Compiler Project(2020 Spring): AutoGrad

Compiler Project(2020 Spring): AutoGrad
    Overview
        Usage
            Dependencies
                Ubuntu 18.04
            Compilation & Test
        Docker
    Let us Start From Examples!
        Index Category
        Derivation w/o Index
        Derivation w/ Index
        Brief Summay
    Formalized Automatic Derivation Process
        Chain Rule on AST
        Gaussian Elimination For Linear index Transform
    How to Implement Automatic Derivation?
        Chain Rule
        Gaussian Elimination Method
        Printer of CPP
        Function Signature
    Summary of Compilation Technology
    Division of Labor within the Group

Overview

本次project在第一个project的基础上，对通过Token Analysis和简单的Syntax Analysis得到的AST进行自动求导.

从理论上来讲，本project实现的自动求导器可对任意线性下标的张量进行自动求导，其使用的线性下标变换下求导方法主要基于此篇论文，主要的区别是在进行线性坐标变换的时我们使用了 Gaussian Elimination 而不是 Smith Normalization.

Usage

Dependencies

- flex: a fast lexical analyzer - scanner generator for lexing in C and C++
- bison: a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables.
- nlohmann/json: JSON Parser for Modern C++
    - In this project, we use the single required header file `json.hpp`

Ubuntu 18.04

```
apt-get -y install bison flex
```

Compilation & Test

```
mkdir build && cd build
cmake ..
make -j4
./project1/test2
```

一个CI的测试结果见此.

Docker

此外，为防止编译环境不同造成的错误，我们还提供了一个统一的docker环境，详细见文件docker/Dockerfile.remote-cpp-env的头部注释.

```
# CLion remote docker environment (How to build docker container, run and stop it)
#
# Build and run:
#   docker build -t clion/remote-cpp-env-boost:0.5 -f Dockerfile.remote-cpp-env .
#   docker run -d --cap-add sys_ptrace -p127.0.0.1:2233:22 --name clion_remote_env-boost clion/remote-cpp-env-boost:0.5
#   ssh-keygen -f "$HOME/.ssh/known_hosts" -R "[localhost]:2233"
#
# ssh credentials (test user):
#   ssh user@localhost -p 2233
#
# restart:
#   docker restart clion_remote_env-boost
# stop:
#   docker stop clion_remote_env-boost
# remove:
#   docker rm clion_remote_env-boost
#
```

Let us Start From Examples!

Index Category

首先，我们对index大致分为两类: spacial index 和 reduced index.

- spacial index: 在move statement的左边出现过的index
- reduced index: 在move statement的右边出现过，但是在左边未出现的index

如对于下面这个例子，i和$j$spacial index，而k和l是reduced index.

$$A<16,32>[i,j]=B<16,32,4>[i,k,l]*C<32,32>[k,j]*D<4,32>[l,j] \tag{1}$$

利用project得到的结果，我们还可以得到各个index的domain.

```
A<16, 32>[i, j] =  B<16, 32, 4>[i, k, l] * C<32, 32>[k, j] * D<4, 32>[l, j];
i
 index type: Spatial
 domain: [0, 16)
j
 index type: Spatial
 domain: [0, 32)
k
 index type: Reduce
 domain: [0, 32)
l
 index type: Reduce
 domain: [0, 4)
```

Derivation w/o Index

我们先来看一个不带下标，仅仅进行符号求导的例子，以$C = \frac{x}{x+y}$为例.
求导的方法是，针对每个等式右边出现的被求导变量，把其他的被求导变量视为常量，然后进行求导.
比如在$C = \frac{x}{x+y}$中对x进行求导，我们将得到两个部分：

- 只针对分子中出现的x， 把分母中出现的x进行视为常量,得到 $\frac{dC}{x+y}$
- 只针对分母中出现的x， 把分子中出现的x进行视为常量,得到 $\frac{xdC}{-(x+y)^2}$

把这两部分相加，我们得到

$$dx = \frac{dC}{x+y} + \frac{xdC}{-(x+y)^2} \tag{2}$$

经验证，这样的求导方式是正确的，同时我们可以发现，这些部分的个数等于在等式右边出现的被求导变量x的个数.

