MiniDecaf-Stage3实验报告

计06 赵翊哲 2020010998

一、实验目的

在Stage2的基础上引入作用域和循环，丰富了编译器的功能。

语句块的部分主要包括对作用域的支持以及修改了部分关于寄存器分配的功能，可以为可达的基本块变量分配寄存器，而不需要额外处理不可达的基本块的变量。

循环的部分主要包括if语句和条件表达式

二、实验内容

2.1 Step 7

- 修改内容：增加了对块语句和作用域的支持
  - 提供对作用域的支持
  - 为可达基本块的变量分配寄存器
- 实现：
  - 前端：修改内容主要在 `namer.py` ：
    - 主要需要在进入一个语句块的时候新开一个局部作用域，压入栈中，并且退出语句块的时候将该作用域从栈中弹出。
    - 核心代码：

```
def visitBlock(self, block: Block, ctx: ScopeStack) -> None:
  # Step7-1 设置局部作用域
  localScope = Scope(ScopeKind.LOCAL)
  # 压栈
  ctx.open(localScope)
  for child in block:
    child.accept(self, ctx)
  # 出栈
  ctx.close()
```

  - 后端：修改部分为变量分配寄存器相关的代码
    - 主要是为了判断不同的基本块是否可达，若不可达则不分配寄存器
    - 可达的判断主要依赖于 `cfg.py` 中存储的图信息，如果一个基本块具备前驱，则意味着图中包括一条可达边。（参考了QA文档的思路，否则应使用搜索算法）
    - 核心代码

      `cfg.py`

```
def unreachable(self, id) -> Boolean:
  # Step7-2 判断某个基本块是否可达
  # 已经获得了完整的图，只需要判断有无前驱
  if id == 0:
    # 第一块基本块没有前驱，但是一定可达
    return False
  if self.getPrev(id) == set():
    # 没有前驱
    return True
  return False
```

`bruteregalloc.py`

```
def accept(self, graph: CFG, info: SubroutineInfo) -> None:
  subEmitter = self.emitter.emitSubroutine(info)
  for bb in graph.iterator():
    # Step7-3 跳过不可达基本块
    # you need to think more here
    # maybe we don't need to alloc regs for all the basic blocks
    if graph.unreachable(bb.id):
      continue
    if bb.label is not None:
      subEmitter.emitLabel(bb.label)
      self.localAlloc(bb, subEmitter)
  subEmitter.emitEnd()
```

2.2 Step 8 循环语句

- 修改内容：
  - 增加了循环语句，包括 `for` 、 `do-while`
  - 增加了对 `break` 和 `continue` 的支持
- 实现：
  - 本次修改的部分较多，因此只展示较为重要的核心代码，其他部分仅描述修改内容
  - 前端：
    - AST：增加对循环关键字的节点类定义，增加 `continue` 类的节点定义，值得关注的是在 `For` 类定义中，因为循环变量初始化、循环条件、循环变量更新都不是必须的，因此要给出缺省值。
    - 词法：添加相应保留字即可。
    - 语法：主要是针对循环语句给出相关文法，因为 `for` 循环中可以省略部分语句，因此要将全部情况包含进来。
  - 中端：
    - 语义分析： `namer.py` 中实现相关函数的构造，重要的是 `for` 循环中的循环变量应当自己属于一个作用域，因此在 `visitFor` 函数中，需要再循环体之前将局部作用域压栈。
    - 中间代码生成：仿照 `while` 的处理方式，通过 `looplabel` 、 `beginlabel` 、 `exitlabel` 等入口标签完成循环过程中的条件判断、循环跳转、变量更新、退出循环等诸多操作。
  - 后端：没有加入指令，因此没有添加新的代码。

- 核心代码：

```
# ply_parser.py
def p_for(p):
  """
  statement_matched : For LParen expression Semi expression Semi expression
RParen statement_matched
        | For LParen declaration Semi expression Semi expression RParen
statement_matched
  statement_unmatched : For LParen expression Semi expression Semi expression
RParen statement_unmatched
        | For LParen declaration Semi expression Semi expression RParen
statement_unmatched
    """
  p[0] = For(p[9], p[3], p[5], p[7])
# namer.py
def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
  # for 循环需要自带一个作用域
  # 要模仿 visitBlock 函数，打开/关闭对应的局部作用域 "Scope(ScopeKind.LOCAL)"。
  localScope = Scope(ScopeKind.LOCAL)
  ctx.open(localScope)
  if not stmt.init is NULL:
    stmt.init.accept(self, ctx)
  if not stmt.cond is NULL:
    stmt.cond.accept(self, ctx)
  ctx.openLoop()
  stmt.body.accept(self, ctx)
  if not stmt.update is NULL:
    stmt.update.accept(self, ctx)
  ctx.closeLoop()
  ctx.close()
# tacgen.py
def visitFor(self, stmt: For, mv: FuncVisitor) -> None:
  # print(stmt)
  beginLabel = mv.freshLabel()
  loopLabel = mv.freshLabel()
  breakLabel = mv.freshLabel()
  mv.openLoop(breakLabel, loopLabel)
  # init
  if stmt.init is not NULL:
    stmt.init.accept(self, mv)
  # begin label
  mv.visitLabel(beginLabel)
  # condition
  if stmt.cond is not NULL:
    stmt.cond.accept(self, mv)
    mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"),
breakLabel)
  # body
```

```
    stmt.body.accept(self, mv)
  # loop label
    mv.visitLabel(loopLabel)
  # update
    if not stmt.update is NULL:
      stmt.update.accept(self, mv)
  # j begin label
  mv.visitBranch(beginLabel)
  # break label
  mv.visitLabel(breakLabel)
  mv.closeLoop()
```

