Compiler Construction
Winter 2020

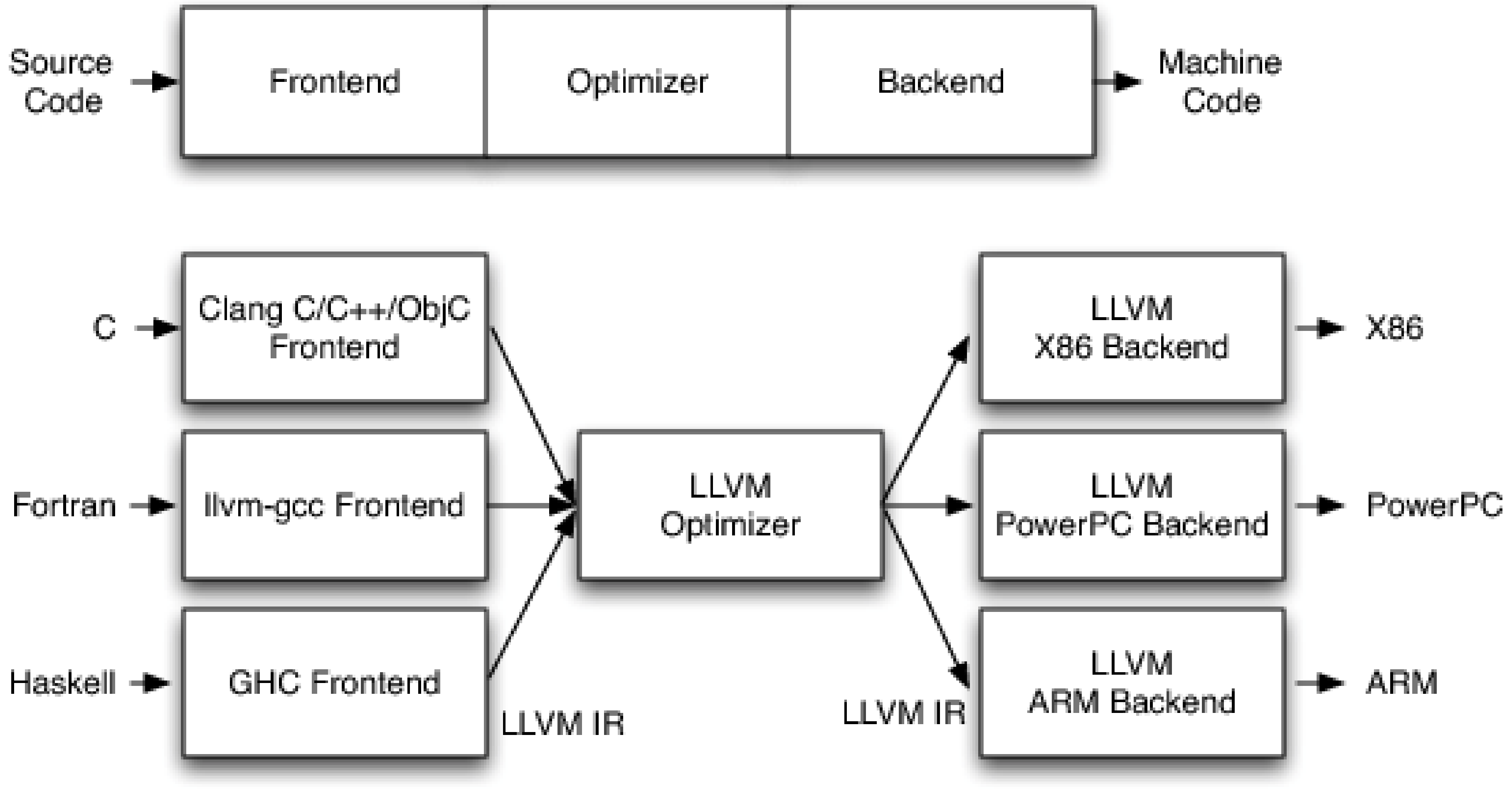
Recitation 4:
LLVM IR

Yotam Feldman

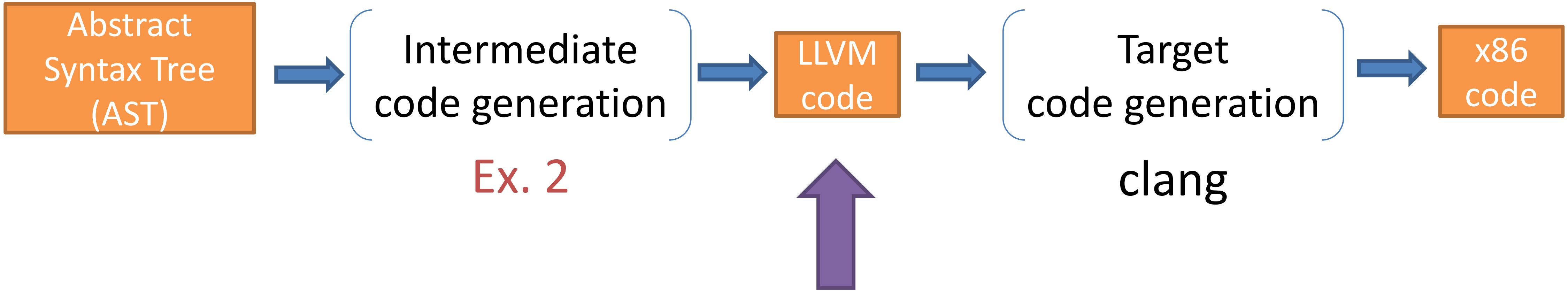
http://www.aosabook.org/en/llvm.html

http://www.llvm.org/docs/LangRef.html

Retargetability



Intermediate Representation



Today: the LLVM IR/Assembly language

LLVM Assembly/IR

“**LLVM IR is designed to host mid-level analyses and transformations** that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that **it is itself defined as a first class language with well-defined semantics**.”

Hello World

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

```
unsigned add1(unsigned a, unsigned b)
{
  return a+b;
}
```

Hello World

```
define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

```
unsigned add2(unsigned a, unsigned b)
{
       if (a == 0) return b;
       return add2(a-1, b+1);
}
```

Identifiers

- Prefix:
  - @: global identifiers (global vars, functions)
  - %: local identifiers (register names, labels)
- Compilers don’t need to worry about name clashes with reserved words
  - Set of reserved words can be extended
- Named temporaries start with a character
  - %res or %_0 vs. %0

Registers / Temporaries

Demo

```
define i32 @bar(i32 %a, i32 %b) {
  %_0 = add i32 %a, 1
  %_1 = mul i32 %_0, %_0
  %_2 = sub i32 %_1, %b
  ret i32 %_2
}
```

Unbounded number of temporaries
(register allocation in compilation to assembly)

Static Single Assignment (SSA)

Demo

- Each temporary variable assigned **exactly once**
- Simplifies and enhances compiler optimizations
- A requirement of LLVM IR

```
define i32 @bar(i32 %a, i32 %b) {
  %_0 = add i32 %a, 1
  %_0 = mul i32 %_0, %_0
  %_2 = sub i32 %_0, %b
  ret i32 %_2
}
```

Binary Operations

- LLVM IR is 3 address code
- Type matters

```
…
%_0 = and i32 %a, 15
%_1 = add i1 %b, 1
…
```

- (Numeric casts)

Stack Variables

- Memory on the stack frame of the current executing function
- Automatically released when the function returns

- Not SSA
  - Can’t use registers for e.g. loop counter
- Has an address available
