编译原理课程实践报告

周书予
2000013060@stu.pku.edu.cn

2023 年 6 月 2 日

项目地址: gitlab github (将在 ddl 之后转为 public)

1 编译器概述

1.1 基本功能

本编译器具备如下功能:

1. 运行指令 `./compiler -koopa hello.c -o hello.koopa`, 可以将 `hello.c` 中的 SysY 语言编译为中间表示 Koopa IR, 并输出到 `hello.koopa` 中.
2. 运行指令 `./compiler -riscv hello.c -o hello.S`, 可以将 `hello.c` 中的 SysY 语言直接编译为 RISC-V 汇编, 输出到 `hello.S` 中.
3. 运行指令 `./compiler --help` 或不符合前两者描述的指令时, 将输出帮助文档:

```
usage: ./compiler -koopa input_file -o output_file
       ./compiler -riscv input_file -o output_file
```

之所以没有支持从 Koopa IR 到 RISC-V 汇编的生成, 是因为后半部分的实现仅支持会被前半部分生成的 Koopa IR, 而非所有的 Koopa IR.

1.2 主要特点

平平无奇, 没什么特点.

2 编译器设计

2.1 主要模块组成

- `main.cpp`: 包含编译器的 `main` 函数, 其中分别调用了把 SysY 编译成 Koopa IR 和把 Koopa IR 编译成 RISC-V 的前后两部分.
- `sysy.l`: Flex 源文件, 包含 SysY 语言的词法规则.
- `sysy.y`: Bison 源文件, 包含 SysY 语言的语法规则 (包括应用各种语法规则时采取的语义动作).
- `ast.cpp, ast.hpp`: 包含对抽象语法树的定义和 Koopa IR 生成的代码 (采用的是 on-the-fly 边扫描边生成的方式).
- `util.cpp, util.hpp`: 包含前者需要使用到的一些功能性函数.
- `asm.cpp, asm.hpp`: 包含对内存形式 Koopa IR 的处理, 用于生成 RISC-V 汇编代码.

- `koopa.h`: 包含内存形式 Koopa IR 的定义, 来源.

```
root@ae3e77a80c4d:~/compiler/src# ll
total 100
drwxr-xr-x 1 root root  4096 Apr  3 12:31 ./
drwxr-xr-x 1 root root  4096 May 22 12:27 ../
-rwxrwxrwx 1 root root 16602 May 12 15:08 asm.cpp*
-rwxrwxrwx 1 root root    38 Apr  3 12:33 asm.hpp*
-rwxrwxrwx 1 root root 19196 May  8 15:05 ast.cpp*
-rwxrwxrwx 1 root root  5219 Apr  8 17:49 ast.hpp*
-rwxrwxrwx 1 root root 13943 Mar  8 16:13 koopa.h*
-rwxrwxrwx 1 root root  2009 May 22 12:44 main.cpp*
-rwxrwxrwx 1 root root  1745 May  8 13:35 sysy.l*
-rwxrwxrwx 1 root root 15955 Apr  8 17:56 sysy.y*
-rwxrwxrwx 1 root root  5168 May  8 14:30 util.cpp*
-rwxrwxrwx 1 root root  2040 May  8 14:31 util.hpp*
root@ae3e77a80c4d:~/compiler/src#
```

2.2 主要数据结构

2.2.1 Koopa IR 生成

Koopa IR 的生成主要包含对源代码建立抽象语法树以及遍历抽象语法树生成 IR 的两部分. 仿照实验文档中提供的设计思路, 对于 EBNF 中的每种非终结符 (分别对应于 SysY 中的一种语法结构) 定义了一个类, 详见如下代码片段.

```
class BaseAST;
class ProgramAST;
class FuncDefAST;
class TypeAST;
class FuncParamAST;
class BlockAST;
class ConstDeclAST;
class VarDeclAST;
class ConstDefAST;
class VarDefAST;
class InitValAST;
class StmtAST;
class LValAST;
class ExpAST;
class LogicalOrExpAST;
class LogicalAndExpAST;
class EqExpAST;
class RelExpAST;
class AddExpAST;
class MulExpAST;
class UnaryExpAST;
class PrimaryExpAST;
```

与实验文档略有出入的是, 考虑到不同语法结构间实在缺乏共性, 仅有可能作为 Computation Units 或 Block Items 的 `FuncDefAST`, `ConstDeclAST`, `VarDeclAST` 和 `StmtAST` 继承自公有基类 `BaseAST`.

