

编译Project1报告

0. 小组成员及分工

- 尹子昊：Parse Json Input、Form IR Tree、最终测试（对应报告2.1、2.2.1、2.2.2）
- 王育民：boundcheck、生成加减项并组装成最后的S（对应报告2.2.3、3.2）
- 韩佳衡：后端实现，将抽象语法树翻译为目标代码，代码是CCPrinter.cc和CCPrinter.h（对应于报告2.3、3.1）
- 徐德民：参与讨论，造case测试，与助教沟通询问项目存在问题。

项目链接：<https://github.com/GoldExcalibur/Compiler-Project1-2020>

1. 任务描述

1.1 输入表达式文法

P为开始符号。终结符号为Id（字母、数字、下划线组成的非数字开头字符串），FloatV（仅考虑小数点表示法，不考虑科学计数法，无起始0），IntV（仅含数字，无起始0），分别表示变量名（可能是张量变量的名字或标量变量的名字），浮点数常量（32位），整数常量（32位）。其余符号解释：

S：语句
LHS：左值
RHS：右值
TRef：张量引用
SRef：标量引用
Const：常量
CList：常量列表
AList：变量列表

```
P ::= P S | S
S ::= LHS = RHS ;
LHS ::= TRef
RHS ::= RHS + RHS
      | RHS * RHS
      | RHS - RHS
      | RHS / RHS
      | RHS % RHS
      | RHS // RHS
      | (RHS)
      | TRef
      | SRef
      | Const
TRef ::= Id < CList > [ AList ]
SRef ::= Id < CList >
CList ::= CList , IntV | IntV
AList  ::= AList , IdExpr | IdExpr
IdExpr ::= Id | IdExpr + IdExpr | IdExpr * IntV | IdExpr // IntV |
```

```
IdExpr % IntV | (IdExpr)
Const ::= FloatV | IntV
```

另外，虽然文法允许变量名字出现在`AList`中，但是在语义上这是不允许的，所以在测试例子中不会出现这种输入。

1.2 问题描述

针对上述文法接收的所有表达式，生成对应的C/C++源代码。具体而言，利用项目中已有的IR框架表示出想要生成的程序，然后改写IRPrinter，使得打印出来的代码是符合C/C++语法的代码。难点在于利用词法分析、语法分析技术自动化地完成用IR框架表示程序这一步骤。

1.3 解决思路

分以下三步

- Parse Input Json Files
 - 记录对应case的名字，输入输出的Id，数据类型以及要分析的语句
 - 完成词法分析
- Form IR Tree
 - 根据词法分析结果进行语法分析
 - 结合利用自顶向下和自底向上的语法分析技术，构建抽象语法树
 - 设计SDD和SDT，对应节点完成对应的生成中间代码工作
 - 必要时调用IR Visitor获取节点信息，调用IR Mutator更改信息（Project1用不到）
- Generate C/C++ source code
 - 重新实现一个IRVisitor的子类CCPrinter
 - 遍历上述语法树的同时，打印出C/C++源码

2. 代码开发

2.1 Parse Input Json Files

```
class Case{
public:
    std::string name;
    std::vector<std::string> ins;
    std::vector<std::string> outs;
    std::string instr;
    std::string outstr;
    std::string data_type;
    std::string kernel;
}
```

定义Case类如下，通过对json文件不断执行getline函数，一行行读取分析后填写进对应属性。读取后可以确定以下事情，为之后的语法分析做准备：

- 左右值的最终类型，即为Case中的data_type。
- 输出代码的文件名，即为Case中name.cc。

- instr和oustr记录了输入和输出，在去重后，生成代码的函数签名的参数部分得到解决。

2.2 Form IR Tree

形成IR树的工作大体分为两步，首先建立抽象语法分析树，在建过程中存储必要信息。然后再遍历该树结构，以使得每个节点make出对应IR框架中的节点类指针。

2.2.1 build AST

总体而言，结合了自顶向下和自底向上的分析方法。

- P：不断寻找";",将代码分割成一个个S
- S：寻找"="，将代码分析为 LHS, =, RHS三部分
- LHS：直接分析为TRef
- RHS：类似中缀转后缀算法的思想，可以自底向上地对由TRef、SRef、Const组成的算术表达式完成分析。
- TRef: 寻找"<>"和"[]"，分析出Id, AList, CList
- SRef: 寻找"<1>", 分析出Id即可
- AList、Clist: 不断寻找",", 分析出index和其对应的dom上界（下界是0）