Derivation w/ Index

以

$$B < 8,8 > [i,j] = A < 8,10 > [i,j] * A < 8,10 > [i,j+2] \tag{3}$$

为例，我们来说明带有下标时的求导方式.

首先根据project1的结果，我们知道i和j的取值范围都是$[0,8)$.

$$\begin{cases} 0 \le i < 8 \\ 0 \le j < 8 \end{cases}$$

分析出来的求导表达式的左侧的下标索引上不能有加减乘除等运算，也就是不能出现`A[i+1] = B[i]`的形式，不妨新设两个下标变量z_0和z_1，分析出来的求导表达式的左侧为$dA < 8,10 > [z_0, z_1]$，并且z_0和z_1的遍历范围分别为对应的dA的shape：

$$\begin{cases} 0 \le z_0 < 8 \\ 0 \le z_1 < 10 \end{cases}$$

首先先对在等式右边出现的第一个被求导变量$A < 8,10 > [i,j]$进行求导，得到

$$dB < 8,8 > [i,j] * A < 8,10 > [i,j+2] \tag{4}$$

接下来需要进行下标变换，即对被求导变量$A < 8,10 > [i,j]$与求导表达式的左侧为$dA < 8,10 > [z_0, z_1]$进行变换，我们有方程

$$\begin{cases} i = z_0 \\ j = z_1 \end{cases}$$

同时，我们仍要保证i和j替换后的约束，即

$$\begin{cases} 0 \le z_0 < 8 \\ 0 \le z_1 < 8 \end{cases}$$

利用此结果替换原求导表达式的i和j，得到

$$Select(0 \le z_0 < 8 \&\& 0 \le z_1 < 8, dB < 8, 8 > [z_0, z_1] * A < 8, 10 > [z_0, z_1 + 2], 0) \tag{5}$$

同样，对于第二个被求导变量$A < 8, 10 > [i, j + 2]$进行求导，得到

$$dB < 8, 8 > [i, j] * A < 8, 10 > [i, j] \tag{6}$$

在进行下标变换的时候，即对被求导变量$A < 8, 10 > [i, j + 2]$与求导表达式的左侧为$dA < 8, 10 > [z_0, z_1]$，方程有些不同：

$$\begin{cases} i = z_0 \\ j + 2 = z_1 \end{cases}$$

解得

$$\begin{cases} i = z_0 \\ j = z_1 - 2 \end{cases}$$

同时，我们仍要保证i和j替换后的约束，即

$$\begin{cases} 0 \le z_0 < 8 \\ 0 \le z_1 - 2 < 8 \end{cases}$$

利用此结果替换原求导表达式的i和j，得到

$$Select(0 \le z_0 < 8 \&\& 0 \le z_1 - 2 < 8, dB < 8, 8 > [z_0, z_1 - 2] * A < 8, 10 > [z_0, z_1 - 2], 0) \tag{7}$$

最终得到的式子为

$$dA < 8, 10 > [z_0, z_1] = Select(0 \le z_0 < 8 \&\& 0 \le z_1 < 8, dB < 8, 8 > [z_0, z_1] * A < 8, 10 > [z_0, z_1 + 2], 0) \tag{8}$$
$$dA < 8, 10 > [z_0, z_1] = Select(0 \le z_0 < 8 \&\& 0 \le z_1 - 2 < 8, dB < 8, 8 > [z_0, z_1 - 2] * A < 8, 10 > [z_0, z_1 - 2], 0) \tag{9}$$

在上面的下标变换中，其实在在解线性方程组，我们使用了高斯消元法实现．上述的$z0$和$z1$在求导表达式中为spacial index；当然，在高斯消元法的过程中，会出现*自由变量*和*独立变量*的概念，显然这些下标变量无需替换，只需要变为reduced变量即可．关于*自由变量*和*独立变量*属于代数内容，这里不再详细展开．

