Lecture 3.3

线性IR和解释执行

徐 辉
xuh@fudan.edu.cn

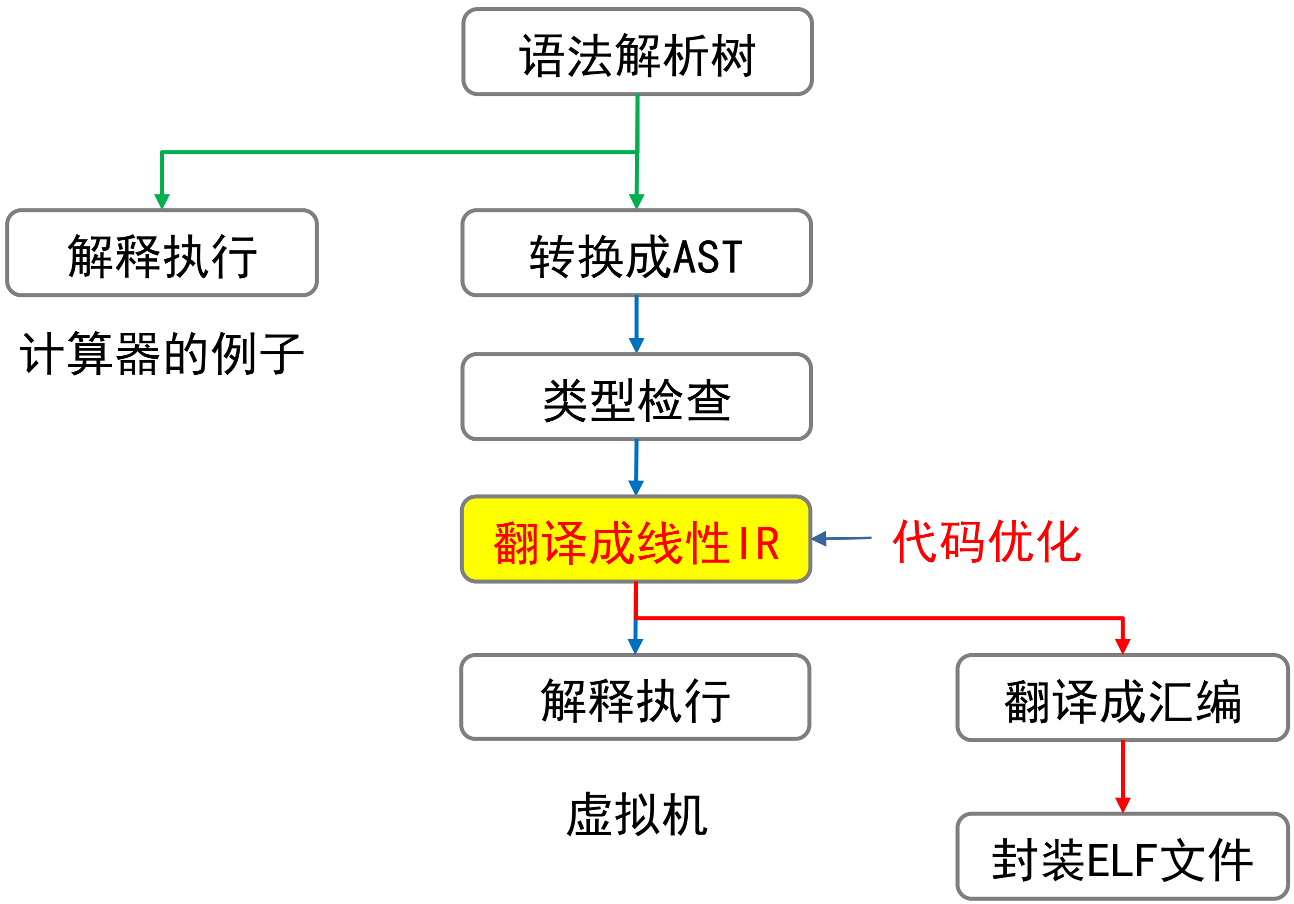
大纲

一、设计线性IR

二、翻译线性IR

三、解释执行和虚拟机

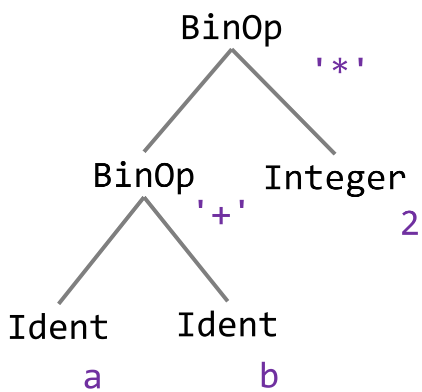
展望



语法解析树
解释执行
转换成AST
计算器的例子
类型检查
翻译成线性IR
代码优化
解释执行
翻译成汇编
虚拟机
封装ELF文件

线性IR的基本形式

- 三地址代码，由指令和地址组成；
  - 地址：变量名、常量、编译器生成的临时变量或存储单元



抽象语法树

```
%1 = a + b;
%2 = %1 * 2;
```

三地址线性IR

op	arg1	arg2	res
+	a	b	%1
*	%1	2	%2

四元式

	op	arg1	arg2
(1)	+	a	b
(2)	*	(1)	2

三元式

op	arg1	arg2
mov	a	r1
add	b	r1
mul	2	r1

两地址形式

```
mov   -4(%rbp), %eax
add   -8(%rbp), %eax
shl   $1, %eax
```

实际汇编代码

IR定义1：标识符和基本运算

- 标识符
  - 全局变量：@name
  - 局部变量：%name
  - 临时变量：%1、%2
  - 常量：1、2
- 二元整数运算：
  - add/sub/mul/div/rem
- 二元浮点数运算：
  - fadd/fsub/fmul/fdiv/frem
- 二元位运算：
  - and/or/xor
  - shl/ashr/lshr

```
%1 = a + b;
%2 = %1 * 2;
```

⇩

```
%1 = add i32, %a, %b;
%1 = mul i32, %1, 2;
```

练习：

- 假设%0=-2，%1=1，计算下列运算结果

`%3 = and i32 %0, %1`	00000000
`%4 = or i32 %0, %1`	ffffffff
`%5 = xor i32 %0, %1`	ffffffff
`%6 = shl i32 %0, 1`	fffffffc
