

COMP130014 编译

第十讲：IR过程间优化

徐 辉

xuh@fudan.edu.cn



IR过程间优化

一、内联优化

二、尾递归优化

三、其它优化

一、内联优化

内联的好处

- 减少函数调用规约带来的运行时开销
- 带来更多IR过程内代码优化的可能性

```
fn foo(a:int) -> int {  
    let b:int = a + 1;  
    ret bar(b);  
}
```



```
fn bar(b:int) -> int {  
    let c:int = b - 1;  
    ret c;  
}
```



```
fn foo(a:int) -> int {  
    let b:int = a + 1;  
    let c:int = b - 1;  
    ret c;  
}
```



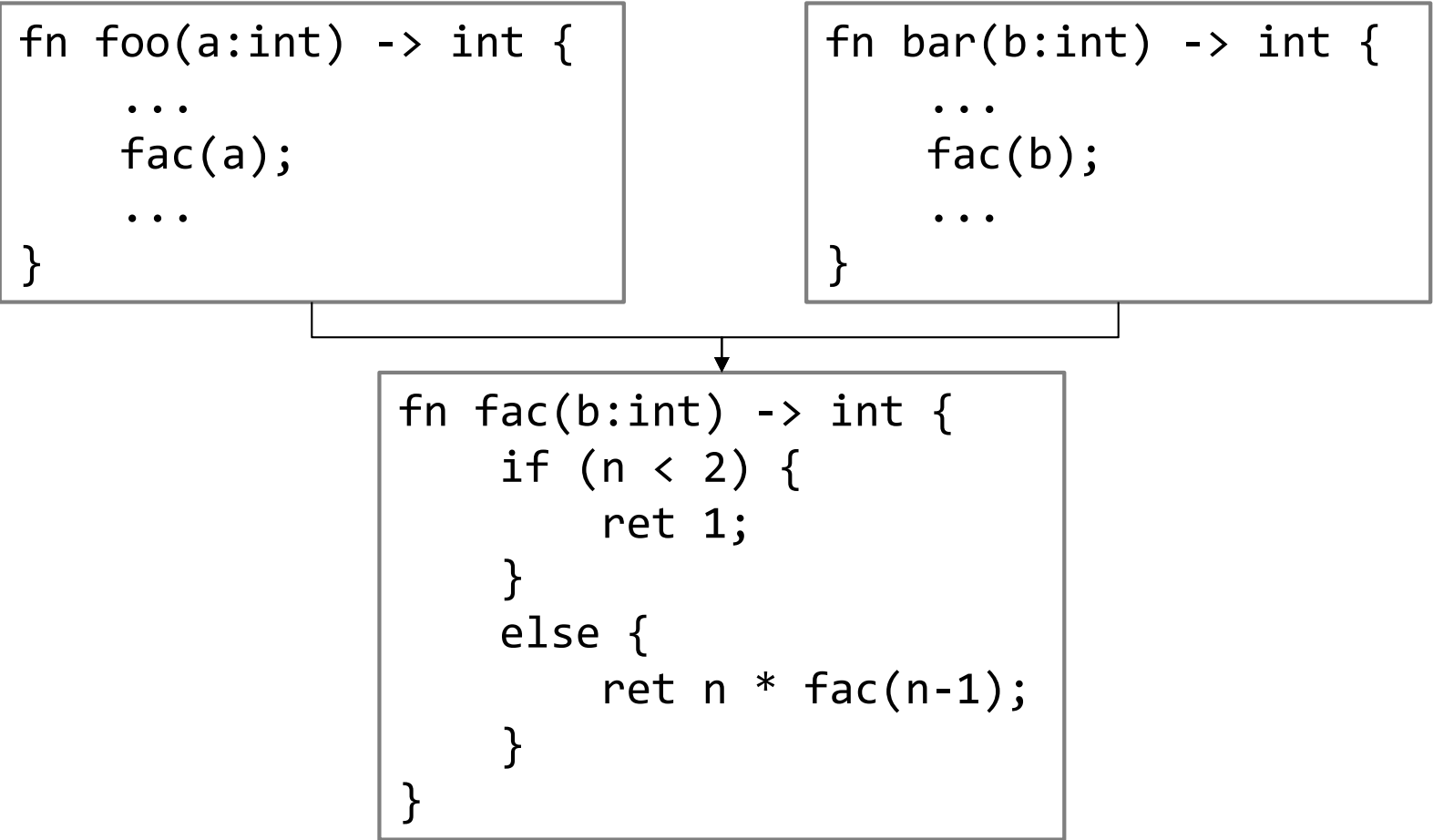
```
fn foo(a:int) -> int {  
    ret a;  
}
```

内联的副作用

- 代码复制可能会增大代码体积
- 加剧指令获取开销：cache miss
- 加重寄存器分配负担（可以忽略不计）

```
fn foo(a:int) -> int {  
    ...  
    fac(a);  
    ...  
}
```

```
fn bar(b:int) -> int {  
    ...  
    fac(b);  
    ...  
}
```



```
fn fac(b:int) -> int {  
    if (n < 2) {  
        ret 1;  
    }  
    else {  
        ret n * fac(n-1);  
    }  
}
```

