STAGE7 REPORT: SSA

致理-数理 1 赵宇峰 2021013376

本实验主要希望实现 Mem2Reg 技术，从而实现静态单赋值，为进一步优化做准备。本实现主要改动 TACFunc 类，完成了构建支撑树—计算插入位置并插入—重命名步骤。

1. 总体流程

   在原本的 TACFunc visitEnd 的位置完成全部流程:

```
def visitEnd(self) -> TACFunc:
    if (len(self.func.instrSeq) == 0) or (not self.func.instrSeq[-1].isReturn()):
        self.func.add(Return(None))
    self.func.tempUsed = self.getUsedTemp()
    self.labelManager.funcs.append(self.func)
    self.func.generate_cfg()//建立 cfg
    self.func.generate_DomT()//建立支撑树
    self.func.insertPhi()//插入 phi 指令，依据要求修改所有指令
return self.func
```

2 分步实现

首先需要更改所有赋值语句，改为 Alloc，Store, Load 的模式。这里主要展示一下 visitAssignment 和 visitBinary，其他类似:

```
def visitAssignment(self, dst: Temp, src: Temp) -> Temp:
        if not self.allocDict.get(dst):#如果没有分配内存
            self.func.add(Alloc(dst))#先分配
            self.allocDict[dst] = True#记录所有分配情况
        if self.allocDict.get(src):
            temp = self.freshTemp()
            self.func.add(LoadWord(temp, src))
            self.func.add(StoreWord(dst, temp))
        self.func.add(StoreWord(dst, src))
    return src



    def visitBinary(self, op: BinaryOp, lhs: Temp, rhs: Temp) -> Temp:
        if self.allocDict.get(lhs) == None:
            if self.allocDict.get(rhs) == None:
                temp = self.freshTemp()
                self.func.add(Binary(op, temp, lhs, rhs))
                return temp
            else:
                rhs_temp = self.freshTemp()
                self.func.add(LoadWord(rhs_temp,rhs))
                temp = self.freshTemp()
                self.func.add(Binary(op, temp, lhs, rhs_temp))
```

```
                return temp
        else:
            if self.allocDict.get(rhs) == None:
                lhs_temp = self.freshTemp()
                self.func.add(LoadWord(lhs_temp,lhs))
                temp = self.freshTemp()
                self.func.add(Binary(op, temp, lhs_temp, rhs))
                return temp
            else:
                result_temp = self.freshTemp()
                lhs_temp = self.freshTemp()
                rhs_temp = self.freshTemp()
                self.func.add(LoadWord(lhs_temp,lhs))
                self.func.add(LoadWord(rhs_temp,rhs))
                self.func.add(Binary(op, result_temp, lhs_temp, rhs_temp))
            return result_temp
```

主要是从内存读数据一定要先 Load，写内存数据一定要后 Store.

TACInstr 中也增加了 Alloc,LoadWord,StoreWord 和 PhiFunc,没有比较特殊的地方，只有 PhiFunc 稍微特别，需要传入一个字典结构来实现多种选择的特性。

```
class PhiFunc(TACInstr):
    def __init__(self, result: Temp, relations: {}) -> None:
        super().__init__(InstrKind.SEQ, [result], [], None)
        self.result = result
        self.operands = []
        self.relations = relations

    def __str__(self) -> str:
        operand_str = ", ".join([f"{value},{id}" for id, value in self.relations.items()])
        return f"{self.result} = PHI({operand_str})"

    def accept(self, v: TACVisitor) -> None:
        v.visitPhiFunc(self)
```

现在开始写正式的流程：

```
def generate_cfg(self) -> None:
    self.cfg = self.builder.buildFrom(self.instrSeq)
```

这里就直接用了后端的 cfgbuilder

```
def generate_DomT(self) -> None:
    self.DomTree = DominatorTree(self.cfg)
```