`%7 = ashr i32 %0, 1`	ffffffff
`%8 = lshr i32 %0, 1`	7fffffff

浮点数运算需要单独的指令

- 浮点数表示比较独特：IEEE-754标准
- 计算方式：$mantissa \times (2^{exp} - 127)$
  - 如200可表示成01000011010010000000000000000000
  - 1.5625*2^7=200

01000011010010000000000000000000

exponent (8 bits)

mantissa (23 bits)

$$2^7 + 2^2 + 2^1 - 127 = 7$$

$$1 + 2^{-1} + 2^{-4} = 1.5625$$

将实数转换为浮点数

- 将实数11.25转换为二进制表示

```
11/2 = 5 + 1
5/2  = 2 + 1
2/2  = 1 + 0
1/2  = 0 + 1
```

```
0.25 * 2 = 0.5 + 0
0.50 * 2 = 0.0 + 1
```

```
  1011.01
1+.01101
```

⇨ exp = 3

```
01000001001101000000000000000000
```

练习

- 下列哪个小数可以使用浮点数精确表示?
  - 0.1, 0.2, 0.3, 0.4, 0.5

```
0.1 * 2 = 0.2 + 0      0.3 * 2 = 0.6 + 0      0.5 * 2 = 0 + 1
0.2 * 2 = 0.4 + 0      0.6 * 2 = 0.2 + 1
0.4 * 2 = 0.8 + 0      0.2 * 2 = 0.4 + 0
0.8 * 2 = 0.6 + 1      …
0.6 * 2 = 0.2 + 1
…
0001.100110011…        01.001100110011…       1.000…

0.1 = 001111011100110011…
0.2 = 001111100100110011…
0.3 = 001111101001100110…
0.4 = 001111101100110011…
0.4 = 001111110000000000…
```

