实验5 目标代码生成

202011081001 卢皓宇 2022年1月5日

实验要求

在词法分析、语法分析、语义分析和中间代码生成程序的基础上，将C−−源代码翻译为MIPS32指令序列（可以包含伪指令），并在 SPIM Simulator上运行

> 面向样例，不好说能不能实际运行/doge

基本要求

将中间代码正确转化为MIPS32汇编代码

“正确”是指该汇编代码在SPIM Simulator（命令行或Qt版本均可）上运行结果正确

附加要求

1. 对寄存器分配进行优化，如使用局部寄存器分配办法等
2. 对指令进行优化

化简要求

- 寄存器的使用与指派可以不必遵循MIPS32的约定。只要不影响在SPIM Simulator中的正常运行，你可以随意分配 MIPS体系结构中的32个通用寄存器，而不必在意哪些寄存器应该存放参数、哪些存放返回值、哪些由调用者负责保存、哪些由被调用者负责保存，等等。
- 栈的管理（包括栈帧中的内容及存放顺序）也不必遵循MIPS32的约定。你甚至可以使用栈以外的方式对过程调用间各种数据的传递进行管理，前提是你输出的目标代码“正确”。

实验分工

- 卢皓宇：基于树形IR完成大多数生成目标代码的程序（模式 -> 代码），并基于样例1和2进行完善和优化；设计查询变量表VarReg以确定变量所属寄存器、暂存当前生成的函数/过程结点以便查询的方法并基本实现；GitHub项目管理
- 姜潮涌：完成确定变量所属寄存器方法findMark的封装，以及RELOP中涉及的常量的管理方法findNum2的封装；完善代码细节
- 段欣然：提出BFS生成各函数/过程的理念并通过队列（链表）实现；完善代码细节
- 杜隆清：修改、增加样例以测试代码鲁棒性，提出修改建议

实验环境

- Bison: v3.5.1
- Flex: v2.6.4
- 虚拟机程序irsim
- 本地程序编写与测试
  - Windows Subsystem for Linux 2: Ubuntu 20.04 LTS
  - gcc: 9.4.0
- 云主机测试

- BNUCLOUD云主机: Ubuntu 18.04 LTS
- gcc: 7.5.0

实验设计

指令选择

由于我们在实验四中采用了树形IR，且并未采用其他数据结构对中间代码进行存储，而是直接依托于语法树完成，所以实验五的实现也基本依托于**语法树**。

相应地，指令选择机制也就是DFS语法树上的结点，以确定该结点及其兄弟孩子是否符合相应的模式。

我们举一个比较复杂的例子：赋值语句，对应到产生式应该是Exp: Exp ASSIGNOP Exp。赋值语句对应指令的可能性相对较多：我们假设已声明变量a，其对应的寄存器是$a

- a=1 -> `li $a, 1`
- a=b -> `move $a, $b`
- a=b+c -> `add $a, $b, $c`（这个可能的写法比较多）

所以我们需要对ASSIGNOP构建出不同模式，每对应一个模式就生成其相应的代码。下面是我们代码的部分展示

```
if(head->child->bro) {
        if(!strcmp(head->child->bro->name, "ASSIGNOP")) {
            // Exp ASSIGNOP Exp
            if(!strcmp(head->child->bro->bro->child->name, "INT")) {
                // Exp(ID) = Exp(INT)
                int mark = findMark(head->child->child->id);
                fprintf(f, "\tli $t%d, %d\n", mark, head->child->bro->bro->child->intValue);

                sprintf(expReg, "t%d", mark);
                return expReg;
            }
            else if(!strcmp(head->child->bro->bro->child->name, "ID") && !head->child->bro->bro->child->bro) {
                // Exp(ID) = Exp(ID)
                int mark = findMark(head->child->child->id);
                fprintf(f, "\tmove $t%d, $t%d\n", mark, findMark(head->child->bro->bro->child->id));

                sprintf(expReg, "t%d", mark);
                return expReg;
            }
            else if(head->child->bro->bro->child->bro->bro->bro) {  // 一定要注意先写长的再写短的
                // ID LP Args RP
                pushQueue(hed, head->child->bro->bro->child->id);
                genExp(head->child->bro->bro, f);
                int mark = findMark(head->child->child->id);
                fprintf(f, "\tmove $t%d, $v0\n", mark); // 赋值

                sprintf(expReg, "t%d", mark);
                return expReg;
            }
```

```
            else if(head->child->bro->bro->child->bro) {
                // 运算式/函数调用赋值给变量
                char tmp[5];
                strcpy(tmp, genExp(head->child->bro->bro, f));

                sprintf(expReg, "t%d", findMark(head->child->child->id));
                if(strcmp(expReg, tmp)) { // 不是自增
                    fprintf(f, "\tmove $%s, $%s\n", expReg, tmp);
                }
                return expReg;
            }
            else {
                printf("no\n");
            }
        }
```

