COMP130014 编译

第七讲：线性IR

徐 辉

xuh@fudan.edu.cn

大纲

一、线性IR

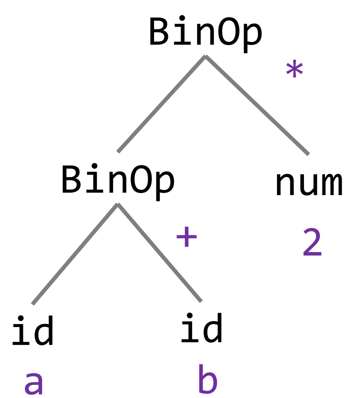
二、翻译线性IR

三、解释执行

一、线性IR定义

线性IR的基本形式

- 指令名 + 参数
  - 参数：变量名、常量、编译器生成的临时变量或存储单元
- 比较有名的IR：LLVM IR、 GCC GIMPLE、Java Bytecode



抽象语法树

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

三地址线性IR

TeaPL的IR

- 选取LLVM IR的子集
  - LLVM IR参考: https://llvm.org/docs/LangRef.html
- 可使用现成工具执行IR：lli

```
@g = global i32 10

define i32 @fib(i32 %0) {
    %x = alloca i32
    store i32 %0, i32* %x
    %g0 = load i32, i32* @g
    ret i32 %g0
}

define i32 @main() {
    %r0 = call i32 @fib(i32 1)
    ret i32 %r0;
}
```

```
#: lli foo.ll
#: echo $?
```

标识符和基础类型

- 全局变量/函数名称：@name
- 局部变量/临时变量：%x、%0（不可重复，数字编号需连续）

```
@g = global i32 10

define i32 @fib(i32 %0) {
    %x = alloca i32
    store i32 %0, i32* %x
    %g0 = load i32, i32* @g
    ret i32 %g0
}

define i32 @main() {
    %r0 = call i32 @fib(i32 1)
    ret i32 %r0;
}
```

全局变量g

函数fib

局部变量%x

临时变量%g0

数据存取

- 类型：void、i32、i32*、i8、i8*、i1
- 栈空间分配：alloca
- 数据存取：load/store

```
@g = global i32 10

define i32 @fib(i32 %0) {
    %x = alloca i32
    store i32 %0, i32* %x
    %g0 = load i32, i32* @g
    ret i32 %g0
}

define i32 @main() {
    %r0 = call i32 @fib(i32 1)
    ret i32 %r0;
}
```

声明全局变量@g：类型为i32*，初始值10

为局部变量%x分配空间：类型为i32*

加载%g临时变量%g0：类型为i32

函数

- 定义：define
- 调用：call
- 返回：ret

```
@g = global i32 10

define i32 @fib(i32 %0) {
    %x = alloca i32
    store i32 %0, i32* %x
    %g0 = load i32, i32* @g
    ret i32 %g0
}

define i32 @main() {
    %r0 = call i32 @fib(i32 1)
    ret i32 %r0;
}
```

函数fib：类型为(i32)->i32

返回%g0

函数main：类型为(void)->i32

调用函数fib

函数声明

- 声明：declare
- 声明和定义不能在一个ll文件中，使用llvm-link工具链接

```
declare i32 @fib( i32 )

define i32 @main() {
    %r0 = call i32 @fib(i32 1)
    ret i32 %r0;
}
```

在a.ll文件中声明函数fib

```
define i32 @fib(i32 %0) {
    %x = alloca i32
    store i32 %0, i32* %x
    %g0 = load i32, i32* @g
    ret i32 %g0
}
```

在b.ll文件中定义函数fib

```
#: llvm-link a.ll b.ll -o c.ll
```

数组类型存取

- 获取地址：getelementptr

```
%1 = alloca [2 x i32]
%2 = getelementptr [2 x i32],
     [2 x i32]* %1,
     i32 0,
     i32 0
store i32 99, i32* %2
%3 = load i32, i32* %2
```

创建一维数组

数组基地址

索引为0的元素

```
@a = global [2 x i32]
            [i32 1, i32 2]
```

全局数组声明和初始化

结构体类型数据存取

```
%mystruct = type { i32, i32 }
define i32 @main() {
  %1 = alloca %mystruct
  %2 = getelementptr %mystruct,
      %mystruct* %1,
      i32 0,
      i32 0
  store i32 1, i32* %2
  ret i32 0
}
```

定义mystruct数据类型

创建mystruct类型的对象

获取mystruct第一个成员的指针

算数运算

- 加、减、乘法运算：add/sub/mul
- 除法：sdiv

```
%2 = alloca i32
%3 = add i32 %0, 1
%4 = sub i32 %3, 2
%5 = mul i32 %3, 3
%6 = sdiv i32 %4, 4
store i32 %6, i32* %2
```