设计的AST节点结构如下：

```
class AST {
public:
    nodetype t;           // type of this node
    std::string str;       // str of this node
    std::vector<AST> child;
    AST* father;
    Expr ep;
    std::vector<Stmt> sp;
    Group gp;
}

// 0P 1S 2LHS 3RHS 4TRef 5SRef 6Const 7CList 8AList 9 Op 10 IdExpr 11 Id
enum nodetype {P, S, LHS, RHS, TRef, SRef, Const, CList, AList, Op, IdExpr, Id};
```

其中：

- t是节点对应的枚举类型
- str是该节点要表达的字符串
- child是语法分析出的儿子节点数组
- father是指向父节点的指针
- 指针ep, gp以及Statement的数组sp在后续SDT中用到。

2.2.2 design SDT

对上述建成的AST进行后序遍历，待子节点make出对应的IR节点，再利用子节点的结果完成自身的make。同时对于boundcheck等有关index范围的工作，还需要查看兄弟节点信息。

- P: 每条S make成功即可。

- S：先声明一个temp数组，之后完成temp = 0, 每个最外层加减项的loop, LHS = temp的make，最后嵌套入LHS对应index的最外层Loop。
- LHS：等待儿子节点TRef的make，然后再make自身。
- RHS：根据子节点中操作符的类型，找到对应的BinaryOpType, 完成Binary::make。此时对于每个最外层的“加减项”，还要记录下需要后续Loop的index。
- TRef: CList, Alist make后，完成Var::make。
- SRef: CList (“<1>”) make后，完成Var::make(shape是{1}, index为空)
- CList: 每个index的upper bound, make成一个Expr(int)
- AList: 查看兄弟节点CList中信息，推断出上下界，完成Dom::make

2.2.3 boundcheck

在后序遍历的同时我们也可以实现boundcheck。主要思路：后序遍历的时候在树的低层（TRef节点）收集它的儿子也就是Alist和Clist的信息，并把每个index（包括复合的index，例如i + j）的str、expr、和范围存入本节点的成员boundcheck_list和boundcheck_value中，这是两个map。相同的index的范围取小的那个。同时也维护了叫做expr_index_list的map(std::map<std::string, Expr>)存入当前节点中所有index的str和expr(与前者不同在于它只存单个的index，例如是 [i + j] 的话，map里面是[i]和[j]两个index)。这个结构是为了之后在后序遍历到节点S构造加减项的时候提供信息，因为加减项遍历的index是加减项本身有但是LHS中没有的index。

```

/**类AST中的public成员
std::map<std::string, Expr> boundcheck_list;
std::map<std::string, int> boundcheck_value;

/**Tref中搜集儿子信息构造boundcheck_list和boundcheck_value
std::map<std::string, Expr>::iterator j;
for(int i = 0; i < child[2].child.size(); ++i){
//把Alist中的所有儿子的str和ep存入boundcheck_list，用于加减项的boundcheck
    if(boundcheck_list.find(child[2].child[i].str) == boundcheck_list.end()) {
        boundcheck_list[child[2].child[i].str] = child[2].child[i].ep;
        boundcheck_value[child[2].child[i].str] = int(child[1].child[i].value);
    } else if (boundcheck_value[child[2].child[i].str] >
int(child[1].child[i].value))
    {
        //如果是相同的index且限制的范围更小则更改value
        boundcheck_value[child[2].child[i].str] = int(child[1].child[i].value);
    }
    // 把Alist中的所有儿子的expr_index_list合并存入expr_index_list
    for (j = child[2].child[i].expr_index_list.begin(); j !=
child[2].child[i].expr_index_list.end(); ++j) {
        if(expr_index_list.find(j->first) == expr_index_list.end())
expr_index_list[j->first] = j->second;
    }
}

/**把RHS儿子的boundcheck_list和boundcheck_value合并给自己
if(child.size() == 3) {
    // boundcheck_list
    for (std::map<std::string, Expr>::iterator i =
child[0].boundcheck_list.begin(); i != child[0].boundcheck_list.end(); ++i) {
        if(boundcheck_list.find(i->first) == boundcheck_list.end()) {

