编译原理研讨课实验PR003实验报告

任务说明

本实验PR003在PR002的基础上，要求对具有相同int类型静态数组的'+','*','='三种操作生成正确的中间表示（LLVM IR)，并生成正确的可执行文件。

具体要求：生成合法AST，不破坏原有语义；支持'+'和'*'两种操作；支持'='操作；支持C语言标准的int类型；操作数应为静态大小的数组；操作数类型匹配：大小必须相同；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；对于非法代码在编译器报错。

成员组成

骆嵩涛 2018K8009907001

毛啸腾 2018K8009907033

陈彦帆 2018K8009918002

实验设计

设计思路

首先，在生成AST表示后，elementWise的+,*,=操作都具有constantarray类型，在llvm中归为Aggregate（聚集）类型。对相应的表达式进行代码生成时，由EmitAggExpr处理。

对于=操作（形如L=R)，EmitAggExpr会调用AggExprEmitter::VisitBinAssign方法，直接将R的值复制到L上，并返回L的值。不需要额外操作即可实现此功能。

对于+,*操作（形如LHS+RHS)，EmitAggExpr会调用AggExprEmitter::VisitBinaryOperator方法，该方法不支持左、右操作数为constantarray类型，将会报错。为此，只需要在此方法中增加对constantarray的支持即可。

在EmitAggExpr中，每个表达式的返回值被存在AggValueSlot中，设其对应的地址为DHS。对于+,*操作，我们将生成形如do { *DHS++ = *LHS++ op *RHS++; } while (LHS != end); 的中间代码。其中，LHS和RHS将递归地生成。因此，我们的设计也支持elementWise的复合表达式。

实验实现

首先，在VisitBinaryOperator中增加对constantarray的类型支持：

```
void AggExprEmitter::VisitBinaryOperator(const BinaryOperator *E) {
  if (E->getOpcode() == BO_PtrMemD || E->getOpcode() == BO_PtrMemI)
    VisitPointerToDataMemberBinaryOperator(E);
  else if(E->getLHS()->getType()->isConstantArrayType() &&
          E->getRHS()->getType()->isConstantArrayType() &&
          (E->getOpcode() == BO_Add || E->getOpcode() == BO_Mul)){
  assert(CGF.getContext().hasSameUnqualifiedType(E->getLHS()->getType(),E-
>getRHS()->getType())&& "Invalid assignment");
      ...
  }
    else
    CGF.ErrorUnsupported(E, "aggregate binary expression");
}
```

递归地处理LHS和RHS，通过AggValueSlot获得其返回值

```
llvm::Value *RHS, *LHS, *DHS;
// handle LHS and RHS recursively
RHS = CGF.EmitAnyExpr(E->getRHS()).getAggregateAddr();
LHS = CGF.EmitAnyExpr(E->getLHS()).getAggregateAddr();
DHS = Dest.getAddr();
```

获得数组长度：由于我们将生成一个do-while循环，故需要判断数组长度是否为0，若为0直接结束。

```
// get NumElements
llvm::PointerType *APType = cast<llvm::PointerType>(LHS->getType());
llvm::ArrayType *AType = cast<llvm::ArrayType>(APType->getElementType());
uint64_t numElements = AType->getNumElements();
if (!numElements) return;
```

仿照AggExprEmitter::EmitArrayInit对数组初始化的处理，生成一个do-while循环，形如：

do { *DHS++ = *LHS++ op *RHS++; } while (LHS != end);

首先，获取首元素的基地址：

```
// LHS is an array*. Construct an elementType* by drilling down a level.
llvm::Value *zero = llvm::ConstantInt::get(CGF.SizeTy, 0);
llvm::Value *indices[] = {zero, zero};
llvm::Value *begin = Builder.CreateInBoundsGEP(DHS,
indices,"elementWise.begin");
llvm::Value *begin1 = Builder.CreateInBoundsGEP(LHS,
indices,"elementWise.begin1");
llvm::Value *begin2 = Builder.CreateInBoundsGEP(RHS,
indices,"elementWise.begin2");
```

生成基本块entryBB,bodyBB和endBB。

在body中，通过一个PHINode维护当前指针的index，获取该index下LHS，RHS和DHS的地址，加载LHS和RHS的值，相加或相乘后存至DHS的地址中。随后，index增加一个单位。比较index是否到达数组结尾，若是，则结束循环。

```
llvm::BasicBlock *entryBB = Builder.GetInsertBlock();
llvm::BasicBlock *bodyBB = CGF.createBasicBlock("elementWise.body");

// Jump into the body.
CGF.EmitBlock(bodyBB);
llvm::PHINode *index =
    Builder.CreatePHI(zero->getType(), 2, "elementWise.index");
index->addIncoming(zero, entryBB);
// count element addr
llvm::Value *element = Builder.CreateInBoundsGEP(begin, index);
llvm::Value *element1 = Builder.CreateInBoundsGEP(begin1, index);
llvm::Value *element2 = Builder.CreateInBoundsGEP(begin2, index);
// load element val
llvm::Value *val1 = Builder.CreateLoad(element1,"l1");
llvm::Value *val2 = Builder.CreateLoad(element2,"l2");
// res = val1 op val2
llvm::Value *res;
if(E->getOpcode() == BO_Add)
  res = Builder.CreateAdd(val1,val2,"add");
else
  res = Builder.CreateMul(val1,val2,"mul");
// store res to aggslot
Builder.CreateStore(res,element);

// Move on to the next element.
llvm::Value *nextIndex = Builder.CreateNUWAdd(
  index, llvm::ConstantInt::get(CGF.SizeTy, 1), "elementWise.next");
index->addIncoming(nextIndex, Builder.GetInsertBlock());

// Leave the loop if we're done.
llvm::Value *done = Builder.CreateICmpEQ(
    nextIndex, llvm::ConstantInt::get(CGF.SizeTy, numElements),
    "elementWise.done");
llvm::BasicBlock *endBB = CGF.createBasicBlock("elementWise.end");
Builder.CreateCondBr(done, endBB, bodyBB);
CGF.EmitBlock(endBB);
```

