[TOC]

编译原理研讨课实验PR003实验报告

任务说明

任务要求

1. 支持'+'和'*'两种操作
2. 支持'='操作
3. 支持C语言标准的int类型
4. 操作数应为静态大小的数组
5. 操作数类型匹配：大小必须相同
6. 编译器可以直接编译符合规范的源代码文件生成二进制文件并正确执行

任务实现

通过修改llvm-3-3/tool/clang/lib/CodeGen/CGExpr.cpp中的CodeGenFunction::EmitAnyExpr函数，使用得到的AST的信息构建IR。

成员组成

蔡昕、段江飞、资威

实验设计

设计思路

处理每个表达式的时候都会调用EmitAnyExpr函数，我们就在这个函数上面作文章。设计思路就是在处理表达式的时候看看表达式对应的AST是不是我们上个实验写出来的样子，然后再对应判断是 '+' , '*' , '=' 中的哪一个，生成对应的IR表示。

实验实现

```
//deal with #elementWise operation(+/*/=)
    //首先我们要判断是不是我们应该支持的静态数组变量
    if(E->getType()->getTypeClass() == Type::ConstantArray)
    {
      //需要支持的三种情形顶层表达式都是BinaryOperator的'='
      if(BinaryOperator::classof(E))
      {
          //使用整数类型
          QualType Ty = getContext().IntTy;
          llvm::Type *LTy = ConvertTypeForMem(Ty);
          //分配临时变量，i是数组的偏移，设置4字节对齐
          llvm::AllocaInst *Alloca = CreateTempAlloca(LTy);
          Alloca->setName("i");
          Alloca->setAlignment(4);
          //使用Store初始化临时变量
```

```
            llvm::StoreInst *Store = Builder.CreateStore(llvm::ConstantInt::get(LTy,
llvm::APInt(32,0)), (llvm::Value*)Alloca, false);
            Store->setAlignment(4);
            //需要确认顶层运算符是赋值运算
            const BinaryOperator *bo = dyn_cast<BinaryOperator>(E);
            assert(bo->getOpcode() == BO_Assign);
            //取出左运算数和右运算数
            Expr *lhs = bo->getLHS();
            Expr *rhs = bo->getRHS();
            //需要左运算数类型也是静态数组，并且不是一个表达式
            assert(lhs->getType()->getTypeClass() == Type::ConstantArray);
            assert(DeclRefExpr::classof(lhs));
            //得到左运算数对应数组的大小
            const Type* lhs_T = lhs->getType().getTypePtr();
            const ConstantArrayType *lhs_type = dyn_cast<ConstantArrayType>(lhs_T);
            const llvm::APInt lhs_size = lhs_type->getSize();
            //llvm的每个指令都会返回一个Value、StoreInst、AllocInst
            //LoadInst都属于Value
            llvm::Value *baseA, *addrA ,*baseB, *addrB, *baseC, *addrC;
            //我们实现成for循环，下面是for循环中所需要的BasicBlock
            llvm::BasicBlock *ForCond = createBasicBlock("for.cond");
            llvm::BasicBlock *ForBody = createBasicBlock("for.body");
            llvm::BasicBlock *ForInc = createBasicBlock("for.inc");
            llvm::BasicBlock *ForEnd = createBasicBlock("for.end");
            //开始for循环中的条件判断部分
            EmitBlock(ForCond);
            //重新加载i作为数组访问的偏移
            llvm::LoadInst *index = Builder.CreateLoad((llvm::Value*)Alloca, "");
            index->setAlignment(4);
            //类型提升
            llvm::Value *index_promoted = Builder.CreateIntCast(index, IntPtrTy,
false, "index");
            //生成比较的逻辑
            QualType cmp_Ty = getContext().UnsignedLongTy;
            llvm::Type *cmp_LTy = ConvertType(cmp_Ty);
            llvm::Value *cmp = Builder.CreateICmpSLT(index_promoted,
llvm::ConstantInt::get(cmp_LTy, lhs_size));
            //如果条件满足就跳转到ForBody，不满足就跳转到ForEnd
            Builder.CreateCondBr(cmp, ForBody, ForEnd);
            //开始进入ForBody
            EmitBlock(ForBody);
            //如果右边是一个二元操作符表达式的话，即'+' , '*'这两种情况
            if(BinaryOperator::classof(rhs))
            {
              //把右边表达式的左右操作数拿出来
              const BinaryOperator *bo1 = dyn_cast<BinaryOperator>(rhs);
              Expr *lhs1 = bo1->getLHS();
              Expr *rhs1 = bo1->getRHS();
              //左右操作数都必须要是静态数组类型
              assert(lhs1->getType()->getTypeClass() == Type::ConstantArray);
              assert(rhs1->getType()->getTypeClass() == Type::ConstantArray);
              //只支持'+' , '*'两种操作
              if(bo1->getOpcode() == BO_Add || bo1->getOpcode() == BO_Mul)
              {
```

```
                //针对C[i]
                const DeclRefExpr *declRef =  dyn_cast<DeclRefExpr>(lhs);
                //拿到C对应的Decl
                const ValueDecl* decl = declRef->getDecl();
                //找到C对应的llvm value
                baseC = LocalDeclMap.lookup((Decl*)decl);
                //首先需要左右两个操作数(A、B)的类型正确，然后把他们转化成const的
                assert(ImplicitCastExpr::classof(lhs1));
                assert(ImplicitCastExpr::classof(rhs1));
                const ImplicitCastExpr* lhs2 = dyn_cast<ImplicitCastExpr>(lhs1);
                const ImplicitCastExpr* rhs2 = dyn_cast<ImplicitCastExpr>(rhs1);
                //找到A对应的llvm value
                const DeclRefExpr *declRefR1 = dyn_cast<DeclRefExpr>(lhs2-
>getSubExpr());
                const ValueDecl* declr1 = declRefR1 ->getDecl();
                baseA = LocalDeclMap.lookup((Decl*)declr1);
                //找到B对应的llvm value
                const DeclRefExpr *declRefR2 = dyn_cast<DeclRefExpr>(rhs2-
