链滴

Kaleidoscope 系列第三章：生成 LLVM 中间代码 IR

作者：Hanseltu
原文链接：https://ld246.com/article/1570000872211
来源网站：链滴

原文链接:Kaleidoscope系列第三章：生成LLVM中间代码IR

本文是使用LLVM开发新语言Kaleidoscope教程系列第三章，主要实现将AST转化为LLVM IR的功能。

第三章简介

欢迎来到“使用LLVM开发新语言Kaleidoscope教程”教程的第三章。本章介绍如何将第二章中构建的象语法树转换为LLVM IR。本章将告诉我们一些有关LLVM如何工作的知识，并演示它的易用性。构词法分析器和解析器要比生成LLVM IR代码多得多。

请注意：本章及更高版本中的代码要求LLVM 3.7或更高版本。LLVM 3.6及更低版本将无法使用。还注意，我们需要使用与您的LLVM版本匹配的本教程版本：如果你使用的是官方LLVM版本，请使用你版本随附的文档版本或llvm.org版本页面上的文档版本。

中间代码生成配置

为了生成LLVM IR，我们希望开始一些简单的配置。首先，我们在每个AST类中定义虚拟代码生成（codegen）方法：

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;

public:
  NumberExprAST(double Val) : Val(Val) {}
  virtual Value *codegen();
};
...
```

codegen() 方法表示要为该AST节点生成中间代码IR及其依赖的所有东西，并且它们都返回 LLVM Value 对象。“Value”是用于表示LLVM中的“静态单一赋值（SSA）寄存器”或“SSA value”的类。SA值最明显的方面是，它们的值是在相关指令执行时计算的，并且直到（如果有）指令重新执行前，都不会获得新值。换句话说，没有办法“change” SSA值。有关更多信息，请阅读“静态单一赋值”一旦掌握了这些概念，理解它们就会非常自然。

请注意，与其将虚拟方法添加到ExprAST类层次结构中，还可以使用 访问者模式或其他方式对此建模再次说明，本教程将不讨论良好的软件工程实践：就我们的目的而言，添加虚拟方法最简单。

我们要做的第二件事是像用于解析器那样的“LogError”方法，该方法将用于报告在代码生成过程发现的错误（例如，使用未声明的参数）：

```
static LLVMContext TheContext;
static IRBuilder<> Builder(TheContext);
static std::unique_ptr<Module> TheModule;
static std::map<std::string, Value *> NamedValues;
```

```
Value *LogErrorV(const char *Str) {
  LogError(Str);
  return nullptr;
}
```

静态变量将在代码生成期间使用。TheContext 是一个不透明的对象，它拥有很多核心的LLVM数据构，例如类型表和常量值表。我们不需要详细了解它，我们只需要一个实例即可传递给需要它的API。

该 Builder 对象是一个帮助程序对象，可轻松生成LLVM指令。IRBuilder类模板的实例跟踪要插入指的当前位置，并具有创建新指令的方法。

TheModule 是包含函数和全局变量的LLVM方法。在许多方面，它是LLVM IR用来包含代码的顶层结。它将拥有我们生成的所有IR的内存，这就是为什么codegen（）方法返回原始 Value* 而不是uniqu_ptr <Value>的原因。

该 NamedValues 映射跟踪当前范围中定义了哪些值，以及它们的LLVM表示形式是什么（换句话说它是代码的符号表）。在这种形式的Kaleidoscope中，唯一可以引用的是功能参数。这样，在为函数体生成代码时，函数参数将位于此映射中。

有了这些基础知识之后，我们就可以开始讨论如何为每个表达式生成代码了。请注意，这假设 Builder 已将设置为将代码生成的配置。现在，我们假设这已经完成，并且仅使用它来生成IR代码。

表达式代码生成

为表达式节点生成LLVM IR代码非常简单：对于我们所有四个表达式节点，少于45行的注释代码就能定。首先，对于数字表达式：

```
Value *NumberExprAST::codegen() {
  return ConstantFP::get(TheContext, APFloat(Val));
}
```

在LLVM IR中，数字常量由 ConstantFP 类表示，该类将数字值保存在APFloat内部（APFloat具有保任意精度的浮点常量的功能）。这段代码基本上只是创建并返回一个ConstantFP。请注意，在LLVM I中，所有常量都唯一并共享。因此，API使用 foo :: get (...) 惯用语代替 new foo (...) 或 foo :: Create ...)。

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    LogErrorV("Unknown variable name");
  return V;
}
```

使用LLVM，对变量的引用也非常简单。在Kaleidoscope的简单版本中，我们假定变量已经在某个位生成并且其值可用。实际上，NamedValues 映射中唯一可以包含的值就是函数参数。此代码只是检以查看指定的名称是否在映射中（如果不在映射中，则引用一个未知变量）并返回其值。在以后的章中，我们将在符号表中添加对循环归纳变量和局部变量的支持。

