代码优化

1 基础要求实验流程

1.1 建立控制流图以及消除不可达指令和基本块

```
int a = 5;
if(a > 5){
    return 5;
    if(a > 6){
        return 7;
    }
}
return 0;
int b = 10;
```

```
  %r4 = alloca i32
  %r1 = alloca i32
  store i32 5, ptr %r1
  %r2 = load i32, ptr %r1
  %r3 = icmp sgt i32 %r2, 5
  br i1 %r3, label %L1, label %L2
L1:
  ret i32 5
  %r5 = load i32, ptr %r1
  %r6 = icmp sgt i32 %r5, 6
  br i1 %r6, label %L3, label %L4
L3:
  ret i32 7
  br label %L4
L4:
  br label %L2
L2:
  ret i32 0
  store i32 10, ptr %r4
```

```
  %r4 = alloca i32
  %r1 = alloca i32
  store i32 5, ptr %r1
  %r2 = load i32, ptr %r1
  %r3 = icmp sgt i32 %r2, 5
  br i1 %r3, label %L1, label %L2
L1:
  ret i32 5
L2:
  ret i32 0
```

该优化要求你建立控制流图并删除不可达的指令和基本块，算法流程如下：

1. 从函数的入口基本块开始进行一次 dfs，在 dfs 过程中检查每个基本块的跳转指令，并根据跳转指令建立控制流图。如果该基本块是第一次遇到，就遍历该基本块直到末尾或者第一次出现跳转/函数返回指令。如果出现了跳转/函数返回指令，就将该基本块该指令后的其余指令全部删除。

2. 在完成 dfs 后，如果该基本块没有被访问过，即意味着该基本块永远不会被执行到，我们可以将这些基本块全部删除。

1.2 mem2reg 的简单情况

在中间代码生成这一次实验中，你已经成功生成了 SSA 形式的 LLVMIR，但是这个 SSA 形式并不是真正的 SSA，而是借助内存操作指令 alloca/load/store 来存储局部变量的 SSA 形式。并且这种 SSA 形式会频繁的对内存进行读写，同时也不利于后续的编译优化，所以我们需要尽可能消除对局部变量的 alloca，load，store 指令的同时维护 SSA 性质。

我们考虑每一个 alloca 指令产生的指针，我们记对该指针的 load 为 use，对该指针的 store 为 def。mem2reg 优化要做的是在维持 SSA 性质的前提下，删去该 alloca 指令和相关的 load，store 指令，将在 alloca 出来的内存中存储的值改为用寄存器存储。

对于基础要求，我们只需要实现 mem2reg 的两种简单情况即可。

1.2.1 alloca 的指针没有被 use

```c
int main()
{
    int a = 5;
    int b = 10;
    return 0;
}
```

```
define i32 @main(){
  %r1 = alloca i32
  %r2 = alloca i32
  store i32 5, ptr %r1
  store i32 10, ptr %r2
  ret i32 0
}
```

```
define i32 @main(){
  ret i32 0
}
```

这种情况是最简单的一种，你只需要遍历一次所有指令，如果发现某个 alloca 结果没有任何 use(即没有被 load 过)，就将与该 alloca 相关的指令全部删除。

框架中同一基本块中的指令采用 deque 存储，单次删除是线性的时间复杂度，这对于一些大型代码来说是不可接受的。所以我们可以先将要删除的指令标记。在检查完所有的 alloca 后，新建一个 deque，将没被标记的指令插入进该 deque 中，随后更换基本块中的 deque 即可，这样只需要线性时间就可以完成所有的删除操作。

框架中的测试脚本对你编译时间有一定的要求，我们要求你的时间复杂度为线性或者线性对数 ($\Theta(nlogn), \Theta(nlognlogn)$ 均可，n 为指令数)，如果你算法的时间复杂度不满足要求，会无法通过测试。

1.2.2 alloca 的指针的 def 和 use 均在同一基本块中

```c
int main()
{
    int a = getint();
    int b = getint();
    a = a + 5;
    return a+b;
}
```

```
define i32 @main() {
  %r2 = alloca i32
  %r3 = alloca i32
  %r4 = call i32 @getint()
  store i32 %r4, ptr %r2
  %r5 = call i32 @getint()
  store i32 %r5, ptr %r3
  %r6 = load i32, ptr %r2
  %r7 = add i32 %r6, 5
  store i32 %r7, ptr %r2
  %r8 = load i32, ptr %r2
  %r9 = load i32, ptr %r3
  %r10 = add i32 %r8, %r9
  ret i32 %r10
}
```

```
define i32 @main() {
  %r4 = call i32 @getint()
  %r5 = call i32 @getint()
  %r7 = add i32 %4, 5
  %r10 = add i32 %7, %5
  ret i32 %r10
}
```