三、思考题

3.1 Step7 思考题

Q1：请画出下面 MiniDecaf 代码的控制流图。

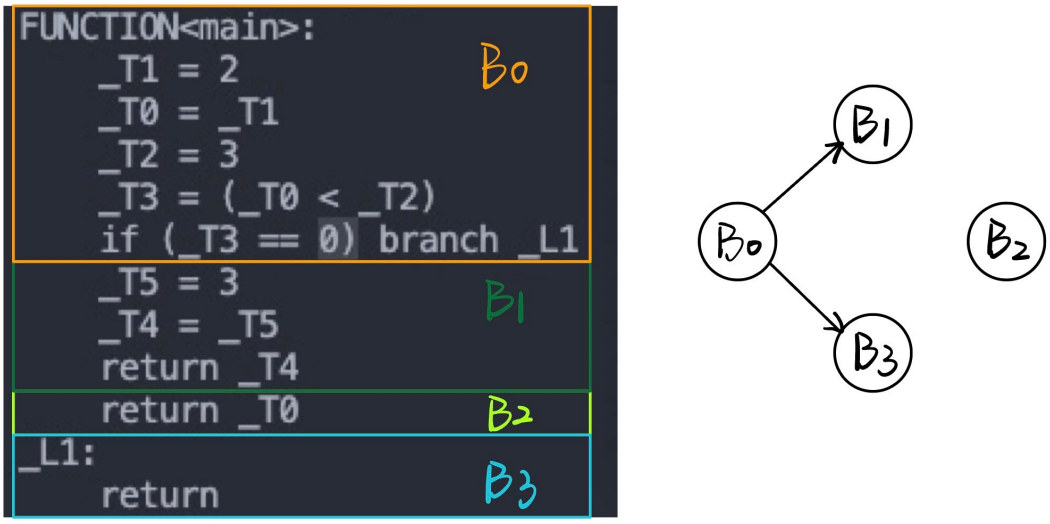
```
int main(){
 int a = 2;
 if (a < 3) {
     {
         int a = 3;
         return a;
     }
     return a;
 }
}
```

> 先划分基本块，再根据基本块的出口跳转语句判断有向边关系即可

A：程序生成的三地址码如下

```
● (base) zhaoyizhe@bogon minidecaf-2020010998 % python3 main.py --input minidecaf-tests/testcases/step7/multi_nesting.c --tac
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
```

对应的基本块划分和控制流图如下：



3.2 Step8 思考题

Q1：将循环语句翻译成 IR 有许多可行的翻译方法，例如 while 循环可以有以下两种翻译方式：

第一种（即实验指导中的翻译方式）：

1. `label BEGINLOOP_LABEL`：开始下一轮迭代
2. `cond 的 IR`
3. `beqz BREAK_LABEL`：条件不满足就终止循环
4. `body 的 IR`
5. `label CONTINUE_LABEL`：continue 跳到这
6. `br BEGINLOOP_LABEL`：本轮迭代完成
7. `label BREAK_LABEL`：条件不满足，或者 break 语句都会跳到这儿

第二种：

1. `cond 的 IR`
2. `beqz BREAK_LABEL`：条件不满足就终止循环
3. `label BEGINLOOP_LABEL`：开始下一轮迭代
4. `body 的 IR`
5. `label CONTINUE_LABEL`：continue 跳到这
6. `cond 的 IR`
7. `bnez BEGINLOOP_LABEL`：本轮迭代完成，条件满足时进行下一次迭代
8. `label BREAK_LABEL`：条件不满足，或者 break 语句都会跳到这儿

从执行的指令的条数这个角度（`label` 指令不计算在内，假设循环体至少执行了一次），请评价这两种翻译方式哪一种更好?

> 感性理解可以发现在第二种执行过程中将“条件满足”和“跳到循环体”两个跳转结合到一句 `bnez BEGINLOOP_LABEL` 中了，指令使用的更少；定量分析可以比较 n 次循环中两种翻译方式的指令执行次数。

A：第二种方式更好。n次循环中，两种方式都需要执行 (n + 1) 次条件语句，n 次循环体。但是第一种翻译方式需要执行 (n + 1) 次判断（`beq`） 和 n 次迭代完成之后的跳转（`br`）；而第二种翻译方式只需要进行 （n + 1） 次判断（1次 `beqz` 和 n 次 `bnez` ）即可，所以第二种方式执行的指令条数更少，是更好的翻译方式。

四、参考资料

实现过程中主要参考了实验文档和实验思路QA墙