由于 DomTree 支配树就是 cfg 的一个解释，所以在 cfg 的 python 程序中同时增加了 DominatorTree 类，并计算支配边界。

```
class DominatorTree:
    def __init__(self, cfg: CFG):
        self.cfg = cfg
        self.num_nodes = len(cfg)
        self.original_ids = [node.id for node in self.cfg.nodes.values()] #由于一些寄存器是冗余的，最后剩下的寄存器的 id 是断开的，这里避免麻烦，就采用了映射的办法
        self.id_translation = self.translate_ids(self.original_ids)
        self.back_translation = {v: k for k, v in self.id_translation.items()}
        self.back_translation[-1] = -1
        self.idoms = self.compute_dominator_tree()#每一个的直接支配者
        self.domFrontier = self.calculate_dominance_frontier()


    def successors(self, v):
        original_successors = self.cfg.getSucc(self.back_translation[v])
        translated_successors = [self.id_translation[s] for s in original_successors]
        return translated_successors

    def predecessors(self, v):
        original_predecessors = self.cfg.getPrev(self.back_translation[v])
        translated_predecessors = [self.id_translation[p] for p in original_predecessors]
        return translated_predecessors

    def translate_ids(self, original_ids):
        id_mapping = {original_id: i for i, original_id in enumerate(original_ids)}
        return id_mapping

    def translate_back(self, translated_ids):
        return {v: k for k, v in translated_ids.items()}

    def compute_dominator_tree(self):
    #这里是Lengauer-Tarjan的较为直接的实现,来自Modern compiler implementation in java
        # Translate node IDs to a contiguous range
        N = 0

        bucket = [set() for _ in range(self.num_nodes)]
        dfnum = [0] * self.num_nodes
        vertex = [-1] * self.num_nodes
        parent = [-1] * self.num_nodes
        semi = [-1] * self.num_nodes
        ancestor = [-1] * self.num_nodes
        idom = [-1] * self.num_nodes
        samedom = [-1] * self.num_nodes
```

```
best = [-1] * self.num_nodes

def dfs():
    nonlocal N
    stack = [(-1, 0)]
    while stack:
        p, n = stack.pop()
        if dfnum[n] == 0:
            dfnum[n] = N
            vertex[N] = n
            parent[n] = p
            N += 1

            for w in self.successors(n):
                stack.append((n, w))

def ancestor_with_lowest_semi(v):
    a = ancestor[v]
    if ancestor[a] >= 0:
        b = ancestor_with_lowest_semi(a)
        ancestor[v] = ancestor[a]
        if dfnum[semi[b]] < dfnum[semi[best[v]]]:
            best[v] = b
    return best[v]

def link(p, n):
    ancestor[n] = p
    best[n] = n

dfs()

for i in range(N - 1, 0, -1):
    n = vertex[i]
    p = parent[n]
    s = p

    for v in self.predecessors(n):
        s_prime    =    v    if    dfnum[v]    <=    dfnum[n]    else
                            semi[ancestor_with_lowest_semi(v)]
        if dfnum[s_prime] < dfnum[s]:
            s = s_prime

    semi[n] = s
    bucket[s] |= {n}
```

```
                link(p, n)

                for v in bucket[p]:
                    y = ancestor_with_lowest_semi(v)
                    if semi[y] == semi[v]:
                        idom[v] = p
                    else:
                        samedom[v] = y
                bucket[p] = set()

            for i in range(1, N):
                n = vertex[i]
                if samedom[n] >= 0:
                    idom[n] = idom[samedom[n]]

            # Translate back to the original node IDs
            idom_translation = {self.back_translation[a]:self.back_translation[idom[a]] for a
in range(len(idom))}
            return idom_translation

    #这里是 DF 算法的一种实现，来自 Engineering a Compiler:
        def calculate_dominance_frontier(self):
            dominance_frontier = {node: set() for node in self.idoms}

            for node, idom_node in self.idoms.items():
                if idom_node != -1:
                    for pred in self.cfg.getPrev(node):
                        runner = pred
                        while runner != idom_node:
                            dominance_frontier[runner].add(node)
                            runner = self.idoms[runner]

            return dominance_frontier
```

