中国科学技术大学
University of Science and Technology of China

中间语言与中间代码生成 I

《编译原理和技术》

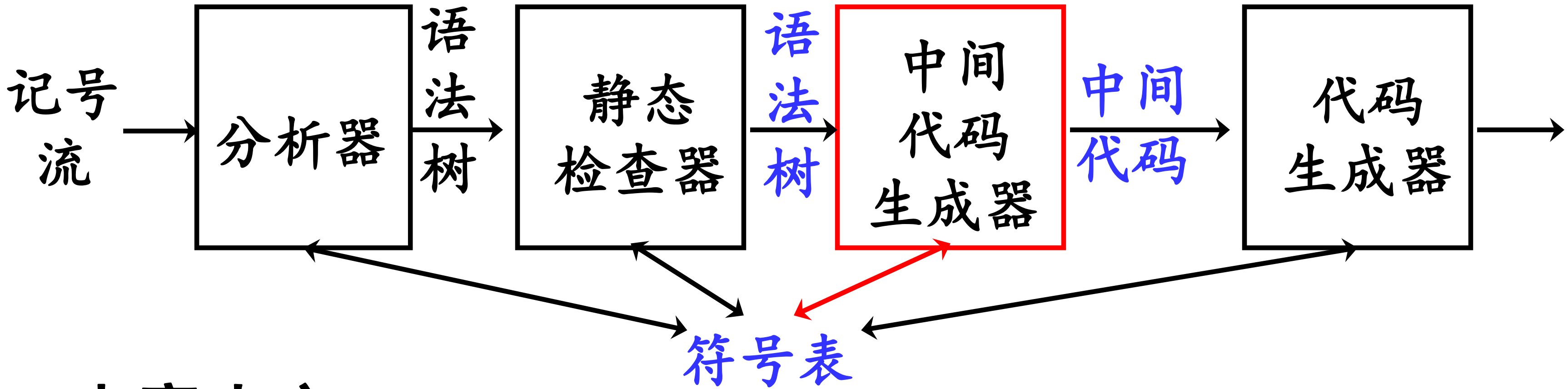
张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容

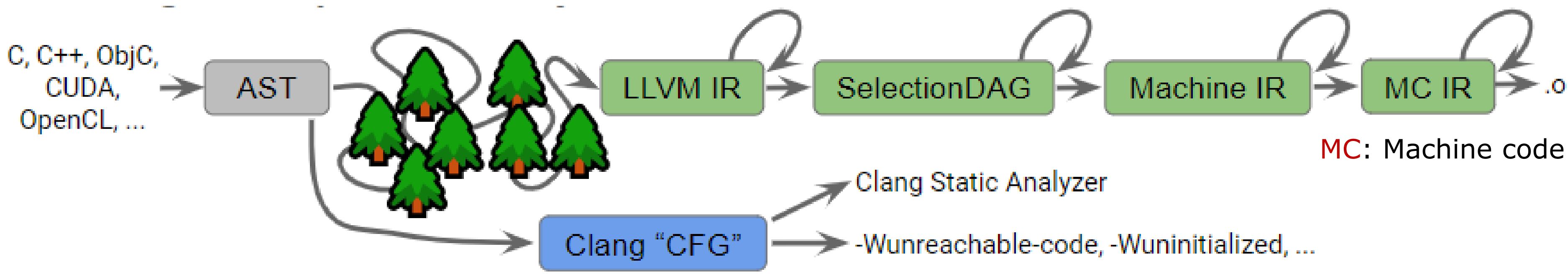


本章内容

- 中间语言：常用的中间表示(Intermediate Representation)
  - 后缀表示、图表示、三地址代码、LLVM IR
- 基本块和控制流图
- 中间代码的生成
  - 声明语句(=>更新符号表)
  - 表达式、赋值语句（=>产生临时变量、查符号表）
  - 布尔表达式、控制流语句（=>标号/回填技术、短路计算）

