四.ir学习

（1）[北航教程][https://buaa-se-compiling.github.io/miniSysY-tutorial/pre/llvm_ir_quick_primer.html]

1.快速入门

```
clang -S -emit-llvm test.c
```

生成中间代码

```
clang -S -emit-llvm -O3 test.c
```

o3优化

```
llc test.ll
```

生成汇编代码

然后用操作系统自带的汇编器和链接器生成可执行文件

这是一个基于llvm的编译器

```
.c --frontend--> AST --frontend--> LLVM IR --LLVM opt--> LLVM IR --LLVM llc--> .s Assembly --OS Assembler--> .o --OS Linker--> executable
```

手册https://llvm.org/docs/LangRef.html

- 关于llvm 和clang的关系

Clang 作为 LLVM project 的一个子项目，是 LLVM 项目中 `c/c++/obj-c` 语言的前端，其用法与 GCC 基本相同

用ir的好处：

1.有一些优化技术是目标平台无关的（例如作为我们实验挑战任务的死代码删除和常量折叠），我们只需要在 IR 上做这些优化，再翻译到不同的汇编，可以减少工作量

2.减少编译器的数量

前端：源语言变成ir

中端:ir的优化

后端：生成ricsv(编译成目标语言)

llvm有三种表达形式 其中.ll是人可读的代码语言 （其他形式的也是等价的）

- 读llvm

`align` 字段描述了程序的对齐属性； `dso_local` 是变量和函数的的运行时抢占说明符；以 `;` 开头的字符串是 LLVM IR 的注释（这个最好记住）

alloc 是指针

局部变量的作用域是单个函数

全局变量与局部变量由前缀区分，全局变量和函数名以 `@` 为前缀，局部变量以 `%` 为前缀

内联函数： 作为代码直接插入 省去了跳转的代价 是优化的一个方面

- br指令

```
br + 标志位 + truelabel + falselabel`，或者`br + label（强制跳转）
```

- icmp

```
%9 = icmp eq i32 %7, %8
```

相等置为1 不相等置为0

类型系统 llvm的优秀之处

integer type i1 i32

label type

后面还有数组 type和变量type

2.ssa

存取内存形式的 IR 以及带有 `phi` 指令的 SSA IR

静态单赋值（Static Single Assignment, **SSA**）

且每个变量只能被赋值一次（如果套用 C++ 的术语，就是说每个变量只能被初始化，不能被赋值）类似rust...?

- 对于for循环 load/store形式可以改为Phi形式

```
3:                                                ; preds = %5, %1
  %4 = phi i32 [ 1, %1 ], [ %8, %5 ]              ; 如果是从块 %1 来的，那么值就是
1，如果是从
  ret i32 %4                                      ; 块 %5 来的，那么值就是 %8 的值
```

内存四字节对齐

```
align 4
```

```
//把内存内容load进入寄存器
%6 = load i32, i32* %4, align 4
```