IR定义2：类型转换

- 数据截断：trunc/fptrunc
- 数据扩充：zext/setx/fpext
- 浮点数整数互换：fptoui/fptosi/uitofp/sitofp
- 指针整数互换：ptrtoint/inttoptr

```
%1 = trunc i32 %0, i16
%2 = sitofp i32 %0, float
%3 = uitofp i32 %0, float
%4 = sext i32 %0, i64
%5 = zext i16 %0, i64
%6 = fptoui float %2, i32
%7 = fpext float %2, double
%8 = fptrunc double %7, float
%9 = inttoptr i64 %4, i8*
%10 = ptrtoint i8* %9, i32
```

IR定义3：数据存取

- 栈空间分配：stackalloc
- 堆空间分配：heapalloc
- 数据读取：load
- 数据存入：store

```
int a = 1;
a = a + 1;
```

⇨

```
%a = stackalloca i32
store i32 1, %a
%1 = load i32, %a
%2 = add i32 %1, 1
store i32 %2, %a
```

IR定义4：指针和地址操作

- 偏移地址：getptr

```
let a[2]:int;
a[1] = 99;
```

⇨

```
%a = alloca [2 x i32]
%1 = getptr *i32, %a, 1
store i32 99, %1
```

```
struct st{
  i:int;
  f:float;
};
struct st s;
s.i = 1;
```

⇨

```
%struct.st = type { i32, float }
%s = alloca %struct.st
%1 = getptr %struct.st, %s, 0
store i32 1, %1
```

IR定义5：控制流语句

- 比较运算：
  - icmp/fcmp
    - eq/ne
    - gt/ge/lt/le
    - ugt/uge/ult/ule
- 跳转指令：
  - jmp：直接跳转
  - cjmp：条件跳转
  - match

```
%0 = icmp eq i32 4, 5            F
%1 = icmp ne float 0.1, 0.2      F
%2 = icmp ult i16  4, 5          T
%3 = icmp sgt i16  4, 5          F
%4 = icmp ule i16 -4, 5          F
%5 = icmp sge i16  4, 5          F
```

```
cjmp %0, %BB1, %BB2
```

```
match i32 %0, %BBdefault [
  i32 0, %BB1
  i32 1, %BB2
  i32 2, %BB3 ]
```

if-else语句的IR

```
if(a)
    b++;
else
    b--;
```

⇒

```
  %1 = load i32 %a;
  %2 = load i32 %b;
  %3 = icmp ne i32 %1, 0;
  cjmp %3, %BB1, %BB2;

%BB1:
  %4 = add i32 %2, 1;
  store i32 %4, %b;
  br %BB3;

%BB2:
  %5 = sub i32 %2, 1;
  store i32 %5, %b;
  br %BB3;

%BB3:
```

while语句的IR

```
while(a)
    a--;
```

```
BB0:
  br %BB1;

%BB1:
  %1 = load i32 %a;
  %2 = icmp ne i32 %1, 0;
  br i1 %2, %BB2, %BB3;

%BB2:
  %3 = sub i32 %1, 1;
  store i32 %3, %a;
  br %BB1;

BB3:
  ...
```

match语句的IR

```
match(x){
    0: => { x = 0; }
    1: => { x = 1; }
    _: => { x = -1; }
}
```

```
%0 = load i32 %x;
match i32 %0, %BB3 [
    i32 0, %BB1
    i32 1, %BB2
  ]

%BB1:
  store i32 0, %x;
  jmp %BB4

%BB2:
  store i32 1, %x;
  jmp %BB4

%BB3:
  store i32 -1, %x;
  jmp %BB4

%BB4:
  ...
```

