编译原理研讨课实验 PR003 实验报告

一、任务说明

本次任务负责完善 elementWise 从 AST 到 LLVM IR 的代码生成的通路。

二、成员组成

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

三、实验设计

在 EmitAnyExpr 函数中新增分支，跳向自定义的 EmitVectorExpr()函数，这个函数负责给 elementWise 类型的元素生成对应 LLVM IR 代码。

四、设计思路

将 elementWise 的部分转为对应的 for 循环来实现其功能。
例如将 a = b 转化为：
for (int i = 0; i < n ; i++) a[i] = b[i]

可以参考我们编译原理作业中为 for 循环生成中间代码的实现：

为 for 语句定义类 *For* 如下：

```
class For extends Stmt{
    Expr E1, E2, E3; Stmt S;
    public For(Expr x1, Expr x2, Expr x3, Stmt y){
        E1=x1; E2=x2; E3=x3; S=y;
        label1=newlabel(); label2=newlabel();
    }
    //以下假设 Expr 也实现了 gen()
    public void gen(){
        E1.gen();
        emit(label1+":");
        Expr n=E2.rvalue();
        emit("ifFalse "+n.toString()+" goto "+label2);
        S.gen();
        E3.gen();
        emit("goto "+label1);
        emit(label2+":");
    }
}
```

对应过来，E1 为 i＝0，E2 为 i＜n，E3 为 i++，S 为 a[i]＝b[i]。这样我们根据上面的代码生成片段，可以类似地完成本次实验的编写。

五、实验实现

首先找到入口函数

From /home/clang7/llvm/tools/clang/lib/CodeGen/CGExpr.cpp

```
RValue CodeGenFunction::EmitAnyExpr(const Expr *E,
                                    AggValueSlot aggSlot,
                                    bool ignoreResult) {
```

这个函数的作用是产生代码去计算具有任何类型的特定的表达式，并返回一个右值结构。在这个函数最开始处，我们加入：

```
if (E->getType()->getTypeClass() == Type::ConstantArray)
{
  return RValue::get(EmitVectorExpr(E));
}//ADD for PR003
```

这个 if 分支专门用于应对形如 a = b 以及 a + b 这类数组运算的表达式，即 elementWise 的情况。

下面来看我们的重点，即 EmitVectorExpr 函数：

```
llvm::Value *CodeGenFunction::EmitVectorExpr(const Expr *E)
```

该函数负责为 VectorExpr 产生代码。

首先，表达式如果是二元运算符形式：

```
if (BinaryOperator::classof(E)) {
  // Get/Emit LHS & RHS
  const BinaryOperator* bo = dyn_cast<BinaryOperator>(E);
  Expr* LHS = bo->getLHS();
  Expr* RHS = bo->getRHS();
  llvm::Value *LHSBase, *RHSBase;
  if (bo->getOpcode() == BO_Assign) {
    const DeclRefExpr *declRef = dyn_cast<DeclRefExpr>(LHS);
    const ValueDecl* decl = declRef->getDecl();
    LHSBase = LocalDeclMap.lookup((Decl*)decl);
  }
  else {
    LHSBase = EmitVectorExpr(LHS);
  }
  RHSBase = EmitVectorExpr(RHS);
```

此时分两种情况，一种是二元运算符是赋值，那么 LHSBase 已经存在，需要去查找；否则，递归生成 LHSBase。RHSBase 始终采用递归生成。

我们用 for 循环的形式来进行 elementWise 下的运算：

首先我们分配循环变量。注意变量以 4 字节对齐。

```
// 分配循环变量
QualType Ty = getContext().UnsignedIntTy;
llvm::Type *LTy = ConvertTypeForMem(Ty);
llvm::AllocaInst *Alloc = CreateTempAlloca(LTy);
Alloc->setName("compiler.for.i");
Alloc->setAlignment(4);
```