浮点数运算用fadd/fsub/fmul/fdiv

关系运算

- 一条指令：icmp
- 多种参数：sgt/sge/slt/sle/eq/ne

```
%2 = load i32, i32* %1
%3 = icmp sgt i32 %2, 0
%4 = icmp sge i32 %2, 0
%5 = icmp slt i32 %2, 0
%6 = icmp sle i32 %2, 0
%7 = icmp eq i32 %2, 0
%8 = icmp ne i32 %2, 0
```

s: signed
g: greater
l: less
e: equal
n: not

类型转换

- 扩充：zext
- 截断：trunc

```
%a = alloca i32
%b = alloca i8
%t0 = load i32, i32* %a
%t1 = icmp ne i32 %t0, 0
%t2 = zext i1 %t1 to i32
store i32 %t2, i32* %a
%t3 = trunc i32 %t2 to i8
```

类型转换：i1=>i32

类型转换：i32=>i8

逻辑运算

- LLVM中没有专门的逻辑运算指令
- 基于位运算实现

```
%r = xor i1 %a, true      → !%a
%r = and i1 %a, %b        → %a && %b
%r = or i1 %a, %b         → %a || %b
```

控制流指令

- 直接跳转：br + 目标
- 条件跳转：br + 条件 + 目标1 + 目标2

```
  %2 = alloca i32
  store i32 0, i32* %2
  %3 = load i32, i32* %2
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %bb1, label %bb2      → 条件跳转
bb1:
  store i32 1, i32* %2
  br label %bb3                         → 直接跳转
bb2:
  store i32 0, i32* %2
  br label %bb3
bb3:
  %r0 = phi i32 [0, %bb1], [%3, %bb2]
  ret i32 %r0
}
```

数据流指令

- 条件赋值：Phi

```
  %2 = alloca i32
  store i32 0, i32* %2
  %3 = load i32, i32* %2
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %bb1, label %bb2
bb1:
  store i32 1, i32* %2
  br label %bb3
bb2:
  store i32 0, i32* %2
  br label %bb3
bb3:
  %r0 = phi i32 [0, %bb1], [%3, %bb2]
  ret i32 %r0
}
```

如前序代码块为%bb1，则%r0=0
如前序代码块为%bb2，则%r0=%3

基于短路控制流实现逻辑或和与

```
bb1:
  br i1 %a, label %bb2, label %bb3
bb2:
  br label %bb3
bb3:
  %10 = phi i1 [false, %bb1], [%b, %bb2]
```

