编译原理研讨课实验PR003实验报告

任务说明

> 基于P1函数标注以及P2的AST生成的实现上，进一步产生llvm中间代码（IR）。并能得到正确运行的二进制文件。

成员组成

陈灿宇
宋鹏皓
金越

实验设计

设计思路

> PR3代码生成主要做的就是将`#pragma elementWise`标注函数里形如`C=A+B`、`C=A*B`、`C=A`的表达式进行语义翻译，即产生等价的`for`循环。
>
> 1.设计参考：`CodeGenFunction::EmitForStmt(const ForStmt &S)`函数
> 该函数用来产生`for`语句的中间代码，从该函数我们知道了`for`语句在被翻译为IR时，被分为四个基本块（所谓基本快就是除了最后一条语句之外，中间不会有跳转语句）：
>
> > (1)`for.body` 循环的主体部分，即`for`语句用`{}`括起来的部分。对于elementWise翻译来说，就是`C[i] = A[i] + B[i]`（以加法为例）。该基本块最后无条件跳转到`for.inc`基本快。
> > (2)`for.cond` 循环继续的判断部分，比较累加量和边界值的大小，判断循环是否继续，在elemtnWise翻译中是遍历数组用的index `i` 和 数组大小 `data_num`的比较。 该基本块最终需要做条件跳转，在满足循环条件时跳转到基本快`for.inc`。在不满足时跳转到基本快`for.end`结束循环。
> > (3)`for.inc` 循环累加部分，即通常的`i++`部分.在elemtnWise翻译中是将遍历数组用的index加1. 该基本快结束时无条件跳转到`for.cond`基本快。
> > (4)`for.end` 循环结束部分。

2.关键API：

> 在中间代码API `IRBuilder.h` 文件中定义了很多用于产生特定IR指令的函数，例如用于数据搬运的 `load` 和 `store` 指令，以及无条件跳转指令 `BR` 和条件跳转指令 `CondBR` 。更有用于数据运算的 `Add` 和 `Mul` 指令。调用这些函数，实际上我们就是根据表达式调用API写IR汇编。

3.语法树适配

> 我们生成的AST和老师的大致上相同，但在处理 `C=A` 这种赋值语句时，我们并没有将等号右边的表达式转换为 `ImplicitExpr` ，因此如果按老师给的判断条件的话，就不会进入 `if(ImplicitCastExpr::classof(rhs))` 正确时执行的操作。因此这里进行了我们自己的AST的适配。

4.类型问题

> 由于Clang中的AST和LLVM的IR有着很大的类型区别，所以我们在生成IR指令时，必须要做Clang中的类型到LLVM中类型的转换。

实验实现

> 1. 主要修改的代码位于 `llvm-3.3\tools\clang\lib\CodeGen\CGExpr.cpp` 的函数 `EmitAnyExpr()` 中。llvm在根据AST生成IR调用此函数分析表达式。

> (1). `elementWise` 标注的判断: 这个和老师实验说明里给的条件一致，即表达式的结果是类型是一个数组就进行我们的处理，具体代码：

```
if(E->getType()->getTypeClass() == Type::ConstantArray){
    //elementWise 表达式IR生成
}
```

> (2). 处理的表达式限定：本实验中我们处理的表达式只有形如 `C=A+B` 、 `C=A*B` 、 `C=A` 的形式，而这三种情形下，顶层表达式的 `BinaryOperator` 都是"=" 故我们的第二层筛选条件就是：

```
if(BinaryOperator::classof(E)){
    //上述三种表达式IR生成
}
```

> (3). 遍历数组的变量：即我们等价 `for` 循环中的 `i` 。我们在IR中分配一个AllocaIn类型的临时变量，该变量的类型为无符号整数（但要转换为llvm中的类型），并用一条 `store` 指令对其赋值0从而初始化：

```
      //分配临时变量并做类型转换
      QualType Ty = getContext().UnsignedIntTy;
      llvm::Type *LTy = ConvertTypeForMem(Ty);
      llvm::AllocaInst *Alloc = CreateTempAlloca(LTy);
      Alloc->setName("compiler");
      //设置对齐
      Alloc->setAlignment(4);
      // 初始化临时变量，使用Store指令
      // 此时的常量0需要调用LLVM的API转为LLVM的表示
      llvm::StoreInst *Store = Builder.CreateStore(llvm::ConstantInt::get(LTy, llvm::APInt(32,
0)), \
                                                   (llvm::Value*)Alloc, false);
      Store->setAlignment(4);
```

> (4). 等号左右表达式的获取以及IR中数组地址声明

```
        // 根据AST的形式修改逻辑
        const BinaryOperator* bo = dyn_cast<BinaryOperator>(E);
        assert(bo->getOpcode() == BO_Assign);
        // C = A + B中的C
        Expr* lhs = bo -> getLHS();
        // C = A + B中的 A + B
        Expr* rhs = bo -> getRHS();
        // base指代数组基地址，而addr指代base+offset的地址（值地址）
        llvm::Value *baseC, *addrC;
        llvm::Value *baseA, *addrA;
        llvm::Value *baseB, *addrB;
        //assert函数保证等号左边是DeclRefExpr类型的数组名表达式
        assert(lhs->getType()->getTypeClass() == Type::ConstantArray);
        assert(DeclRefExpr::classof(lhs));
```

> (5). 数组长度获取，复用了P2的方法

```
        //get length of the constant array
        const Type* lhs_type = lhs->getType().getTypePtr();
        const ConstantArrayType* lhs_type_ptr = dyn_cast<ConstantArrayType>(lhs_type);
        const llvm::APInt lhs_data_num = lhs_type_ptr->getSize();
```

