

Stage-3 实验报告

沙之洲 2020012408

Step 7

实验内容

在这一阶段，我们需要支持局部作用域。具体来说，要在 visitBlock 的时候打开一个新的作用域，namer.py 的代码如下

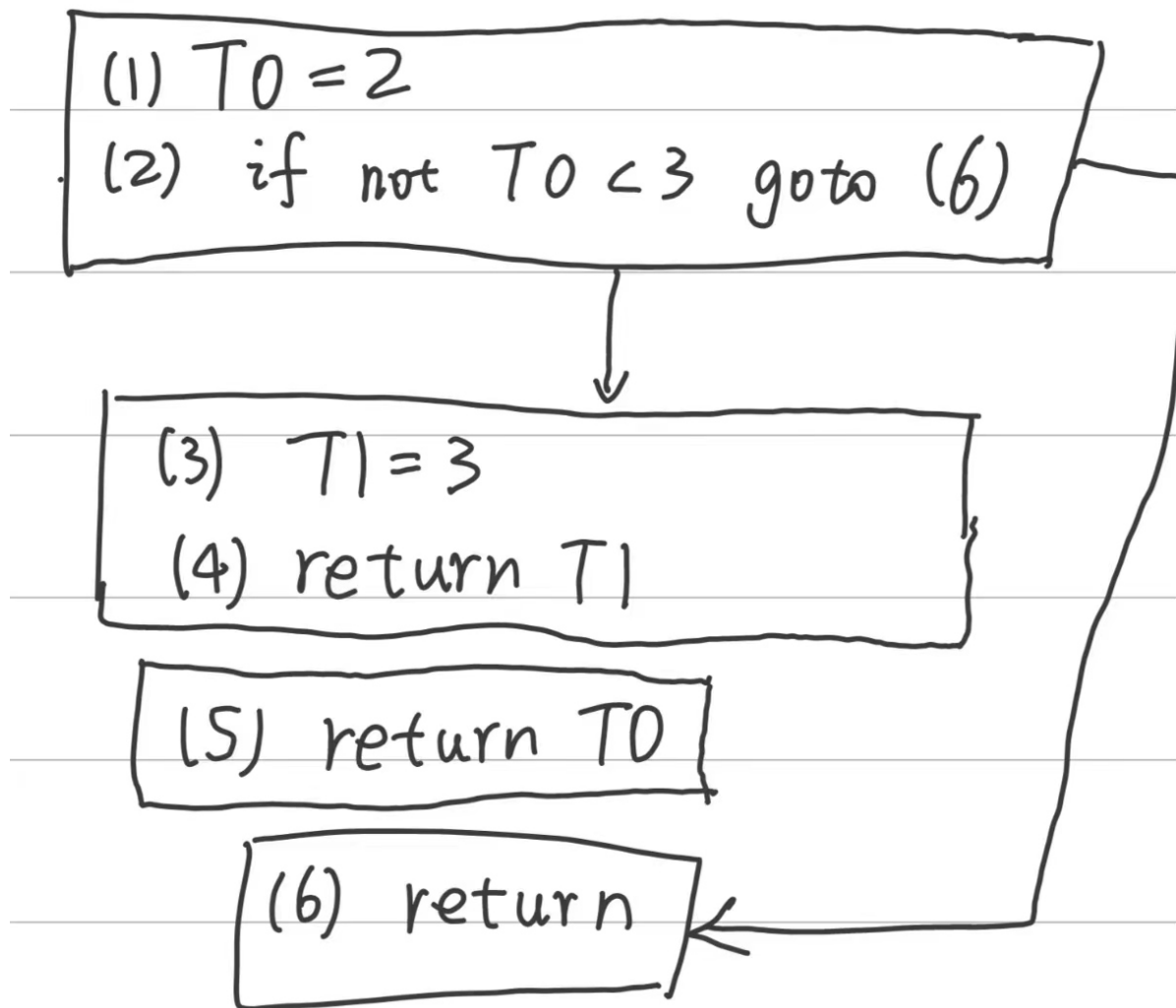
```
50     ....func.body.accept(self, ctx) # then visit the block of the function, recursively
51
52     ....def visitBlock(self, block: Block, ctx: ScopeStack) -> None:
53     ....    ....# a function block is constructed by several sentence, we should visit all of them
54     ....    ....
55     ....    ....# we have to define a scope for each block
56     ....    ....# open a scope here
57     ....    ....ctx.open(Scope(ScopeKind.LOCAL))
58
59     ....    ....for child in block:
60     ....    ....    ....child.accept(self, ctx)
61
62     ....    ....# close the scope
63     ....    ....ctx.close()
64
```