对比下列代码的性能

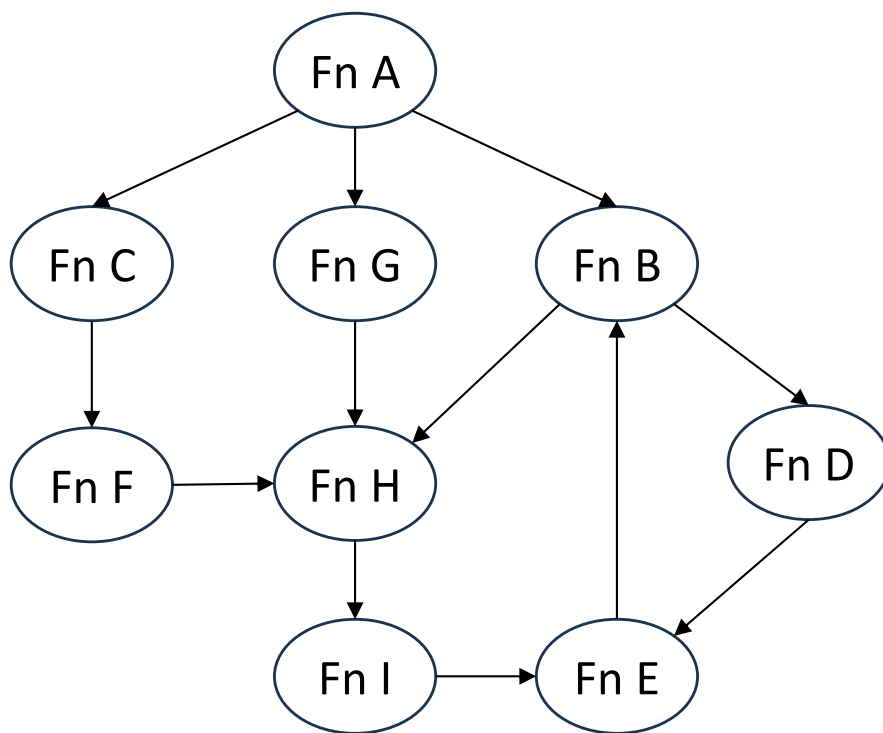
```
// C代码
inline void inline_callee(int* data, int size) {
    for (int i = 0; i < size; i++) {
        data[i] += 1;
    }
}

void non_inline_callee(int* data, int size) {
    for (int i = 0; i < size; i++) {
        data[i] += 1;
    }
}

void inline_caller(int* data, int size) {
    for (int i = 0; i < 1000; i++) {
        inline_callee(data, size);
        // non_inline_callee(data, size);
    }
}
```

内联优化问题

- 函数调用图：有向有环图（含有非自然循环）
- 如何选取优化的callsites?
 - 背包问题（NP-hard）：给定bugget上限，选取最优的内联函数组合



$$\begin{aligned} \max \quad & \sum_{i=1}^n reward_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n cost_i x_i \leq budget \text{ and } x_i \in \{0,1\} \end{aligned}$$

内联收益和开销评价

- 被调用频次较高的callsites，估计方法：
 - 循环内（hot region）
 - 函数入口处
 - 利用运行信息统计调用频次（profile-guided）
- 内联后有利于过程内优化
 - 参数为常量

利用函数调用上下文信息优化代码

普通内联

partial evaluation
(程序特化, 函数克隆)

编译时执行

参数均未知

部分参数已知

全部参数已知

```
fn fac(n:int, r:int) -> int {  
    if (n < 2) {  
        ret r;  
    } else {  
        ret factorial(n-1, n*r);  
    }  
}  
  
fn foo(x:int) -> int {  
    foo(x, x);  
    foo(1, x);  
    foo(0, 0);  
}
```

内联开销评价

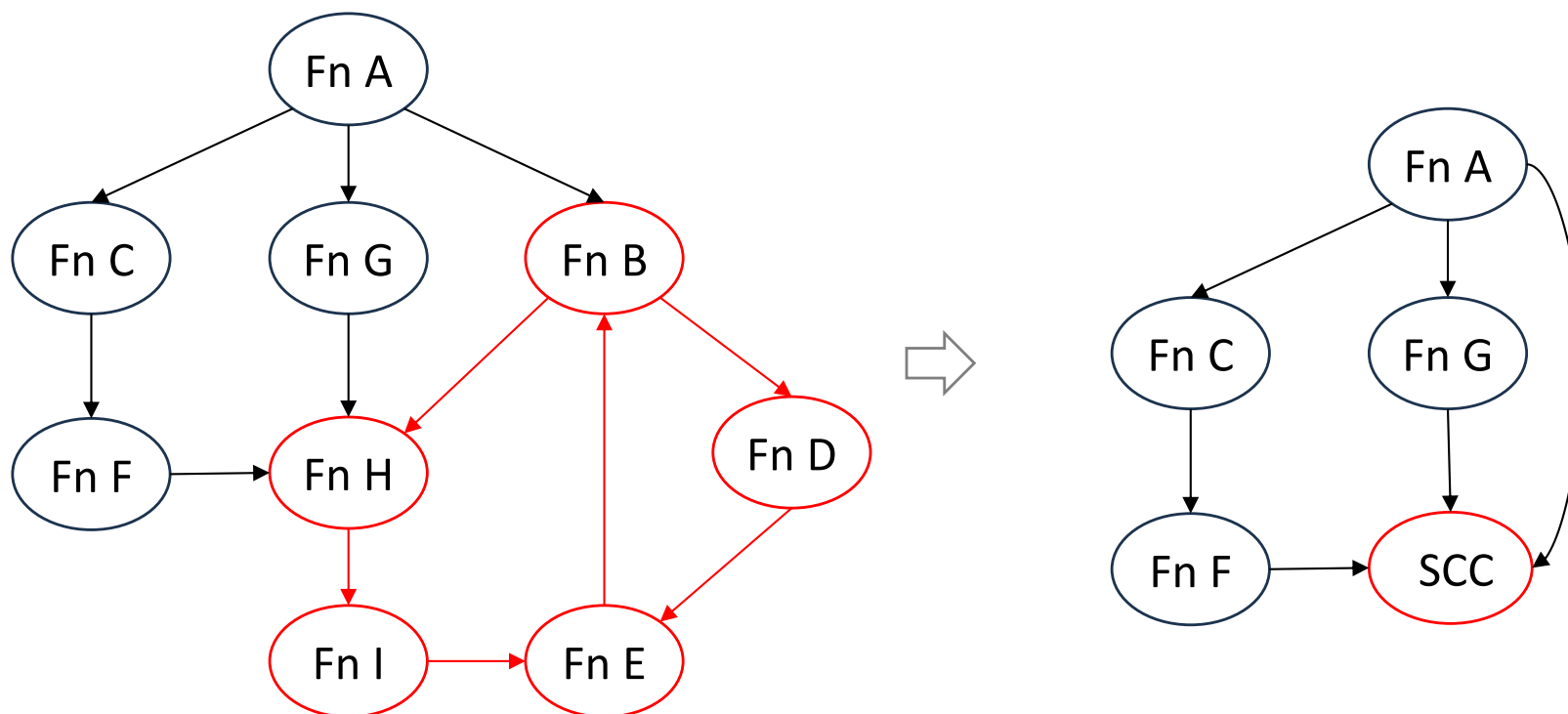
- 代码体积
 - 内联小函数：影响较小
 - 内联private函数：可全部内联，无需备份
- 加剧指令获取开销
 - 内联的callee含有循环

