编译技术 - project 02 - 实习报告

- **组号**

  > **37**

- **成员**

  > **1700017796 王垠浩**
  >
  > **1700012122 金浩杰**

1. 说明

- 基本实现思路就是将输入的 `kernel` 转化为 `project1` 中的形式，

  然后调用 `project1` 生成 `C++` 代码

- 由于在这一部分中输入的 `kernel` 与 `project1` 有些区别，

  因此需要修改 `BNF` 以及在代码生成的时候的上下界

- 由于自动求导在整体的实现时存在一些较为复杂的问题，我们在很多地方进行了简单化处理

  - 具体实现或者未实现的功能将会在第 `2` 部分给出

2. 分工

- 由于这一部分主要难点在于自动求导技术的设计，分工主要是大家负责几个函数的编写
- 另一方面，金浩杰同学负责语法树部分的修改，王垠浩同学负责 `IR` 的修改

3. 自动求导技术设计

- 首先我们将给出我们自动求导设计的思路，然后会用几个例子说明

3.1 自动求导技术设计思路与正确性

- 首先有以下约定的说明
  - 读取 `kernel` 的过程中去掉了所有的空格
  - 为了处理方便我们将整除符号处理成 `$`，主要是为了保持单字符的一致性和处理方便
  - 下一步的输入都是上一步生成的 `1` 个语句

(1) 处理 kernel 中的多个语句

- 这一部分的实现比较简单，只需要将 `kernel` 按照 `;` 断成多个语句就行了
- 代码体现为一个 `while` 循环

(2) 去除语句中的常量

- 我们知道在求导的过程中，单独的常量是不起作用的，我们去除之
- 例如 `A=B*C+10086`，求导的结果为 `dB=dA*C` ，这和 `A=B*C` 的求导结果一样
- 在实现上这样的常量具有以下特点
  - 不在任何括号内
  - 左右均为加法或者减法 `+/-`
- 一个特殊情况，若常数在最左边，而且紧跟的为减号 `-` 时，补 `0`
  - `A=1.0-B`，我们处理成 `A=0-B`，这主要是为了符合 `project1` 的 `BNF`
- 通过一个 `bracket` 变量记录
- 代码实现上体现为如下函数

```
void clear_constant(std::string &old);
```

(3) 处理语句中的加法

- 包括减法
- 这里的加法指的是 **最外层** 的 项 与 项 之间的加法，即不被任何括号包被的加法

  不包括 `A=B*(A+C)` 这些加法
- 例如：输入的语句中可能含有 `A=B*C+B*D` 的式子，按照求导法则结果应该是 `dB=dA*C+dA*D`

  为了我们处理上的方便，我们对其进行拆分，拆分成 `A=B*C;A=A+B*D;`
- 代码实现上体现为如下函数

```
void clip_sum_stmt(
    std::string &deal_str,
```

```
    std::vector<std::string> &statement
);
```

(4) 进行乘法分配律

- 对于 `A=(B+C)/3.0` 这样的式子，求导结果为 `dB=dA/3.0`，为了方便处理，我们使用分配律进行展开，再进行加法拆分，得到 `A=B/3.0;A=A+C/3.0;`
- 在这里我们只是简单的检查语句是否有 `()`
- 在具体实现上我们通过识别 `(,)` 的位置，将括号中的语句按照因子的形式拆分

  拆分方式和 **(3)** 中一致，剩余部分直接接上就行

  然后对于非第一个语句，我们在最左侧补上 `out+` 的操作，实现结果的累加，保证正确性

  - 对于 `out+` 的操作可能会在 **(3)** 的基础上重复加
  - 因此需要进行判断是否在 **(3)** 上进行了 `out+` 的操作
  - 这个判断比较简单，只需要检测语句等式右边的最左边是否已经是 `out+` 即可
    - 因为经过 **(3)** 的处理，不存在其他的最外层加法，而且根据我们的处理方式

