编译原理研讨课实验 PR002 实验报告

一、任务说明

本次实验的任务为，为 elementwise 新建语法/语义规则，使其支持赋值，加法，乘法三种操作，并由此构建新的 AST 表示。

二、成员组成

李昊宸 2017K8009929044
李颖彦 2017K8009929025
陆润宇 2017K8009929027

三、实验设计

对赋值，加法，乘法三种操作进行逐一改动，使其支持含 elementWise 的 AST 的生成：

1） 提供对 elementWise 标注的函数内部对数组的“+”，“*”，“=”操作
2） 提供对“int”数据类型的支持（而非 integer）
3） 支持一维静态大小输入

1. 设计思路

重点在于在 CreateBuiltinBinOP 函数中对于以下三个函数的修改：
CheckAssignmentOperands，CheckMultiplyDivideOperands，CheckAdditionOperands

我们需要增加与 elementWise 相关的 if 分支来处理 elementWise 的相关情况，并且对一些特殊情况（例如是否存在合法的类型转换）进行处理。

2. 实验实现

首先找到入口函数
/home/clang7/llvm/tools/clang/lib/Sema/SemaExpr.cpp

```
ExprResult Sema::CreateBuiltinBinOp(SourceLocation OpLoc,
                                    BinaryOperatorKind Opc,
                                    Expr *LHSExpr, Expr *RHSExpr) {
```

这个函数负责创造 AST 的有关 bulitin 的算符的结点，而实验要求的=，*，+均属于 builtin 算符。

对 opc 进行分析：

case1：赋值

```
case BO_Assign:
  ResultTy = CheckAssignmentOperands(LHS.get(), RHS, OpLoc, QualType());
```

此时我们要改动 check 函数，使其支持类似 A=B 的数组赋值功能。

修改 CheckAssignmentOperands 函数中的 CheckSingleAssignmentConstraints 函数，在函数中加入下列代码：

```
///ADD THIS IN PROO2
if (this->ElementWiseContext)
{
  QualType RHSType = RHS.get()->getType();
  if (ConstantArrayType::classof(LHSType.getTypePtr()) &&
      ConstantArrayType::classof(RHSType.getTypePtr()))
  {
    const ConstantArrayType *RH_arraytype = dyn_cast<ConstantArrayType>(RHSType);
    QualType RH_elemtype = RH_arraytype->getElementType().getUnqualifiedType();
    LHSType = Context.getCanonicalType(LHSType).getUnqualifiedType();
    RHSType = Context.getCanonicalType(RHSType).getUnqualifiedType();
    if (LHSType == RHSType && RH_elemtype->is_only_IntType())
    {
      return Compatible;
    }
    else
    {
      return Incompatible;
    }
  }
}
```

代码当 ElementWiseContext 为 true 时开始执行，ElementWiseContext 即为在 PRO01 我们实现的对函数定义的 elementWise 标注，如果为真则说明该定义支持 element Wise 操作。

首先调用 ConstantArrayType::classof 函数，判定两个数组是否均为常数组类型。如果都是，首先调用 getElmentType 返回 QualType 类型的 ElementType，再调用此类型的 public 操作 getUnqualifiedType 得到每个元素的类型。

之后，调用 AST 语法树中的 public 操作 getCanonicalType：首先调用 QualType 类型的 LHSType（RHSType）下的 public 操作 getCanonicalType（这里有名字空间的调用）返回一个切割好的 QualType 类型的变量，随后调用 CanQualType 名字空间下的 CreateUnsafe 创建对象，并最终返回。

最后，判断 lefthandside 与 righthandside 的类型是否相同，并且 righthandside 的元素必须为 int 类型（通过调用自己实现的 is_only_IntType 函数），若满足就返回可编译。

