

STAGE-1：常量表达式

计04 何秉翔 2020010944

1. 实验内容

在 stage-1 中，主要完成的部分是 TAC 的生成以及 riscv 汇编的生成，未涉及到词法、语法的分析

1.1 step-2：一元运算符

首先在 tacgen.py 文件中的 visitUnary 函数中加上从 AST 节点到 TAC 运算符的映射，类似已经提供的取负运算的实现，我们给出另外两种运算符的映射。

```
1 op = {
2     node.UnaryOp.Neg: tacop.UnaryOp.NEG,
3     node.UnaryOp.BitNot: tacop.UnaryOp.NOT,
4     node.UnaryOp.LogicNot: tacop.UnaryOp.SEQZ,
5 }[expr.op]
```

此外，我们在 tacinstr.py 中修改 Unary 类中 __str__ 函数的实现，用类似 Binary 类的 __str__ 实现的方式，使得原来代码不仅作用于 -、~ 以及 !，还可以较容易地拓展：

```
1 def __str__(self) -> str:
2     opStr = {
3         UnaryOp.NEG: "-",
4         UnaryOp.NOT: "~",
5         UnaryOp.SEQZ: "!",
6     }[self.op]
7     return "%s = %s %s" % (
8         self.dst,
9         opStr,
10        self.operand,
11    )
```

1.2 step-3：加减乘除模

同理，我们在 tacgen.py 文件中的 visitBinary 函数中加上从 AST 节点到 TAC 运算符的映射，类似已经提供的加法运算的实现，我们给出另外四种运算符的映射。

```
1 op = {
2     node.BinaryOp.Add: tacop.BinaryOp.ADD,
3     node.BinaryOp.Sub: tacop.BinaryOp.SUB,
4     node.BinaryOp.Mul: tacop.BinaryOp.MUL,
5     node.BinaryOp.Div: tacop.BinaryOp.DIV,
6     node.BinaryOp.Mod: tacop.BinaryOp.REM,
7 }[expr.op]
```

1.3 step-4：比较和逻辑表达式

框架对于二元表达式的逻辑与和按位与似乎有些问题，于是在 tacop.py 中新增加逻辑与和逻辑或的枚举节点，并在 tacinstr.py 中的 Binary 的 __str__ 属性中新增对应的 TAC 运算符。

先类似前两个 step 对所有的比较和逻辑运算符进行 AST 节点到 TAC 运算符的映射如下：

```

1  op = {
2      node.BinaryOp.LogicAnd: tacop.BinaryOp.LOGICAND,
3      node.BinaryOp.BitAnd: tacop.BinaryOp.AND,
4      node.BinaryOp.BitOr: tacop.BinaryOp.OR,
5      node.BinaryOp.LogicOr: tacop.BinaryOp.LOGICOR,
6      node.BinaryOp.EQ: tacop.BinaryOp.EQU,
7      node.BinaryOp.NE: tacop.BinaryOp.NEQ,
8      node.BinaryOp.GE: tacop.BinaryOp.GEQ,
9      node.BinaryOp.LE: tacop.BinaryOp.LEQ,
10     node.BinaryOp.GT: tacop.BinaryOp.SGT,
11     node.BinaryOp.LT: tacop.BinaryOp.SLT,
12 }[expr.op]

```

但跑测试的时候发现，对于 `neq`、`equ`、`geq`、`leq`、`logicand`、`logicor` 这六个指令会出现错误

对于逻辑与和逻辑或，我们将两边的表达式通过 `SNEZ` 转化为布尔变量值，再通过按位与和按位或即可。

对于剩下四条指令，报的其中一个错误如下：

```

1  ERR testcases/step4/le_false.c
2  ==== Error information =====
3  testcases/step4/le_false.s: Assembler messages:
4  testcases/step4/le_false.s:12: Error: unrecognized opcode `leq t2,t0,t1'
5  =====

```

通过查找 `riscv` 指令集发现，没有上述那六条指令，于是自己写一个测试程序如下：

```

1  // foo.c
2  int foo(int x, int y) {
3      return x <= y;
4  }

```

通过命令 `riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 foo.c -S -O3 -o foo.s` 生成 `foo.s` 如下：

```

1  ...
2  foo:
3      sgt a0,a0,a1
4      xori    a0,a0,1
5      ret
6      .size   foo, .-foo
7      .ident  "GCC: (SiFive GCC 10.1.0-2020.08.2) 10.1.0"

```

发现实际上编译器是这么处理的：

- 要比较 \leq ，可以利用已有的指令 `sgt` 来判断是否 $>$
- 最后将结果取个逻辑反即可
- 我们不采用与 1 异或的方法，我们直接用已有的一元运算符 `SEQZ` 进行逻辑取反

综合以上步骤可以得知，再从 TAC 生成 `riscv` 时，若遇到 `tacop.BinaryOp.LEQ` 或者其他几条指令，需要生成两条汇编指令，因此我们定位到生成 `riscv` 汇编的地方，即 `riscvasmemitter.py` 文件，修改