贪心式内联优化算法

```
Input: Call Graph  $G(V,E)$ 
Init:
     $S = \text{NULL}$  // 记录可以被内联的函数调用
     $C = 0$  // 记录内联代价
Foreach  $e$  in  $E$ :
    If (inlineable( $e$ )): // 排除不可内联的函数调用, 如间接调用
        BenefitEstimation( $e$ )
         $S.\text{insert}(e)$  // 基于收益排序
Foreach  $s$  in  $S$ :
     $\text{cost} = \text{CostEstimation}(s)$ 
     $C = C + \text{cost}$ 
    if ( $C > \text{budget}$ ):
         $S.\text{remove}(s)$ 
```

算法应用准备

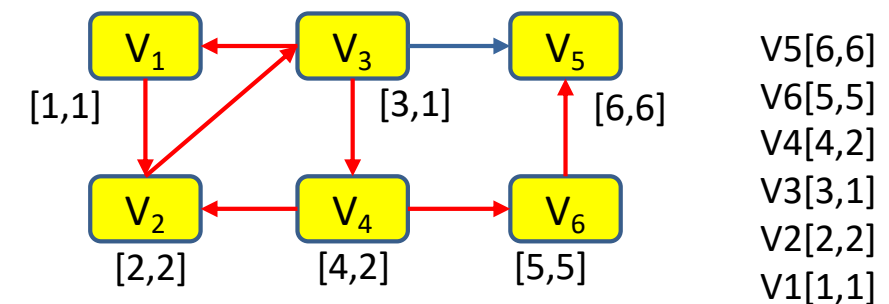
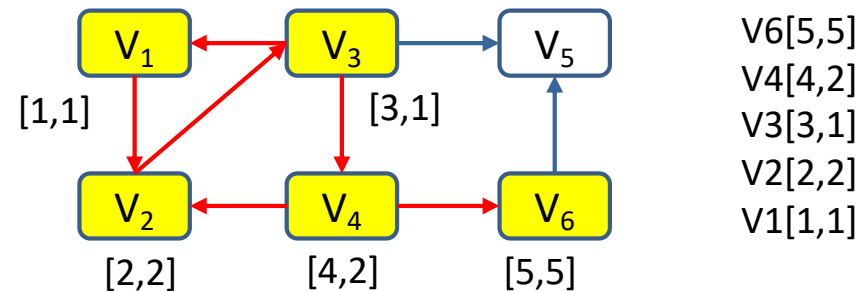
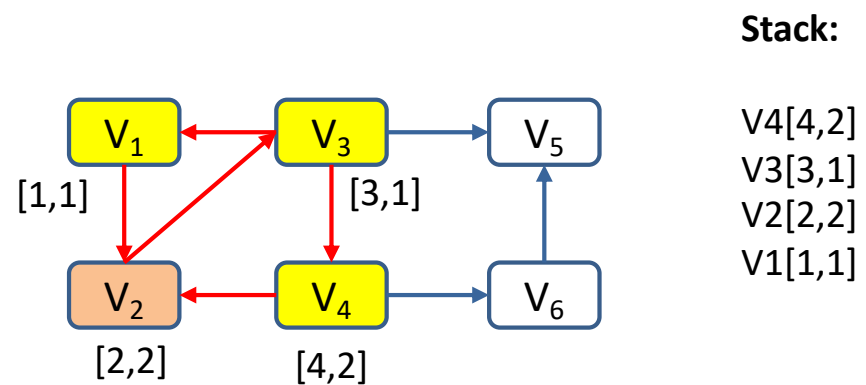
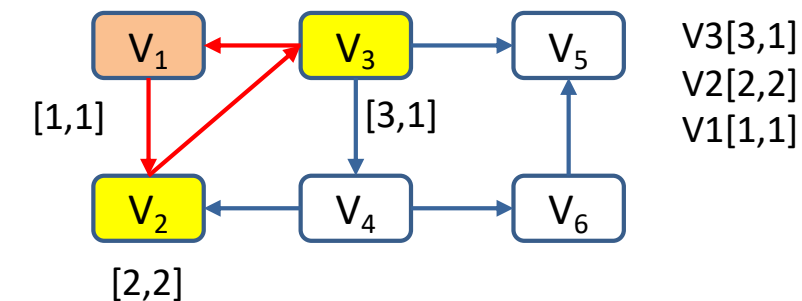
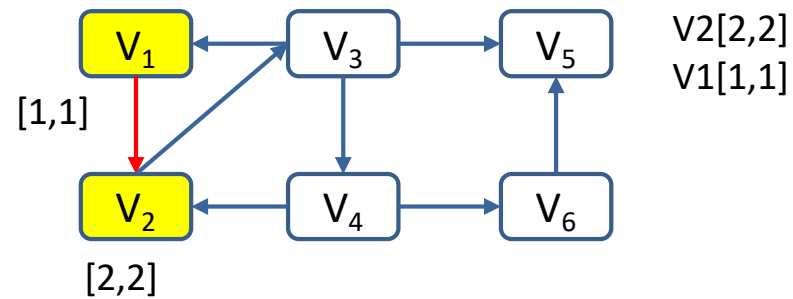
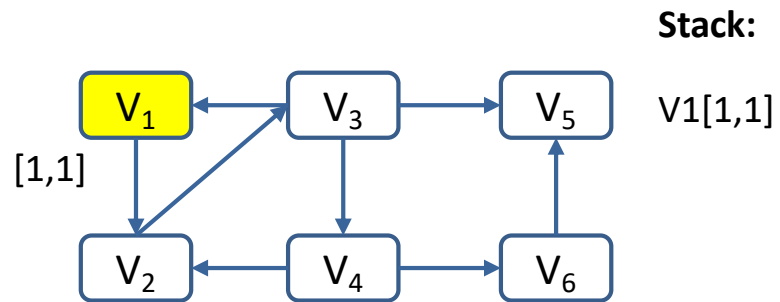
- 转换为有向无环图，按照bottom-up或top-down顺序分析
- 可能有递归调用（强连通分量）=>缩环