Brief Summay

不妨设被求导变量为B，其有t个维度．经过上面分析，我们知道，最终的结果一定可以表示成若干个如下的loopnest statement:

$$dB[z_0, z_1, \ldots, z_{t-1}] = Select([cond], [derivation], 0) \tag{10}$$

其中这样的loopnest的个数等于在等式右边出现的被求导变量的个数．$[cond]$是坐标变换后可能需要的对新产生的index的约束，$[derivation]$是针对每个等式右边出现的被求导变量，把其他的被求导变量视为常量，然后进行求导得到的表达式．

> 具体到当前的IR表示上，我们可以找出例子，说明结果不一定能直接用一个LoopNest单独表示最终的结果，即而不能写成外层若干个循环，然后最终内部若干个不包含并列循环的statement的形式．

如对于例子

$$B<10,10>[i,j]=A<10,10>[i,k]*A<10,10>[k,j] \tag{11}$$

对A求导后我们可以得到

$$\begin{aligned} dA<10,10>[z_0,z_1] =& dB<10,10>[z_0,j]*A<10,10>[z_1,j] &(12)\\ &+A<10,10>[i,z_0]*dB<10,10>[i,z_1] &(13)\end{aligned}$$

根据爱因斯坦求和规范，上面的式子应该写成

$$\begin{aligned} dA<10,10>[z_0,z_1] =& \sum_j dB<10,10>[z_0,j]*A<10,10>[z_1,j] &(14)\\ &+\sum_i A<10,10>[i,z_0]*dB<10,10>[i,z_1] &(15)\end{aligned}$$

对应到C代码上，我们得到的应该是外层分别是spacial index z_0和z_1的两层循环，然后内部是reduced index i和j的两个并行的循环；而不能写成外层若干个循环，然后最终内部若干个不包含并列循环的statement的形式.

Formalized Automatic Derivation Process

Chain Rule on AST

以`A<8, 8>[i, j] = (B<10, 8>[i, j] + B<10, 8>[i + 1, j] + B<10, 8>[i + 2, j]) / 3.0`对B求导为例。语法树的根节点为$/$，设其左右子树分别为f和g。访问根节点时带参数dA，当访问左子树时，根据链法则，

$df=\frac{\partial l}{\partial A}\cdot\frac{\partial A}{\partial f}=dA\cdot\frac{\partial \frac{f}{g}}{\partial f}=\frac{dA}{g}=\frac{dA}{3}$
因此访问左子树的根节点$+$时带参数$\frac{dA}{3}$。类似地，当访问$+$的右子树（设为f_2，左子树为f_1）时，
$df_2=\frac{\partial l}{\partial f}\cdot\frac{\partial f}{\partial f_2}=df\cdot\frac{\partial (f_1+f_2)}{\partial f_2}=df$
访问参数不变，仍为$df=\frac{dA}{3}$。

Gaussian Elimination For Linear index Transform

承上一小节，当访问到f_2时，
$dB=\frac{\partial l}{\partial f_2}\cdot\frac{\partial f_2}{\partial B}=df_2\cdot\frac{\partial B<i+2,j>}{\partial B}=\frac{dA}{3}<z_0,z_1>$
根据下标关系，可列出方程

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}z_0\\ z_1\end{bmatrix}=\begin{bmatrix}i+2\\ j\end{bmatrix} \tag{16}$$

可用高斯消元法解线性方程组，解为$z_0=i-2,z_1=j$。因此
$dB=\frac{dA<i-2,j>}{3}$

How to Implement Automatic Derivation?

这里主要介绍了目录`project2\solution`下各个类的设计和功能作用.

其中`IndexAnalyst.cpp/h`和`IRPolisher.cpp/h`分别实现了下标范围推断和对IR进行polish(主要是把推断得到的下标范围信息加入到AST中)，其属于project1的内容，这里不再详细介绍.