对于后端而言，这一阶段我们要考虑寄存器的分配。我们知道中间代码 tac 可以根据一些规则分为若干个 basic block，这里 bruteallocate 实现的策略是对于每一个 bb 重新分配寄存器，因此我们要做的是只给那些程序可能到达的 bb 分配寄存器，也就是那些可达的 bb，这里我才用的是 dfs 判断是否可达，具体代码如下

```
44
45     ....# my codes are below
46     ....def unreachable(self, node) -> bool:
47     ....    ....# true for unreachable, false for reachable
48
49     ....    ....# search for index
50     ....    ....id = self.nodes.index(node)
51     ....    ....
52     ....    ....# Implement dfs here
53     ....    ....stack = []
54     ....    ....stack.append(0)
55     ....    ....have_searched_list = []
56     ....    ....have_searched_list.append(0)
57
58     ....    ....while stack:
59     ....    ....    ....current = stack.pop()
60     ....    ....    ....nodes = list(self.getSucc(current))
61     ....    ....    ....for i in nodes[::-1]: # since the last in stack, will be popped first, therefore need reverse here
62     ....    ....    ....    ....if i not in have_searched_list:
63     ....    ....    ....        ....stack.append(i)
64     ....    ....    ....        ....have_searched_list.append(i)
65
66     ....    ....    ....if current == id:
67     ....    ....    ....        ....return False
68
69     ....    ....    ....# unreachable! sad
70     ....    ....    ....return True
71
```

思考题

控制流图如下



Step 8

实验内容

首先我们需要在解析 ast 树的时候增加 for 和 do while 的解析，代码如下

```
115 |... p[0] = While(p[3], p[5])
116 |
117 |def p_for(p):
118 |    """
119 |    ... statement_matched : For LParen opt_expression Semi opt_expression Semi opt_expression RParen statement_matched
120 |    ... | For LParen declaration Semi opt_expression Semi opt_expression RParen statement_matched
121 |    ... statement_unmatched : For LParen opt_expression Semi opt_expression Semi opt_expression RParen statement_unmatched
122 |    ... | For LParen declaration Semi opt_expression Semi opt_expression RParen statement_unmatched
123 |    ... | For LParen opt_expression Semi opt_expression Semi opt_expression RParen statement_matched
124 |    ... | For LParen declaration Semi opt_expression Semi opt_expression RParen statement_matched
125 |    """
126 |    p[0] = For(p[3], p[5], p[7], p[9])
127 |
128 |def p_dowhile(p):
129 |    """
130 |    ... statement_matched : Do statement_matched While LParen expression RParen Semi
131 |    ... statement_unmatched : Do statement_unmatched While LParen expression RParen Semi
132 |    ... | Do statement_matched While LParen expression RParen Semi
133 |    """
134 |    p[0] = DoWhile(p[5], p[2])
135 |
136 |def p_continue(p):
```

接下来需要在前端的 namer 和 tacgen 的对应部分增加对应的 visit 函数，这里以 tacgen 中的 visitFor 为例

```

169 .....mv.closeLoop()
170
171 ✓ .....def visitFor(self, stmt: For, mv: FuncVisitor) -> None:
172 .....beginLabel = mv.freshLabel()
173 .....loopLabel = mv.freshLabel()
174 .....breakLabel = mv.freshLabel()
175
176 ✓ .....if not isinstance(stmt.init, node.NullType):
177 .....stmt.init.accept(self, mv)
178
179 .....mv.openLoop(breakLabel, loopLabel)
180
181 .....mv.visitLabel(beginLabel)
182 .....
183 ✓ .....if not isinstance(stmt.cond, node.NullType):
184 .....stmt.cond.accept(self, mv)
185 .....mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"), breakLabel)
186 .....# stmt.cond.accept(self, mv)
187 .....# mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"), breakLabel)
188 .....
189 .....stmt.body.accept(self, mv)
190 .....
191 .....mv.visitLabel(loopLabel)
192 .....
193 ✓ .....if not isinstance(stmt.update, node.NullType):
194 .....stmt.update.accept(self, mv)
195 .....mv.visitBranch(beginLabel)
196 .....
197 .....mv.visitLabel(breakLabel)
198 .....mv.closeLoop()
199
200

```

在 visit 函数中需要做的操作包括，打开局部作用域，指定标签等。

思考题

1. 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 : 开始新一轮迭代
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 : 开始新一轮迭代
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？

首先给出结论：第二种翻译方式使用的指令条数更少，因此第二种翻译方式更好。

原因如下：

可以看到第一种和第二种翻译方式的区别主要在于，第一种翻译方式是在进入 body 之前进行 condition 的判断，也就是说，每次 body 结束之后，都需要一个无条件的跳转，才能开始下一轮的 condition 的判断和迭代。但是对于第二种翻译方式，将跳转和 condition 判断这两步操作放到同一条指令中进行，也就是第 7 条的 bnez BEGIN_LABEL 指令。

不难看出，对于每一轮迭代，第一种翻译方式都会比第二种翻译方式多执行一条指令。因此，第二种翻译方式更好。