中间语言的重要性

- 中间表示：Intermediate Representation
- 不同级别的IR：支持不同层次的**程序分析**和**优化**
- Clang编译器

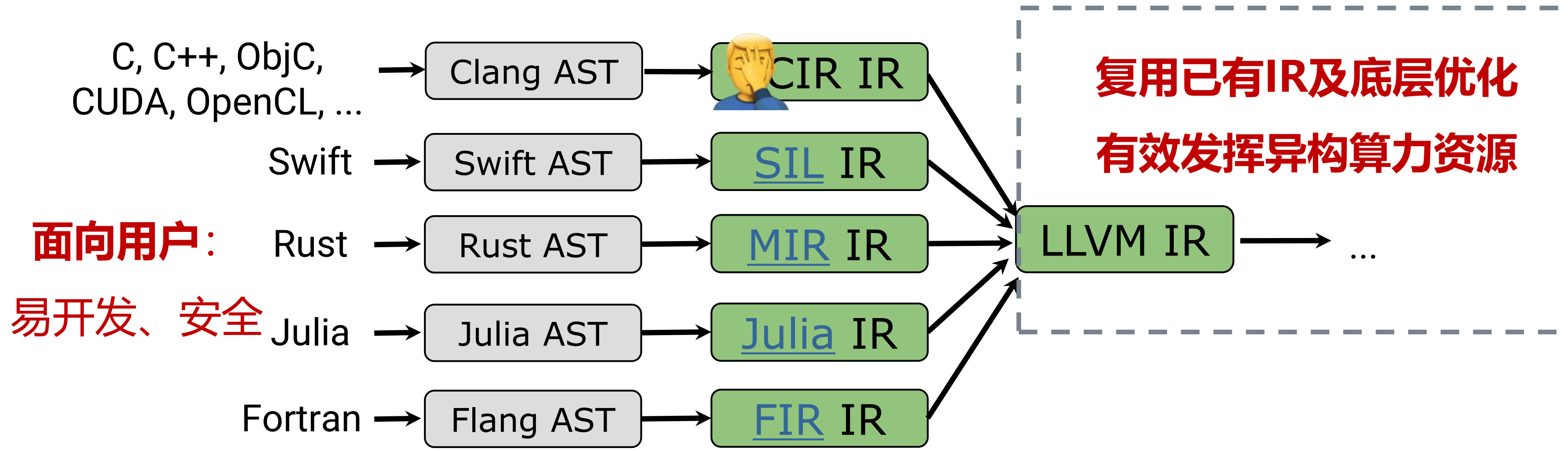


- **C++与LLVM IR之间存在鸿沟**
  - 需要更高级的IR

https://llvm.org/

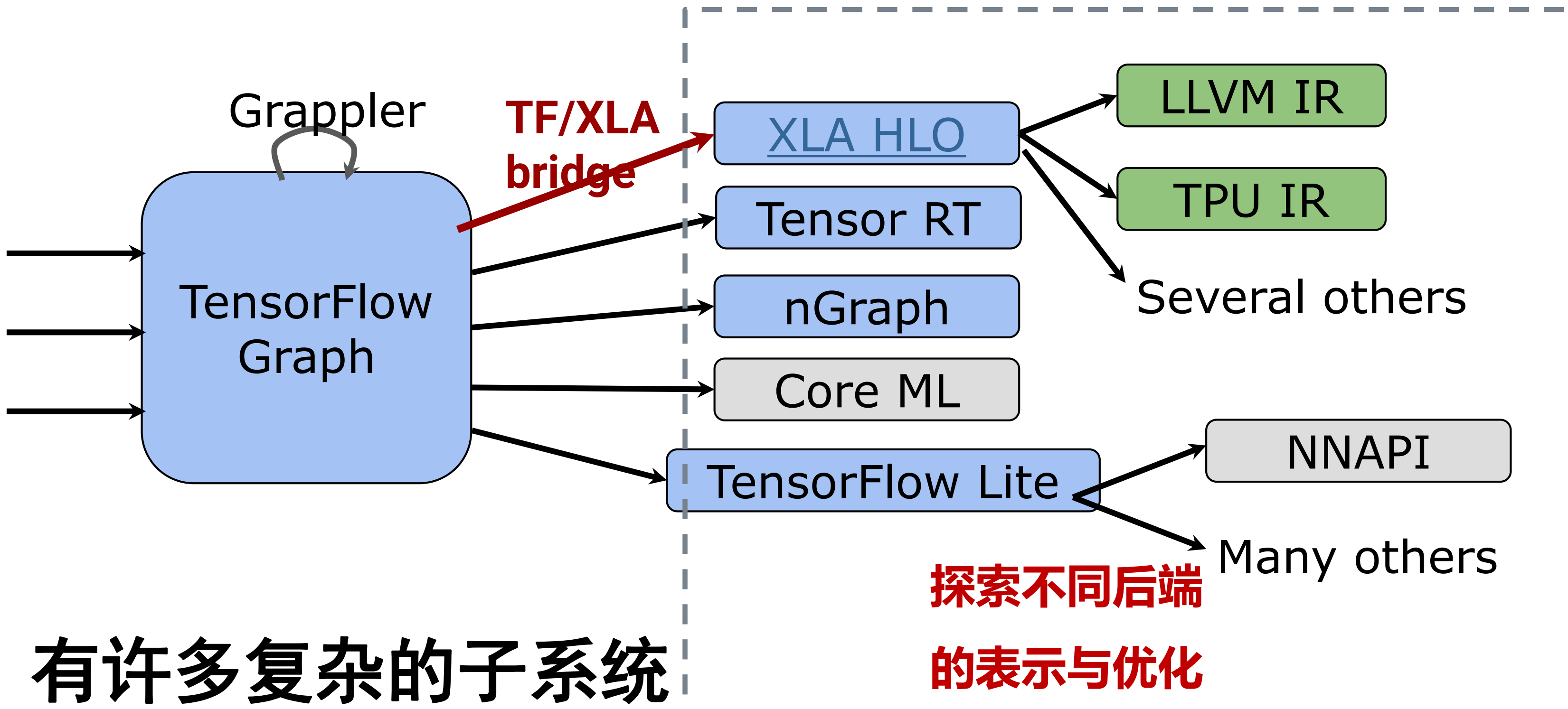
高级中间表示

现代编程语言不断推出Higer IRs



- 语言相关的优化
- 数据流驱动的类型检查：定义的初始化、borrow检查
- 从高级编程抽象逐步降低lowering

TensorFlow编译器



- 有许多复杂的子系统
  - 每个有自己的抽象和表示
  - 需要在TensorFlow与不同的后端建立桥梁

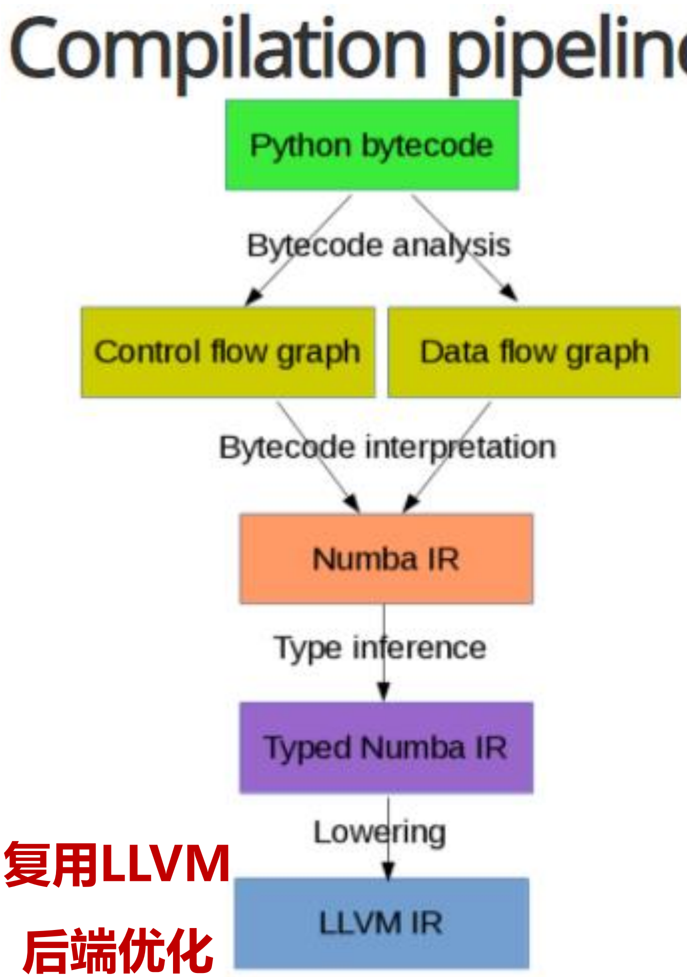
提出MLIR--可扩展的多级中间表示及可复用的编译基础设施

[CGO 2021] MLIR: Scaling Compiler Infrastructure for Domain

各种在研的IRs

中国科学技术大学
University of Science and Technology of China

Numba： Accelerate Python Functions

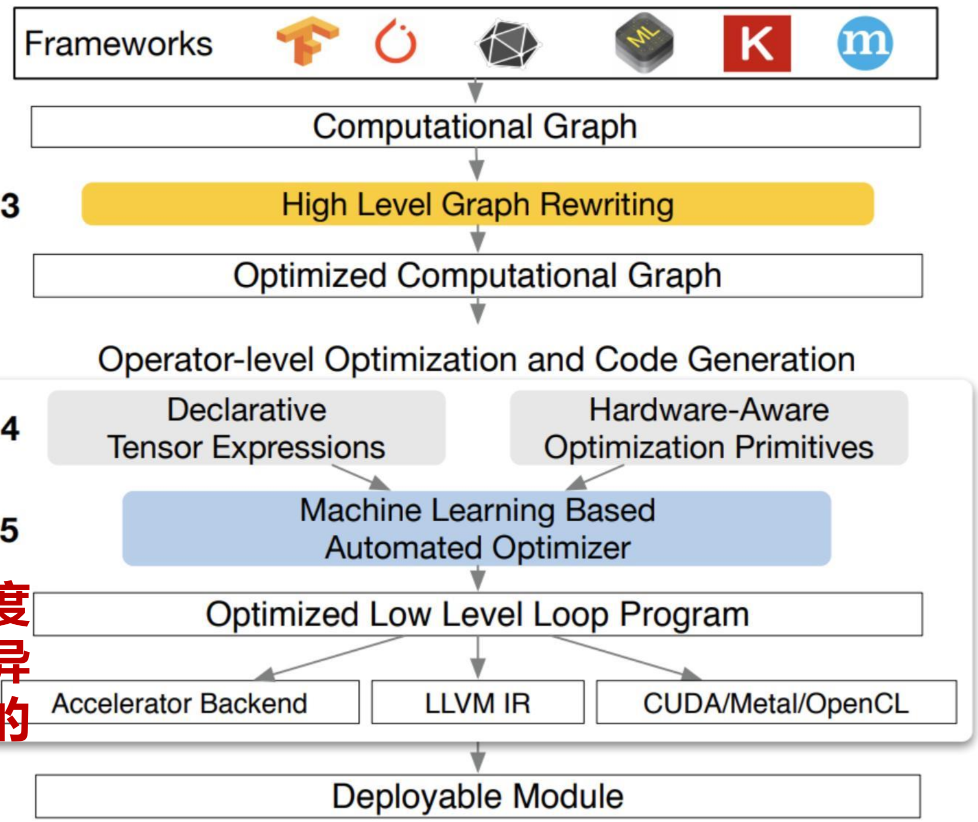


复用LLVM
后端优化

https://numba.pydata.org/

TVM： Tensor Virtual Machine

与不同的深度
学习框架接口

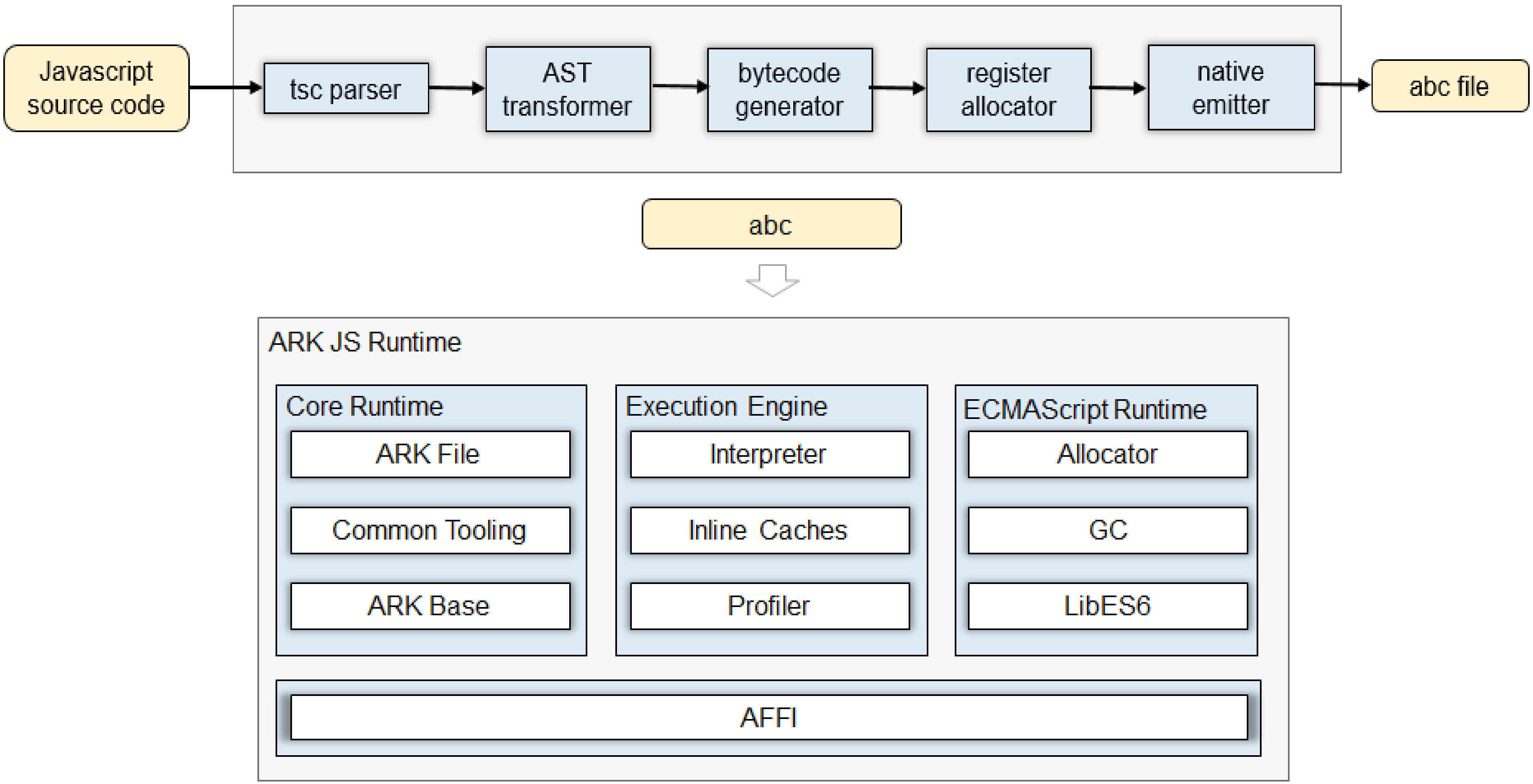


面向深度
学习和异
构设备的
优化

https://tvm.apache.org/

方舟JS Runtime

- https://gitee.com/openharmony/ark_js_runtime



1. 中间语言

- 后缀形式、图形表示
- 三地址代码
- 静态单赋值

后缀表示

- 后缀表示(逆波兰式)：运算符在其运算对象之后

(8 – 5) + 2 的后缀表示是 8 5 –2 +　不需要括号　前提：算符无二义

- 后缀表示的最大优点：便于计算机处理表达式，如求值、代码生成等