在描述一种语法结构的 AST 类中, 通常包含

- 指向其下一层 AST 的指针 (由 std::unique_ptr 实现内存管理).
- 其自身具有的属性 (如 identifier, PrimaryExpAST 中可能含有的常数等).
- 一个枚举类对象, 用于标识该 AST 节点将采用 EBNF 中的哪一条规则.
- 对于 EBNF 中的不定长结构 (例如一个 BlockAST 中可能包含若干个 Block Items, 一个 ConstDeclAST 节点会拥有若干 ConstDefAST 子节点), 使用 std::vector<std::unique_ptr<*AST>> 结构来实现管理.

同时, 对每个类实现 DumpKoopa() 方法, 用于以遍历 AST 的形式生成 Koopa IR.

注意到在 Koopa IR 中, 参与二元运算的运算数可能来自寄存器[1], 也可能是立即数. 为了实现对立即数与寄存器计算结构的统一处理, 定义了一种新的数据类型 class Result:

```
class Result {
public:
    enum class ResultEnum {imm, reg} which;
    int val;
    Result() {}
    Result(ResultEnum which_, int val_): which(which_), val(val_) {}
    friend std::ostream & operator << (std::ostream &ofs, Result res) {
        if (res.which == Result::ResultEnum::reg)
            ofs << '%';
        ofs << res.val;
        return ofs;
    }
};
#define Imm(v) (Result(Result::ResultEnum::imm, (v)))
#define Reg(v) (Result(Result::ResultEnum::reg, (v)))
```

其中 val 字段对于立即数表示其值, 对于寄存器表示其编号 (就是名称中 % 后面的数). 同时, 实现计算方法

```
Result calc(std::string oprand, Result s1, Result s2);
```

若参与运算的两数中存在寄存器, 则分配新的寄存器并将运算结果储存其中 (同时输出运算对应的 Koopa IR), 否则会在编译期计算出运算结果, 以立即数的形式返回.

2.2.2 RISC-V 生成

这一部分涉及到的数据结构均在 koopa.h 中被定义.

2.3 主要设计考虑及算法选择

2.3.1 符号表的设计考虑

支持单个作用域的符号表可以简单地使用一个 std::unordered_map 实现, 其 key 为符号的名称 (identifier), value 则被设计为一种处理不同数据类型的新结构:

[1]Koopa IR 中并没有寄存器的概念, 这里使用寄存器一词代指名称可被 %[0-9]|[1-9][0-9]+ 匹配的临时符号, 下文中也将沿用.

```
class DataType {
public:
    enum class DataTypeEnum {const_, var_, pointer_, param_pointer_, func_int, func_void} which;
    int val;
    DataType();
    DataType(DataTypeEnum which_, int const_val_);
};
```

其中 `val` 字段用于记录常量的值, 以及标识指针或参数指针的级数 (得益于 Koopa IR 中语法 `getelemptr` 与 `getptr` 的支持, 在保存指针类型时我们无需记录每一维的大小, 只需要记录这是几级指针. 至于为什么需要区分普通指针与函数参数指针, 与此种种将在编译器具体实现中的 Lv.9 部分详细展开讨论).

对于嵌套作用域下的符号表, 首先可以注意到符号的作用域恰好对应其所属 `Block`, 故可以对每个 `Block` 分别维护符号表, 也即分别维护前文中提到的 `std::unordered_map` 结构. 定义一个 `Block` 的深度为包含其在内的 `Block` 数量加一, 全局变量与函数所在的“`Block`”的深度为 0, 则可以发现任意时刻同一深度只有至多一个 `Block` 的符号表是活跃的. 故可以使用 `std::unordered_map` 的数组实现嵌套符号表, 其以 `Block` 深度为下标, 在进入 `Block` 时新建符号表, 并在离开 `Block` 时清除该表.

当涉及到符号解析时, 只需要按照深度顺序由深及浅查询符号表即可.