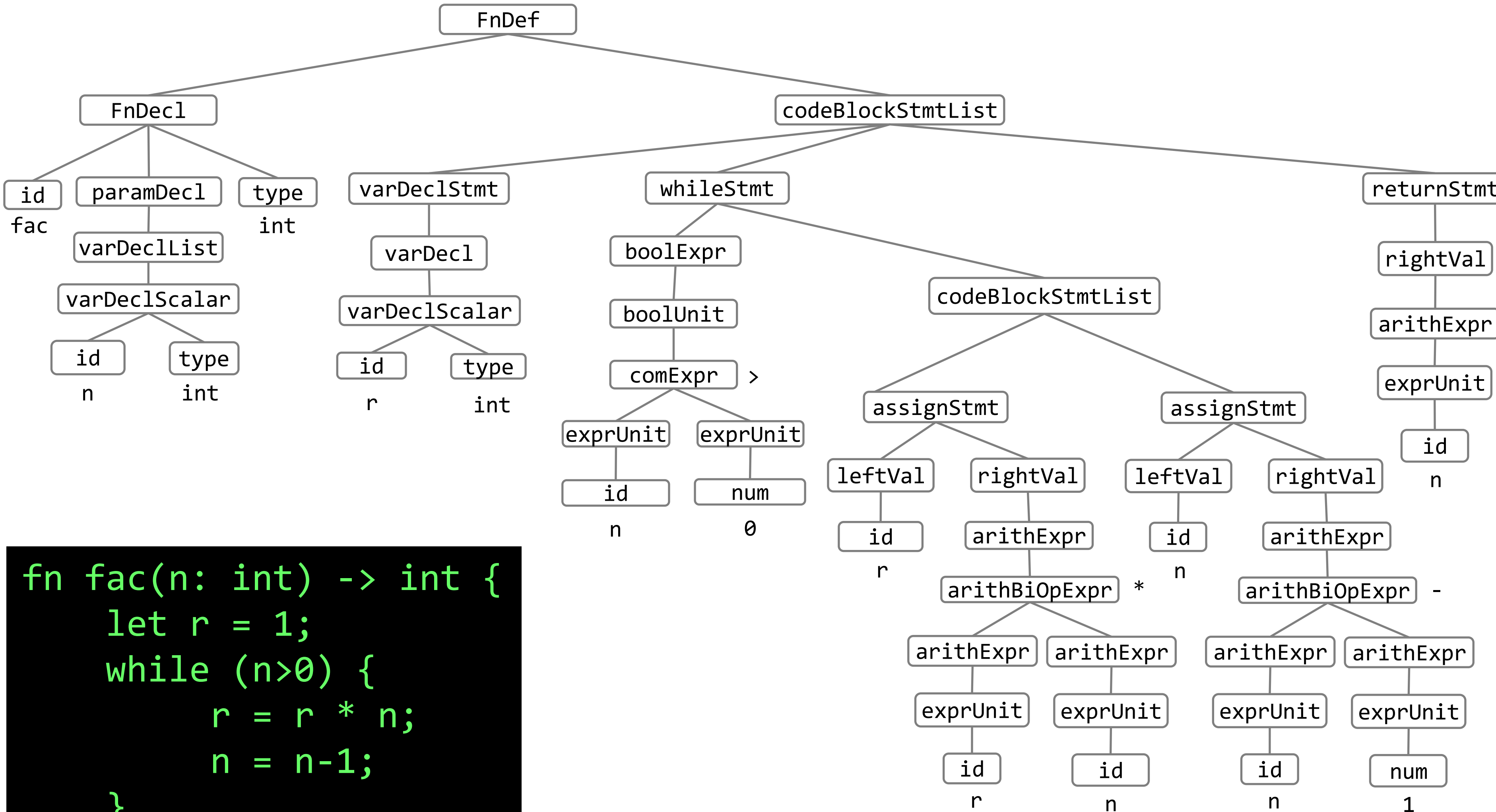
→ %a && %b

```
bb1:
  br i1 %a, label %bb3, label %bb2
bb2:
  br label %bb3
bb3:
  %10 = phi i1 [true, %bb1], [%b, %bb2]
```

→ %a || %b

二、翻译线性IR

思考：如何将AST翻译为线性IR




```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

AST=>LLVM IR

- 基本思路：
  1) 遍历AST，创建全局函数/变量IR
  2) 遍历函数AST，创建代码块编号
  3) 翻译每个代码块的内容
- 关键：
  - 代码块编号和引用（br）
  - 变量编号和引用（def-use）

```
struct ProgIR { // 程序IR
    gvs: Vec<GlobalVar>,
    fns: Vec<FnIR>,
}
struct FnIR { // 函数组成
    id: int,
    sign: FnSignIR,
    bbs: Vec<BB>,
}
struct BB { // 代码块组成
    id: int,
    insts: Vec<Inst>,
}
```

目标IR数据结构示例

代码块编号和引用

- 每个代码块都应以terminator结尾：br/ret

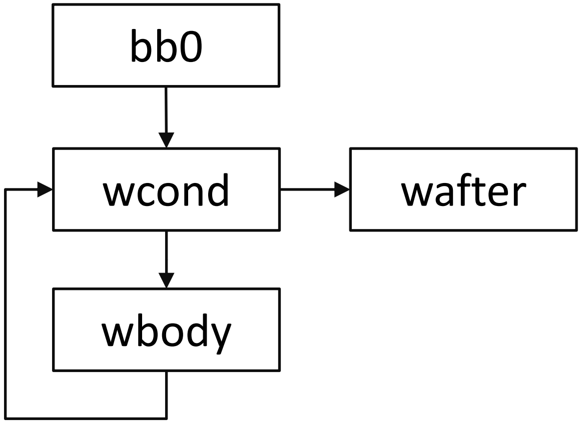
```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n - 1;
    }
    ret r;
}
```