计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

后缀表示

- 后缀表示不需要括号(前提：算符无二义)

  (8 – 5) + 2 的后缀表示是 8 5 –2 +

- 后缀表示的最大优点是便于计算机处理表达式

- 后缀表示的表达能力
  - 可以拓广到表示赋值语句和控制语句
  - 但很难用栈来描述控制语句的计算

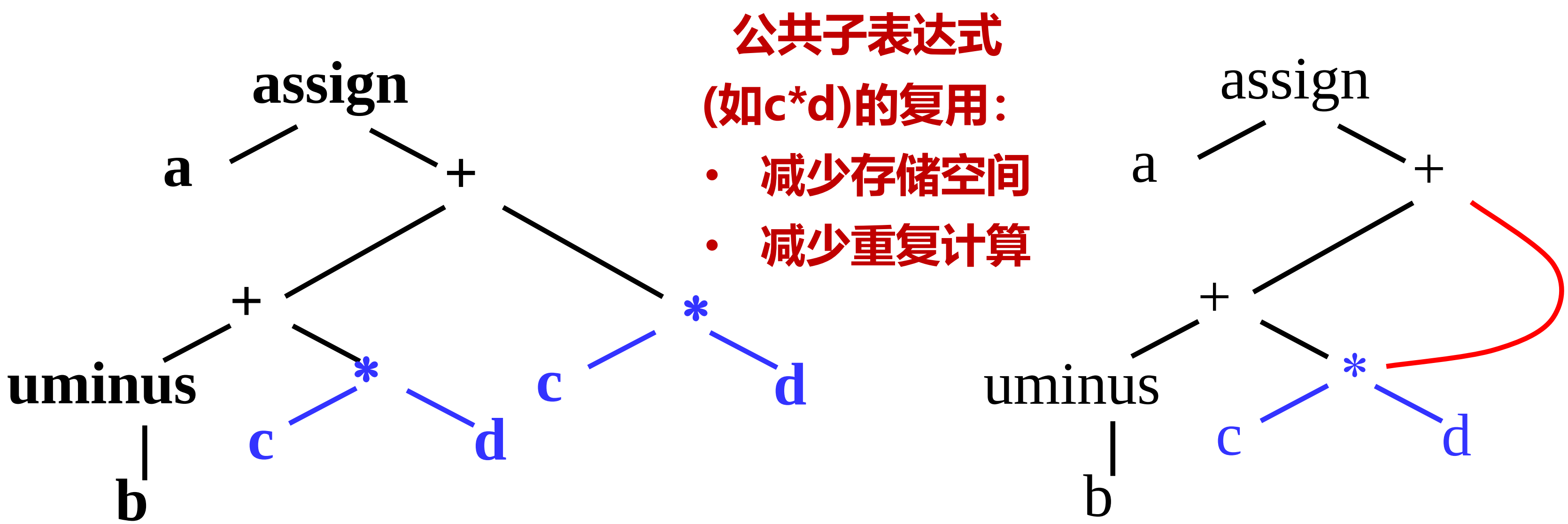
  适合底层实现

- 前缀表示（波兰式）
  - 一种逻辑、算术和代数的表示方法，如 op(a, b, c)
  - 用于简化命题逻辑

  适合上层表达

图形表示

- 语法树是一种图形化的中间表示
- 有向无环图也是一种中间表示



(a) 语法树　　　　(b) DAG

a = (–b + c*d) + c*d的图形表示

图形表示

构造赋值语句语法树的语法制导定义

修改构造结点的函数*mkNode*可生成有向无环图：

—判断是否已有**计算等价**的表达式树，如用 ValueNumbering (VN)

产 生 式	语 义 规 则
$S \rightarrow \mathrm{id} = E$	$S.nptr = mkNode(\text{'assign'}, mkLeaf(\mathrm{id}, \mathrm{id}.entry), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr = mkNode(\text{'+'}, E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr = mkNode(\text{'*'}, E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr = mkUNode(\text{'uminus'}, E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr = E_1.nptr$
$F \rightarrow \mathrm{id}$	$E.nptr = mkLeaf(\mathrm{id}, \mathrm{id}.entry)$

三地址代码

三地址代码(three-address code)

一般形式：$x = y\ op\ z$

最多1个算符，最多3个计算分量(运算对象的地址)

