lab3-2实验报告

小组成员：

PB17111644全映菱（队长）

PB17111650陈炳楠

PB17111637张陶竞

一、实验要求

使用opt工具，并阅读源代码来了解LLVM中如何使用Pass对IR进行优化

二、报告内容

1.Constprop

(1)Pass的类型与作用：

①类型

按功能划分为Transform Pass

是Function Pass的子类

②作用

进行常量传播：

实现常量传播与合并，查找只包含常量操作数的指令并将它们替换成一个常量，比如将`add i32 1,2`替换成了`i32 3`

(2)给出一个LLVM IR，进行优化、通过优化后的LLVM IR与源IR的对比说明Pass的优化：

①测试LLVM IR：

```
define i32 @main() {
entry:
    %a = alloca i32
    %b = alloca i32
    %c = alloca i32
    %d = alloca i32
    %e = alloca i32
    store i32 5, i32* %d
    store i32 3, i32* %a
    store i32 2, i32* %b
    %0 = load i32, i32* %a
    %1 = load i32, i32* %b
    %2 = add nsw i32 %0, %1 ;a+b
    %3 = add nsw i32 1,2 ;1+2
    %4 = sub nsw i32 3,4 ;3-4
    %5 = mul nsw i32 5,6 ;5*6
    store i32 %3, i32* %d ;d=1+2
```

```
    store i32 %4, i32* %d ;d=3-4
    store i32 %5, i32* %d ;d=5*6
    %6 = add nsw i32 %3,%4 ;1+2 + 3-4
    %7 = xor i32 123,-1 ;123 xor -1
    store i32 %2, i32* %c ;c=a+b
    ret i32 %7
}
define i1 @test_cmp(){;多层传播
    %A = icmp ule i1 true, false
    %B = icmp uge i1 true, false
    %C = icmp ult i1 false, true
    %D = icmp ugt i1 true, false
    %E = icmp eq i1 false, false
    %F = icmp ne i1 false, true
    %G = and i1 %A, %B
    %H = and i1 %C, %D
    %I = and i1 %E, %F
    %J = and i1 %G, %H
    %K = and i1 %I, %J
    ret i1 %K
}
define i32 @test_divzero(){
    %test_div = sdiv i32 12,0
    ret i32 %test_div
}
define i32 @test_zext() {
        %test_zext = zext i1 true to i32
        ret i32 %test_zext
}
define i1 @test_trunc() {
        %test_trunc = trunc i32 320 to i1
        ret i1 %test_trunc
}
@G_sdiv = global i32 sdiv (i32 0, i32 -1)
declare double @sin(double)
declare double @cos(double)
declare double @tan(double)
declare double @sqrt(double)
declare double @log(double)
declare float @sinf(float)
declare float @cosf(float)
declare float @tanf(float)
declare float @sqrtf(float)
declare float @logf(float)
define double @test_libcall() {
    %result = alloca double
    %resultf = alloca float
    %t_sin = call double @sin(double 0.000000e+00)
    store double %t_sin, double* %result
    %t_cos = call double @cos(double 0.000000e+00)
    store double %t_cos, double* %result
    %t_tan = call double @tan(double 0.000000e+00)
    store double %t_tan, double* %result
    %t_sqrt = call double @sqrt(double 4.000000e+00)
    store double %t_sqrt, double* %result
    %t_log = call double @log(double 3.000000e+00)
    store double %t_log, double* %result
    %t_sinf = call float @sinf(float 3.000000e+00)
```

```
    store float %t_sinf, float* %resultf
    %t_cosf = call float @cosf(float 3.000000e+00)
    store float %t_cosf, float* %resultf
    %t_tanf = call float @tanf(float 3.000000e+00)
    store float %t_tanf, float* %resultf
    %t_sqrtf = call float @sqrtf(float 3.000000e+00)
    store float %t_sqrtf, float* %resultf
    %t_logf = call float @logf(float 3.000000e+00)
    store float %t_logf, float* %resultf
    %1 = load double , double* %result
    ret double %1
}
```

②优化结果：

```
; ModuleID = 'test_constprop.ll'
source_filename = "test_constprop.ll"
@G_sdiv = global i32 0
define i32 @main() {
entry:
    %a = alloca i32
    %b = alloca i32
    %c = alloca i32
    %d = alloca i32
    %e = alloca i32
    store i32 5, i32* %d
    store i32 3, i32* %a
    store i32 2, i32* %b
    %0 = load i32, i32* %a
    %1 = load i32, i32* %b
    %2 = add nsw i32 %0, %1
    store i32 3, i32* %d
    store i32 -1, i32* %d
    store i32 30, i32* %d
    %3 = add nsw i32 3, -1
    store i32 %2, i32* %c
    ret i32 -124
}
define i1 @test_cmp() {
    ret i1 false
}
define i32 @test_divzero() {
    ret i32 undef
}
define i32 @test_zext() {
    ret i32 1
}
define i1 @test_trunc() {
    ret i1 false
}
declare double @sin(double)
declare double @cos(double)
declare double @tan(double)
declare double @sqrt(double)
declare double @log(double)
declare float @sinf(float)
declare float @cosf(float)
```

```
declare float @tanf(float)
declare float @sqrtf(float)
declare float @logf(float)
define double @test_libcall() {
    %result = alloca double
    %resultf = alloca float
    store double 0.000000e+00, double* %result
    store double 1.000000e+00, double* %result
    store double 0.000000e+00, double* %result
    store double 2.000000e+00, double* %result
    store double 0x3FF193EA7AAD030B, double* %result
    store float 0x3FC2103860000000, float* %resultf
    store float 0xBFEFAE04C0000000, float* %resultf
    store float 0xBFC23EF720000000, float* %resultf
    store float 0x3FFBB67AE0000000, float* %resultf
    store float 0x3FF193EA80000000, float* %resultf
    %1 = load double, double* %result
    ret double %1
}
```

③源IR与优化后IR的对比：

i.`main`函数的优化：

```
源IR：
    %3 = add nsw i32 1,2 ;1+2
    %4 = sub nsw i32 3,4 ;3-4
    %5 = mul nsw i32 5,6 ;5*6
    store i32 %3, i32* %d ;d=1+2
    store i32 %4, i32* %d ;d=3-4
    store i32 %5, i32* %d ;d=5*6
    %6 = add nsw i32 %3,%4 ;1+2 + 3-4
    %7 = xor i32 123,-1 ;123 xor -1
    store i32 %2, i32* %c ;c=a+b
    ret i32 %7
优化后IR：
    store i32 3, i32* %d
    store i32 -1, i32* %d
    store i32 30, i32* %d
    %3 = add nsw i32 3, -1
    store i32 %2, i32* %c
    ret i32 -124
```

说明：对于`%3、%4、%5`，它们被赋的值都是两个常量的运算，且在之后会被存储到`%d`里，或参与加法运算`%6 = add nsw i32 %3,%4`；它们满足被使用且可以进行常量折叠，同时可以被操作，则它们的所有使用者对它们的使用都被替换成了常量计算结果，即一个常量；`%7`同理，它会作为main函数的返回值，因此被计算成一个常数，直接返回；而对于`%6`，它的计算结果在后续并没有使用，所以对它的优化只是将它对其他结果为常数的寄存器的使用替换成了常数，没有将这条指令直接计算成常量

ii.`test_cmp`函数的优化：

```
源IR：
define i1 @test_cmp(){;澶氱晫涘犳挒
    %A = icmp ule i1 true, false
    %B = icmp uge i1 true, false
    %C = icmp ult i1 false, true
```

```
    %D = icmp ugt i1 true, false
    %E = icmp eq i1 false, false
    %F = icmp ne i1 false, true
    %G = and i1 %A, %B
    %H = and i1 %C, %D
    %I = and i1 %E, %F
    %J = and i1 %G, %H
    %K = and i1 %I, %J
    ret i1 %K
}
优化后IR:
define i1 @test_cmp() {
    ret i1 false
}
```

