stage-7

计13 沈佳茗 2021010745

一、实验内容

本实验所有内容都在中端进行。

1. 中端

1.1 中间代码生成

首先，为了记录每条 `Alloc/SaveWord/LoadWord` 指令分别对应了那个变量方便在变量重名名阶段处理，在这三个类中新增 `ident` 变量用来记录对应的标识符。

在中间代码生成阶段，修改 `visitReturn` 函数，增加对返回值类型的判断，如果返回值类型是 `Identifier`，就使用 `LoadWord` 指令将该变量载入虚拟寄存器作为返回值，否则直接使用 `getattr("val")` 即可。

```
def visitReturn(self, stmt: Return, mv: TACFuncEmitter) -> None:
    stmt.expr.accept(self, mv)
    val = stmt.expr.getattr("val")
    if isinstance(stmt.expr, Identifier):
        temp = mv.freshTemp()
        mv.visitLoadWord(temp, stmt.expr.getattr('address'), 0, stmt.expr)
        val = temp
    mv.visitReturn(val)
```

在 `visitDeclaration` 函数中，将之前设置为 val 的变量作为 address，使用 `Alloc` 指令为其分配地址。如果 `decl.init_expr` 存在并且为 `Identifier` 类型，则先用 `LoadWord` 指令载入，再使用 `SaveWord` 指令将其保存到分配的地址中；否则直接将 `decl.init_expr.getattr('val')` 保存到分配的地址中。

```
+         decl.setattr("address", symbol.temp)
          decl.setattr("symbol", symbol)
+         decl.ident.setattr("address", symbol.temp)
```

```
         else:
+            mv.visitAlloc(symbol.temp, 4, decl.ident)
             if decl.init_expr is not NULL:
                 decl.init_expr.accept(self, mv)
-                mv.visitAssignment(decl.getattr("symbol").temp,
decl.init_expr.getattr('val'))
+                if isinstance(decl.init_expr, Identifier):
+                    temp = mv.freshTemp()
+                    mv.visitLoadWord(temp, decl.init_expr.getattr('address'), 0,
decl.init_expr)
+                    mv.visitSaveWord(temp, decl.getattr("address"), 0, decl.ident)
+                else:
+                    mv.visitSaveWord(decl.init_expr.getattr('val'),
decl.getattr("address"), 0, decl.ident)
```

在 `visitIdentifier` 函数中，如果 `ident` 是一个局部的 int，就使用 `setattr` 将 address 属性设为 `symbol.temp`

```
-            ident.setattr('val', temp)
+            ident.setattr('address', temp)
```

在 `visitAssignment` 函数中，如果左式是局部变量且不是数组，右式的类型是 `Identifier` 则需要则先用 `LoadWord` 指令载入，再使用 `SaveWord` 指令将其保存到 `lhs` 地址中；否则直接将 `expr.rhs.getattr('val')` 保存到 `lhs` 的地址中。

```
@@ -327,8 +349,16 @@ class TACGen(Visitor[TACFuncEmitter, None]):
                 expr.setattr('val', val)
             else:
                 temp = expr.lhs.getattr('symbol').temp
-                mv.visitAssignment(temp, expr.rhs.getattr('val'))
-                expr.setattr('val', expr.rhs.getattr('val'))
+                if isinstance(expr.rhs, Identifier):
+                    rhs_temp = mv.freshTemp()
+                    mv.visitLoadWord(rhs_temp, expr.rhs.getattr("address"), 0,
expr.rhs)
+                    mv.visitSaveWord(rhs_temp, lhs_address, 0, expr.lhs)
+                    expr.setattr('val', expr.rhs.getattr('val'))
+                else:
+                    mv.visitSaveWord(expr.rhs.getattr('val'), lhs_address, 0,
expr.lhs)
+                    expr.setattr('val', expr.rhs.getattr('val'))
+                # mv.visitAssignment(temp, expr.rhs.getattr('val'))
+                # expr.setattr('val', expr.rhs.getattr('val'))
```