完成这两步后，就可以计算 phi 函数的插入位置了。这里就直接用了提供的 SSA 教材的算法

Algorithm 3.1: Standard algorithm for inserting ϕ-functions

for v: variable names in original program **do**
    $F \leftarrow \{\}$ ▷ *set of basic blocks where ϕ is added*
    $W \leftarrow \{\}$ ▷ *set of basic blocks that contain definitions of v*
    for $d \in \text{Defs}(v)$ **do**
        let B be the basic block containing d
        $W \leftarrow W \cup \{B\}$
    while $W \neq \{\}$ **do**
        remove a basic block X from W
        for Y: basic block $\in$ **DF**(X) **do**
            if $Y \notin F$ **then**
                add $v \leftarrow \phi(\ldots)$ at entry of Y
                $F \leftarrow F \cup \{Y\}$
                if $Y \notin \text{Defs}(v)$ **then**
                    $W \leftarrow W \cup \{Y\}$

```
def getdefs(self) -> None:#通过 Store 命令找到所有定值点
    for instr in self.instrSeq:
        if isinstance(instr, StoreWord) or isinstance(instr, StoreImm4):
            self.variables.setdefault(instr.dst, []).append(instr)
def insertPhi(self) -> None:
    self.getdefs()
    for v in self.variables.keys():
        fset = set()
        wset = set()
        phi_dict = {}
        WorkList = set()
        for bbid in self.cfg.nodes.keys():
            for v_instr in self.variables[v]:
                if self.cfg.getBlock(bbid).hasInstr(v_instr):
                    wset.add(bbid)
                    WorkList.add(bbid)
                    phi_dict[bbid] = v_instr.src

        while len(wset) > 0:
            blockx_id = wset.pop()
            for blocky_id in self.DomTree.domFrontier[blockx_id ]:
                if not blocky_id in fset:
                    prevs = self.cfg.get_all_Prev(blocky_id)
                    mydict = {}
                    for intb in phi_dict.keys():
                        if intb in prevs:
```

```
                mydict[intb] = phi_dict[intb]
        v_phi_func = PhiFunc(v,mydict)
        if len(mydict) > 0:
            self.cfg.getBlock(blocky_id).locs.insert(0,Loc(v_phi_func))
        fset.add(blocky_id)
        if not blocky_id in WorkList:
            wset.add(blocky_id)
```

需要解释的是，这里实现 Phi 函数时只包含了所有可能到达这个块的定义，所以用 dfs 实现了 get_all_Prev，但在最后的部分测例上还是多了一些并不需要 Phi 语句，由于时间有限没有做删除.

最后就是比较核心的重命名阶段：

Algorithm 3.3: Renaming algorithm for second phase of SSA construction

```
▷ rename variable definitions and uses to have one definition per variable name
foreach v : Variable do
    v.reachingDef ← ⊥
foreach BB: basic Block in depth-first search preorder traversal of the dom. tree do
    foreach i : instruction in linear code sequence of BB do
        foreach v : variable used by non-φ-function i do
            updateReachingDef(v, i)
            replace this use of v by v.reachingDef in i
        foreach v : variable defined by i (may be a φ-function) do
            updateReachingDef(v, i)
            create fresh variable v′
            replace this definition of v by v′ in i
            v′.reachingDef ← v.reachingDef
            v.reachingDef ← v′
    foreach φ: φ-function in a successor of BB do
        foreach v : variable used by φ do
            updateReachingDef(v, φ)
            replace this use of v by v.reachingDef in φ
```

```
#Rename Period
reaching_defs = {}
for v in self.variables.keys():
    reaching_defs[v] = []
    #为了方便，给每一个变量加了一个 list 来存所有可能的定义。
def updateReachingDef(var,bb_id):
    for def_obj in reversed(reaching_defs[var]):
        if def_obj[0] == bb_id:
            return def_obj[1]
        parent = self.DomTree.idoms[bb_id]#验证支配关系，避免与本块无关的定义
        while parent != -1:
```

```
                if def_obj[0] == parent:
                    return def_obj[1]
                parent = self.DomTree.idoms[parent]
        return None
#删去 Alloc,Load,Store,维护 phi,实现 mem2reg:
def changephi(pbb:int):
    children_id = [child for child, parent in self.DomTree.idoms.items() if parent == pbb]
    bb = self.cfg.getBlock(pbb)
    loc_rank = 0
    while(loc_rank < len(bb.locs)):
        loc = bb.locs[loc_rank]
        if isinstance(loc.instr,Alloc):
            bb.locs.remove(loc)
            loc_rank = loc_rank - 1

        #A. If instruction is a load instruction from location L (where L is a promotable candidate) to value V
        #    delete load instruction, replace all uses of V with most recent value of L i.e, IncomingVals[L].
        #所有的 load 都是使用点。所以，删去 load 的同时，将该基本块中所有对 load 出来的值的使用改为对到达定义的使用
        if isinstance(loc.instr,LoadWord):
            reaching_def_load = updateReachingDef(loc.instr.src,bb.id)
            for loc_rest_rank in range(loc_rank+1,len(bb.locs)):
                rest_loc = bb.locs[loc_rest_rank]
                if loc.instr.dst in rest_loc.instr.srcs:
                    for srcs_rank in range(0,len(rest_loc.instr.srcs)):
                        if(rest_loc.instr.srcs[srcs_rank].index == loc.instr.dst.index):
                            rest_loc.instr.srcs[srcs_rank].index = reaching_def_load.index
                elif isinstance(rest_loc.instr,Return):#return 的结构不太一样，这里分开讨论。
                    if(rest_loc.instr.value.index == loc.instr.dst.index):
                        rest_loc.instr.value.index = reaching_def_load.index
            bb.locs.remove(loc)
            loc_rank = loc_rank - 1

        #B. If instruction is a store instruction to location L (where L is a promotable candidate) with value V,
        #    delete store instruction, set most recent name of L i.e, IncomingVals[L] = V.
        #对于 storeword 主要就是更新到达定义
        if isinstance(loc.instr,StoreWord):
            for var in self.variables.keys():
```

```
                    if loc.instr.dst == var:
                        assert loc.instr.src is not None
                        reaching_defs[var].append([bb.id,loc.instr.src])
                bb.locs.remove(loc)
                loc_rank = loc_rank - 1



            loc_rank += 1
```