此外, 为了避免 Koopa IR 中的同名符号冲突, 在生成 Koopa IR 中的符号名时, 在原有的符号名后额外添加了下划线与当前 Block ID (即以下代码中的 `WRAP` 宏, 注意不能是 Block Depth), 从而做到唯一标识.

```
std::unordered_map<std::string, DataType> Map[1024]; // symbol table for different depth
int BlockID[1024], BlockDepth = 0;
#define WRAP(ident, sub) ((ident) + "_" + std::to_string(sub))
/* ..... */
Result LValAST::DumpKoopa() const  {
    for (int d = BlockDepth; d >= 0; --d) {
        auto it = Map[d].find(this->ident);
        if (it != Map[d].end()) {
            if (it->second.which == DataType::DataTypeEnum::const_) {
                // constant
                return Imm(it->second.val);
            }
            else if (it->second.which == DataType::DataTypeEnum::var_) {
                // variable
                Result res = Reg(RegCount++);
                koopa_inst(res, " = load @", WRAP(this->ident, BlockID[d]));
                return res;
            }
            else {
                assert(it->second.which == DataType::DataTypeEnum::pointer_
                       || it->second.which == DataType::DataTypeEnum::param_pointer_);
                // pointer
                // notice that a array pointer can be partially dereferenced, so "<=" instead of "=="
                assert(this->subscripts.size() <= it->second.val);
                std::vector<Result> subs;
                for (auto &ptr: this->subscripts) {
```

```
                    subs.push_back(ptr->DumpKoopa());
                }
                return koopa_dereference(WRAP(this->ident, BlockID[d]), subs, it->second);
            }
        }
    }
}
```

2.3.2 寄存器分配策略

全部放栈上!

具体来说, 为每个可能产生局部结果的 `koopa_raw_value_t` 结构, 在栈上分配一块 4 bytes 空间, 因而需要维护一个以 `koopa_raw_value_t` 为键值, 以栈偏移量为权值的 `std::unordered_map`. 值得注意的是对于 `alloc` 指令的处理, 除去在栈上分配对应大小的空间之外, 还需要额外分配 4 bytes 用于存放指针, 从而在类型系统上实现统一.

3 编译器实现

3.1 Lv.1 main 函数 & Lv.2 初试目标代码生成

确实没什么可说的.

起初还把 Block Comment 的正则表达式写错了. 一种正确写法是:

```
BlockComment  "/*"([^*]|("*"+[^*/]))*"*"+"/"
```

即, 以斜杠加星号开头, 中间的可多次重复的部分要么是非星号的单个字符, 要么是星号加非星号、斜杠的两字符组合, 最后, 以若干个星号加斜杠结尾.

3.2 Lv.3 表达式

关于对 Koopa IR 中立即数与寄存器的统一化处理已经在前文提及, 在此不多赘述.

这一部分虽然运算种类较多, 但逻辑相近, 仅举 `AddExp` 一例展示处理模式.

```
class AddExpAST {
public:
    enum class AddExpEnum {into_mul, add, minus} which;
    std::unique_ptr<AddExpAST> add_exp;
    std::unique_ptr<MulExpAST> mul_exp;
    Result DumpKoopa() const;
};
```

首先, 在 `sysy.l` 语法分析中, 对于 `AddExp` 的三种可能的产生式, 分别填写 `which`, `add_exp` 以及 `mul_exp` 字段. 产生式中下一级运算的非终结符写在运算符的右侧, 以保证运算的左结合.

```
AddExp:
  MulExp {
    auto ast = new AddExpAST();
    ast->which = AddExpAST::AddExpEnum::into_mul;
    ast->mul_exp = unique_ptr<MulExpAST>($1);
```

```
    $$ = ast;
  } |
  AddExp '+' MulExp {
    auto ast = new AddExpAST();
    ast->which = AddExpAST::AddExpEnum::add;
    ast->add_exp = unique_ptr<AddExpAST>($1);
    ast->mul_exp = unique_ptr<MulExpAST>($3);
    $$ = ast;
  } |
  AddExp '-' MulExp {
    auto ast = new AddExpAST();
    ast->which = AddExpAST::AddExpEnum::minus;
    ast->add_exp = unique_ptr<AddExpAST>($1);
    ast->mul_exp = unique_ptr<MulExpAST>($3);
    $$ = ast;
  };
```