> (6). 基本块声明：包括实验设计中提到的4个基本快：

```
        //create basicblock ForCond ForBody ForInc ForEnd
        llvm::BasicBlock *ForCond = createBasicBlock("for.cond");
        llvm::BasicBlock *ForBody = createBasicBlock("for.body");
        llvm::BasicBlock *ForInc  = createBasicBlock("for.inc");
        llvm::BasicBlock *ForEnd  = createBasicBlock("for.end");
```

> (7). `for.cond`基本块：

```
        //通过Emit函数产生标号
        EmitBlock(ForCond);
        //将compilerload到一个IR中间量中进行类型提升，之后用这个idx进行累加遍历数组
        llvm::LoadInst *idx = Builder.CreateLoad((llvm::Value*)Alloc, "");
        idx->setAlignment(4);
        // 类型提升（i32->i64）
        llvm::Value* idxPromoted = Builder.CreateIntCast(idx, IntPtrTy, false, "idxprom");

        //需要设置数组大小值的类型并转换为llvm类型
        QualType Ty_cmp = getContext().UnsignedLongTy;
        llvm::Type *LTy_cmp = ConvertType(Ty_cmp);
        //产生比较语句和条件跳转语句：当i的值小于数组长度时跳转到for.body
        //否则结束循环跳转到for.end
        llvm::Value *cmp = Builder.CreateICmpSLT(idxPromoted, \
                                                 (llvm::Value*)llvm::ConstantInt::get(LTy_cmp,
lhs_data_num));

        Builder.CreateCondBr(cmp, ForBody, ForEnd);
```

(8). `for.body`分为加法乘法和赋值两个部分：

> i. 加法乘法：判断条件为等号右边表达式为一个`BinaryOperator`。首先获取这个`BinaryOperator`，并分别获得它的左子表达式和右子表达式。（如果有一个不是静态数组则报错退出），当这个`BinaryOperator`为`+`或者`*`号时，生成对应的计算代码。首先需要获取前面声明的数组基地址`baseC`，具体操作时根据表达式的声明(先转换为`DeclRefExpr`类然后再调用`getDecl()`得到`ValueDecl`指针)在局部变量表里获得C对应的地址（llvm的值）。A和B的基址也类似得到。只不过在等号右边的表达式需要先转换为`ImplicitCastExpr`。得到基址后，需要将这三个基址用`LValue`类存储（这是参考了`EmitDeclRefLValue函数`）。然后调用这个类的`getAddress`函数得到三者的基地址指针（用llvm的`Value`类存储）。然后调用IR的数组API中的`CreateInBoundsGEP`函数，输入参数(包括基地址值、边界检测函数（i32 的 0)以及数组遍历变量(i64 前面进行了类型提升的`idxPromoted`)),我们就得到了数组元素的指针（存在了前面声明的三个`addr`变量中），然后我们需要将这个地址里的值`load`到一个IR中间量里进行运算，即根据`BinaryOperator`的不同，调用不同的`Create运算指令`，最后结果存在`addrC`地址里。

```
EmitBlock(ForBody);
     if(BinaryOperator::classof(rhs)){
       // Case : C = A + B or C = A * B
       const BinaryOperator* bo1 = dyn_cast<BinaryOperator>(rhs);
       // A + B中的A和B
       Expr* lhs1 = bo1 -> getLHS();
       Expr* rhs1 = bo1 -> getRHS();
       assert(lhs1->getType()->getTypeClass() == Type::ConstantArray);
       assert(rhs1->getType()->getTypeClass() == Type::ConstantArray);
       if(bo1->getOpcode() == BO_Add || bo1 -> getOpcode() == BO_Mul){
         // 针对C[compiler]
         const DeclRefExpr *declRef = dyn_cast<DeclRefExpr>(lhs);
         // 拿到C对应的Decl
         const ValueDecl* decl = declRef -> getDecl();
         // 根据C的Decl从局部变量表中取到C对应的LLVM Value
         baseC = LocalDeclMap.lookup(decl);

         assert(ImplicitCastExpr::classof(rhs1));
         assert(ImplicitCastExpr::classof(lhs1));
         //等号左边的表达式做转换
         const ImplicitCastExpr* rhs2 = dyn_cast<ImplicitCastExpr>(rhs1);
         const ImplicitCastExpr* lhs2 = dyn_cast<ImplicitCastExpr>(lhs1);
         // A和B对应的LLVM Value
         const DeclRefExpr *declRefR1 = dyn_cast<DeclRefExpr>(lhs2->getSubExpr());
         baseA = LocalDeclMap.lookup(declRefR1->getDecl());
         const DeclRefExpr *declRefR2 = dyn_cast<DeclRefExpr>(rhs2->getSubExpr());
         baseB = LocalDeclMap.lookup(declRefR2->getDecl());
         // 对齐信息
         const ValueDecl *VD = declRefR2->getDecl();
         CharUnits Alignment = getContext().getDeclAlign(VD);
         // 类型信息
         QualType T = declRefR2->getType();
         // 三个左值
         LValue LVC, LVA, LVB;
         // 参照EmitDeclRefLValue, 来获得C、A、和B的指针
         LVC = MakeAddrLValue(baseC, T, Alignment);
         LVA = MakeAddrLValue(baseA, T, Alignment);
         LVB = MakeAddrLValue(baseB, T, Alignment);
         llvm::Value *arrayPtrC = LVC.getAddress();
         llvm::Value *arrayPtrA = LVA.getAddress();
         llvm::Value *arrayPtrB = LVB.getAddress();
         // 边界检查参数
         llvm::Value *Zero = llvm::ConstantInt::get(Int32Ty, 0);
         llvm::Value *Args[] = { Zero, idxPromoted };
         // 参照EmitArraySubscriptExpr
         // GEP:  Get Element Pointer
         addrC = Builder.CreateInBoundsGEP(arrayPtrC, Args, "arrayidx");
         addrA = Builder.CreateInBoundsGEP(arrayPtrA, Args, "arrayidx");
         addrB = Builder.CreateInBoundsGEP(arrayPtrB, Args, "arrayidx");
         // 读A[compiler]和B[compiler]
         llvm::LoadInst *valueA = Builder.CreateLoad(addrA, "");
         llvm::LoadInst *valueB = Builder.CreateLoad(addrB, "");

         valueA->setAlignment(4);
```

```
          valueB->setAlignment(4);

          if(bo1->getOpcode() == BO_Add){
            //生成加法指令
            llvm::Value* add = Builder.CreateAdd((llvm::Value*)valueA, (llvm::Value*)valueB,
"add");
            // 写C[compiler]
            llvm::StoreInst *valueC = Builder.CreateStore(add, addrC, false);
            valueC ->setAlignment(4);
          }else if(bo1->getOpcode() == BO_Mul){
            //生成乘法指令
            llvm::Value* add = Builder.CreateMul((llvm::Value*)valueA, (llvm::Value*)valueB,
"mul");
            // 写C[compiler]
            llvm::StoreInst *valueC = Builder.CreateStore(add, addrC, false);
            valueC ->setAlignment(4);
          }
```