#C. For each PHI-node corresponding to a alloca — L , in each successor of B, #   fill the corresponding PHI-node argument with most recent name for that location i.e, IncomingVals[L]

```
                for suc_bb_id in self.cfg.getSucc(pbb):
                    for sub_loc in self.cfg.getBlock(suc_bb_id).locs:
                        if isinstance(sub_loc.instr,PhiFunc):
                            for r_id in sub_loc.instr.relations.keys():
                                    sub_loc.instr.relations[r_id].index ==
reaching_defs.get(sub_loc.instr.relations[r_id],sub_loc.instr.relations[r_id]).index




        for child_id in children_id:
            changephi(child_id)

changephi(0)
```

#为了方便打印，这里一并删去了原来的 basicblock label,替换成 basicblock id 的形式

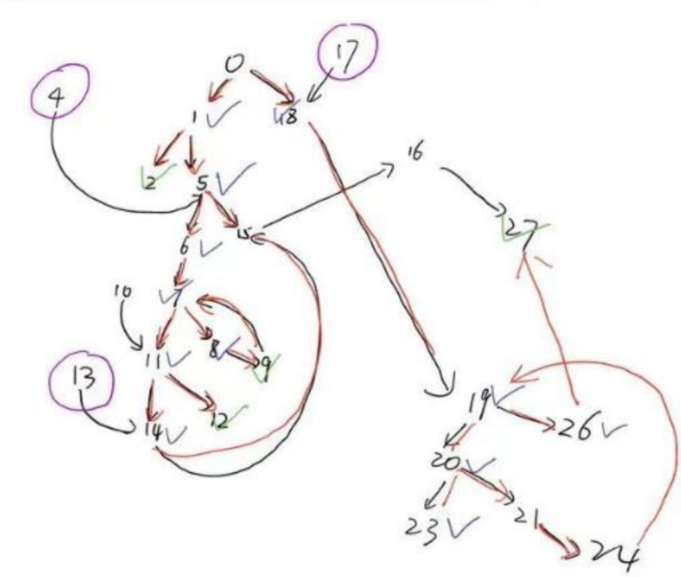
```
for bb in self.cfg.nodes.values():
        if not bb.id in self.cfg.reachable:
            pass
        bb_label_int = BBLabel(bb.id)
        bb_label_int_loc=Loc(bb_label_int)
        self.new_instr_Seq.append(bb_label_int_loc)
        for loc in bb.locs:
            if isinstance(loc.instr,Branch):
                new_target =
Label(InstrKind.JMP,str(self.builder.labelsToBBs[loc.instr.target]))
                loc.instr.target = new_target
            if isinstance(loc.instr,CondBranch):
                new_target =
Label(InstrKind.COND_JMP,str(self.builder.labelsToBBs[loc.instr.target]))
                loc.instr.target = new_target
```