  - C code that takes that address of the variable
  - Address arithmetic

Stack Variables

Demo

```
%p = alloca i32
…
store i32 %a, i32* %p
…
%res = load i32, i32* %p
```

Jumps and Conditional Jumps

Demo

```
define i32 @bar(i32 %a, i1 %b) {
  %p = alloca i32
  br i1 %b, label %then, label %else

then:
  %_0 = add i32 %a, 1
  store i32 %_0, i32* %p
  br label %join

else:
  %_1 = sub i32 %a, 1
  store i32 %_1, i32* %p
  br label %join

join:
  %res = load i32, i32* %p
  ret i32 %res
}
```

Instruction before a label must be br!

Function Calls

```
define i32 @double(i32 %x) {
     %_0 = mul i32 %x, 2
     ret i32 %_0
}

define i32 @bar(i32 %a, i32 %b) {
  %_0 = call i32 @double(i32 %a)
  %result = add i32 %_0, %b
  call void @print_int(i32 %result)
  ret i32 %result
}
```

Demo

Abstracts away asm calling conventions

Recursion

```
define i32 @factorial(i32 %a) {
  %_0 = icmp eq i32 %a, 0
  br i1 %_0, label %then, label %else

then:
  ret i32 1

else:
  %_1 = sub i32 %a, 1
  %_2 = call i32 @factorial(i32 %_1)
  %_3 = mul i32 %_2, %a
  ret i32 %_3
}
```

Demo

Heap Allocation + Bitcasting

Demo

```
declare i8* @calloc(i32, i32)
  …
  %v = call i8* @calloc(i32 1, i32 8)
  %p = bitcast i8* %v to i32*
  …
  store i32 %a, i32* %p
  …
  %res = load i32, i32* %p
```

Array Types

- [40 x i32] – array of 40 32-bit elements
- [7 x i8*] – array of 7 i8* elements
- [3 x [4 x i32]] – 3x4 array of 32-bit elements

- The number of elements is a constant integer value

getelementptr

Demo

```
@.array = global [5 x i32] [i32 0, i32 1, i32 2, i32 3, i32 4]
...
  %_1 = getelementptr [5 x i32], [5 x i32]* @.array, i32 0, i32 3
  %_2 = load i32, i32* %_1
```

- The first type indexed into must be a pointer value, subsequent types can be arrays, (vectors, and structs).
- Subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing calculation.

Example: Function Pointers

Demo

And Much More!

- Non-int constants
- struct types
- **Many** levers
- …

Summary

- LLVM IR
  - Unbounded number of registers
  - SSA
  - Typed
- Up next: compiling to LLVM IR