选做部分

数据流分析和优化简介

- 数据流分析是一组用来获取程序执行路径上的数据流信息的技术
- 数据流分析技术的应用
  - 到达-定值分析（Reaching-Definition Analysis)：
  - 活跃变量分析（Live-Variable Analysis)
  - 可用表达式分析（Available-Exprssion Analysis）

> 在每一种数据流分析应用中，都会把每个**程序点**和一个**数据流值**关联起来。

- 数据流分析的模式
  - `IN[s]`：语句s之前的数据流值
    `OUT[s]`：语句s之后的数据流值
  - f_s: 语句s的 **迁移函数**（transfer function）
  - 迁移函数有两种风格：信息沿执行路径正向传播或者逆向传播
    - 信息沿执行路径前向传播（前向数据流问题）：$OUT[s] = fs(IN[s])$
    - 信息沿执行路径逆向传播（逆向数据流问题）：$IN[s] = fs(OUT[s])$
  - 基本块中相邻两个语句之间的数据流值的关系:
    - 设基本块B由语句$s1, s2, \ldots, sn$顺序组成，则

      $IN[si + 1] = OUT[si]$
- 基本块的数据流模式
  - `IN[B]`：紧靠基本块B之前的数据流值
    `OUT[B]`：紧靠基本块B之后的数据流值
  - 设基本块B由语句$s1, s2, \ldots, sn$顺序组成，则
    - $IN[B] = IN[s1]$
    - $OUT[B] = OUT[sn]$
  - f_B:基本块B的迁移函数，同样也分为前向和后向数据流问题
    - 前向数据流问题：$OUT[B] = fB(IN[B])$
    - 后向数据流问题：$IN[B] = fB(OUT[B])$
- 数据流优化的有关应用
  - **常量传播**：在编译优化阶段，能够计算出结果的变量，直接替换为常量。通常分为过程内和过程间常量传播
  - **循环不变式外提**：把 `LLVM IR` 中的循环不变量外提，减少指令的执行次数，从而对 `LLVM IR` 进行优化 。
  - **函数内联替换**：对IR中的函数调用进行适当的内联展开
  - **公共子表达式删除**：消除冗余的表达式，减少重复计算
  - **强度削弱**：用一个较廉价的运算代替一个花费较大的运算
  - **归纳变量删除**：找到循环中的归纳变量，用以强度削弱以及死代码删除
  - **死代码删除**：移除对程序运行结果没有任何影响的代码
  - **循环展开**：降低循环开销，为具有多个功能单元的处理器提供指令级并行
  - **部分冗余优化**

数据流分析

函数基本信息的分析

统计函数的相关信息，包括

- 函数名
- 函数参数的个数
- 函数被调用的次数
- 函数包含的基本块个数
- 函数包含的LLVM IR指令数

```
for (auto iter = M.begin(); iter != M.end(); ++iter){
    Function &func = *iter;
    //函数名
```

```
    llvm::outs() << "Name:" << func.getName() << "\n";
    //函数参数个数
    llvm::outs() << "Number of Arguments: " << func.arg_size() << "\n";
    //函数被调用的次数（Module中定义的非main如果被调用，则输出为调用次数+1）（main函数对应为0）
    llvm::outs() << "Number of Direct Call Sites in the same LLVM module: " << func.getNumUses() << "\n";
    //函数的基本块个数
    llvm::outs() << "Number of Basic Blocks: " << func.size() << "\n";
    //函数所包含的IR数
    llvm::outs() << "Number of Instructions: " << func.getInstructionCount() << "\n";
    transformed = true;
}
```

测试和输出

```
//funcInfo.c
int add(int a, int b){
    return a + b;
}

int main(){
    int x, y;
    x = add(1, 2);
    y = add(3, 4);
    return 0;
}
```

输出结果

```
Functions Information Pass
Name:add
Number of Arguments: 2
Number of Direct Call Sites in the same LLVM module: 2
Number of Basic Blocks: 1
Number of Instructions: 2
Name:main
Number of Arguments: 0
Number of Direct Call Sites in the same LLVM module: 0
Number of Basic Blocks: 1
Number of Instructions: 3
```

活跃变量分析

算法思路

- class Liveness: 选择继承FunctionPass，是针对每一个函数的基本块作分析。
- 存储：使用`std::map<BasicBlock *, std::set<Value *>>`来存放每个块的IN[B],OUT[B], def_B, use_B, 一方面方便与基本块对应，另一方面set的特性使得集合元素互异
- 主要函数为runOnFunction
    - 首先初始化函数所有的基本块的def和use集合

        - 遍历每个基本块的每条指令的每个右操作数
            - 若不是常量、不是label、不是函数名、同时在该基本块该指令前无定值，则加入use集合
        - 对每条指令的左值，若指令不是跳转或返回指令`!instr.isTerminator()`则加入def集合。