总体上流程是这样，但实际上还需要做一些补充。

对于 A = （B + C）这样的表达式，是可以进行编译的，因为对于赋值号而言，右侧的（B + C）在 AST 中存在一个可访问的右值，所以可以拿来给 A 赋值；但是对于（B + C）= A 这样的表达式，却是不可以赋值的，因为左侧不存在一个可访问的左值。为解决这样的左右差异，我们需要查看 CheckAssignmentOperands 函数中是如何处理左侧的左值性，如下：

```
  // Verify that LHS is a modifiable lvalue, and emit error if not.
  if (CheckForModifiableLvalue(LHSExpr, Loc, *this))
    return QualType();
```

可以发现，基本的处理逻辑就是调用 CheckForModifiableLvalue 检查 left_hand_side 是否为可编辑的左值：

```
static bool CheckForModifiableLvalue(Expr *E, SourceLocation Loc, Sema
&S) {
……
  unsigned Diag = 0;
  bool NeedType = false;
  switch (IsLV) {
  ……
    case Expr::MLV_ArrayTemporary:
    Diag = diag::err_typecheck_array_not_modifiable_lvalue;
    NeedType = true;
    break;
……
```

此处的 MLV_ArrayTemporary 就是描述的形如（A+B）产生的表达式类型，其中 A 和 B 都是数组。如果检查时进入了该 case，说明在赋值号左侧出现了形如（A+B）的表达式，这样的赋值是不合法的，我们做出以下修改：

```
case Expr::MLV_ArrayTemporary:
    if (S.ElementWiseContext) return false;
    Diag = diag::err_typecheck_array_not_modifiable_lvalue;
    NeedType = true;
    break;
```

进入该 case，如果发现 ElementWiseContext 为 1，也就是编译制导含有 ElementWise，就要对 CheckAssignmentOperands 返回 false，编译报错。

case2：乘除法

修改函数：

```
QualType Sema::CheckMultiplyDivideOperands(ExprResult &LHS, ExprResult &RHS,
                                           SourceLocation Loc,
                                           bool IsCompAssign, bool IsDiv) {
```

```
///ADD THIS IN PROO2
if (this->ElementWiseContext)
{
  Expr *l = LHS.get(), *r = RHS.get();
  const Type *ltype = l->getType().getTypePtr(),
             *rtype = r->getType().getTypePtr();
  if (ConstantArrayType::classof(ltype) &&
      ConstantArrayType::classof(rtype))
  {
    const ConstantArrayType *LH_arraytype = dyn_cast<ConstantArrayType>(ltype);
    const ConstantArrayType *RH_arraytype = dyn_cast<ConstantArrayType>(rtype);
    //获取数组长度
    llvm::APInt LH_arraylen = LH_arraytype->getSize();
    llvm::APInt RH_arraylen = RH_arraytype->getSize();
    QualType LH_elemtype = LH_arraytype->getElementType().getUnqualifiedType();
    QualType RH_elemtype = RH_arraytype->getElementType().getUnqualifiedType();
```

当 ELementWiseContext 为 true 时，即函数有#elementWise 的定义时：

与之前的类似，先确定左右表达式是否均为常数组类型，之后通过类中的 getSize 方法获得数组的长度，通过 getType 方法获得数组元素的类型（此部分过程与 case1 完全相同）。之后：

```
if (LH_arraylen == RH_arraylen && LH_elemtype == RH_elemtype && LH_elemtype->is_only_IntType())
{
  Qualifiers t;
  if (!l->isRValue())
  {
    ImplicitCastExpr *l12r =\
     ImplicitCastExpr::Create(Context,\
      Context.getUnqualifiedArrayType(l->getType().getUnqualifiedType(), t), \
      CK_LValueToRValue, const_cast<Expr *>(l), 0, VK_RValue);
    LHS = l12r;
  }
  if (!r->isRValue())
  {
    ImplicitCastExpr *rl2r = \
    ImplicitCastExpr::Create(Context, Context.getUnqualifiedArrayType(r->getType().getUnqualifie
    RHS = rl2r;
  }
  return LHS.get()->getType();
}
```