在生成 Koopa IR 时, 分别讨论 `AddExp` 的三种可能的产生式, 利用递归调用的子表达式的结果进行一次立即数或者寄存器的运算即可.

```
Result AddExpAST::DumpKoopa() const {
    if (this->which == AddExpAST::AddExpEnum::into_mul) {
        return this->mul_exp->DumpKoopa();
    }
    Result s1 = this->add_exp->DumpKoopa();
    Result s2 = this->mul_exp->DumpKoopa();
    if (this->which == AddExpAST::AddExpEnum::add) {
        return calc("add", s1, s2);
    } else {
        return calc("sub", s1, s2);
    }
}
```

3.3 Lv.4 常量和变量

常量与变量的出现首先引出了符号表, 同时变量也导致了左值的产生: 必须对 `LValAST` 实现除了 `DumpKoopa()` 之外的, 用于实现 store 的一种方法.

此外, 声明部分首次引入了 EBNF 中的“Extended”: 一个声明语句中会包含若干个定义. 这里采用的处理方式一方面是引入新的非终结符将 EBNF 转换为 BNF, 另一方面使用 `std::vector<std::unique_ptr<*AST>>` 来存储不定长结构.

```
ConstDecl:
  CONST Type ConstDef MoreConstDefs ';' {
    auto ast = new ConstDeclAST();
    ($4)->insert(($4)->begin(), unique_ptr<ConstDefAST>($3));
    ast->const_defs = std::move(*($4));
    $$ = ast;
```

```
  };
MoreConstDefs:
  MoreConstDefs ',' ConstDef {
    ($1)->push_back(unique_ptr<ConstDefAST>($3));
    $$ = $1;
  } |
  %empty {
    $$ = new vector<unique_ptr<ConstDefAST>>();
  };
ConstDef:
  IDENT MoreSubscripts '=' InitVal {
    auto ast = new ConstDefAST();
    ast->ident = *unique_ptr<string>($1);
    ast->subscripts = std::move(*($2));
    ast->init_val = unique_ptr<InitValAST>($4);
    $$ = ast;
  };
```

在这里, MoreConstDefs 对应的返回值是一个指向 std::vector 的指针 (raw pointer), 这个指针指向的对象最终会被绑定到 ConstDeclAST 中的 const_defs 字段上, 遍历该 std::vector, 即可得到该语句中包含的所有常量定义. ConstDefAST 中的 subscripts 字段是 std::vector 形式的数组下标, 先不聊.

3.4 Lv. 5 语句块和作用域

引入语句块 Block 后, 一个 Block 中可以包含若干个 Block Item, 而一个 Block Item 又可以是 ConstDecl, VarDecl 或者 Stmt, 故希望三者可以以统一形式存放在一个结构内. 考虑如下定义 BlockAST:

```
class BlockAST {
public:
    std::vector<std::unique_ptr<BaseAST>> block_items; // could be ConstDecl, VarDecl or Stmt
    void DumpKoopa(const std::vector<std::unique_ptr<FuncParamAST>> &func_params
                        = std::vector<std::unique_ptr<FuncParamAST>>()) const;
};
```

然后让 ConstDeclAST, VarDeclAST 和 StmtAST 都继承自 BaseAST 即可.

你要问那一大长串的 func_param 是什么? 哈哈, 顾名思义, 这是在该 Block 作为函数定义中的那个 Block 时, 由 FuncDefAST::DumpKoopa() 传入的参数, 表示该函数的参数列表, 目的是在 Block 内注册符号表. 空的 std::vector 的默认参数是为了照顾其他 Block: 它们可没有参数列表.

3.5 Lv.6 if 语句 & Lv.7 while 语句

这二位说实话也没有太多含金量, 就放在一起说了.

首先谈谈关于 dangling else 产生二义性的问题. 两眼一闭心无旁骛一气呵成写完 BNF 后编译, bison 会告诉你它发现了 shift/reduce conflicts. 没错, 还是 conflicts. 这时候运行

```
bison -d -o sysy.tab.cpp ../src/sysy.y -Wconflicts-sr -Wconflicts-rr --verbose
```

就能在 build/ 目录下找到 sysy.output 文件. 打开一看, 好家伙, 还真在 if-else 上发生了移入归约冲突!

sysy.output 中的一个片段形如:

```
State 148

    38 Stmt: IF '(' Exp ')' Stmt .
    39     | IF '(' Exp ')' Stmt . ELSE Stmt

    ELSE      shift, and go to state 150
    ELSE      [reduce using rule 38 (Stmt)]
%    $default  reduce using rule 38 (Stmt)
```

一个好消息是 bison 优先采用了 shift, 而这正符合我们的预期, 也就是说我们什么特殊处理都不用做便解决了 if-else 产生的二义性问题.