      `out+` 出现在最左边
- 以上方法的正确性是由 **(3)** 保证的，**(3)** 消除了最外层的加减法，因此直接拼接不会出错
- 举一个没有处理过的式子
  - `A=B*C+B*(C+D)` 按照分配律处理的结果为 `A=B*C+B*C;A=A+B*C+B*D`

    这显然是错误的
  - 但是先经过 **(3)** 的处理，最终的结果如下，这是等价的

```
(3) 输出:
    A=B*C;
    A=A+B*(C+D)
(4) 输出:
    A=B*C;
    A=A+B*C;
    // A=A+A+B*D; 不进行重复判断(ERROR)
    A=A+B*D;
```

- 代码实现上体现为如下函数

```
void distribute(std::string &str);
```

(5) 对于无效语句的过滤

- 我们观察到所有的变量都有初始化，因此我们对于 `A=0;` 这样的语句不进行处理
  - 在这个地方，单独的赋值语句右边只能是 `0`
- 同时我们对于 `A=B`，但是对 `C` 求偏导的语句直接过滤，这里的正确性是显然的
  - 例如 `A=B+C; => A=B;A=A+C => A=A+C; => dC=dC+dA;`
  - 另外一方面，在清理常数步骤， `A=B+1.0*1.0;` 中 `1.0*1.0` 是无法处理的

    但可以在这一步中实现过滤

(6) 对于每一个语句进行求偏导

- 因为我们知道这里的语句只能是简单的一项 `C=A*B;`，或者是带有左边一项的加法 `C=C+A*B;`
- 对于上述情况，我们可以进行以下的推导
  - `C=A*B; => dA=dC*B;`
  - `C=C+A*B; => dA=dA+dC*B;`
    - 这里的 `dA+` 如果是第一个语句，由于初始化`dA`为全零，不会产生影响。
    - 如果不是第一个语句，说明此语句来自于语句的拆分，此时必须加上 `dA+`，以累加之前产生的结果，以保证正确
      - 一个例子

```
(0) 初始 kernel  (grad_to:A)
C = A*B + B*A;

(6) 的输入
C = A*B;
C = C + B*A;

(6) 的输出
dA = dC*B;
dA = dA + B*dC; // 若没有 dA+, 则 ERROR
```

- 因此总结出来的求导规则就是将 `out` 换成 `d(grad_to)`，`grad_to` 换成 `d(out)`

  此时对于变量下标直接进行移动，不改写
- 对于一些含有复杂的乘法情况 `C=A*A*A;`

  我们通过在等号右边从左到右扫描，每碰到一个 `grad_to` 的目标 `id` 就进行一次求导操作

  每一次求导的下标按照检测到的 `id` 进行操作

产生的第一个语句不用进行 `d(grad_to)+` 操作，剩余的语句需要（即进行加和）

- 一个例子

```
(0) 初始 kernel  (grad_to:A)
C = A*A*A;

(6) 的输入
C = A*A*A;

(6) 的输出
dA = dC*A*A;
dA = dA + A*dC*A;
dA = dA + A*A*dC;
```

- 代码实现上体现为如下函数

```
std::string generate(
    const std::string &deal_str,
    const std::string grad_to,
    const std::string out
);
```

(7) 等式左边复杂下标的处理

- 以上未对变量的下标进行处理，可能会得到 `dA[i+j,k] = dC[i,k] + dC[j][k];` 这样的结果
- 为消除等号左侧的复杂下标，我们为每个复杂下标结构分配一个新的下标
- 在这里我们做了一些简化假设，复杂的下标只能是连着的加法 `i+j+k` 或者是单步整除/取模 `i//j`

  而且每一个复杂下标的元素都会在表达式中存在单独的出现（这个假设是合理的）
- 对于连着的加法我们处理如下
  - 以 `dA[i+j+l,k] = dC[i,k] + dC[j,k] + dC[l,k];` 为例
  - 首先找到复杂的下标 `i+j+k` ， 我们为其分配一个新的下标 `__a`
  - 找到 `i+j+k` 的基本下标集合 `{i,j,k}`
  - 搜索等号右侧，进行下标替换，由于引入了一个多余变量，我们需要减少一个自由度恢复