`visitBinary` 函数的实现如下：

```

1  # for neq, equ, geq and leq
2  if instr.op == tacop.BinaryOp.NEQ:
3      self.seq.append(Riscv.Binary(tacop.BinaryOp.SUB, instr.dst, instr.lhs,
                                     instr.rhs))

```

```

4     self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.dst, instr.dst))
5     elif instr.op == tacop.BinaryOp.EQU:
6         self.seq.append(Riscv.Binary(tacop.BinaryOp.SUB, instr.dst, instr.lhs,
instr.rhs))
7         self.seq.append(Riscv.Unary(tacop.UnaryOp.SEQZ, instr.dst, instr.dst))
8     elif instr.op == tacop.BinaryOp.GEQ:
9         self.seq.append(Riscv.Binary(tacop.BinaryOp.SLT, instr.dst, instr.lhs,
instr.rhs))
10        self.seq.append(Riscv.Unary(tacop.UnaryOp.SEQZ, instr.dst, instr.dst))
11    elif instr.op == tacop.BinaryOp.LEQ:
12        self.seq.append(Riscv.Binary(tacop.BinaryOp.SGT, instr.dst, instr.lhs,
instr.rhs))
13        self.seq.append(Riscv.Unary(tacop.UnaryOp.SEQZ, instr.dst, instr.dst))
14    elif instr.op == tacop.BinaryOp.LOGICAND:
15        self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.lhs, instr.lhs))
16        self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.rhs, instr.rhs))
17        self.seq.append(Riscv.Binary(tacop.BinaryOp.AND, instr.dst, instr.lhs,
instr.rhs))
18    elif instr.op == tacop.BinaryOp.LOGICOR:
19        self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.lhs, instr.lhs))
20        self.seq.append(Riscv.Unary(tacop.UnaryOp.SNEZ, instr.rhs, instr.rhs))
21        self.seq.append(Riscv.Binary(tacop.BinaryOp.OR, instr.dst, instr.lhs,
instr.rhs))
22    else:
23        self.seq.append(Riscv.Binary(instr.op, instr.dst, instr.lhs, instr.rhs))

```

即针对上述六条指令做特殊处理，其他指令保持原样即可。

2. 思考题

2.1 step-2 思考题

问题：我们在语义规范中规定整数运算越界是未定义行为，运算越界可以简单理解成理论上的运算结果没有办法保存在32位整数的空间中，必须截断高于32位的内容。请设计一个 minidecaf 表达式，只使用 `~` `!` 这三个单目运算符和从 0 到 2147483647 范围内的非负整数，使得运算过程中发生越界。

解答：`~2147483647` 即可，2147483647 的十六进制表示是 `0x7FFFFFFF`，通过一步取反得到 `0x80000000`，即十进制意义下的 `-2147483648`，在经过一步取负运算得到 `2147483648` 即可超出 `int` 表示范围。

2.2 step-3 思考题

问题：我们知道“除数为零的除法是未定义行为”，但是即使除法的右操作数不是 0，仍然可能存在未定义行为。请问这时除法的左操作数和右操作数分别是什么？请将这时除法的左操作数和右操作数填入下面的代码中，分别在你的电脑（请标明你的电脑的架构，比如 x86-64 或 ARM）中和 RISC-V-32 的 qemu 模拟器中编译运行下面的代码，并给出运行结果。（编译时请不要开启任何编译优化）

解答：

```

1  #include <stdio.h>
2  int main() {
3      int a = -2147483648;
4      int b = -1;
5      printf("%d\n", a / b);
6      return 0;
7  }

```

此时，`a / b` 理论上得到 `2147483648` 的结果，也会发生越界。

- 在 x86-64 电脑上运行结果如下：

```
(MiniDecafEnv) minidecaf-2020010944 ➤ gcc foo.c git:stage-1*
(MiniDecafEnv) minidecaf-2020010944 ➤ ./a.out git:stage-1*
[1] 2110 floating point exception ./a.out
```

- 在 qemu 模拟器中运行结果如下：

```
(MiniDecafEnv) minidecaf-2020010944 ➤ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 foo.c git:stage-1*
(MiniDecafEnv) minidecaf-2020010944 ➤ qemu-riscv32 a.out git:stage-1*
-2147483648
```

2.3 step-4 思考题

问题：在 MiniDecaf 中，我们对于短路求值未做要求，但在包括 C 语言的大多数流行的语言中，短路求值都是被支持的。为何这一特性广受欢迎？你认为短路求值这一特性会给程序员带来怎样的好处？

解答：

- 可以提高程序的运行效率：有的时候条件判断时需要计算逻辑运算符的左右两个表达式，可能其中一个表达式的值计算时间较长，通过先计算另一个耗时较短的表达式，再结合逻辑运算符，可能可以直接判断整体的结果，而不必再运算另一个复杂的表达式，减少了程序的计算量。
- 可以防止出现一些边界情况使得程序崩溃，提高程序的鲁棒性：比如对于字符串中某一位的读写，或者数组中某一位的读写，需要首先判断是否为空串、是否为空或者是否越界等情况，此时表达式的执行就有先后顺序，如果前者失败了还去执行后者，可能会导致程序崩溃。