LLVM & Backend Optimization

Colton Williams, Megan Greening,
Kathy Grimes, and Aniq Shahid

Our Python Class Compiler

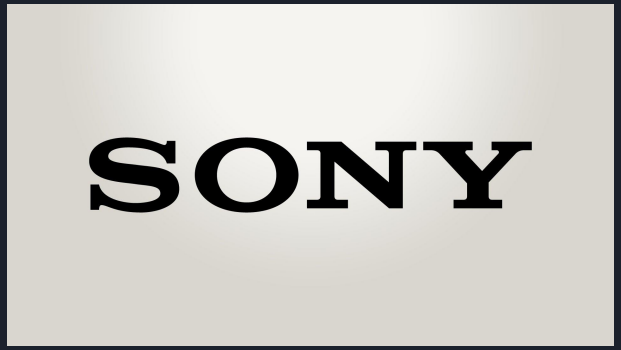
- Inefficient
- Optimizations frequently ignored
- What if we want to compile a different source language?

LLVM

“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.” -llvm.org

What is LLVM?

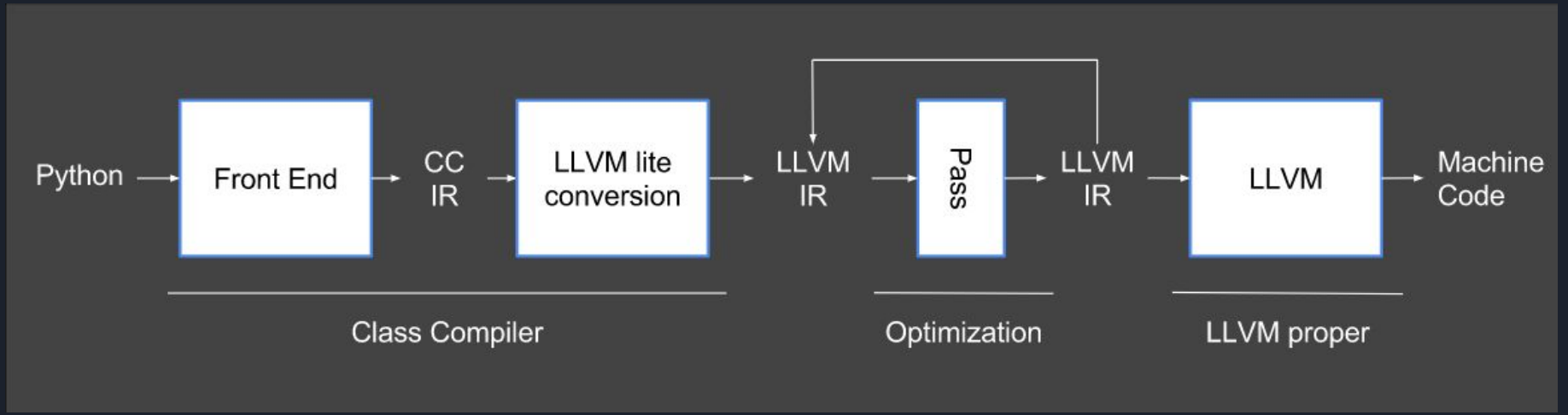
- Compiler infrastructure project
  - Modular
  - Separation of compiler frontend and backend
- Commonly used in industry, so a little more “real world”



LLVM Project Goals

- Building a Python subset specializer for LLVM
- Build and Test simple optimizations
- Look at improvements
  - Length of assembly code
  - Time taken to compile programs