    我们随便选择右侧的一个下标 `i` 然后将下标中的 `i` 替换成 `__a-j-k`
    - 对于每个基本下标集合，应当只有一个下标被替换
- 对于整除/取模的处理如下
  - 首先检测 `$/%` 符号，检测到了则进行处理

    - 以 `B<32>[i] = A<2, 16>[i//16, i%16];` 为例
    - 处理后的输入为 `dA<2,16>[i$16,i%16]=dB<32>[i];`
    - 我们先检测 `$`，检测到 `i$16`，此时将 `i$16` 用一个新下标 `__a` 代替，

      并将下标 `i` 的地方需要更新的目标记作 `__a*16` （不更新）
    - 接着检测 `%`，检测到 `i%16`，此时将 `i$16` 用一个新下标 `__b` 代替，

      并将下标 `i` 的地方需要更新的目标记作 `__a*16+__b`（不更新）
    - 更新 `i`
- 在整除部分我们做了简单的假设 `a//b,c%d => a=c,b=d`，否则报错退出
- 对于具体应用中有实际意义的运算，这一假设是合理的
- 代码实现上体现为如下函数

```
void complex2easy(std::string &str);
```

3.2 一个样例

- 我们选择最强的一个样例 `case10` 进行过程演绎

```
(0) 初始 kernel (grad_to : B)
    A<8,8>[i,j]=(B<10,8>[i,j]+B<10,8>[i+1,j]+B<10,8>[i+2,j])/3.0;

(1) 处理 kernel 中的多个语句
    A<8,8>[i,j]=(B<10,8>[i,j]+B<10,8>[i+1,j]+B<10,8>[i+2,j])/3.0;

(2) 去除语句中的常量
    A<8,8>[i,j]=(B<10,8>[i,j]+B<10,8>[i+1,j]+B<10,8>[i+2,j])/3.0;

(3) 处理语句中的加法
    A<8,8>[i,j]=(B<10,8>[i,j]+B<10,8>[i+1,j]+B<10,8>[i+2,j])/3.0;

(4) 进行乘法分配律
    A<8,8>[i,j]=B<10,8>[i,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+1,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+2,j]/3.0;

(5) 对于无效语句的过滤
    A<8,8>[i,j]=B<10,8>[i,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+1,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+2,j]/3.0;

(6) 对于每一个语句进行求偏导
    dB<10,8>[i,j]=dB<10,8>[i,j]+dA<8,8>[i,j]/3.0;
    dB<10,8>[i+1,j]=dB<10,8>[i+1,j]+dA<8,8>[i,j]/3.0;
    dB<10,8>[i+2,j]=dB<10,8>[i+2,j]+dA<8,8>[i,j]/3.0;

(7) 等式左边复杂下标的处理
    dB<10,8>[i,j]=dB<10,8>[i,j]+dA<8,8>[i,j]/3.0;
    dB<10,8>[__a,j]=dB<10,8>[__a,j]+dA<8,8>[__a-1,j]/3.0;
    dB<10,8>[__a,j]=dB<10,8>[__a,j]+dA<8,8>[__a-2,j]/3.0;
```