根据 elementWise 的定义，我们需要两个数组等长，即 LH_arraylen==RH_arraylen，元素相同且为整数，即 LH_elemtype == RH_elemtype && LH_elemtype->is_only_IntType()。

之后，如果表达式不是右值，那么新建一个类型转化，将表达式转化为右值的情况。ImplicitCastExpr::Create 实现加入一层 ImplicitCastExpr 后的 AST 节点的创建，我们随后将原有结点替换为新生成的结点即可。

case3：加法

实现过程与 case2 乘除法完全相同。

四、 结果测试：

Test 1：

```
#pragma elementWise
void foo1(){
  int A[1000];
  int B[1000];
  int C[1000];
  C = A + B;
  C = A * B;
  C = A;
}
```

问题 输出 终端 调试控制台 1: bash

```
TranslationUnitDecl 0x5702e70 <<invalid sloc>>
|-TypedefDecl 0x5703350 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x57033b0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5703700 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]
'
`-FunctionDecl 0x57037a0 <test.c:2:1, line:9:1> foo1 'void ()'
  `-CompoundStmt 0x572fec0 <line:2:12, line:9:1>
    |-DeclStmt 0x5703918 <line:3:3, col:14>
    | `-VarDecl 0x57038c0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x57039c8 <line:4:3, col:14>
    | `-VarDecl 0x5703970 <col:3, col:13> B 'int [1000]'
    |-DeclStmt 0x5703a78 <line:5:3, col:14>
    | `-VarDecl 0x5703a20 <col:3, col:13> C 'int [1000]'
    |-BinaryOperator 0x5703b60 <line:6:3, col:11> 'int [1000]' '='
    | |-DeclRefExpr 0x5703a90 <col:3> 'int [1000]' lvalue Var 0x5703a20 'C' '
int [1000]'
    | `-BinaryOperator 0x5703b38 <col:7, col:11> 'int [1000]' '+'
    |   |-ImplicitCastExpr 0x5703b08 <col:7> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x5703ab8 <col:7> 'int [1000]' lvalue Var 0x57038c0 '
A' 'int [1000]'
    |   `-ImplicitCastExpr 0x5703b20 <col:11> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x5703ae0 <col:11> 'int [1000]' lvalue Var 0x5703970
 'B' 'int [1000]'
    |-BinaryOperator 0x572fe20 <line:7:3, col:11> 'int [1000]' '='
    | |-DeclRefExpr 0x572fd50 <col:3> 'int [1000]' lvalue Var 0x5703a20 'C' '
int [1000]'
    | `-BinaryOperator 0x572fdf8 <col:7, col:11> 'int [1000]' '*'
    |   |-ImplicitCastExpr 0x572fdc8 <col:7> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x572fd78 <col:7> 'int [1000]' lvalue Var 0x57038c0 '
A' 'int [1000]'
    |   `-ImplicitCastExpr 0x572fde0 <col:11> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x572fda0 <col:11> 'int [1000]' lvalue Var 0x5703970
 'B' 'int [1000]'
    `-BinaryOperator 0x572fe98 <line:8:3, col:7> 'int [1000]' '='
      |-DeclRefExpr 0x572fe48 <col:3> 'int [1000]' lvalue Var 0x5703a20 'C' '
int [1000]'
      `-DeclRefExpr 0x572fe70 <col:7> 'int [1000]' lvalue Var 0x57038c0 'A' '
int [1000]'
0
```

【测试结果】

打印 0，未报编译错误，合法。

Test 2:

```
1  void foo2(){
2    int A[1000];
3    int B[1000];
4    int C[1000];
5    C = A + B;
6    C = A * B;
7    C = A;
8  }
```

问题 输出 终端 调试控制台 1: bash

```
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -Xclang -as
t-dump -fsyntax-only test.c; echo $?
test.c:5:9: error: invalid operands to binary expression
      ('int *' and 'int *')
  C = A + B;
      ~ ^ ~
test.c:6:9: error: invalid operands to binary expression
      ('int *' and 'int *')
  C = A * B;
      ~ ^ ~
test.c:7:5: error: array type 'int [1000]' is not assignable
  C = A;
```