IR定义6：函数

- 函数声明：fn
- 函数调用：call
- 返回指令：ret

```
@define void @foo(i32 x){
    ...
}
@define i32 @bar(i32 y){
    ...
}
call void @foo(1)
%2 = call i32 @test(i32 %1)
```

大纲

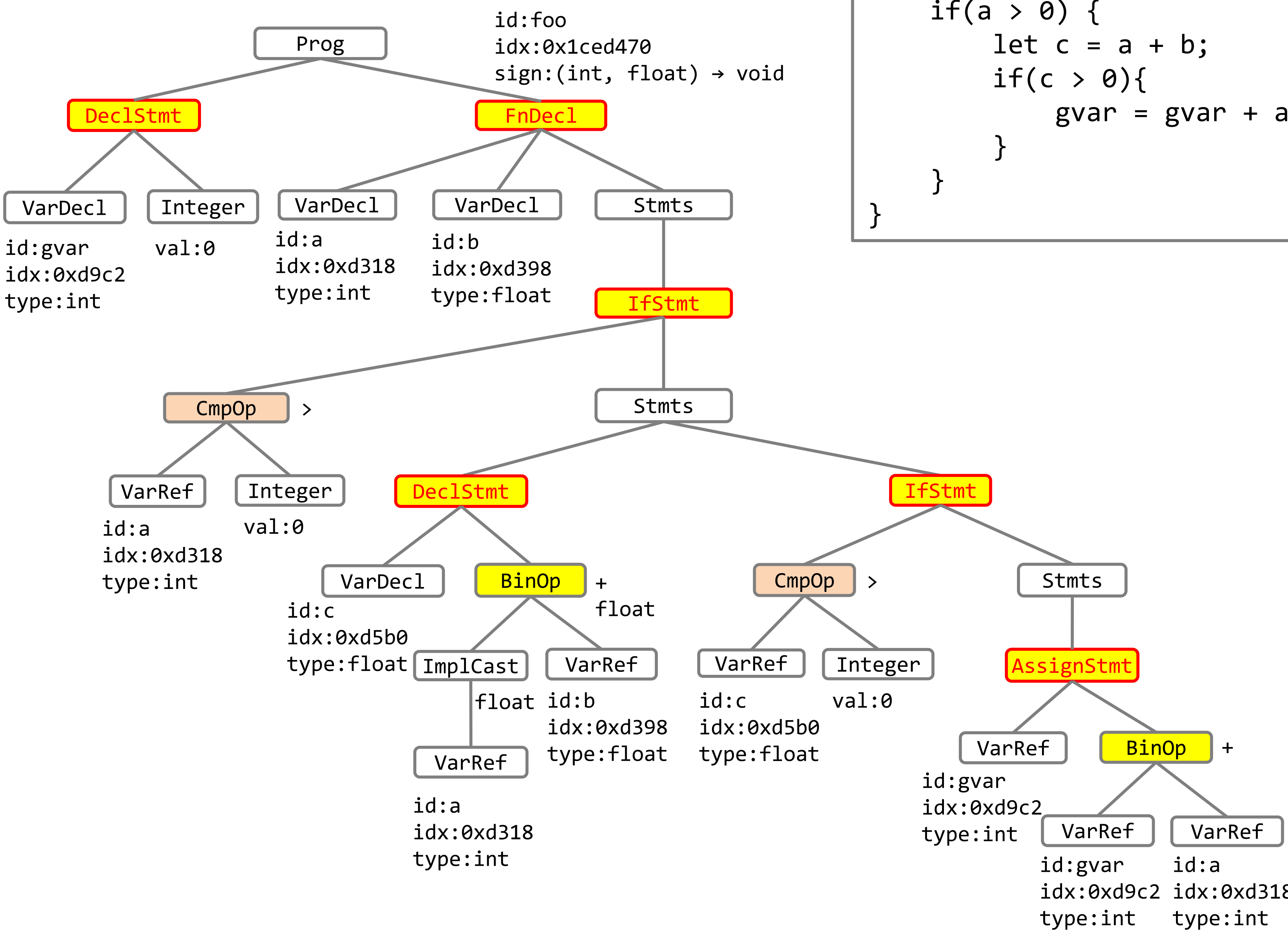
一、设计线性IR

二、翻译线性IR

三、解释执行和虚拟机

AST->线性IR

```
let gvar:int = 0;
fn foo(int a, float b){
    if(a > 0) {
        let c = a + b;
        if(c > 0){
            gvar = gvar + a;
        }
    }
}
```