➔三地址

例 表达式 $x + y * z$ 翻译成的三地址语句序列是

$t_1 = y * z$

$t_2 = x + t_1$

三地址代码

- 三地址代码是语法树或DAG的一种**线性**表示

存储布局是线性的；
按字节寻址

例 a = (–b + c*d) + c*d

语法树的代码

$t_1 = -b$

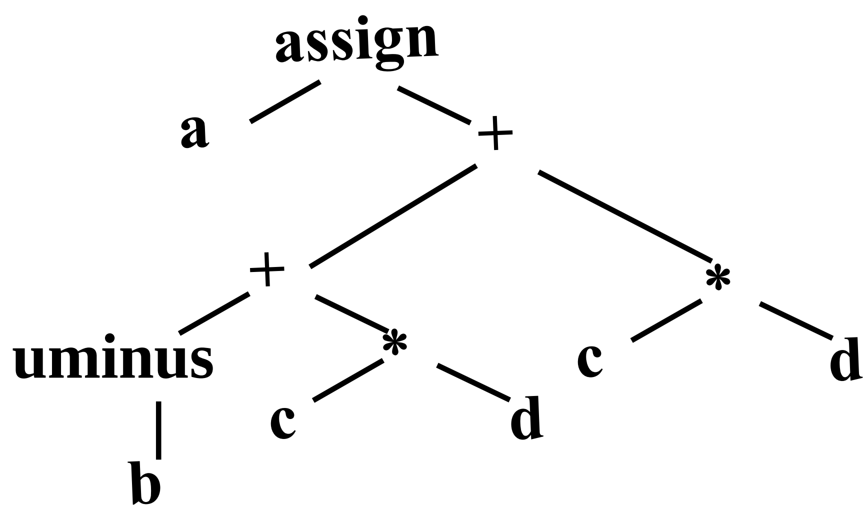
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$



对语法树进行后序遍历，输出三地址代码

——体现后缀式的应用价值

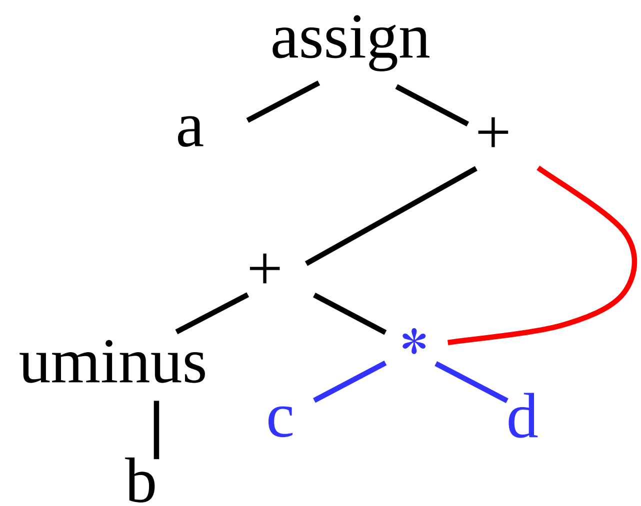
编译器实现中会建立后序线索化树，方便代码生成、求值等

三地址代码

- 三地址代码是语法树或DAG的一种线性表示

例 a = (–b + c*d) + c*d

语法树的代码	DAG的代码
$t_1 = -b$	$t_1 = -b$
$t_2 = c * d$	$t_2 = c * d$
$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$
$t_4 = c * d$	$t_4 = t_3 + t_2$
$t_5 = t_3 + t_4$	$a = t_5$
$a = t_5$	



(b) DAG

按DAG结点的拓扑序，输出三地址代码

三地址代码

常用的三地址语句

- 赋值语句 $x = y\ op\ z$， $x = op\ y$
- 复写语句 $x = y$
- 无条件转移 goto L
- 条件转移 if x *relop* y goto L
- 过程调用

  param x 参数设置

  call p , n 调用含n个参数的子过程p

- 过程返回 return y
- 索引赋值 $x = y[i]$和 $x[i] = y$
- 地址和指针赋值 $x = \&y$， $x = *y$ 和 $*x = y$

要注意遵循的约定(convention)

如多个参数的param出现的先后次序

静态单赋值

- 静态单赋值形式(static single-assignment form)
  - 一种便于某些代码优化的中间表示
  - 和三地址代码的主要区别

所有赋值指令都是对不同名字的变量的赋值

三地址代码	静态单赋值形式
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

对p的定值

对p的引用

SSA的优势： 明确知道所引用的变量在哪定值

静态单赋值

- 一种便于某些代码优化的中间表示
- 和三地址代码的主要区别

  所有赋值指令都是对不同名字的变量的赋值

  同一个变量在不同控制流路径上都被定值的解决办法：

  if (flag) x = –1; else x = 1;

  y = x * a;

改成

  if (flag) x_1 **= –1; else** x_2 **= 1;**

  $x_3 = \phi(x_1, x_2)$; // 由flag的值决定用x_1还是x_2

  $y = x_3 * a$;

Phi算子：汇合对多个可能定值的引用

2. 基本块和控制流图

- 基本块
- 流图

基本块

程序举例

源程序

```
prod = 0;
i = 1;
do {
    prod = prod + a[i] * b[i];
    i = i +1;
} while (i <= 20);
```

第i个元素的类型为int

三地址码

(1)prod = 0

(2) i = 1

(3) t_1 = 4 * i

(4) t_2= a[t_1]

(5) t_3 = 4 * i

(6) t_4 = b[t_3]

(7) t_5 = t_2 * t_4

(8) t_6 = prod + t_5

(9) prod = t_6

(10) t_7 = i +1

(11) i = t_7

(12) if i <= 20 goto (3)

元素的地址要转换成按字节寻址

基本块和流图

中国科学技术大学 University of Science and Technology of China

- 基本块(basic block)
  - 是连续的语句序列
  - 控制流从它的开始进入-单入口，并从它的末尾离开-单出口，没有停止或分支的可能性(末尾除外)
- 流图(flow graph)
  - 用有向边表示基本块之间的控制流信息
  - 基本块作为流图的结点

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

基本块的划分

基本块的划分方法

- 首先确定所有入口语句
  - 序列的第一个语句
  - 能由(无)条件转移语句转到的语句
  - 紧跟在(无)条件转移语句后面的语句
- 每个入口语句到下一个入口语句之前（或到程序结束）的语句序列构成一个基本块

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

流图

(1) $prod = 0$
(2) $i = 1$
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if $i <= 20$ goto (3)

B_1:

(1) $prod = 0$
(2) $i = 1$

B_2:

(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if $i <= 20$ goto (3)

流图

(1)prod = 0
(2) i = 1
(3) t_1 = 4 ∗ i
(4) t_2= a[t_1]
(5) t_3 = 4 ∗ i
(6) t_4 = b[t_3]
(7) t_5 = t_2 ∗ t_4
(8) t_6 = prod + t_5
(9) prod = t_6
(10) t_7 = i +1
(11) i = t_7
(12) if i <= 20 goto (3)

B_1

(1)prod = 0
(2) i = 1

B_2

(3) t_1 = 4 ∗ i
(4) t_2= a[t_1]
(5) t_3 = 4 ∗ i
(6) t_4 = b[t_3]
(7) t_5 = t_2 ∗ t_4
(8) t_6 = prod + t_5
(9) prod = t_6
(10) t_7 = i +1
(11) i = t_7
(12) if i <= 20 goto (3)

流图(变换成 SSA 格式)

(1)prod = 0

(2) i_1 = 1

(3) $i_3 = \phi(i_1, i_2)$

(4) $t_1 = 4 * i_3$

(5) $t_2 = a[t_1]$

(6) $t_3 = 4 * i_3$

(7) $t_4 = b[t_3]$

(8) $t_5 = t_2 * t_4$

(9) $t_6 = prod + t_5$

(10) prod = t_6

(11) $t_7 = i_3 + 1$

(12) $i_2 = t_7$

(13) if i_2 <= 20 goto (3)

B_1

(1)prod = 0

(2) i_1 = 1

B_2

(3) $i_3 = \phi(i_1, i_2)$

(4) $t_1 = 4 * i_3$

(5) $t_2 = a[t_1]$

(6) $t_3 = 4 * i_3$

(7) $t_4 = b[t_3]$

(8) $t_5 = t_2 * t_4$

(9) $t_6 = prod + t_5$

(10) prod = t_6

(11) $t_7 = i_3 + 1$

(12) $i_2 = t_7$

(13) if i_2 <= 20 goto (3)