分配过程按照 IR 的语言格式。

分配加法和乘法操作时所需要的临时数组。需要说明的是，如果运算符是赋值运算，临时数组还是需要被分配，虽然后续不会用到。

```
// 分配临时变量
// In fact for assignment operations, the temp array is unused
QualType TyR = E->getType().getUnqualifiedType();
llvm::Type *LTyR = ConvertTypeForMem(TyR);
llvm::AllocaInst *AllocR;
```

初始化循环变量。变量的初始化仿照老师给出的框架进行。

```
// 初始化循环变量
llvm::StoreInst *InitI = Builder.CreateStore(llvm::ConstantInt::get(LTy, llvm::APInt(32, 0)), (llvm::Value*)Alloc, false);
InitI->setAlignment(4);
```

之后我们创建 for 循环所需要的三个基本块：

```
// 创建基本块
llvm::BasicBlock *CondBlock = createBasicBlock("compiler.for.cond");
llvm::BasicBlock *BodyBlock = createBasicBlock("compiler.for.body");
llvm::BasicBlock *ExitBlock = createBasicBlock("compiler.for.exit");
```

cond 部分：

```
EmitBlock(CondBlock);
// 重新加载compiler作为数组访问的偏移
// LLVM的每个指令都会返回一个Value，StoreInst，AllocInst
// LoadInst都属于Value
llvm::LoadInst *CondLoadI = Builder.CreateLoad((llvm::Value*)Alloc, "");
CondLoadI->setAlignment(4);
//类型提升
llvm::Value *idx = Builder.CreateIntCast(CondLoadI, IntPtrTy, false, "idxprom");

const ConstantArrayType *CATy = dyn_cast<ConstantArrayType>(TyR);
llvm::Value *CondCmpRHS = llvm::ConstantInt::get(IntPtrTy, CATy->getSize());

// Create unsigned less-than comparison
llvm::Value *CondCmp = Builder.CreateICmpULT(idx, CondCmpRHS, "compiler.for.cond.cmp");
Builder.CreateCondBr(CondCmp, BodyBlock, ExitBlock);
```

首先我们使用一条 Load 将循环变量读进来，并将该变量类型提升为 Int，得到在 Int 类型下的值；接着我们获取数组的 size 所对应的值，用这两个值做 Compare，并将 Compare 的结果作为操作数，生成一条条件跳转语句，分别跳转到 body 和 exit 对应的标号。

body 部分：

```
EmitBlock(BodyBlock);

// 获取指针
llvm::Value *LHSArrayPtr = MakeAddrLValue(LHSBase, TyR).getAddress();
llvm::Value *RHSArrayPtr = MakeAddrLValue(RHSBase, TyR).getAddress();

// 边界检查
llvm::Value *Zero = llvm::ConstantInt::get(Int32Ty, 0);
llvm::Value *Args[] = { Zero, idx };
llvm::Value *LHSAddr = Builder.CreateInBoundsGEP(LHSArrayPtr, Args, "arrayidx");
llvm::Value *RHSAddr = Builder.CreateInBoundsGEP(RHSArrayPtr, Args, "arrayidx");
if (bo->getOpcode() == BO_Assign) {
  llvm::LoadInst *load = Builder.CreateLoad(RHSAddr, "");
  load->setAlignment(4);
  llvm::StoreInst *store = Builder.CreateStore((llvm::Value*)load, LHSAddr, false);
  store->setAlignment(4);
}
```

```
else {
  llvm::LoadInst *valueL = Builder.CreateLoad(LHSAddr, "");
  valueL->setAlignment(4);
  llvm::LoadInst *valueR = Builder.CreateLoad(RHSAddr, "");
  valueR->setAlignment(4);
  AllocR = CreateTempAlloca(LTyR);
  AllocR->setName("compiler.temp_result");
  AllocR->setAlignment(4);
  llvm::Value *AllocRBase = (llvm::Value *)AllocR;
  llvm::Value *AllocRArrayPtr = MakeAddrLValue(AllocRBase, TyR).getAddress();
  llvm::Value *AllocRAddr = Builder.CreateInBoundsGEP(AllocRArrayPtr, Args, "arrayidx");
  llvm::Value *res;
  if(bo->getOpcode() == BO_Add)
    res = Builder.CreateAdd((llvm::Value *)valueL, (llvm::Value *)valueR, "add");
  if(bo->getOpcode() == BO_Mul)
    res = Builder.CreateMul((llvm::Value *)valueL, (llvm::Value *)valueR, "mul");
  llvm::StoreInst *storeAllocR = Builder.CreateStore(res, (llvm::Value *)AllocRAddr, false);
  storeAllocR->setAlignment(4);
}
```