线性IR

```
let gvar:int = 0;
fn foo(a:int, b:float){
    if(a > 0) {
        let c = a + b;
        if(c > 0){
            gvar = gvar + a;
        }
    }
}
```

```
@gvar = i32 0;
define fn foo(i32 %a0, float %b0){
    %a = stackalloc i32;
    store %a0, %a;
    %b = i32 stackalloc float;
    store float %b0, %b;
    %1 = load i32 %a;
    %2 = icmp ge %1, 0;
    cjmp %2, %BB1, %BB4;
%BB1:
    %3 = load i32 %a;
    %4 = sitofp %3;
    %5 = load float %b;
    %c = stackalloc float;
    %6 = fadd float %4, %5;
    store float %6, %c;
    %7 = load float %c;
    %8 = fcmp ge %7, 0;
    cjmp %8, %BB2, %BB3;
BB2:
    %9 = load i32 @gvar;
    %10 = load i32 %a;
    %11 = add i32 %9 %10;
    store %11, @gvar;
    jmp %BB3;
BB3:
    jmp %BB4;
BB4:
    ret;
}
```

基本思路

- 为每一种AST节点定义转换方式
- 前序遍历AST树

```
GenIR(node) {
    match (node.type) {
        FNDECL => { ... }
        DECLSTMT => { ... }
        ASSIGN => { ... }
        BINOP => { ... }
        CMPOP => { ... }
        IFSTMT => { ... }
        ...
    }
}
```

IR数据结构设计

```
struct ProgIR {
    gvlist:list<GlobalVar>;
    fnlist:list<FnIR>;
}

struct FnIR {
    id:int;
    bblist:list<BB>;
}

struct BB {
    id:int;
    list<InstType> ilist;
}
```

程序IR组成：全局变量IR+函数

函数组成：id+代码块

代码块组成：id+指令列表

BINOP

```
BINOP => {
    if (!IsLeaf(node.child[0])) {
        GenIR(curbb,node.child[0]);
    }
    if (!IsLeaf(node.child[1])) {
        GenIR(curbb,node.child[1]);
    }
    ir = CreateIR(node.op,
                  node.child[0].id,
                  node.child[1].id);
    curbb.irlist.add(ir);
}
```

如子节点不是叶子节点，
=>递归生成子节点的IR，
=>子节点取得临时变量标识

使用子节点变量标识
如为临时变量，则直接使用，
如为局部变量，则先load

将IR加入当前代码块

ASSIGN

```
ASSIGN => {
    if (!IsLeaf(node.child[1])) {
        GenIR(curbb,node.child[1]);
    }
    ir = CreateIR(ASSIGN,
                  node.child[0].id
                  node.child[1].id);
    curbb.irlist.add(ir);
}
```

先计算，后store

DECLSTMT

```
DECLSTMT => {
    if (node.child.length < 2)) {
        ir = CreateIR(DECL,
                      node.child[0]);
    } else {
        if (!IsLeaf(node.child[1])) {
            GenIR(curbb,node.child[1]);
        }
        ir = CreateIR(DECL,
                      node.child[0]
                      node.child[1].id);
    }
    curbb.irlist.add(ir);
}
```

无初始化，仅stackalloc

含初始化，先stackalloc后赋值

IFSTMT

```
IFSTMT => {
    let childnum = node.child.size();
    let bbs:[BB;] = CreateBB(curfn,childnum+1)
    if (!IsLeaf(node.child[0])) {
        GenIR(curbb,node.child[0]);
    }
    ir = CreateIR(CJMP,
                  node.child[0].id,
                  bbs);
    curbb.irlist.add(ir);
    GenIR(bbs);
}
```