>getSubExpr());
                const ValueDecl* declr2 = declRefR2->getDecl();
                baseB = LocalDeclMap.lookup((Decl*)declr2);
                //获得对其信息
                CharUnits Alignment = getContext().getDeclAlign(declr2);
                //获得类型信息
                QualType T = declRefR2->getType();
                //定义三个左值
                LValue LVA, LVB, LVC;
                //获得A、B、C的指针
                LVA = MakeAddrLValue(baseA, T, Alignment);
                LVB = MakeAddrLValue(baseB, T, Alignment);
                LVC = MakeAddrLValue(baseC, T, Alignment);

                llvm::Value *arrayPtrA = LVA.getAddress();
                llvm::Value *arrayPtrB = LVB.getAddress();
                llvm::Value *arrayPtrC = LVC.getAddress();
                //边界检查参数
                llvm::Value *Zero = llvm::ConstantInt::get(Int32Ty, 0);

                llvm::Value *args[] = {Zero, index_promoted};
                //获得元素的指针
                addrA = Builder.CreateInBoundsGEP(arrayPtrA, args, "arrayindex");
                addrB = Builder.CreateInBoundsGEP(arrayPtrB, args, "arrayindex");
                addrC = Builder.CreateInBoundsGEP(arrayPtrC, args, "arrayindex");
                //读A、B数组中对应的元素
                llvm::LoadInst *valueA = Builder.CreateLoad(addrA, "");
                llvm::LoadInst *valueB = Builder.CreateLoad(addrB, "");
                valueA->setAlignment(4);
                valueB->setAlignment(4);
                //处理加法和乘法不同的部分
                if(bo1->getOpcode() == BO_Add)
                {
                  //执行加法并且保存值到C中
                  llvm::Value *add = Builder.CreateAdd((llvm::Value *)valueA,
(llvm::Value *)valueB, "add");
```

```
                llvm::StoreInst *valueC = Builder.CreateStore(add, addrC, false);
                valueC->setAlignment(4);
              }
              else
              {
                //执行乘法并且保存值到C中
                llvm::Value *add = Builder.CreateMul((llvm::Value *)valueA,
(llvm::Value *)valueB, "mul");
                llvm::StoreInst *valueC = Builder.CreateStore(add, addrC, false);
                valueC->setAlignment(4);
              }
            }
          }
          else
          {
            //对直接赋值情况的处理
            //还是需要右操作数是静态数组
            assert(rhs->getType()->getTypeClass() == Type::ConstantArray);
            assert(ImplicitCastExpr::classof(rhs));
            //获得左操作数的llvm value
            const DeclRefExpr *declRef = dyn_cast<DeclRefExpr>(lhs);
            const ValueDecl* decl = declRef->getDecl();
            baseC = LocalDeclMap.lookup((Decl*)decl);
            //获得右操作数的llvm value
            const ImplicitCastExpr* rhs1 = dyn_cast<ImplicitCastExpr>(rhs);
            const DeclRefExpr *declRefr = dyn_cast<DeclRefExpr>(rhs1-
>getSubExpr());
            const ValueDecl* declr = declRefr->getDecl();
            baseA = LocalDeclMap.lookup((Decl*)declr);
            //获得对齐信息和类型信息，定义两个左值
            CharUnits Alignment = getContext().getDeclAlign(decl);
            QualType T = declRef->getType();
            LValue LVA, LVC;
            //得到数组地址
            LVA = MakeAddrLValue(baseA, T, Alignment);
            LVC = MakeAddrLValue(baseC, T, Alignment);
            llvm::Value *arrayPtrA = LVA.getAddress();
            llvm::Value *arrayPtrC = LVC.getAddress();
            //边界检查参数
            llvm::Value *Zero = llvm::ConstantInt::get(Int32Ty, 0);

            llvm::Value *args[] = {Zero, index_promoted};
            //获得数组元素的地址
            addrA = Builder.CreateInBoundsGEP(arrayPtrA, args, "arrayindex");
            addrC = Builder.CreateInBoundsGEP(arrayPtrC, args, "arrayindex");
            //把右操作数的值赋给左操作数
            llvm::LoadInst *valueA = Builder.CreateLoad(addrA, "");
            valueA->setAlignment(4);
            llvm::StoreInst *valueC = Builder.CreateStore((llvm::Value *)valueA,
addrC, false);
            valueC->setAlignment(4);
          }
          //for循环的递增部分，对于数组的偏移需要递增
          EmitBlock(ForInc);
```

```
            llvm::Value *inc_index = Builder.CreateAdd((llvm::Value*)index,
(llvm::Value*)llvm::ConstantInt::get(LTy,llvm::APInt(32,1)), "add");
            llvm::StoreInst * inc_store = Builder.CreateStore(inc_index,
(llvm::Value*)Alloca, false);
            inc_store->setAlignment(4);
            //无条件跳转到for循环的条件判断处
            Builder.CreateBr(ForCond);
            //退出for循环，随便返回一个值
            EmitBlock(ForEnd);
            return RValue::get(addrA);
            //llvm::LoadInst *idx = B
        }
      }
```