          当然，用到tmpDef集合来记录基本块里已经分析到的定值变量。
    - 然后开始按书上算法迭代计算
        - IN、OUT集时，只需对对应的live_in, live_out里的一项set作插入删除操作即可
        - 需重点注意phi结点的特殊处理，因为若块1(有变量x1)和块2（有变量x2）的后继都是块3，并且在块3里有指令形如`%11 = phi i32 [ %x1, %1 ], [ %x2, %2 ]`，那么在计算块1的OUT集时，由于x2存在于块3的IN集中，所以需要剔除掉。

```
for(BasicBlock* block : successors(&bb)){//对bb块的所有直接后继块遍历
    for(auto
iter=live_in[block].begin();iter!=live_in[block].end();iter++){
        //先把该后继块的In集全部加入bb块的OUT集
        live_out[&bb].insert(*iter);
    }
    //再根据phi结点作删减
    for(auto phi_iter = block->phis().begin(); phi_iter != block-
>phis().end(); phi_iter++){//对所有phi指令遍历
        PHINode & phi_inst = *phi_iter;
        for(auto phi_inst_iter = phi_inst.block_begin(); //对当前phi
指令的所有前驱块遍历
            phi_inst_iter != phi_inst.block_end(); phi_inst_iter++){
            // 获取PHI指令中的各个前驱基础块
            BasicBlock* &curr_bb = *phi_inst_iter;
            // 如果该前驱基础块不是现在的基础块
            if(curr_bb != &bb){
                Value* curr_val =
phi_inst.getIncomingValueForBlock(curr_bb);//得到该前驱基础块对应在phi指
令中的变量
                live_out[&bb].erase(curr_val);//删除之
            }
        }
    }
}
```

- 在对一条指令的操作数遍历时，要注意
    - `for(auto iter = instr.op_begin(); iter != instr.op_end(); iter++)`
        - 有等号的指令：会依次遍历该指令等号右边的操作数； 无等号指令：全部操作数

          包括
            - 常量
            - 变量，label（IR里均会打上%）

              如`br label %return`的return ； `ret i32 %retval.0`中的retval.0
            - 对于函数调用等情况

              `%call = call i32 @gcd(i32 %v, i32 %sub)`,会依次看 v,sub, gcd (即先局部的% ,再全局的@)
            - 特别的，对于phi指令：不会遍历标号

`%retval.0 = phi i32 [ %u, %if.then ], [ %call, %if.else ]`只有u,call

- 对于instr本身，也代表了等式左边操作数(若不是等式的指令，则`instr.printAsOperand(llvm::outs(),false);`输出为`<badref>`

测试和输出

验证文件：`./tests/activeVar[1-4].c`

这里以`./tests/activeVar1.c`为例

输出的LLVM IR（-show-ir-after-pass命令后输出在./build/Promote_Memory_to_Register_0.ll）

```
define dso_local i32 @gcd(i32 %0, i32 %1) #0 {
  %3 = icmp eq i32 %1, 0
  br i1 %3, label %4, label %5

4:                                                ; preds = %2
  br label %10

5:                                                ; preds = %2
  %6 = sdiv i32 %0, %1
  %7 = mul nsw i32 %6, %1
  %8 = sub nsw i32 %0, %7
  %9 = call i32 @gcd(i32 %1, i32 %8)
  br label %10

10:                                               ; preds = %5, %4
  %11 = phi i32 [ %0, %4 ], [ %9, %5 ]
  ret i32 %11
}

; Function Attrs: noinline nounwind uwtable
define dso_local void @main() #0 {
  %1 = icmp slt i32 7, 8
  br i1 %1, label %2, label %3

2:                                                ; preds = %0
  br label %3

3:                                                ; preds = %2, %0
  %4 = phi i32 [ 7, %2 ], [ 8, %0 ]
  %5 = phi i32 [ 8, %2 ], [ 7, %0 ]
  %6 = call i32 @gcd(i32 %5, i32 %4)
  %7 = call i32 (i32, ...) bitcast (i32 (...)* @output to i32 (i32, ...)*)(i32
%6)
  ret void
}
```

找到的活跃变量：

```
function  gcd:
active vars :
IN:
label %2 [ %0 , %1 ,  ]
label %4 [ %0 ,  ]
label %5 [ %0 , %1 ,  ]
label %10 [ %0 , %9 ,  ]

OUT:
label %2 [ %0 , %1 ,  ]
label %4 [ %0 ,  ]
label %5 [ %9 ,  ]
label %10 [  ]
```

```
function  main:
active vars :
IN:
label %0 [  ]
label %2 [  ]
label %3 [  ]

OUT:
label %0 [  ]
label %2 [  ]
label %3 [  ]
```

可用表达式分析

算法思路

- GetAvailExpr 类从 FunctionPass 继承，具体计算都在 runOnFunction 函数中完成。
- 1. 计算各个基本块中的gen，kill集合。
  - 第一次遍历块中指令，将指令加入all集合中以便迭代计算in，out，并且得到该基本块中有定值的变量合集，存入def向量中。
  - 第二次遍历块中指令，若该条指令后没有对其中变量定值，则将对应表达式加入gen集合（def向量是按顺序存储的，可以只查找该条指令之后的def）；并且将含有该条指令等号左边变量的表达式加入kill集合。
- 2. 将各个基本块对应的gen，kill集合转换为BitVector形式存储。
- 3. 初始化各个基本块in，out集合，即新建BitVector变量。
- 4. 根据可用表达式算法进行迭代计算。计算过程中的集合操作更改为BitVector变量按位操作。如：

```
//tmp = (tmp - kill_e[&bb]) & gen_e[&bb];
for (int i = 0; i < bit_vector_size; i++){
    tmp[i] = (tmp[i] && !kill_e[&bb][i]) && gen_e[&bb][i];
 }
```