Chain Rule

chain rule的实现主要在`AutoDiffer.cpp/h`. 一般来说，在访问AST各结点的时候，我们需要把当前整棵子树的微分传入通过传参的方式传入，这里我们通过手动模拟栈的方式实现：

```
class AutoDiffer : public  IRVisitor{
public:
  /**
   *
   * @param expr expression that has been be differentiated
   * @param grad_to string name of target differential variable
   * @param differential differential expression w.r.t. current expression |expr|
   * @return differential statement w.r.t. variable |grad_to_str|
   */
  Group operator ()(const Stmt &stmt, const string &grad_to_str);

protected:
  void visit(Ref<const Binary>) override;
  void visit(Ref<const Var>) override;
private:
  string grad_to_str_;
  map<string,Expr> str2old_indexes_;
  map<string,Expr> str2vars_;
  map<string,int> str2matrix_column_;
  vector<Expr> matrix_column2old_indexes_;

  vector<Expr> new_grad_to_indexes_;
  Expr new_grad_to_var_;
  stack<Expr> differentials_stack_;            // artificial stack
  vector<Stmt> results;
};
```

在访问`Binary`结点时,直接根据上一小结的方式使用链式法则求各子树的微分，然后递归访问：

```
void AutoDiffer::visit(Ref<const Binary> op) {
  if (op->op_type == BinaryOpType::Add) {
    (op->a).visit_expr(this);
    (op->b).visit_expr(this);
  } else if (op->op_type == BinaryOpType::Sub) {
    (op->a).visit_expr(this);
    auto current_diff = differentials_stack_.top();
    auto new_differential = SimplifiedNegation(current_diff);
    differentials_stack_.emplace(new_differential);
    (op->b).visit_expr(this);
    differentials_stack_.pop();
  } else if (op->op_type == BinaryOpType::Mul) {
    auto current_diff = differentials_stack_.top();
    auto new_differential = SimplifiedMultiplication(op->b, current_diff);
    differentials_stack_.emplace(new_differential);
    (op->a).visit_expr(this);
    differentials_stack_.pop();

    new_differential = SimplifiedMultiplication(op->a, current_diff);
    differentials_stack_.emplace(new_differential);
    (op->b).visit_expr(this);
    differentials_stack_.pop();

  } else if (op->op_type == BinaryOpType::Div) {
    /**
     * TODO: We only support the situation where the denominator is an immediate
     * currently. However, this code could be modified to support the case where
     * the denominator consists of variable with ease. :)
```

```
     */
    LOG(ERROR) << COND(op->b.as<FloatImm>() == nullptr &&
                       op->b.as<IntImm>() == nullptr &&
                       op->b.as<UIntImm>() == nullptr)
               << "We only support the situation "
                  "where the denominator is an immediate currently."
               << std::endl;

    auto current_diff = differentials_stack_.top();
    // FIXME: use expression |current_diff|'s type by default
    auto new_differential = SimplifiedDivision(current_diff, op->b);
    differentials_stack_.emplace(new_differential);
    (op->a).visit_expr(this);
    differentials_stack_.pop();
  } else {
    LOG(ERROR) << "Unsupported Binary operation." << std::endl;
  }
}
```

递归访问的终止结点是立即数或变量`Var`.

- 如果在立即数类型处终止，那么直接退出，因为不会对最终答案产生贡献.
- 如果访问的变量`Var`结点的变量名与被求导变量不同，那么也是直接退出，因为不会对最终答案产生贡献

```
void AutoDiffer::visit(Ref<const Var> op) {
  if (op->name != grad_to_str_) {
    return;
  }
  ...
```

- 如果访问的变量`Var`结点的变量名与被求导变量相同，类似于上一个section，提取系数，使用高斯消元法(在下面介绍)然后在传入的求导表达式中替换变量，最后加入到答案中即可.核心代码如下:

```
for (size_t i = 0; i < op->args.size(); i++) {
      auto extracted = coeffi_extractor(op->args[i], str2matrix_column_);
      rhs.push_back(SimplifiedSubtraction(new_grad_to_indexes_[i],
                                          Expr(int32_t(extracted.imm))));
      coefficients.push_back(extracted.coefficients);
    }
auto matrix = Matrix::make(coefficients, rhs);
ImplGaussianEliminationMethod impl_gaussian_elimination_method;
rv = impl_gaussian_elimination_method(matrix, matrix_column2old_indexes_);
IndexReplacer index_replacer;
auto current_diff = index_replacer(differentials_stack_.top(), rv.solutions);
```

Gaussian Elimination Method

高斯消元法的代码主要在`arith.cpp/h`中实现.

