必做部分实验报告

基于Driver的循环分析器

实验要求

实现一个基于driver的分析器，使其能够 统计给定Function中的循环信息：

- 并列循环的数量
- 每个循环的深度信息

算法实现

> 核心思想

将控制流图分解为若干个极大强连通分量，每个强连通分量代表一个外层循环（包含内部的嵌套循环）。对每个极大强连通分量，找到他的base结点，并从控制流图上删去，相当于脱去一层循环。然后对剩下的结点，递归地分解强连通分量，可得内层循环信息 。

> 具体实现

数据结构：

```
struct CFGNode  //控制流图的结点
{
    std::unordered_set<CFGNode *> succs;  //后继结点集合
    std::unordered_set<CFGNode *> prevs;  //前驱结点集合
    BasicBlock *bb;                 //基本快
    int index;   // the index of the node in CFG
    int lowlink; // the min index of the node in the strongly connected
componets //一个用于计算强联通分量的量
    bool onStack; //用于计算强连通分量
};

using CFGNodePtr = CFGNode*;
using CFGNodePtrSet = std::set<CFGNode*>;  // 可以 存储强连通分量的结点
```

算法：

- 首先创建控制流图 。`void build_cfg(Function &func, std::set<CFGNode *> &result)` 传入`Function`. 根据`BasicBlocks` 间的继承关系，创建控制流图，维护先驱后继结点 。

- 分解强连通分量 `bool strongly_connected_components(CFGNodePtrSet &nodes,std::set<CFGNodePtrSet *> &result)` ，传入控制流图 。分解强连通分量，用的是 `Tarjin` 算法。该算法的时间复杂度较好，只有O(V+E) 。
- 根据分解出的强连通分量，递归的分析循环 `void rec_analyse_loop(CFGNodePtrSet &result,CFGNodePtrSet reserved,int depth, std::vector<int> ith)` 第一个参数是，控制流图，第二个可用于扩展，可以存循环的base 结点 。第三个为深度 ， 便于递归的时候记录当前层数,第四个ith ,记录当前循环是，当前层的第几个循环。重点解释：

```
void rec_analyse_loop(CFGNodePtrSet &result,CFGNodePtrSet reserved,int
depth,std::vector<int> ith){
    std::list<CFGNodePtrSet *> sccs;
    if(strongly_connected_components(result,sccs)){  //如果有强连通分量
         depth++ ;                    //递归一次，深度加一
         int this_ith = 1;            //并列数初始化为1 ，
         ith.insert(ith.begin()+depth-1,0);    // 并列次序插入容器
        for(auto scc: sccs){     // 对每一个强连通分量迭代 ， 一个强连通分量代表一个
并列的 循环
            ith[depth-1] = this_ith++ ;    //并列数加一
            print_depth_space(depth,ith);        //打印当前的深度，和并列关系
            auto base = find_loop_base(scc, reserved);  //返回该分量的 base
结点（循环入口）
            reserved.insert(base);         //保存base
            scc->erase(base);              //在当前的强连通分量里 删去base ， 为了
分析内层循环
            for (auto su : base->succs)
            {
             su->prevs.erase(base);
            }
            for (auto prev : base->prevs)
            {
             prev->succs.erase(base);
            }                              //至此是 ， 删去 base的入边和出边。
            for (auto node : scc){         //还原 ， CFG结点 ， 为了递归分析
                node->index = node->lowlink = -1;
                node->onStack = false;
                // node->prevs.clear();
                // node->succs.clear();
            }
            rec_analyse_loop( *scc, reserved,depth,ith) ;  //递归进入
        }
    }else
    {
        return ;  //无强联通分量
    }
    for (auto scc : sccs)  //释放内存
        delete scc;
    sccs.clear();

}
```

测试样例

1. 对 `loop_z1.c` 是一个多个函数的例子

```
int main(){
    int num = 0 ,i,j,k;
    for(int i=0;i<10;i++){
        for(int j=i;j<10;j++){
            num++;
        }
    }
    for(int j=i;j<10;j++){
        for(int k=j;k<10;k++){
            num++;
        }
    }

    return num;
}
int fun(){
    int num;
    while(1){
        for(int i = 0;i<3 ; i++){
            if(num == i){
                for(int j=i;j<10;j++){
                    for(int k=j;k<10;k++){
                        num++;
                    }
                 }
            }
            else
            {
                int j;
                for(int k=j;k<10;k++){
                        num++;
                }
            }
        }
    }
}
```

分析结果的输出如下:

```
Processing function  main, loop message is as follow
{
        L1  depth:1
                L11  depth:2
        L2  depth:1
                L21  depth:2
}
Processing function  fun, loop message is as follow
{
        L1  depth:1
                L11  depth:2
                        L111  depth:3
                                L1111  depth:4
                        L112  depth:3
}
```

