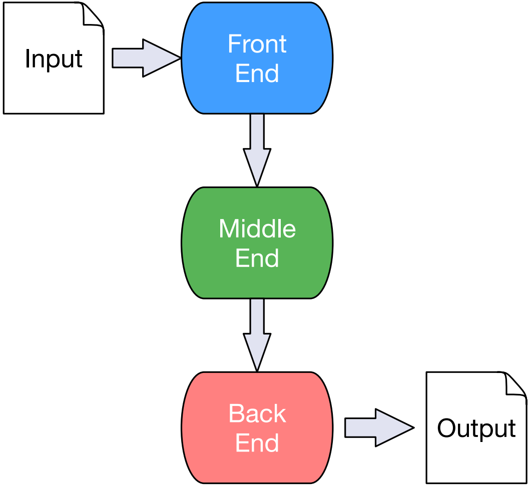
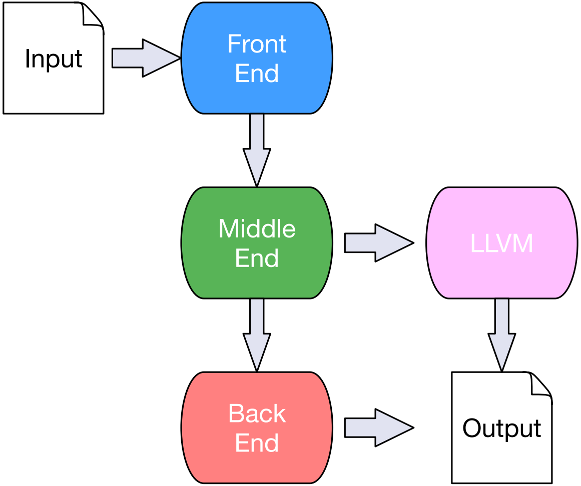
Compiler Construction

LLVM

Back to compiler architecture



LLVM or how to avoid writing a Backend



LLVM origins

LLVM project

- started in 2000 at the University of Illinois at Urbana–Champaign
- supervised by Vikram Adve and Chris Lattner.
- 2005: Apple hired Chris Lattner

Original goal

Investigate dynamic compilation techniques for static and dynamic programming languages.

LLVM nowadays

LLVM

Collection of modular and reusable compiler and toolchain technologies

- Takes IR code and emits an optimized IR
- Converted IR to ASM
- Allows static compilation or JIT
- Collection of libraries

Some tools using LLVM

- Clang
- LLDB
- AddressSanitizer, ThreadSanitizer, MemorySanitizer, and DataFlowSanitizer.
- Polly
- NVVM CUDA Compiler
- Xcode
- Emscriptem
- Rust, Kotlin, swift
- Cling
- ...

Main idea

Plug LLVM to our compiler

Translate our LIR into LLVM-IR.
Use LLVM to generate ASM.

High Level Example in Tiger

Consider this tiger example:

```
let
  function sum(a: int ,
               b: int) : int =
   a + b + 42
in
   sum(5,7)
end
```

Possible translation in LLVM

```
define i32 @sum(i32 %0,
                i32 %1) {
entry:
  %sum2 = add i32 42, %0
  %result = add i32 %sum2, %1
  ret i32 %result
}


define void @tc_main() {
entry__main:
  %call_sum =
     call i32 @sum(i32 5,
                   i32 7)
  ret void
}
```

Let's manually build this example

Step 1: Prepare your environment!

```
#include <iostream>
#include "llvm/IR/LLVMContext.h"
int main() {
    llvm::LLVMContext context;
    std::cout << &context
              << std::endl;
    return 0;
};
```

```
clang++ `llvm-config
                --cxxflags
                --libs engine`
        -std=c++17
        main.cc
```

Step 2: Build function sum (1/3)

```
llvm::Function* createSumFunction(Module* module) {
    LLVMContext &context = module->getContext();
    IRBuilder<> builder(context);

    // Define function's signature
    std::vector<Type*> Integers(2, builder.getInt32Ty());
    auto *funcType = FunctionType::get(builder.getInt32Ty(),
                                       Integers, false);

    // create the function "sum" and bind it to the
    // module with ExternalLinkage,
    // so we can retrieve it later
    auto *sumFunc = Function::Create(
        funcType, Function::ExternalLinkage, "sum", module
    );
```

Step 2: Build function sum (2/3)

```
    // Define the entry block and fill it with an appropriate
    auto *entry = BasicBlock::Create(context,
                                     "entry",
                                     sumFunc);
    builder.SetInsertPoint(entry);

    // Define constant equal to 42
    Value *constant = ConstantInt::get(builder.getInt32Ty(),
                                       42);
```

Step 2: Build function sum (3/3)

```
    // Retrieve arguments and proceed with further adding...
    auto args = sumFunc->arg_begin();
    Value *arg1 = &(*args);
    args = std::next(args);
    Value *arg2 = &(*args);
    auto *sum = builder.CreateAdd(constant, arg1, "sum");
    auto *result = builder.CreateAdd(sum, arg2, "result");

    // ...and return
    builder.CreateRet(result);

    // Verify at the end
    verifyFunction(*sumFunc);
    return sumFunc;
};
```

Step 3: putting it all together (1/3)

```
int main(int argc, char* argv[]) {
    // Initialize native target
    llvm::TargetOptions Opts;
    InitializeNativeTarget();
    InitializeNativeTargetAsmPrinter();

    LLVMContext context;
    auto myModule = std::make_unique<Module>
                        ("My First JIT", context);
    auto* module = myModule.get();
```

Step 3: putting it all together (2/3)

```
    // Create JIT engine
    llvm::EngineBuilder factory(std::move(myModule));
    factory.setEngineKind(llvm::EngineKind::JIT);
    factory.setTargetOptions(Opts);
    factory.setMCJITMemoryManager(std::move(MemMgr));
    auto executionEngine =
                std::unique_ptr<llvm::ExecutionEngine>
                        (factory.create());
    module->setDataLayout(executionEngine->getDataLayout());
```

This part is code bloat!

Step 3: putting it all together (2/3)

```
    auto* func = createSumFunction(module);
    executionEngine->finalizeObject();
    module->print(llvm::errs(), nullptr);

    // Get raw pointer
    auto* raw_ptr = executionEngine
                      ->getPointerToFunction(func);
    auto* func_ptr = (int(*)(int, int))raw_ptr;

    // Execute
    int arg1 = 5;
    int arg2 = 7;
    int result = func_ptr(arg1, arg2);
    std::cout << "RESULT => "<< result << std::endl;
    return 0;
}
```

Step 3: putting it all together (3/3)

```
define i32 @sum(i32 %0, i32 %1) {
entry:
  %sum2 = add i32 42, %0
  %result = add i32 %sum2, %1
  ret i32 %result
}


RESULT => 54
```

Remarks

The translation from LIR to LLVM can be done though a visitor

LIR already contains $fp references that must be handled during the translation (with alloca, store, load)

Constant folding

LLVM IR Optimizer can do a lot of things!
For instance constant folding.

Try it by yourself:

```
return a + b + 21 + 21;
```

```
Value *constant =
  ConstantInt::get
  (builder.getInt32Ty(), 21);
auto *tmp =
  builder.CreateAdd
  (constant, constant, "tmp");
```

Summary

LLVM