在下标变换这个问题里，系数矩阵的列数为原表达式中不同index的数量(记为`n_cols`)，行数为被求导变量的维数(记为`n_rows`). 在`MatrixRow`这个类里，我们保存了一行的`n_cols`个系数，以及以`Expr`形式保存的方程右侧. 为了方便，我们还重载了`+`, `*` 和 `-` 等运算符.

```

```

```
typedef Ref<MatrixRow> RefMatrixRow;

class MatrixRow{
  DISALLOW_COPY_AND_ASSIGN(MatrixRow);
  public:
    explicit MatrixRow(vector<int> coefficients, const Expr &rhs)
      : n_cols_(coefficients.size()),
        coefficients_(std::move(coefficients)),
        rhs_(rhs){}
    static RefMatrixRow make(const vector<int> &coefficients, const Expr &rhs);

    int get(size_t i) const ;
    vector<int> get_coefficients(){return coefficients_;}
    Expr get_rhs() {return rhs_;}
    friend RefMatrixRow operator+(RefMatrixRow a, RefMatrixRow b);
    friend RefMatrixRow operator-(RefMatrixRow a, RefMatrixRow b);
    friend RefMatrixRow operator-(RefMatrixRow a);
    friend RefMatrixRow operator*(RefMatrixRow a, int32_t scale);

    friend Matrix;
    friend ImplGaussianEliminationMethod;

  private:
    size_t n_cols_;
    vector<int> coefficients_;
    Expr rhs_;
};
```

整个矩阵则用类`Matrix`表示，包括`n_rows`行：

```
typedef Ref<Matrix> RefMatrix;

class Matrix{
  DISALLOW_COPY_AND_ASSIGN(Matrix);
  public:
    explicit Matrix(vector<RefMatrixRow> rows)
      : n_rows_(rows.size()), n_cols_(rows[0]->n_cols_),
        rows_(std::move(rows)){}
    static RefMatrix make(const vector<vector<int>> &coefficients, vector<Expr>
all_rhs);
    friend ImplGaussianEliminationMethod;
    size_t get_n_rows() const {return n_rows_;}
    size_t get_n_cols() const {return n_cols_;}
    vector<RefMatrixRow> get_rows(){ return rows_; }
  private:
    size_t n_rows_, n_cols_;
    vector<RefMatrixRow> rows_;
};
```

高斯消元法的实现在类`ImplGaussianEliminationMethod`中．其运算过程基本是代数中的初等行变换操作，与代数上的实现基本相同．这里有一个小trick是，为了不引入分数，我们可以通过类似与**辗转相除法**类似的方式进行消元，其核心代码如下：

```
for(int i = 0; i < n_cols && current_row < n_rows; i++){
    int non_zero_row_index = current_row;
    for(int j = current_row; j < n_rows; j++) if(rows[j]->get(i) != 0)
```

```
      { non_zero_row_index = j; break; }
    if(non_zero_row_index != current_row){
      std::swap(rows[current_row], rows[non_zero_row_index]);
    }
    if(rows[current_row]->get(i) == 0){
      // independent_variable
      is_independent_variable[i] = true;
      continue;
    }
    last_main_col = i;
    main_row[i] = current_row;
    for(int j = current_row + 1; j < n_rows; j++){
      while(rows[j]->get(i) != 0){
        int32_t t = rows[j]->get(i) / rows[current_row]->get(i);
        rows[j] = rows[j] - rows[current_row] * t;
        if (rows[j]->get(i) != 0)
          std::swap(rows[current_row], rows[j]);
      }
    }
    ....
```

对于零行，我们需要添加约束

```
  // zero row
  for(int j = current_row; j < n_rows; j++){
    auto compare_type = Type::int_scalar(32);
    constraints.push_back(Compare::make(compare_type, CompareOpType::EQ,
      rows[j]->rhs_, Expr(int32_t(0))));
  }
```