其他的目标代码生成也遵照了类似的设计方式。

> 感觉我们在上一步走偏了，中间代码在这种指令选择机制下好像起不到太大的作用，例如ARG、PARAM、CALL之类的就完全没用到。虽然直接用语法树多少也能做，但这样总感觉差了点意思，而且也不够方便。。。sad

寄存器分配

类似朴素寄存器分配算法。我们没有引入释放寄存器的操作，不过在一些中间变量的处理上（例如上文提到的自增、相同的常量），我们尽可能地节约了寄存器的使用。所以我们也简单介绍一下我们的做法

- 对于变量，我们设置了一个二维char数组VarReg作为变量名表，在建立语法树时就将所有**声明的变量**（函数形参列表除外）按顺序加入该表，以其位置作为对应的寄存器编号；同时引入变量计数Regcnt并同时维护，以确定VarReg的终止位置。当变量被使用时，就从起始位置开始直到Regcnt查找该变量名，并返回其位置作为对应的寄存器编号
- 在前面VarReg的基础上，我们将**第一次被使用的常量**加入该表，但此时不维护Regcnt（为区分变量与常量），后续使用同一常量直接从Regcnt位置开始向后搜索即可找到该常量对应的寄存器编号。当然其实应该也只有RELOP因为伪指令格式要求需要传入常量所属的寄存器

```
char expReg[10]; // Exp中间变量的寄存器名

int findMark(char* id) {
    int mark = 0;
    for(int i = 0; i < Regcnt; i++) {
        if(!strcmp(VarReg[i], id)) {
            mark = i;
            break;
        }
    }
    return mark;
}

int findNum2(int n, FILE* f) {
    if(n == 0){
        fprintf(f, "\tli $t0, 0\n");
        return 0;
    }
    for(int i = Regcnt; i < 20; i ++ ) {
        if(ImmReg[i] == n) {
```

```
            fprintf(f, "\tli $t%d, %d\n", i, n);
            return i;
        }
        else if(ImmReg[i] == 0) {
            fprintf(f, "\tli $t%d, %d\n", i, n);
            ImmReg[i] = n;
            return i;
        }
    }
}
```

> 也是感觉这个步骤树形IR不太好操作，毕竟没有将代码设置成一个线性结构，不知道怎么与课上的基本块、活性分析对应起来。。。而且时间和精力确实比较紧张

函数调用

宽搜：确定各函数/过程翻译顺序

基本思路

- 整个生成目标代码是一个BFS函数/过程队列的过程，初始从main开始
- 遇到调用函数（ID LP RP或ID LP Args RP），将该函数push进队列中（pushQueue方法），等待当前过程生成完毕再去生成被调用函数的目标代码
- write/read函数一开始在文件头部生成，故不考虑

```
struct Queue {
    char name[10];
    struct Queue* nxt;
};

struct Queue* hed; // 队首存main

void pushQueue(struct Queue* Qhed, char* name) {
    if(!strcmp(name, "read") || !strcmp(name, "write")) return;
    struct Queue* tmp;
    tmp = (struct Queue*)malloc(sizeof(struct Queue*));
    strcpy(tmp->name, name);
    tmp->nxt = NULL;
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue*));
    q = Qhed;
    while(q->nxt){
        if(!strcmp(q->name, name)) return;
        q = q->nxt;
    }
    q->nxt = tmp;
}

void bfsgenExtDefList(struct node* head, FILE* fp) {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue*));
    currentFunc = (struct node*)malloc(sizeof(struct node));
    q = hed;
    struct node p = *head;
    while(q){
        strcpy(arg, "");
        // strcpy(stackRet, "");
```

```
        // strcpy(regName, "");

        p = *head;
        while(p.child && strcmp(q->name, p.child->child->bro->child->id)) {
            p = *(p.child->bro);
        }
        currentFunc = p.child->child->bro;
        genExtDef(p.child, fp);
        // if(q->nxt)
        q = q->nxt;
        // else break;
    }
}

void genAsm(struct node* head, FILE* fp) {
    // ...

    hed = (struct Queue*)malloc(sizeof(struct Queue*));
    strcpy(hed->name, "main");
    hed->nxt = NULL;
    bfsgenExtDefList(head->child, fp);
    free(hed);
}

char* genExp(struct node* head, FILE* fp) {
    // ...
    else if(head->child->bro->bro->child->bro->bro->bro) {  // 一定要注意先写长的再写短的
        // ID LP Args RP
        pushQueue(hed, head->child->bro->bro->child->id);
        genExp(head->child->bro->bro, f);
        int mark = findMark(head->child->child->id);
        fprintf(f, "\tmove $t%d, $v0\n", mark); // 赋值

        sprintf(expReg, "t%d", mark);
        return expReg;
    }
    // ...
    else if(!strcmp(head->child->bro->name, "LP") && !strcmp(head->child->bro->bro->name, "RP")) {
            // ID LP RP
        pushQueue(hed, head->child->id);
        fprintf(f, "\taddi $sp, $sp, -4\n");
        fprintf(f, "\tsw $ra, 0($sp)\n");
        fprintf(f, "\tjal %s\n", head->child->id);
        fprintf(f, "\tlw $ra, 0($sp)\n");
        fprintf(f, "\taddi $sp, $sp, 4\n");

        strcpy(expReg, "v0");
        return expReg;
    }
}
```