其它

在这里贴一下测试文件和测试结果：
正确性测试，即我们期望能够正确运行的文件：Ttest3.c

```
#include <stdio.h>
  //老师给的测试样例
  #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]);
  }
  //老师给的测试样例
  #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]);
  }
```

```
//自己做的测试样例，测试了三种方式，并且逐一比较
#pragma elementWise
void fooh1(){
  int A[1000];
  int B[1000];
  int C[1000];
    int D[1000];
  for(int i = 0; i < 1000; i++){
    A[i] = i;
    B[i] = 2*i;
  }
  C = A;
    for(int i = 0; i < 1000; i++){
        D[i] = A[i];
    }
    for(int i = 0; i < 1000; i++){
        if (D[i] != C[i]){
            printf("ERROR!\n");
            return;
        }
    }
  C = B;
    for(int i = 0; i < 1000; i++){
        D[i] = B[i];
    }
    for(int i = 0; i < 1000; i++){
        if (D[i] != C[i]){
            printf("ERROR!\n");
            return;
        }
    }
    C = A + B;
    for(int i = 0; i < 1000; i++){
        D[i] = A[i] + B[i];
    }
    for(int i = 0; i < 1000; i++){
        if (D[i] != C[i]){
            printf("ERROR!\n");
            return;
        }
    }
    C = A * B;
    for(int i = 0; i < 1000; i++){
        D[i] = A[i] * B[i];
    }
    for(int i = 0; i < 1000; i++){
        if (D[i] != C[i]){
            printf("ERROR!\n");
            return;
        }
    }
    printf("PASS!!\n");
    return;
}
```

```
//这个测试和上一个比起来只是把数组B换成了const
#pragma elementWise
void add(){
  int A[10];
  const int B[10] = {1,2,3,4,5,6,7,8,9,10};
  int C[10];
  int D[10];
  for(int i = 0; i < 10; i++){
    A[i] = 2 * i;
  }
  C = A;
  for(int i = 0; i < 10; i++){
      D[i] = A[i];
   }
 for(int i = 0; i < 10; i++){
   if (D[i] != C[i]){
     printf("ERROR!\n");
     return;
   }
 }
 C = B;
 for(int i = 0; i < 10; i++){
     D[i] = B[i];
  }
 for(int i = 0; i < 10; i++){
   if (D[i] != C[i]){
     printf("ERROR!\n");
     return;
   }
 }
 C = A + B;
 for(int i = 0; i < 10; i++){
     D[i] = A[i] + B[i];
  }
 for(int i = 0; i < 10; i++){
   if (D[i] != C[i]){
     printf("ERROR!\n");
     return;
   }
 }
 C = A * B;
 for(int i = 0; i < 10; i++){
     D[i] = A[i] * B[i];
  }
 for(int i = 0; i < 10; i++){
   if (D[i] != C[i]){
     printf("ERROR!\n");
     return;
   }
 }
 printf("add check pass\n");
}

int main(){
```

```
        foo1();
        foo2();
        fooh1();
        add();
        return 0;
    }
```

```
[clang7@host2 test]$ ./Ttest3
2
1
1
2
PASS!!
```