同理，对于出现的独立变量和自由变量，我们不需要进行替换，把其改成reduced index即可

```
for (int i = n_cols - 1; i >= 0 ; i--) {
    if (is_independent_variable[i] || i > last_main_col) {
      // independent variable or free variable
      auto index = indexes[i].as<Index>();
      auto index_type = Type::int_scalar(32);
      solutions[index->name] = Index::make(index_type, index->name, index->dom,
IndexType::Reduce);
    } else {
    ...
```

最后，利用`IndexReplacer.cpp/h`进行下标变量的替换即可.

Printer of CPP

与proj1类似，通过继承`IRprinter`类来实现类`IRcppPrinter`，该类实现从自动求导分析后得到的AST生成Cpp代码的翻译过程。
通过观察自动求导后的AST语法可知，需要重载的函数为如下函数：

```
void visit(Ref<const Select>) override;
void visit(Ref<const Var>) override;
void visit(Ref<const Index>) override;
void visit(Ref<const LoopNest>) override;
void visit(Ref<const Move>) override;
```

比较关键的是`select`和`LoopNest`的翻译：

1. `Select`可以翻译为cpp语言的条件表达式。

```
void IRcppPrinter::visit(Ref<const Select> op) {
  (op->cond).visit_expr(this);
  oss << " ? ";
  (op->true_value).visit_expr(this);
  oss << " : ";
  (op->false_value).visit_expr(this);
}
```

2. `LoopNest`如下翻译，利用`op->index_list`依次找到嵌套的多层循环，对于每层循环首先找到循环参数名，然后从`dom`中找到表示循环上下限的两个元素，之后进行循环体的翻译，之后加上右大括号即可。

```
void IRcppPrinter::visit(Ref<const LoopNest> op) {
  for (const auto &index : op->index_list) {
    print_indent();
    auto name = index.as<Index>()->name;
    auto dom = index.as<Index>()->dom.as<Dom>();
    int begin = dom->begin.as<IntImm>()->value();
    int end = begin + dom->extent.as<IntImm>()->value();
    oss << "for (int " << name << " = " << begin << "; " << name << " < " << end
        << "; " << name << "++){\n";
    enter();
  }
  for (const auto &body : op->body_list) {
    body.visit_stmt(this);
  }
  for (auto index : op->index_list) {
    exit();
    print_indent();
    oss << "}\n";
  }
}
```

实例化一个该类变量，然后对自动求导后获得的AST进行依次遍历即可获得翻译的Cpp代码。

Function Signature

函数签名中传入的参数，都是求导表达式中用到的变量，因此实现中是通过对中间代码的解析来生成函数签名。

具体来说，实现了继承自类IRPrinter的类signPrinter2。

其中关键的成员属性为：

```
std::map<std::string, std::string> ranges;
```

ranges记录了一个string类型的变量名到它对应的string类型的数组大小范围的映射。比如有一个变量$A[4][5]$，那么在ranges里就会记录一个<"A","[4][5]">。若变量不是数组，则映射的字符串为空字符串。

signPrinter2中对ranges的维护是在访问到Var结点的时候进行的，具体代码如下。

```
void signPrinter2::visit(Ref<const Var> op) {
```

```
    std::string name = op->name;
    if (op->shape.size() == 1 && op->shape[0] == 1)
    {
        ranges[name] = "";
        return;
    }
    std::string size = "[";

    for (size_t j = 0; j < op->shape.size(); ++ j) {
        size = size + std::to_string(op->shape[j]);
        if (j < op->shape.size() - 1) {
            size = size +  "][";
        }
    }
    size += "]";
    ranges[name] = size;
}
```

此外，signPrinter2还实现了一个get函数。它的作用是从传入的根结点开始遍历语法树，提取出该段中间代码用到的变量名及它们的数组大小范围，维护在ranges中，最后返回ranges。

```
std::map<std::string, std::string> signPrinter2::get(const Group &group) {
    ranges.clear();
    group.visit_group(this);
    return ranges;
}
```