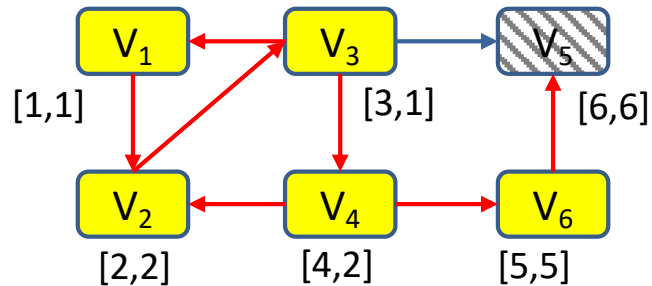
强联通分量检测：Tarjan算法

```
t = 0;
Visit(v) {
    Arrive[v] = t; // 记录每个节点的到达时间
    NextArrive[v] = t; // 记录下一跳的最早到达时间
    t++;
    push v onto the stack;
    for each n in OUT(v) {
        if Arrive[n] == UNDEFINED {
            Visit(n);
            NextArrive[v] = min(NextArrive[v], NextArrive[n]);
        } else if n is on the stack {
            NextArrive[v] = min(NextArrive[v], Arrive[n]);
        }
    }
    if NextArrive[v] == Arrive[v] { // 找到强联通分量
        pop vertices off stack down to v;
    }
}
```

Tarjan算法找SCC：示例



Tarjan算法找SCC：示例

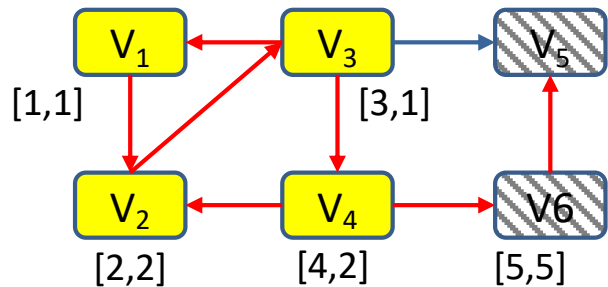


Stack:

V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

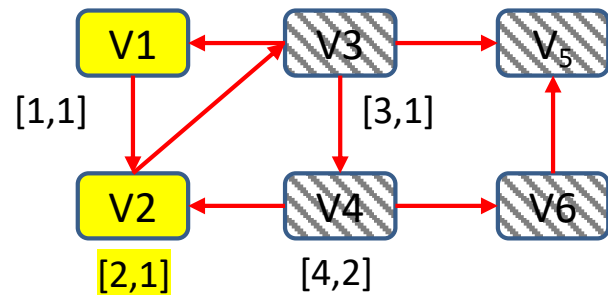
SCC:

{V5}



V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

{V5}
{V6}



更新NextArrive[V2]

