AI编译器系列

LLVM架构和原理

ZOMI

Talk Overview

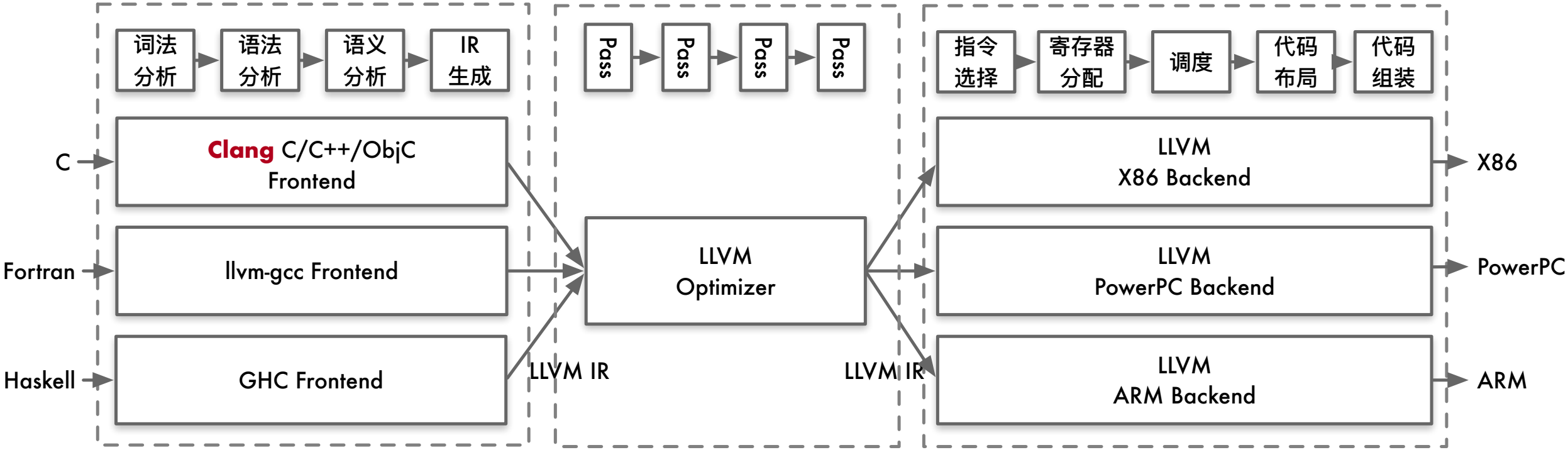
1. **传统编译器**
   - History of Compiler - 编译器的发展
   - GCC process and principle – GCC 编译过程和原理
   - LLVM/Clang process and principle – LLVM 架构和原理
2. **AI编译器**
   - History of AI Compiler – AI编译器的发展
   - Base Common architecture – AI编译器的通用架构
   - Different and challenge of the future – 与传统编译器的区别，未来的挑战与思考

Talk Overview

LLVM/Clang process and principle – LLVM 架构和原理

- LLVM 项目发展历史
- LLVM 基本设计原则和架构
- LLVM 中间表示 LLVM IR
- LLVM 前端过程
- LLVM 中间优化
- LLVM 后端生成
- 基于 LLVM 项目