```

```

        boundcheck_list[i->first] = i->second;
    }
}
for (std::map<std::string, Expr>::iterator i =
child[2].boundcheck_list.begin(); i != child[2].boundcheck_list.end();++i) {
    if(boundcheck_list.find(i->first) == boundcheck_list.end()) {
        boundcheck_list[i->first] = i->second;
    }
}
//boundcheck_value
for (std::map<std::string, int>::iterator i =
child[0].boundcheck_value.begin(); i != child[0].boundcheck_value.end();++i) {
    if(boundcheck_value.find(i->first) == boundcheck_value.end()) {
        boundcheck_value[i->first] = i->second;
        //std::cout << "((((((((((((((((((()))))))))"))))" << i->second <<
std::endl;
    }
}
for (std::map<std::string, int>::iterator i =
child[2].boundcheck_value.begin(); i != child[2].boundcheck_value.end();++i) {
    if(boundcheck_value.find(i->first) == boundcheck_value.end()) {
        boundcheck_value[i->first] = i->second;
        //std::cout << "((((((((((((((((((((((()))))))))"))))" << i->second <<
std::endl;
    }
}
} else {
    for (std::map<std::string, Expr>::iterator i =
child[0].boundcheck_list.begin(); i != child[0].boundcheck_list.end();++i) {
        if(boundcheck_list.find(i->first) == boundcheck_list.end()) {
            boundcheck_list[i->first] = i->second;
        }
    }
    //boundcheck_value
    for (std::map<std::string, int>::iterator i =
child[0].boundcheck_value.begin(); i != child[0].boundcheck_value.end();++i) {
        if(boundcheck_value.find(i->first) == boundcheck_value.end()) {
            boundcheck_value[i->first] = i->second;
        }
    }
}
}

```

在TRef的父辈节点——S、LHS、RHS节点中，我们就把其儿子节点的boundcheck_list和boundcheck_value合并来作为本节点的boundcheck_list和boundcheck_value。这样TRef以上所有节点中的boundcheck_list和boundcheck_value就表示了本身节点构造boundcheck所需要的index和范围信息。expr_index_list也是一样的操作。之后接着的就是在树的S节点进行构造构造加减项、boundcheck等操作。即在AST类的成员函数void IR_S()中。

2.3 Generate C/C++ source code

从IR结点的抽象语法树 (AST) 生成C/C++目标代码，这个过程属于后端。我们采用的方法是自顶向下地遍历AST，在每个节点执行相应的翻译动作。

具体来说，我们参考了助教提供的IRPrinter的实现方法，从IRVisitor类继承产生了CCPrinter类。该类主要是实现了对visit方法的在各个IR节点上的重载，并利用了预先实现好的visit_expr、visit_stmt等方法来实现对AST的自顶向下的遍历过程。

例如，当访问Move节点时，分别调用左值和右值的visit_expr方法（实际上也就是间接调用了访问左右值节点的visit方法），另外在其中输出一个“=”，这就构成了一个符合C/C++语法的赋值语句。其他IR节点也是类似的SDT实现方法，在这里就不再一一说明了。

当然，在实现的过程中使用了一些工具函数，例如

- (1) print_indent(), enter(), exit()等方法和int indent变量是用来调整缩进（代码很简单，在此不再赘述）。
- (2) declaretemp()方法是用来生成将temp变量初始化的代码。其实就是产生一个形如

```
float temp[16][32] = {};
```

这样的声明语句。需要注意的是，这里temp的类型和形状是直接按照outs[0]来生成的，所以需要保证测试用例的json文件中outs向量非空。

- (3) arglist方法是用来生成函数签名的参数列表（在3.1部分会作进一步阐述）。

最后要指出的是，后端提供给前端的接口是

```
std::string print(const Group&);
```

只需要将前端得到的指向kernel的Ref类指针传递给CCPrinter实例的print方法，就可以得到string形式的c/c++代码。最后将string输出到指定文件夹下，就可以被run.cc方法调用了。

3. 实例测试

3.1 input和output重复的情况

在后端的实际实现中，发现测试用例可能存在以下情况：

```
{
  "name": "kernel_example",
  "ins": ["A", "B"],
  "outs": ["A"],
  "data_type": "float",
  "kernel": ""
}
```

如果按照最初的参数列表实现方法——依次将ins和outs打印出来——那么在上述例子中，参数A将会重复出现。因此需要在打印参数之前对参数列表进行去重，也就是说前面出现过的后面就不要再打印出来了。

明白了任务要求，实现起来很简单。构造了一个

```
std::vector<std::string> inputnames
```

的中间变量，用来在每打印出一个参数时记录其名字，然后当要访问输出变量时，需要先检查其是否在inputnames中出现过。

3.2 加减项问题

对于每个S我们看成 $LHS = RHS$ ，RHS是由多个加减项构成的。 $RHS \rightarrow \{P[0], P[1], P[2], P[3]...\}$