- 5. 输出计算结果。输出全集all中包含的表达式，直接以BitVector的01串形式输出in, out集合。

数据结构和其他细节

- 定义了Expr类，用于表示一个表达式。类成员和构造函数：

```
    class Expr{
        public:
            unsigned opcode;
            Value *loperand, *roperand, *inst;
            bool isUnary;

            Expr(Instruction *i){
                inst = i;
                opcode = i->getOpcode();
                loperand = i->getOperand(0);
                isUnary = (isa<llvm::UnaryOperator>(i));
                if (!isUnary) roperand = i->getOperand(1);
                else roperand = nullptr;
            }
    }
```

原先设计希望能够处理单目运算符和双目运算符，但实际中发现LLVM IR将 -a 翻译为 0 - a ，算数运算中基本没有单目运算符。

- Expr类的输出：使用 Instruction::getOpcodeName 函数输出运算符，使用 Value.printAsOperand(llvm::outs(), false) 函数输出变量。
- 上述提到的各个集合的数据类型：

```
std::map<BasicBlock *, std::set<Expr *>> gen, kill;
std::set<Expr *> all;
std::vector<Value *> def;       // temp variable
std::map<BasicBlock *, llvm::BitVector> in, out, gen_e, kill_e;
```

- 遍历基本块中指令时，只对 BinaryOperator, UnaryOperator 类型的指令进行处理。大多数遍历指令的循环中都有这一行：

```
if (!isa<llvm::BinaryOperator>(i) && !isa<llvm::UnaryOperator>(i)) continue;
```

- 在初始化in, out和输出时，entry块需要单独处理。由于 `for (BasicBlock &bb : F)` 遍历时第一个得到的块就是entry，只需要在第一次for循环体执行时处理就可以。
- 由于IR代码是ssa形式，各个块必定有 `kill[bb] = ∅`，`out[bb] = gen[bb]`，各个gen集合是对全集的一个划分。在该可用表达式分析的基础上写公共子表达式删除也没有意义。

测试和输出

- 测试代码：`/tests/Availexpr.c`

```
int a=1, b=2, c=3, d=4, e=5;

int main(){
    c = a + b;
    d = a + b;
    e = -a;
    b = c - d;
    a = b + e;
    c = b + e;
```

```
    for (int i = 0; i < 5; i++){
        if (b<a) b = b*2;
        else b = a+b;
    }
    return 0;
}
```

- 生成IR:

```
; ModuleID = '../tests/Availexpr.c'
source_filename = "../tests/Availexpr.c"
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-unknown-linux-gnu"

@a = dso_local global i32 1, align 4
@b = dso_local global i32 2, align 4
@c = dso_local global i32 3, align 4
@d = dso_local global i32 4, align 4
@e = dso_local global i32 5, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  %0 = load i32, i32* @a, align 4
  %1 = load i32, i32* @b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* @c, align 4
  %2 = load i32, i32* @a, align 4
  %3 = load i32, i32* @b, align 4
  %add1 = add nsw i32 %2, %3
  store i32 %add1, i32* @d, align 4
  %4 = load i32, i32* @a, align 4
  %sub = sub nsw i32 0, %4
  store i32 %sub, i32* @e, align 4
  %5 = load i32, i32* @c, align 4
  %6 = load i32, i32* @d, align 4
  %sub2 = sub nsw i32 %5, %6
  store i32 %sub2, i32* @b, align 4
  %7 = load i32, i32* @b, align 4
  %8 = load i32, i32* @e, align 4
  %add3 = add nsw i32 %7, %8
  store i32 %add3, i32* @a, align 4
  %9 = load i32, i32* @b, align 4
  %10 = load i32, i32* @e, align 4
  %add4 = add nsw i32 %9, %10
  store i32 %add4, i32* @c, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %11 = load i32, i32* %i, align 4
  %cmp = icmp slt i32 %11, 5
  br i1 %cmp, label %for.body, label %for.end
```

```
for.body:                                         ; preds = %for.cond
  %12 = load i32, i32* @b, align 4
  %13 = load i32, i32* @a, align 4
  %cmp5 = icmp slt i32 %12, %13
  br i1 %cmp5, label %if.then, label %if.else

if.then:                                          ; preds = %for.body
  %14 = load i32, i32* @b, align 4
  %mul = mul nsw i32 %14, 2
  store i32 %mul, i32* @b, align 4
  br label %if.end

if.else:                                          ; preds = %for.body
  %15 = load i32, i32* @a, align 4
  %16 = load i32, i32* @b, align 4
  %add6 = add nsw i32 %15, %16
  store i32 %add6, i32* @b, align 4
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  br label %for.inc

for.inc:                                          ; preds = %if.end
  %17 = load i32, i32* %i, align 4
  %inc = add nsw i32 %17, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-
sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="all" "less-
precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false"
"no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-
math"="false" "no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 11.0.0"}
```

