CS443: Compiler Construction

Lecture 5: LLVM

Stefan Muller

Based on lecture material by Steve Chong, Steve Zdancewic and Greg Morrisett

LLVM is a very general compiler framework

- 2003, University of Illinois Urbana-Champaign
    - Chris Lattner, Vikram Adve

- Originally: “Low-Level Virtual Machine”
    - Now: not really relevant, no longer an acronym

- Based around LLVM IR

Features of LLVM IR

- Compiler/language-independent
- Typed(!)
- Static Single Assignment
  - Briefly: every variable can be assigned to only once.
  - (Yes, this is a big restriction; we have a whole lecture on it later)

In this lecture/class: LLVM <15.0.0

- It’s pretty old
- I provide an interpreter

Variables

- Can be global (start with @) or local (start with %)
- (Won't be using globals much)
- Locals defined with instructions of the form `%x = …`
    - Can't be redefined (like `let x = …` in OCaml)

Types

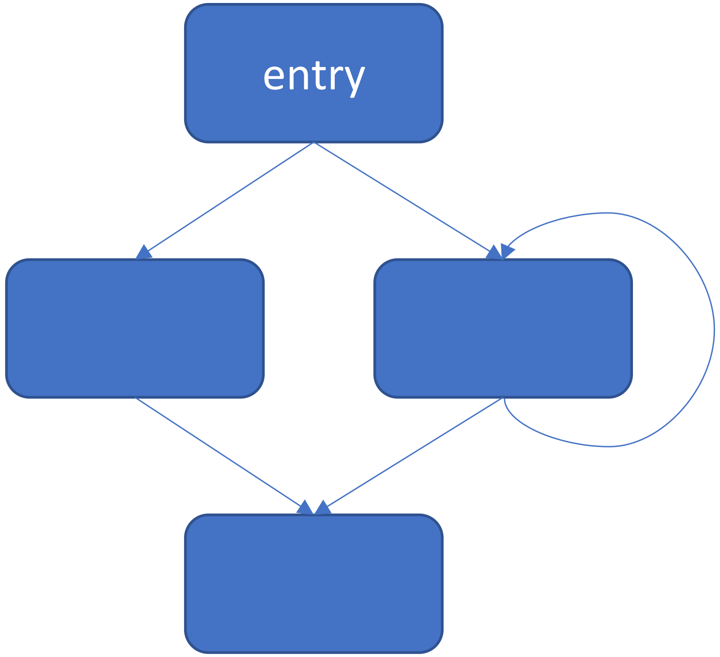
- Void (`void`)
- Integers (`iN`)
  - N specifies the number of bits in the integer
  - i1, i8, i16, i32, i64, …
- A bunch of other “first class types” (floats, etc.)
- Pointers to a type (`t*`)
- Functions (e.g, `i32(i32, i1)`)
- Labels (i.e., code addresses) (`label`)

Structured as *functions* consisting of a number of *basic blocks*

```
define i32 myfunc (i32 %myarg) {
 …
}
```

Basic blocks are sequences of *instructions* that execute starting at the beginning

- (i.e., can’t jump to the middle of a basic block)
- Flat structure



Structured as *functions* consisting of a number of *basic blocks*

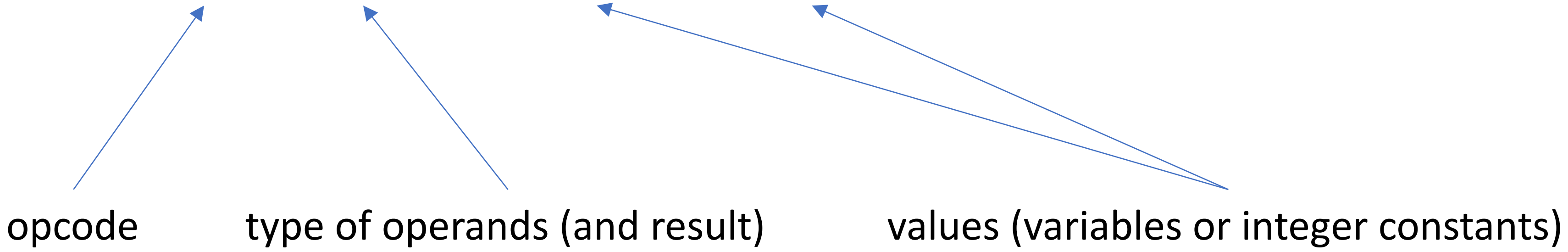
```
define i32 myfunc (i32 %myarg) {
  myfunc__entry:
    …
  block1:
    …
  block2:
    …
}
```