此外, 前面提到过的另一个 shift/reduce conflict 是

```
State 97

    37 Stmt: LVal . '=' Exp ';'
    78 PrimaryExp: LVal .

    '='       shift, and go to state 125
    '='       [reduce using rule 78 (PrimaryExp)]
%    $default  reduce using rule 78 (PrimaryExp)
```

这其实也是符合我们的预期的.

然后就是关于控制流跳转的处理模式. 其实有点简单, 都不知道该说些什么.

```
template<typename T>
void koopa_print(T arg) {
    koopa_ofs << arg << "\n";
}
template<typename T, typename... Ts>
void koopa_print(T arg0, Ts... args) {
    koopa_ofs << arg0;
    koopa_print(args...);
}
void koopa_br(Result cond, std::string label1, std::string label2, std::string next_label) {
    koopa_print("  br ", cond, ", ", label1, ", ", label2);
    koopa_print(next_label, ":");
}
void koopa_jump(std::string dst_label, std::string next_label) {
    koopa_print("  jump ", dst_label);
    koopa_print(next_label, ":");
}
```

```
void StmtAST::DumpKoopa() const {
    if (this->which == StmtAST::StmtEnum::if_) {
        Result cond = this->exp->DumpKoopa();
        std::string id = std::to_string(++BranchCount);
        if (this->else_stmt != nullptr) {
```

```
            koopa_br(cond, "%then" + id, "%else" + id, "%then" + id);
            this->then_stmt->DumpKoopa();
            koopa_jump("%if_end" + id, "%else" + id);
            this->else_stmt->DumpKoopa();
            koopa_jump("%if_end" + id, "%if_end" + id);
        }
        else {
            koopa_br(cond, "%then" + id, "%if_end" + id, "%then" + id);
            this->then_stmt->DumpKoopa();
            koopa_jump("%if_end" + id, "%if_end" + id);
        }
    }
    else if (this->which == StmtAST::StmtEnum::while_) {
        std::string id = std::to_string(WhileID[++WhileDepth] = ++BranchCount);
        koopa_jump("%while_entry" + id, "%while_entry" + id);
        Result cond = this->exp->DumpKoopa();
        koopa_br(cond, "%while_body" + id, "%while_end" + id, "%while_body" + id);
        this->then_stmt->DumpKoopa();
        koopa_jump("%while_entry" + id, "%while_end" + id);
        --WhileDepth;
    }
    else if (this->which == StmtAST::StmtEnum::break_) {
        assert(WhileDepth > 0);
        static int BreakCount = 0;
        koopa_jump("%while_end" + std::to_string(WhileID[WhileDepth]),
                   "%break" + std::to_string(++BreakCount));
    }
    else if (this->which == StmtAST::StmtEnum::continue_) {
        assert(WhileDepth > 0);
        static int ContinueCount = 0;
        koopa_jump("%while_entry" + std::to_string(WhileID[WhileDepth]),
                   "%continue" + std::to_string(++ContinueCount));
    }
    else ...
}
```

关于 `break` 与 `continue` 的处理, 其实只需要知道当前 while 循环的编号, 从而便可得到需要跳转到的 label. 注意到 while 可能嵌套, 故需要用栈来维护当前 label.

Koopa IR 要求任何“基本块”以 `jump`, `br` 或者 `ret` 结尾, 其中前两者只会在 if 和 while 中出现, 且其后面紧接着的都是明确的 label, 不用为这两种指令担忧. 相比之下 `ret` 是可能由程序凭空产生的, 为了避免一些问题, 一个通用的解决方式是在每个 `ret` 后强行插入 label, 同时在每个函数的末尾再添加一次不跟随 label 的 `ret`, 目的是避免空基本块的出现.