`alloca` 指令的作用是在当前执行的函数的栈帧上分配内存并返回一个指向这片内存的指针，当函数返回时内存会被自动释放（一般是改变栈指针）。

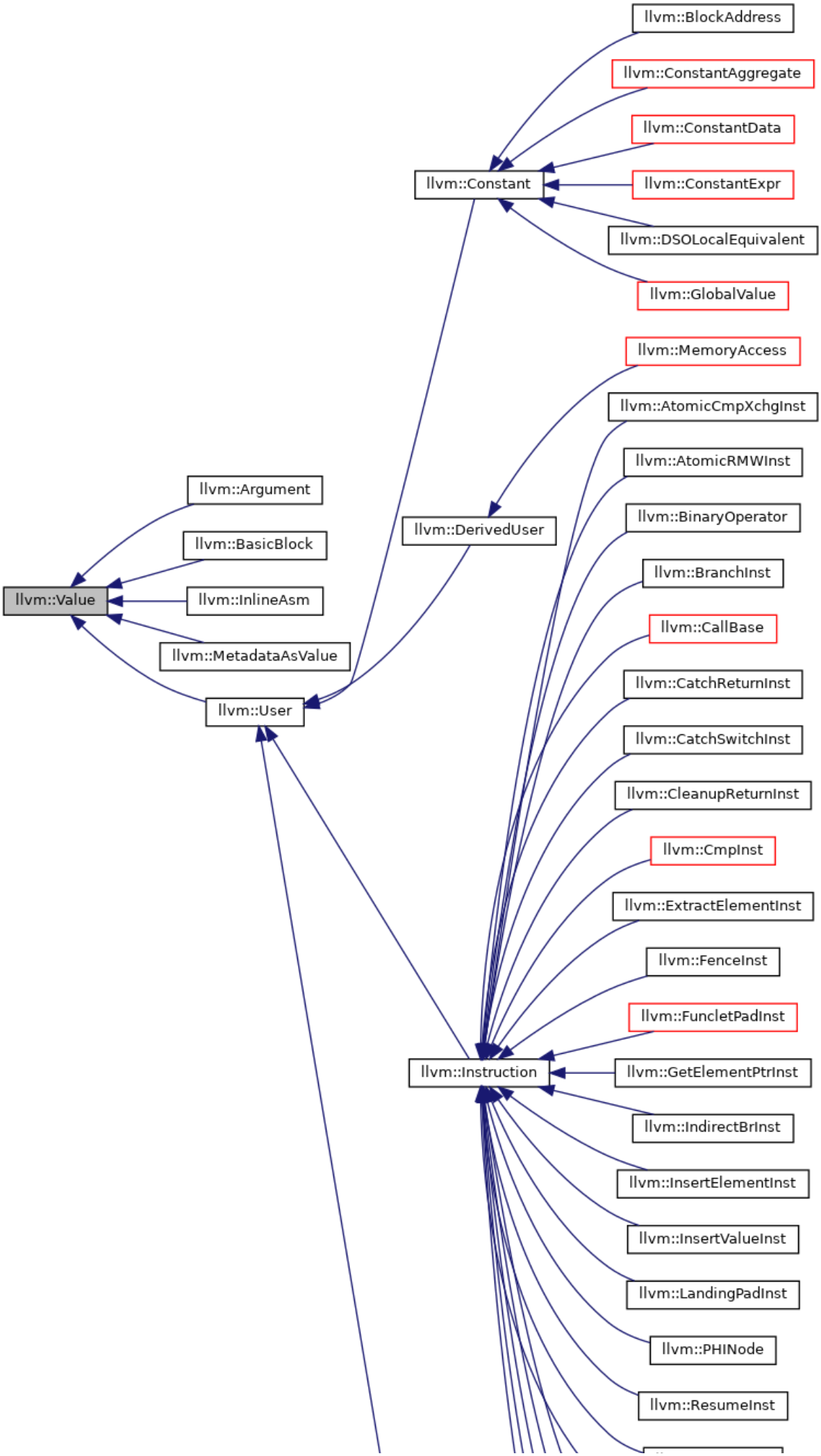
由于内存不需要ssa,可以经受多次数值的改变 这就是load store操作-

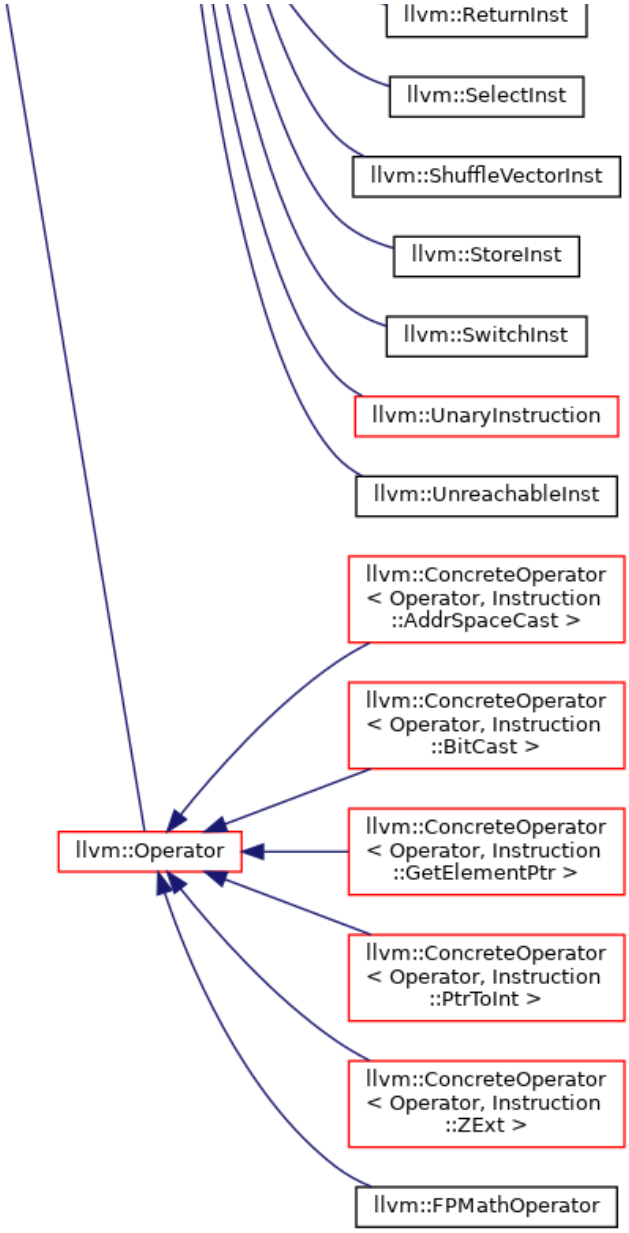
- 论`alloca + mem2reg`技术

alloca 在内存上可以无限次更改，但是频繁的内存访问会导致性能较差

mem2reg 对于每一个寄存器只能够一次赋值 要求比较高

3.value use user


llvm::BlockAddress
llvm::ConstantAggregate
llvm::ConstantData
llvm::Constant
llvm::ConstantExpr
llvm::DSOLocalEquivalent
llvm::GlobalValue
llvm::MemoryAccess
llvm::AtomicCmpXchgInst
llvm::AtomicRMWInst
llvm::Argument
llvm::DerivedUser
llvm::BinaryOperator
llvm::BasicBlock
llvm::BranchInst
llvm::Value
llvm::InlineAsm
llvm::CallBase
llvm::MetadataAsValue
llvm::CatchReturnInst
llvm::User
llvm::CatchSwitchInst
llvm::CleanupReturnInst
llvm::CmpInst
llvm::ExtractElementInst
llvm::FenceInst
llvm::FuncletPadInst
llvm::Instruction
llvm::GetElementPtrInst
llvm::IndirectBrInst
llvm::InsertElementInst
llvm::InsertValueInst
llvm::LandingPadInst
llvm::PHINode
llvm::ResumeInst



instructions

llvm ir	usage	intro
add	`<result> = add <ty> <op1>, <op2>`	/
sub	`<result> = sub <ty> <op1>, <op2>`	/
mul	`<result> = mul <ty> <op1>, <op2>`	/
sdiv	`<result> = sdiv <ty> <op1>, <op2>`	有符号除法
icmp	`<result> = icmp <cond> <ty> <op1>, <op2>`	比较指令
and	`<result> = and <ty> <op1>, <op2>`	与
or	`<result> = or <ty> <op1>, <op2>`	或
call	`<result> = call [ret attrs] <ty> <fnptrval>(<function args>)`	函数调用
alloca	`<result> = alloca <type>`	分配内存
load	`<result> = load <ty>, <ty>* <pointer>`	读取内存
store	`store <ty> <value>, <ty>* <pointer>`	写内存