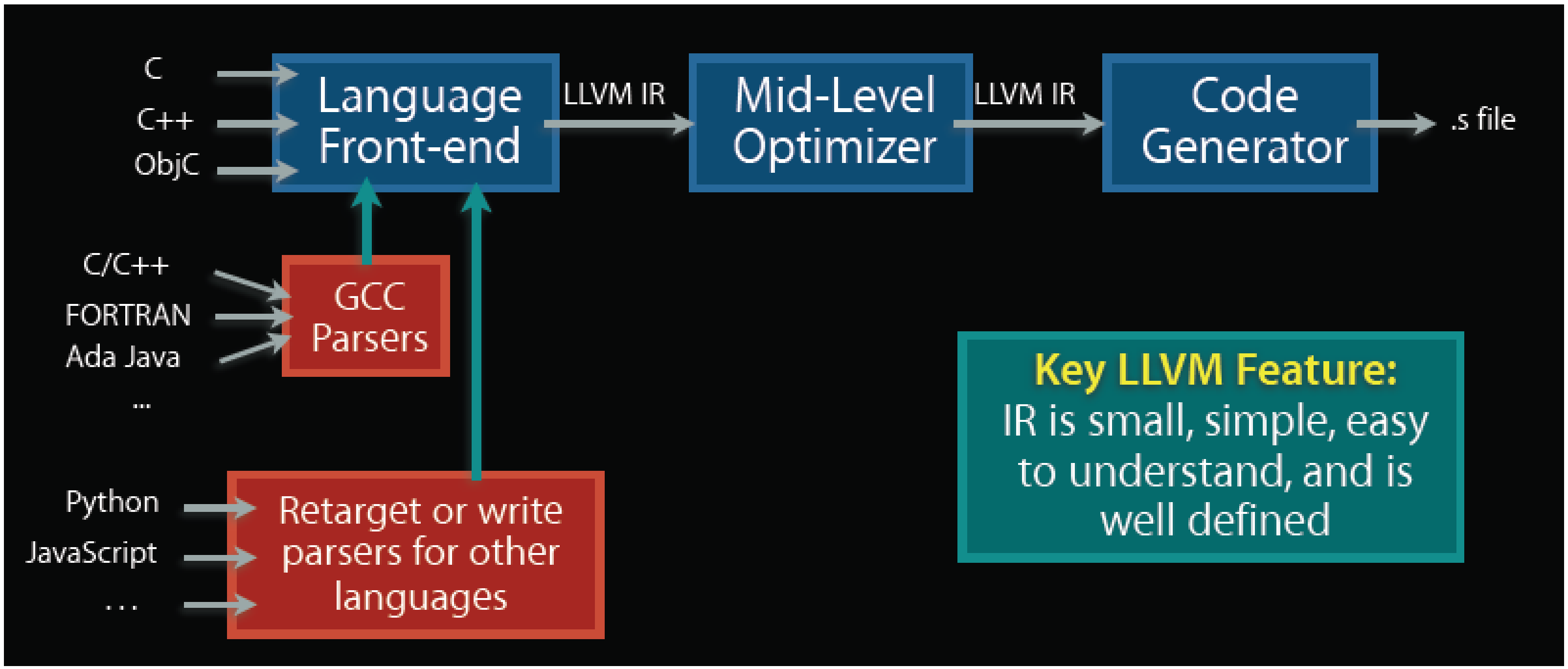
利用流图，可快速找到B_2的前驱基本块，按控制流逆向找到最近对 i 的定值

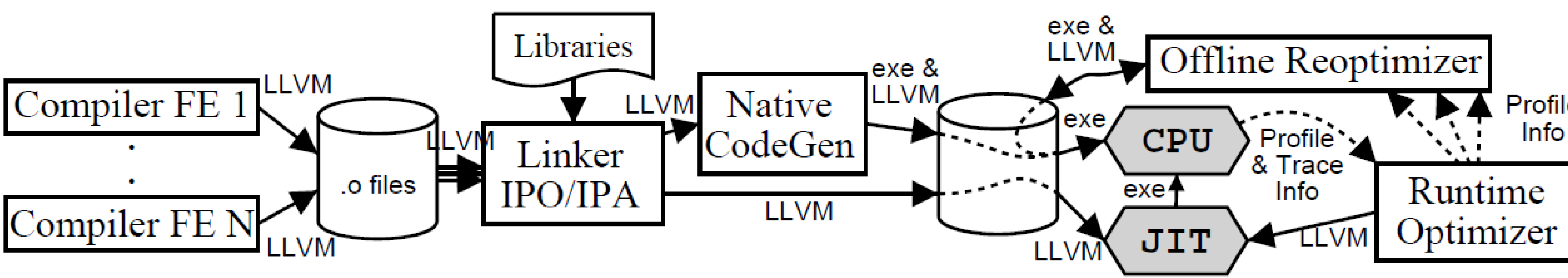
中国科学技术大学
University of Science and Technology of China

3. LLVM 编译器框架和基础设施

- 总体结构
- LLVM IR
- LLVM Pass Manager
- LLVM Tools

LLVM编译系统


C
C++
ObjC
Language Front-end
LLVM IR
Mid-Level Optimizer
LLVM IR
Code Generator
.s file
C/C++
FORTRAN
Ada Java
...
GCC Parsers
Python
JavaScript
. . .
Retarget or write parsers for other languages
Key LLVM Feature:
IR is small, simple, easy to understand, and is well defined


Compiler FE 1
Compiler FE N
LLVM
LLVM
.o files
Libraries
LLVM
Linker IPO/IPA
LLVM
Native CodeGen
LLVM
exe & LLVM
exe & LLVM
Offline Reoptimizer
exe
CPU
Profile & Trace Info
Profile Info
exe
JIT
LLVM
LLVM
Runtime Optimizer

LLVM IR

- 参考资料
  - LLVM IR参考手册 (http://llvm.org/docs/LangRef.html)
  - 教程(http://llvm.org/docs/tutorial/LangImpl03.html)
- 主要特征
  - RISC风格的三地址代码
  - SSA格式、无限的虚拟寄存器
  - 简单、低级的控制流结构
  - load/store指令带类型化指针
- IR的格式：text(.ll)、binary(.bc)、in-memory

LLVM IR速览

C program language	*LLVM IR*
• Scope: *file, function*	*module, function*
• Type: *bool, char, int, struct{int, char}*	*i1, i8, i32, {i32, i8}*
• A statement with multiple expressions	A sequence of instructions each of which is in a form of "x = y *op* z".
• Data-flow: a sequence of reads/writes on variables	1. load the values of memory addresses (variables) to registers; 2. compute the values in registers; 3. store the values of registers to memory addresses * each register must be assigned exactly once (SSA)
• Control-flow in a function: if, for, while, do while, switch-case,…	A set of basic blocks each of which ends with a conditional jump (or return)

LLVM类型系统

- 类型系统的组成
  - **Primitives: integer, floating point, label, void**
    - no “signed” integer types
    - arbitrary bitwidth integers (i32, i64, i1)
  - **Derived: pointer, array, structure, function, vector,…**

No high-level types: type-system is language neutral!

- **Type system allows arbitrary casts:**
  - **Allows expressing weakly-typed languages, like C**
  - ***Front-ends can implement safe languages***
  - ***Also easy to define a type-safe subset of LLVM***

示例：C 编译到LLVM

```
int callee(const int *X) {
  return *X+1;   // load
}
int caller() {
  int T;         // on stack
  T = 4;         // store
  return callee(&T);
}
```