在 `visitUnary` 和 `visitBinary` 中分别判断 `expr.operand`，`expr.lhs`，`expr.rhs` 是否是 Identifier 类，如果是的话则需要用 `LoadWord` 载入，否则直接使用 `getattr("val")` 即可。

```
# visitUnary
+        operand_temp = expr.operand.getattr("val")
+        if isinstance(expr.operand, Identifier):
+            operand_temp = mv.freshTemp()
+            mv.visitLoadWord(operand_temp, expr.operand.getattr("address"), 0,
expr.operand)
+        expr.setattr("val", mv.visitUnary(op, operand_temp))
```

```
# visitBinary
+        lhs_temp = expr.lhs.getattr("val")
+        rhs_temp = expr.rhs.getattr("val")
+        # print("binary", type(expr.lhs), type(expr.rhs))
+        if isinst ance(expr.lhs, Identifier):
+            # print("ident")
+            lhs_temp = mv.freshTemp()
+            mv.visitLoadWord(lhs_temp, expr.lhs.getattr("address"), 0, expr.lhs)
+        if isinstance(expr.rhs, Identifier):
+            rhs_temp = mv.freshTemp()
+            mv.visitLoadWord(rhs_temp, expr.rhs.getattr("address"), 0, expr.rhs)
         expr.setattr(
-            "val", mv.visitBinary(op, expr.lhs.getattr("val"),
expr.rhs.getattr("val"))
+            "val", mv.visitBinary(op, lhs_temp, rhs_temp)
         )
```

在 `transform` 函数中记录每一个函数用到的寄存器数量和 label 数量，方便在之后的过程中可以为新添加的 Phi 指令设置寄存器与为每个块添加一个label。

1.2 mem2reg 转化

依据 $Dom(x) = \{x\} \cup (\cap_{m \in preds(x)} Dom(m))$ 来求解 x 的支配集合。具体实现为循环迭代知道每一个节点的支配集合都不再改变。

```
self.dominates = []
    # 节点本身属于它自己的支配集合
    for i in range(len(self.nodes)):
        self.dominates.append(set())
        self.dominates[i].add(i)

    sum = 0
    # 循环迭代知道所有节点的支配集合都不再变化
    while True:
        sum += 1
        flag = True
        for i, link in enumerate(self.links):
            s = set()
            idx = 0
            # 计算所有前驱节点支配集合的并集再与自己求交
            for pred in link[0]:
                if idx == 0:
                    s = self.dominates[pred]
```

```
                elif idx < i:
                    s = s.intersection(self.dominates[pred])
                idx += 1
            if self.dominates[i] != set.union(s,set({i})):
                flag = False
            self.dominates[i] = set.union(self.dominates[i], s)
        if flag == True:
            break
```

根据直接支配者的定义"严格支配 n，且不严格支配任何严格支配 n 的节点的节点"遍历每个节点的严格支配者，若 n 只有一个严格支配者，则其就为 n 的直接支配者；否则验证 n 的严格支配者 i，其均不严格支配任何严格支配 n 的节点的节点，则 i 为 n 的直接支配者。

```
 # 去除本身，得到严格支配集合
for i, dom in enumerate(self.dominates):
    dom.remove(i)
# print("strict dominates",self.dominates)
self.idoms = []
# 找到严格支配 n（idx），且不严格支配任何严格支配 n（idx） 的节点的节点
for idx, dominate in enumerate(self.dominates):
    idom = None
    for i in dominate:
        if len(dominate) == 1:
            idom = i
        for j in dominate:
            if i != j:
                if i not in self.dominates[j]:
                    idom = i
    self.idoms.append(idom)
```