> ii . 赋值表达式，和加法乘法类似，但注意这里等号右边的表达式不需要进行向`ImplicitCastExpr`类的转换，因为在我们生成的语法树里，赋值表达式等号左右两边都是`DeclRefExpr`。

```
else{
      // 针对C[compiler]
      const DeclRefExpr *lhs_declRef = dyn_cast<DeclRefExpr>(lhs);
      // 拿到C对应的Decl
      const ValueDecl* lhs_decl = lhs_declRef -> getDecl();
      // 根据C的Decl从局部变量表中取到C对应的LLVM Value
      baseC = LocalDeclMap.lookup(lhs_decl);

      // 针对A[compiler]
      const DeclRefExpr *rhs_declRef = dyn_cast<DeclRefExpr>(rhs);
      baseA = LocalDeclMap.lookup(rhs_declRef -> getDecl());

      // 对齐信息
      const ValueDecl *VD = rhs_declRef->getDecl();
      CharUnits Alignment = getContext().getDeclAlign(VD);
      // 类型信息
      QualType T = rhs_declRef->getType();

      // 三个左值
      LValue LVC, LVA;
      // 参照EmitDeclRefLValue, 来获得C的指针
      LVC = MakeAddrLValue(baseC, T, Alignment);
      LVA = MakeAddrLValue(baseA, T, Alignment);
      llvm::Value *arrayPtrC = LVC.getAddress();
      llvm::Value *arrayPtrA = LVA.getAddress();
      // 边界检查参数
      llvm::Value *Zero = llvm::ConstantInt::get(Int32Ty, 0);
      llvm::Value *Args[] = { Zero, idxPromoted };
      // 参照EmitArraySubscriptExpr
      // GEP:  Get Element Pointer
      addrC = Builder.CreateInBoundsGEP(arrayPtrC, Args, "arrayidx");
      addrA = Builder.CreateInBoundsGEP(arrayPtrA, Args, "arrayidx");
      // 读A[compiler]和B[compiler]
      llvm::LoadInst *valueA = Builder.CreateLoad(addrA, "");
      valueA->setAlignment(4);

      llvm::StoreInst *valueC = Builder.CreateStore((llvm::Value*)valueA, addrC, false);
      valueC->setAlignment(4);
    }
```

> (9). `for.inc`基本块，完成对遍历数组变量`idx`的累加。

```
      EmitBlock(ForInc);
      llvm::Value* incIdx = Builder.CreateAdd((llvm::Value*)idx,
                                              (llvm::Value*)llvm::ConstantInt::get(LTy,
llvm::APInt(32,1)), "add");
      llvm::StoreInst* Store_inc = Builder.CreateStore(incIdx, (llvm::Value*)Alloc, false);
      Store_inc->setAlignment(4);
      Builder.CreateBr(ForCond);
```

> (10). `for.end`基本块，随便返回一个Rvalue

> 2. P3中还修复了P2中的一个bug:利用P2原来的代码，在`elementWise`标注的函数中调用`printf`等库函数会产生`"type mismatch in call argument!"`的错误，跟踪发现是`CheckSingleAssignmentConstraints()`其被调用的某些地方（该函数在不同地方被调用了多次），在有`elementWise`标注的情况下会返回不正确的值而产生错误，于是重写了这个函数，为`CheckSingleAssignmentConstraintsForElementWisePragma`专门供P2生成AST中使用，其他地方的调用的还是函数`CheckSingleAssignmentConstraints()`的原来版本：

> (1). `llvm-3.3\tools\clang\include\clang\Sema\Sema.h`中添加函数声明

```
AssignConvertType
  CheckSingleAssignmentConstraintsForElementWisePragma(QualType LHSType,
                                                       ExprResult &RHS,
                                                       bool Diagnose = true);
```