getelementptr	`<result> = getelementptr <ty>, * {, [inrange] <ty> <idx>}*` `<result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, [inrange] <ty> <idx>}*`	计算目标元素的位置（仅计算）
phi	`<result> = phi [fast-math-flags] <ty> [ <val0>, <label0>], ...`	
zext..to	`<result> = zext <ty> <value> to <ty2>`	类型转换，将 `ty` 的 `value` 的type转换为 `ty2`

terminator insts

llvm ir	usage	intro
br	`br i1 <cond>, label <iftrue>, label <iffalse>` `br label <dest>`	改变控制流
ret	`ret <type> <value>` , `ret void`	退出当前函数，并返回值（可选）

（2）[github repo教程][https://github.com/Evian-Zhang/llvm-ir-tutorial/blob/master/LLVM]

1.数据表示

- 栈上指针 比如在函数内部malloca

在栈上创建数组时，不能使用含有变量的表达式，如：int a[x+2];错误

原因：在栈上创建数组时编译器编译的时候就需要在栈上分配内存，可是有了变量以后，编译器就无法知道该分配多大的内存空间，故编译器会报错。但是定义一般变量如：int a；编译器会自动识别 int 占多大内存分配给他。

对比：如果是动态创建数组时（也就是在堆上创建数组时）可以出现变量如：new a [x+1]；正确；
原因：在堆上创建数组时，编译器不会在编译的时候为它分配内存，而是在程序运行的时候为它分配内存，我们可以知道，程序运行时变量的值就会明确是多少，故动态创建数组时可以出现变量；

堆里的是动态数组，是程序运行过程中动态加载的，而栈不一样，申请数组必须要是确定大小的的数字，在编译时就要确定下来，如果你const int x=7就不会报错

```
+------------------------------+
|          stack_data          |
|         heap_pointer         |  <------------- stack
+------------------------------+
|                              |
|                              |  <------------- available memory space
|                              |
+------------------------------+
| data pointed by heap_pointer |  <------------- heap
+------------------------------|
|          global_data         |  <------------- .DATA section
+------------------------------+
```