Instructions perform one operation

- General types: terminators, arithmetic operations, memory operations, type conversions
- Basic blocks must end with a terminator (doesn't usually return a value, jumps somewhere else)
- Most other instructions return a value
- General form: `%dest = <opcode> <ty1> <op1>, <ty2> <op2>, …`

Instructions perform one operation (e.g. arithmetic operations)

- `<var> = add <ty> <op1> <op2>`



opcode

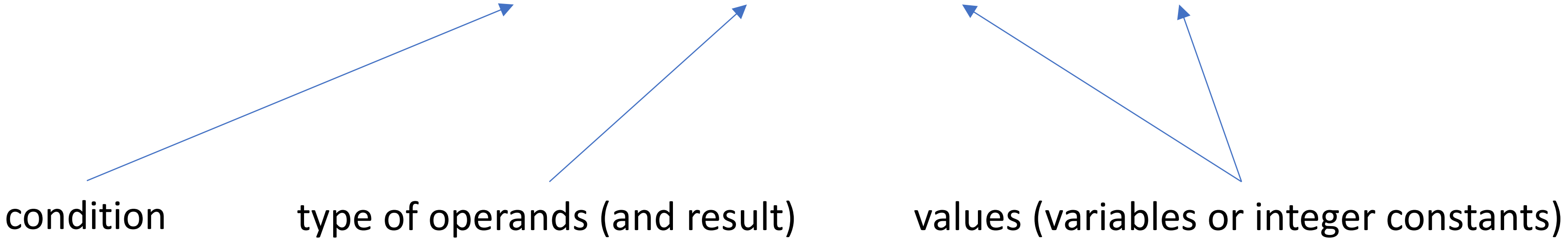
type of operands (and result)

values (variables or integer constants)

- ex. `%dest = add i32 %x 1`
- (Declares `%dest as i32`)
- Similar: `sub, mul, udiv, sdiv, …`

Instructions perform one operation (e.g. comparison operations)

- `<dest> = icmp <cond> <ty> <op1> <op2>`



condition

type of operands (and result)

values (variables or integer constants)

- Conditions: `eq, ne, (u/s)gt, (u/s)ge, (u/s)lt, (u/s)le`
- Result type: `i1`

Terminators

- `ret void`
- `ret <ty> <op>`
- Return from the current function

- `br label <dest>`
- Jump to the label
- `br i1 <op>, label <truedest>, label <falsedest>`
- Jump to either label depending on the value of the condition

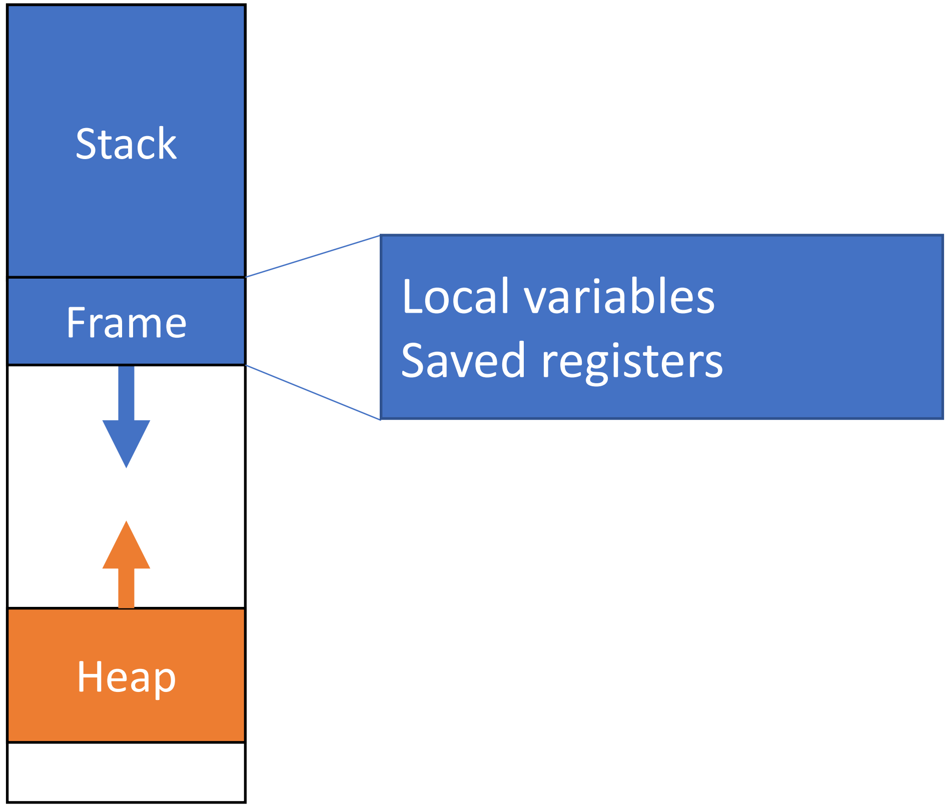
Terminators (cont'd)

-
  ```
  switch <ty> <val> label <default>
  [<ty> <val1>, label <label1>
   <ty> <val2>, label <label2> …
  ]
  ```