2. 对`loop.c`，是一个单个函数的例子

```
int main(){
    int num = 0 ,i,j,k;
    for(int i=0;i<10;i++){
        for(int j=i;j<10;j++){
            num++;
        }
    }
    for(int j=i;j<10;j++){
            for(int k=j;k<10;k++){
                num++;
            }
        }
    for(int k=j;k<10;k++){
                num++;
     }
    for(int i=0;i<10;i++){
        for(int j=i;j<10;j++){
            for(int k=j;k<10;k++){
                num++;
            }
        }
        for(int j=i;j<10;j++){
            for(int k=j;k<10;k++){
                num++;
            }
        }
    }
    return num;
}
```

分析结果的输出如下:

```
Processing function  main, loop message is as follow
{
        L1  depth:1
                L11  depth:2
        L2  depth:1
                L21  depth:2
        L3  depth:1
        L4  depth:1
                L41  depth:2
                        L411  depth:3
                L42  depth:2
                        L421  depth:3
}
```

实验总结

1. 用强连通分量，分析循环是一种直观的方法，且采用Tarjin算法分解强连通分量的时间复杂度并不高只有O(V+E)。但是在递归分析循环的时候，需要在去掉base结点后重新分解强连通分量，实则是一种浪费。
2. 本实验，在`BasicBlock`层面的控制流图上，又包装了一层`CFGNode`，然后在`CFGNode`的层面上再创建控制流图，这是为了给 分解强连通分量提供一些信息，`int index;`,`int`

`lowlink;`,`bool onStack`

思考题

请解释FindFunctionBackedges函数中InStack变量的物理意义（例如Visited变量的物理意义为存储已访问的BB块集合、VisitStack变量的物理意义为栈中待处理的边集合）

```
/// FindFunctionBackedges - Analyze the specified function to find all of the
/// loop backedges in the function and return them.  This is a relatively
cheap
/// (compared to computing dominators and loop info) analysis.
///
/// The output is added to Result, as pairs of <from,to> edge info.
void llvm::FindFunctionBackedges(const Function &F,
     SmallVectorImpl<std::pair<const BasicBlock*,const BasicBlock*> > &Result)
{
  const BasicBlock *BB = &F.getEntryBlock();     //获得函数入口BB
  if (succ_empty(BB))                            //若没有后继BB
    return;

  SmallPtrSet<const BasicBlock*, 8> Visited;     //已访问的BB块集合
  SmallVector<std::pair<const BasicBlock *, const_succ_iterator>, 8>
VisitStack;     //栈，里面是待处理的边集合
  SmallPtrSet<const BasicBlock*, 8> InStack;

  Visited.insert(BB);    //标记当前(入口)BB已访问
  VisitStack.push_back(std::make_pair(BB, succ_begin(BB))); //将BB引出的第一条边加
入到待处理集合
  InStack.insert(BB);        //记录当前BB
  do {
    std::pair<const BasicBlock *, const_succ_iterator> &Top =
VisitStack.back();     //栈顶的边出栈
    const BasicBlock *ParentBB = Top.first;      //ParentBB为引出该边的BB
    const_succ_iterator &I = Top.second;         //I为该边

    bool FoundNew = false;  //标记
    while (I != succ_end(ParentBB)) {    //遍历ParentBB的所有出边
      BB = *I++;          //记录当前的后继BB，并更新迭代器I
      if (Visited.insert(BB).second) {  //insert的声明类型: std::pair<iterator,
bool> insert(PtrType Ptr)，这里即判断是否插入成功，成功证明该基本块未访问过
        FoundNew = true;     //发现新的基本块
        break;
      }
      // Successor is in VisitStack, it's a back edge.
      // count - Return 1 if the specified pointer is in the set, 0 otherwise.
      //当前BB==succ_end（ParentBB）
      if (InStack.count(BB))
        Result.push_back(std::make_pair(ParentBB, BB)); //记录发现的回边
    }

    if (FoundNew) {
      // Go down one level if there is a unvisited successor.
      InStack.insert(BB);    //记录当前BB
      VisitStack.push_back(std::make_pair(BB, succ_begin(BB))); //标记当前边待处
理
```

```
    } else {
      // Go up one level 回溯
      InStack.erase(VisitStack.pop_back_val().first);    //从InStack中删除对应部分
    }
  } while (!VisitStack.empty());
}
```

具体代码解析如上所述

`InStack`变量的物理意义：用于记录**深度优先搜索**路径上的结点(基本块)的栈，基于`InStack`的信息来寻找回边