> (2). `llvm-3.3\tools\clang\include\clang\Sema\SemaExpr.cpp`中添加重写函数的定义：

```
Sema::AssignConvertType
Sema::CheckSingleAssignmentConstraintsForElementWisePragma(QualType LHSType, ExprResult &RHS,
                                                           bool Diagnose) {

  if(IsElementWise == 1 && ConstantArrayType::classof(RHS.get()->getType().getTypePtr())) {
    return Compatible;
  }

...
}
```

其它

总结

实验结果总结

分成员总结

> 组员：金越
> 这次实验完成了`#pragma elementWise`编译制导的最后一部分：代码生成。实际上就是对我们数组运算表达式进行语义翻译，翻译为等价的`for`循环的过程。参考`llvm`中原有的`for`语句翻译是很有帮助的，这使得我们知道了将`for`循环划分为基本快，其间采用跳转语句实现连接的逻辑。根据这个我们又找到了产生IR指令的API，那么实际上整个实验的实现就很清除明了了，剩下的只是实现细节的问题，比如类型转换，AST适配等。在Debug的过程中我们遇到了很多不可思议的错误，比如一个加了`#pragma elementWise`标注的`main()`函数在调用printf等库函数时会报错：发生`"type mismatch in call argument!"`的`fail`。这一开始让我们束手无策，但通过dump`main`函数的AST我们发现有和没有`#pragma elementWise`标注的两种情况下，printf参数的类型有很大的不同，于是我们猜想可能是我们PRJ2更改的函数在别的地方调用时错误的转换了类型，而在这些修改了的函数中`CheckSingleAssignmentConstraints()`在非我们调用的其他地方调用了很多次，于是我们重写了一个`CheckSingleAssignmentConstraints()`单独用于我们AST中的调用，剩余的调用用的是原来函数。果然，这样做就将这个问题完美的解决了。Debug的过程很辛苦，但由此也积累了很多经验，也算是值得的付出。至此我们完成了`#pragma elementWise`的实验，最终使得编译器支持数组整体的加法、乘法和赋值运算。

> 组员：宋鹏皓
> 本次实验一开始以为比较简单，但是在上手时还是花费了很多精力。阅读老师给的实验说明后，对整个实验思路有了一个基础的了解。在将单次操作C[0]=A[0]+B[0]扩展到循环时，我一开始的想法是使用直接使用for循环来解决，但这样一是for循环并不好实现对APInt类型数据进行计算和赋值操作，二是生成的IR代码会特别长，特别是在数组很大的情况下。所以我们改变思路去直接生成跳转的中间代码，这也是第一个难点，即如何找到生成跳转的IR指令的函数接口。这一点上我们向同学寻求了帮助。完成代码编写后我们还遇到了一些棘手的问题，比如pragma elementWise编译指导下的函数如果调用printf会报错，以及printf函数在有无pragma elementWise的时候的AST树不一样。为了解决这些问题，我们不断从基础出发，思考这些问题产生的根源，然后在代码中做比对，如此反复。这个过程虽然有一些辛苦，但是这种寻找问题解决思路的技巧和能力得到了长足的提升。

> 组员：陈灿宇
> 本次实验过程还是比较简单的，对照clang对于for循环语句生成的IR代码，我们可以得到很大的启发，但是DEBUG的过程却比前两次实验更加困难，出现了一些意想不到的情况，但这也让我积累了很多经验，在一个庞大的代码库中添加代码时，如何才能保证不对原有功能造成影响呢？一方面是需要写更多的测试样例，另一方面是要保证有良好的编写代码的习惯。比如我们在本实验遇到的`"type mismatch in call argument!"`的BUG，就是因为P2的一个小疏漏造成的，但是因为P2的测试样例太少，不可能保证不引入任何BUG，结果就造成在P3中花费了很多时间。因此可以总结一些经验：另可写更多的函数，不同的函数完成更小的功能，也不再一个函数内整合过多的功能，可能暂时看起来没有问题，但是长期来看，增加了调用了此函数的风险。对于已经被调用很多次的函数来说，更要谨慎增加或删减功能，除非完全理解了函数的全部功能以及与外界的所有调用关系；精准设计函数对于某一个小问题，在一个大的项目中是有必要的，尤其是顶层函数的精准设计，而对于小的功能性的经过检验的的函数要尽可能复用。

成员贡献

> 组员：金越 ： 共同讨论实现代码，撰写实验报告
> 组员：陈灿宇 ： 共同讨论实现代码，补充实验报告
> 组员：宋鹏皓 ： 共同讨论实现代码，补充实验报告

测试结果

> 执行命令：`sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_1.c P3_test_1 (P3_test_1.c ~ P3_test_9.c)`