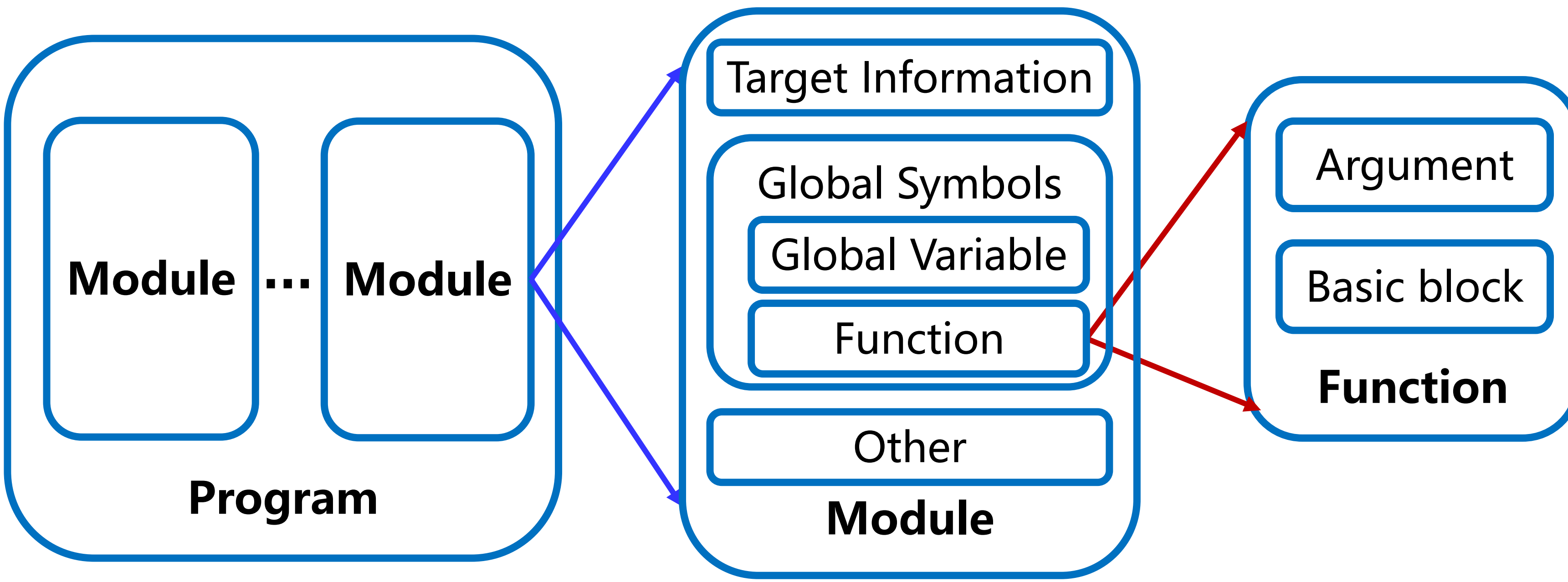
```
define internal i32 @callee(i32* %X) {
entry:
  %tmp2 = load i32* %X
  %tmp3 = add i32 %tmp2, 1
  ret i32 %tmp3
}

define internal i32 @caller() {
entry:
  %T = alloca i32
  store i32 4, i32* %T
  %tmp1 = call i32 @callee( i32* %T )
  ret i32 %tmp1
}
```

Stack allocation is explicit in LLVM

All loads/stores are explicit in the LLVM representation

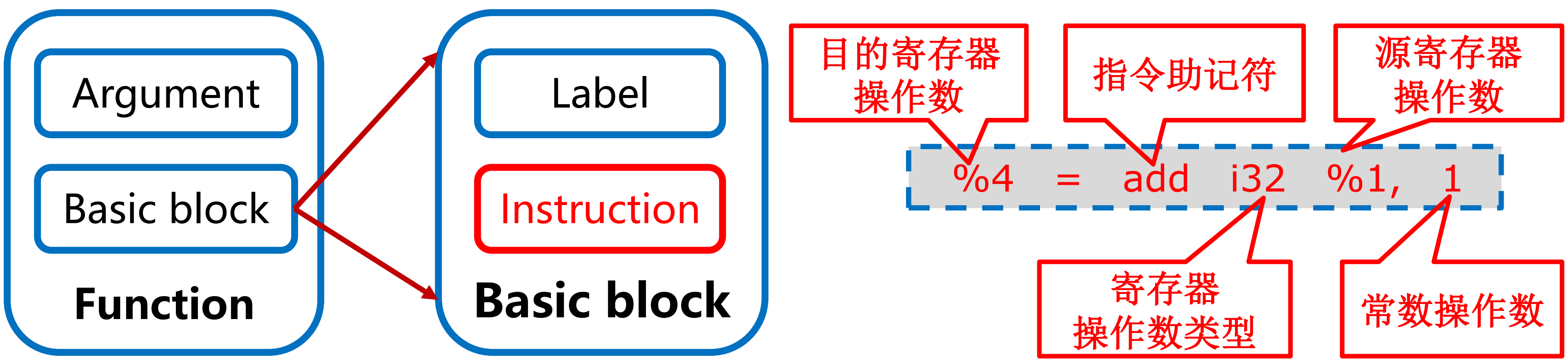
LLVM IR的程序结构

- 模块Module：包含函数和全局变量
  - 是编译/分析/优化的基本单位，对应一个程序文件
- 函数Function：包含基本块/参数
- 基本块BasicBlock: 指令序列



LLVM IR的程序结构

- 模块Module：包含函数和全局变量
  - 是编译/分析/优化的基本单位，对应一个程序文件
- 函数Function：包含基本块/参数
- 基本块BasicBlock: 指令序列
- 指令Instruction: opcode + vector of operands
  - 所有操作数operands都有类型、指令结果是类型化的



LLVM IR

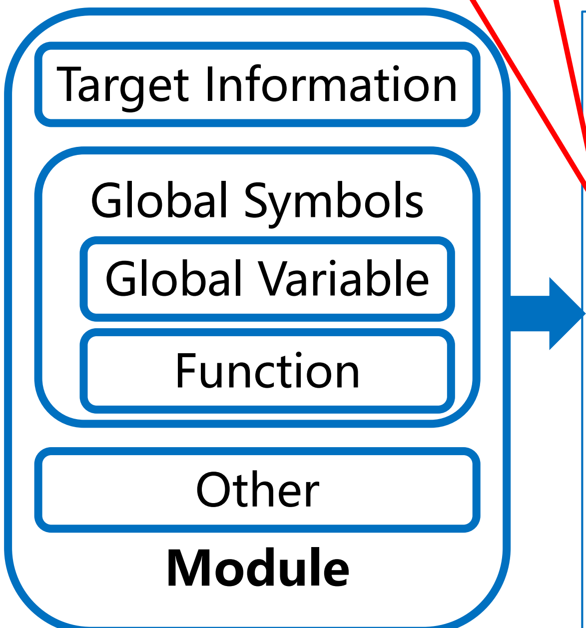
Module结构

```c
#include <stdio.h>
int main(){
    printf("hello, world\n");
    return 0;
}
```

helloworld.c

$1: 程序文件名
$2: 附加的参数，如 **–m32** 表示生成32位机器代码

```
clang -emit-llvm -S $1.c -o $1$2.ll $2
```



全局标识符

局部标识符

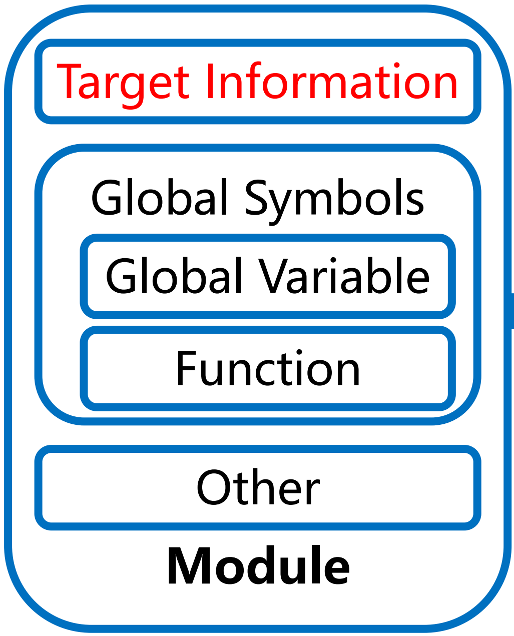
```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [15 x i8] c"hello,\C2\A0world\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...) #1
```

LLVM IR

Module结构

```c
#include <stdio.h>
int main(){
    printf("hello, world\n");
    return 0;
}
```

helloworld.c



目标内存排布信息

目标宿主信息

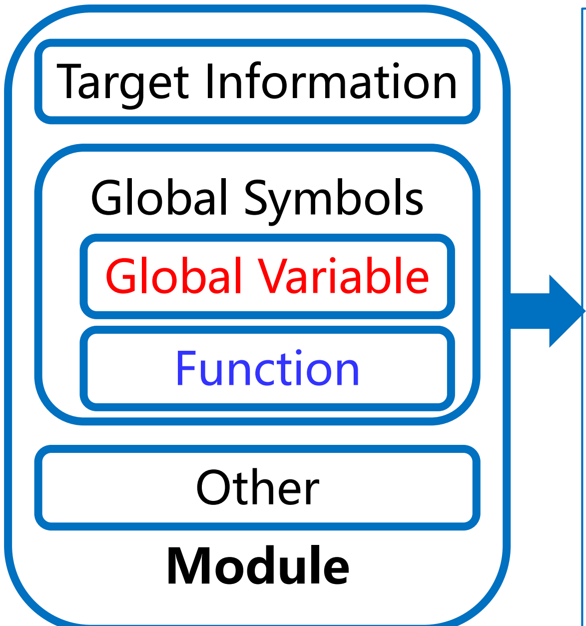
```llvm
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [15 x i8] c"hello,\C2\A0world\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...) #1
```