说明：这个函数中涉及的数值都是i1类型，`%A、%B、%C、%D、%E、%F`都是对确定布尔值的比较运算，且此后这些计算结果又被用来计算`%G、%H、%I`的值，因此全部被优化成了i1类型的数值；然后`%J`的计算使用了`%G、%H`，则`%G、%H`也被优化掉；最后`%K`使用了`%I、%J`的值，则`%I、%J`也被优化，然后`%K`作为返回值被使用，优化时直接将`%K`的数值计算出来作为返回值，直接返回`ret i1 false`；这个函数就体现了一层一层进行常量传播的过程：首先遍历所有指令，将被使用且能够变成常量的指令计算出来，将对它们的使用替换成对常量的使用；然后对被优化的指令的所有使用者继续遍历，将被使用且能够变成常量的指令计算出来，将对它们的使用替换成对常量的使用，直到所有常量传播完毕。

iii.除零、zext指令、trunc指令常量传播的优化：

```
源IR:
define i32 @test_divzero(){
    %test_div = sdiv i32 12,0
    ret i32 %test_div
}
define i32 @test_zext() {
    %test_zext = zext i1 true to i32
    ret i32 %test_zext
}
define i1 @test_trunc() {
    %test_trunc = trunc i32 320 to i1
    ret i1 %test_trunc
}
优化后IR:
define i32 @test_divzero() {
    ret i32 undef
}
define i32 @test_zext() {
    ret i32 1
}
define i1 @test_trunc() {
    ret i1 false
}
```

说明：除零指令中`%test_sdiv`的结果是12/0，是非法的无定义的，将`%test_sdiv`作为返回值是对它的使用，此处被优化成了`undef`；对`zext`指令同理，将i1类型的`true`转化成i32类型就是`1`，作为返回值被使用，则直接优化成了`ret i32 1`；对`trunc`指令，被使用的`%test_trunc`的结果为i1类型的`false`，优化成`ret i1 false`

iv.全局变量的优化：

```
源IR:
    @G_sdiv = global i32 sdiv (i32 0, i32 -1)
优化后IR:
    @G_sdiv = global i32 0
```

说明：这是对全局变量的初始化数值进行了常量传播，全局变量`G_sdiv`的数值为`sdiv (i32 0,i32 -1)`指令的结果，指令的操作数都是常量，被优化成一个常量

v.库函数调用、double、float类型的优化：

```
源IR:
    %t_sin = call double @sin(double 0.000000e+00)
    store double %t_sin, double* %result
    %t_cos = call double @cos(double 0.000000e+00)
    store double %t_cos, double* %result
    %t_tan = call double @tan(double 0.000000e+00)
    store double %t_tan, double* %result
    %t_sqrt = call double @sqrt(double 4.000000e+00)
    store double %t_sqrt, double* %result
    %t_log = call double @log(double 3.000000e+00)
    store double %t_log, double* %result
    %t_sinf = call float @sinf(float 3.000000e+00)
    store float %t_sinf, float* %resultf
    %t_cosf = call float @cosf(float 3.000000e+00)
    store float %t_cosf, float* %resultf
    %t_tanf = call float @tanf(float 3.000000e+00)
    store float %t_tanf, float* %resultf
    %t_sqrtf = call float @sqrtf(float 3.000000e+00)
    store float %t_sqrtf, float* %resultf
    %t_logf = call float @logf(float 3.000000e+00)
    store float %t_logf, float* %resultf
优化后IR:
    store double 0.000000e+00, double* %result
    store double 1.000000e+00, double* %result
    store double 0.000000e+00, double* %result
    store double 2.000000e+00, double* %result
    store double 0x3FF193EA7AAD030B, double* %result
    store float 0x3FC2103860000000, float* %resultf
    store float 0xBFEFAE04C0000000, float* %resultf
    store float 0xBFC23EF720000000, float* %resultf
    store float 0x3FFBB67AE0000000, float* %resultf
    store float 0x3FF193EA80000000, float* %resultf
```

说明：分别调用`sin、cos、tan、sqrt、log`库函数的float与double形式，传入的参数均为常量，使用所有的调用返回结果，最后被直接优化成了对常量的使用；是由于传入常量参数后这些库函数的返回值（计算结果）也是一个常量，这些返回值被使用了，所以被直接优化成了常量的使用

(3)Pass的流程：

①首先，如果这个函数不需要优化，跳过这个函数

②对需要优化的函数，遍历函数中的每一条指令，将指令插入工作列表与工作列表向量

③遍历工作列表向量，将遍历到的指令从工作列表中擦除，然后处理这条指令：

i.如果这条指令会被使用，同时它可以成功地进行常数折叠（它成为了一个常数）且应该被操作，就将它的所有使用者加入到工作列表与“新的工作列表”向量中，否则查看下一条指令