【测试结果】

编译报错：不合法的操作符，以及不允许的赋值对象。

这是因为缺少了编译制导 elementwise。

Test 3:

```
1  #pragma elementWise
2  void foo3(){
3    int A[1000];
4    int B[1000];
5    int C[1000];
6    int *D;
7    C = D;
8  }
```

问题 输出 终端 调试控制台 1: bash

```
test.c:7:5: error: assigning to 'int [1000]' from incompatible type 'int *'
  C = D;
```

【测试结果】

编译报错：不合法的赋值，赋值左右类型不能兼容。

Test 4:

```
1   #pragma elementWise
2   void foo4(){
3     int A[1000];
4     int B[1000];
5     int C[1000];
6     int *D;
7     (A + B) = C;
8   }
```

问题 输出 终端 调试控制台 1: bash

```
test.c:7:11: error: expression is not assignable
  (A + B) = C;
```

【测试结果】

编译报错：表达式不可被赋值。

Test 5:

```
2   void foo5(){
3     int A[1000];
4     int B[1000];
5     int C[1000];
6     int *D;
7     C = A + D;
8     C = D + A;
9     C = D + D;
10  }
```

问题 输出 终端 调试控制台 1: bash

```
test.c:7:9: error: invalid operands to binary expression
      ('int *' and 'int *')
  C = A + D;
      ~ ^ ~
test.c:8:9: error: invalid operands to binary expression
      ('int *' and 'int *')
  C = D + A;
      ~ ^ ~
test.c:9:9: error: invalid operands to binary expression
      ('int *' and 'int *')
  C = D + D;
```

【测试结果】

编译报错：不合法的操作对象，二元操作对象为指针类。

Test 6:

```
1   #pragma elementWise
2   void foo11(){
3     int C[1000];
4     int *D;
5     D = C;
6   }
```

```
问题  输出  终端  调试控制台  1: bash
clang7@teacher-PowerEdge-M640:~/lhc$ ~/lhc/llvm-install/bin/clang -Xclang -as
t-dump -fsyntax-only test.c; echo $?
TranslationUnitDecl 0x5899e70 <<invalid sloc>>
|-TypedefDecl 0x589a350 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x589a3b0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x589a700 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]
'
`-FunctionDecl 0x589a7a0 <test.c:2:1, line:6:1> foo11 'void ()'
  `-CompoundStmt 0x589aa70 <line:2:13, line:6:1>
    |-DeclStmt 0x589a918 <line:3:3, col:14>
    | `-VarDecl 0x589a8c0 <col:3, col:13> C 'int [1000]'
    |-DeclStmt 0x589a9c8 <line:4:3, col:9>
    | `-VarDecl 0x589a970 <col:3, col:8> D 'int *'
    `-BinaryOperator 0x589aa48 <line:5:3, col:7> 'int *' '='
      |-DeclRefExpr 0x589a9e0 <col:3> 'int *' lvalue Var 0x589a970 'D' 'int *
'
      `-ImplicitCastExpr 0x589aa30 <col:7> 'int *' <ArrayToPointerDecay>
        `-DeclRefExpr 0x589aa08 <col:7> 'int [1000]' lvalue Var 0x589a8c0 'C'
 'int [1000]'
0
```

【测试结果】

输出 0，编译无报错。

这里需要注意该测试与 Test 3 的区别，由于 CheckSingleAssignmentConstraints 中存在

```
// C99 6.5.16.1p1: the left operand is a pointer and the right is
// a null pointer constant.
if ((LHSType->isPointerType() ||
     LHSType->isObjCObjectPointerType() ||
     LHSType->isBlockPointerType())
    && RHS.get()->isNullPointerConstant(Context,
                                        Expr::NPC_ValueDependentIsNull)) {
  RHS = ImpCastExprToType(RHS.take(), LHSType, CK_NullToPointer);
  return Compatible;
}
```