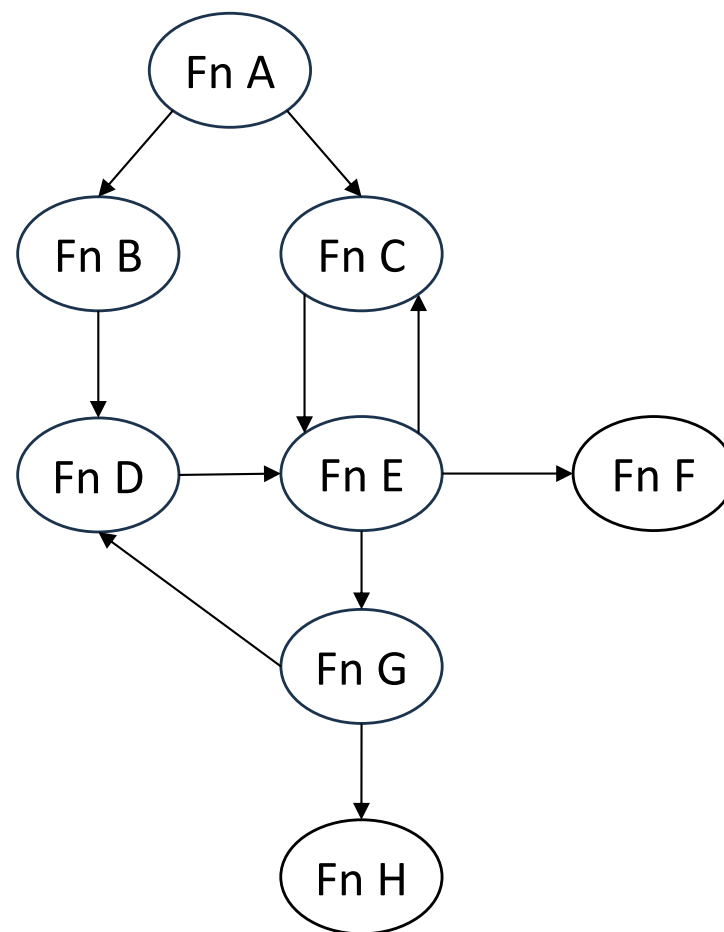
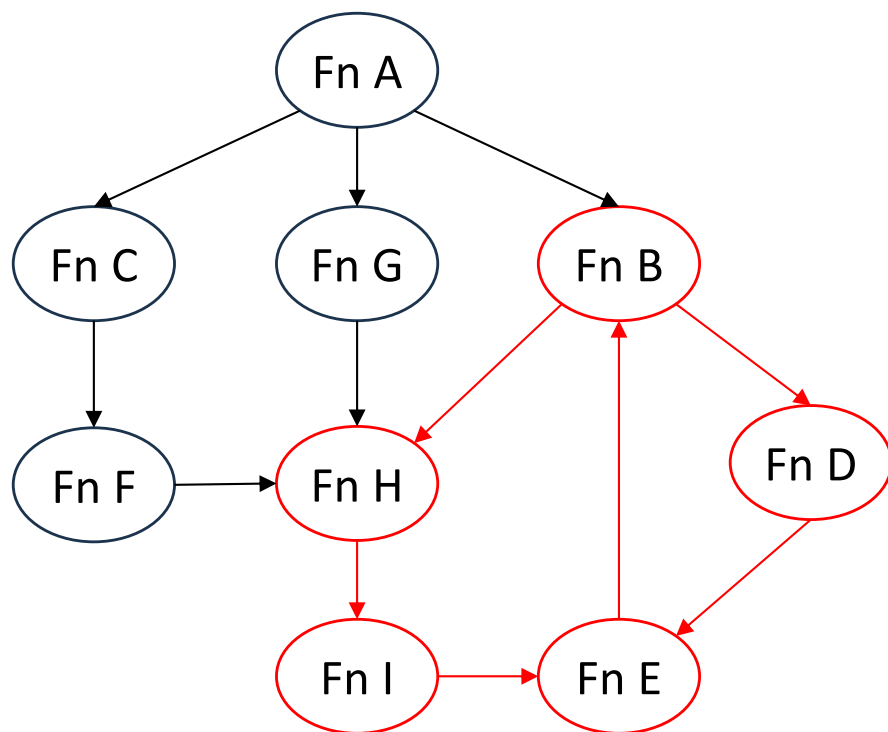
=> min(NextArrive[V3], NextArrive[V2])

V2[2,1]
V1[1,1]

{V5}
{V6}
{4,3,2,1}

练习1：算法实现

- 实现Tarjan算法，检测下列控制流图中的强联通分量



二、尾递归优化

尾递归函数

- return前的最后一条语句调用自己


```
fn fac(n:int, r:int) -> int {  
    if (n < 2) {  
        ret r;  
    }  
    else {  
        ret fac(n-1, n*r);  
    }  
}
```



```
define i32 @fac(i32 %n0, i32 %r0) {  
bb0:  
    %n = alloca i32  
    %r = alloca i32  
    store i32 %n0, i32* %n  
    store i32 %r0, i32* %r  
    br label %bb1  
bb1:  
    %n1 = load i32, i32* %n  
    %t0 = icmp slt i32 %n1, 2  
    br i1 %t0, label %bb2, label %bb3  
bb2:  
    %r1 = load i32, i32* %r  
    ret i32 %r1  
bb3:  
    %n2 = load i32, i32* %n  
    %r2 = load i32, i32* %r  
    %n3 = sub i32 %n2, 1  
    %r3 = mul i32 %n2, %r2  
    %t1 = call i32 @fac(i32 %n3, i32 %r3)  
    ret i32 %t1  
}
```

非尾递归

```
fn fac(n:int) -> int {  
    if (n < 2) {  
        ret 1;  
    }  
    else {  
        ret n * fac(n-1);  
    }  
}
```



编译器可能会自动改写为尾递归形式

```
define i32 @fac(i32 %n0) {  
bb0:  
    %n = alloca i32  
    store i32 %n0, i32* %n  
    br label %bb1  
bb1:  
    %n1 = load i32, i32* %n  
    %t0 = icmp slt i32 %n1, 2  
    br i1 %t0, label %bb2, label %bb3  
bb2:  
    ret i32 1;  
bb3:  
    %n3 = load i32, i32* %n  
    %t1 = sub i32 %n3, 1  
    %t2 = call i32 @fac(i32 %t1)  
    %n4 = load i32, i32* %n  
    %t3 = mul i32 %n4, %t2  
    ret i32 %t3  
}
```