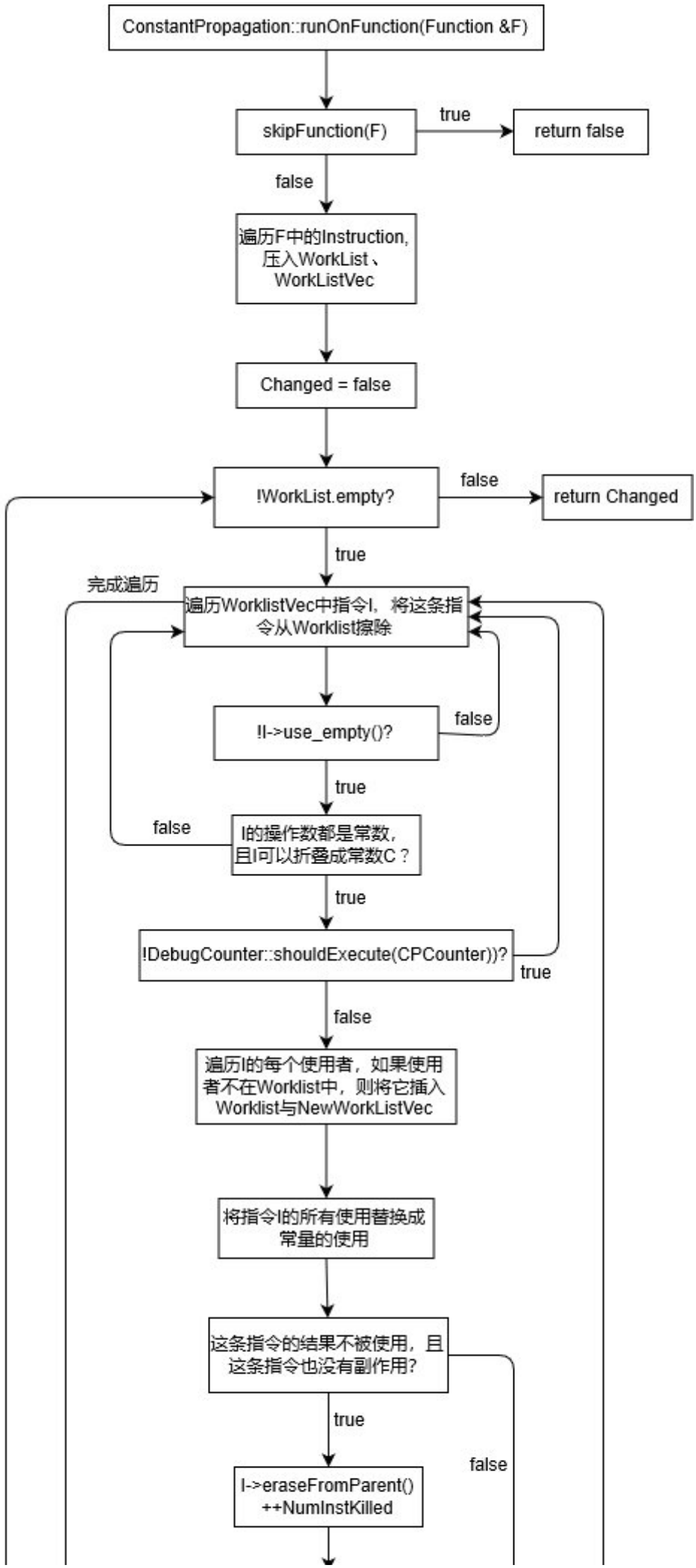
ii.将所有对这条指令的使用替换成对常数C的使用

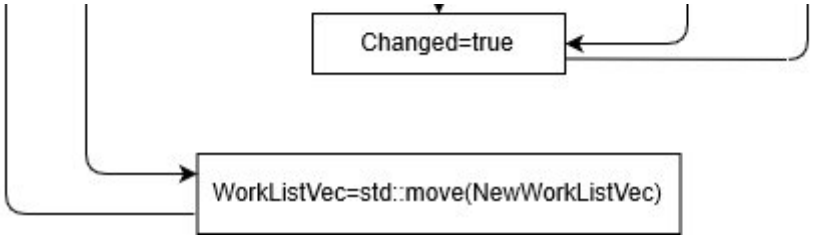
iii.替换完毕后，如果这条指令不再被使用且没有副作用，将这条指令从包含它的基本块中删除，同时指令条数的计数器+1

iv.标记这个函数发生了优化（如果这条指令无法进行常量折叠，就不会将优化发生标志置为真）

④将工作列表向量变成“新的工作列表”向量，再次遍历工作列表向量重复操作，直到工作列表为空，结束优化

Constprop的流程图：


ConstantPropagation::runOnFunction(Function &F)
skipFunction(F)
true
return false
false
遍历F中的Instruction，压入WorkList、WorkListVec
Changed = false
!WorkList.empty?
false
return Changed
true
完成遍历
遍历WorklistVec中指令I，将这条指令从Worklist擦除
!I->use_empty()?
false
true
false
I的操作数都是常数，且I可以折叠成常数C？
true
!DebugCounter::shouldExecute(CPCounter))?
true
false
遍历I的每个使用者，如果使用者不在Worklist中，则将它插入Worklist与NewWorkListVec
将指令I的所有使用替换成常量的使用
这条指令的结果不被使用，且这条指令也没有副作用?
true
false
I->eraseFromParent()
++NumInstKilled



2.ADCE

(1)Pass的类型与作用：

①类型

按功能划分为Transform Pass

是Function Pass的子类

②作用

进行激进的死代码删除

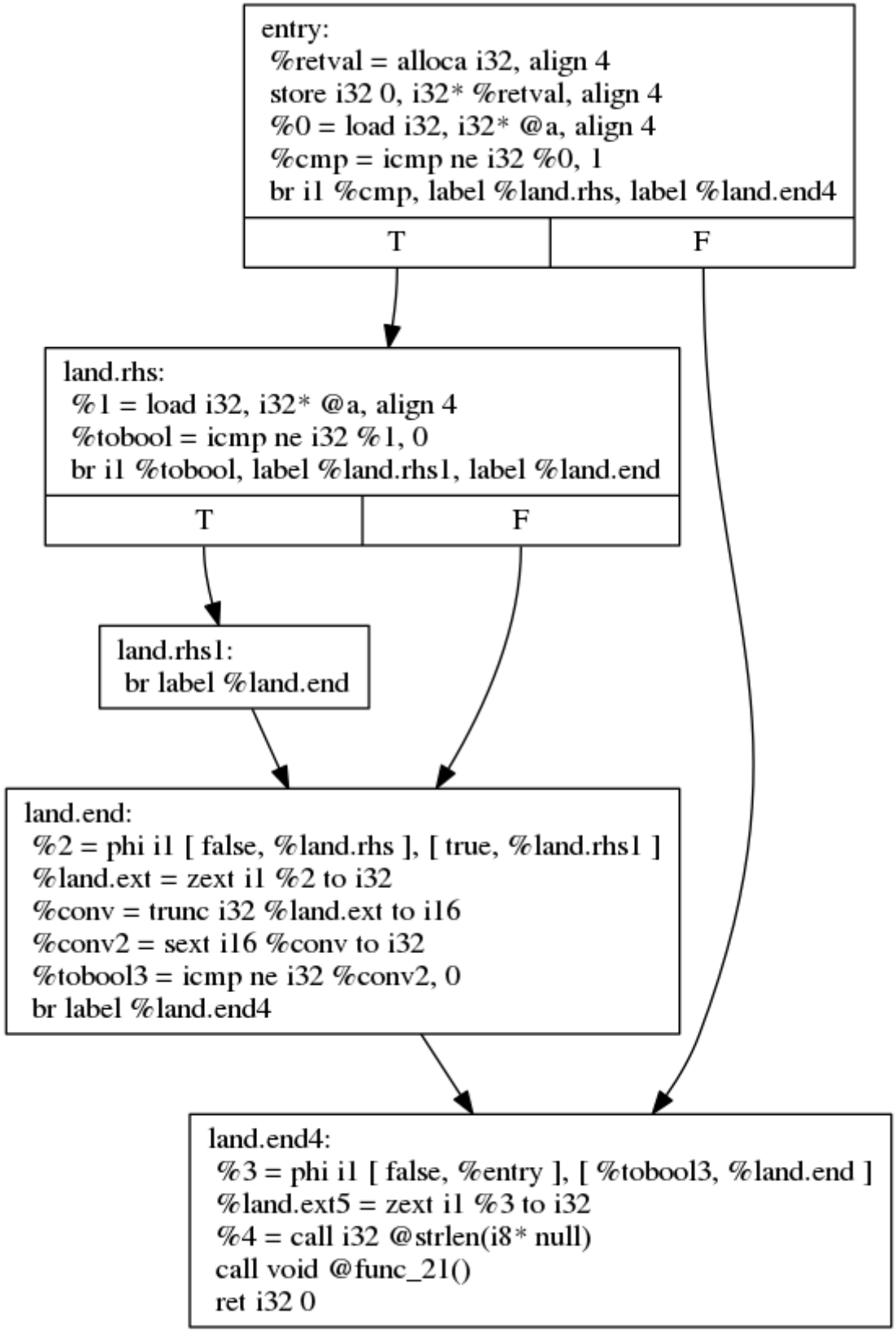
(2)给出一个LLVM IR，进行优化、通过优化后的LLVM IR与源IR的对比说明Pass的优化：

①测试LLVM IR：

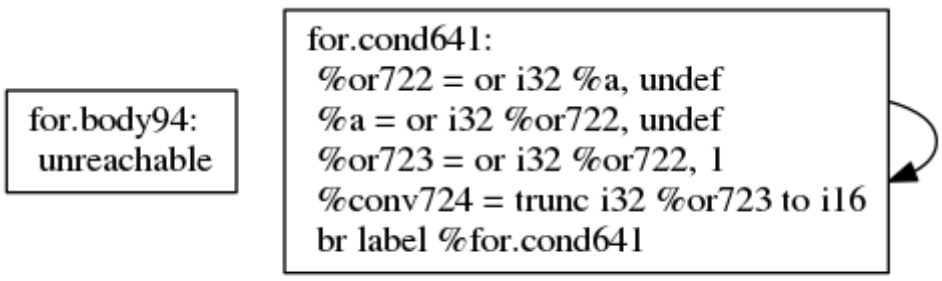
```
@a = common global i32 0, align 4
declare i32 @strlen(i8*) readonly nounwind
define void @func_21() {
for.body94:
    unreachable
for.cond641:
    %or722 = or i32 %a, undef
    %a = or i32 %or722, undef
    %or723 = or i32 %or722, 1
    %conv724 = trunc i32 %or723 to i16
    br label %for.cond641
}
define void @simple(){
    %a = alloca i32
    %add = add nsw i32 1,2
    ret void
}
define i32 @main() {
entry:
    %retval = alloca i32, align 4
    store i32 0, i32* %retval, align 4
    %0 = load i32, i32* @a, align 4
    %cmp = icmp ne i32 %0, 1
    br i1 %cmp, label %land.rhs, label %land.end4
land.rhs:                                         ; preds = %entry
    %1 = load i32, i32* @a, align 4
    %tobool = icmp ne i32 %1, 0
    br i1 %tobool, label %land.rhs1, label %land.end
land.rhs1:                                        ; preds = %land.rhs
    br label %land.end
land.end:                                         ; preds = %land.rhs1,
%land.rhs
```

```
    %2 = phi i1 [ false, %land.rhs ], [ true, %land.rhs1 ]
    %land.ext = zext i1 %2 to i32
    %conv = trunc i32 %land.ext to i16
    %conv2 = sext i16 %conv to i32
    %tobool3 = icmp ne i32 %conv2, 0
    br label %land.end4
land.end4:                                        ; preds = %land.end, %entry
    %3 = phi i1 [ false, %entry ], [ %tobool3, %land.end ]
    %land.ext5 = zext i1 %3 to i32
    call i32 @strlen( i8* null )
    call void @func_21()
    ret i32 0
}
```