如果我们确定了某一个 alloca 的 use 和 def 在同一基本块内，我们只需要遍历该基本块，同时维护一个变量 val 并将该变量初始化为 undef。如果我们遇到了一条与该 alloca 相关的 store 指令，就将 val 设置为该 store 指令要向内存中写的值，并删除该指令；如果我们遇到了一条与该 alloca 相关的 load 指令，就将所有使用该 load 结果的寄存器替换为 val，并删除该指令。

和之前的优化一样，我们同样对你的算法时间复杂度有一定要求，你需要在一次遍历中处理完所有的 alloca 指令，在处理 alloca 指令的过程中，我们可以先保存要替换的寄存器以及标记要删除的指令，但不实际执行相关操作；遍历完成后，我们可以在后续的常数次遍历中完成寄存器的替换和指令的删除。

对于这一优化，**框架中的测试脚本同样对你编译时间有一定的要求，你的时间复杂度依旧只能为线性或者线性对数 ($\Theta(nlogn), \Theta(nlognlogn)$ 均可，n 为指令数)，如果你算法的时间复杂度不满足要求，会无法通过测试。**

2 完整 mem2reg 实验流程

基础要求中，你已经实现了 mem2reg 的两种简单情况，但实际上，mem2reg 要远比这两种情况复杂。

如图2.1所示，局部变量 a 在基本块 B1 中被定义，我们将其重命名为 a_1, a 在 B2 中也被定义了一次，我们将其重命名为 a_2，那么在基本块 B3 中对局部变量 a 的使用是 a_1 还是 a_2 呢，我们不能确定，因为我们无法知道程序执行时是从 B1 还是 B2 跳转到 B3 的。

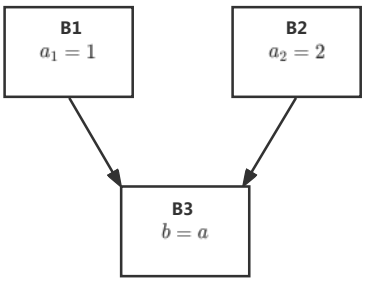

图 2.1: 重命名产生的问题

SSA 形式引入了 ϕ 指令来解决这一问题，如图2.2所示，我们在 B3 的开头插入了 ϕ 指令 $a_3 = \phi(a_1, a_2)$，如果从 B1 跳转到了 B3，那么 $\phi(a_1, a_2) = a_1$，如果从 B2 跳转到了 B3，那么 $\phi(a_1, a_2) = a_2$

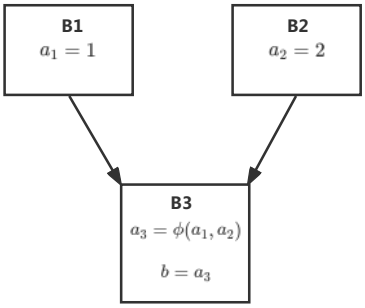

图 2.2: 静态单赋值形式

上述只是一个简单示例，真实环境下的代码会有非常复杂的控制流，如何在合适的位置插入 ϕ 指令，并且让 ϕ 指令的数量最少就成了一个关键问题，为了解决这一问题，我们需要建立支配树，计算支配边界得到合适的 ϕ 指令插入位置，并完成变量重命名，下面会提供一些参考资料，具体细节需要同学们自己探索。