堆中的数据不会独立存在 一般要不是栈上的数据要不是全局变量的数据

程序中的数据类型如下

- 寄存器中的数据
- 栈上的数据
- 数据区里的数据

数据区数据

```
i32 类型的只读全局变量和全局变量
@global_constant = constant i32 0
@global_variable = global i32 0
```

如果不考虑内存地址操作 栈可以认为是扩大的寄存器

*x86的一些特性

将两者中比较大的那个局部变量存储在栈上的-4(%rbp)上（由于x86_64架构不允许直接将内存中的一个值拷贝到另一个内存区域中，所以得先把内存区域中的值拷贝到eax寄存器里，再从eax寄存器里拷贝到目标内存中）

正是因为内存操作过于复杂 llvm引入了虚拟寄存器的概念

- 寄存器的使用

LLVM IR内部自动帮我们做了这些事。如果我们把所有没有被保留的寄存器都用光了，那么LLVM IR会帮我们把这些被保留的寄存器放在栈上，然后继续使用这些被保留寄存器。当函数退出时，会帮我们自动从栈上获取到相应的值放回寄存器内。

- 全局变量和栈上的变量都是指针

```
@global_variable = global i32 0
```

这个操作包含了alloca 但@global_variable存储 相当于一个寄存器

```
%1 = load i32, i32* @global_variable
store i32 1, i32* @global_variable
```

都用指针操作

```
nsw 是 "No Signed Wrap" 的缩写，表示无符号值运算
```

add i32指的是三个的类型

2.类型系统

聚合类型

数组

```
int a[4]
```

```
%a = alloca [4 x i32]
```

结构体

```
struct MyStruct {
    int x;
    char y;
};
```

```
%MyStruct = type {
    i32,
    i8
}
```

无论是数组还是结构体，其作为全局变量或栈上变量，依然是指针

对聚合类型进行操作：使用getelementptr

llvm是强类型语言

```
struct MyStruct {
    int x;
    int y[5];
};

struct MyStruct my_structs[4];
```

那么如果我们想获得`my_structs[2].y[3]`的地址，只需要

```
%MyStruct = type {
    i32,
    [5 x i32]
}
%my_structs = alloca [4 x %MyStruct]

%1 = getelementptr [4 x %MyStruct], [4 x %MyStruct]* %my_structs, i64 2, i32 1,
i64 3
```

我自己用clang编译的结果

```
@my_structs = dso_local global [4 x %struct.MyStruct] zeroinitializer, align 16
@my_structs_ptr = dso_local global %struct.MyStruct* getelementptr inbounds ([4 x
%struct.MyStruct], [4 x %struct.MyStruct]* @my_structs, i32 0, i32 0), align 8
@y = dso_local global i32 0, align 4
@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32, void
()*, i8* } { i32 65535, void ()* @_GLOBAL__sub_I_main.cpp, i8* null }]

; Function Attrs: noinline uwtable
define internal void @__cxx_global_var_init() #0 section ".text.startup" {
  %1 = load %struct.MyStruct*, %struct.MyStruct** @my_structs_ptr, align 8
  %2 = getelementptr inbounds %struct.MyStruct, %struct.MyStruct* %1, i64 2
  %3 = getelementptr inbounds %struct.MyStruct, %struct.MyStruct* %2, i32 0, i32
2
  %4 = load i32, i32* %3, align 4
  store i32 %4, i32* @y, align 4
  ret void
}
```

inbound是防止越界而使用的

`getelemtptr` expects the type that you are indexing (without the pointer) as it's first argument.

- 关于dso_local

```
dso_local
```

The compiler may assume that a function or variable marked as `dso_local` will resolve to a symbol within the same linkage unit. Direct access will be generated even if the definition is not within this compilation unit.

```
load 类型 源类型 地址 ,align 8
```

我们的指针只能够一层一层展开 暂时不能一层描述所有的情况

- 控制函数

比较指令

```
%comparison_result = icmp uge i32 %a, %b
bool comparison_result = ((unsigned int)a >= (unsigned int)b);
```

条件跳转

```
br i1 %comparison_result, label %A, label %B
```

无条件跳转

```
br label %start
```

basic block

- 一个函数由许多基本块(Basic block)组成
- 每个基本块包含：
  - 开头的标签（可省略）
  - 一系列指令
  - 结尾是终结指令
- 一个基本块没有标签时，会自动赋给它一个标签

phi指令：

```
%y = phi i32 [1, %btrue], [2, %bfalse]
```

根据从哪一个block来来进行赋值

前一个basic block是`%btrue`，那么返回`1`，如果前一个basic block是`%bfalse`，那么返回`2`