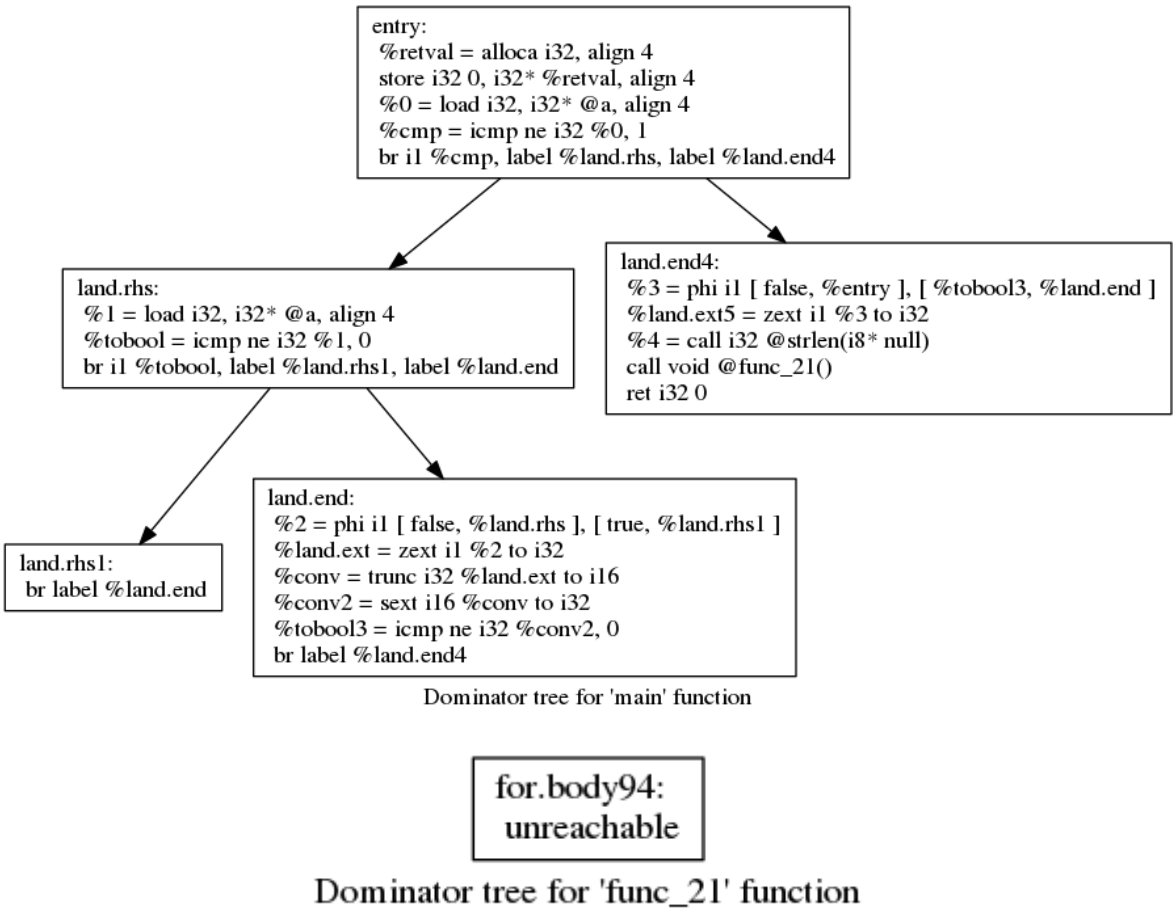
函数`main`、`func_21`优化前的CFG与支配树：



CFG for 'main' function



CFG for 'func_21' function

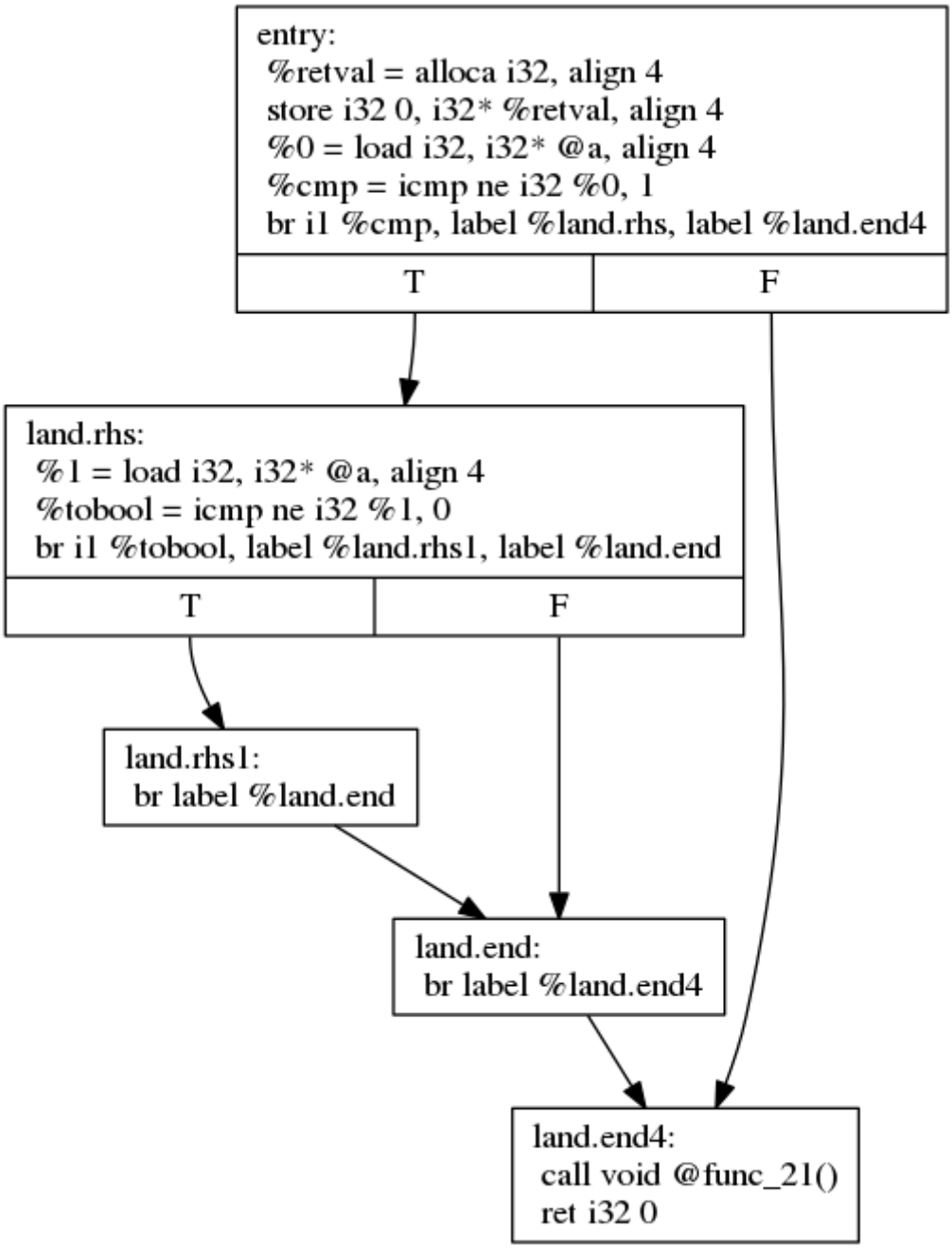


②优化结果：

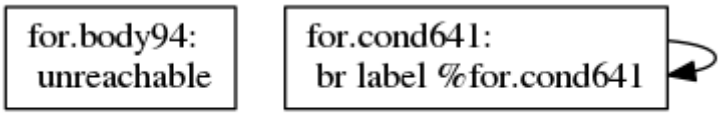
```
; ModuleID = 'combine.ll'
@a = common global i32 0, align 4
; Function Attrs: nounwind readonly
declare i32 @strlen(i8*) #0
define void @func_21() {
for.body94:
    unreachable
for.cond641:                                      ; preds = %for.cond641
    br label %for.cond641
}
define void @simple() {
  ret void
}
define i32 @main() {
entry:
    %retval = alloca i32, align 4
    store i32 0, i32* %retval, align 4
    %0 = load i32, i32* @a, align 4
    %cmp = icmp ne i32 %0, 1
    br i1 %cmp, label %land.rhs, label %land.end4
land.rhs:                                         ; preds = %entry
    %1 = load i32, i32* @a, align 4
    %tobool = icmp ne i32 %1, 0
    br i1 %tobool, label %land.rhs1, label %land.end
land.rhs1:                                        ; preds = %land.rhs
    br label %land.end
land.end:                                         ; preds = %land.rhs1,
%land.rhs
    br label %land.end4
```

```
land.end4:                                        ; preds = %land.end, %entry
    call void @func_21()
    ret i32 0
}
attributes #0 = { nounwind readonly }
```

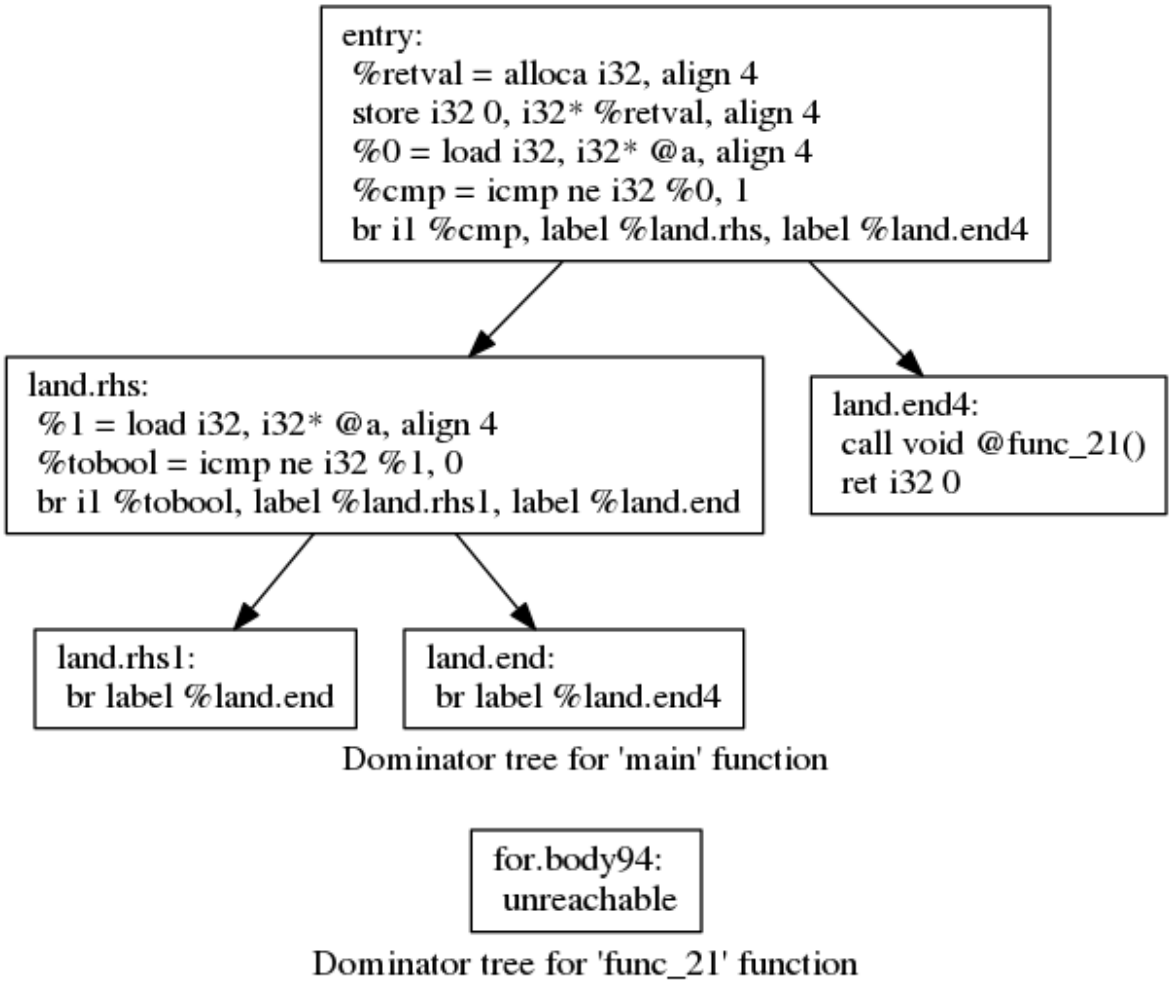
函数`main`、`func_21`优化后的CFG和支配树：



CFG for 'main' function



CFG for 'func_21' function



③源IR与优化后IR的对比：

i.

```
源IR：
for.cond641:
    %or722 = or i32 %a, undef