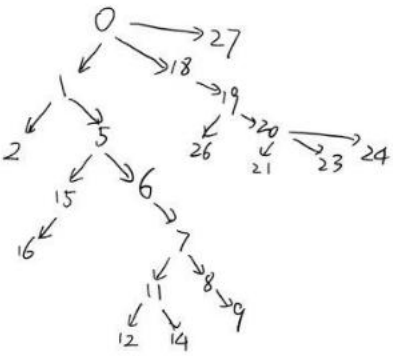
```
        self.new_instr_Seq.append(loc)
```

本次实验主要采用了一个复杂的测例：

```
int main() {
    int a = 1;
    int b = 2;
    if (a + b > 0){
        if (a - b > 0){
            a = a + b;
            return a;
        }
        else if (a - b < 0){
            for (int i = 2; i <= 12; i = i + 1){
                b = a - (i + b);
            }
            if ((b - a) >= (a - b + 3)){
                b = b - a * 2;
                a = b - a * 3;
                return (b - 2 + a * 2);
            }
        }
    }
    else{
        while(a > b){
            if((a - b) > (1 - a))
                continue;
            a = -b - a;
            b = a - b;
        }
    }
    return a-b;
}
```

CFG 和支配树分别如下，并和程序输出结果一致（略去验证图）





可以看到删去了一些不可到达块，有一定的优化作用。
插入 load,store,alloc 的形式如下：

```
FUNCTION<main>:
    _T1 = 1
    _T0 = ALLOC 4
    STORE _T0 _T1
    _T3 = 2
    _T2 = ALLOC 4
    STORE _T2 _T3
    LOAD _T1 _T0
    LOAD _T3 _T2
    _T4 = (_T1 + _T3)
    _T7 = 0
    _T8 = (_T4 > _T7)
    if (_T8 == 0) branch 18
    LOAD _T1 _T0
    LOAD _T3 _T2
    _T9 = (_T1 - _T3)
    _T12 = 0
    _T13 = (_T9 > _T12)
```

```
        if (_T13 == 0) branch 5
        LOAD _T1 _T0
        LOAD _T3 _T2
        _T14 = (_T1 + _T3)
        STORE _T0 _T14
        LOAD _T14 _T0
        return _T14
        branch _L5
_L4:
        LOAD _T1 _T0
        LOAD _T3 _T2
        _T18 = (_T1 - _T3)
        _T21 = 0
        _T22 = (_T18 < _T21)
        if (_T22 == 0) branch 15
        _T24 = 2
        _T23 = ALLOC 4
        STORE _T23 _T24
_L7:
        _T25 = 12
        LOAD _T24 _T23
        _T27 = (_T24 <= _T25)
        if (_T27 == 0) branch 11
        LOAD _T24 _T23
        LOAD _T3 _T2
        _T28 = (_T24 + _T3)
        LOAD _T1 _T0
        _T32 = (_T1 - _T28)
        STORE _T2 _T32
_L8:
        _T33 = 1
        LOAD _T24 _T23
        _T35 = (_T24 + _T33)
        STORE _T23 _T35
        branch 7
_L9:
        LOAD _T3 _T2
        LOAD _T1 _T0
        _T36 = (_T3 - _T1)
        LOAD _T1 _T0
        LOAD _T3 _T2
        _T39 = (_T1 - _T3)
        _T42 = 3
        _T43 = (_T39 + _T42)
```

```
	_T44 = (_T36 >= _T43)
	if (_T44 == 0) branch 14
	_T45 = 2
	LOAD _T1 _T0
	_T47 = (_T1 * _T45)
	LOAD _T3 _T2
	_T49 = (_T3 - _T47)
	STORE _T2 _T49
	_T50 = 3
	LOAD _T1 _T0
	_T52 = (_T1 * _T50)
	LOAD _T49 _T2
	_T54 = (_T49 - _T52)
	STORE _T0 _T54
	_T55 = 2
	LOAD _T49 _T2
	_T57 = (_T49 - _T55)
	_T58 = 2
	LOAD _T54 _T0
	_T60 = (_T54 * _T58)
	_T61 = (_T57 + _T60)
	return _T61
_L10:
_L6:
_L5:
	branch 27
_L2:
_L11:
	LOAD _T1 _T0
	LOAD _T3 _T2
	_T62 = (_T1 > _T3)
	if (_T62 == 0) branch 26
	LOAD _T1 _T0
	LOAD _T3 _T2
	_T65 = (_T1 - _T3)
	_T68 = 1
	LOAD _T1 _T0
	_T70 = (_T68 - _T1)
	_T71 = (_T65 > _T70)
	if (_T71 == 0) branch 23
	branch 24
_L14:
	LOAD _T3 _T2
	_T72 = - _T3
```

```
	LOAD _T1 _T0
	_T75 = (_T72 - _T1)
	STORE _T0 _T75
	LOAD _T75 _T0
	LOAD _T3 _T2
	_T76 = (_T75 - _T3)
	STORE _T2 _T76
_L12:
	branch 19
_L13:
_L3:
	LOAD _T1 _T0
	LOAD _T3 _T2
	_T79 = (_T1 - _T3)
	return _T79
```