最后是关于短路操作. 涉及到改动的是 `LogicalOrExpAST` 和 `LogicalAndExpAST`, 有点类似于在进行与或运算时加入了一个 if 结构, 以 `LogicalOrExpAST` 为例:

```
Result LogicalOrExpAST::DumpKoopa() const {
```

```
    if (this->which == LogicalOrExpAST::LogicalOrExpEnum::into_logical_and) {
        return this->logical_and_exp->DumpKoopa();
    }
    Result s1 = this->logical_or_exp->DumpKoopa();
    if (s1.which == Result::ResultEnum::imm) {
        if (s1.val != 0)
            return Imm(1);
        else {
            Result s2 = this->logical_and_exp->DumpKoopa();
            return calc("ne", s2, Imm(0));
        }
    } else {
        std::string id = std::to_string(++BranchCount);
        koopa_inst("@result", id, " = alloc i32");
        koopa_br(s1, "%then" + id, "%else" + id, "%then" + id);
        koopa_inst("store 1, @result", id);
        koopa_jump("%end" + id, "%else" + id);
        Result s2 = this->logical_and_exp->DumpKoopa();
        s2 = calc("ne", s2, Imm(0));
        koopa_inst("store ", s2, ", @result", id);
        koopa_jump("%end" + id, "%end" + id);
        Result res = Reg(RegCount++);
        koopa_inst(res, " = load @result", id);
        return res;
    }
}
```

3.6 Lv.8 函数和全局变量

函数出现了!

没有特别本质的内容出现, 只需要对一些细节加以补充:

- 在 `UnaryExpAST::DumpKoopa()` 实现对函数调用表达式的支持.
- 在含参数的函数进入 `Block` 时, 记得对参数分配符号表.
- 需要再次明确一下类型系统: 在 `func foo(@x: i32)` 与 `@x = alloc i32` 中, 前者的 `@x` 类型为 `i32` 而后者为 `*i32`, 因此不能将参数符号名与通过 `alloc` 得到的符号名一概而论. 一种偷懒的统一方式是对所有参数再 `alloc` 一遍, 从而将其类型变为指针.

```
void BlockAST::DumpKoopa(const std::vector<std::unique_ptr<FuncParamAST>> &func_params) const {
    BlockID[++BlockDepth] = ++BlockCount;
    Map[BlockDepth].clear();
    for (auto &ptr: func_params) {
        if (ptr->subscripts.size() == 0) {  // integer
            koopa_inst("@", WRAP(ptr->ident, BlockCount), " = alloc i32\n");
            koopa_inst("store %", WRAP(ptr->ident, BlockCount),
                       ", @", WRAP(ptr->ident, BlockCount));
```

```
            Map[BlockDepth][ptr->ident] = Var;
        }
        else {  // array pointer
            koopa_ofs << "  @" << WRAP(ptr->ident, BlockCount) << " = alloc *";
            /* output type of a param_array, such as [[i32, 2], 3] */
            koopa_param_array_type(ptr->subscripts);
            koopa_ofs << "\n";
            koopa_inst("store %", WRAP(ptr->ident, BlockCount),
                       ", @", WRAP(ptr->ident, BlockCount));
            Map[BlockDepth][ptr->ident] = ParamPointer(ptr->subscripts.size());
        }
    }
    for (auto &ptr: this->block_items) {
        ptr->DumpKoopa();
    }
    --BlockDepth;
}
```

- 编译 RISC-V 时, 要记得把前 8 个参数放在寄存器里, 更多的参数放在栈上. 考虑到可能存在的嵌套函数调用, 可以在进入函数时, 将寄存器中保存的参数复制到栈上, 避免受到后续函数调用的影响.

3.7 Lv.9 数组

大的要来了!

首先是关于多维数组初始化的问题. 通过一些资料查询以及实操, 总结出了如下的赋值规律:

```
/*
* initialize a k-dimension array of size len[0] * len[1] * ... * len[k - 1]
* and write them into buffer
*/
void InitValAST::DumpKoopa(int *len, int k, Result *buffer) const {
    assert(this->which == InitValAST::InitValEnum::list);
    int *suffix_prod = new int [k + 1];
    suffix_prod[k] = 1;
    for (int i = k - 1; i >= 0; --i)
        suffix_prod[i] = suffix_prod[i + 1] * len[i];
    int cnt = 0;
    for (auto &ptr: this->init_vals) {
        if (ptr->which == InitValAST::InitValEnum::single) {
            buffer[cnt++] = ptr->exp->DumpKoopa();
        }
        else {
            int p = k;
            while (p > 1 && cnt % suffix_prod[p - 1] == 0)
                --p;
            // k - p is subarray's dimension, which is <= k - 1
            ptr->DumpKoopa(len + p, k - p, buffer + cnt);
```

```
            cnt += suffix_prod[p];
        }
        if (cnt == suffix_prod[0])
            break;
    }
    delete [] suffix_prod;
}
```

接下来就是如何实现指针解引用的问题. 大方向上需要使用 `getelemptr` 和 `getptr` 指令, 前者会把指针级数降低一级而后者会保持不变.