关于形参

由于能力有限，我们仅考虑了单个参数函数和无参数函数两种情况

- 当一个单参数函数调用一个无参数函数，此时栈中只需要存储返回地址寄存器`$ra`
- 当一个单参数函数调用另一个单参数函数，由于传参寄存器均使用`$a0`，所以除了返回地址`$ra`之外，我们还需要将其压入栈
- 当一个无参数函数调用一个单参数函数，那么这个参数可以由`move $a0, $<register name>`挪至函数参数寄存器中，然后再将返回地址寄存器`$ra`入栈即可

另外值得一提的是，因为递归函数在调用自身时可能不需要赋值，而是直接利用其形参（如`int fact(int n): return n*fact(n-1)`），所以我们在处理递归函数调用自身的目标代码时，会特判删去`move $a0, $<register name>`这一步传参，直接对`$a0`进行操作。

```
if(strcmp(head->child->id, currentFunc->child->id)) { // 如果当前翻译的函数不是递归的，则需单独传参
    fprintf(f, "\tmove $a0, $t%d\n", findMark(head->child->bro->bro->child->child->id));
}

if(!strcmp(currentFunc->child->bro->bro->name, "VarList")) { // 如果当前函数是有参函数
    fprintf(f, "\taddi $sp, $sp, -8\n");
    fprintf(f, "\tsw $a0, 0($sp)\n");
    fprintf(f, "\tsw $ra, 4($sp)\n");
    genArgs(head->child->bro->bro, f);
    fprintf(f, "\tjal %s\n", head->child->id);
    fprintf(f, "\tlw $a0, 0($sp)\n");
    fprintf(f, "\tlw $ra, 4($sp)\n");
    fprintf(f, "\taddi $sp, $sp, 8\n");
}
else { // 如果当前函数是无参函数
    fprintf(f, "\taddi $sp, $sp, -4\n");
    fprintf(f, "\tsw $ra, 0($sp)\n");
    genArgs(head->child->bro->bro, f);
    fprintf(f, "\tjal %s\n", head->child->id);
    fprintf(f, "\tlw $ra, 0($sp)\n");
    fprintf(f, "\taddi $sp, $sp, 4\n");
}
```

实验结果

样例1

```
./compiler test1.cmm out.s
```

```
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0, 4
```

```
    la $a0, _prompt
    syscall
    li $v0, 5
    syscall
    jr $ra
write:
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, _ret
    syscall
    move $v0, $0
    jr $ra
main:
    li $t1, 0
    li $t2, 1
    li $t3, 0
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal read
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    move $t4, $v0
label1:
    blt $t3, $t4, label2
    j label3
label2:
    add $t5, $t1, $t2
    move $a0, $t2
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal write
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    move $t1, $t2
    move $t2, $t5
    add $t3, $t3, 1
    j label1
label3:
    move $v0, $0
    jr $ra

```

样例2

```
./compiler test2.cmm out.s
```

```
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0, 4
```

```
    la $a0, _prompt
    syscall
    li $v0, 5
    syscall
    jr $ra
write:
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, _ret
    syscall
    move $v0, $0
    jr $ra
main:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal read
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    move $t1, $v0
    li $t3, 1
    bgt $t1, $t3, label4
    j label5
label4:
    move $a0, $t1
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal fact
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    move $t2, $v0
    j label6
label5:
    li $t2, 1
label6:
    move $a0, $t2
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    jal write
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    move $v0, $0
    jr $ra

fact:
    li $t3, 1
    beq $a0, $t3, label1
    j label2
label1:
    move $v0, $a0
    jr $ra
    j label3
label2:
    addi $sp, $sp, -8
    sw $a0, 0($sp)
```

```
    sw $ra, 4($sp)
    sub $a0, $a0, 1
    jal fact
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    mul $v0, $v0, $a0
    jr $ra
label3:

```

> 这里多余的label3仍然合法，不影响运行，故没有进行优化

实验反思

1. 树形IR相对较好的一点在于通过DFS识别模式以生成目标代码，相比线性IR朴素的逐句翻译可以节省操作指令数，更加高效；但相应地，在做寄存器分配优化时就更加不直观，不便于实现优化算法，且因为我们没有设计可以完整存储中间代码的数据结构、而是直接依托于语法树，导致中间代码中ARG、PARAM、CALL等关键字没有派上用场
2. 学期共五次环环相扣的实验，让我深刻理解了管理项目的诸多不易/(ㄒoㄒ)/~~但也着实提高了我的代码实现能力（从画饼到做饼，好不好吃另说）。今后继续加油吧，谢谢师兄和老师的指导！