而在 Mem2Reg 后如下：（事实证明 get_all_prev 而不是 getPrev 是必要的，否则会出错）

```
-----AFTER MEM2REG-----
BLOCK0:
	_T1 = 1
	_T3 = 2
	_T4 = (_T1 + _T3)//3
	_T7 = 0
	_T8 = (_T4 > _T7)//1
	if (_T8 == 0) branch 18
BLOCK1:
	_T9 = (_T1 - _T3)//-1
	_T12 = 0
	_T13 = (_T9 > _T12)//0
	if (_T13 == 0) branch 5
BLOCK2:
	_T14 = (_T1 + _T3)
	return _T14
BLOCK5:
	_T18 = (_T1 - _T3)//-1
	_T21 = 0
	_T22 = (_T18 < _T21)//1
	if (_T22 == 0) branch 15
BLOCK6:
	_T24 = 2
BLOCK7:
	_T23 = PHI(_T24,6, _T24,9)
	_T2 = PHI(_T3,0, _T3,8)//2，3，4，…,11
	_T25 = 12
```

```
        _T27 = (_T23 <= _T25)//1
        if (_T27 == 0) branch 11
BLOCK8:
        _T28 = (_T24 + _T3)//5
        _T32 = (_T1 - _T28)//-4
BLOCK9:
        _T33 = 1
        _T35 = (_T24 + _T33)//6
        branch 7
BLOCK11:
        _T36 = (_T3 - _T1)
        _T39 = (_T1 - _T3)
        _T42 = 3
        _T43 = (_T39 + _T42)
        _T44 = (_T36 >= _T43)
        if (_T44 == 0) branch 14
BLOCK12:
        _T45 = 2
        _T47 = (_T1 * _T45)
        _T49 = (_T3 - _T47)
        _T50 = 3
        _T52 = (_T1 * _T50)
        _T54 = (_T49 - _T52)
        _T55 = 2
        _T57 = (_T49 - _T55)
        _T58 = 2
        _T60 = (_T54 * _T58)
        _T61 = (_T57 + _T60)
        return _T61
BLOCK14:
BLOCK15:
        _T23 = PHI(_T24,6, _T35,9)
        _T2 = PHI(_T3,0, _T3,8)
BLOCK16:
        branch 27
BLOCK18:
BLOCK19:
        _T2 = PHI(_T3,0, _T3,23)
        _T0 = PHI(_T1,0, _T1,23)
        _T62 = (_T1 > _T3)
        if (_T62 == 0) branch 26
BLOCK20:
        _T65 = (_T1 - _T3)
        _T68 = 1
```

```
        _T70 = (_T68 - _T1)
        _T71 = (_T65 > _T70)
        if (_T71 == 0) branch 23
BLOCK21:
        branch 24
BLOCK23:
        _T72 = - _T3
        _T75 = (_T72 - _T1)
        _T76 = (_T75 - _T3)
BLOCK24:
        _T2 = PHI(_T3,0, _T3,23)
        _T0 = PHI(_T1,0, _T1,23)
        branch 19
BLOCK26:
BLOCK27:
        _T23 = PHI(_T24,6, _T35,9)
        _T2 = PHI(_T3,0, _T3,8, _T3,23)
        _T0 = PHI(_T1,0, _T1,23)
        _T79 = (_T1 - _T3)
        return _T79
```

手动计算可得结果准确。