LLVM IR

□ Module结构

```c
#include <stdio.h>
int main(){
    printf("hello, world\n");
    return 0;
}
```

helloworld.c




```llvm
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
@.str = private unnamed_addr constant [15 x i8] c"hello,\C2\A0world\0A\00", align 1
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}
declare dso_local i32 @printf(i8*, ...) #1
```

全局变量定义

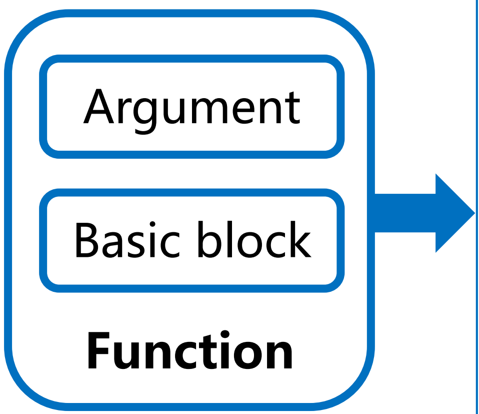
函数定义

函数声明

□ Function结构

dso: dynamic shared object
dso_local:解析为模块内的符号
dso_preemptable:
在运行时可能被外部符号取代

```
double foo();
double bar(float a) {
        return foo(a, 4.0) + bar(31337);
}
```

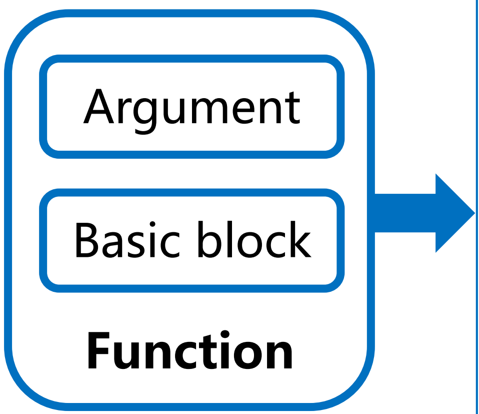



```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local double @bar(float %0) #0 {
  %2 = alloca float, align 4
  store float %0, float* %2, align 4
  %3 = load float, float* %2, align 4
  %4 = fpext float %3 to double
  %5 = call double (double, double, ...) bitcast
(double (...)* @foo to double (double,
double, ...)*)(double %4, double 4.000000e+00)
  %6 = call double @bar(float 3.133700e+04)
  %7 = fadd double %5, %6
  ret double %7
}

declare dso_local double @foo(...) #1
```

LLVM IR

Function结构

```
double foo();
double bar(float a) {
        return foo(a, 4.0) + bar(31337);
}
```



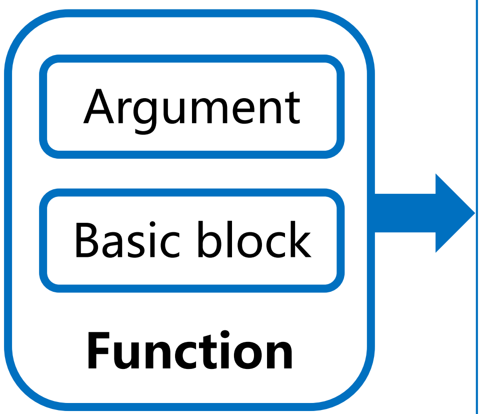

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local double @bar(float %0) #0 {
  %2 = alloca float, align 4
  store float %0, float* %2, align 4
  %3 = load float, float* %2, align 4
  %4 = fpext float %3 to double
  %5 = call double (double, double, ...) bitcast
(double (...)* @foo to double (double,
double, ...)*)(double %4, double 4.000000e+00)
  %6 = call double @bar(float 3.133700e+04)
  %7 = fadd double %5, %6
  ret double %7
}

declare dso_local double @foo(...) #1
```

参数%0的值存储到新分配的虚拟寄存器%2

不仅指明了类型，还指明了按多少字节齐

□ Function结构

```
double foo();
double bar(float a) {
        return foo(a, 4.0) + bar(31337);
}
```




```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local double @bar(float %0) #0 {
  %2 = alloca float, align 4
  store float %0, float* %2, align 4
  %3 = load float, float* %2, align 4
  %4 = fpext float %3 to double
  %5 = call double (double, double, ...) bitcast
(double (...)* @foo to double (double,
double, ...)*)(double %4, double 4.000000e+00)
  %6 = call double @bar(float 3.133700e+04)
  %7 = fadd double %5, %6
  ret double %7
}

declare dso_local double @foo(...) #1
```

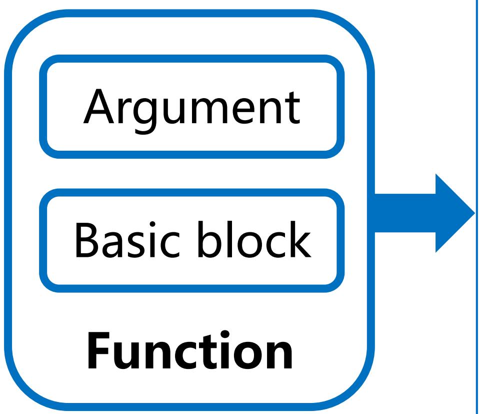
加载参数值，将float类型的数扩展为double型

自动类型提升：**float→double**

LLVM IR

Function结构

```
double foo();
double bar(float a) {
        return foo(a, 4.0) + bar(31337);
}
```



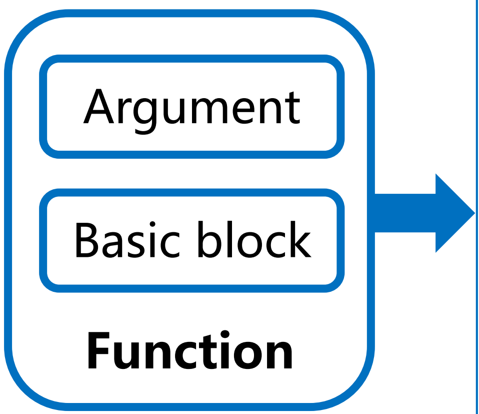

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local double @bar(float %0) #0 {
  %2 = alloca float, align 4
  store float %0, float* %2, align 4
  %3 = load float, float* %2, align 4
  %4 = fpext float %3 to double
  %5 = call double (double, double, ...) bitcast
(double (...)* @foo to double (double,
double, ...)*)(double %4, double 4.000000e+00)
  %6 = call double @bar(float 3.133700e+04)
  %7 = fadd double %5, %6
  ret double %7
}

declare dso_local double @foo(...) #1
```

调用foo函数，将foo强制为至少有2个double型参数的函数类型

bitcast强制类型转换

Function结构

```
double foo();
double bar(float a) {
        return foo(a, 4.0) + bar(31337);
}
```

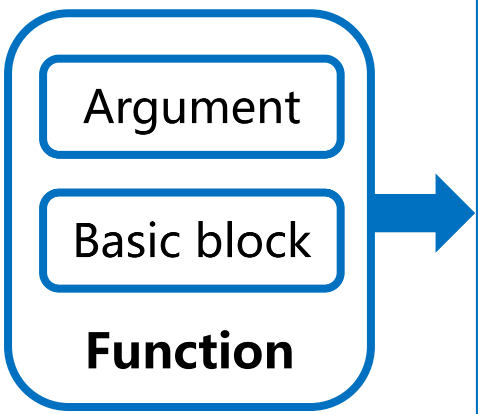



```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local double @bar(float %0) #0 {
  %2 = alloca float, align 4
  store float %0, float* %2, align 4
  %3 = load float, float* %2, align 4
  %4 = fpext float %3 to double
  %5 = call double (double, double, ...) bitcast
(double (...)* @foo to double (double,
double, ...)*)(double %4, double 4.000000e+00)
  %6 = call double @bar(float 3.133700e+04)
  %7 = fadd double %5, %6
  ret double %7
}

declare dso_local double @foo(...) #1
```

