编译大作业第二部分报告

王雯琦 1700012879 王晨茜 1700013013 张佳琪 1700012888

1.设计思路

本次project2的设计思路是，首先构造正向传播表达式的IR树，然后通过IRMutator重构得到反向传播表达式的IR树，再通过IRPrinter翻译出c/c++代码。其中构造正向表达式IR树的方法基本与project1中的一致，只有部分细节地方的修改。

在重写IRMutator中，主要完成等式左侧结点与被求导结点的对调，以及下标和一些特殊情况的处理（如乘方处理），返回新的IR结点。

在重写IRPrinter过程中，继承project1中已实现的IRPrinter并进行部分重写，主要完成等式中常数项的处理，以生成最终的求导代码。

2.实现流程

2.1重写IRMutator

主要有以下属性方便IR树的重构：

- mode：用来表示当前的模式，不同的模式下对同一个结点会有不同的操作
- new_indexs：下标变换可能引入新下标，它记录所有的新下标
- match：在下标变换中可能需要将某些下标替换成相应的运算式，match建立了需要被替换的下标名到相应运算式的映射
- match_dom：与match配合，match_dom建立了下标名到下标范围的映射
- const_match：在下标变换中可能出现标识符与常数的运算，替换后，与该标识符相关的其余下标也要做变换，const_match建立了标识符到下标的映射
- extra_index：当存在下标只出现在赋值语句右侧而不出现在左侧时，实际上是一个求和，extra_index包含了所有只出现在赋值语句右侧的下标，生成代码时需要为它们生成一个内循环
- input_var：记录正向传播表达式的输入有哪些在反向传播表达式中仍然需要

下面是主要的设计和实现：

- 对调正向传播表达式的输出和待求导的变量，同时修改名字为"d+原名字"，并做下标变换，以形成相应的反向传播表达式：
    - 在Move结点进行对调，首先设置mode为1，此时需要分别获取正向传播表达式的输出和待求导的变量，同时要改变名字并做下标变换
    - 然后设置mode为2，此时进行**对调**，由于只考虑正向传播表达式有且仅有一个输出的情况，因而可能出现待求导的变量出现多次，都需要换成输出变量导数但下标略有不同的情况，如case10，对这种情况的处理也会做详细说明
    - 恢复mode为初始值0，并返回新生成的Move结点
- 下标变换：
    - 由于要求分析出来的求导表达式左侧的下标索引不能有运算，而目标表达式的左侧即为待求导的变量，因而需要将待求导变量带运算的下标索引整体替换成一个新的索引，例如将p+r替换为_p，这个工作在mode为1访问待求导变量的args时做，访问时设置mode为3