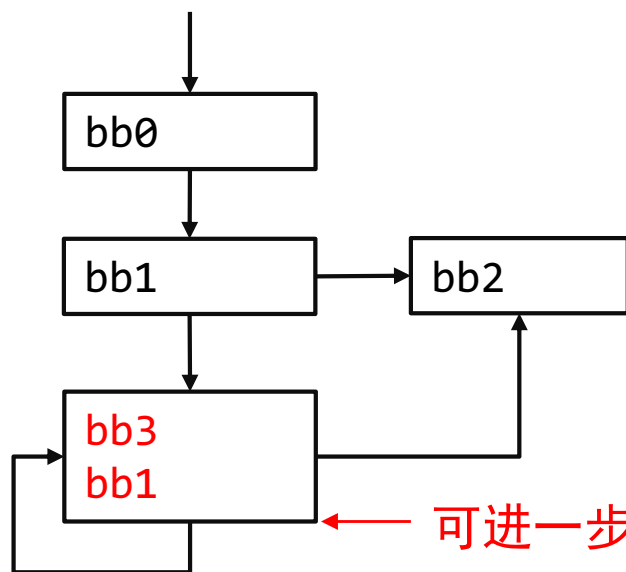
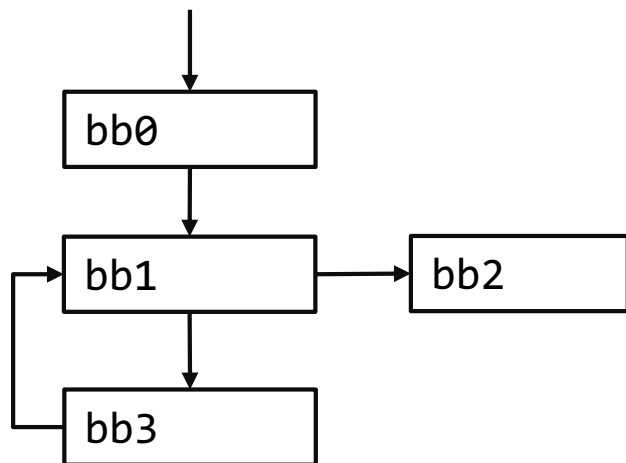
尾递归消除

```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %n0, i32* %n
    store i32 %r0, i32* %r
    br label %bb1
bb1:
    %n1 = load i32, i32* %n
    %t0 = icmp slt i32 %n1, 2
    br i1 %t0, label %bb2, label %bb3
bb2:
    %r1 = load i32, i32* %r
    ret i32 %r1
bb3:
    %n2 = load i32, i32* %n
    %r2 = load i32, i32* %r
    %n3 = sub i32 %n2, 1
    %r3 = mul i32 %n2, %r2
    %t1 = call i32 @fac(i32 %n3, i32 %r3)
    ret i32 %t1
}
```



```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %n0, i32* %n
    store i32 %r0, i32* %r
    br label %bb1
bb1:
    %n1 = load i32, i32* %n
    %t0 = icmp slt i32 %n1, 2
    br i1 %t0, label %bb2, label %bb3
bb2:
    %r1 = load i32, i32* %r
    ret i32 %r1
bb3:
    %n2 = load i32, i32* %n
    %r2 = load i32, i32* %r
    %n3 = sub i32 %n2, 1
    %r3 = mul i32 %n2, %r2
    store i32 %n3, %n
    store i32 %r3, %r
    br %bb1
}
```

尾递归优化



```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %n0, i32* %n
    store i32 %r0, i32* %r
    br label %bb1
bb1:
    %n1 = load i32, i32* %n
    %t0 = icmp slt i32 %n1, 2
    br i1 %t0, label %bb2, label %bb3
bb2:
    %r1 = load i32, i32* %r
    ret i32 %r1
bb3:
    %n2 = load i32, i32* %n
    %r2 = load i32, i32* %r
    %n3 = sub i32 %n2, 1
    %r3 = mul i32 %n2, %r2
    store i32 %n3, %n
    store i32 %r3, %r
    %n4 = load i32, i32* %n
    %t1 = icmp lt i32 %n4, 2;
    br i1 %t1 label %bb2, label %bb3
}
```

优化结果

- 避免了函数调用
- 减少一条load语句

```
define i32 @fac(i32 %n0, i32 %r0) {  
bb0:  
    %n = alloca i32  
    %r = alloca i32  
    store i32 %n0, i32* %n  
    store i32 %r0, i32* %r  
    br label %bb1  
bb1:  
    %n1 = load i32, i32* %n  
    %t0 = icmp slt i32 %n1, 2  
    br i1 %t0, label %bb2, label %bb3  
bb2:  
    %r1 = load i32, i32* %r  
    ret i32 %r1  
bb3:  
    %n2 = load i32, i32* %n  
    %r2 = load i32, i32* %r  
    %n3 = sub i32 %n2, 1  
    %r3 = mul i32 %n2, %r2  
    store i32 %r3, %r  
    %t1 = icmp lt i32 %n3, 2;  
    br i1 %t1 label %bb2, label %bb3  
}
```

尾递归函数：SSA版本

```
define i32 @fac(i32 %n0, i32 %r0) {  
bb0:  
    br label %bb1  
bb1:  
    %t0 = icmp slt i32 %n0, 2  
    br i1 %t0, label %bb2, label %bb3  
bb2:  
    ret i32 %r0  
bb3:  
    %t1 = sub i32 %n0, 1  
    %t2 = mul i32 %n0, %r0  
    %t3 = call i32 @fac(i32 %t1, i32 %t2)  
    ret i32 %t3  
}
```