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;
```

```
  switch (Op) {
  case '+':
    return Builder.CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder.CreateFSub(L, R, "subtmp");
  case '*':
    return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(TheContext),
                                "booltmp");
  default:
    return LogErrorV("invalid binary operator");
  }
}
```

二元运算符开始变得越来越意思。这里的基本思想是我们递归地为表达式的左侧生成代码，然后再为侧生成代码，然后计算二进制表达式的结果。在此代码中，我们对操作码进行了简单的切换以创建正的LLVM指令。

在上面的示例中，LLVM构建器类开始显示其值。IRBuilder知道在何处插入新创建的指令，我们要做就是指定要创建的指令（例如，使用CreateFAdd），要使用的操作数（L以及R此处），并可以选择生成的指令提供名称。

LLVM的一个好处是名称只是一个提示。例如，如果上面的代码发出多个“ addtmp 变量，则LLVM自动为每个变量提供一个递增的唯一数字后缀。指令的本地值名称纯粹是可选的，但是它使读取IR转更加容易。

LLVM指令受到严格的规则约束：例如，一条add指令的Left和Right运算符必须具有相同的类型，并add的结果类型必须与操作数类型匹配。因为Kaleidoscope中的所有值都是双精度的，所以这使得用add，sub和mul的代码非常简单。

另一方面，LLVM指定fcmp指令始终返回 i1 值（一位整数）。问题在于Kaleidoscope希望该值为0.0 1.0。为了获得这些语义，我们将fcmp指令与uitofp指令结合在一起。该指令通过将输入视为无符号，将其输入整数转换为浮点值。相反，如果我们使用sitofp指令，则Kaleidoscope < 运算符将根据入值返回0.0和-1.0。

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }
```

```
  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

使用LLVM，函数调用的代码生成非常简单。上面的代码最初在LLVM模块的符号表中进行功能名称查。回想一下，LLVM模块是包含我们正在JITing的功能的容器。通过为每个函数指定与用户指定的名相同的名称，我们可以使用LLVM符号表为我们解析函数名称。

一旦有了要调用的函数，就可以递归地对要传递的每个参数进行代码生成，并创建LLVM调用指令。注意，默认情况下，LLVM使用本机C调用约定，从而使这些调用也可以调用标准库函数（如“sin”“cos”）而无需付出额外的努力。

这总结了我们到目前为止在Kaleidoscope中使用的四个基本表达式的处理。我们也可以随意进入并添更多内容。例如，通过浏览LLVM语言手册我们会发现其他一些有趣的指令，这些指令确实很容易插到我们的基本框架中。

函数代码生成

原型和函数的代码生成必须处理许多细节，这使得它们的代码不如表达式代码生成简单，但可以让我举例说明一些重点。首先，我们首先看原型的代码生成：它们既用于函数体，又用于外部函数声明。代码以以下内容开头：

```
Function *PrototypeAST::codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(TheContext));
  FunctionType *FT =
    FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false);

  Function *F =
    Function::Create(FT, Function::ExternalLinkage, Name, TheModule.get());
```

此代码将大量功能打包成几行。首先请注意，此函数返回 Function * 而不是 Value * 。因为“prototype”实际上是在谈论函数的外部接口（而不是表达式计算的值），所以有意义的是，它返回的是代码成时对应的LLVM函数。

调用 FunctionType::getcreateFunctionType 应该用于给定的原型。由于Kaleidoscope中的所有函参数均为double类型，因此第一行将创建一个 N 个LLVM double类型的向量。然后，它使用该Functontype::get方法创建一个函数类型，该函数类型将 N 个双精度值作为参数，并返回一个双精度值，不是vararg（假参数表明了这一点）。请注意，LLVM中的类型就像常量一样是唯一的，因此我们不“新建”一个类型，而是直接“获取”它。

上面的最后一行实际上创建了与原型相对应的IR函数。这表明要使用的类型，链接和名称，以及要插的模块。“外部链接”是指该功能可以在当前模块外部定义和/或可以由模块外部的函数调用。传入名称是用户指定的名称：由于指定了“TheModule”，因此该名称已注册在“TheModule”的符号中。

```
// Set names for all arguments.
unsigned Idx = 0;
for (auto &Arg : F->args())
  Arg.setName(Args[Idx++]);

return F;
```

最后，我们根据Prototype中提供的名称设置函数的每个参数的名称。此步骤不是绝对必要的，但是持名称的一致性可使IR更具可读性，并允许后续代码直接引用其名称的参数，而不必在Prototype AS中进行查找。

至此，我们有了一个没有主体的函数原型。这就是LLVM IR表示函数声明的方式。对于万花筒中的外陈述，这是我们需要做的。但是对于函数定义，我们需要代码生成并附加一个函数体。

```
Function *FunctionAST::codegen() {
   // First, check for an existing function from a previous 'extern' declaration.
  Function *TheFunction = TheModule->getFunction(Proto->getName());

  if (!TheFunction)
    TheFunction = Proto->codegen();

  if (!TheFunction)
    return nullptr;

  if (!TheFunction->empty())
    return (Function*)LogErrorV("Function cannot be redefined.");
```

对于函数定义，我们首先在TheModule的符号表中搜索该函数的现有版本（如果已经使用 extern 语创建了该版本）。如果 Module :: getFunction 返回null ，则不存在以前的版本，因此我们将从Proto ype中返回一个。无论哪种情况，我们都想在开始之前断言该函数为空（即没有主体）。