- 可用表达式分析输出：

```
all expressions: %9.add.%10   %5.sub.%6   %2.add.%3   %7.add.%8   %0.add.%1
0.sub.%4   %i.add.1   %13.mul.2   %14.add.%15
%entry      in: 000000000   out: 000000000  gen: 111111000  kill: 000000000
%for.cond       in: 111111111   out: 000000000  gen: 000000000  kill: 000000000
%for.body       in: 111111111   out: 000000000  gen: 000000000  kill: 000000000
%if.then        in: 111111111   out: 000000010  gen: 000000010  kill: 000000000
%if.else        in: 111111111   out: 000000001  gen: 000000001  kill: 000000000
%if.end     in: 111111111   out: 000000000  gen: 000000000  kill: 000000000
%for.inc        in: 111111111   out: 000000100  gen: 000000100  kill: 000000000
%for.end        in: 111111111   out: 000000000  gen: 000000000  kill: 000000000
```

数据流优化

强度削弱

除以2运算

- 将除法削弱为算术右移指令

乘2运算

- 将乘法削弱为左移指令

针对算术不变量的优化

加法

- 检测加法表达式的某个操作数是否为0
- 对于形如`x1 = x + 0`，将`x1`的使用替换为`x`，并将`x1`对应的指令删除

乘法

- 检测乘法表达式的某个操作数是否为1
- 对于形如`x1 = x * 1`，将`x1`的使用替换为`x`，并将`x1`对应的指令删除

局部连续指令的优化

将形如如下的指令序列（中间可以有其他指令）

```
b = a + 1
...
c = b - 1
...
d = c + 2
```

优化为

```
b = a + 1
...
d = a + 2
```

包含以上三部分的测试和输出

```c
//three.c
int main()
{
    int x = 1;
    if(x < 0){
        x = 2;
    }

    int val1 = x + 0;
    int val2 = x * 1;
    int val3 = 2 * x;
    int val4 = x / 2;

    int val5 = x + 1;
    int val6 = val5 - 1;
    int val7 = val6 + 8;
    //为了防止死代码删除将一些变量优化掉，加了temp变量并return
    int temp = val1 + val2 + val3 + val4 + val5 + val6 + val7;
    return temp;
}
```

优化后的LLVM IR如下

```
define dso_local i32 @main() #0 {
entry:
  %cmp = icmp slt i32 1, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  br label %if.end

if.end:                                           ; preds = %if.then, %entry
  %x.0 = phi i32 [ 2, %if.then ], [ 1, %entry ]
  %0 = shl i32 %x.0, 1
  %1 = ashr i32 %x.0, 1
  %add2 = add nsw i32 %x.0, 1
  %add3 = add nsw i32 %x.0, 8
  %add4 = add nsw i32 %x.0, %x.0
  %add5 = add nsw i32 %add4, %0
  %add6 = add nsw i32 %add5, %1
  %add7 = add nsw i32 %add6, %add2
  %add8 = add nsw i32 %add7, %x.0
  %add9 = add nsw i32 %add8, %add3
  ret i32 %add9
}
```

可以看到，`val1`和`val2`对应算术不变量，对应的指令直接被删除，替换为`x`

乘2和除以2的指令被替换为位移指令

`val6`对应的减法指令被删掉了，因为在后面的加法指令中`val6`的值和`x`是相同的，因此可以直接用`x`代替

同时会输出对应的统计信息

```
Transformations applied:
        The num of Algebraic Identity: 2
        The num of Strength Reduction: 2
        The num of Two-Inst Optimization: 1
```

循环不变式外提

- 由于循环不变量的种类与情况比较复杂，刻画起来也较难，所以本次是验只针对了，部分满足一定特征的循环不变式。该特征是左值的确定，使用的操作数没有在循环中被定值。

算法思路

> 核心思想

对一个循环，先确定循环内被定值的变量集合，然后对每一条指令查看其操作数，如果该指令的所有操作数不是循环内的定值变量，则说明该指令是循环不变量。然后把该指令加到循环入口的所有前驱节点中。

> 具体实现

- 首先先获取循环的信息，`get_loop_information( func)`。这个过程复用了必做部分的循环分析。然后将循环的相关信息存放在容器中。具体存放的信息有：

```
    // loops found in a function
    std::unordered_map<Function *, std::unordered_set<BBset_t *>> func2loop;
    // { entry bb of loop : loop }
    std::unordered_map<BasicBlock *, BBset_t *> base2loop;
    // { loop : entry bb of loop }
    std::unordered_map<BBset_t *, BasicBlock *> loop2base;
```

- 然后是确定每个循环的定值变量集合。这里采用的方法是左值都是循环内被定值的。

```
 auto var= dynamic_cast<Value *>(&instr);     //最左边的参数
 var2loop[var].insert(loop);
```

- 然后对每一条指令判断是否循环不变式，判断的条件是：

```
if( !isa<Constant>(var)&& var2loop.find(var)!=var2loop.end()&&
(var2loop[var].find(loop)!=var2loop[var].end())){
                                    //说明该指令使用的变量是在循环里面定值的

                                    }
```

- 至此得到了循环不变式。最后一步是把指令外提到循环外。这里直接使用了原生的工具：

```
   /// Unlink this instruction from its current basic block and insert it
into
   /// the basic block that MovePos lives in, right before MovePos.
   void moveBefore(Instruction *MovePos);
```