> 测试结果如下：

- 实验1 ：

  测试程序：P3_test_1.c

```c
#include <stdio.h>

#pragma elementWise
void foo1()
{
    int A[1000];
    int B[1000];
    int C[1000];
    for(int i = 0; i < 1000; i++){
        A[i] = i;
        B[i] = i;
    }
    C = A + B;
    printf("%d\n", C[1]);
    C = A * B;
    printf("%d\n", C[1]);
}

int main(){
    foo1();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_1.c P3_test_1
[testing] /home/clang9/PR003/test/P3_test_1.c
[generating] /home/clang9/PR003/bin/P3_test_1
[clang9@host2 ~]$ ./PR003/bin/P3_test_1
2
1
```

```
|-FunctionDecl 0x70e1000 </home/clang9/PR003/test/P3_test_1.c:4:1, line:17:1> foo1 'void ()'
| `-CompoundStmt 0x70e31b0 <line:5:1, line:17:1>
|   |-DeclStmt 0x70e2738 <line:6:5, col:16>
|   | `-VarDecl 0x70e26e0 <col:5, col:15> A 'int [1000]'
|   |-DeclStmt 0x70e27e8 <line:7:5, col:16>
|   | `-VarDecl 0x70e2790 <col:5, col:15> B 'int [1000]'
|   |-DeclStmt 0x70e2898 <line:8:5, col:16>
|   | `-VarDecl 0x70e2840 <col:5, col:15> C 'int [1000]'
|   |-ForStmt 0x70e2c68 <line:9:5, line:12:5>
|   | |-DeclStmt 0x70e2938 <line:9:9, col:18>
|   | | `-VarDecl 0x70e28c0 <col:9, col:17> i 'int'
|   | |   `-IntegerLiteral 0x70e2918 <col:17> 'int' 0
|   | |-<<<NULL>>>
|   | |-BinaryOperator 0x70e29b0 <col:20, col:24> 'int' '<'
|   | | |-ImplicitCastExpr 0x70e2998 <col:20> 'int' <LValueToRValue>
|   | | | `-DeclRefExpr 0x70e2950 <col:20> 'int' lvalue Var 0x70e28c0 'i' 'int'
|   | | `-IntegerLiteral 0x70e2978 <col:24> 'int' 1000
|   | |-UnaryOperator 0x70e2a00 <col:30, col:31> 'int' postfix '++'
|   | | `-DeclRefExpr 0x70e29d8 <col:30> 'int' lvalue Var 0x70e28c0 'i' 'int'
|   | `-CompoundStmt 0x70e2c40 <col:34, line:12:5>
|   |   |-BinaryOperator 0x70e2b08 <line:10:9, col:16> 'int' '='
|   |   | |-ArraySubscriptExpr 0x70e2aa0 <col:9, col:12> 'int' lvalue
|   |   | | |-ImplicitCastExpr 0x70e2a70 <col:9> 'int *' <ArrayToPointerDecay>
|   |   | | | `-DeclRefExpr 0x70e2a20 <col:9> 'int [1000]' lvalue Var 0x70e26e0 'A' 'int [1000]'
|   |   | | `-ImplicitCastExpr 0x70e2a88 <col:11> 'int' <LValueToRValue>
|   |   | |   `-DeclRefExpr 0x70e2a48 <col:11> 'int' lvalue Var 0x70e28c0 'i' 'int'
|   |   | `-ImplicitCastExpr 0x70e2af0 <col:16> 'int' <LValueToRValue>
|   |   |   `-DeclRefExpr 0x70e2ac8 <col:16> 'int' lvalue Var 0x70e28c0 'i' 'int'
|   |   `-BinaryOperator 0x70e2c18 <line:11:9, col:16> 'int' '='
|   |     |-ArraySubscriptExpr 0x70e2bb0 <col:9, col:12> 'int' lvalue
|   |     | |-ImplicitCastExpr 0x70e2b80 <col:9> 'int *' <ArrayToPointerDecay>
|   |     | | `-DeclRefExpr 0x70e2b30 <col:9> 'int [1000]' lvalue Var 0x70e2790 'B' 'int [1000]'
|   |     | `-ImplicitCastExpr 0x70e2b98 <col:11> 'int' <LValueToRValue>
|   |     |   `-DeclRefExpr 0x70e2b58 <col:11> 'int' lvalue Var 0x70e28c0 'i' 'int'
|   |     `-ImplicitCastExpr 0x70e2c00 <col:16> 'int' <LValueToRValue>
|   |       `-DeclRefExpr 0x70e2bd8 <col:16> 'int' lvalue Var 0x70e28c0 'i' 'int'
|   |-BinaryOperator 0x70e2d78 <line:13:5, col:13> 'int [1000]' '='