SSA版本：尾递归消除

```
define i32 @fac(i32 %n0, i32 %r0) {  
bb0:  
    br label %bb1  
bb1:  
    %r1 = phi i32 [%r0, %bb0], [%r2, %bb3]  
    %n1 = phi i32 [%n0, %bb0], [%n2, %bb3]  
    %t0 = icmp slt i32 %n0, 2  
    br i1 %t0, label %bb2, label %bb3  
bb2:  
    ret i32 %r1  
bb3:  
    %n2 = sub i32 %n1, 1  
    %r2 = mul i32 %n1, %r1  
    br label %bb1  
}
```


SSA版本：尾递归优化

```
define i32 @fac(i32 %n0, i32 %r0) {  
bb0:  
    br label %bb1  
bb1:  
    %r1 = phi i32 [%r0, %bb0], [%r2, %bb3]  
    %n1 = phi i32 [%n0, %bb0], [%n2, %bb3]  
    %t0 = icmp slt i32 %n0, 2  
    br i1 %t0, label %bb2, label %bb3  
bb2:  
    %r3 = phi i32 [%r1, %bb1], [%r2, %bb3]  
    ret i32 %r3  
bb3:  
    %n2 = sub i32 %n1, 1  
    %r2 = mul i32 %n1, %r1  
    %t0 = icmp slt i32 %n2, 2  
    br i1 %t0, label %bb2, label %bb3  
}
```

Sibling Call

- Caller和callee函数签名相同并且是tail call
- 内联后可以复用栈帧结构

```
fn foo(a:int) -> int {  
    let b:int = a + 1;  
    ret bar(b);  
}
```

```
fn bar(b:int) -> int {  
    let c:int = b - 1;  
    ret c;  
}
```

```
define i32 @foo(i32 %a0) {  
    %a = alloca i32  
    %b = alloca i32  
    store i32 %a0, i32* %a  
    %a1 = load i32, i32* %a  
    %b0 = add i32 %a1, 1;  
    store i32 %b0, i32* %b  
    %b1 = load i32, i32* %b  
    %t0 = call i32 @bar(%b1);  
    ret i32, %t0  
}
```

```
define i32 @bar(i32 %b0) {  
    %b = alloca i32  
    %c = alloca i32  
    store i32 %b0, i32* %b  
    %b1 = load i32, i32* %b  
    %c0 = sub i32 %b1, 1;  
    store i32 %c0, i32* %c  
    %c1 = load i32, i32* %b  
    ret i32, %c1  
}
```

Sibling Call优化

```
define i32 @foo(i32 %a0) {  
    %a = alloca i32  
    %b = alloca i32  
    store i32 %a0, i32* %a  
    %a1 = load i32, i32* %a  
    %b0 = add i32 %a1, 1;  
    store i32 %b0, i32* %b  
    %b1 = load i32, i32* %b  
    %t0 = call i32 @bar(%b1);  
    ret i32, %t0  
}
```



```
define i32 @bar(i32 %b0) {  
    %b = alloca i32  
    %c = alloca i32  
    store i32 %b0, i32* %b  
    %b1 = load i32, i32* %b  
    %c0 = sub i32 %b1, 1;  
    store i32 %c0, i32* %c  
    %c1 = load i32, i32* %b  
    ret i32, %c1  
}
```



```
define i32 @foo(i32 %a0) {  
    %a = alloca i32  
    %b = alloca i32  
    %c = alloca i32  
    store i32 %a0, i32* %a  
    %a1 = load i32, i32* %a  
    %b0 = add i32 %a1, 1;  
    store i32 %b0, i32* %b  
    %b1 = load i32, i32* %b  
    store i32 %b1, i32* %a  
    %b2 = load i32, i32* %a  
    %c0 = sub i32 %b2, 1;  
    store i32 %c0, i32* %c  
    %c1 = load i32, i32* %b  
    ret i32, %c1  
}
```

练习2：LLVM优化功能测试分析

- 1) 测试LLVM的尾递归优化功能，对比优化前后的效果

```
#: opt -passes=tailcallelim -S in.ll -o out.ll
```