    %a = or i32 %or722, undef
    %or723 = or i32 %or722, 1
    %conv724 = trunc i32 %or723 to i16
    br label %for.cond641
优化后IR：
for.cond641:                                      ; preds = %for.cond641
    br label %for.cond641
```

说明：

对整个module调用initialize()，对第一个函数func_21的每一条指令调用isAlwayslive()，对符合条件的指令，调用marklive()（marklive()里会将活跃指令压入Worklist）；由于for.cond641是函数func_21的entryblock,并且该entryblock的terminator为无条件跳转，调用marklive()，并获取该指令的debug location，对该debug location调用collectliveScope(DILocation) 对该无条件跳转所在BB调用marklive(BBInfo)。在以上两个操作后Worklist里保存该BB的要被标活的指令。
之后调用markLiveInstruction（），将Worklist里的每一项标记为活跃。
在该BB中只有无条件跳转`br label %for.cond641`为活跃，调用removeDeadInstructions()。

ii.

```
源IR：
```

```
land.end:                                         ; preds = %land.rhs1,
%land.rhs
    %2 = phi i1 [ false, %land.rhs ], [ true, %land.rhs1 ]
    %land.ext = zext i1 %2 to i32
    %conv = trunc i32 %land.ext to i16
    %conv2 = sext i16 %conv to i32
    %tobool3 = icmp ne i32 %conv2, 0
    br label %land.end4
优化后IR:
land.end:                                         ; preds = %land.rhs1,
%land.rhs
    br label %land.end4

源IR:
land.end4:                                        ; preds = %land.end, %entry
    %3 = phi i1 [ false, %entry ], [ %tobool3, %land.end ]
    %land.ext5 = zext i1 %3 to i32
    call i32 @strlen( i8* null )
    call void @func_21()
    ret i32 0
优化后IR:
land.end4:                                        ; preds = %land.end, %entry
    call void @func_21()
    ret i32 0
```