测试和输出

1. 外提一层循环

   loopinvhoist_1.c

```
int main(){
    int num = 0 ,i,j;
    int a = 3;
    int b;
    for(int i=0;i<10;i++){
        num = i + j;
        b = a + 9;
        num = i + j;
    }
}
```

   运行指令 `./mClang  ../tests/loopinvhoist_1.c -show-ir-after-pass` 。得到的结果为：

```
Processing function  main,  LoopInvHoist result is as follow
{
------
loop begin
LCM instruction add  %1 = add nsw i32 3, 9
}
```

   `Pass` 过后的 `IR` 为：

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = add nsw i32 3, 9
  br label %2

2:                                                ; preds = %8, %0
  %3 = phi i32 [ 0, %0 ], [ %9, %8 ]
  %4 = icmp slt i32 %3, 10
  br i1 %4, label %5, label %10

5:                                                ; preds = %2
  %6 = add nsw i32 %3, undef
  %7 = add nsw i32 %3, undef
  br label %8

8:                                                ; preds = %5
  %9 = add nsw i32 %3, 1
  br label %2

10:                                               ; preds = %2
  ret i32 0
}
```

   可以明确地看到指令是被外提到了循环外面。

2. 外提两层循环

   loopinvhoist_2.c

```
int main(){
    int num = 0 ,i,j;
    int a = 3;
    int b;
    for(int j=0;j<10;j++)
    for(int i=0;i<10;i++){
        num = i + j;
        b = a + 9;
        b = a + 9;
        num = i + j;
    }
}
```

运行指令`./mClang  ../tests/loopinvhoist_2.c -show-ir-after-pass`得到输出的提示信息为:

```
Processing function  main,  LoopInvHoist result is as follow
{
------
loop begin
LCM instruction add  %5 = add nsw i32 3, 9
loop begin
LCM instruction add  %1 = add nsw i32 3, 9
LCM instruction add  %2 = add nsw i32 3, 9
}
```

最终的`IR`为:

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = add nsw i32 3, 9
  %2 = add nsw i32 3, 9
  br label %3

3:                                                ; preds = %16, %0
  %4 = phi i32 [ 0, %0 ], [ %17, %16 ]
  %5 = icmp slt i32 %4, 10
  br i1 %5, label %6, label %18

6:                                                ; preds = %3
  br label %7

7:                                                ; preds = %13, %6
  %8 = phi i32 [ 0, %6 ], [ %14, %13 ]
  %9 = icmp slt i32 %8, 10
  br i1 %9, label %10, label %15

10:                                               ; preds = %7
  %11 = add nsw i32 %8, %4
  %12 = add nsw i32 %8, %4
  br label %13

13:                                               ; preds = %10
  %14 = add nsw i32 %8, 1
  br label %7
```

```
15:                                               ; preds = %7
  br label %16

16:                                               ; preds = %15
  %17 = add nsw i32 %4, 1
  br label %3

18:                                               ; preds = %3
  ret i32 0
}
```

可以明确地看到指令是被外提到了循环外面。并且外提了两层。

常量传播

算法思路

- 定义
  - 常量折叠：多个变量进行计算时，而且能够直接计算出结果，那么变量将由常量直接替换。

    ```
    int a = 3+1-1*5;
    printf("%d",a);
    替换为
    printf("%d",-1);
    ```

  - 常量传播：能够计算出结果的变量直接替换为常量，并且能够把SSA形式下冗余的代码删去

    ```
    int a = 1;
    int b=a+c;
    换为
    int a = 1;
    int b=1+c;
    最后删去int a=1;
    ```

- 具体思路
  - 将所有的常数运算及其所有的表现形式都做一个替换，以便删除

    遍历所有指令->判断是否为运算指令或cmp指令等->判断operands是否都为常数

    - 若全为常数：将其入栈(wait_delete)，并且对每个左值进行替换，这样之后分析后续语句时该条指令的对应的变量可以视为const
    - 若不是：continue
    - 结束后将栈里的冗余东西都删去
  - 代码实现上：
    - 采用ModulePass，本质还是对每个基本块的遍历。
    - 在runOnBasicBlock函数里遍历块内每一条指令，若为二元运算或者Cmp指令等，则取其操作数，并利用llvm提供的dyn_cast<> API来判断是否为常量（浮点数和整数分别作处理）
    - 所有操作数均为常量的情况下，调用compute（二元运算）或者compare（cmp指令）把对应的值计算出来
    - 最后replaceAllUsesWith替换掉所有该变量的use，删除该冗余指令。