这里我们分别得到两个数组第 i 个元素的地址（假设 i 为循环变量的值）。如果是赋值，则直接用 Load 从右边地址位置读取元素到 register 中，再 Store 到左边地址位置中，由此实现值从右到左的传递；如果是加法或者乘法，则进行相应的运算并 Store 结果至临时空间中相应的位置，等到递归分析到赋值号时，将临时空间中的值传递到赋值号左端。

下面这一片段对应了循环变量值+1 和跳转。

```
// Increment & BR
llvm::Value *One = llvm::ConstantInt::get(Int32Ty, 1);
llvm::LoadInst *IncLoadI = Builder.CreateLoad((llvm::Value*)Alloc, "");
IncLoadI->setAlignment(4);
llvm::Value* IncAdd = Builder.CreateAdd((llvm::Value*)IncLoadI, One, "");
llvm::StoreInst *IncStoreI = Builder.CreateStore(IncAdd, (llvm::Value*)Alloc, false);
IncStoreI->setAlignment(4);
Builder.CreateBr(CondBlock);
```

最后是 exit 部分：

```
  EmitBlock(ExitBlock);

  if (bo->getOpcode() == BO_Assign) {
    return LHSBase;
  }
  else {
    return (llvm::Value*)AllocR;
  }
```

可以看到，这个部分的 IR 其实只包含一个标号。

如果最初的 if(BinaryOperator::classof(E))不成立，则跳转到下面的 else 部分，这部分实际上是代表常数组的引用（也就是递归生成 RHSBase 时，发现 RHS 只是一个数组表达式，那么执行递归函数的最底层，通过查询 AST 找到数组基址）：

```
else {
  // Otherwise E is a reference to an array
  if (ImplicitCastExpr::classof(E)) {
    const ImplicitCastExpr* Ecast = dyn_cast<ImplicitCastExpr>(E);
    E = Ecast->getSubExpr();
  }
  assert(DeclRefExpr::classof(E));
  const DeclRefExpr *E2 = dyn_cast<DeclRefExpr>(E);
  return LocalDeclMap.lookup((Decl*)E2->getDecl());
}
```

六、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。

七、总结

本次实验需要对 LLVM IR 的语言格式有一定的理解，还要能将 for 循环各部分具体实现。由于编译原理理论课上曾经也进行过中间代码生成的一些练习，因此我们可以将所学知识和代码框架的细节相结合，进而完成本次实验。

八、分成员总结

李昊宸:

本次实验负责实验代码的书写。起初在老师给出的基本框架下进行编写，但对于 for 循环部分的处理总是有些问题。于是经查阅资料，实现了通过建立基本块的方式，单独存放各部分变量的值。在修改主体，将参考资料给出书写基本块的形式嵌入到当前框架下后，可以进行正常的编译，实现了目的要求。主要的处理流程还是相对较为直观的，关键在于对 clang 下处理各种情况的函数仍然不够熟悉，导致过程较为困难。

李颖彦:

本次实验负责实验报告的书写。本次代码看上去代码量不算大，但实际上要接触很多新知识，需要对 AST 转 LLVM IR 的过程及方法有一个细致的理解，也要能对 for 循环进行底层实现。

陆润宇:

本次实验负责实验报告的书写和修改。可以看到，本次实验的代码结构基本与我们理论课上所学一致，而为了能与框架相结合，我们还需要进行例如类型提升这样的额外操作，并注意对齐等细节的实现。