    - 只考虑二元运算的情况，处理运算是在Binary结点。以加法为例，首先生成一个新的下标，并将其加入new_indexs中，然后判断第二个操作数是一个标识符还是常数，若为标识符则为该标识符生成新的表示，并记录在match中，例如在p+r替换为_p的例子中，有match[r]对应于(_p-p)的表达式结点，若为常数，则在const_match中添加一项
    - 特别地，对于某些有特别意义的下标运算，例如case8，当待求导的变量有其中两维分别是同一标识符除/模同一常数，则前者的含义是商，后者是余数，那么可以用不同的下标来替换它们，而原标识符变换为（商*常数+余数）
- 一对多对调：
    - 为方便描述，设正向表达式的输出变量为A，待求导变量为B
    - 一对一的对调是直观的，只需注意要将表达式下标中所有出现在match中的标识符替换掉
    - 但存在一对多的情况是，B在正向表达式中出现多次，并且每次出现的下标还不同，若是单个下标或是不带常数的运算式，则与一对一的情况一样，直接对调即可，若是带常数的运算式，则须根据运算式**反解**出对调后dA的下标表达式
    - 事先记录正向表达式中每个B的下标，在进行对调时，首先判断A的某下标是否出现在const_match中，若否则同一对一一样处理，若是则将mode设为5，访问事先记录的下标
        - 若为单个下标，返回对应的新下标
        - 若为加法运算，返回对应的新下标-常数
        - 若为减法运算，反对对应的新下标+常数
        - ...
    - 需要注意反解后dA下标的范围，为**每个dA**生成一个Select语句，保证不会数组越界
    - 需要注意所有根据match替换的下标表达式的范围，为**整个Move表达式**生成一个Select语句
- 隐式求和：
    - 当存在下标只出现在赋值语句右侧而不出现在左侧时，实际上是一个求和，需要为extra_index里的下标生成一个求和的内循环
    - 首先创建一个临时变量temp，赋值为0，然后是所有extra_index中下标的内循环，循环体内即是Move语句，将Move调整为temp = temp + src，循环结束后再将temp写入到dst
    - 对于外循环，去除所有出现在match，extra_index和const_match中的下标，因为它们或被替换掉，或已经出现在内循环
- 多个待求导的变量：
    - **每次只**考虑一个待求导的变量，对正向表达式的IR树进行重构得到一个相应的反向表达式
- 函数标签声明顺序的调整：
    - 为正向表达式生成的函数标签中，参数的声明顺序为先输入后输出
    - 反向表达式的函数标签根据正向表达式的进行调整，首先是出现在input_var中的输入，然后是改名为"d+原名字"的输出，最后是改名为"d+原名字"的待求导变量
- 对乘方的特殊处理：
    - 由于乘方的存在，单纯对调结点并调整下标并无法完成对其求导，故在Binary结点对其进行特殊处理，如果当前为被求导变量的乘方运算，则生成其相加的结点并于等式左侧结点相乘，例如B = A * A，则会先生成A + A结点，并返回dB * (A + A)结点。

2.2重写IRPrinter

增加以下属性便于常数的处理：

- op_flag：整型变量，初始化为-1，若当前运算符为Add或Sub，该变量为0；若当前运算符为Mul，该变量为1；若当前运算符为Div，该变量为2；若当前运算符为Mod，该变量为3；若当前运算符为And或Or，该变量为4。
- index_flag：布尔型变量，初始化为false，若当前表达式的计算在下标内或选择语句时，该变量为true。

主要对一下几个结点进行重写：

- Var

  在处理Var的下标时，将index_flag设为true，当处理结束时，将index_flag设为原始值。

- Select

  在处理Select结点之前，将index_flag设为true，当处理结束时，将index_flag设为原始值。

- Binary

  对于不同的运算符，对op_flag进行不同的赋值，并在函数返回之前，将op_flag置为-1

- IntImm、UIntImm、FloatImm

  若op_flag为0且index_flag为false（即当前运算符为Add或Sub且运算不在下标或Select结点），则输出0.0，否则输出原值

2.3可行性与正确性解释

下面以case1为例解释该技术的可行性与正确性。

```
{
    "name": "grad_case1",
    "ins": ["A", "B"],
    "outs": ["C"],
    "data_type": "float",
    "kernel": "C<4, 16>[i, j] = A<4, 16>[i, j] * B<4, 16>[i, j] + 1.0;",
    "grad_to": ["A"]
}
```

由上述代码可以看出，原始表达式为：$C[i][j] = A[i][j] * B[i][j] + 1.0$，求导对象为A，根据上述技术，应将等式左侧的C结点与等式右侧的A结点交换，并对其的名称进行相应的变化（该过程可以通过两次遍历语法树实现，第一遍记录要交换的结点，第二遍进行结点的交换），此时生成的表达式应为：$dA[i][j] = dC[i][j] * B[i][j] + 1.0$，由于在求导过程中，原始表达式中加或减的常数求导后为0，故经过IRPrinter的处理，该表达式中的1.0变为0.0输出，生成表达式：$dA[i][j] = dC[i][j] * B[i][j] + 0.0$。

而根据求导的数学方法有：

$$dA[i,j] = \frac{\partial \text{loss}}{\partial A[i,j]} = \frac{\partial \text{loss}}{\partial C[i,j]} \times \frac{\partial C[i,j]}{\partial A[i,j]} = dC[i,j] \times B[i,j]$$