其它

在C99 6.3.2.1p2中，一个出现在表达式中的数组将首先被隐式地转换为指向数组第一个元素的指针。由于我们支持了elementWise类型，在符合elementWise语法的表达式中，这样的转换不会进行，而是进行elementWise操作。在不符合elementWise语法的表达式中，我们仍需支持这样的约定。

比如A和B都是int[10]类型，在elementWise制导下，我们需要支持int* p = A+B;

上式中，A+B的结果是一个右值，由于等号左边是一个指针，A+B将被转换为指向该右值第一个元素的指针。

为此，在CodeGenFunction::EmitBinaryOperatorLValue中，我们增加对这种情况的考虑，从而将该右值数组转换为一个左值，获取其地址：

```
assert((E->getOpcode() == BO_Assign || getEvaluationKind(E-
>getType())==TEK_Aggregate)
         && "unexpected binary l-value");
...
 case TEK_Aggregate:
    return EmitAggExprToLValue(E);
```

这样，对于形如int* p = A+B;的表达式，经过EmitArrayToPointerDecay -> EmitLValue -> EmitBinaryOperatorLValue -> EmitAggExprToLValue，我们能够处理上述情况。

总结

实验结果总结

查看生成的llvm IR

用于测试的函数：

```
#pragma elementWise
void f()
{
  int a[10];
  int b[10];
  int c[10];
  a = b+c;
}
```

生成llvm IR结果

```
define void @f() #0 {
entry:
  %a = alloca [10 x i32], align 16
  %b = alloca [10 x i32], align 16
  %c = alloca [10 x i32], align 16
  %agg-temp = alloca [10 x i32], align 4
  %agg-temp1 = alloca [10 x i32], align 4
  %0 = bitcast [10 x i32]* %agg-temp to i8*
  %1 = bitcast [10 x i32]* %c to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %0, i8* %1, i64 40, i32 4, i1 false)
  %2 = bitcast [10 x i32]* %agg-temp1 to i8*
  %3 = bitcast [10 x i32]* %b to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %2, i8* %3, i64 40, i32 4, i1 false)
  %elementWise.begin = getelementptr inbounds [10 x i32]* %a, i64 0, i64 0
  %elementWise.begin1 = getelementptr inbounds [10 x i32]* %agg-temp1, i64 0,
i64 0
  %elementWise.begin2 = getelementptr inbounds [10 x i32]* %agg-temp, i64 0, i64
0
  br label %elementWise.body

elementWise.body:                                 ; preds = %elementWise.body,
%entry
  %elementWise.index = phi i64 [ 0, %entry ], [ %elementWise.next,
%elementWise.body ]
  %4 = getelementptr inbounds i32* %elementWise.begin, i64 %elementWise.index
  %5 = getelementptr inbounds i32* %elementWise.begin1, i64 %elementWise.index
```

```
  %6 = getelementptr inbounds i32* %elementWise.begin2, i64 %elementWise.index
  %l1 = load i32* %5
  %l2 = load i32* %6
  %add = add i32 %l1, %l2
  store i32 %add, i32* %4
  %elementWise.next = add nuw i64 %elementWise.index, 1
  %elementWise.done = icmp eq i64 %elementWise.next, 10
  br i1 %elementWise.done, label %elementWise.end, label %elementWise.body

elementWise.end:                                  ; preds = %elementWise.body
  ret void
}
```

测试可执行文件

用于测试的代码：

```
#include <stdio.h>
#pragma elementWise
typedef int s32;
int c[100] = {4,5,6};
void f()
{
  s32 a[100] = {-1,2};
  static int b[100] = {2,3};
  int e[100];
  a = (b = e = a+c)*a;
  int *d = (a+b*b);
  printf("b: %d %d %d\n",b[0],b[1],b[2]);
  printf("a: %d %d %d\n",a[0],a[1],a[2]);
  printf("d: %d %d %d\n",d[0],d[1],d[2]);
}
#pragma elementWise
void g()
{
  for(int i=0;i<sizeof(c)/sizeof(c[0]);i++)
    c[i] = 2;
  c = c*(c+c);
  int sum;
  for(int i=0;i<sizeof(c)/sizeof(c[0]);i++)
    sum += c[i];
  printf("sun: %d\n",sum);
}

int main()
{
  f();
  g();
  return 0;
}
```

输出结果：

```
b: 3 7 6
a: -3 14 0
d: 6 63 36
sum: 800
```

结果与预期相符。

分成员总结

陈彦帆

完成本次代码。代码的主要部分参考llvm中AggExprEmitter::EmitArrayInit对数组初始化的处理。用gdb跟踪代码生成的执行过程，我发现llvm的代码生成是一个递归下降的过程，在本实验插入的代码也采用递归下降的方法来处理。

毛啸腾

复现代码后，对实验报告进行了完善。本次实验代码量不大，关键是在理解生成LLVM IR代码的过程的基础上，仿照源码中已有部分对于ElementWise的功能实现扩展。对于源码中的处理，我们理解执行过程，会使用已有函数即可，对于源码中个别代码的实现细节其实无需深究。

骆嵩涛

参与讨论，对于代码进行了测试。本次实验的重点是结合理论课，理解通过基本块处理的方式。