根据 `self.idoms` 可得到支配树的边，仿照 CFG 中的做法得到 `tree_edges` 和 `tree_links`。

依据 $DF(n) = x \mid n$支配x的前驱节点，n不严格支配x 计算每个结点的支配边界。

```
# 再把 self.dominates 变成支配集合（加上自己）
for i, dom in enumerate(self.dominates):
    dom.add(i)

self.dfs = []
for i in range(len(self.nodes)):
    self.dfs.append((set()))
# 找到所有 v，使得 x 支配 v 的前驱节点，x 不严格支配 v
for (u, v) in self.edges:
    x = u
    while x not in self.dominates[v]:
        self.dfs[x] = set.union(self.dfs[x], set({v}))
        x = self.idoms[x]
```

插入 phi 函数

根据 https://szp15.com/post/how-to-construct-ssa/ 中提供的算法，首先遍历每个块以及块中的指令，获取每个块的 worklist，然后对每个变量计算需要在那些块放置 phi 函数。在获取每个块的 worklist 的时候，考虑到每个变量的地址是唯一的，所以使用地址和名称的 tuple 作为索引。

```
def insert_phi(self):
    """
    插入 phi 函数
    https://szp15.com/post/how-to-construct-ssa
    """
    temp_id = self.new_temp_id
    var_dict = {}
    var_address = []
    # 首先计算每个变量的 worklist（在哪一个块中被复制），
    # 由于局部变量的地址是唯一的，所以选用 tuple(address, ident.value) 的形式作为 key 保存
    for idx_block, node in enumerate(self.nodes):
        for l in node.locs:
            if isinstance(l.instr, Alloc) or isinstance(l.instr, SaveWord):
                if l.instr.ident != None:
                    if l.instr.ident.getattr("address") not in var_address:
                        var_address.append(l.instr.ident.getattr("address"))
                        var_dict[(l.instr.ident.getattr("address"),
l.instr.ident.value)] = [idx_block]
                    else:
                        var_dict[(l.instr.ident.getattr("address"),
l.instr.ident.value)].append(idx_block)
    # 对每一个变量
    for var in var_dict.keys():
        worklist = list(set(var_dict[var]))
        visited = [False for i in range(len(self.nodes))]
        placed = [False for i in range(len(self.nodes))]
        while len(worklist) != 0: # 当 worklist 不为空时循环
            x = worklist.pop(-1) # 弹出最后一个节点
            for y in self.dfs[x]:
                # 对所有 x 的支配边界的元素如果还没有 phi 指令就插入一条
                if placed[y] == False:
                    placed[y] = True
                    self.nodes[y].locs.insert(0, Loc(Phi(Temp(self.new_temp_id),
var)))
                    # print("label: ", self.nodes[y].label)
                    self.new_temp_id += 1
                    if visited[y] == False: # 如果还没有访问过这一块就将这一块添加进
worklist 中
                        visited[y] = True
                        worklist.append(y)
    self.var_dict = var_dict
    self.var_address = var_address
```

变量重命名

在变量重命名阶段首先为每一个变量都新设置一个 stack，用于在接下来的过程中获取当前活跃的定义。然后调用 search 函数。

search 函数仿照 https://szp15.com/post/how-to-construct-ssa/ 实现。首先对 Alloc 指令，记录在块中的位置（是第几条指令）以待之后删除。对 `LoadWord` 指令，对之后所有块中的指令依次遍历，若指令的 srcs 中出现该寄存器，则把该寄存器替换成 `stack[key][-1]` 即对应变量栈顶的元素。对 `SaveWord` 指令，把该指令的 `dsts[0]` 入栈。对 `Phi` 指令，先将寄存器入栈，再对之后所有块中的指令依次遍历，若指令的 srcs 中出现该寄存器，则把该寄存器替换成 `stack[key][-1]` 的元素。所有指令遍历完后将 `Alloc/LoadWord/SaveWord` 指令移除。