    ```
    for (Instruction &instr : B) {//从基本块第一条指令开始遍历
    ```

```
    bool allConst = false;
    llvm::outs()<<"instr opcode:"<<instr.getOpcode()<<"\n";
    if (instr.isBinaryOp()) {
        Value *oper0,*oper1;
        auto opID = instr.getOpcode();
        oper0 = instr.getOperand(0);
        oper1 = instr.getOperand(1);
        if(oper0->getType()->isFloatTy()){
            if(auto constFP0=dyn_cast<ConstantFP>(oper0)){
                if(auto constFP1=dyn_cast<ConstantFP>(oper1)){
                    allConst=true;
                    auto ans = compute(opID,constFP0,constFP1,&instr);
                    instr.replaceAllUsesWith(ans);
                }
            }
        }
        else if(oper0->getType()->isIntegerTy()){
            if(auto constInt0=dyn_cast<ConstantInt>(oper0)){
                if(auto constInt1=dyn_cast<ConstantInt>(oper1)){
                    allConst=true;
                    auto ans = compute(opID,constInt0,constInt1,&instr);
                                instr.replaceAllUsesWith(ans);
                }
            }
        }
    }
    if(instr.getOpcode()==llvm::Instruction::ICmp){
        Value *oper0, *oper1;
        oper0 = instr.getOperand(0);
        oper1 = instr.getOperand(1);
        if(auto constInt0=dyn_cast<ConstantInt>(oper0)){
            if(auto constInt1=dyn_cast<ConstantInt>(oper1)){
                allConst=true;
                auto ans = compare(constInt0,constInt1,&instr);
                instr.replaceAllUsesWith(ans);
            }
        }
    }
    if (allConst) {
        wait_delete.push_back(&instr);
    }
}
//删除需要删除的instr
for (Instruction *inst : wait_delete)
    if (inst->isSafeToRemove())
        inst->eraseFromParent();
}
```

测试和输出

- 结果示例
  - 优化前

```
define dso_local i32 @main() #0 {
entry:
  %cmp = icmp sgt i32 0, 3
  br i1 %cmp, label %if.then, label %if.else
```

```
if.then:                                          ; preds = %entry
  br label %if.end

if.else:                                          ; preds = %entry
  br label %if.end

if.end:                                           ; preds = %if.else,
%if.then
  %b.0 = phi i32 [ 2, %if.then ], [ 3, %if.else ]
  ret i32 %b.0
}
```

- 优化后

```
define dso_local i32 @main() #0 {
entry:
  br i1 false, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  br label %if.end

if.else:                                          ; preds = %entry
  br label %if.end

if.end:                                           ; preds = %if.else,
%if.then
  %b.0 = phi i32 [ 2, %if.then ], [ 3, %if.else ]
  ret i32 %b.0
}
```

思考题

理解DCE Pass

在 MyPasses.hpp 中定义了myDCEPass类，这是一个FunctionPass，它重写了 `bool runOnFunction(Function &F)`。请阅读并实践，回答：

1）简述 `skipFunction()` 、 `getAnalysisIfAvailable<TargetLibraryInfoWrapperPass>()` 的功能

2）请简述DCE的数据流分析过程，即 `eliminateDeadCode()` 和 `DCEInstruction()`

---

1. - `skipFunction()`：该函数检查是否跳过可选的passes。如果 `optimization bisect` 超过了限制，或者 `Attribute::OptimizeNone` 被置位，则可选的passes会被跳过， `runOnModule` 方法直接返回 `false`
   - `getAnalysisIfAvailable<TargetLibraryInfoWrapperPass>()`：
     - 子类使用这个函数来获取可能存在的分析信息，例如更新它。
     - 经常被用来转换API。在转换执行时该方法自动更新一次pass的分析结果
     - 与 `getAnalysis` 不同，因为 `getAnalysis` 可能会失败(如果分析结果还没有计算出来)，所以应该在分析不可用的情况下且你能够处理时使用 `getAnalysis`。

2. ○ `eliminateDeadCode()`

```
//根据当前可用的库函数的信息，删除函数F中的死代码
static bool eliminateDeadCode(Function &F, TargetLibraryInfo *TLI) {
   bool MadeChange = false;
   SmallSetVector<Instruction *, 16> WorkList;
   std::cout << "The Elimated Instructions: {" << std::endl;
   //遍历F中的每一条指令，找到其中的死代码并加入WorkList
   for (inst_iterator FI = inst_begin(F), FE = inst_end(F); FI != FE;) {
         Instruction *I = &*FI;
         ++FI;

         if (!WorkList.count(I)) //如果I不在WorkList中
         MadeChange |= DCEInstruction(I, WorkList, TLI); //判断I是否为死代
码，如果是则加入WorkList并将MadeChange置为真
   }

   //WorkList相当于一个栈，现在从栈顶开始，判断I是否为死代码，如果是则加入WorkList
并将MadeChange置为真
   while (!WorkList.empty()) {
         Instruction *I = WorkList.pop_back_val();
         MadeChange |= DCEInstruction(I, WorkList, TLI);
   }
   std::cout << "}" << std::endl;
   return MadeChange;  //返回MadeChange，表示IR是否发生变化
}
```