分配执行完后的BB

条件语句模块

参考if跳转关系

FNDECL

```
FNDECL => {
    let curfn = CreateIR(FNDECL,node.sign);   创建函数IR
    prog.add(curfn);
    let childnum = node.child.size();
    let curbb = CreateBB(curfn);
    for i in 0..childnum-1 {
        if (node.child[i].type == PAR) {   参数节点，stackalloc
            ir = CreateIR(DECL,
                          node.child[i]);
            curbb.irlist.add(ir);
        }
        if (node.child[i].type == STMTS) {   函数体
            GenIR(curbb,node.child[i]);
        }
        if (node.child[i].type == RETTY) {   返回值，stackalloc
            ir = CreateIR(DECL,
                          node.child[i]);
            curbb.irlist.add(ir);
        }
    }
}
```

大纲

一、设计线性IR

二、翻译线性IR

三、解释执行和虚拟机

解释执行：interpreter

- 解释执行另外一个程序：源代码/AST/IR
- 无需考虑后端，简化了语言的实现
- 计算器的例子：直接翻译为目标机器指令

如何设计IR解释执行器？

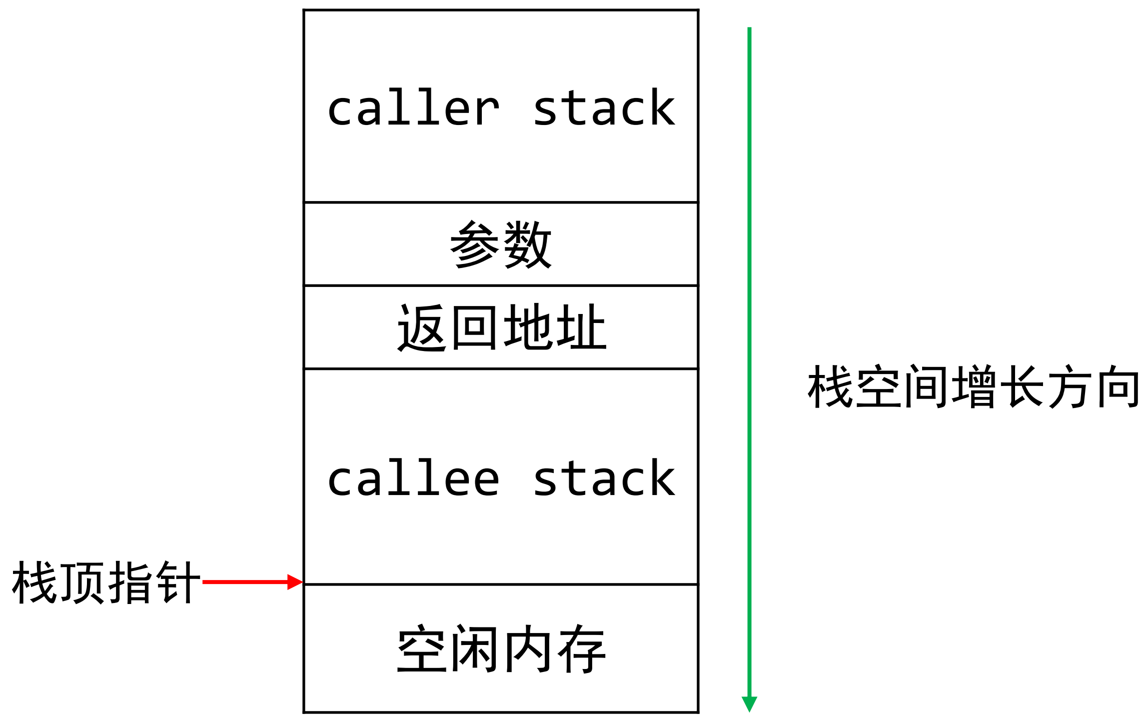
- 通过循环不断获取下一条指令并执行

```
enum {
    addInst,
    subInst,
    mulInst,
    divInst,
    icmpInst,
    cjmpInst,
    ...
} instType;
```

```
static prog:[instType;n] = { ... };
let pc:*instType = prog;
while(1) {
    match (*pc++) {
        addInst => { ... }
        subInst => { ... }
        ...
    }
}
```