整体算法流程 整体算法流程和介绍推荐参考LLVM 如何构造 SSA：Mem2Reg Pass。

建立支配树 支配树和支配边界的计算网上参考资料较多，大家可以自行搜索，这里不提供具体的参考资料。

插入 ϕ 指令 推荐参考Static Single Assignment Book的 3.1.1 节和 3.1.2 节。

变量重命名 推荐参考Static Single Assignment Book的 3.1.3 节

实验结果 如果你实现了完整的 mem2reg，那么你的编译器可以完成如下的优化：

```c
int main()
{
    int x = 0;
    int n = getint();
    int i = 0, S = 0;
    while(i < n){
        if(i==10){
            S = S*2;
        }
        S = S + 2;
        i = i + 1;
    }
    return S;
}
```

```
define i32 @main(){
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i32
  %4 = alloca i32
  store i32 0, ptr %1
  %5 = call i32 @getint()
  store i32 %5, ptr %2
  store i32 0, ptr %3
  store i32 0, ptr %4
  br label %6
6:
  %7 = load i32, ptr %3
  %8 = load i32, ptr %2
  %9 = icmp slt i32 %7, %8
  br i1 %9, label %10, label %21
10:
  %11 = load i32, ptr %3
  %12 = icmp eq i32 %11, 10
  br i1 %12, label %13, label %16
13:
  %14 = load i32, ptr %4
  %15 = mul i32 %14, 2
  store i32 %15, ptr %4
  br label %16
16:
  %17 = load i32, ptr %4
  %18 = add i32 %17, 2
  store i32 %18, ptr %4
  %19 = load i32, ptr %3
  %20 = add i32 %19, 1
  store i32 %20, ptr %3
  br label %6
21:
  %22 = load i32, ptr %4
  ret i32 %22
}
```

```
define i32 @main() {
  %1 = call i32 @getint()
  br label %2
2:
  %3 = phi i32 [ 0, %0 ], [ %13, %10 ]
  %4 = phi i32 [ 0, %0 ], [ %12, %10 ]
  %5 = icmp slt i32 %3, %1
  br i1 %5, label %6, label %14
6:
  %7 = icmp eq i32 %3, 10
  br i1 %7, label %8, label %10
8:
  %9 = mul i32 %4, 2
  br label %10
10:
  %11 = phi i32 [ %9, %8 ], [ %4, %6 ]
  %12 = add i32 %11, 2
  %13 = add i32 %3, 1
  br label %2
14:
  ret i32 %4
}
```

3 进阶要求其他优化实验流程

在 Mem2Reg 中，你已经实现了支配树 (AnalysisPass) 以及 mem2reg(TransformPass)，下面你还需要实现更多的优化。这一部分属于进阶要求，我们不会提供详细的指导，但是会提供大致做法以及一些参考文献，需要同学们自行阅读并探索如何实现。

3.1 AnalysisPass

AliasAnalysis 指针分析，你需要实现函数 QueryAlias(Operand op1,Operand op2,...)，返回指针 op1 和 op2 指向的内存是否可能会重叠；以及 QueryInstModRef(Instruction I, Operand op)，返回指令 I 是否会读/写指针 op 指向的内存，由于 SysY 唯一会产生 ptr 类型的只有数组，所以不需要复杂的算法。

LoopAnalysis 循环分析，查找循环，建立循环森林等。该 Pass 网上参考资料较多，大家可以自行搜索相关资料。

ScalarEvolution 推荐参考Scalar Evolution:Change in the Value of Scalar Variables Over Iterations of the Loop

MemorySSA 内存依赖分析，推荐参考llvm 的 MemorySSA 文档，同时也有一些等价的实现方法，大家可以自行探索。

LoopCarriedDependencyAnalysis 推荐参考Dependence Analysis and Loop Transformations。

3.2 TransformPass

DCE 普通的死代码消除，算法可参考该博客。

TCO 尾调用优化，你只需要实现尾递归转循环这一优化即可。该 Pass 算法较简单，大家可以自行设计算法来实现这一优化。

Inline 函数内联以及递归调用展开。对于递归调用展开，可以设置一个阈值，例如函数超过 200 行指令就不再展开。该 Pass 算法较简单，大家可以自行设计算法来实现这一优化。

SCCP 稀疏条件常量传播，推荐参考Static Single Assignment Book的 Part II，第 8 节。

ADCE 激进的死代码消除，算法可参考该博客。

Fully Redundancy Elimination(Scalar) 标量运算的完全冗余消除，你首先需要实现判断两条指令是否计算了相同值的函数，然后通过在支配树上进行一次 dfs 来消除完全冗余，可以参考 llvm/Transforms/Scalar/EarlyCSE.cpp的注释进行编写

Fully Redundancy Elimination in BasicBlock(Memory+Call) 基本块内对内存操作和纯函数调用的完全冗余消除，需要完成的前置 Pass 可能有 AliasAnalysis。实现前置 Pass 后，具体冗余消除步骤较为简单，需要大家自行探索。

Fully Redundancy Elimination(Memory) 内存操作的完全冗余消除，需要完成的前置 Pass 可能有 AliasAnalysis 和 MemorySSA。可以参考 llvm/Transforms/Scalar/EarlyCSE.cpp的注释进行编写。

GCM 原论文为Global Code Motion/Global Value Numbering

SSAPRE 原论文为A new algorithm for partial redundancy elimination based on SSA form

LoopStrengthReduce 循环归纳变量强度削弱，你需要实现 add, mul 运算和 GEP(getelementptr) 运算的归纳变量强度削弱，需要完成的前置 Pass 可能有 LoopAnalysis，ScalarEvolution，LoopSimplify。对于 GEP 指令，后面有一个示例可供参考。