说明：

遍历main函数的每一条指令，调用isAlwayslive()，对符合条件的指令，调用marklive()；之后对函数的后序支配树，遍历根节点（可以理解为这个函数的Block)的所有子节点，再对每一个子节点深度优先遍历，这些子节点都有对应的BB,拿到每一个BB的terminator调用marklive（）。

iii.

```
源IR:
define void @simple(){
    %a = alloca i32
    %add = add nsw i32 1,2
    ret void
}
优化后IR:
define void @simple() {
    ret void
}
```

说明：

对于`%a=alloca i32`指令，这个分配的空间在后续并没有使用；`%add=add nsw i32 1,2`的结果在后续也没有使用；所以经过ADCE优化之后，它们都不会被标记成活跃，优化后只剩下`ret void`。

iv.对`main`函数的分析：

```
entry:
    %retval = alloca i32, align 4
    store i32 0, i32* %retval, align 4
    %0 = load i32, i32* @a, align 4
    %cmp = icmp ne i32 %0, 1
    br i1 %cmp, label %land.rhs, label %land.end4
```

经过分析,在main函数entry Block中，第二条store指令对内存进行修改，满足可能会有副作用的条件，标记为活跃，因为store指令中使用了寄存器%retval的值，所以第一条指令也标记活跃。entry Block的Terminator为有条件跳转，标记为活跃，该Terminator使用了寄存器%cmp的值所以第四条指令也标记为活跃，第四条指令使用了寄存器%0的值，所以第三条指令也为活跃。综上，entry Block中所有指令为活跃，ADCE没有进行删除。

```
land.rhs:                                         ; preds = %entry
  %1 = load i32, i32* @a, align 4
  %tobool = icmp ne i32 %1, 0
  br i1 %tobool, label %land.rhs1, label %land.end
```

经过分析,在main函数land.rhs Block中，该Block的Terminator为有条件跳转，标记为活跃，该Terminator使用了寄存器%tobool的值所以第二条指令也标记为活跃，第二条指令使用了寄存器%1的值，所以第一条指令也为活跃。综上，land.rhs Block中所有指令为活跃，ADCE没有进行删除。

```
land.rhs1:                                        ; preds = %land.rhs
  br label %land.end
```

在land.rhs1 Block中只有一条无条件跳转指令，标记为活跃，ADCE没有进行删除。

```
源IR:
land.end:                                         ; preds = %land.rhs1,
%land.rhs
    %2 = phi i1 [ false, %land.rhs ], [ true, %land.rhs1 ]
    %land.ext = zext i1 %2 to i32
    %conv = trunc i32 %land.ext to i16
    %conv2 = sext i16 %conv to i32
    %tobool3 = icmp ne i32 %conv2, 0
    br label %land.end4
优化后IR:
land.end:                                         ; preds = %land.rhs1,
%land.rhs
    br label %land.end4
```

在land.end Block中,最后一条无条件跳转指令标记为活跃，其它指令没有副作用且不在其它Block中使用，不改变活跃性，仍为死指令，全部删除。

```
源IR:
land.end4:                                        ; preds = %land.end, %entry
    %3 = phi i1 [ false, %entry ], [ %tobool3, %land.end ]
    %land.ext5 = zext i1 %3 to i32
    call i32 @strlen( i8* null )
    call void @func_21()
    ret i32 0
优化后IR:
land.end4:                                        ; preds = %land.end, %entry
    call void @func_21()
    ret i32 0
```

在land.end Block中,最后一条ret指令标记为活跃，`call void @func_21()`,调用了func_21函数,func_21中有活跃指令（如无条件跳转），所以标记为活跃。`call i32 @strlen( i8* null )`调用了strlen函数，却没有使用它的返回值，因此ADCE认为对strlen函数的调用是无用的，不改变该指令的活跃性，仍为死指令。其它指令没有副作用且不在其它Block中使用，不改变活跃性，仍为死指令。删除land.end Block的所有死指令。

v.对`func_21`的分析：

```
源IR：
for.cond641:
    %or722 = or i32 %a, undef
    %a = or i32 %or722, undef
    %or723 = or i32 %or722, 1
    %conv724 = trunc i32 %or723 to i16
    br label %for.cond641
优化后IR：
for.cond641:                                      ; preds = %for.cond641
    br label %for.cond641
```

(3)Pass的流程：

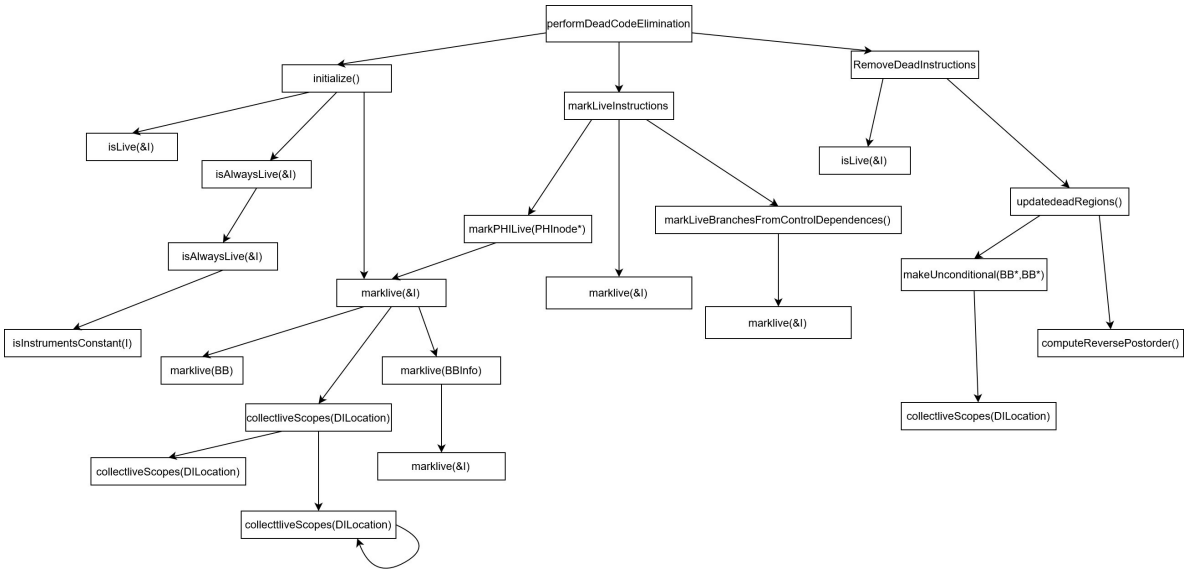
①ADCE pass 调用函数performDeadCodeElimination来实现激进的死代码删除，它调用了3个子函数

②在子函数initialize()中，首先对函数的每一条指令调用isAlwayslive()，对符合条件的指令，调用marklive()。之后深度优先遍历函数，通过getEntryBlock()获得Block，若该Block的后继有回边，调用markLive()标记Block的terminator为活跃。之后对函数的后序支配树，遍历根节点（可以理解为这个函数的Block)的所有子节点，再对每一个子节点深度优先遍历，这些子节点都有对应的BB,拿到每一个BB的terminator调用marklive（）；最后对于函数的entryBlock，若它的terminator为无条件跳转，则对terminator调用markLive()。