可见，根据上述技术得到的表达式，与通过数学方法推导出的求导表达式可以得到相同的结果。

最终生成的自动求导代码为：

```
#include "../run2.h"

void grad_case1(float (&B)[4][16], float (&dC)[4][16], float (&dA) [4][16]) {
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 16; ++j) {
      float temp = 0;
      temp = (temp + (dC[i][j] * B[i][j] + 0.0));
      dA[i][j] = temp;
    }
  }
}
```

3.实验结果

通过最终程序的运行，可以看出，10个例子均可正确通过。

4.成员分工

重写IRMutator：王晨茜、王雯琦、张佳琪

重写IRPrinter：王雯琦

5.总结

- 在project1和project2的实现过程中，都要从最初的json中识别出kernel，进行词法分析和语法分析，构建IR树。此部分用到了编译课程中词法分析、语法分析以及一些SDT部分的知识：
  - 在parser.cc中实现
  - 实现了词法分析器，其中定义了Token类来表示词法单元，读取输入字符串，识别出当前的词法单元
  - 将文法改写为LL(1)文法，使用递归下降子程序法来构建表达式的IR树，在每个非终结符添加一系列动作，返回相应的构造出的结点
- 在重写IRPrinter和IRMutator的过程中，运用到了编译课程中学习到的有关SDT的知识，即需要根据需求在结点添加相应的动作：
  - 在solution2.cc中实现
  - 在IRMutator中要考虑在哪些结点要做哪些转化，从而构造出反向传播表达式的IR树。使用递归下降函数法来实现，即在某结点的函数体内，首先调用子结点相应的函数，并记录返回值，得到所有转化后的子结点以后，创建转化后的新结点并返回，例如在Move结点：

```
Stmt visit(Ref<const Move> op) override {
        Expr new_src;
        Expr new_dst;
        mode = 1;       //设置mode为1，分别获取正向表达式输出和待求导变量
        mutate(op->src);  //访问右侧子结点
        mutate(op->dst);  //访问左侧子结点
        mode = 2;   //设置而mode为2，进行对调

        //当产生下标变换时，为防止数组越界而生成select语句
        if (!match.empty()) {
            //...
```

```
        }
        //通常的情况（为简洁这里只是一个简化的表述）
        else {
            new_src = mutate(op->src);  //新的赋值表达式右侧结点
            new_dst = mutate(op->dst);  //新的赋值表达式左侧结点
        }
        mode = 0;  //恢复mode为初始值0
        return Move::make(new_dst, new_src, op->move_type);  //返回转化后的新结点
}
```

- 在IRPrinter中需要考虑对各个结点如何翻译，从而生成正确的c/c++代码。使用on-the-fly的方法实现，即在遍历表达式语法树的过程中**逐步**生成相应的代码，例如在Binary结点：

```
void visit(Ref<const Binary> op) {
        if (op->op_type == BinaryOpType::Add || op->op_type == BinaryOpType::Sub){
            oss << "(";  //为加法和减法生成括号
        }
        (op->a).visit_expr(this);  //访问第一操作数
        if (op->op_type == BinaryOpType::Add) {
            op_flag = 0;
            oss << " + ";  //若为加法运算生成"+"符号
        } else if (op->op_type == BinaryOpType::Sub) {
            op_flag = 0;
            oss << " - ";  //若为减法运算生成"-"符号
        } else if (op->op_type == BinaryOpType::Mul) {
            op_flag = 1;
            oss << " * ";  //乘法
        } else if (op->op_type == BinaryOpType::Div) {
            op_flag = 2;
            oss << " / ";  //除法
        } else if (op->op_type == BinaryOpType::Mod) {
            op_flag = 3;
            oss << " % ";  //模
        } else if (op->op_type == BinaryOpType::And) {
            op_flag = 4;
            oss << " && ";  //与
        } else if (op->op_type == BinaryOpType::Or) {
            op_flag = 4;
            oss << " || ";  //或
        }
        (op->b).visit_expr(this);   //访问第二操作数
        if (op->op_type == BinaryOpType::Add || op->op_type == BinaryOpType::Sub){
            oss << ")";   //右括号
        }
        op_flag = -1;  //恢复op_flag为初始值-1
    }
```