Introduction to LLVM

Zhaomo Yang
January 11, 2018
Architecture of LLVM
Architecture of LLVM

Front-end

Optimizer

Back-end
LLVM Optimizer

The optimizer analyzes, optimizes and secures programs.

The optimizer operates on LLVM Intermediate Representation (IR) code, which makes it source- and target-independent.

Functionalities are implemented as passes.
Optimizer Passes

A pass is an operation on a unit of LLVM Intermediate Representation (IR) code.

There are multiple types of passes:

- ModulePass, CallGraphSCCPass, FunctionPass, LoopPass, RegionPass, BasicBlockPass
Optimizer Passes

A pass is an operation on a unit of LLVM Intermediate Representation (IR) code.

There are multiple types of passes:

- ModulePass, CallGraphSCCPass, **FunctionPass**, LoopPass, RegionPass, BasicBlockPass

How to write a function pass:
http://releases.llvm.org/5.0.1/docs/WritingAnLLVMPass.html#writing-an-llvm-pass-basiccode
LLVM IR

- A low-level **strongly-typed** language-independent, SSA-based representation.
- Tailored for static analyses and optimization purposes.
- LLVM IR language reference: [http://releases.llvm.org/5.0.1/docs/LangRef.html](http://releases.llvm.org/5.0.1/docs/LangRef.html)
int foo (int x) {
    int i = 0;
    volatile int count = 0;
    for (; i<x; i++) {
        count ++;
    }
    return count;
}

define i32 @foo(i32) #0 {
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    store i32 0, i32* %3, align 4
    store volatile i32 0, i32* %4, align 4
    br label %5

    ; <label>:5:
    %6 = load i32, i32* %3, align 4
    %7 = load i32, i32* %2, align 4
    %8 = icmp slt i32 %6, %7
    br i1 %8, label %9, label %15

    ; <label>:9:
    %10 = load volatile i32, i32* %4, align 4
    %11 = add nsw i32 %10, 1
    store volatile i32 %11, i32* %4, align 4
    br label %12

    ; <label>:12:
    %13 = load i32, i32* %3, align 4
    %14 = add nsw i32 %13, 1
    store i32 %14, i32* %3, align 4
    br label %5

    ; <label>:15:
    %16 = load volatile i32, i32* %4, align 4
    ret i32 %16
}
Hierarchy of structures of IR programs

Module
Function
Basic Block
Instruction
A module == A compilation unit

gcc -c mytest.c -o mytest.o

The compilation unit consists of

- Code in mytest.c
- Code that is included in mytest.c (#include ...)

Project Part 1 overview

There are three sections:

- Count Static Instructions
- Count Dynamic Instructions
- Branch Profiling
Project Part 1 overview

There are three sections:

- Count Static Instructions: analysis pass
- Count Dynamic Instructions: transformation pass
- Branch Profiling: transformation pass
Project Part 1 overview

There are three sections:

- **Count Static Instructions**: analysis pass
- Count Dynamic Instructions: transformation pass
- Branch Profiling: transformation pass
How an analysis pass works

```c
void foo (uint32_t int,
         uint32_t * p) {
    ...
}
```

- `test.c`
- `Clang`
- `LLVM IR`
- `opt`
- `mypass.so`
- `stderr`
LLVM IR

```c
int foo (int x) {
    int i = 0;
    volatile int count = 0;
    for (; i<x; i++) {
        count ++;
    }
    return count;
}
```

```c
define i32 @foo(i32) #0 {
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %2, align 4
    store i32 0, i32* %3, align 4
    store volatile i32 0, i32* %4, align 4
    br label %5

    ; <label>:5:
    %6 = load i32, i32* %3, align 4
    %7 = load i32, i32* %2, align 4
    %8 = icmp slt i32 %6, %7
    br i1 %8, label %9, label %15

    ; <label>:9:
    %10 = load volatile i32, i32* %4, align 4
    %11 = add nsw i32 %10, 1
    store volatile i32 %11, i32* %4, align 4
    br label %12

    ; <label>:12:
    %13 = load i32, i32* %3, align 4
    %14 = add nsw i32 %13, 1
    store i32 %14, i32* %3, align 4
    br label %5

    ; <label>:15:
    %16 = load volatile i32, i32* %4, align 4
    ret i32 %16
}
```
Count Static Instructions

How to traverse a function (and how to write to stderr)

http://releases.llvm.org/5.0.1/docs/ProgrammersManual.html#basic-inspection-and-traversal-routines
Project Part 1 overview

There are three sections:

- Count Static Instructions: analysis pass
- **Count Dynamic Instructions: transformation pass**
- Branch Profiling: transformation pass
```llvm
define i32 @foo(i32) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  store i32 0, i32* %3, align 4
  store volatile i32 0, i32* %4, align 4
  br label %5

; <label>:5:
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %2, align 4
  %8 = icmp slt i32 %6, %7
  br i1 %8, label %9, label %15

; <label>:9:
  %10 = load volatile i32, i32* %4, align 4
  %11 = add nsw i32 %10, 1
  store volatile i32 %11, i32* %4, align 4
  br label %12

; <label>:12:
  %13 = load i32, i32* %3, align 4
  %14 = add nsw i32 %13, 1
  store i32 %14, i32* %3, align 4
  br label %5

; <label>:15:
  %16 = load volatile i32, i32* %4, align 4
  ret i32 %16
}
```
Clang++

Void foo (uint32_t int, uint32_t * p) {
    ...
}

test.cpp

Int main (){
    ...
    foo ()
    ...
}

main.cpp

Int main (){
    ...
    foo ()
    ...
}

lib.cpp

How a transformation pass works

Clang++

LLVM IR

test.bc

opt

LLVM IR

main.bc

LLVM IR

lib.bc

mypass.so

LLVM IR

test-ins.bc

Clang++

Executable
How to insert a function call to IR code

First of all, we need to find the function we want to call.