的特判，我们允许给指针赋常数值（如数组起始地址），并且会将一层到指针的类型转换附加在原 RHS 外。

Test 7:

```
1   #pragma elementWise
2   void foo8(){
3     int A[1000];
4     int B[1000];
5     const int C[1000];
6     C = A;
7     C = A + B;
8   }
```

问题 输出 终端 调试控制台 1: bash

```
t-dump -fsyntax-only test.c; echo $?
test.c:6:5: error: read-only variable is not assignable
  C = A;
  ~ ^
test.c:7:5: error: read-only variable is not assignable
  C = A + B;
```

【测试结果】

编译报错：常量不可被赋值。

Test 8:

```
1   #pragma elementWise
2   void foo9(){
3     int A[1000];
4     const int B[1000];
5     int C[1000];
6     C = B;
7     C = A + B;
8   }
```

问题 输出 终端 调试控制台 1: bash

```
TranslationUnitDecl 0x659fe70 <<invalid sloc>>
|-TypedefDecl 0x65a0350 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x65a03b0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x65a0700 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x65a07a0 <test.c:2:1, line:8:1> foo9 'void ()'
  `-CompoundStmt 0x65ccdf8 <line:2:12, line:8:1>
    |-DeclStmt 0x65a0918 <line:3:3, col:14>
    | `-VarDecl 0x65a08c0 <col:3, col:13> A 'int [1000]'
    |-DeclStmt 0x65a0a08 <line:4:3, col:20>
    | `-VarDecl 0x65a09b0 <col:3, col:19> B 'const int [1000]'
    |-DeclStmt 0x65a0ab8 <line:5:3, col:14>
    | `-VarDecl 0x65a0a60 <col:3, col:13> C 'int [1000]'
    |-BinaryOperator 0x65a0b20 <line:6:3, col:7> 'int [1000]' '='
    | |-DeclRefExpr 0x65a0ad0 <col:3> 'int [1000]' lvalue Var 0x65a0a60 'C' 'int [1000]'
    | `-DeclRefExpr 0x65a0af8 <col:7> 'const int [1000]' lvalue Var 0x65a09b0 'B' 'const int [1000]'
    `-BinaryOperator 0x65ccdd0 <line:7:3, col:11> 'int [1000]' '='
      |-DeclRefExpr 0x65a0b48 <col:3> 'int [1000]' lvalue Var 0x65a0a60 'C' 'int [1000]'
      `-BinaryOperator 0x65ccda8 <col:7, col:11> 'int [1000]' '+'
        |-ImplicitCastExpr 0x65ccd78 <col:7> 'int [1000]' <LValueToRValue>
        | `-DeclRefExpr 0x65a0b70 <col:7> 'int [1000]' lvalue Var 0x65a08c0 'A' 'int [1000]'
        `-ImplicitCastExpr 0x65ccd90 <col:11> 'int [1000]' <LValueToRValue>
          `-DeclRefExpr 0x65ccd50 <col:11> 'const int [1000]' lvalue Var 0x65a09b0 'B' 'const int [1000]'
0
```