P可以是RHS、TRef、SRef、Const中的某类。我们把S解释为

```
for(index of LHS){
    (band_check for LHS) {
        temp[{index}] = 0;
    }
    //如果加减项中有n个是RHS或TRef,m个是SRef或Const
    //对于那n个加减项
    for (index of P[0] - index of LHS){
        (band_check for P[0]) {
            temp[{index}] = temp[{index}] +/- P[0][{index}];
        }
    }
    ...
    for (index of P[n] - index of LHS){
        (band_check for P[n]) {
            temp[{index}] = temp[{index}] +/- P[n][{index}];
        }
    }
    //对于那m个加减项
    (band_check for P[n+1] - index of LHS) {
        temp[{index}] = temp[{index}] +/- P[n+1][{index}];
    }
    ...
    (band_check for P[n+m] - index of LHS) {
        temp[{index}] = temp[{index}] +/- P[m+n][{index}];
    }
}
for(index of LHS) {
    (band_check for LHS) {
        LHS[{index}] = temp[{index}];
    }
}
```

首先我们得把S中所有得加减项找出来，我们在IR_S中调用函数find_item(&(child[1]), 0);0表示当前这个节点外面是正的，如果是1则表示为负，这个函数会从树上的这个S节点的RHS子节点(对应传入的是child[1]的指针)出发开始找加减项，并把找到的加减项存入全局变量for_temp_list（这是一个vector）中，IR_S构建完之后会清空这个vector以免影响下一个S的构造。找加减项的主要思路是递归，如果当前节点有三个儿子，即可被分解为A op

B. 若op是*或者/号，当前这个节点就是一个加减项，我们把它压入for_temp_list。如果操作符号op是+或者-号，那我们继续find_item()下面的两个儿子，左儿子的正负为本节点的正负，右儿子的正负是op的正负。我们find_item(这个儿子)的时候，如果发现这个儿子只有一个儿子，即它不能在分解，它也是一个加减项，不用再继续向下找。否则向上面一样再判断op。（大概思路是这样，还有一些特殊情况的判断具体看代码。比如当前节点如果有括号包住，那它不管是什么也算一个加减项，还有右值可能直接是一个Const等等）


找到加减项目之后生成一个结构，压入for_temp_list (vector) 中，存入的信息包括，节点的str, expr, for循环涉及到的index信息、boundcheck涉及到的index信息、这个加减项节点的类型、这个加减项的正负。

```
void find_item(AST *RHS, int p_or_n) {
    if (RHS->child.size() == 3 && RHS->t == 3) {
        std::string a = RHS->str;
        erase_blank(a);
        if(a[0] != '(' || a[a.size() - 1] != ')') {
            if (RHS->child[1].str[0] == '+') {
                find_item(&(RHS->child[0]), p_or_n);
                find_item(&(RHS->child[2]), 0);
            }
            else if (RHS->child[1].str[0] == '-') {
                find_item(&(RHS->child[0]), p_or_n);
                find_item(&(RHS->child[2]), 1);
            }
            else {
                Temp_for temp(RHS->str, p_or_n, RHS->ep, RHS->expr_index_list,
RHS->t, RHS->boundcheck_list, RHS->boundcheck_value);
                for_temp_list.push_back(temp);
            }
        } else {
            Temp_for temp(RHS->str, p_or_n, RHS->ep, RHS->expr_index_list, RHS->t,
RHS->boundcheck_list, RHS->boundcheck_value);
            for_temp_list.push_back(temp);
        }
    }
    else {
        if (RHS->t == 4 || RHS->t == 5 || RHS->t == 6 )
        {
            Temp_for temp(RHS->str, p_or_n, RHS->ep, RHS->expr_index_list, RHS->t,
RHS->boundcheck_list, RHS->boundcheck_value);
            for_temp_list.push_back(temp);
        } else {
            find_item(&(RHS->child[0]), p_or_n);
        }
    }
}
```

最后根据这个for_temp_list中存入的加减项信息我们就可以在IR_S中完成组装，完成S这个IR节点的组装。

3.3 最终测试结果


```
mkdir build  
cd build  
cmake ..  
make -j 4  
./project1/test1
```

即可看到测试结果如下图：result 可以看到全部6个case包括example都生成成功。