○ `DCEInstruction()`

```
   //判断I是否为死代码，如果是，则将其加入WorkList并返回真，否则返回假
   static bool DCEInstruction(Instruction *I, SmallSetVector<Instruction
*, 16> &WorkList,
      const TargetLibraryInfo * TLI) {
      //当前指令use_empty()返回值为真 且 不是跳转指令或者返回指令 且 没有副作用
      //Terminator Instruction--In LLVM, a BasicBlock will always end
with a TerminatorInst.
      //TerminatorInsts cannot appear anywhere else other than at the
end of a BasicBlock.
      //对于一个Instruction *I，I指向的是该指令的%开头的左值，其右值有三种情况：1.
常数 2.@开头的全局变量 3.%开头的局部变量
      if (I->use_empty() && !I->isTerminator() && !I-
>mayHaveSideEffects()) {
            //对于该指令的每个操作数（右值）
            for (unsigned i = 0, e = I->getNumOperands(); i != e; ++i) {
               Value *OpV = I->getOperand(i);  //OpV为指向当前操作数的指针
               I->setOperand(i, nullptr);      //设置指向该操作数的指针为空

               if (!OpV->use_empty() || I == OpV)  //如果该操作数
use_empty()返回值为假 或 指令的左值为当前操作数
                  continue;                        //进入下一轮循环

               if (Instruction *OpI = dyn_cast<Instruction>(OpV))  //如果
该操作数不符合前面if的条件，且可以动态转换为对应的IR指令（也就是说它是%开头的）
                  if (isInstructionTriviallyDead(OpI, TLI))   //如果这条指
令产生的该值没被使用过且这个指令没有其他影响
```

```
                        WorkList.insert(OpI);           //将该指令插入
WorkList
            }
            //print:将对象格式化到指定的缓冲区。如果成功，将返回格式化字符串的长
度。如果缓冲区太小，则返回一个大于BufferSize的长度以供重试。
            I->print(llvm::outs()); //
            std::cout << " " << I->getOpcodeName() << std::endl;    //输
出该指令的操作码
            I->eraseFromParent();                                   //将
该指令从包含它的基本块中解除链接并删除它
            return true;    //成功删除一条死代码,返回真
        }
        return false;   //否则返回假
    }
```

理解Global Pass

在 MyPasses.hpp 中定义了myGlobalPass类，这是一个ModulePass，它重写了 `bool runOnModule(llvm::Module &M)`。请阅读并实践，回答：

1）简述 `skipModule()` 的功能

2）请扩展增加对Module中类型定义、全局变量定义等的统计和输出
（张昱老师和刘硕助教指出：类型定义指形如 `typedef struct` 的类型别名）

---

1. 该函数检查是否跳过可选的passes。如果 `optimization bisect` 超过了限制，则可选的passes会被跳过， `runOnModule` 方法直接返回 `false`
2. 具体实现在 `MyPasses.hpp` 中，主要部分如下

```
int num_of_globalVariable = 0;
std::cout << "The global variables are as follows:" << std::endl;
for(llvm::Module::global_iterator GI = M.global_begin(), GE =
M.global_end(); GI != GE; ++GI, ++num_of_globalVariable){
    llvm::outs() << GI->getName() << "\n";
}
//注意printf和scanf函数会引入全局变量，应该是对应的格式化字符串
std::cout << "The number of global variable is " << num_of_globalVariable <<
"." << std::endl;
std::cout << "The type alias are as follows:" << std::endl;
for(auto i : M.getIdentifiedStructTypes()){
    llvm::outs() << i->getStructName() << "\n";
}
std::cout << "The number of type alias is " <<
M.getIdentifiedStructTypes().size() << "." << std::endl;
```

测试和对应的输出

1. 全局变量定义： 输出了全局变量的名称和数量

```
//global_variable.c
#include <stdio.h>
int a;
int b;
char c;
```

```
int pqz(){
    printf("123");
    int c;
    return 0;
}

int main(){
    printf("hello world.");
    return 0;
}
```

输出

```
The global variables are as follows:
.str
.str.1
a
b
c
The number of global variable is 5.
```

2. 类型定义： 输出了类型定义的名称、每个类型定义下包含的类型数目、类型定义的总数量

```
//type_alias.c
#include<stdio.h>

typedef struct stu1{
    int a;
}temp1;

typedef struct stu2
{
    int a;
    char b;
}temp2;

typedef struct stu3
{
    int a;
    char b;
    float c;
} temp3;

temp1 t1;
temp2 t2;
temp3 t3;

int main()
{
    temp1 b;
    return 0;
}
```

输出

```
The type alias are as follows:
struct.stu1
1
struct.stu2
2
struct.stu3
3
The number of type alias is 3.
```

问题、难点与展望

强度削弱

- 对于和2的幂次有关的乘法、除法运算，可以优化为对应的位移运算
- 如果可以找到归纳变量，则可对归纳变量进行强度削弱
- 其他运算的强度削弱

连续指令的优化

- 当前的优化较为局限，只是针对两个指令，一加一减，且在后续没有对变量的重新赋值的情况下进行的优化

循环不变式外提

现存问题：

- 由于对 `void moveBefore(Instruction *MovePos)` 的行为不够理解， 导致每个循环只能外提指令的时候不能把所有的循环不变量外提。如果循环中有多个不变量 ，外提一个后，就无法继续遍历该基本块中的指令。
- 还有问题是，只对运算指令外提。对于直接赋常量语句，并没有识别成循环不变式。这个问题是一开始提到的，对循环不变式的刻画不够完备。

如何解决：

好在该实验已经对循环的分析比较完备 ，算法也是清晰的 。 相当于输入是可以的，剩余的问题是对循环不变式的刻画 ， 以及对移动指令后产生的结果的理解。

可用表达式分析

现存问题：

- 考虑将该pass扩展到删除公共子表达式
- 由于生成的LLVM IR代码SSA形式，实际中没有公共子表达式。

如何解决

- 删除公共子表达式大致思路：遍历基本块，再对基本块中每条指令，查看它是否在in[bb]集合中，如果在，则从该基本块的前驱中查找该表达式，将等号右边Value*变量替换为查找到的表达式。
- 关于SSA格式代码的问题考虑了两种解决方法：
    1. 将前继的load，store指令一同处理，用被load的变量替换临时变量的操作数，处理得到C语言文件中的表达式。在这个基础上进行可用表达式分析和公共子表达式删除
    2. 在生成SSA代码前运行上述分析和优化。
    3.