之后对每个变量遍历所有后继块，如果由 Phi 指令就加入当前块的 Label 和栈顶寄存器作为 Phi 的一个选择。

最后采用类似 dfs 的形式依次处理所有的孩子节点，处理完毕后将每个元素在此块中新定义的 Temp 退栈。

```
def rename(self):
        stack = {}
        counters = self.new_temp_id
        for key in self.var_dict.keys():
            stack[key] = []
        self.search(0, self.nodes[0], stack, counters)


        print("--------------------")
        for idx_block, node in enumerate(self.nodes):
            # print(node.kind)
            if node.label is not None:
                if idx_block == 0:
                    print(str(self.func_name)+":")
                print(str(node.label) + ":")
            for l in node.locs:
                print("    "  + str(l.instr))

        print("--------------------")

    def search(self, idx_block, node, stack, counters):
        remove_index = []
        stack_push = {}
        for key in self.var_dict.keys():
            stack_push[key] = 0
        for idx, l in enumerate(node.locs):
            if isinstance(l.instr, Alloc) and l.instr.ident is not None:
                remove_index.append(l)
            elif isinstance(l.instr, LoadWord) and l.instr.ident is not None:
                dst = l.instr.dsts[0]
                key = (l.instr.ident.getattr("address"), l.instr.ident.value)
                for i in range(idx + 1, len(node.locs)):
                    for src_idx, src in enumerate(node.locs[i].instr.srcs):
                        if src == dst:
```

```
                            node.locs[i].instr.srcs[src_idx].index = stack[key]
[-1].index
                remove_index.append(l)
            elif isinstance(l.instr, SaveWord) and l.instr.ident is not None:
                key = (l.instr.ident.getattr("address"), l.instr.ident.value)
                counters += 1
                stack[key].append(l.instr.dsts[0])
                stack_push[key] += 1
                remove_index.append(l)
            elif isinstance(l.instr, Phi):
                key = l.instr.ident
                stack[key].append(l.instr.dsts[0])
                stack_push[key] += 1
                dst = l.instr.dsts[0]
                for i in range(idx + 1, len(node.locs)):
                    for src_idx, src in enumerate(node.locs[i].instr.srcs):
                        if src == dst:
                            node.locs[i].instr.srcs[src_idx].index = stack[key]
[-1].index
        for remove_idx in remove_index:
            node.locs.remove(remove_idx)
        for var in self.var_dict.keys():
            for succ in self.links[idx_block][1]:
                for idx_l, l in enumerate(self.nodes[succ].locs):
                    if isinstance(l.instr, Phi) and l.instr.ident == var:
                        label = node.label
                        l.instr.add_label(label)
                        l.instr.add_src(stack[var][-1], label)
        for child in self.tree_links[idx_block][1]:
            self.search(child, self.nodes[child], stack, counters)
        for key in self.var_dict.keys():
            if stack_push[key] != 0:
                stack[key] = stack[key][:-stack_push[key]]
```

二、运行结果

本阶段实验在 `Dominate` 类中的构造函数插入 Phi 指令之前打印原始 TAC，在 `rename` 的最后打印出经过处理后满足 SSA 的代码，具体输出格式如下：

```
--------------------
原始 TAC 码
--------------------
--------------------
满足 SSA 的 TAC 码
--------------------
```

1. 测试用例

本阶段选用了两个测试用例，分别为分支结构（实验指导使用的例子）和循环结构（实验要求中提到的阶乘的例子）。

```
// ssa_test.c
int main() {
  int x = 1;
  int cond = 1;
  if (cond > 0) {
    x = 1;
  } else {
    x = -1;
  }
  return x;
}
```

```
// ssa_test2.c
int main(){
    int temp = 1;
    int val = 5;
    for (int i = 2; i <= val; i = i + 1)
        temp = temp * i;
    return temp;
}
```

