编译原理

MiniDecaf 编译器实验报告 -- STAGE 1

2021010706 岳章乔

一、 思考题

step 2:

要求：对一个$[0, 2^{31} - 1]$内的立即数使用 `~`, `!` , `-` 单目运算符，使得发生运算越界。

解：

`-~2147483647` 。

对 `2147483647` 按位取反，得 `-2147483648` ，然后取负，理应是 `2147483648` ，但这个数超出 `int32` 的表示范围，出现溢出。

step 3:

整数除法的未定义行为。

解：

```c
#include <stdio.h>

int main() {
  int a = -2147483648;
  int b = -1;
  printf("%d\n", a / b); // 按理是 2147483648， 但这个数超出整形范围
  return 0;
}

```

step 4:

短路求值：

例： 下方两行代码都不调用 `foo()` 。

```c
false && foo()
```

```c
true || foo()
```

由于 `foo()` 不一定是常数时间复杂度，因此引入短路，当已知 `foo` 执行与否不影响结果，可以有效减少程序的运行时间，优化执行效能。

正是如此，不应在 `foo` 函数内修改外部变量。

二、实验内容

2.0. 约定

以下的讨论基于上下文无关文法和上下文无关语言。

上下文无关文法是一个四元组：$G = (V, T, P, S)$，V 是非终结符，T 是终结符，P 是产生式，S 是开始符号。

除了另外标明，下文的工作目录是实验框架的 `src` 文件夹。

2.1. 实验需求

实现单目运算符 `~!-` ，和双目运算符 `+-*/%` ，以及逻辑运算符 `==` 、`!=`、`<`、`>`、`<=`、`>=`、`&&` 和 `||`。

2.2. 需求分析

编译过程分为下列的几个部分(句末括号对应下方 mind 编译器的各个阶段)：

- 词法分析(tokenizer)，分离单词
- 语法分析(parser)，生成 AST (1)
- 语义分析(semantical analysis) (2)
- 生成中间代码(3)
- 优化中间代码
- 生成目标代码(5)

mind 编译器提供编译选项 `-l #`

#	1	2	3	4	5
功能	生成 AST	输出符号表	输出中间代码	输出数据流	输出 RISC-V 汇编代码

相应的，

1. 修改词法、语法分析器，在 阶段一 见效；
2. 修改语义分析器，在 阶段二 见效；
3. 修改三地址码生成器，在 阶段三见效；
4. 修改寄存器分配器(目标代码生成器)，在 阶段五 见效。

2.3. 具体实现

由于运算符的实现类似，因此理论上只要实现了一个运算符，其他运算符也能按照同样的规则实现。

下面不区分 step2, step3 和 step4。

2.3.0. 编译器工作原理

先打开 `compiler.cpp` ，观察其编译指令。

```
/* Compiles the input file into the output file.
 *
 * PARAMETERS:
 *   input  - the input file name (stdin if NULL)
 *   result - the output stream
 * EXCEPTIONS:
 *   if any errors occur, the function will not return.
 */
void MindCompiler::compile(const char *input, std::ostream &result) {
```

```
10      // syntatical analysis
11      ast::Program *tree = parseFile(input);
12      err::checkPoint();
13      // semantical analysis
14      buildSymbols(tree);
15      err::checkPoint();
16      checkTypes(tree);
17      err::checkPoint();
18      // translating to linear IR
19      tac::Piece *ir = translate(tree);
20      md->emitPieces(tree->ATTR(gscope), ir, result);
21  }
22
```

上述的代码过程恰好对应 2.2 的说明。

下文提到的编译函数指代上面的 `MindCompiler::compile(const char *, std::ostream &)`

2.3.1. 词法分析

编译函数第 11 行 `parseFile` 调用语法分析器，语法分析器调用词法分析器。

由于在 `frontend/parser.y` 已经定义了终结符，因此只要按定义补全 `frontend/scanner.l` 的单词表即可，例如对加法运算，有

```
1  "+"          { return yy::parser::make_PLUS  (loc);       }
```

后面的返回值，是 Bison 编译器生成器自动生成的单词类，形如 `make_*` 的后缀对应 `frontparser.y` 的终结符(`SUBSECTION 2.1`)。

2.3.2. 语法分析

修改 `frontend/parser.y` 产生式的定义。

由于在 `SUBSECTION 2.2` 已经定义了运算符的结合律和优先级，因此大可直接对各个运算符编写其产生式，单目运算符参考 取负运算符，双目运算符参考加法运算符，后面调用的构造函数对应 `ast/ast.hpp` 上对各个运算符的定义。

例如，对于乘法运算符，有产生式

```
1  | Expr TIMES Expr
2                  { $$ = new ast::MulExpr($1, $3, POS(@2)); }
```