Must all be ints (of the same size?)

Actual syntax, not optional arguments

LLVM Memory Model



Local variables
Saved registers

```
void foo () {
  int a = 1;
  int b = 2;
  return 3 + bar(a, b);
}

void bar(int a, int b) {
  return a + b;
}
```

LLVM Memory Model

- `%ptr = alloca <ty>, <intty> <num>`
- Allocates **stack** space for <num> elements of type <ty>
- Returns type `<ty>*` - the **stack address** of the allocated memory
- Access memory with `load, store`

    -
      ```
      %ptr = alloca i64
      store i64 443, i64* %ptr
      %bestclass = load i64, i64* %ptr
      ```
- (`Load, store`) work the same for heap but allocate using malloc

Type conversions

- `<dest> = trunc <ty1> <value> to <ty2>` Truncate
- `<dest> = zext <ty1> <value> to <ty2>` Zero-extend
- `<dest> = sext <ty1> <value> to <ty2>` Sign-extend

(Copy highest bit)

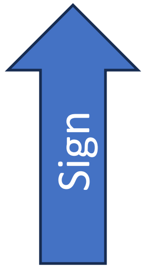
- `<dest> = bitcast <ty1> <value> to <ty2>`
  Bitwise conversion (doesn't change value at all)

- `<dest> = inttoptr <intty> <value> to <ptrty>`
- `<dest> = ptrtoint <ptrty> <value> to <intty>`

Refresher(?): Two's complement (8-bit)

To convert(/multiply by -1):
Invert bits, add 1

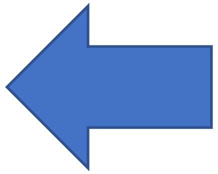
-1	1	1	1	1	1	1	1	1
-2	1	1	1	1	1	1	1	0
-3	1	1	1	1	1	1	0	1
…								
-2^7	1	0	0	0	0	0	0	0



Can just add two's complement
#s without casing on sign!

Two’s complement means two ways to extend integers to the left

```
1010101
```



- If signed int: want to sign-extend (extend with MSB)
  - LLVM: sext
  - 101 as 3-bit int = -3 = 11101 as 5-bit int

- If unsigned: want to zero-extend (extend with 0s)

Example

```
int factorial (int n) {
  int result = 1;
  while (n > 1) {
    result = result * n;
    n = n - 1;
  }
  return result;
}
```

```
define i32 @factorial(i32 %0) {
  %n = alloca i32
  %result = alloca i32
  store i32 %0, i32* %n
  store i32 1, i32* %result
  br label %4
```

```
4:
  %n1 = load i32, i32* %n
  %temp1 = icmp sgt i32 %n1, 1
  br i1 %temp1, label %7, label %11

7:
  %result1 = load i32, i32* %result
  %n2 = load i32, i32* %n
  %temp2 = mul i32 %result1, %n2
  store i32 %temp2, i32* %result
  %temp4 = sub i32 %n2, 1
  store i32 %temp4, i32* %n
  br label %4

11:
  %temp3 = load i32, i32* %result
  ret i32 %temp3
}
```