2. 运行结果

对于第一个测例，运行 `python main.py --input ssa_test.c  --tac > ssa_test.txt` 得到的结果为：

```
--------------------
FUNCTION<main>:
_L3:
    _T0 = ALLOC 4 # for x
    _T1 = 1
    SAVE _T1, 0(_T0) # for x
    _T2 = ALLOC 4 # for cond
    _T3 = 1
    SAVE _T3, 0(_T2) # for cond
    _T4 = 0
    _T5 =  LOAD 0(_T2) # for cond
    _T6 = (_T5 > _T4)
    if (_T6 == 0) branch _L1
_L4:
    _T7 = 1
    SAVE _T7, 0(_T0) # for x
    branch _L2
_L1:
    _T8 = 1
    _T9 = - _T8
```

```
    SAVE _T9, 0(_T0) # for x
_L2:
    _T10 =  LOAD 0(_T0) # for x
    return _T10
-------------------
-------------------
FUNCTION<main>:
_L3:
    _T1 = 1
    _T3 = 1
    _T4 = 0
    _T6 = (_T3 > _T4)
    if (_T6 == 0) branch _L1
_L4:
    _T7 = 1
    branch _L2
_L1:
    _T8 = 1
    _T9 = - _T8
_L2:
    _T11 = Phi [(_T7, _L4), (_T9, _L1)]  # for x
    return _T11
-------------------
```

可以看到在原始的 TAC 代码中，局部的 int 变量 x 和 cond 都使用 `ALLOC 4` 为其在栈上分配了空间，随后分别使用 `SaveWord` 指令将初始值存入对应地址中，在比较语句中需要调用 `cond` ，正确使用 `LoadWord` 指令将 `cond` 载入虚拟寄存器中，在之后对 x 的重新赋值中，也都将计算结果存入了 x 对应的地址。在最后的 return 语句中也将 x 用 Load 指令载入虚拟寄存器中作为返回值。

在运行 SSA 构造算法后，可以看到先前的 `Alloc/SaveWord/LoadWord` 都被删掉了，在 _L2 块中的 _T11 寄存器使用 phi 指令正确合并了来自 _L1 和 _L4 的结果，满足 SSA 要求。

对于第二个测例，运行 `python main.py --input ssa_test2.c  --tac > ssa_test2.txt` 得到的结果为：

```
-------------------
FUNCTION<main>:
_L4:
    _T0 = ALLOC 4 # for temp
    _T1 = 1
    SAVE _T1, 0(_T0) # for temp
    _T2 = ALLOC 4 # for val
    _T3 = 5
    SAVE _T3, 0(_T2) # for val
    _T4 = ALLOC 4 # for i
    _T5 = 2
    SAVE _T5, 0(_T4) # for i
_L1:
    _T6 =  LOAD 0(_T4) # for i
    _T7 =  LOAD 0(_T2) # for val
    _T8 = (_T6 <= _T7)
```

```
    if (_T8 == 0) branch _L3
_L5:
    _T9 =  LOAD 0(_T0) # for temp
    _T10 =  LOAD 0(_T4) # for i
    _T11 = (_T9 * _T10)
    SAVE _T11, 0(_T0) # for temp
_L2:
    _T12 = 1
    _T13 =  LOAD 0(_T4) # for i
    _T14 = (_T13 + _T12)
    SAVE _T14, 0(_T4) # for i
    branch _L1
_L3:
    _T15 =  LOAD 0(_T0) # for temp
    return _T15
--------------------
--------------------
FUNCTION<main>:
_L4:
    _T1 = 1
    _T3 = 5
    _T5 = 2
_L1:
    _T17 = Phi [(_T5, _L4), (_T14, _L2)]  # for i
    _T16 = Phi [(_T1, _L4), (_T11, _L2)]  # for temp
    _T8 = (_T17 <= _T3)
    if (_T8 == 0) branch _L3
_L5:
    _T11 = (_T16 * _T17)
_L2:
    _T12 = 1
    _T14 = (_T17 + _T12)
    branch _L1
_L3:
    return _T16
--------------------
```

在原始的 TAC 码中，使用 `Alloc/SaveWord/LoadWord` 指令使得不存在对一个寄存器多次赋值。

在运行 SSA 构造算法后，可以看到先前的 `Alloc/SaveWord/LoadWord` 都被删掉了，在 _L1 块中正确使用 Phi 指令对 i 和 temp 进行更新，由于循环结构， _L1 中 i 的取值有可能是来自于 _L4 也有可能来自于 _L2，temp 同理，此处正确处理了分别来自这两个块的可能的取值。