LLVM Project Pipeline

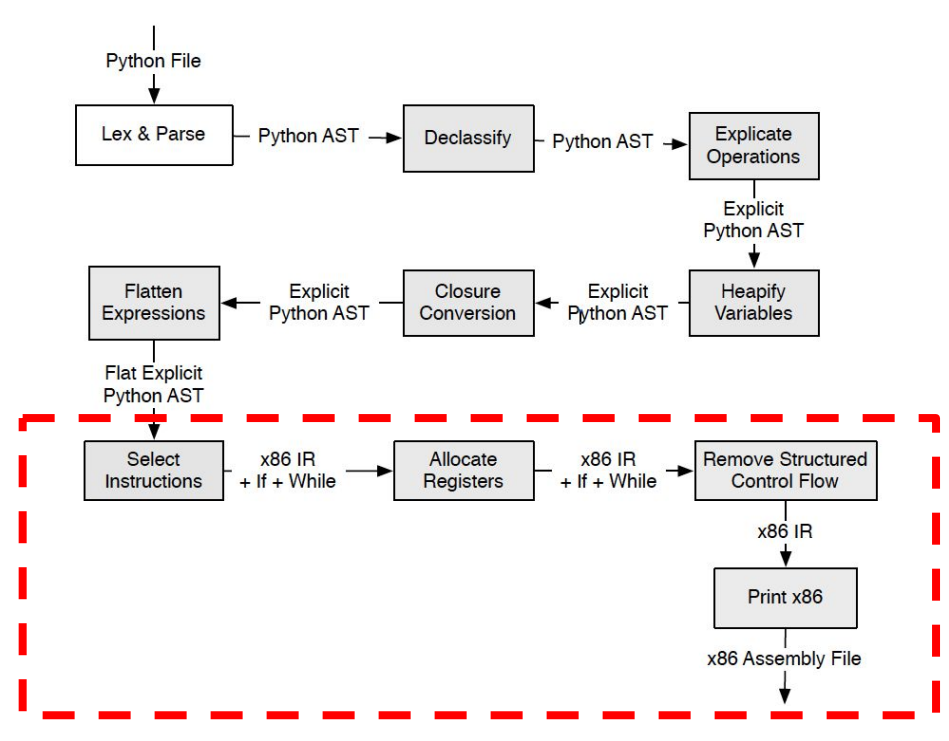


Generating LLVM IR

Input Python Program	Output LLVM IR

a = 1 + 2	define i32 @"main"()
	{
	entry:
	  %".2" = call i64 @"inject_int"(i32 1)
	  %".3" = call i64 @"inject_int"(i32 2)
	  %".4" = call i64 @"llvm_runtime_add"(i64 %".2", i64 %".3")
	  ret i32 0
	}

Using LLVM to Implement P0-P1



Generating LLVM IR

Input Python Program	Output LLVM IR

a = 1 + 2	define i32 @"main"()
	{
	entry:
	  %".2" = call i64 @"inject_int"(i32 1)
	  %".3" = call i64 @"inject_int"(i32 2)
	  %".4" = call i64 @"llvm_runtime_add"(i64 %".2", i64 %".3")
	  ret i32 0
	}

Generating LLVM IR

Input Python Program	Output LLVM IR

tmp0 = inject_int(1)
tmp1 = inject_int(2)
tmp3 = llvm_runtime_add(tmp0, tmp1)
a = tmp3
	define i32 @"main"()
	{
	entry:
	  %".2" = call i64 @"inject_int"(i32 1)
	  %".3" = call i64 @"inject_int"(i32 2)
	  %".4" = call i64 @"llvm_runtime_add"(i64 %".2", i64 %".3")
	  ret i32 0
	}

