STAGE 1 REPORT

赵宇峰 致理-数理 1 2021013376

Step 1

1. 在我们的框架中，从 AST 向 TAC 的转换经了 `namer.transform`, `typer.transform` 两个步骤，如果没有这两个步骤，以下代码能正常编译吗，为什么？

```
int main(){
    return 10;
}
```

可以正常编译。原因：本代码未涉及全局变量 GlobalScope,且程序无语法错误，typecheck 未对程序做出改变。

2. 我们的框架现在对于 main 函数没有返回值的情况是在哪一步处理的？报的是什么错？

在语义分析步骤进行处理。对应以下内容：

```
def visitReturn(self, stmt: Return, ctx: Scope) -> None:
    stmt.expr.accept(self, ctx)
```

并未报错

3. 为什么框架定义了 `frontend/ast/tree.py:Unary`、`utils/tac/tacop.py:TacUnaryOp`、`utils/riscv.py:RvUnaryOp` 三种不同的一元运算符类型？

这是因为三个阶段需要使用不同的表示方式来处理一元运算符。
frontend/ast/tree.py:Unary：在抽象语法树（AST）的构建阶段使用的 Node 类型。
utils/tac/tacop.py:TacUnaryOp：用于生成三地址码表示的中间代码。
utils/riscv.py:RvUnaryOp：用于生成 RiscV 汇编代码。
三个阶段中对一元运算符的构建和使用都不同，因此需要不同的类型完成不同的功能。

Step2

模仿已经提供的 NEG 指令，增加以下内容：

//riscvasmemitter.py

```
    def visitUnary(self, instr: Unary) -> None:
        op = {
            TacUnaryOp.NEG: RvUnaryOp.NEG,
            # You can add unary operations here.
            TacUnaryOp.NOT: RvUnaryOp.NOT,
            TacUnaryOp.SEQZ: RvUnaryOp.SEQZ,

        }[instr.op]
        self.seq.append(Riscv.Unary(op, instr.dst, instr.operand))
```

//tacop.py

```
# Kinds of unary operations.
```

```
@unique
class TacUnaryOp(Enum):
    NEG = auto()
    NOT = auto()
      SEQZ = auto()

//riscv.py
@unique
class RvUnaryOp(Enum):
        NEG = auto()
        NOT = auto()
        SEQZ = auto()
```

Step2 思考题：语义规范中给出-(-2147483647-1) 是未定义行为。2147483647 是一个 32 位有符号整数的最大值：0111 1111 1111 1111 1111 1111 1111 1111。
如果我们对其进行取反操作 ~，结果是：
1000 0000 0000 0000 0000 0000 0000 0000
这个二进制数表示的是一个负数，它是 32 位有符号整数的最小值，即 -2147483648。
因此-~2147483647 越界。

STEP 3
模仿 ADD,依据 RISC-V 指令，增加以下内容：

```
//riscvasmemitter.py
      def visitBinary(self, instr: Binary) -> None:
          """
          For different tac operation, you should translate it to different RiscV code
          A tac operation may need more than one RiscV instruction
          """
          if instr.op == TacBinaryOp.LOR:
              self.seq.append(Riscv.Binary(RvBinaryOp.OR, instr.dst, instr.lhs, instr.rhs))
              self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
          else:
              op = {
                  TacBinaryOp.ADD: RvBinaryOp.ADD,
                  TacBinaryOp.SUB: RvBinaryOp.SUB,
                  TacBinaryOp.MUL: RvBinaryOp.MUL,
                  TacBinaryOp.DIV: RvBinaryOp.DIV,
                  TacBinaryOp.REM: RvBinaryOp.REM,
                  # You can add binary operations here.
              }[instr.op]
              self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```

```
//tacop.py
@unique
class TacBinaryOp(Enum):
    ADD = auto()
    SUB = auto()
    LOR = auto()
    MUL = auto()
    DIV = auto()
    REM = auto()

//riscv.py
@unique
class RvBinaryOp(Enum):
    ADD = auto()
    OR = auto()
    MUL = auto()
    DIV = auto()
    REM = auto()
    SUB = auto()
```

Step3 思考题:

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

结果 2147483648 溢出，所以会抛出异常。

自己的电脑（x86-64）:

Thread 1 received signal SIGFPE, Arithmetic exception.

0x000000000040156f in main ()

RISCV-32 qemu 仿真