期中 `TIMES` 对应上面的终结符表， `MulExpr` 对应 `ast.hpp` 乘法运算的定义。

到了这个阶段，执行 `./mind <your_code> -l 1` 就能看到更新定义后的语法树了。

2.3.3. 语义分析

编译函数第 14 行 第一次调用语义分析器，进入函数发现实际上在遍历语法树：

```
1  void MindCompiler::buildSymbols(ast::Program *tree) {
2      tree->accept(new SemPass1());
3  }
```

观察后发现这个模块与函数、跳转指令，以及变量声明有关，与运算符关系不大，遂跳过。

编译函数第 16 行 第二次调用语义分析器：

```
void MindCompiler::checkTypes(ast::Program *tree) {
    tree->accept(new SemPass2());
}
```

发现 `SemPass2` 与运算规则有关，遂修改 `translation/type_check.cpp` 。

在词法分析模块，返回的单词同时包含单词的属性。在这个部模块，需要遍历语法树上的运算符节点：对于每一个运算符，其操作数是否符合类型要求（现阶段应为`int`）。

例如，对于减法运算，需要分别检查左、右操作数，实际上是后序遍历：

```
// 1. Define visit method for subtraction.
class SemPass2 : public ast::Visitor {
    // definiton skipped
    virtual void visit(ast::SubExpr *); // add this line to the class
    // definition skipped
};

// 2. Implement the method added in (1).
/* Visits an ast::SubExpr node.
 *
 * PARAMETERS:
 *   e      - the ast::SubExpr node
 */
void SemPass2::visit(ast::SubExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);   // Traverse the left child --
                                    //check the base type of
                                    // the left operand

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);   // Traverse the right child --
                                    // check the base type of
                                    // the right operand

    e->ATTR(type) = BaseType::Int;  // then set the base type of
                                    // this node as int
}
```

到了这个阶段，执行 `./mind <your_code> -l 2` 就能看到更新定义后的符号表了。

2.3.4. 中间代码生成

通过查阅编译函数第19行的`translate` 函数，发现其实际上也是在遍历 AST：

打开 `translation/translation.cpp` 可以得知

```
Piece *MindCompiler::translate(ast::Program *tree) {
    TransHelper *helper = new TransHelper(md);

    tree->accept(new Translation(helper));

    return helper->getPiece();
}
```

可见编译器一边在遍历AST，一边生成三地址码，遍历函数是 `Translation::visit` 因此需要扩展遍历函数，通过查看 `Piece` 发现编译器用链表维护三地址码。

综上，需要修改 `translation/translation.hpp` 和 `translation/translation.cpp` ，把节点翻译为三地址码。

例如，对于乘法运算，有

```
1  // 1. Define visit method for multiplication
2  //    in translation/translation.hpp.
3  class Translation : public ast::Visitor {
4      // skipped
5      virtual void visit(ast::MulExpr *);
6      // skipped
7  };
8  // 2. Implement the visit method in translation/translation.cpp .
9  /* Translating an ast::MulExpr node.
10  */
11 void Translation::visit(ast::MulExpr *e) {
12     e->e1->accept(this);
13     e->e2->accept(this);
14
15     e->ATTR(val) = tr->genMul(e->e1->ATTR(val), e->e2->ATTR(val));
16     // DFS: post-order traversal of the expression tree.
17 }
18
```

第15行，是对 `Piece` 这个链表后方，接一条三地址指令。生成函数通过查看 `trans_helper.hpp` 而得（因为`trans_helper.hpp` 包含 `TransHelper` 类的定义）：

```
1  Temp TransHelper::genMul(Temp a, Temp b) {
2      Temp c = getNewTempI4();
3      chainUp(Tac::Mul(c, a, b));
4      return c;
5  }
```

`getNewTempI4` 生成一个32位的临时变量，`chainUp` 就是上文提到的，具体的“接链表”函数了。

如此一来，执行 `./mind <your_code> -l 3` 就能看到更新定义后的三地址中间代码了。

2.3.5. 中间代码优化

不是本次作业的考核范围，略去。

2.3.6. 目标代码生成

寻览 `md->emitPieces(tree->ATTR(gscope), ir, result);` 发现，

```
1  void RiscvDesc::emitPieces(scope::GlobalScope *gscope, Piece *ps,
2                             std::ostream &os) {
3      // code skipped
4      // for all nodes in the linked list,
5      emitFuncty(ps->as.functy);
6  }
```

对 `emitFunty` 展开，有

```
/* Translates a "Functy" object into assembly code and output.
 *
 * PARAMETERS:
 *   f     - the Functy object
 */
void RiscvDesc::emitFuncty(Functy f) {
    mind_assert(NULL != f);

    _frame = new RiscvStackFrameManager(-3 * WORD_SIZE);
    FlowGraph *g = FlowGraph::makeGraph(f);
    // dataflow analysis

    // pre-process variables (skipped)
    for (FlowGraph::iterator it = g->begin(); it != g->end(); ++it) {
        BasicBlock *b = *it;
        // code skipped
        b->instr_chain = prepareSingleChain(b, g);
        //code skipped
    }


    // display the assembly code
    mind_assert(!f->entry->str_form.empty()); // this assertion should hold for every Functy
    // outputs the header of a function
    emitProlog(f->entry, _frame->getStackFrameSize());
    // chains up the assembly code of every basic block and output.
    for (FlowGraph::iterator it = g->begin(); it != g->end(); ++it)
        emitTrace(*it, g);
}
```