```
define i32 @fac(i32 %0) {
bb0:
    ...
    br label %bb1
bb1: ; while cond
    ...
    br i1 %cond? label %bb2, label %bb3
bb2: ; while body
    ...
    br label %bb1
bb3:
    ...
    ret
}
```

控制流嵌套的例子：递归下降编号

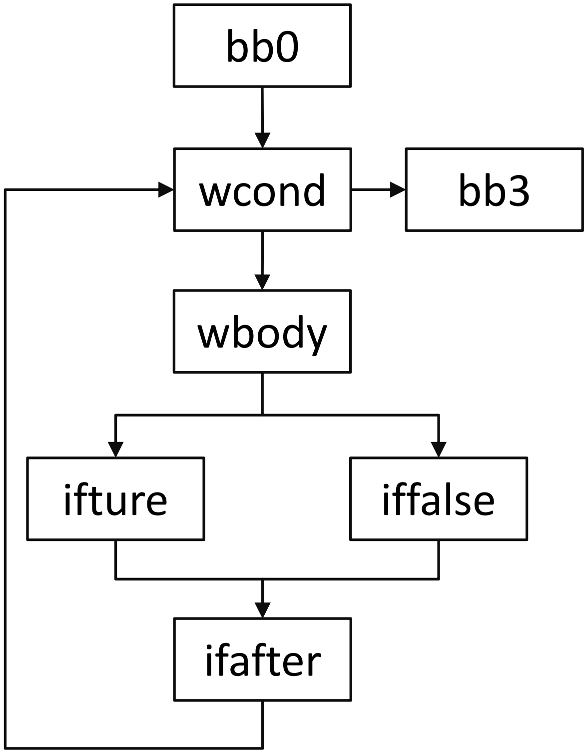
```
fn collatz(n:int) -> int {
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    ret n;
}
```

```
define i32 @collatz(i32 %0) {
bb0:
...
ret i32 ...
}
```




```
define i32 @collatz(i32 %0) {
bb0:
    ...
    br label %wcond
wcond: ; while cond
    ...
    br i1 %t1, label %wbody, label %wafter
wbody: ; while body
    ...
    br %bb1
wafter: ; after while
    ...
    ret i32 ...
}
```

控制流嵌套的例子：递归下降编号




```
define i32 @collatz(i32 %0) {
bb0:
    ...
    br label %bb1
wcond:    ; while condition
    br i1 %t1, label %wbody, label %wafter
wbody:    ; while body; if condition
    ...
    br i1 %t2, label %ifture, label %iffalse
ifture:  ; if true branch
    ...
    br label %ifafter
iffalse:  ; if false branch
    ...
    br label %ifafter
ifafter:
    br label %wcond
wafter:
    ...
    ret %r
}
```

变量编号和引用

- 消除块与块之间的数据依赖关系
- 块内依赖：使用变量前先load，更新后立即store

```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    ...
}
```

编号要求和方法

- lli要求：
  - 每个变量（编号）只能定义一次
  - 如果使用纯数字编号，必须从%0开始且连续（代码块和变量名共享）
- 编号方法：
  - 翻译IR时为由于顺序影响，如难以保证编号连续性，避免重复即可
  - 按出现顺序（线性）重命名每一个代码块和变量名
  - 可读性考虑：
    - 代码块用bb编号或纯数字
    - 局部变量用%x名称或纯数字
    - 临时变量用%r1或纯数字

IR翻译结果

```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

练习：翻译IR

```
fn collatz(n:int) -> int{
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    ret n;
}
```

```
define i32 @collatz(i32 %0) {
bb0:
    ...
    br label %bb1
bb1:   ; while condition
    br i1 %t1, label %bb2, label %bb3
bb2:   ; while body; if condition
    ...
    br i1 %t2, label %bb2.1, label %bb2.2
bb2.1: ; if true branch
    ...
    br label %bb2.3
bb2.2: ; if false branch
    ...
    br label %bb2.3
bb2.3:
    br label %bb2
bb3:
    ...
    ret %r
}
```

练习：翻译IR

```
let a[10]:int = {1,2,3,4,5,6,7,8,9,10};
fn binsearch(x:int) -> int {
    let high:int = 9;
    let low:int = 0;
    let mid:int = (high+low)/2;
    while(a[mid]!=x && low < high) {
        mid=(high+low)/2;
        if(x<a[mid]) {
            high = mid-1;
        } else {
            low = mid +1;
        }
    }
    if(x == a[mid]) {
        ret mid;
    }
    else {
        ret -1;
    }
}
```

```
fn main() -> int {
    let r = binsearch(2);
    ret r;
}
```

三、解释执行

解释执行

- 解释执行对象：线性IR
- 主要思路：
  - 找到程序入口，按照线性IR指令出现顺序和跳转关系执行
  - 遇到函数创建栈帧，为变量分配空间
  - 为全局变量分配空间

按照IR指令顺序执行

- 通过循环不断获取下一条IR指令并执行

```
enum {
    loadInst,
    addInst,
    subInst,
    mulInst,
    divInst,
    brInst,
    callInst,
    ...
} instType;
```