之前有个伏笔提及普通指针与函数参数指针的区别, 可以在此揭晓: 由于数组参数在传入时缺失第一维的长度, 对于同级的普通指针与参数指针来说, 前者多一维长度, 类型形如 `*[...]` 而后者形如 `**[...]`, 因此在第一轮解引用时, 在 `getelemptr` 和 `getptr` 的选取上会产生区别. 此外, 需要注意对 `*i32` 的解引用需要使用 `load`. 最后就是要考虑到有时候会部分解引用, 因此下标的长度会小于等于实际的指针级数.

```
/*
* dereference a pointer ident on subs
*     ident is a (ty.val + 1)-level pointer, and subs.size() <= ty.val
*     notice that there is a 1-level dereference by default
*
*     when ty.which == pointer_, ident is something like *[...]
*     but when ty.which == param_pointer_, it is like **[...]
*
*     so in the later case, we use "getptr" instead of "getelemptr" in the first round of dereference
*
*     in the last round of dereference, use "load %ptr" instead of "getelemptr %ptr, 0" for i32
*/
Result koopa_dereference(std::string ident, std::vector<Result> &subs, DataType ty) {
    Result ptr;
    for (int i = 0; i < subs.size(); ++i) {
        Result new_ptr = Reg(RegCount++);
        if (i == 0) {
            if (ty.which == DataType::DataTypeEnum::pointer_) {
                koopa_inst(new_ptr, " = getelemptr @", ident, ", ", subs[i]);
            }
            else {
                Result tmp = Reg(RegCount++);
                koopa_inst(tmp, " = load @", ident);
                koopa_inst(new_ptr, " = getptr ", tmp, ", ", subs[i]);
            }
        }
        else
            koopa_inst(new_ptr, " = getelemptr ", ptr, ", ", subs[i]);
        ptr = new_ptr;
    }
    Result res = Reg(RegCount++);
```

```
    if (subs.size() == ty.val) {
        // completely dereferenced
        koopa_inst(res, " = load ", ptr);
    }
    else {
        // partially dereferenced
        if (subs.size() == 0) {
            if (ty.which == DataType::DataTypeEnum::pointer_)
                koopa_inst(res, " = getelemptr @", ident, ", 0");
            else {
                Result tmp = Reg(RegCount++);
                koopa_inst(tmp, " = load @", ident);
                koopa_inst(res, " = getptr ", tmp, ", 0");
            }
        }
        else
            koopa_inst(res, " = getelemptr ", ptr, ", 0");
    }
    return res;
}
```

可以发现通篇基本上没有什么谈论 RISC-V 生成的部分. 一方面是因为有些无趣, 在明确了寄存器分配的摆烂之后就是需要对 Koopa IR 一条一条地生成汇编, 一方面是, 我在 RISC-V 上 WA 了几个测试点还没调出来..... 所以, 受篇幅所限, 对编译器具体实现的介绍便在此告一段落.

4 实习总结

4.1 收获和体会

收获首先是写出一个 (极其简陋的, 毫无错误检查的, 无法通过全部测试点的) 编译器后的成就感.

此外就是编写大型程序的实践. 虽然不能说先前毫无经验, 但这次课程实习无疑也是一次宝贵的经历.(又在 github 上水了点东西.)

~~体会是, 我觉得我在学这门课之前就会写这个 lab, 然后学过这门课似乎并没有让我更会写?~~

4.2 学习过程中的难点，以及对实习过程和内容的建议

文档写得很好! 引导合理, 描述清晰, 对具体问题的讲解也很细致, 尚不乏诙谐幽默, 给作者好评!

4.3 对老师讲解内容与方式的建议

我不好说