Stage 3 报告

程思翔 2021010761

1 实验内容

1.1 step 6

语义分析: 实现了 `ScopeStack` 数据结构维护层次嵌套的作用域.

```python
# frontend/scope/scopestack.py
class ScopeStack:
    def __init__(self, globalScope: Scope) -> None:
        self.globalScope = globalScope
        self.scopeStack = [globalScope]
        self.scopeDepth = 512

    def open(self, scope: Scope) -> None:
        if len(self.scopeStack) < self.scopeDepth:
            self.scopeStack.append(scope)
        else:
            raise ScopeOverflowError

    def close(self) -> None:
        self.scopeStack.pop()

    def top(self) -> Scope:
        if self.scopeStack:
            return self.scopeStack[len(self.scopeStack) - 1]
        return self.globalScope

    # 复用 Scope 类的成员函数
    def isGlobalScope(self) -> bool:
        return self.top().isGlobalScope()

```

```
    def declare(self, symbol: Symbol) -> None:
        self.top().declare(symbol)

    def lookup(self, name: str) -> Optional[Symbol]:
        return self.top().lookup(name)

    def lookupOverStack(self, name: str) -> Optional[Symbol]:
        for scope in reversed(self.scopeStack):
            if scope.containsKey(name):
                return scope.get(name)
        return None
```

将 `frontend/typecheck/namer.py`, `frontend/typecheck/typer.py` 的上下文信息修改为 “作用域栈” 后, 修改符号表建立过程的 `visitBlock` 函数, 开启一个代码块时, 新建作用域并压栈; 退出代码块时, 弹栈关闭作用域. `ScopeStack` 中为定义变量复用了 `Scope.lookup` 函数, 逐层查找变量实现 `ScopeStack.lookupOverStack` 函数.

```
# frontend/typecheck/namer.py
def visitBlock(self, block: Block, ctx: ScopeStack) -> None:
    ctx.open(Scope(ScopeKind.LOCAL))
    for child in block:
        child.accept(self, ctx)
    ctx.close()

def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -
> None:
    if ctx.lookup(decl.ident.value):
        ...

def visitIdentifier(self, ident: Identifier, ctx: ScopeStack) ->
None:
    symbol = ctx.lookupOverStack(ident.value)
    ...
```

后端: 增加了 `reachable` 函数, 使用 BFS 算法判断某个基本块是否可达.

```
# backend/dataflow/cfg.py
def __init__(self, nodes: list[BasicBlock], edges: list[(int,
int)]) -> None:
    ...
    self.reachability = []
    reachable = [0]
    for i in range(len(nodes)):
        self.reachability.append(False)

    while True:
        if not reachable:
            break
        cur = reachable.pop()
        self.reachability[cur] = True
        for succ in self.getSucc(cur):
            if not self.reachability[succ]:
                self.reachability[succ] = True
                reachable.append(succ)

def reachable(self, id):
    return self.reachability[id]
```

如果一个基本块不可达, 那么无须为它分配寄存器.

```
# backend/reg/bruteregalloc.py
def accept(self, graph: CFG, info: SubroutineInfo) -> None:
    ...
    for (index, bb) in enumerate(graph.iterator()):
        ...
        if graph.reachable(index):
            self.localAlloc(bb, subEmitter)
```

2 思考题

2.1 step 6

1. 请画出下面 MiniDecaf 代码的控制流图.

```
int main(){
    int a = 2;
    if (a < 3) {
        {
            int a = 3;
            return a;
        }
        return a;
    }
}
```

答: 这段代码的可能 TAC 码以及控制流图如下.

```
FUNCTION<main>:
    _T1 = 2
    _T0 = _T1
    _T2 = 3
    _T3 = (_T0 < _T2)
    if (_T3 == 0) branch _L1
    _T5 = 3
    _T4 = _T5
    return _T4
    return _T0
_L1:
    return
```

基本块 0

基本块 1

基本块 2

基本块 3