`emitProlog` 显示函数标头。

注意到第 19 行的 `prepareSingleChain` 把三地址码链表生成汇编指令链表：

```
RiscvInstr *RiscvDesc::prepareSingleChain(BasicBlock *b, FlowGraph *g) {
    RiscvInstr leading;
    int r0;

    _tail = &leading;
    for (Tac *t = b->tac_chain; t != NULL; t = t->next)
        emitTac(t);
    // skip
    return leading.next;
}
```

生成过程如下：

```
void RiscvDesc::emitTac(Tac *t) {
    //code skipped
    switch (t->op_code) {
    case Tac::NEG:
        emitUnaryTac(RiscvInstr::NEG, t);
        break;
    case Tac::ADD:
        emitBinaryTac(RiscvInstr::ADD, t);
```

```
        break;
    // TODO
    default:
        mind_assert(false); // should not appear inside a basic block
    }
}
```

在这里，观察到 `emitUnaryTac` 和 `emitBinaryTac` 都生成汇编指令链表节点，对于大部分运算符，模仿上述代码实现即可。

但是，由于 RISC-V 汇编指令与 8086 有一定的差异，因此需要对部分的运算符有特别的处理，例如逻辑或需要分解为一次按位或和一次逻辑非，具体是对最后的 `addInstr` 的调用做出修改。

接着把目光转移到 `emitTrace` 上。

```
void RiscvDesc::emitTrace(BasicBlock *b, FlowGraph *g) {
    // a trace is a series of consecutive basic blocks
    if (b->mark > 0)
        return;
    b->mark = 1;
    // display the section label
    emit(std::string(b->entry_label), NULL, NULL);

    RiscvInstr *i = (RiscvInstr *)b->instr_chain;

    //print out the instructions
    while (NULL != i) {
        emitInstr(i);
        i = i->next;
    }
    // skipped
}
```

对于一个块，先显示块的标签，然后打印每一条指令的汇编码。

打开 `emitInstr` 就能看到打印方法了，这部分仿照加法，实现其他运算符指令的输出即可，例如对于乘法运算，有

```
// TODO
case RiscvInstr::MUL:
        oss << "mul" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;
```

对于其他的运算符对应的汇编指令，可以利用命令行指令 `riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O3 -S <test_code>` 查看。

综上所述，应当修改 `asm/riscv_md.hpp` 和 `asm/riscv_md.cpp` ，而这个部分，把（优化后的）三地址码翻译为 RISC-V 汇编语言。

首先对于 `RiscvInstr::OpCode` ，扩展汇编指令枚举，接着按上述的标了 `TODO` 的部分，补全实现即可。

三、附录

3.1. 汇编指令

涉及到的 RISC-V 汇编指令如下：

运算	汇编指令
`op1 + op2`	`add t2, t0, t1`
`op1 - op2`	`sub t2, t0, t1`
`op1 * op2`	`mul t2, t0, t1`
`op1 / op2`	`div t2, t0, t1`
`op1 % op2`	`rem t2, t0, t1`
`op1 == op2`	`sub t2, t0, t1` `seqz t2, t2`
`op1 != op2`	`add t2, t0, t1` `snez t2, t2`
`op1 < op2`	`slt t2, t0, t1`
`op1 > op2`	`sgt t2, t0, t1`
`op1 <= op2`	`sgt t2, t0, t1` `xori t2, t2, 1`
`op1 >= op2`	`slt t2, t0, t1` `xori t2, t2, 1`
`op1 && op2`	<见 3.2.>
`op1 || op2`	`or t2, t0, t1` `snez t2, t2`
`-op`	`neg t1, t0`
`!op`	`seqz t1, t0`
`~op`	`not t1, t0`

3.2. 讨论

对于逻辑与，至少有三个实现方式：

1. 迪摩根定律；

```
seqz  t0, t0
seqz  t1, t1
or    t2, t0, t1
seqz  t2
```

2. 对两个操作数先取逻辑非，再求两者的逻辑与；

```
snez  t0, t0
snez  t1, t1
and   t2, t0, t1
```

3. 使用跳转语句。

```
snez  t2, t0
bne   t1, 0, .Label
li    t2, 0
.Label:
```

目前使用的是第二种方式。

另一方面，从上述讨论易知，只要对 `riscv_md.cpp` 做出修改，即可生成其他架构的处理器的汇编语言代码。