Stage-5 实验报告

沙之洲 2020012408

Step 11

实验内容

首先对于 parse 阶段，我们需要处理的是数组的定义和引用。数组的定义可以合并到 Declaration 当中，只需要为 Declaration 加上一个 size list 属性即可。对于引用阶段，我也是将变量引用和数组引用进行了合并，成为 Refer。

```
361     unary(p)
362
363 def p_refer_array(p):
364     """
365     refer : Identifier refer_arguments
366     """
367     p[0] = Refer(p[1], p[2])
368
369 def p_refer_arguments(p):
370     """
371     refer_arguments : LBracket expression RBracket refer_arguments
372     """
373     p[0] = [p[2]] + p[4]
374
375 def p_refer_empty(p):
376     """
377     refer_arguments : empty
378     """
379     p[0] = []
380
```

通过这种方式，在 namer 和 typer 的阶段，我们可以通过 Refer 中的 argument list 的长度是否为 0 来判断是数组引用还是变量引用。

在中间代码生成阶段，也即 tac 阶段。我们需要根据引用是否是数组进行不同的处理。

```python
    def visitRefer(self, refer: Refer, mv: FuncVisitor) -> None:
        # accept ident first
        refer.ident.accept(self, mv)
        refer.setattr("val", refer.ident.getattr("val"))
        if len(refer.argument_list):
            # array
            # prepare the size list
            array_symbol = refer.getattr("symbol")
            size_list = array_symbol.size_list
            # array call
            argument_temp_list = []
            for argument in refer.argument_list:
                argument.accept(self, mv)
                # use param to declare arguments
                argument_temp_list.append(argument.getattr("val"))
            # argument_temp_list.reverse()
            offset = mv.visitLoad(0)
            for i in range(len(argument_temp_list)):
                argument_temp = argument_temp_list[i]
                # calculate the size
                size = 1
                for j in range(i+1, len(size_list)):
                    size *= size_list[j]
                arg_mul_temp = mv.visitBinary(tacop.BinaryOp.MUL, argument_temp, mv.visitLoad(size))
                mv.visitBinarySelf(tacop.BinaryOp.ADD, offset, arg_mul_temp)
            mv.visitBinarySelf(tacop.BinaryOp.MUL, offset, mv.visitLoad(4))
            base_temp = refer.ident.getattr("val")
            mv.visitBinarySelf(tacop.BinaryOp.ADD, offset, base_temp)
            # record the position of the array item
            refer.setattr("addr", offset)
            value_temp = mv.visitLoadFromMem(offset, 0)
            refer.setattr("val", value_temp)

```

对于变量引用，这里无需特殊的处理。但是对于数组引用，我们需要计算偏移量。这里由于数组的各个维度都能在编译阶段确定。所以我们这里先用 python 计算每一个维度需要乘以的数值，再和实际的 虚拟寄存器 相乘，得到最终的偏移量。

对于 step 11 而言，后端似乎没有特别多的工作量。借助之前的工作将 tac 增加的语句进行翻译即可。

Remark：我在实现 step 11 的时候，一个 step 9 的 bug 困扰了我很久，最终花了很长时间才找出来。

```python
        # if call, calculate all space need
        elif isinstance(instr, Riscv.Call):
            # store calller save
            # here i sill check if it is used
            used_caller_saved = []

            for reg in self.emitter.callerSaveRegs:
                if reg.occupied:
                    used_caller_saved.append(reg)

            # caller save reg name, correspond to its offset accrod to SP
            caller_save_dict = {}

```

在 bruteregalloc 阶段，在处理 Call 的时候，之前的我错误理解了 reg.occupied 和 reg.isUsed 含义，之前这里是将所有 isUsed 的 reg 全部存储到栈上。但是这样做会在 quicksort 的测例中的递归时候会产生一些莫名其妙的覆盖，导致程序错误执行。最终经过我苦苦思索，找到了这个问题。可以说我的 step 11 的很大一部分工作量都是在这里。