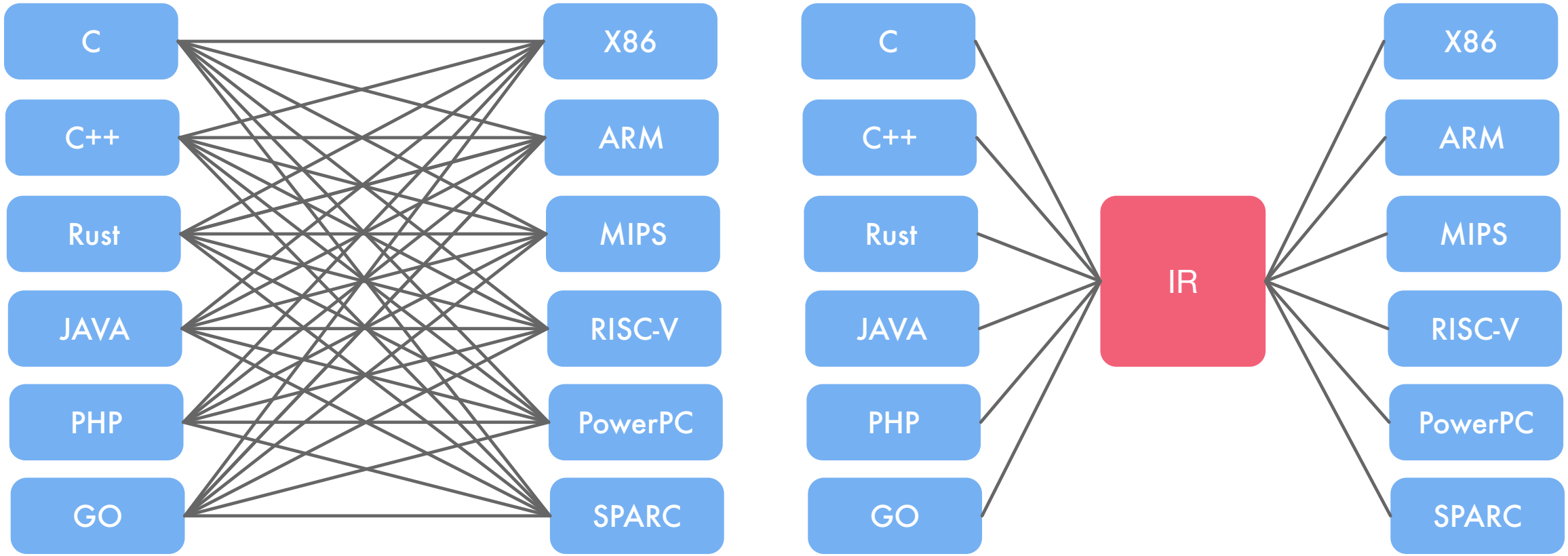
LLVM Architecture



LLVM IR

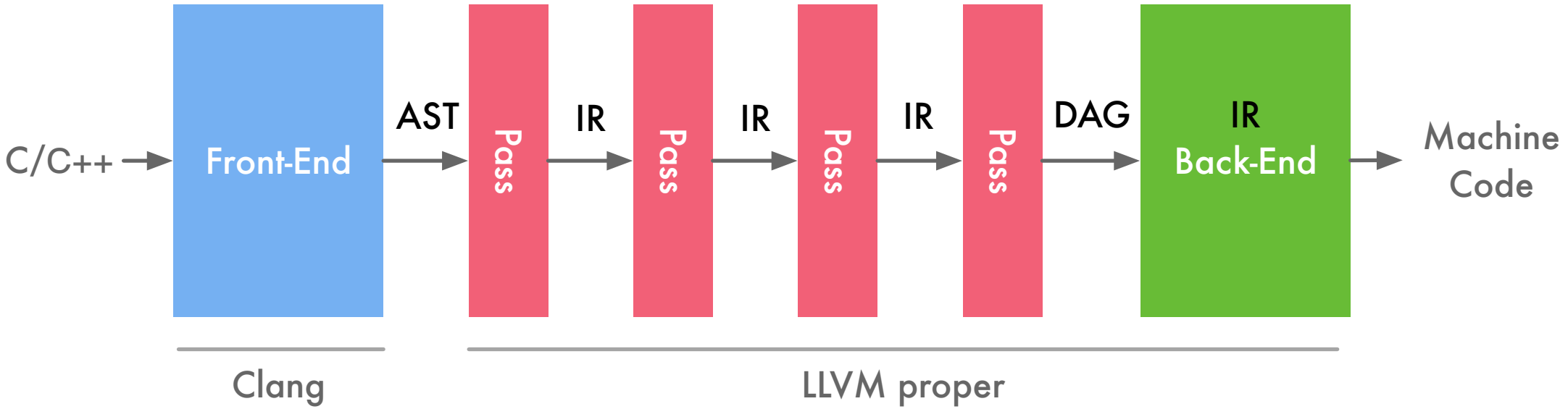
中间表达

LLVM IR



LLVM IR

这并不意味着LLVM使用单一 IR 表示方式， 在编译不同阶段会采用不同的数据结构：



LLVM IR 表示

```
1   void test(int a, int b){
2     int c = a * b + 100;
3   }
```

clang -S -emit-llvm test.c

LLVM IR 表示

```
; Function Attrs: noinline nounwind optnone ssp uwtable
define void @test(i32, i32) #2 { ; 有个全局函数@test (a,b)
  %3 = alloca i32, align 4 ; 局部变量 c
%4 = alloca i32, align 4 ; 局部变量 d
  %5 = alloca i32, align 4 ; 局部变量 e
  store i32 %0, i32* %3, align 4 ; %0 赋值给%3 c = a
  store i32 %1, i32* %4, align 4 ; %1 赋值给%4 d = b
  %6 = load i32, i32* %3, align 4 ; 读取%3,赋值给%6 就是函数参数a
  %7 = load i32, i32* %4, align 4 ; 读取%4,赋值给%7 就是函数参数b
  %8 = mul nsw i32 %6, %7 ; a * b
  %9 = add nsw i32 %8, 100 ; a * b + 100
  store i32 %9, i32* %5, align 4 ; 参数 %9 赋值给 %5 e ===> 就是转换前函数写的int c变量
  ret void
}
```