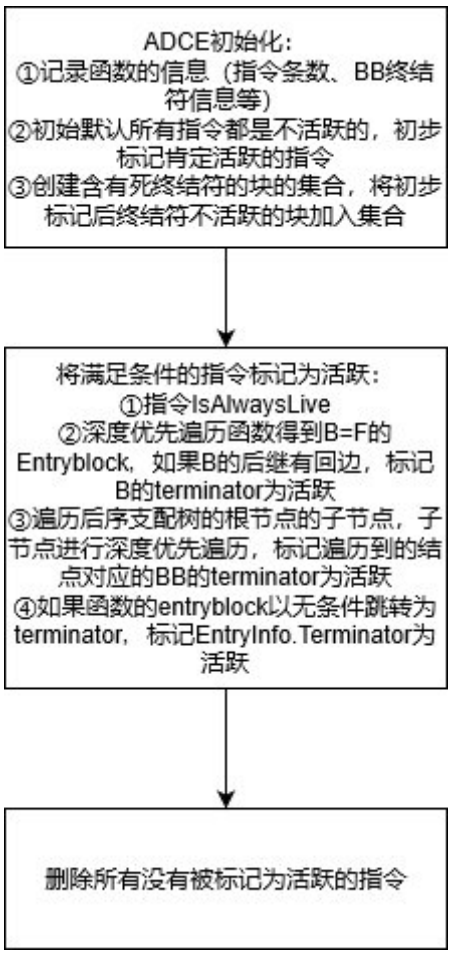
③在子函数markLiveInstructions()中，当Worklist非空时取出每一条活跃指令LiveInst，取出LiveInst每一个operands,如果operands是Instruction*类型，对它调用markLive();之后，如果LiveInst是PHINode类型，对它调用markPhiLive()，最后调用markLiveBranchesFromControlDependences()

④在子函数RemoveDeadInstructions()中，先调用updateDeadRegions(),对函数的每一条指令，先判断指令是否活跃，若活跃则继续找下一条指令，若指令不活跃，继续判断scope of this variable location 是否活跃，若条件AliveScopes.count(DII->getDebugLoc()->getScope())为true,则继续寻找下一条指令。否则，将该指令加入到Worklist（准备删除的指令）中，调用dropAllReferences()丢弃对操作数的所有引用并且将所有use counts归零。遍历Worklist，将Worklist里的指令从Block里移除（unlink&delete)，并且增加NumRemoved（记录该函数移除的指令条数）。

I.ADCE的函数调用关系：



II.ADCE流程图：



III.对ADCE.CPP中函数的分析：

0.

ADCE乐观地假设所有指令都是无效地，直到得到另外的证明，从而使其可以消除其他DCE过程无法捕获的无效计算，尤其是涉及循环计算的情况

1.void AggressiveDeadCodeElimination::initialize()

(1)功能：

对激进的死代码删除PASS进行初始化

(2)具体实现：

首先获取函数的size，保存在BB的信息里；然后对函数中的每一个BB，统计指令条数来调整InstInfo哈希表的大小，并记录BB的终结符是否为无条件分支。保存指令条数，初始化指令信息map并设置块信息的指针，记录每个BB的终结符活跃信息。

收集已知活跃的"根指令"集合：遍历函数中的指令，如果指令I总是活跃的（isAlwaysLive(I)为true），标记指令I为活跃（markLive(I)），如果RemoveControlFlowFlag为0，不能修改控制流，则直接返回；否则，如果RemoveLoops为0，不能删除循环，就保存深度优先迭代器的信息，不仅要记录哪个结点被访问过，还要记录某个结点是否在当前结点的活跃祖先的堆栈上。状态也需要记录函数的size，然后在前序深度优先块上迭代，将已被视为循环回边的块的所有边处理成循环回边，如果某一个分支有回边，将它标记为活跃，即查看每一个深度优先遍历块的终结符term，如果终结符不是活跃的，找到一个BB的后继，这个后继在当前结点的活跃祖先的堆栈上，则将这个终结符标记成活跃（markLive(Term)），如果找不到，这个终结符就无法标记成活跃

如果没有从块到函数的返回的路径，则将该块标记为活跃，通过查看支配树根的哪一个孩子退出程序来实现，且对于其他所有子树，标记子树为活跃，即对后续支配树的根结点，遍历它所有的子结点，如果这个孩子的BB的终结符不是return（是其他的比如无限循环），对这个孩子进行深度优先遍历，标记深度优先结点的BB的终结符是活跃的

将函数的入口BB设置为活跃的，如果函数的入口BB是无条件分支，标记入口BB的终结符是活跃的。

建立含有死终结符的块的初始集合：如果这个块的终结符不是活跃的，将这个集合加入有死终结符的BB集合里。

2.bool AggressiveDeadCodeElimination::isAlwaysLive(Instruction &I)

(1)功能：

一直被认为是活跃的操作，返回True；反之，返回False。用来判断活跃性。

(2)具体实现：

①False：

-1.Ins是EH-Block的一个变量/Ins可能有副作用+Ins是常量

-2.Ins不是终结符

-3.RemoveControlFlowFlag为真+Ins是分支Ins / Ins是switch ins

②True:

-1.Ins是EH-Block的一个变量/Ins可能有副作用+Ins不是常量

-2.Ins是终结符

-3.RemoveControlFlowFlag为假+Ins不是是switch Ins / Ins不是分支Ins+Ins不是switch

3.bool AggressiveDeadCodeElimination::isInstrumentsConstant(Instruction &I)

(1)功能：

判断指令是否是运行时调用+是否指令是常数(官方解释：Return value for instrumentation instructions for value profiling).

(2)具体实现：

①True:

-1.如果指令I是CallInst类型可以成功转换+指令CI(I做了类型转换之后的)的调用函数非空(不是间接调用)+CI->getArgOperand(0)是Constant类型

②False:

不满足True即为false

4.void AggressiveDeadCodeElimination::markLiveInstructions()

(1)功能：

将活跃性传播到能到达的定义。

(2)具体实现：

当Worklist不为空时，不断从Worklist中拿出活跃指令LiveInst，对于LiveInst
的所有操作数，对它们作类型转换，对它们调用markLive；之后对LiveInst判断类型是否为
PHINode，如果满足条件，则对PN(类型转换后的LiveInst)调用markPhiLive。
最后调用markLiveBranchesFromControlDependences()

5.void AggressiveDeadCodeElimination::markLive(Instruction *I)

(1)功能：

标记指令为活跃的。

(2)具体实现：

如果I(拿到Ins的Info信息)已经标记为活跃，直接返回；否则标记它为活跃，并且将它压入WorkList

之后拿到I->getDebugLoc(),设为DL,对DL调用collectLiveScopes
之后拿到Info信息的BBInfo(指令对应的BB)。如果BBInfo的终结符是Ins I
将该BBInfo从BlocksWithDeadTerminators集合中删去，之后如果BBInfo不是无条件
跳转指令，那么遍历I的parent的后继，对它们调用markLive。最后对BBInfo调用
markLive。

6.void AggressiveDeadCodeElimination::markLive(BlockInfoType &BBInfo)

(1)功能：

标记一个块为活跃的。

(2)具体实现：

当这个block有一个活指令时，标记它的Live信息为true。当BBInfo的
CFLive为false时，标记它的该信息为true，并将BBInfo.BB插入到NewLiveBlocks(是
一系列控制依赖活跃的块并且不能独立分析)中。如果BBInfo.UnconditionalBranch，
则对BBInfo的终结符调用markLive。

7.void AggressiveDeadCodeElimination::collectLiveScopes(const DILocalScope &LS)

(1)功能：

记录周围有活跃debug信息的debug scopes。

(2)具体实现：

如果AliveScopes.insert(&LS).second为false，直接返回；
如果LS是DISubprogram类型，也直接返回。之后对LS的scope调用collectLiveScopes。

8.void AggressiveDeadCodeElimination::collectLiveScopes(const DILocation &DL)

(1)功能：

记录周围有活跃debug信息的debug scopes。

(2)具体实现：

如果AliveScopes.insert(&DL).second不成立，则直接返回；
之后对DL->getScope()调用collectLiveScopes来从scope链中收集活的
scope。之后在inlined-at链中尾递归并对其中的每一项调用collectLiveScopes。

9.void AggressiveDeadCodeElimination::markPhiLive(PHINode *PN)