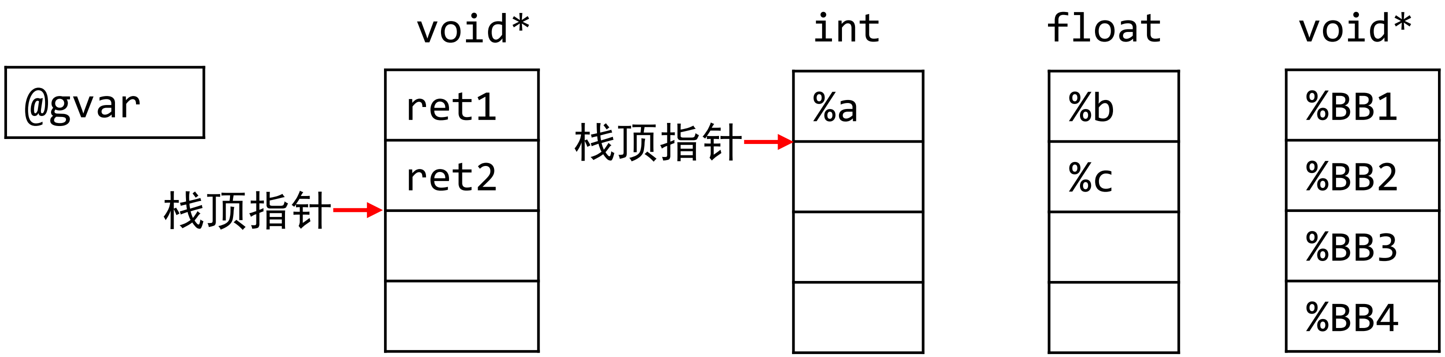
如何保存数据：Activation Record

- 为每个函数调用分配一块儿内存空间
- 函数自身所需栈空间可在编译时确定（栈vs堆）
- 函数返回后收回



简单实现思路

- 全局变量在程序运行期间有效，单独存放
- 函数调用栈使用数组模拟，记录返回地址
- 局部变量使用数组保存（模拟栈），每个数组一个数据类型
  - 数组大小=>内存空间
- 动态维护栈顶指针