LICM(Scalar) 标量运算的循环不变量外提。需要完成的前置 Pass 可能有 LoopAnalysis。该优化网上参考资料较多，大家可以自行搜索相关资料。

LICM(Memory+Call) 内存操作和纯函数调用的循环不变量外提，推荐阅读 llvm 源代码的 lib/Transforms/Scalar/LICM.cpp注释来了解大致算法。需要完成的前置 Pass 可能有 LoopAnalysis，AliasAnalysis。

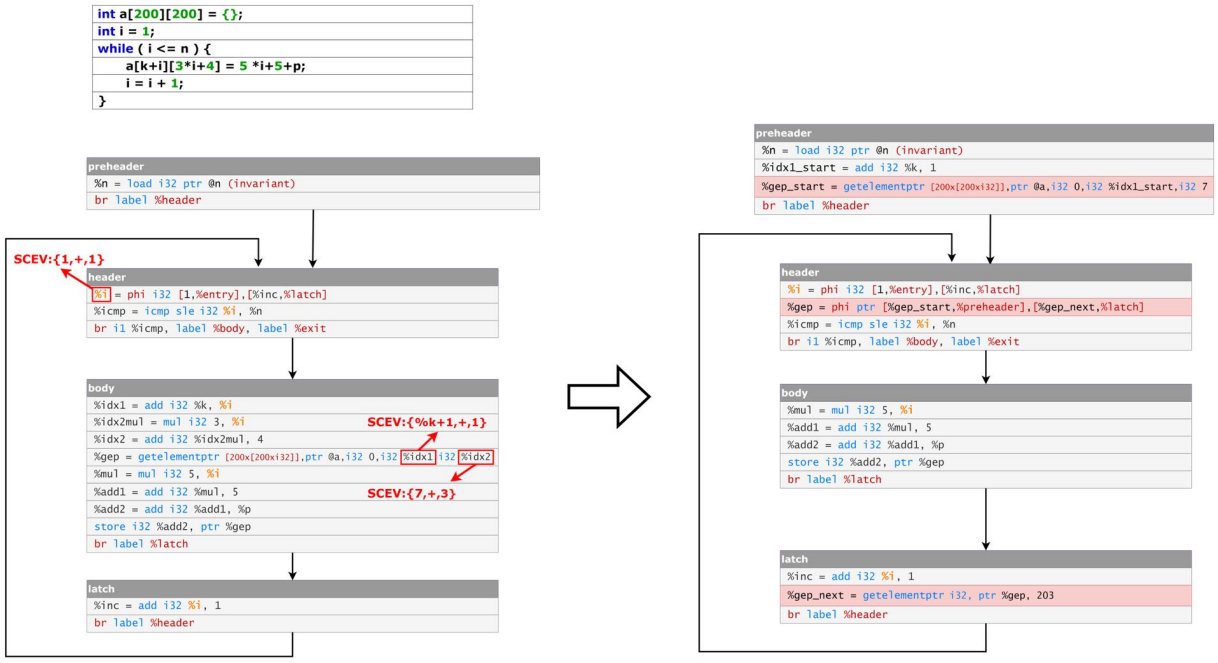

图 3.3: 循环强度削弱示例

LoopUnroll 循环展开，对于常量次数的循环，如果该常量较小，需要将循环完全展开。对于执行次数为变量的循环，如果循环步长为常量（为了实现简单，可以只考虑步长为 1 的情况），需要将循环展开为 4 份 + 不被 4 整除的剩余部分的形式。需要完成的前置 Pass 可能有 LoopAnalysis，ScalarEvolution，LoopSimplify，LCSSA。完成前置 Pass 后，对循环的变换只是工作繁琐，但是思路较简单，需要同学们自行探索具体实现方法。

LoopFuse 循环合并，推荐参考Dependence Analysis and Loop Transformations和Revisiting Loop Fusion and its place in the loop transformation framework。需要完成的前置 Pass 可能有 LoopAnalysis，ScalarEvolution，AliasAnalysis，LoopCarriedDependencyAnalysis，LoopSimplify，LCSSA。

LoopParallel 循环自动并行化，推荐参考Dependence Analysis and Loop Transformations。需要完成的前置 Pass 可能有 LoopAnalysis，ScalarEvolution，AliasAnalysis，LoopCarriedDependencyAnalysis，LoopSimplify，LCSSA。你可能还需要使用 pthread 或者系统调用编写自己的多线程链接库。

Vectorize 自动向量化，推荐参考llvm 自动向量化文档和SLP 矢量化介绍来了解大致流程，然后根据你的理解自己调研一些论文并进行实现。