【测试结果】

输出 0，编译无报错。

Test 9:

```
1   #pragma elementWise
2   void foo10(){
3     int A[1000];
4     int B[1000];
5     int C[1000];
6     int D[1000];
7     D = A + B + C;
8     D = A * B + C;
9     D = (D = A + B);
10    D = (A + B) * C;
11    D = (A + B) * (C + D);
12  }
```

```
问题  输出  终端  调试控制台                                                    1: b
|   `-ImplicitCastExpr 0x67d5290 <col:17> 'int [1000]' <LValueToRValue>
|     `-DeclRefExpr 0x67d5268 <col:17> 'int [1000]' lvalue Var 0x67a8a20 'C' 'int [1000]'
`-BinaryOperator 0x67d54d8 <line:11:3, col:23> 'int [1000]' '='
  |-DeclRefExpr 0x67d52f8 <col:3> 'int [1000]' lvalue Var 0x67a8ad0 'D' 'int [1000]'
  `-BinaryOperator 0x67d54b0 <col:7, col:23> 'int [1000]' '*'
    |-ParenExpr 0x67d53c8 <col:7, col:13> 'int [1000]'
    | `-BinaryOperator 0x67d53a0 <col:8, col:12> 'int [1000]' '+'
    |   |-ImplicitCastExpr 0x67d5370 <col:8> 'int [1000]' <LValueToRValue>
    |   | `-DeclRefExpr 0x67d5320 <col:8> 'int [1000]' lvalue Var 0x67a88c0 'A' 'int [1000]'
    |   `-ImplicitCastExpr 0x67d5388 <col:12> 'int [1000]' <LValueToRValue>
    |     `-DeclRefExpr 0x67d5348 <col:12> 'int [1000]' lvalue Var 0x67a8970 'B' 'int [1000]'
    `-ParenExpr 0x67d5490 <col:17, col:23> 'int [1000]'
      `-BinaryOperator 0x67d5468 <col:18, col:22> 'int [1000]' '+'
        |-ImplicitCastExpr 0x67d5438 <col:18> 'int [1000]' <LValueToRValue>
        | `-DeclRefExpr 0x67d53e8 <col:18> 'int [1000]' lvalue Var 0x67a8a20 'C' 'int [1000]'
        `-ImplicitCastExpr 0x67d5450 <col:22> 'int [1000]' <LValueToRValue>
          `-DeclRefExpr 0x67d5410 <col:22> 'int [1000]' lvalue Var 0x67a8ad0 'D' 'int [1000]'
0
```

【测试结果】

输出 0，编译无报错。

Test 10:

```
1   #pragma elementWise
2   void foo7(){
3     int A[1000];
4     int B[1000];
5     int C[1000];
6     int *D;
7     int E[10][100];
8     E = A;
9     E = A + B;
10    E = A * B;
11  }
```

```
问题  输出  终端  调试控制台                                    1: bash
test.c:8:5: error: assigning to 'int [10][100]' from incompatible type
      'int [1000]'
  E = A;
    ^ ~
test.c:9:5: error: assigning to 'int [10][100]' from incompatible type
      'int [1000]'
  E = A + B;
    ^ ~~~~~
test.c:10:5: error: assigning to 'int [10][100]' from incompatible type
      'int [1000]'
  E = A * B;
```

【测试结果】

编译报错：类型不匹配。

五、总结

本次实验主要是熟悉 class Sema 的结构，尤其是 type 相关方面。通过判断数组类型，数组长度，数组元素类型以及进行左值与右值的转换，使得 AST 中对于数组也支持形如 A = B + C 的子树。

在结合测试样例阅读理解了相关的代码框架之后，我们可以检查已有代码是否对各种情况进行了正确的处理，从而最终通过所有测试。

1. 实验结果总结

本次实验成功实现了任务要求，覆盖了各种要求情况。

2. 分成员总结

李昊宸：

本次实验负责实验代码的书写，结果测试和实验报告的补充。在 PRO 01 的基础上，本次实验需要修改的地方少了很多，需要修改哪里以及如何修改老师也给了很多提示。仿照 llvm 源码的格式，增加新的功能确实很有趣（除了一但出 bug 几乎无法确定是哪里的问题外，因为不能查看全部的源码）。

李颖彦：

本次实验负责实验报告的书写。本次实验相较于第一次实验更加清晰，可能是老师给了更多提示的缘故。三种运算的实现大同小异，重点注意加法和乘法时左值右值的类型转换。

陆润宇：

本次实验负责实验报告的修改和补充。本次实验基础的部分（三种运算的类型检查和 AST 维护）可以结合讲义中的提示，参考代码框架中其他的代码段落进行补充。Assignment 情况下的类型转换会相对复杂一点，也是测试样例中比较特殊的地方，需要更加仔细地理解框架才能给出恰当的处理。