```
2021013376@compiler-lab:~/minidecaf-2021013376$ cat bug.c
#include <stdio.h>

int main() {
  int a = -2147483648;
  int b = -1;
  printf("%d\n", a / b);
  return 0;
}2021013376@compiler-lab:~/minidecaf-2021013376$riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O3 -S bug.c
2021013376@compiler-lab:~/minidecaf-2021013376$ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 bug.s
2021013376@compiler-lab:~/minidecaf-2021013376$ file a.out
a.out: ELF 32-bit LSB executable, UCB RISC-V, soft-float ABI, version 1 (SYSV), statically linked, with debug_info, not stripped
2021013376@compiler-lab:~/minidecaf-2021013376$ qemu-riscv32 a.out
-2147483648
```

得到结果-2147483648

STEP 4

根据文档提示，逻辑运算符的 RISC-V 表示如下：

```
//riscvasmemitter.py
    def visitBinary(self, instr: Binary) -> None:
        """
        For different tac operation, you should translate it to different RiscV code
        A tac operation may need more than one RiscV instruction
        """
        if instr.op == TacBinaryOp.AND:
            self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.lhs))
            self.seq.append(Riscv.Binary(RvBinaryOp.SUB,instr.dst,Riscv.ZERO,instr.dst))
            self.seq.append(Riscv.Binary(RvBinaryOp.AND,instr.dst,instr.dst, instr.rhs))
            self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
            return
        if instr.op == TacBinaryOp.OR:
            self.seq.append(Riscv.Binary(RvBinaryOp.OR, instr.dst, instr.lhs, instr.rhs))
            self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
```

此外<= >= == !=都需要多步 RISC-V 指令。<=可以对>（SGT）取 Set Equal Zero 指令。其余类似。

```
        if instr.op == TacBinaryOp.LEQ:
            self.seq.append(Riscv.Binary(RvBinaryOp.SGT, instr.dst, instr.lhs, instr.rhs))
            self.seq.append(Riscv.Unary(RvUnaryOp.SEQZ, instr.dst, instr.dst))
            return
        if instr.op == TacBinaryOp.GEQ:
            self.seq.append(Riscv.Binary(RvBinaryOp.SLT, instr.dst, instr.lhs, instr.rhs))
            self.seq.append(Riscv.Unary(RvUnaryOp.SEQZ, instr.dst, instr.dst))
            return
        if instr.op == TacBinaryOp.EQU:
            self.seq.append(Riscv.Binary(RvBinaryOp.SUB, instr.dst, instr.lhs, instr.rhs))
            self.seq.append(Riscv.Unary(RvUnaryOp.SEQZ, instr.dst, instr.dst))
            return
        if instr.op == TacBinaryOp.NEQ:
            self.seq.append(Riscv.Binary(RvBinaryOp.SUB, instr.dst, instr.lhs, instr.rhs))
            self.seq.append(Riscv.Unary(RvUnaryOp.SNEZ, instr.dst, instr.dst))
            return
        else:
            op = {
                TacBinaryOp.ADD: RvBinaryOp.ADD,
                TacBinaryOp.SUB: RvBinaryOp.SUB,
                TacBinaryOp.MUL: RvBinaryOp.MUL,
                TacBinaryOp.DIV: RvBinaryOp.DIV,
```

```
            TacBinaryOp.REM: RvBinaryOp.REM,
            TacBinaryOp.AND: RvBinaryOp.AND,
            TacBinaryOp.SLT: RvBinaryOp.SLT,
            TacBinaryOp.SGT: RvBinaryOp.SGT,
        }[instr.op]
        self.seq.append(Riscv.Binary(op, instr.dst, instr.lhs, instr.rhs))
```

其余部分做对应的修改即可。

思考题：短路求值的好处

首先，短路求值可以提高代码执行的效率。如果计算表达式第一部分后可以确定结果，则无需计算第二部分，可以节省处理时间和资源。这也可以被用于使代码更简洁，增加可读性。

其次，有时可以用于错误预防。当评估表达式的第二部分会导致不必要的副作用或错误时，短路评估可以帮助防止错误。例如，在检查 null 或空值时，如果表达式的第一部分计算结果为 true（表示有效值），则无需检查第二部分。这有助于避免潜在的空引用异常或其他错误。