
llvm::GlobalValue
llvm::GlobalObject
llvm::GlobalIndirectSymbol
llvm::Function
llvm::GlobalVariable

LLVM-IR

LLVM-IR is SSA-based:

- every register is **statically assigned** exactly **once**

Statically means that:

- inside each function...
- ...for each variable `%foo`...
- ...there is **only one** statement in the form `%foo = ...`

Static (compile time) $\neq$ **dynamic** (runtime)

- Single *Dynamic* Assignment:
  in the execution trace there is only one assignment to a variable `x`
- Single *Static* Assignment:
  in the code listing there is only one assignment to a variable `x`
  - Assignments **can** be performed multiple times (in a loop for example)

Static Single Assignment

Scalar SAXPY

```
float saxpy(float a, float x, float y) {
    return a * x + y;
}
```

Scalar LLVM SAXPY

```
define float @saxpy(float %a, float %x, float %y) {
  %1 = fmul float %a, %x
  %2 = fadd float %1, %y
  ret float %2
}
```

Temporary %1 not reused! %2 is used for the second assignment!

Static Single Assignment

Array SAXPY

```
void saxpy(float a, float x[4], float y[4], float z[4]) {
  for(unsigned i = 0; i < 4; ++i)
    z[i] = a * x[i] + y[i];
}
```

Array LLVM SAXPY

```
for.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i32 %i.0, 4
  br i1 %cmp, label %for.body, label %for.end
  [...]
for.inc:
  %inc = add i32 %i.0, 1
  br label %for.cond
```

One assignment for loop counter `%i.0`

Static Single Assignment

Max

```
float max(float a, float b) {
  return a > b ? a : b;
}
```

LLVM Max – WRONG

```
  %1 = fcmp ogt float %a, %b
  br i1 %1, label %if.then, label %if.else
if.then:
  %2 = %a
  br label %if.end
if.else:
  %2 = %b
  br label %if.end
if.end:
  ret float %2
```

Why is it **wrong**?

Static Single Assignment

The `%2` variable must be statically assigned once!
How do we handle conditional assignments then?

LLVM Max

```
  %1 = fcmp ogt float %a, %b
  br i1 %1, label %if.then, label %if.end
if.then:
  br label %if.end
if.else:
  br label %if.end
if.end:
  %2 = phi float [ %a, %if.then ], [ %b, %if.else ]
  ret float %2
```

The **`phi`** instruction is a *conditional move*:

- it takes $(\text{variable}_i, \text{label}_i)$ pairs
- if coming from predecessor identified by label_i, its value is variable_i

Static Single Assignment

Each SSA variable is assigned only once:

- variable **definition**

Each SSA variable can be referenced by multiple instructions:

- variable **uses**

Algorithms and technical language abuse of these terms!

Let %foo be a variable. If the definition of %foo does not have side-effects nor uses, the aforementioned %foo variable can be erased from the CFG without altering program semantics.

SSA & LLVM-IR

Important observation

SSA means that
there is always a 1:1 correspondence
between a register and the instruction that assigns it.

Consequence

As a result, in LLVM-IR
registers are not separate objects
but **every LLVM Instruction**
is the output “register” of itself.

Static Single Assignment

Old compilers are not SSA-based:

- converting non-SSA input into SSA form is expensive
- cost must be amortized

New compilers are SSA-based:

- SSA easier to work with
- SSA-based analysis/optimizations are faster

Contents

A step back...

Remember how we described the internal structure of an LLVM-IR module:

- `llvm::Module` is a list of `llvm::GlobalValues`.
  - `llvm::Function` is a kind of `llvm::GlobalValue`.
- `llvm::Function` is a list of `llvm::BasicBlocks`.
- `llvm::BasicBlock` is a list of `llvm::Instruction`.

Functions and basic blocks act like containers:

- STL-like accessors: `front()`, `back()`, `size()`, …
- STL-like iterators: `begin()`, `end()`

Each contained element is aware of its container:

- `getParent()`

Warning for BBs: order of iteration $\neq$ order of execution!

A step back...

In a `llvm::BasicBlock`, the `llvm::Instructions` execute in the order specified by the list.

- In which order do the `llvm::BasicBlocks` execute?

The way the basic blocks are executed is implcitly described by the **branches** in each block.

- These branches describe the **Control Flow Graph** of the function.

Control Flow Graph

LLVM automatically maintains a simple API for operating on the CFG:

- no need to run passes
- no need to search the branch instructions in each basic block

Every CFG has an **entry** basic block:

- the **first** executed basic block
- it is the **root/source** of the graph
- get it with `llvm::Function::getEntryBlock()`

Control Flow Graph

At the end of a basic blocks there's always a **terminator** instruction:

- **`ret`**, **`br`**, **`switch`**, **`unreachable`**, …

More than one **exit** block can be present in a function:

- they are the **leaves/sinks** of the graph
- their terminator instructions are always **`ret`**s
    1. `llvm::BasicBlock::getTerminator()`
    2. check the opcode of the terminator

Side Note

For performance reasons, a custom casting framework is used:

- you cannot use **`static_cast`** and **`dynamic_cast`** with types/classes provided by LLVM

LLVM Casting Functions

Static cast of `Y*` to `X`	`X *llvm::cast<X>(Y *)`
Dynamic cast of `Y*` to `X`	`X *llvm::dyn_cast<X>(Y *)`
Is `Y*` an instance of `X`?	`bool llvm::isa<X>(Y *)`