然后是测试报错能力的测试文件:Ftest3.c 只展示输出结果：

```
[clang7@host2 test]$ clang Ftest3.c
Ftest3.c:18:11: error: expression is not assignable
  (A + B) = C;
  ~~~~~~~ ^
Ftest3.c:27:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A + D;
      ~ ^ ~
Ftest3.c:28:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = D + A;
      ~ ^ ~
Ftest3.c:29:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = D + D;
      ~ ^ ~
Ftest3.c:38:11: error: expression is not assignable
  (A + B) = C;
  ~~~~~~~ ^
Ftest3.c:47:5: error: array type 'int [10][100]' is not assignable
  E = A;
  ~ ^
Ftest3.c:48:9: error: invalid operands to binary expression ('int *' and 'int *')
  E = A + B;
      ~ ^ ~
Ftest3.c:49:9: error: invalid operands to binary expression ('int *' and 'int *')
  E = A * B;
      ~ ^ ~
Ftest3.c:57:5: error: read-only variable is not assignable
  C = A;
  ~ ^
Ftest3.c:58:5: error: read-only variable is not assignable
  C = A + B;
  ~ ^
```

```
Ftest3.c:65:5: error: array type 'int [1000]' is not assignable
  C = A;
  ~ ^
Ftest3.c:66:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A + B;
      ~ ^ ~
Ftest3.c:67:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A * B;
      ~ ^ ~
Ftest3.c:75:5: error: assigning to 'char [1000]' from incompatible type 'char [1000]'
  C = A;
    ^ ~
Ftest3.c:76:9: error: invalid operands to binary expression ('char *' and 'char *')
  C = A + B;
      ~ ^ ~
Ftest3.c:77:9: error: invalid operands to binary expression ('char *' and 'char *')
  C = A * B;
      ~ ^ ~
Ftest3.c:85:5: error: assigning to 'float [1000]' from incompatible type 'float [1000]'
  C = A;
    ^ ~
Ftest3.c:86:9: error: invalid operands to binary expression ('float *' and 'float *')
  C = A + B;
      ~ ^ ~
Ftest3.c:87:9: error: invalid operands to binary expression ('float *' and 'float *')
  C = A * B;
      ~ ^ ~
```

```
[clang/@host2 test]$ clang Ftest3.c
Ftest3.c:94:5: error: assigning to 'short [1000]' from incompatible type 'short [1000]'
  C = A;
    ^ ~
Ftest3.c:95:9: error: invalid operands to binary expression ('short *' and 'short *')
  C = A + B;
      ~ ^ ~
Ftest3.c:96:9: error: invalid operands to binary expression ('short *' and 'short *')
  C = A * B;
      ~ ^ ~
Ftest3.c:107:9: error: invalid operands to binary expression ('int *' and 'int *')
  C = A / B;
      ~ ^ ~
Ftest3.c:115:5: error: assigning to 'char [999]' from incompatible type 'char [1000]'
  C = A;
    ^ ~
Ftest3.c:116:9: error: invalid operands to binary expression ('char *' and 'char *')
  C = A + B;
      ~ ^ ~
Ftest3.c:117:9: error: invalid operands to binary expression ('char *' and 'char *')
  C = A * B;
      ~ ^ ~
Ftest3.c:125:5: error: assigning to 'char [1000]' from incompatible type 'char [1000]'
  C = A;
    ^ ~
Ftest3.c:126:9: error: invalid operands to binary expression ('char *' and 'char *')
  C = A + B;
      ~ ^ ~
Ftest3.c:127:9: error: invalid operands to binary expression ('char *' and 'char *')
  C = A * B;
      ~ ^ ~
```

总结

实验结果总结

我们通过了老师给的所有测试，并且还测试了const数组不能被赋值的情况，测试了除了int类型不能支持的情况和除法不能支持的情况，并且通过了所有的测试。

分成员总结

蔡昕：负责写出思路正确的代码。
段江飞：负责debug和最后提交的正确性。
资威：负责测试和撰写实验报告。