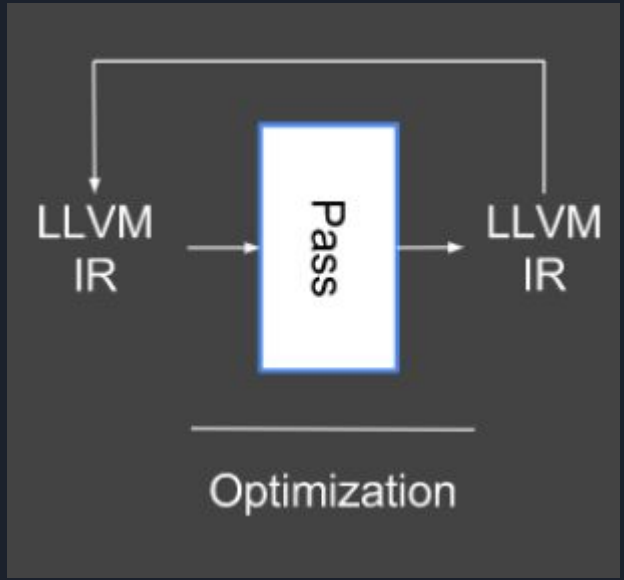
Generating LLVM IR

Input Python Program	Output LLVM IR

Import llvmlite as ir
 	
tmp0 = builder.call(inj_f, [ir.Constant(1)])
tmp1 = builder.call(inj_f, [ir.Constant(2)])
tmp2 = builder.call(llvm_runtime_add_f,
[tmp0, tmp1])
	define i32 @"main"()
	{
	entry:
	%".2" = call i64 @"inject_int"(i32 1)
	%".3" = call i64 @"inject_int"(i32 2)
	%".4" = call i64 @"llvm_runtime_add"(i64 %".2", i64 %".3")
	ret i32 0
	}

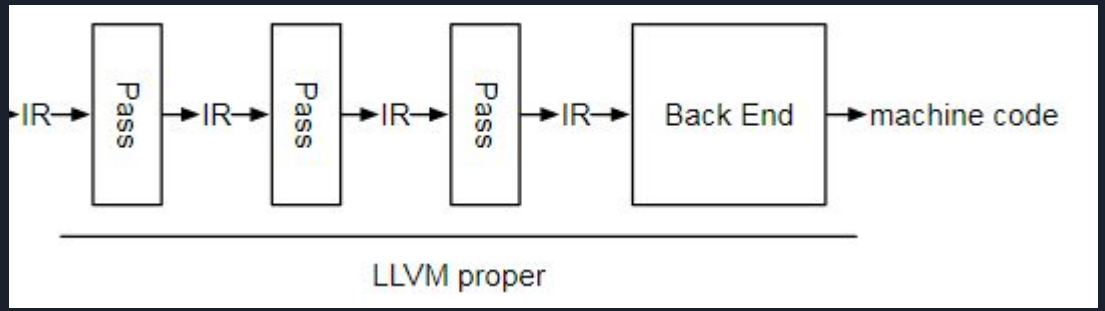
Optimizations

- LLVM Pass



LLVM Passes

- Chaining Passes



- Kinds of Passes
  - Analysis Pass
  - Transform Pass

In-Built Constant Propagation Pass

Python Code
r = 1 + 5 print r

Pre-optimized IR	Post Constant Propagation Pass IR

define void @"main"()
{
entry:
  %"res" = add i32 1, 5
  call void @"print_int_nl"(i32 %"res")
  ret void
}
 	
declare void @"print_int_nl"(i32 %".1")
	define void @main()
	{
	entry:
	  call void @print_int_nl(i32 6)
	  ret void
	}
	
	declare void @"print_int_nl"(i32 %".1")

Other Built-in Optimizations Applied

- InstCombine : combine redundant instructions
- ConstMerge : merge duplicate constants
- Dce : dead code eliminator

Custom Optimizations

Succeeded in implementing Analysis pass:

- InstCount : count different instructions in a program

Python Code	Post InstCount pass output
a = 1 + 1 b = 2 + 1 c = 1 + a print c	Function main add: 3 call: 1 ret: 1

Custom Optimizations

Working on implementing Transform pass:

- Addition Simplification
- Simplify multiple additions
- Stretch goal: avoid putting small lists on the heap

Time for Questions!

If you don’t have any, ask us this: What does LLVM stand for?

Challenges

- LLVM setup
- Docker setup
- Understanding LLVM-IR
- Understanding LLVM lite
- Writing our own optimization passes