```
// Create a new basic block to start insertion into.
BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
Builder.SetInsertPoint(BB);

// Record the function arguments in the NamedValues map.
NamedValues.clear();
for (auto &Arg : TheFunction->args()
  NamedValues[Arg.getName()] = &Arg;
```

现在，我们开始进行 Builder 设置。第一行创建一个新的basic block（名为 entry ），将其插入TheF nction 。然后第二行告诉构建者，新指令应插入到新基本块的末尾。LLVM中的基本块是定义Control Flow Graph的功能的重要组成部分。由于我们没有任何控制流，因此我们的函数此时仅包含一个块我们将在第五章中解决此问题：）。

接下来，我们将函数参数添加到 NamedValues 映射中（首先将其清除后），以便 VariableExprAST 节点可以访问它们。

```
if (Value *RetVal = Body->codegen()) {
  // Finish off the function.
  Builder.CreateRet(RetVal);

  // Validate the generated code, checking for consistency.
  verifyFunction(*TheFunction);

  return TheFunction;
}
```

设置插入点并填充 NamedValues 映射后，我们将调用该 codegen() 方法作为函数的根表达式。如没有错误发生，它将发出代码以将表达式计算到输入块中，并返回计算出的值。假设没有错误，我们后创建LLVMret指令，以完成该功能。构建函数后，我们将调用LLVM提供的 verifyFunction，该函对生成的代码进行各种一致性检查，以确定我们的编译器是否在正确执行所有操作。使用它很重要：

可以捕获很多错误。函数完成并验证后，我们将其返回。

```
  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

这里剩下的唯一内容是错误情况的处理。为简单起见，我们仅通过删除使用该eraseFromParent 方生成的函数来处理此问题。这使用户可以重新定义以前错误输入的函数：如果我们不删除它，该函数与主体一起存在于符号表中，以防止将来重新定义。

但是，此代码确实存在一个漏洞：如果该 FunctionAST::codegen() 方法找到了现有的IR函数，则不根据定义自己的原型来验证其签名。这意味着较早的“外部”声明将优先于函数定义的签名，这可能致代码生成失败，例如，如果函数参数的命名不同。有多种方法可以修复此错误，请看你能想到些什！以下是一个测试用例：

```
extern foo(a);     # ok, defines foo.
def foo(b) b;      # Error: Unknown variable name. (decl using 'a' takes precedence).
```

驱动代码及思路总结

就目前而言，LLVM的代码生成并不能真正为我们带来很多好处，只是我们可以查看漂亮的IR调用。例代码将对代码生成的调用插入 HandleDefinition，HandleExtern 等函数中，然后转储LLVM IR。为查看LLVM IR的简单功能提供了一种好方法。例如：

```
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}
```

请注意解析器如何将顶级表达式转换为我们的匿名函数。在下一章中添加JIT支持时，这将非常方便。要注意，该代码是按字面意思转录的，除了IRBuilder进行的简单常量折叠外，没有执行任何优化。我将在下一章中显式添加优化。

```
ready> def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}
```

这显示了一些简单的算法。请注意，它与我们用来创建指令的LLVM构建器调用非常相似。

```
ready> def bar(a) foo(a, 4.0) + bar(31337);
Read function definition:
define double @bar(double %a) {
```

```
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}
```

这显示了一些函数调用。请注意，如果调用此函数，将花费很长时间执行。将来，我们将添加条件控流，以使递归真正有用。

```
ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> cos(1.234);
Read top-level expression:
define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

这显示了libm“cos”函数的外部，以及对其的调用。

```
ready> ^D
; ModuleID = 'my cool jit'

define double @0() {
entry:
  %addtmp = fadd double 4.000000e+00, 5.000000e+00
  ret double %addtmp
}

define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}

define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}

declare double @cos(double)
```

```
define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

退出当前演示时（通过在Linux上通过CTRL + D或在Windows上通过CTRL + Z和ENTER发送EOF）它将退出生成的整个模块的IR。在这里，我们可以看到具有相互参照的所有功能的全景图。

Kaleidoscope教程的第三章到此完成。接下来，我们将描述如何为此[添加JIT和Optimizer支持]，以我们实际上可以开始运行代码！

完整代码清单

这是我们正在运行的示例的完整代码清单，并通过LLVM代码生成器进行了增强处理。因为这使用了LVM库，所以我们需要将它们链接起来。为此，我们使用llvm-config工具通知makefile /命令行有关使用哪些选项的信息：

```
# Compile
clang++ -g -O3 chapter3-Code-generation-to-LLVM-IR.cpp `llvm-config --cxxflags --ldflags -system-libs --libs core` -o toy
# Run
./toy
```

以下是代码清单：

chapter3-Code-generation-to-LLVM-IR.cpp

参考: Kaleidoscope: Code generation to LLVM IR