调用bar
31337看成float

□ Function结构

```
double foo();
double bar(float a) {
        return foo(a, 4.0) + bar(31337);
}
```




```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local double @bar(float %0) #0 {
  %2 = alloca float, align 4
  store float %0, float* %2, align 4
  %3 = load float, float* %2, align 4
  %4 = fpext float %3 to double
  %5 = call double (double, double, ...) bitcast
(double (...)* @foo to double (double,
double, ...)*)(double %4, double 4.000000e+00)
  %6 = call double @bar(float 3.133700e+04)
  %7 = fadd double %5, %6
  ret double %7
}

declare dso_local double @foo(...) #1
```

执行double类型的fadd运算，将计算结果返回

LLVM IR

基本块和流图

```
define dso_local void @f(i32* %0) #0 {
  %2 = alloca i32*, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32* %0, i32** %2, align 8
  store i32 0, i32* %3, align 4
  br label %5

5:                                    ; preds = %14, %1
  %6 = load i32, i32* %3, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %17

8:                                    ; preds = %5
  %9 = load i32*, i32** %2, align 8
  %10 = load i32, i32* %3, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %9, i64 %11
  %13 = call i32 (i32*, i32*, ...) bitcast (i32 (...)*
@Sum to i32 (i32*, i32*, ...)*)(i32* %12, i32* %4)
  br label %14
```

```
define N 10
void f(int A[])
{
        int i, P;
        for (i = 0; i < N; ++i)
                Sum(&A[i], &P);
}
```

分配局部变量，%3和%4分别对应 i 和 P

LLVM IR

基本块和流图

```
define dso_local void @f(i32* %0) #0 {
  %2 = alloca i32*, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32* %0, i32** %2, align 8
  store i32 0, i32* %3, align 4
  br label %5

5:                                    ; preds = %14, %1
  %6 = load i32, i32* %3, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %17

8:                                    ; preds = %5
  %9 = load i32*, i32** %2, align 8
  %10 = load i32, i32* %3, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %9, i64 %11
  %13 = call i32 (i32*, i32*, ...) bitcast (i32 (...)*
@Sum to i32 (i32*, i32*, ...)*)(i32* %12, i32* %4)
  br label %14
```

```
define N 10
void f(int A[])
{
        int i, P;
        for (i = 0; i < N; ++i)
                Sum(&A[i], &P);
}
```

无条件跳转到标号为5的语句

br label 标号

LLVM IR

基本块和流图

```
……
5:                              ; preds = %14, %1
  %6 = load i32, i32* %3, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %17

8:                              ; preds = %5
  %9 = load i32*, i32** %2, align 8
  %10 = load i32, i32* %3, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %9, i64 %11
  %13 = call i32 (i32*, i32*, ...) bitcast (i32 (...)*
@Sum to i32 (i32*, i32*, ...)*)(i32* %12, i32* %4)
  br label %14

14:                             ; preds = %8
  %15 = load i32, i32* %3, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %3, align 4
  br label %5
```

```
define N 10
void f(int A[])
{
        int i, P;
        for (i = 0; i < N; ++i)
                Sum(&A[i], &P);
}
```

基本块5的前驱基本块分别是标号为14和1两个基本块

preds 指明前驱的标号

LLVM IR

□ 基本块和流图

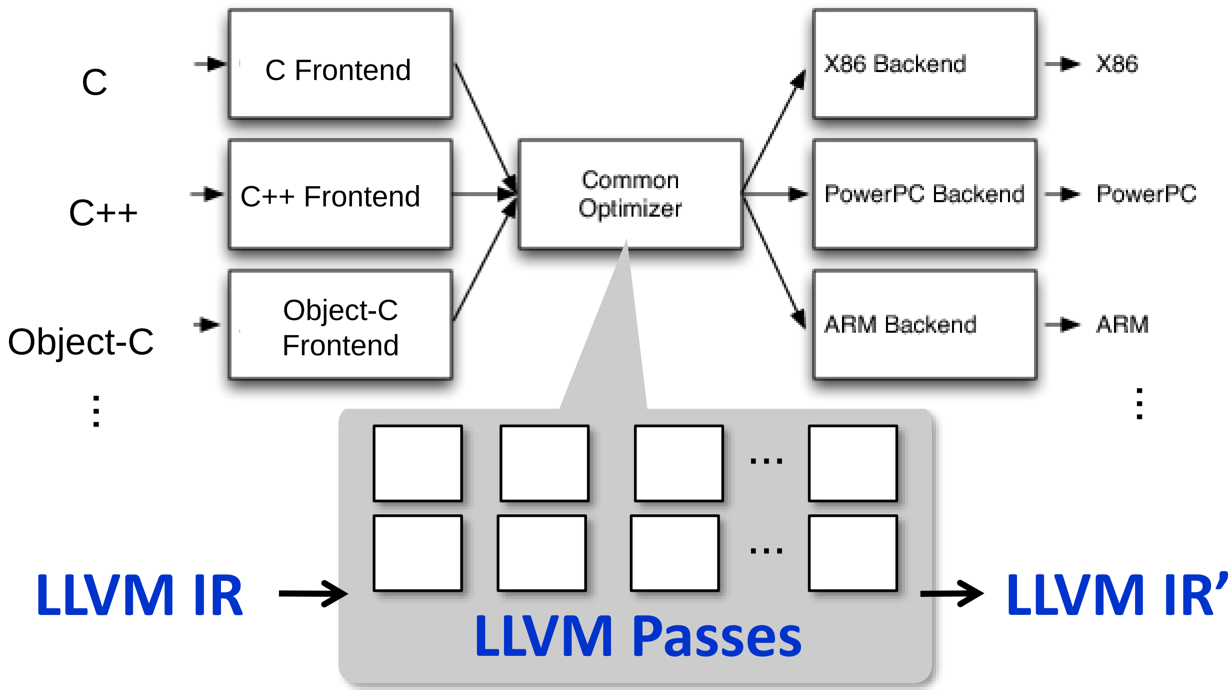
```
……
5:                                   ; preds = %14, %1
  %6 = load i32, i32* %3, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %17

8:                                   ; preds = %5
  %9 = load i32*, i32** %2, align 8
  %10 = load i32, i32* %3, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %9, i64 %11
  %13 = call i32 (i32*, i32*, ...) bitcast (i32 (...)*
@Sum to i32 (i32*, i32*, ...)*)(i32* %12, i32* %4)
  br label %14

14:                                  ; preds = %8
  %15 = load i32, i32* %3, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %3, align 4
  br label %5
```

```
define N 10
void f(int A[])
{
        int i, P;
        for (i = 0; i < N; ++i)
                Sum(&A[i], &P);
}
```

通过getelementptr(gep)获取元素A[i]的地址
inbounds 表示i超出10 (%11)，则gep返回 poison value

LLVM Passes

- LLVM提供108^{+} Passes http://llvm.org/docs/Passes.html
  - 分析器：别名分析、调用图构造、依赖分析等
  - 变换器：死代码消除、函数内联、常量传播、循环展开等
  - 实用组件：CFG viewer、基本块提取器等



LLVM Pass Manager

- 编译器组织成一系列的passes
  - 每个pass是一个分析或变换
- Pass的类型
  - **ModulePass**: general interprocedural pass
  - **CallGraphSCCPass**: bottom-up on the call graph
  - **FunctionPass**: process a function at a time
  - **LoopPass**: process a natural loop at a time
  - **BasicBlockPass**: process a basic block at a time
- 施加的约束 (e.g. FunctionPass):
  - FunctionPass 只能查看当前函数
  - 不能维护跨函数之间的状态

SCC 强连通分量

LLVM工具

- 基础工具
  - llvm-as: Convert from .ll (text) to .bc (binary)
  - llvm-dis: Convert from .bc (binary) to .ll (text)
  - llvm-link: Link multiple .bc files together
  - llvm-prof: Print profile output to human readers
  - llvmc: Configurable compiler driver
- 集成工具
  - bugpoint: automatic compiler debugger
  - llvm-gcc/llvm-g++: C/C++ compilers