LLVM IR基本语法

1. 注释以;开头
2. 全局表示以@开头，局部变量以%开头
3. alloca在函数栈帧中分配内存
4. i32 32位 4个字节的意思
5. align 字节对齐
6. store写入
7. load读取

LLVM IR

LLVM IR 作为一种编译器 IR，它的两个基本原则指导着核心库的开发：

- SSA 表示，代码组织为三地址指令序列和无限寄存器让优化能够快速执行。
- 整个程序的 IR 存储到磁盘让链接时优化易于实现。

LLVM IR

LLVM IR 采用静态单赋值形式（Static single assignment，SSA），具有两个重要特征：

- 代码组织为三地址指令序列
- 寄存器数量无限制

What is SSA(Static Single Assignment) 静态单赋值

当程序中的每个变量都有且只有一个赋值语句时，称一个程序是 SSA 形式的。LLVM IR 中，每个变量都在使用前都必须先定义，且每个变量只能被赋值一次。以 1 * 2 + 3 为例：

```
1   %0 = mul i32 1, 2
2   %0 = add i32 %0, 3
3   ret i32 %0
```

```
1   %0 = mul i32 1, 2
2   %1 = add i32 %0, 3
3   ret i32 %1
```

每个值只有单一赋值定义了它。每次使用一个值，可以立刻向后追溯到给出其定义的唯一的指令。极大简化优化，因为SSA形式建立了平凡的use-def链，也就是一个值到达使用之处的定义的列表。

LLVM IR基本语法

- LLVM IR 是类似于精简指令集（RISC）的底层虚拟指令集；
- 和真实精简指令集一样，支持简单指令的线性序列，例如添加、相减、比较和分支；
- 指令都是三地址形式，它们接受一定数量的输入然后在不同的寄存器中存储计算结果；
- 与大多数精简指令集不同，LLVM 使用强类型的简单类型系统，并剥离了机器差异；
- LLVM IR 不使用固定的命名寄存器，它使用以 % 字符命名的临时寄存器；

三地址码

三地址码指令 Three-address code - Wikipedia

每个三地址码指令，都可以被分解为一个四元组（4-tuple）的形式：（运算符，操作数1，操作数2，结果），由于每个陈述都包含了三个变量，即每条指令最多有三个操作数，所以它被称为三地址码。

指令类型	指令形式	四元组表示
赋值指令	z = x op y（z = x + y）	(op , x , y , z)

LLVM IR 表示形式

LLVM IR 具有三种表示形式，这三种中间格式是完全等价的：

- 在内存中的编译中间语言（无法通过文件的形式得到的指令类等）
- 在硬盘上存储的二进制中间语言（格式为 .bc）
- 人类可读的代码语言（格式为 .ll）

LLVM IR 内存模型

- LLVM IR 文件的基本单位称为 module
- 一个 module 中可以拥有多个顶层实体，比如 function 和 global variavle
- 一个 function define 中至少有一个 basicblock
- 每个 basicblock 中有若干 instruction，并且都以 terminator instruction 结尾