|   | |-DeclRefExpr 0x70e2ca8 <col:5> 'int [1000]' lvalue Var 0x70e2840 'C' 'int [1000]'
|   | `-BinaryOperator 0x70e2d50 <col:9, col:13> 'int [1000]' '+'
|   |   |-ImplicitCastExpr 0x70e2d20 <col:9> 'int [1000]' <LValueToRValue>
|   |   | `-DeclRefExpr 0x70e2cd0 <col:9> 'int [1000]' lvalue Var 0x70e26e0 'A' 'int [1000]'
|   |   `-ImplicitCastExpr 0x70e2d38 <col:13> 'int [1000]' <LValueToRValue>
|   |     `-DeclRefExpr 0x70e2cf8 <col:13> 'int [1000]' lvalue Var 0x70e2790 'B' 'int [1000]'
|   |-CallExpr 0x70e2ec0 <line:14:5, col:24> 'int'
|   | |-ImplicitCastExpr 0x70e2ea8 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|   | | `-DeclRefExpr 0x70e2da0 <col:5> 'int (const char *, ...)' Function 0x70d3570 'printf' 'int (c
onst char *, ...)'
|   | |-ImplicitCastExpr 0x70e2f10 <col:12> 'const char *' <BitCast>
|   | | `-ImplicitCastExpr 0x70e2ef8 <col:12> 'char *' <ArrayToPointerDecay>
|   | |   `-StringLiteral 0x70e2dc8 <col:12> 'char [4]' lvalue "%d\n"
|   | `-ImplicitCastExpr 0x70e2f28 <col:20, col:23> 'int' <LValueToRValue>
|   |   `-ArraySubscriptExpr 0x70e2e58 <col:20, col:23> 'int' lvalue
|   |     |-ImplicitCastExpr 0x70e2e40 <col:20> 'int *' <ArrayToPointerDecay>
|   |     | `-DeclRefExpr 0x70e2df8 <col:20> 'int [1000]' lvalue Var 0x70e2840 'C' 'int [1000]'
|   |     `-IntegerLiteral 0x70e2e20 <col:22> 'int' 1
|   |-BinaryOperator 0x70e3010 <line:15:5, col:13> 'int [1000]' '='
|   | |-DeclRefExpr 0x70e2f40 <col:5> 'int [1000]' lvalue Var 0x70e2840 'C' 'int [1000]'
|   | `-BinaryOperator 0x70e2fe8 <col:9, col:13> 'int [1000]' '*'
|   |   |-ImplicitCastExpr 0x70e2fb8 <col:9> 'int [1000]' <LValueToRValue>
|   |   | `-DeclRefExpr 0x70e2f68 <col:9> 'int [1000]' lvalue Var 0x70e26e0 'A' 'int [1000]'
|   |   `-ImplicitCastExpr 0x70e2fd0 <col:13> 'int [1000]' <LValueToRValue>
|   |     `-DeclRefExpr 0x70e2f90 <col:13> 'int [1000]' lvalue Var 0x70e2790 'B' 'int [1000]'
|   `-CallExpr 0x70e3130 <line:16:5, col:24> 'int'
|     |-ImplicitCastExpr 0x70e3118 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|     | `-DeclRefExpr 0x70e3038 <col:5> 'int (const char *, ...)' Function 0x70d3570 'printf' 'int (c
onst char *, ...)'
|     |-ImplicitCastExpr 0x70e3180 <col:12> 'const char *' <BitCast>
|     | `-ImplicitCastExpr 0x70e3168 <col:12> 'char *' <ArrayToPointerDecay>
|     |   `-StringLiteral 0x70e3060 <col:12> 'char [4]' lvalue "%d\n"
|     `-ImplicitCastExpr 0x70e3198 <col:20, col:23> 'int' <LValueToRValue>
|       `-ArraySubscriptExpr 0x70e30f0 <col:20, col:23> 'int' lvalue
```

```
|-ImplicitCastExpr 0x70e30d8 <col:20> 'int *' <ArrayToPointerDecay>
| `-DeclRefExpr 0x70e3090 <col:20> 'int [1000]' lvalue Var 0x70e2840 'C' 'int [1000]'
`-IntegerLiteral 0x70e30b8 <col:22> 'int' 1
`-FunctionDecl 0x70e3230 <line:19:1, line:21:1> main 'int ()'
  `-CompoundStmt 0x70e3368 <line:19:11, line:21:1>
    `-CallExpr 0x70e3340 <line:20:5, col:10> 'void'
      `-ImplicitCastExpr 0x70e3328 <col:5> 'void (*)()' <FunctionToPointerDecay>
        `-DeclRefExpr 0x70e32d0 <col:5> 'void ()' Function 0x70e1000 'foo1' 'void ()'
```