Example:

- is `BB` a sink?
  `llvm::isa<llvm::ReturnInst>(BB.getTerminator())`

Control Flow Graph

Every basic block `BB` has one or more*:

predecessors from `pred_begin(BB)` to `pred_end(BB)`

successors from `succ_begin(BB)` to `succ_end(BB)`

Other convenience methods available in `llvm::BasicBlock`:

- useful getters
  - `BasicBlock *getUniquePredecessor()`
  - …
- moving a basic block
  - `moveBefore(llvm::BasicBlock *)`
  - `moveAfter(llvm::BasicBlock *)`
- split a basic block:
  - `splitBasicBlock(llvm::BasicBlock::iterator)`
- …

---

*see include/llvm/IR/CFG.h

Control Flow Graph

The `llvm::Instruction` class defines common operations:

- getting an operand
  - `getOperand(unsigned)`

Subclasses provide specialized accessors:

- the **`load`** instruction takes as operand the pointer to the memory to be loaded:
  - `llvm::LoadInst::getPointerOperand()`

Instructions

Instructions are created using:

- constructors
  - `llvm::LoadInst::LoadInst(...)`
- factory methods
  - `llvm::GetElementPtrInst::Create(...)`
- the helper class `llvm::IRBuilder`
  - ```
    llvm::IRBuilder<> builder(insPoint);
    builder.CreateAdd(...);
    ```

Interface is not homogeneous!
Some instructions support all methods, others support only one.
    ```

Instructions

Instructions can be inserted:

- automatically by `IRBuilder`
  - insertion point is given at `IRBuilder` instantiation
- manually by appending to a basic block
- manually by inserting after/before another instruction

From Control Flow to Data Flow

In LLVM, the data flow generated by the various instructions is represented by a simple hierarchy:

value	something that can be used: `llvm::Value`
user	something that can use: `llvm::User`
use	the link between the **value** and the **user**: `llvm::Use`

A value is a **definition**:

- Visiting where a definition is used:
  - `llvm::Value::use_begin()`, `llvm::Value::use_end()`

An user accesses **definitions**:

- Visiting the definitions that are used:
  - `llvm::User::op_begin()`, `llvm::User::op_end()`

From Control Flow to Data Flow

- `llvm::Value` inherits from `llvm::User`
- `llvm::Instruction` inherits from `llvm::Value`
  - $\Rightarrow$ The value produced by the instruction is the **instruction itself**!

Example

```
%6 = load i32, i32* %1, align 4
```

The **`load`** is described by an instance of `llvm::Instruction`.
That instance also represents the `%6` variable.

Not all instances of `llvm::Value` are also `llvm::Instructions`!

i.e. function arguments

From Control Flow to Data Flow

Every `llvm::Value` is typed:

- use `llvm::Value::getType()` to get the type

Since every instruction is a value:

- instructions are typed

Example

```
%6 = load i32, i32* %1, align 4
```

The type of the `%6` variable is the type of the return value of the **`load`** instruction, `i32`

Contents

Conclusions

LLVM is a **production-quality** compiler framework:

$\Rightarrow$ impossible knowing all details

But:

- it is well organized
- if you know compiler theory, it is relatively easy to find what you need inside the source code

Remember it's written in C++!

- To hack around LLVM you need at least basic C++ skills
- C++ $\neq$ C

Contents

Bibliography I

- University of Illinois at Urbana-Champaign.
  Low Level Virtual Machine.
  http://www.llvm.org.

- GNU.
  GNU Compiler Collection.
  http://gcc.gnu.org.

- University of Illinois at Urbana-Champaign.
  Clang: a C language family frontend for LLVM.
  http://clang.llvm.org.

- Chris Lattner and Jim Laskey.
  Writing an LLVM Pass.
  http://llvm.org/docs/WritingAnLLVMPass.html.

Bibliography II

- Chris Lattner and Vikram Adve.
  LLVM Language Reference Manual.
  http://llvm.org/docs/LangRef.html.
- Linus Torvalds.
  Re: SCO: "thread creation is about a thousand times faster than onnative.
  https://lkml.org/lkml/2000/8/25/132.
- Bruce Eckel.
  Thinking in C++ – Volume One: Introduction to Standard C++.
  http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.
- Bruce Eckel and Chuck Allison.
  Thinking in C++ – Volume Two: Practical Programming.
  http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Bibliography III

- AMD.
  Open64.
  http://developer.amd.com/tools-and-sdks/cpu-development/x86-open64-compiler-suite.

- John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner.
  LLVM Testing Infrastructure Guide.
  http://llvm.org/docs/TestingGuide.html.

- LLVM Community.
  LLVM Coding Standards.
  http://llvm.org/docs/CodingStandards.html.

- LLVM Community.
  LLVM Passes.
  http://llvm.org/docs/Passes.html.

Bibliography IV

- LLVM Community.
  Autovectorization in LLVM.
  http://llvm.org/docs/Vectorizers.html.

- LLVM Community.
  LLVM Programmer's Manual.
  http://llvm.org/docs/ProgrammersManual.html.

- Ettore Speziale.
  Compiler Optimization and Transformation Passes.
  https://github.com/speziale-ettore/COTPasses.

- Scott Chacon.
  Pro Git.
  http://git-scm.com/book.