Principles of Compilers
Experiment: Stage-1

Chenghua Liu*
liuch18@mails.tsinghua.edu.cn
Department of Computer Science
Tsinghua University

目录

*刘程华，学号 2018011687

1 实验内容及过程

1.1 step2

step2 中，我们要给整数常量增加一元运算：取负 −、按位取反 ~ 以及逻辑非 !。

我们需要引入新的抽象语法树节点 Unary。在生成 TAC 的过程中，我们需要为运算结点分配一个临时变量，并生成一条指令，该指令根据子结点的临时变量进行计算，将结果赋予该结点的临时变量。对于中间代码指令要选择合适的 RISC-V 指令来完成翻译工作（框架已实现）。

```
#/frontend/tacgen/tacgen.py
#def visitUnary 里的 op 中添加
node.UnaryOp.Neg: tacop.UnaryOp.NEG,
node.UnaryOp.BitNot: tacop.UnaryOp.NOT,
node.UnaryOp.LogicNot: tacop.UnaryOp.SEQZ,
```

1.2 step3

step3 我们要增加的是：加 +、减 −、乘 ∗、整除 /、模 % 以及括号 ( )。

对于算数运算，我们需要引入新的抽象语法树节点 Binary。具体实现和 step2 完全类似，我们只需要选择合适的 TAC 指令。

```
#/frontend/tacgen/tacgen.py
#def visitBinary 里的 op 中添加
node.BinaryOp.Add: tacop.BinaryOp.ADD,
node.BinaryOp.Sub: tacop.BinaryOp.SUB,
node.BinaryOp.Mul: tacop.BinaryOp.MUL,
node.BinaryOp.Div: tacop.BinaryOp.DIV,
node.BinaryOp.Mod: tacop.BinaryOp.REM,
```

1.3 step4

step4 我们要增加的是：

1. 比较大小和相等的二元操作：<、<=、>=, >, ==, !=
2. 逻辑与 &&、逻辑或 ||

对于大于小于逻辑与与逻辑或，一条指令就足够，因此可以像 step3 一样直接添加。

```
#/frontend/tacgen/tacgen.py
#def visitBinary 里的 op 中添加
node.BinaryOp.LT: tacop.BinaryOp.SLT,
```

```
node.BinaryOp.GT: tacop.BinaryOp.SGT,
node.BinaryOp.LogicOr: tacop.BinaryOp.OR,
node.BinaryOp.LogicAnd: tacop.BinaryOp.AND,
```

对于大于等于、小于等于、等于、不等于，需要多条指令来实现。我们在 utils/tac/tacop.py 中的 BinaryOp 中添加对应的 tac 指令（我的命名方式为 XX_HELP），然后和之前一样给出到 TAC 的映射。

```
#/frontend/tacgen/tacgen.py
#def visitBinary 里的 op 中添加
node.BinaryOp.EQ: tacop.BinaryOp.EQ_HELP,
node.BinaryOp.NE: tacop.BinaryOp.NE_HELP,
node.BinaryOp.LE: tacop.BinaryOp.LE_HELP,
node.BinaryOp.GE: tacop.BinaryOp.GE_HELP,
```

在 utils/tac/funcvisitor.py 中我们对 TAC 进行解读：

```
def visitBinary(self, op: BinaryOp, lhs: Temp, rhs: Temp) -> Temp:
    temp = self.freshTemp()
    if op == BinaryOp.EQ_HELP:
        self.func.add(Binary(BinaryOp.SUB, temp, lhs, rhs))
        self.func.add(Unary(UnaryOp.SEQZ, temp, temp))
    elif op == BinaryOp.NE_HELP:
        self.func.add(Binary(BinaryOp.SUB, temp, lhs, rhs))
        self.func.add(Unary(UnaryOp.SNEZ, temp, temp))
    elif op == BinaryOp.LE_HELP:
        self.func.add(Binary(BinaryOp.SGT, temp, lhs, rhs))
        self.func.add(Unary(UnaryOp.SEQZ, temp, temp))
    elif op == BinaryOp.GE_HELP:
        self.func.add(Binary(BinaryOp.SLT, temp, lhs, rhs))
        self.func.add(Unary(UnaryOp.SEQZ, temp, temp))
    else:
        self.func.add(Binary(op, temp, lhs, rhs))
    return temp
```

2 思考题

2.1 step2

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 $-\sim!$ 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。

一个简单的例子为：取 $a = 2^{31} - 1 = 2147483647$，$-\sim a = 2^{31}$，运算过程中发生了越界。

2.2 step3

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 RISCV-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）

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

这时的左操作数和右操作数分别为：

```
int a = 0x80000000;
int b = -1;
```

在我的电脑（x86-64 WSL 20.04 ）上编译运行，得到错误 floating point exception。在 RISCV-32 的 qemu 模拟器中编译运行得到的结果为-2147483648。

2.3 step4

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？

短路求值是一种逻辑运算符的求值策略。只有当第一个运算数的值无法确定逻辑运算的结果时，才对第二个运算数进行求值。例如，当 AND 的第一个运算数的值为 false 时，其结果必

定为 false；当 OR 的第一个运算数为 true 时，最后结果必定为 true，在这种情况下，就不需要知道第二个运算数的具体值。直观的来说，短路求值策略是对程序更加精细化的描述，一个直接的好处是可能省略第二个运算数的求值从而降低计算量。下面给出例子来说明。

```
int a = 0;
if (a && myfunc(b)) {
    do_something();
}
```

在这个例子中，短路求值使得 myfunc(b) 永远不会被调用。这是因为 a 等于 false，而 false AND q 无论 q 是什么总是得到 false。这个特性允许两个有用的编程结构。首先，不论判别式中第一个子判别语句要耗费多昂贵的计算，总是会被执行，若此时求得的值为 false，则第二个子判别运算将不会执行，这可以节省来自第二个语句的昂贵计算。再来，这个结构可由第一个子判别语句来避免第二个判别语句不会导致运行时错误。例如以下例子，短路求值可以避免对空指针进行存取。

```
void * p = NULL;
int ret;
/* ... */
if(p && ret = func(p) ){
    /* ... */
}
/* ... */
```

3 参考

1. step3 思考题参考了 https://stackoom.com/question/38b3w
2. step4 思考题参考 https://en.wikipedia.org/wiki/Short-circuit_evaluation