- 实验2：

  测试程序：P3_test_2.c

```c
#include <stdio.h>

#pragma elementWise
void foo2()
{
    int A[1000];
    int B[1000];
    int C[1000];
    for(int i = 0; i < 1000; i++){
        A[i] = i;
        B[i] = 2*i;
    }
    C = A;
    printf("%d\n", C[1]);
    C = B;
    printf("%d\n", C[1]);
}

int main(){
    foo2();
}
```

  实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_2.c P3_test_2
[testing] /home/clang9/PR003/test/P3_test_2.c
[generating] /home/clang9/PR003/bin/P3_test_2
[clang9@host2 ~]$ ./PR003/bin/P3_test_2
1
2
```

```
|-FunctionDecl 0x523bff0 </home/clang9/PR003/test/P3_test_2.c:4:1, line:17:1> foo2 'void ()'
| `-CompoundStmt 0x523e0f0 <line:5:1, line:17:1>
|   |-DeclStmt 0x523d728 <line:6:5, col:16>
|   | `-VarDecl 0x523d6d0 <col:5, col:15> A 'int [1000]'
|   |-DeclStmt 0x523d7d8 <line:7:5, col:16>
|   | `-VarDecl 0x523d780 <col:5, col:15> B 'int [1000]'
|   |-DeclStmt 0x523d888 <line:8:5, col:16>
|   | `-VarDecl 0x523d830 <col:5, col:15> C 'int [1000]'
|   |-ForStmt 0x523dca0 <line:9:5, line:12:5>
|   | |-DeclStmt 0x523d928 <line:9:9, col:18>
|   | | `-VarDecl 0x523d8b0 <col:9, col:17> i 'int'
|   | |   `-IntegerLiteral 0x523d908 <col:17> 'int' 0
|   | |-<<<NULL>>>
|   | |-BinaryOperator 0x523d9a0 <col:20, col:24> 'int' '<'
|   | | |-ImplicitCastExpr 0x523d988 <col:20> 'int' <LValueToRValue>
|   | | | `-DeclRefExpr 0x523d940 <col:20> 'int' lvalue Var 0x523d8b0 'i' 'int'
|   | | `-IntegerLiteral 0x523d968 <col:24> 'int' 1000
|   | |-UnaryOperator 0x523d9f0 <col:30, col:31> 'int' postfix '++'
|   | | `-DeclRefExpr 0x523d9c8 <col:30> 'int' lvalue Var 0x523d8b0 'i' 'int'
|   | `-CompoundStmt 0x523dc78 <col:34, line:12:5>
|   |   |-BinaryOperator 0x523daf8 <line:10:9, col:16> 'int' '='
|   |   | |-ArraySubscriptExpr 0x523da90 <col:9, col:12> 'int' lvalue
|   |   | | |-ImplicitCastExpr 0x523da60 <col:9> 'int *' <ArrayToPointerDecay>
|   |   | | | `-DeclRefExpr 0x523da10 <col:9> 'int [1000]' lvalue Var 0x523d6d0 'A' 'int [1000]'
|   |   | | `-ImplicitCastExpr 0x523da78 <col:11> 'int' <LValueToRValue>
|   |   | |   `-DeclRefExpr 0x523da38 <col:11> 'int' lvalue Var 0x523d8b0 'i' 'int'
|   |   | `-ImplicitCastExpr 0x523dae0 <col:16> 'int' <LValueToRValue>
|   |   |   `-DeclRefExpr 0x523dab8 <col:16> 'int' lvalue Var 0x523d8b0 'i' 'int'
|   |   `-BinaryOperator 0x523dc50 <line:11:9, col:18> 'int' '='
|   |     |-ArraySubscriptExpr 0x523dba0 <col:9, col:12> 'int' lvalue
|   |     | |-ImplicitCastExpr 0x523db70 <col:9> 'int *' <ArrayToPointerDecay>
|   |     | | `-DeclRefExpr 0x523db20 <col:9> 'int [1000]' lvalue Var 0x523d780 'B' 'int [1000]'
|   |     | `-ImplicitCastExpr 0x523db88 <col:11> 'int' <LValueToRValue>
|   |     |   `-DeclRefExpr 0x523db48 <col:11> 'int' lvalue Var 0x523d8b0 'i' 'int'
|   |     `-BinaryOperator 0x523dc28 <col:16, col:18> 'int' '*'
|   |       |-IntegerLiteral 0x523dbc8 <col:16> 'int' 2
|   |       `-ImplicitCastExpr 0x523dc10 <col:18> 'int' <LValueToRValue>
|   |         `-DeclRefExpr 0x523dbe8 <col:18> 'int' lvalue Var 0x523d8b0 'i' 'int'
|   |-BinaryOperator 0x523dd30 <line:13:5, col:9> 'int [1000]' '='
|   | |-DeclRefExpr 0x523dce0 <col:5> 'int [1000]' lvalue Var 0x523d830 'C' 'int [1000]'
|   | `-DeclRefExpr 0x523dd08 <col:9> 'int [1000]' lvalue Var 0x523d6d0 'A' 'int [1000]'
|   |-CallExpr 0x523de80 <line:14:5, col:24> 'int'
|   | |-ImplicitCastExpr 0x523de68 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|   | | `-DeclRefExpr 0x523dd58 <col:5> 'int (const char *, ...)' Function 0x522e560 'printf' 'int (c
onst char *, ...)'
|   | |-ImplicitCastExpr 0x523ded0 <col:12> 'const char *' <BitCast>
|   | | `-ImplicitCastExpr 0x523deb8 <col:12> 'char *' <ArrayToPointerDecay>
|   | |   `-StringLiteral 0x523dd80 <col:12> 'char [4]' lvalue "%d\n"
|   | `-ImplicitCastExpr 0x523dee8 <col:20, col:23> 'int' <LValueToRValue>
|   |   `-ArraySubscriptExpr 0x523de10 <col:20, col:23> 'int' lvalue
|   |     |-ImplicitCastExpr 0x523ddf8 <col:20> 'int *' <ArrayToPointerDecay>
|   |     | `-DeclRefExpr 0x523ddb0 <col:20> 'int [1000]' lvalue Var 0x523d830 'C' 'int [1000]'
|   |     `-IntegerLiteral 0x523ddd8 <col:22> 'int' 1
|   |-BinaryOperator 0x523df50 <line:15:5, col:9> 'int [1000]' '='
|   | |-DeclRefExpr 0x523df00 <col:5> 'int [1000]' lvalue Var 0x523d830 'C' 'int [1000]'
|   | `-DeclRefExpr 0x523df28 <col:9> 'int [1000]' lvalue Var 0x523d780 'B' 'int [1000]'
|   `-CallExpr 0x523e070 <line:16:5, col:24> 'int'
|     |-ImplicitCastExpr 0x523e058 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|     | `-DeclRefExpr 0x523df78 <col:5> 'int (const char *, ...)' Function 0x522e560 'printf' 'int (c
onst char *, ...)'
|     |-ImplicitCastExpr 0x523e0c0 <col:12> 'const char *' <BitCast>
|     | `-ImplicitCastExpr 0x523e0a8 <col:12> 'char *' <ArrayToPointerDecay>
|     |   `-StringLiteral 0x523dfa0 <col:12> 'char [4]' lvalue "%d\n"
|     `-ImplicitCastExpr 0x523e0d8 <col:20, col:23> 'int' <LValueToRValue>
|       `-ArraySubscriptExpr 0x523e030 <col:20, col:23> 'int' lvalue
|         |-ImplicitCastExpr 0x523e018 <col:20> 'int *' <ArrayToPointerDecay>
|         | `-DeclRefExpr 0x523dfd0 <col:20> 'int [1000]' lvalue Var 0x523d830 'C' 'int [1000]'
|         `-IntegerLiteral 0x523dff8 <col:22> 'int' 1
`-FunctionDecl 0x523e170 <line:19:1, line:21:1> main 'int ()'
  `-CompoundStmt 0x523e2a8 <line:19:11, line:21:1>
    `-CallExpr 0x523e280 <line:20:5, col:10> 'void'
```