思考题

对于这种可变长度的数组，我们并不能在编译期间就知道其长度。因此区别于在进入函数的时候就给局部数组分配指针。我们采用遇到局部数组声明的时候，再为其分配栈空间。

具体来说，当我们遇到 int a[n] 的时候，把 sp 下拉 4 * n 的大小，并记录这个大小，在函数结束的时候整体恢复 sp。

同时，为了使得寄存器的存储和 callee save 寄存器访存行为依旧正常。我们需要维护一个寄存器，来记录函数刚刚结束 prologue 时候的 sp 值，我们这里称这个值为静态 sp。这样之后，我们在函数体里边做 NativeStore 和 NativeLoad 的时候，基于静态 sp 进行偏移即可。这样可以保证寄存器在栈上的访存不受到为局部数组分配栈空间的影响。

Step 12

实验内容

这一阶段需要我们支持数组的初始化，和参数传递。

对于前端而言，我们需要在 parse 阶段增加对于数组初始化的匹配。

```
250
251 def p_declaration_array(p):
252     """
253     declaration : type Identifier array
254     """
255     p[0] = Declaration(p[1], p[2], p[3], None)
256
257 def p_declaration_array_assign(p):
258     """
259     declaration : type Identifier array Assign array_expression
260     """
261     p[0] = Declaration(p[1], p[2], p[3], p[5])
262
263
264 def p_array_expression(p):
265     """
266     array_expression : LBrace array_expression_sub RBrace
267     """
268     p[0] = p[2]
269
270 def p_array_expression_sub(p):
271     """
272     array_expression_sub : expression Comma array_expression_sub
273     """
274     p[0] = [p[1]] + p[3]
275
276 def p_array_expression_sub_single(p):
277     """
278     array_expression_sub : expression
279     """
280     p[0] = [p[1]]
281
```

这里我是将所有初始化参数作为数组传进 Declaration 节点的。

除此之外，我们还需要函数调用的时候，支持数组作为参数，parse 阶段的主要改动如下

```
109     p[0] = Parameter(p[1], p[2], False)
110
111 def p_array_parameter(p):
112     """
113     parameter : type Identifier LBracket maybe_int RBracket
114     """
115     p[0] = Parameter(p[1], p[2], True)
116
117 def p_maybe_int(p):
118     """
119     maybe_int : Integer
120     """
121     p[0] = []
122
123 def p_maybe_int_empty(p):
124     """
125     maybe_int : empty
126     """
127     p[0] = []
128
```

对于中间代码生成阶段，我们需要手动定义数组初始化的行为，这里对于要初始化的数组，我们首先调用 fill_n 函数将所有填充为 0， 接下来在按照初始化的值进行赋值。

```
222
223             # check init
224             if decl.init_expr is not NULL:
225                 # set all 0
226                 zero_temp = mv.visitLoad(0)
227                 size_temp = mv.visitLoad(size)
228                 mv.visitParameter(symbol.temp)
229                 mv.visitParameter(zero_temp)
230                 mv.visitParameter(size_temp)
231                 mv.visitCall("fill_n", [symbol.temp, zero_temp, size_temp])
232
233                 init_value_temp_list = []
234                 # init array
235                 init_expr = decl.init_expr
236                 for item in init_expr:
237                     item.accept(self, mv)
238                     init_value_temp_list.append(item.getattr("val"))
239
240                 # here we only deal with 1 dimension array
241                 # init_value_temp_list.reverse()
242
243                 for i in range(len(init_value_temp_list)):
244                     mv.visitStoreToMem(init_value_temp_list[i], i * 4, symbol.temp)
```

在 visitDeclaration 部分进行上述操作即可。

对于 step 12， 后端部分也没有额外的工作量，只需要将 tac 阶段的指令正确翻译即可。

思考题

因为在 C/C++ 中，数组作为函数参数传递的时候，实际上传递的是指向数组开始的指针。而被调用的子函数访问数组的时候，是基于传进来的指针进行偏移访问的。因此，对于被调用的子函数而言，它并不关心数组究竟有多大，它只关心改如何访问数组，也就是只关心通过参数传递进来的指向数组开始的指针。

因此，在函数参数中声明数组大小的行为，只是程序员之间的君子协议，在实际程序执行的过程中，并不会关心。