(1)功能：

将PHI node的控制前驱的终结符标记为活跃。

(2) 具体实现：

首先拿到PN的parent的Block信息，如果该信息显示已经有活跃的PHINode
结点，则直接返回。否则标记Info的HasLivePhiNodes属性为true。之后
遍历Info.BB的所有前驱，如果前驱的CFLive属性表示为false，则将CFLive
属性标记为true，并且将该前驱插入到NewLiveBlocks(一系列我们确定控制的块，它们的依赖来源是活的并且没有做依赖性分析)的集合中

10.void AggressiveDeadCodeElimination::markLiveBranchesFromControlDependences()

(1)功能：

分析无效分支，找到那些分支是影响活动块的依赖源，这些分支被标记为活跃的。

(2)具体实现：

如果有不活跃terminator的blocks的集合为空，则直接返回；
否则先定义IDFBlocks(反向控制图中活动块X的优势边界是X依赖于控制的一组块,以下序列计算当前具有死终止符的块集，这些终止符是NewLiveBlocks中块的控制依赖源)等相关一系列操作。
之后对于IDFBlocks中的每一个块，对该块的terminator都调用它们的
markLive。

11.bool AggressiveDeadCodeElimination::removeDeadInstructions()

(1)功能：

移除没有标志活跃的指令，返回值指令是否被移除

(2)具体实现：

先调用updateDeadRegions()实现更新DeadRegions(有共同活跃的post-dominator的dead block集合)里的dead blocks周围的控制和数据流。

for (Instruction &I : instructions(F)) 循环通过遍历函数里的指令找出所有可安全删除的指令(没有副作用且不影响控制流或返回函数值的指令)加入Worklist。
实现：for循环里先判断指令是否活跃，若活跃则继续找下一条指令，若指令不活跃，若指令不活跃，继续判断scope of this variable location 是否活跃，若条件AliveScopes.count(DII->getDebugLoc()->getScope())为true,则继续寻找下一条指令。否则，将该指令加入到Worklist（准备删除的指令）中，调用dropAllReferences()丢弃对操作数的所有引用并且将所有use counts归零。遍历Worklist，将Worklist里的指令从Block里移除（unlink&delete），并且增加NumRemoved（记录该函数移除的指令条数）。

返回值：
Worklist为空，说明没有进行指令的删除，函数返回0
Worklist不为空，说明没有进行指令的删除，函数返回1

12.void AggressiveDeadCodeElimination::updateDeadRegions()

(1)功能：

识别CFG中有不活跃Terminator的部分，并修改CFG。

(2)具体实现：

DeadRegions——有共同活跃的 前驱支配结点(post-dominator) 的dead block集合

for (auto *BB : BlocksWithDeadTerminators)遍历函数的Blocks,确定Blocks是否有DeadTerminators
若Block的Terminator是UnconditionalBranch，则设置该Block的Terminator为活跃，继续遍历下一个Block
若找到Block的Terminator不是UnconditionalBranch，计算ReversePostOrder（算过就不用算了），在该Block结尾处加UnconditionalBranch，跳转到该Block后继中最接近函数结尾的那个Block；定义最接近函数结尾的Block——在ReversePostOrder图中post-order序号最大的，设其为PreferredSucc。

for (auto *Succ : successors(BB))
遍历后继BB，删除除了PreferredSucc的后继Block
并将其加入RemovedSuccessors

makeUnconditional(BB, PreferredSucc->BB);给找到的BB加UnconditionalBranch。

for (auto *Succ : RemovedSuccessors)
遍历RemovedSuccessors，将CFG中从BB指向RemovedSuccessors的边加入SmallVector<DominatorTree::UpdateType, 4> DeletedEdges;DeletedEdges为即将被删除的边。

调用DomTreeUpdater，参数为DeletedEdges，更新DomTree

NumBranchesRemoved：被移除的分支数加1

13.void AggressiveDeadCodeElimination::computeReversePostOrder()

(1)功能：

基于反向CFG后序编号设置BlockInfo的PostOrder

(2)具体实现：

for (auto &BB : F)遍历BB
找没有后继的BB,它的PostOrder为0，从它开始DFS（extend the DFS from the block backward through the graph），依DFS遍历顺序，BB的PostOrder递增。

14. void AggressiveDeadCodeElimination::makeUnconditional(BasicBlock *BB,BasicBlock *Target)

(1)功能：

设置BB的Terminator为无条件跳转到Target

(2)具体实现

如果BB的Terminator是UnconditionalBranch，则标记BB的Terminator为活跃。
如果BB的Terminator不是UnconditionalBranch，NumBranchesRemoved加1，BB结尾增加无条件跳转到Target的分支，将BB的Terminator标记为活跃。

15.PreservedAnalyses ADCEPass::run(Function &F, FunctionAnalysisManager &FAM)

(1)功能：

用支配树来更新分析

(2)具体实现：

如果执行激进的死代码删除没有删除指令，就保存所有的分析；否则此时激进的死代码删除执行过，就保存控制流图的分析、支配树分析与后序支配树分析

16.bool runOnFunction(Function &F) override

(1)功能：

执行在函数上的激进的死代码删除

(2)具体实现：

如果"跳过这个函数"为真，返回`false`；否则，计算支配树与后序支配树，将函数、支配树与后序支配树作为激进死代码删除的参数，进行激进的死代码删除

iv.ADCE与DCE的简单比较：

1.DCE流程：

首先初始化DCE pass，调用runOnFunction（），在该函数内调用eliminateDeadCode（），用来删除函数中的死代码，在eliminateDeadCode（）又调用了DCEInstruction（）用来删除单条的死指令。在DCEInstruction（）里判断指令是否活跃，指令没有被使用并且没有副作用，将它的所有操作数压入Worklist，之后将该指令从包含它的Basic Block中分离并且删除它。之后增加DCE删除的指令这个变量的值。

2.DCE思想：

DCE删除明显死亡的指令，但它重新检查已删除指令所使用的指令，以查看它们是否新死的。

3.与ADCE比较：

DCE是删除明显死亡的指令，而ADCE是假设所有指令死亡，然后通过分析来标记活跃的指令，没有标记的指令保持死亡，在最后被删除，这样能删除掉DCE无法发现的死指令。

三、实验总结

1.本次实验学习了中间代码的Constprop优化，常量传播优化是对操作数都是常量的指令，如果指令被使用就直接计算常量计算的结果（也是一个常量），用结果来代替对这条指令的结果的使用，这样这条指令的使用者可能也会出现操作数都是常量的情况，继续进行常量传播，一层一层地进行优化

2.本次实验学习了LLVM Pass的相关知识。在得到llvm IR 形式的输出之后，依赖不同的后端会得到不同的汇编码，再转为汇编码之前。如果对IR进行优化会得到执行效率更高的代码。LLVM IR是基于SSA形式的，这也就意味着对每个变量的赋值都会产生一个新的变量，或者说变量是不可变的，这是SSA的一种经典样例。

在LLVM的架构中，PASS的作用是优化LLVM IR。PASS作用于LLVM IR，处理IR，分析IR，寻找优化的机会，并修改IR，产生优化的代码，命令行工具opt就是用来在LLVM IR上运行各种优化PASS的。

我们这次选择PASS是ConstantProp pass和adce pass。ConstantProp pass会执行简单的常量传播，adce pass使用了更加激进的方式删除死代码。

adce pass首先会收集所有活跃的Terminator指令。基于活跃的Terminator向后传播生存性。之后删除死指令，即没有返回值，不影响控制流，没有副作用的指令。同时检查被删除指令是否被引用，如果存在对它们的引用，则引用也是死的，也删除。

四、实验反馈

1.在实验的过程中，由于对Pass框架不熟悉，有时对某些函数的使用不是特别理解

2.在设计体现优化的test的时候，不容易设计优化效果明显的test

五、参考资料

1.《LLVM Cookbook》中文版，王欢明 译