```
    `-ImplicitCastExpr 0x523e268 <col:5> 'void (*)()' <FunctionToPointerDecay>
      `-DeclRefExpr 0x523e210 <col:5> 'void ()' Function 0x523bff0 'foo2' 'void ()'
```

- 实验3：

  测试程序：P3_test_3.c

```c
#pragma elementWise
void foo3()
{
    int A[1000];
    int B[1000];
    int C[1000];
    int *D;
    C = D;
    D = C;
}

int main(){
    foo3();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_3.c P3_test_3
/home/clang9/PR003/test/P3_test_3.c:8:7: error: assigning to 'int [1000]' from incompatible type
      'int *'
    C = D;
      ^ ~
1 error generated.
[testing] /home/clang9/PR003/test/P3_test_3.c
[generating] /home/clang9/PR003/bin/P3_test_3
```

- 实验4：

  测试程序：P3_test_4.c

```c
#pragma elementWise
void foo4()
{
    int A[1000];
    int B[1000];
    int C[1000];
    int *D;
    (A + B) = C;
}

int main(){
    foo4();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_4.c P3_test_4
/home/clang9/PR003/test/P3_test_4.c:8:13: error: expression is not assignable
    (A + B) = C;
    ~~~~~~~ ^
1 error generated.
[testing] /home/clang9/PR003/test/P3_test_4.c
[generating] /home/clang9/PR003/bin/P3_test_4
```

- 实验5：

  测试程序：P3_test_5.c

```
#pragma elementWise
void foo5()
{
    int A[1000];
    int B[1000];
    int C[1000];
    int *D;
    C = A + D;
    C = D + A;
    C = D + D;
}

int main(){
    foo5();
}
```

  实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_5.c P3_test_5
/home/clang9/PR003/test/P3_test_5.c:8:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    C = A + D;
        ~ ^ ~
/home/clang9/PR003/test/P3_test_5.c:9:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    C = D + A;
        ~ ^ ~
/home/clang9/PR003/test/P3_test_5.c:10:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    C = D + D;
        ~ ^ ~
3 errors generated.
[testing] /home/clang9/PR003/test/P3_test_5.c
[generating] /home/clang9/PR003/bin/P3_test_5
```

- 实验6：

  测试程序：P3_test_6.c

```
#pragma elementWise
void foo6()
{
    int A[1000];
    int B[1000];
    int C[1000];
    int *D;
    (A + B) = C;
}

int main(){
    foo6();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_6.c P3_test_6
/home/clang9/PR003/test/P3_test_6.c:8:13: error: expression is not assignable
    (A + B) = C;
    ~~~~~~~ ^
1 error generated.
[testing] /home/clang9/PR003/test/P3_test_6.c
[generating] /home/clang9/PR003/bin/P3_test_6
```

- 实验7：

  测试程序：P3_test_7.c

```c
void foo7()
{
    int A[1000];
    int B[1000];
    int C[1000];
    int *D;
    int E[10][100];
    E = A;
    E = A + B;
    E = A * B;
}

int main(){
    foo7();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_7.c P3_test_7
/home/clang9/PR003/test/P3_test_7.c:8:7: error: array type 'int [10][100]' is not assignable
    E = A;
    ~ ^
/home/clang9/PR003/test/P3_test_7.c:9:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    E = A + B;
        ~ ^ ~
/home/clang9/PR003/test/P3_test_7.c:10:11: error: invalid operands to binary expression
      ('int *' and 'int *')
    E = A * B;
        ~ ^ ~
3 errors generated.
[testing] /home/clang9/PR003/test/P3_test_7.c
[generating] /home/clang9/PR003/bin/P3_test_7
```

- 实验8：

  测试程序：P3_test_8.c

```
#pragma elementWise
void foo8()
{
    int A[1000];
    int B[1000];
    const int C[1000];
    C = A;
    C = A + B;
}

int main(){
    foo8();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_8.c P3_test_8
/home/clang9/PR003/test/P3_test_8.c:7:7: error: read-only variable is not assignable
    C = A;
    ~ ^
/home/clang9/PR003/test/P3_test_8.c:8:7: error: read-only variable is not assignable
    C = A + B;
    ~ ^
2 errors generated.
[testing] /home/clang9/PR003/test/P3_test_8.c
[generating] /home/clang9/PR003/bin/P3_test_8
```

- 实验9：

  测试程序：P3_test_9.c

```
#include <stdio.h>

#pragma elementWise
void foo9(){
    int A[1000];
    const int B[1000];
    int C[1000];
    for(int i = 0; i < 1000; i++){
        A[i] = i;
        B[i] = 2 * i;
    }
    C = B;
    printf("%d\n", C[1]);
    C = A + B;
    printf("%d\n", C[1]);
}

int main(){
    foo9();
}
```

实验结果：

```
[clang9@host2 ~]$ sh ~/PR003/scripts/compile_and_check.sh ~/PR003/test/P3_test_9.c P3_test_9
/home/clang9/PR003/test/P3_test_9.c:10:14: error: read-only variable is not assignable
        B[i] = 2 * i;
        ~~~~~ ^
1 error generated.
[testing] /home/clang9/PR003/test/P3_test_9.c
[generating] /home/clang9/PR003/bin/P3_test_9
```