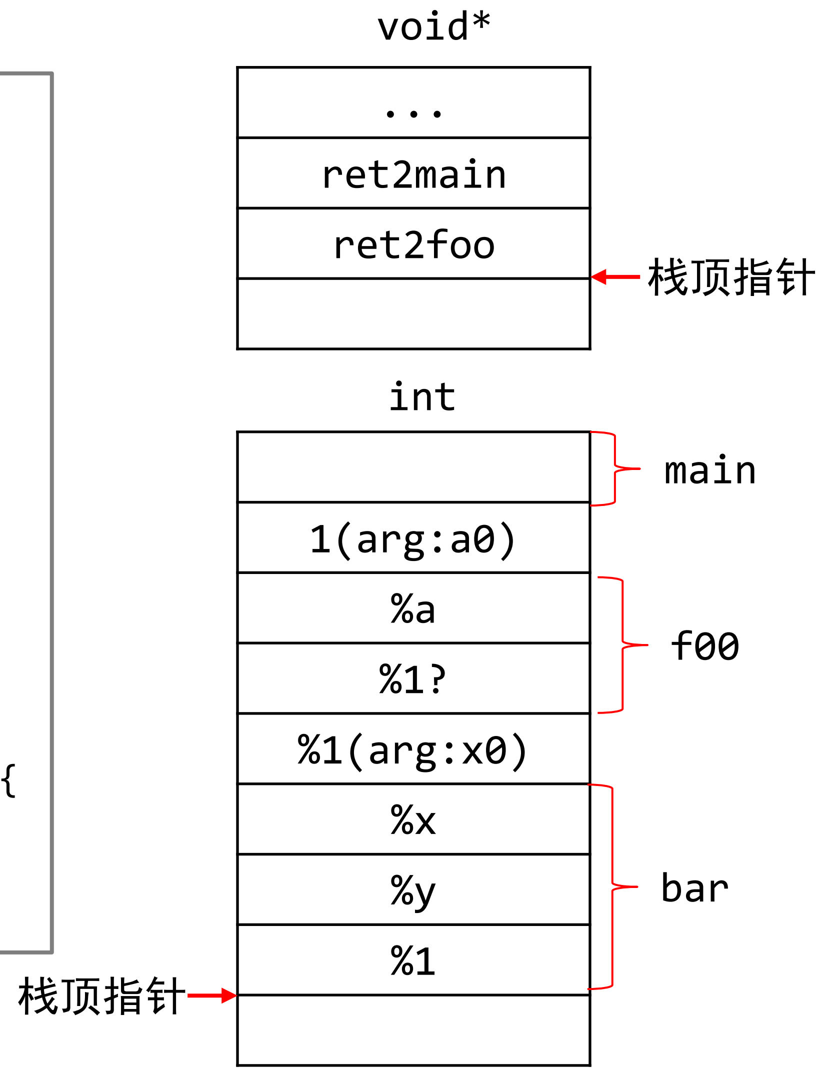


应用举例

代码执行→

```
define fn i32 bar(i32 %x0){
    %x = stackalloc i32;
    store i32 %x0 %x;
    %y = stackalloc i32;
    ...
    ret %1;
}

define fn void foo(i32 %a0){
    %a = stackalloc i32;
    store i32 %a0 %a;
    %1 = load %a
    call bar(%1);
    ret;
}
define fn main(i32 argc, i8** argv) {
    ...
    call foo(1);
}
```



逃逸分析

- 局部变量在函数返回后是否继续被使用？
  - Bug！！！
  - 指向该变量的指针逃逸到其它函数或线程
  - 应在堆上分配内存
- 对于Java这种默认在堆上分配内存的语言，可通过逃逸分析将内存分配在栈上
  - 对象内存与函数栈帧生命周期绑定
  - 降低垃圾回收负担，节约内存

```
fn foo -> &i32(){
    let i:i32 = 999999;
    return &i;
}
```

```
public static void example() {
    Foo foo = new Foo();
}

class Foo {
    private int i;
    public void set(int i) {
        this.i = i;
    }
}
```

虚拟机

- 为解释执行提供了程序运行抽象
  - 内存管理（栈、堆、垃圾回收）
  - 寄存器
  - 多线程
  - ...
- 优点：
  - 高效：优化策略
  - 方便："Write once, run anywhere"
- 比较有名的虚拟机：
  - Java：HotSpot、Dalvik（Android）
  - Javascript：Chrome v8、Chakra、SpiderMonkey、JavaScriptCore

优化思路

- 使用寄存器储存临时变量
- Threaded code
- JIT
- ...

使用Threaded Code

- Match-Case的问题：需要两次跳转
  - 一次间接跳转：跳转到分支代码
  - 一次直接跳转：返回循环入口
- 可否跳转一次？
  - 为每个指令设计一个处理函数或代码块
  - 开启尾递归

```
static prog:[instType;n] = { ... };
let pc:*instType = prog;

static fn add() {
    ...
    (*++pc.fnaddr)();
}
...
(*pc.fnaddr)();
```

总结

- 线性IR的设计：指令集和翻译模式
  - 标识符、基本运算、数据存取、控制流、函数...
- 如何将AST翻译成线性IR
  - 前序遍历AST树
- 解释执行IR
  - 函数调用栈、Activation Record