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

使用Threaded Code

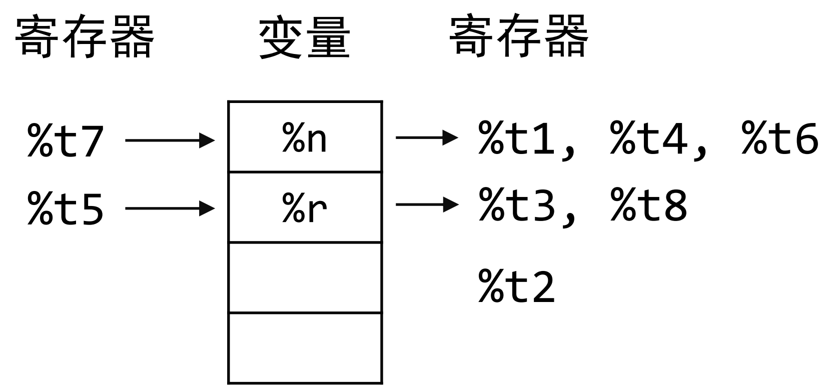
- while-match的问题：需要两次跳转
  - 跳转到分支代码
  - 返回循环入口
- 跳转一次：为每条指令设计一个处理函数或代码块

```
while(1) {
    match (*pc++) {
        addInst => { ... }
        subInst => { ... }
        ...
    }
}
```

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

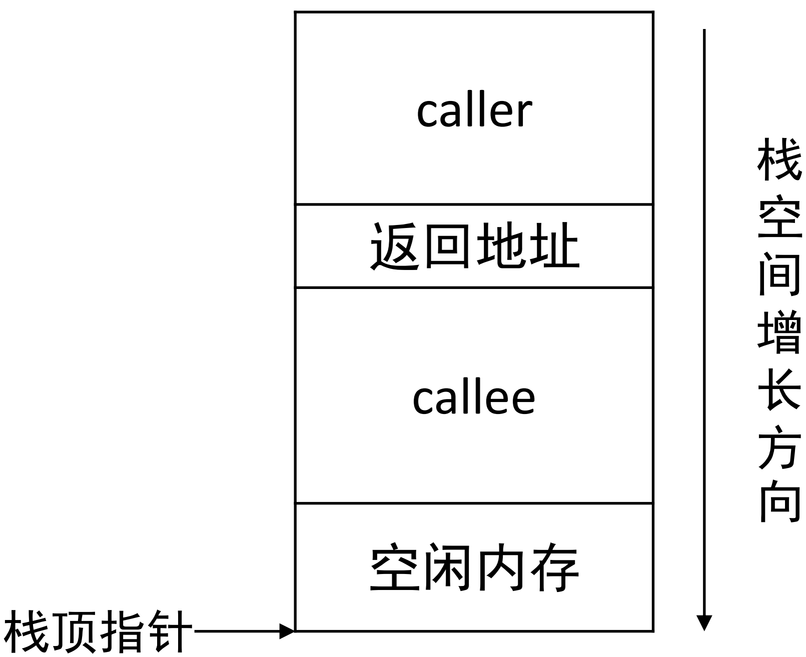
如何保存每条指令的运行效果？

```
define i32 @foo( i32 %0 ) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```



函数栈帧：Activation Record

- 栈帧：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（alloca）
- 栈帧空间在函数返回后收回




```
fn foo() -> &i32(){
    let i:int = 100;
    ret &i;
}
```

Bug!!!

逃逸分析?

栈虚拟机/寄存器虚拟机

- LLVM IR为三地址IR，与Java Bytecode/WebAssembly不同

```
//Java Bytecode
Load a
Load b
Add
Store c
```

```
id = 0;
loadInst => {
    r[id++] = *arg1;
}
addInst => {
    r[id++] = r[id-1]+r[id-2];
}
storeInst => {
    *arg1 = r[id];
}
```

寄存器虚拟机实现方式

```
stack s;
loadInst => {
    s.push(*arg1);
}
addInst => {
    v1 = s.pop();
    v2 = s.pop();
    v2 = v1 + v2;
    s.push(v2);
}
storeInst => {
    v1 = s.pop ();
    *arg1 = v1;
}
```

栈虚拟机实现方式

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
- 比较有名的虚拟机：
  - Java：HotSpot、Dalvik（Android）
  - Javascript：Chrome v8、Chakra、SpiderMonkey
  - WebAssembly: Wasmtime、Wasm3、Wasmer
- 虚拟机优化思路：
  - JIT优化
  - ...


Sun
microsystems


Meta
1 Hacker Way
ONE WAY


facebook
1 Hacker Way