LLVM IR 内存模型

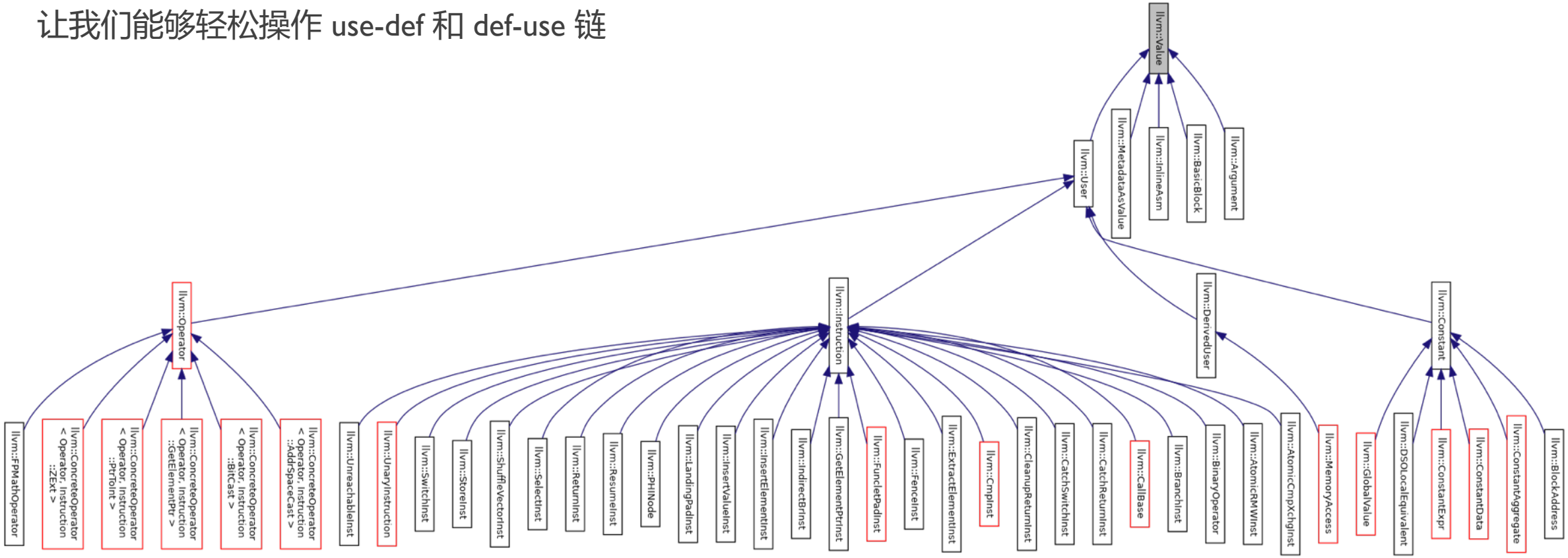
```
; Function Attrs: noinline nounwind optnone ssp uwtable
define void @test(i32, i32) #2 { ; 有个全局函数@test (a,b)
  %3 = alloca i32, align 4 ; 局部变量 c
 %4 = alloca i32, align 4 ; 局部变量 d
  %5 = alloca i32, align 4 ; 局部变量 e
  store i32 %0, i32* %3, align 4 ; %0 赋值给%3 c = a
  store i32 %1, i32* %4, align 4 ; %1 赋值给%4 d = b
  %6 = load i32, i32* %3, align 4 ; 读取%3,赋值给%6 就是函数参数a
  %7 = load i32, i32* %4, align 4 ; 读取%4,赋值给%7 就是函数参数b
  %8 = mul nsw i32 %6, %7 ; a * b
  %9 = add nsw i32 %8, 100 ; a * b + 100
  store i32 %9, i32* %5, align 4 ; 参数 %9 赋值给 %5 e ===> 就是转换前函数写的int c变量
  ret void
}
```

LLVM IR 内存模型

Module	Module类聚合了整个翻译单元用到的所有数据，它是LLVM术语中的“module”的同义词。它声明了Module::iterator typedef，作为遍历这个模块中的函数的简便方法。你可以用begin()和end()方法获取这些迭代器。
Function	Function类包含有关函数定义和声明的所有对象。对于声明来说（用isDeclaration()检查它是否为声明），它仅包含函数原型。无论定义或者声明，它都包含函数参数的列表，可通过getArgumentList()方法或者arg_begin()和arg_end()这对方法访问它。你可以通过Function::arg_iterator typedef遍历它们。如果Function对象代表函数定义，你可以通过这样的语句遍历它的内容：for (Function::iterator i = function.begin(), e = function.end(); i != e; ++i)，你将遍历它的基本块。
BasicBlock	BasicBlock类封装了LLVM指令序列，可通过begin()/end()访问它们。你可以利用getTerminator()方法直接访问它的最后一条指令，你还可以用一些辅助函数遍历CFG，例如通过getSinglePredecessor()访问前驱基本块，当一个基本块有单一前驱时。然而，如果它有多个前驱基本块，就需要自己遍历前驱列表，这也不难，你只要逐个遍历基本块，查看它们的终结指令的目标基本块。
Instruction	Instruction类表示LLVM IR的运算原子，一个单一的指令。利用一些方法可获得高层级的断言，例如isAssociative()，isCommutative()，isIdempotent()，和isTerminator()，但是它的精确的功能可通过getOpcode()获知，它返回llvm::Instruction枚举的一个成员，代表了LLVM IR opcode。可通过op_begin()和op_end()这对方法访问它的操作数，它从User超类继承得到。

LLVM IR 内存模型最重要概念： Value, Use, User

让我们能够轻松操作 use-def 和 def-use 链



https://buaa-se-compiling.github.io/miniSysY-tutorial/pre/design_hints.html