3.3 一个自己造的更强的样例

- 我们构造一个相对较强，但是可能实际意义不大的样例

```
(0) 初始 kernel (grad_to : B)
    A<8,8>[i,j]=2.0*(B<10,8>[i,j]+B<10,8>[i+1,j])/3.0+1.0*1.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+k,j]*B<10,8>[k,j]+1.0;

(1) 处理 kernel 中的多个语句
    A<8,8>[i,j]=2.0*(B<10,8>[i,j]+B<10,8>[i+1,j])/3.0+1.0*1.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+k,j]*B<10,8>[k,j]+1.0;

(2) 去除语句中的常量
    A<8,8>[i,j]=2.0*(B<10,8>[i,j]+B<10,8>[i+1,j])/3.0+1.0*1.0;// 注意 1.0*1.0 还在
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+k,j]*B<10,8>[k,j];

(3) 处理语句中的加法
    A<8,8>[i,j]=2.0*(B<10,8>[i,j]+B<10,8>[i+1,j])/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+1.0*1.0;
    A<8,8>[i,j]=A<8,8>[i,j];
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+k,j]*B<10,8>[k,j];

(4) 进行乘法分配律
    A<8,8>[i,j]=2.0*B<10,8>[i,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+2.0*B<10,8>[i+1,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+1.0*1.0;
    A<8,8>[i,j]=A<8,8>[i,j];
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+k,j]*B<10,8>[k,j];

(5) 对于无效语句的过滤
    A<8,8>[i,j]=2.0*B<10,8>[i,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+2.0*B<10,8>[i+1,j]/3.0;
    A<8,8>[i,j]=A<8,8>[i,j]+B<10,8>[i+k,j]*B<10,8>[k,j];

(6) 对于每一个语句进行求偏导
    dB<10,8>[i,j]=2.0*dB<10,8>[i,j]+dA<8,8>[i,j]/3.0;
    dB<10,8>[i+1,j]=2.0*dB<10,8>[i+1,j]+dA<8,8>[i,j]/3.0;
    dB<10,8>[i+k,j]=dB<10,8>[i+k,j]+dA<8,8>[i,j]*B<10,8>[k,j];
    dB<10,8>[k,j]=dB<10,8>[k,j]+B<10,8>[i+k,j]*dA<8,8>[i,j]+dB<10,8>[k,j];

(7) 等式左边复杂下标的处理
    dB<10,8>[i,j]=2.0*dB<10,8>[i,j]+dA<8,8>[i,j]/3.0;
    dB<10,8>[__a,j]=2.0*dB<10,8>[__a,j]+dA<8,8>[__a-1,j]/3.0;
    dB<10,8>[__a,j]=dB<10,8>[__a,j]+dA<8,8>[i,j]*B<10,8>[__a-i,j];
    dB<10,8>[k,j]=dB<10,8>[k,j]+B<10,8>[i+k,j]*dA<8,8>[i,j]+dB<10,8>[k,j];
```

3.4 一些存在的问题

- 由于进行了简化假设，假设之外的情况会进行报错退出
  - 例如对于分配律，在 `(A+B)*(A+B)` 我们就不能很好的处理，因为代码中只考虑了一个括号的情况

    即只考虑了分配一次，对于上述情况可能出错
  - 对于没有初始化的操作可能会出错（没有处理初始化）
  - 对于整除和取模的操作有很大限制
- 对于多条语句的情况可能出错
  - `A=B*C;A=A+B*D;`
  - 正确的结果应该是 `dB=dA*C+dA*D;`
  - 但是代码输出的结果为 `dB=dA*C;dB=dA*D; <=> dB=dA*D;`
  - 这是有问题的，可能的解决方法是在过滤的时候保留 `A=A+A;` 这样的语句，

    同时需要在处理加法的时候添加一个判断
    - 对于非第一个语句，进行 `out+` 操作的时候需要先进行检测，

      检测在语句中是否存在 `out` 作为最外层项的情况

      若有，则不进行 `out+` 操作
    - 但是以上改进也不能涵盖多语句的情况，因此没有进行操作
- 在处理完复杂下标之后，应该进行一个合并的操作，`a-b+b` 应该处理成 `a`，

  否则后面处理可能会出错，但是这一部分尚未实现
  - 建立在下标中只有 `+` 的前提上，那么转化成的语句下标中也只会有 `+/-`

    此时可以对下表中的 `+` 和 `-` 建立一个集合，两个集合做差即可的出新的表达式
  - `a-b+b` 为例
    - `'+' : a,b`
    - `'-' : b`
    - `'+' - '-' = a`
- 总而言之，在单语句上的表现还算可以