- 2) 测试LLVM的内联优化功能，分析其内联算法的优缺点

```
#: opt -passes=inline -S in.ll -o out.ll
```

- 3) 分析LLVM的O1、O2、O3功能分别集成了哪些优化功能

三、其它优化

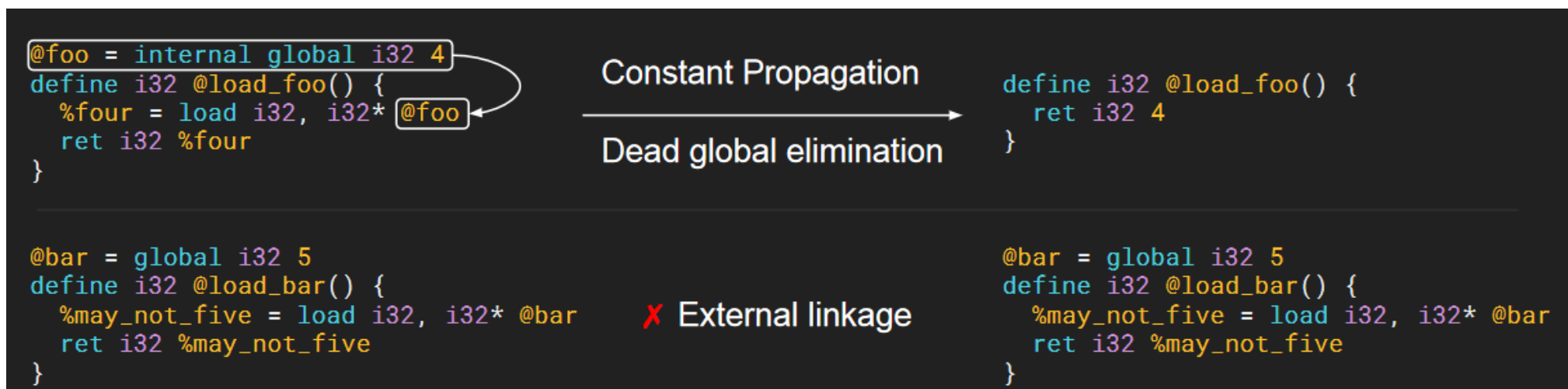
针对仅模块内有效的变量/函数

其它优化思路

- 过程内优化策略在全局变量/常量上的扩展
 - 常量传播
 - 删除冗余变量
- 过程内优化策略在跨函数场景的扩展
 - 合并冗余函数、删除冗余代码
 - 利用函数调用上下文信息优化代码

全局常量传播

- 标记为internal或private的全局常量/变量仅在当前模块有效
- 全局常量分析：如被const标记或没有被store的全局变量



冗余的全局常量/变量

- 全局优化：删除冗余的全局变量
- 常量合并：合并内容相同的常量

```
@A = global i32 0
@D = internal alias i32, i32* @A
@L1 = alias i32, i32* @A
@L2 = internal alias i32, i32* @L1
@L3 = alias i32, i32* @L2
```

```
@A = global i32 0
@L1 = alias i32, i32* @A
@L2 = internal alias i32, i32* @L1
@L3 = alias i32, i32* @L2
```

```
@foo = constant i32 6
@bar = internal unnamed_addr constant i32 6
@baz = constant i32 6

define i32 @use_bar(i32 %arg) {
    %six = load i32, i32* @bar
    %ret = add i32 %arg, %six
    ret i32 %ret
}
```

```
@foo = constant i32 6
@baz = constant i32 6

define i32 @use_bar(i32 %arg) {
    %six = load i32, i32* @foo, align 4
    %ret = add i32 %arg, %six
    ret i32 %ret
}
```


删除无效参数

- 标记为internal的函数仅在当前模块有效，可优化的情况：
 - 参数未在函数体内使用
 - 仅作为另一个函数的无效参数在函数体内使用

```
; Dead arg only used by dead retval
define internal i32 @test(i32 %DEADARG) {
    ret i32 %DEADARG
}

define i32 @test2(i32 %A) {
    %DEAD = call i32 @test(i32 %A) ; 0 uses
    ret i32 123
}
```

```
define internal void @test() {
    ret void ; Argument was eliminated
}

define i32 @test2(i32 %A) {
    call void @test()
    ret i32 123
}
```

函数合并

- 合并功能等价的internal函数；函数形式不必完全相同
- 如何实现快速搜索：函数排序（全序）



```
define internal i64 @foo(i32* %P, i32* %Q) {  
    store i32 4, i32* %P  
    store i32 6, i32* %Q  
    ret i64 0  
}  
  
define internal i64* @bar(i32* %P, i32* %Q) {  
    store i32 4, i32* %P  
    store i32 6, i32* %Q  
    ret i64* null  
}  
  
define i64 @use_foo(i32* %P, i32* %Q) {  
    %ret = call i64 @foo(i32* %P, i32* %Q)  
    ret i64 %ret  
}  
  
define i64* @use_bar(i32* %P, i32* %Q) {  
    %ret = call i64* @bar(i32* %P, i32* %Q)  
    ret i64* %ret  
}
```

```
define internal i64* @bar(i32* %P, i32* %Q) {  
    store i32 4, i32* %P, align 4  
    store i32 6, i32* %Q, align 4  
    ret i64* null  
}  
  
define i64 @use_foo(i32* %P, i32* %Q) {  
    %ret = call i64 bitcast (i64* (i32*, i32*)* @bar to  
i64 (i32*, i32*)*)(i32* %P, i32* %Q)  
    ret i64 %ret  
}  
  
define i64* @use_bar(i32* %P, i32* %Q) {  
    %ret = call i64* @bar(i32* %P, i32* %Q)  
    ret i64* %ret  
}
```

<https://llvm.org/docs/MergeFunctions.html>

参数提升

- 通过修改函数类型优化函数调用

```
> cat test.ll
```

```
%T = type { i32, i32 }
@G = constant %T { i32 17, i32 0 }

define internal i32 @test(%T* %p) {
entry:
  %a.gep = getelementptr %T, %T* %p, i64 0, i32 0
  %a = load i32, i32* %a.gep
  %v = add i32 %a, 1
  ret i32 %v
}

define i32 @caller() {
entry:
  %v = call i32 @test(%T* @G)
  ret i32 %v
}
```

```
> opt -S -argpromotion test.ll
```

```
%T = type { i32, i32 }
@G = constant %T { i32 17, i32 0 }

define internal i32 @test(i32 %p.0.0.val) {
entry:
  %v = add i32 %p.0.0.val, 1
  ret i32 %v
}

define i32 @caller() {
entry:
  %G.idx = getelementptr %T, %T* @G, i64 0, i32 0
  %G.idx.val = load i32, i32* %G.idx
  %v = call i32 @test(i32 %G.idx.val)
  ret i32 %v
}
```

思考

- 应如何排列不同的优化功能执行顺序？
 - 优化效果最佳
 - 执行时间最少