常量传播

现存问题

1. 写完代码才发现，在所有Pass之前生成的中间代码就已经把常量折叠这块优化了，比如3+5会直接就在IR里以8的形式出现。
2. 删除冗余的跳转指令
    - 在已完成的示例里，形如`br i1 false, label %if.then, label %if.else`这样的条件指令是存在不可达基本块的，即实际上其不可能跳转到`label %if.then`，所以还可以考虑对其做修改处理，变为强制性跳转。
    - 对于分支嵌套的情况都能够删除掉无用的分支，这一步之后对于可能出现的无法到达的块都需要进行删除比如从`label1`跳转到`label2`，但是现在由于优化导致不可能跳转到label1,所以label2这里也可以考虑删除。
3. 还可以对常量折叠进行扩展，比如对load，store、整形换浮点、浮点换整形等指令

如何解决

1. 参考issue里关闭优化这一块，可以去探究`TheDriver.InitializePasses();`里添加的一些优化，考虑去除其中有关常量折叠的优化则可以更好的测试这部分的代码。
2. 思路
    - 首先可以考虑判断一个基本块的终结指令是不是cond_br以及cond是否已经确定为true或false
    - 如果flag是确定的，那么通过operand位置判断应该前往和应该删除哪个基本块（即无法到达的）
    - 并且还需对要删除的冗余块往后遍历，把嵌套的分支一并删除。
    - 还需要考虑因为分支存在而存在的phi指令的处理：如果有来自被删除基本块的label，就把这个phi删除，用另一个oper进行replace_all_use

其他问题及展望

- 实现了数个基本块内优化，一个块间优化和数个块间分析。没有实现如函数内联等过程间分析和优化。

  过程间优化可能出现的问题：优化决策导致的依赖性。在以文件为编译单元的情况下，跨文件的函数内联等优化，修改了某些全局变量，导致之前编译完成的部分无效，需要重新编译程序中的其他文件。
- 尚未实现的分析和优化及可能遇到的问题：
    - 函数内联替换：
        - 形参和实参在函数体中的转换
        - 函数局部变量和当前环境中变量的重名问题
        - 决定是否对某个函数进行内联处理的条件
    - 公共子表达式删除：基本思路和问题如上述。
    - 归纳变量删除：应当在强度削弱和公共子表达式删除中完成。
    - 循环展开：
        - 循环内定义的局部变量，展开后与其他变量的重名问题
        - 判断是否展开循环的条件
        - 对有嵌套的循环如何展开
- 分析和优化的关联性不强，一些分析的结果实际上可以用来进行进一步的优化

- 优化之间的关联性不强，一些优化可能为其他优化带来契机。如果可以扩展更多Pass，并合理安排Pass的顺序，有望生成优化程度更高的LLVM IR

问题补充

- 强度削弱，使用位移指令替代乘2、除以2是否不妥当
    - 我后来思考了一下，因为计算机采用补码的方式存储，故对于负数确实不适用，但对于正数是没什么问题的
    - 另外用位移运算代替取余运算也是一个可以优化的点
- 活跃变量分析，LLVMIR对同一个变量的中间表示，怎样判断它是同一个变量
    - 我们的活跃变量分析的实现是直接从SSA格式的IR开始的，所以原本在c文件里的一个变量可能在SSA格式IR下变成了多个变量，我们是默认把这些区分开，不视为相同的变量，这也是符合活跃变量分析的约定的，而且这本身也是SSA格式的不足，要想把这些变量再重新对应起来是不现实的。基于活跃变量分析的后续优化诸如基本块的寄存器分配，基于我们实现的活跃变量分析，直接根据在基本块出口有哪些非活跃变量，然后释放对应的寄存器，也是可以很好的实现。
- 常量传播，替换所有在代码中的use是不妥当的?
    - 实际上是先对一个基本块内每条指令分析，把均为常数的指令计算出来，将该指令对应变量的后续use全部替换为该常量，然后把该条指令加入待删除队列。在基本块全部遍历完后，再删除这些冗余指令。具体实现请参考`my-llvm-driver\include\optimization\ConstPropagation.hpp`的`runOnBasicBlock`函数
- 循环不变式外提
    - 提问： 你说的只能外提一条指令的原因是什么，刚才没听清楚。

      答： 因为在`moveBefore()`后对`BasicBlock`中`instructions`的迭代会出错，为了保证正确性，我的方法是`moveBefore()`后中止掉该块的便利。所以导致一个块只能外提一条。
    - 提问：循环不变式外提对嵌套循环是怎么处理的的。

      答：对循环嵌套的处理是从内而外的外提。至于如何确定循环的嵌套信息在必做部分已经实现，只需要确定对循环的分析顺序即可。