3.5 实验结果

- `10` 个 `case` 都已经通过

4. 语法分析树的修改

4.1 BNF修改

- 因为 `project1` 中的下标只有加法，但是 `project2` 进行转换之后我们出现了减法，需要修改

```
P ::= (S)*
S ::= LHS = RHS
LHS ::= TRef
TRef ::= Id < CList > [ AList ]
SRef ::= Id < CList >
CList ::=  (IntV)*
AList ::= (IdExpr)*
Const ::= FloatV | IntV
IdExpr ::= IdExpr_B (IdExpr_A)*
IdExpr_A ::= OP IntV | OP IdExpr
IdExpr_B ::= Id | (IdExpr)
OP ::= +  | * | - | / | %
RHS ::= RHS_A RHS_B
RHS_B ::= (OP RHS_A)*
RHS_A ::= (RHS) | TRef | SRef | Const
```

- 我们在这里发现 `IdExpr_A` 的地方存在左公因式，但是没有关系，我们使用向前看的技术，

  通过检测 `Follow(op)` 可以实现具体选择哪一个语句

4.2 获取范围的修改

- 由于出现了减法，原来的获取范围的方式会出问题
- 现在的处理方式为，在进入 `AList` 结点的时候，若发现当前的 `IdExpr` 语句中存在非 `+` 的 `op`，

  那么就不进行上界的判断，这么处理的正确性由每一个下标都会单独出现的假设保证，

  由于会单独出现，那么就能获得一个上界
- 以上的假设在输入上是能保证的，在 **合并** 之后也是能保证的，而且也是合理的

5. IR输出的修改

- 由于出现了减号，我们在最终的每一个符合判断的时候都增加一个下界判断 `>=0`

6. 编译理论知识的应用

- 建立 BNF 的过程中使用到了消除左递归和提取左公因式的操作
- 语法分析中使用到的语法分析树
    - 这一部分具体实现和 project1 大致相同
    - 使用两类函数 parse_node,visit_node 建立语法树和进行分析
- 在对具体的 kernel 进行分析的时候用到了语法分析中的向前看操作
    - 具体的实现以 parse_MyIdExpr_A 为例子
    - 通过 is_digit(next_char) 判断下一个字符是否为数字实现

```
void parse_MyIdExpr_A(MyIdExpr_A* arg){
    char next_char = Global::parse_string[Global::now_index + 1];
    if(is_digit(next_char)){
        arg->type = 0;
        arg->op = new MyOP();
        parse_MyOP(arg->op);
        arg->intv = new MyIntV();
        parse_MyIntV(arg->intv);
    }
    else{
        arg->type = 1;
        // match('+');
        arg->op = new MyOP();
        parse_MyOP(arg->op);
        arg->exp = new MyIdExpr();
        parse_MyIdExpr(arg->exp);
    }
}
```

- 使用提供的IR进行最终代码的生成

7. 一些其他的问题

- 由于整个程序使用`float`进行运算，运算的精度受运算顺序影响较大
- 例如对于给出的`case6`，我们的代码生成结果为：

  ```
  dB[n][c][__a][__b]=dA[n][k][__a-r][__b-s]*C[k][c][r][s]
  ```

  答案中给出的结果为

  ```
  dB[n][c][h][w]=dA[n][k][p][q]*C[k][c][h-p][w-q]
  ```

  由于在等式右侧替换的下标存在差异，实际运行时产生了精度问题。

  在一次实际运行时，会产生最大量级在`1e-4`的误差，而给出的`test`程序中限制了`eps=1e-5`，导致出现错误。

- 作为测试，我们尝试了将测试的输入改为整型输入，则我们的生成代码将与答案的代码产生相同的结果
- 为解决这一问题，我们单纯地将等式右侧的下标替换由从左向右查找改为了从右向左查找。