- **Class Function** represents functions in IR programs
- How can we get a handle of the function?
How to insert a function call to IR code

First of all, we need to find the function we want to call.

- **Class Function** represents functions in IR programs
- How can we get a handle of the function?

```cpp
// getOrInsertFunction - Look up the specified function in the module symbol
// table. If it does not exist, add a prototype for the function and return it.
// This version of the method takes a null terminated list of function
// arguments, which makes it easier for clients to use.

Module::getOrInsertFunction
```
How to use getOrInsertFunction

```c
void helper (uint32_t x, uint32_t * p);
```

```c
Function * myHelper =
    cast<Function>(Mod.getOrInsertFunction("helper",
        Type::getVoidTy(context),
        Type::getInt32Ty(context),
        Type::getInt32PtrTy(context)));
```
How to prepare arguments for a function call

If you need a constant integer

- **ConstantInt** represents boolean and integer constants
- From **Class ConstantInt**

```c
static ConstantInt *get(IntegerType *Ty, uint64_t V,
    bool isSigned = false);
```
How to prepare arguments for a function call

If you need a constant integer

```cpp
static ConstantInt *get(IntegerType *Ty, uint64_t V, bool isSigned = false);
```

- How to get the type of the constant integer?

```cpp
class Type

IntegerType *Type::getInt1Ty(LLVMContext &C) { return &C.pImpl->Int1Ty; }
IntegerType *Type::getInt8Ty(LLVMContext &C) { return &C.pImpl->Int8Ty; }
IntegerType *Type::getInt16Ty(LLVMContext &C) { return &C.pImpl->Int16Ty; }
IntegerType *Type::getInt32Ty(LLVMContext &C) { return &C.pImpl->Int32Ty; }
IntegerType *Type::getInt64Ty(LLVMContext &C) { return &C.pImpl->Int64Ty; }
IntegerType *Type::getInt128Ty(LLVMContext &C) { return &C.pImpl->Int128Ty; }
```
How to prepare arguments for a function call

If you need a pointer to a constant array

- Allocate the array somewhere in the address space

The easiest way to do it is to put the array in the static region.

Class GlobalVariable represents static and global variables of a program.
How to prepare arguments for a function call

Class **GlobalVariable** represents static and global variables of a program.

```cpp
GlobalVariable(Module &M, Type *Ty, bool isConstant,
LinkageTypes Linkage, Constant *Initializer,
const Twine &Name = "", GlobalVariable *InsertBefore = nullptr,
ThreadLocalMode = NotThreadLocal, unsigned AddressSpace = 0,
bool isExternallyInitialized = false);
```

This constructor has quite a few parameters but luckily many of them have a default value that we don’t need to change.
How to prepare arguments for a function call

Class `GlobalVariable` represents static and global variables of a program.

```cpp
GlobalVariable(Module &M, Type *Ty, bool isConstant,
LinkageTypes Linkage, Constant *Initializer,
const Twine &Name = "", GlobalVariable *InsertBefore = nullptr,
ThreadLocalMode = NotThreadLocal, unsigned AddressSpace = 0,
bool isExternallyInitialized = false);
```

```cpp
GlobalVariable* VG = new GlobalVariable(*(F.getParent()),
ArrayTy,
true,
GlobalValue::InternalLinkage,
ConstantDataArray::get(context, values),
"value global");
```
How to prepare arguments for a function call

Class **GlobalVariable** represents static and global variables of a program.

“Because GlobalValues are memory objects, they are always referred to by their address. As such, the Type of a global is always a pointer to its contents.”

[http://releases.llvm.org/5.0.1/docs/ProgrammersManual.html#the-globalvariable-class](http://releases.llvm.org/5.0.1/docs/ProgrammersManual.html#the-globalvariable-class)
How to insert a function call

Now that we have the function handle and the arguments, we can finally insert a function call.

**Class IRBuilder** can be used for insert instructions into a basic block.

- First, we need to specify where we want to insert the instruction

Either use function `SetInsertPoint` or specify the insert point in the constructor of `IRBuilder` (which will call `SetInsertPoint`).
How to insert a function call

Now that we have the function handle and the arguments, we can finally insert a function call.

**Class IRBuilder** can be used for insert instructions into a basic block.

- First, we need to specify where we want to insert the instruction
- Second, we need to create the IR call instruction

Use **IRBuilder::CreateCall**

```
Builder.CreateCall(processBBFunction, args);
```
Tips

- Learn from other use cases of the API in the code base
- Read the comments above the definition/declaration of the function you want to use
- Use an IDE (“Open Declaration” and “Open Call Hierarchy”)
- Read the code of the function you want to use
Links

- How to write a basic function pass
  http://releases.llvm.org/5.0.1/docs/WritingAnLLVMPass.html

- Developer Tutorial: covering many common operations
  http://releases.llvm.org/5.0.1/docs/ProgrammersManual.html

- Our tutorial
  https://ucsd-pl.github.io/cse231/wi18/tutorials/part1.html