最后，在solution2.cc中实现了generate_sign函数，用于生成函数签名。它传入的参数有：

- text：json中的"kernel"
- grad：json中的"grad_to"
- ins：json中的"ins"
- outs：json中的"outs"
- type：json中的"data_type"

该函数的第一部分是对grad中的每个变量分别求出它们的求导表达式中间代码，然后通过上述的signPrinter2得到其中涉及的变量，并把这些变量与数组大小范围的映射合并到一个map：range中。

```
std::string generate_sign(const string &text, std::vector<std::string> grad,
std::vector<std::string> ins, std::vector<std::string> outs, std::string type) {
  AutoDiffer auto_differ;
  IRPrinter printer;
  IndexAnalyst index_analyzer;
  IRPolisher polisher;

  auto main_stmt = parser::ParseFromString(text, 0).as<Kernel>()->stmt_list[0];
  auto domains = index_analyzer(main_stmt);
  main_stmt = polisher(main_stmt, domains);
  signPrinter2 sprinter2;
  std::map<std::string, std::string> range;
  range.clear();
  std::map<std::string, std::string>::iterator it;
  for (size_t i = 0; i < grad.size(); ++i) {
    auto lhs = main_stmt.as<Move>()->dst.as<Var>();
    auto differential =
        Var::make(lhs->type(), "d" + lhs->name, lhs->args, lhs->shape);
```

```
    auto rv = auto_differ(main_stmt, grad[i]);
    std::map<std::string, std::string> crange = sprinter2.get(rv);
    for (it = crange.begin(); it != crange.end(); it++)
      range[it->first] = it->second;
  }
```

该函数的第二部分就是通过得到的range，来生成最后的函数签名。具体是依次：

- 枚举ins中的变量名a，检查range中是否有键值为a的元素
- 枚举outs中的变量名a，检查range中是否有键值为"d"+a的元素
- 枚举grad中的变量名a，检查range中是否有键值为"d"+a的元素

对检查到range中存在的元素，生成对应的<变量名>+<数组大小范围>的字符串。

```
  std::string ret = "(";
  bool first = 1;
  for (size_t i = 0; i < ins.size(); ++i) {
    std::string name = ins[i];
    if (range.find(name) == range.end())
      continue;
    std::string size = range[name];
    if (!first)
      ret += ", ";
    ret += type + " ";
    first = 0;
    if (size.length() == 0)
      ret += "&" + name;
    else
      ret += "(&" + name + ")" + size;
  }
  for (size_t i = 0; i < outs.size(); ++i) {
    std::string name = "d" + outs[i];
    if (range.find(name) == range.end())
      continue;
    std::string size = range[name];
    if (!first)
      ret += ", ";
    ret += type + " ";
    first = 0;
    if (size.length() == 0)
      ret += "&" + name;
    else
      ret += "(&" + name + ")" + size;
  }
  for (size_t i = 0; i < grad.size(); ++i) {
    std::string name = "d" + grad[i];
    if (range.find(name) == range.end())
      continue;
    std::string size = range[name];
    if (!first)
      ret += ", ";
    ret += type + " ";
    first = 0;
    if (size.length() == 0)
      ret += "&" + name;
    else
      ret += "(&" + name + ")" + size;
```

```
  }
  ret += ")";
  return ret;
}
```

Summary of Compilation Technology

在这两个project中，我们使用到的编译知识主要有

- 词法分析和语法分析：`lex`和`yacc`的使用，parser的构建
- 中间表示形式(IR Node)，语法树构建，语法树遍历：通过访问者模式进行设计
- SDT和SDD的相关知识：主要在代码生成中，如在从IR表示到C代码生成中用到的on-the-fly generation技术.

Division of Labor within the Group

Name	Student ID	Labor
麦景	1700012751	实现在AST上进行链式法则求导和高斯消元法进行坐标变换
苏灿	1700012779	报告撰写以及代码整理
凌子轩	1700012752	生成函数签名
张灏宇	1700011044	完成从IR到C的代码生成