Kaleidoscope 系列第九章：增加调试信息

作者：Hanseltu
原文链接：https://ld246.com/article/1570114253333
来源网站：链滴

原文链接:Kaleidoscope系列第九章：增加调试信息

本文是使用LLVM开发新语言Kaleidoscope教程系列第九章，将为Kaleidoscope添加调试信息，帮高效开发新语言。

第九章简介

欢迎来到“使用LLVM开发新语言Kaleidoscope教程”教程的第九章。在第一章至第八章中，我们构了一种带有函数和变量的体面的小型编程语言。这时会有一个问题，那就是如果出了问题怎么办，如调试程序?

源代码级调试使用格式化的数据，这些数据可帮助调试器将二进制文件和计算机的状态转换回程序员写的源代码。在LLVM中，我们通常使用一种称为DWARF的格式。DWARF是一种紧凑的编码，表示型，源位置和可变位置。

本章主要介绍为支持调试信息而必须添加到编程语言中的各种内容，以及如何将其转换为DWARF。

警告：目前我们无法通过JIT进行调试，因此我们需要将程序编译为小型且独立的程序。在此过程中，们将对语言的运行方式以及程序的编译方式进行一些修改。这意味着我们将拥有一个用Kaleidoscope写的简单程序而不是交互式JIT的源文件。确实存在一个限制，即我们一次只能有一个“顶层”命令来少必要的更改数量。

这是我们将要编译的示例程序：

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2);

fib(10)
```

为什么很难?

调试信息是一个棘手的问题，其原因有很多-大多围绕优化的代码。首先，优化使保持源位置更加困。在LLVM IR中，我们在指令上保留每个IR级别指令的原始源位置。优化过程应该保留新创建指令的位置，但是合并的指令只能保留一个位置-这可能会导致在逐步浏览优化程序时跳来跳去。其次，优可以以优化的方式移动变量，与其他变量在内存中共享或难以跟踪的方式移动变量。出于本教程的目，我们将避免优化（如我们将在下一组补丁中看到的那样）。

AOT编译模型

为了仅强调将调试信息添加到源语言中的各个方面，而不必担心JIT调试的复杂性，我们将对Kaleidoscope进行一些更改，以支持将前端发出的IR编译为一个简单的独立程序，你可以执行以下代码，调试查看结果。

首先，我们使包含顶层语句的匿名函数成为“ main”：

```
-    auto Proto = std::make_unique<PrototypeAST>("", std::vector<std::string>());
+    auto Proto = std::make_unique<PrototypeAST>("main", std::vector<std::string>());
```

只需更改名称即可。

然后，我们将删除存在的命令行代码：

```
@@ -1129,7 +1129,6 @@ static void HandleTopLevelExpression() {
 /// top ::= definition | external | expression | ';'
 static void MainLoop() {
   while (1) {
-    fprintf(stderr, "ready> ");
     switch (CurTok) {
     case tok_eof:
       return;
@@ -1184,7 +1183,6 @@ int main() {
   BinopPrecedence['*'] = 40; // highest.

   // Prime the first token.
-  fprintf(stderr, "ready> ");
   getNextToken();
```

最后，我们将禁用所有优化过程和JIT，以便在我们完成解析和生成代码后，唯一有区别的事情是LLVM IR变为标准错误：

```
@@ -1108,17 +1108,8 @@ static void HandleExtern() {
 static void HandleTopLevelExpression() {
   // Evaluate a top-level expression into an anonymous function.
   if (auto FnAST = ParseTopLevelExpr()) {
-    if (auto *FnIR = FnAST->codegen()) {
-      // We're just doing this to make sure it executes.
-      TheExecutionEngine->finalizeObject();
-      // JIT the function, returning a function pointer.
-      void *FPtr = TheExecutionEngine->getPointerToFunction(FnIR);
-
-      // Cast it to the right type (takes no arguments, returns a double) so we
-      // can call it as a native function.
-      double (*FP)() = (double (*)())(intptr_t)FPtr;
-      // Ignore the return value for this.
-      (void)FP;
+    if (!F->codegen()) {
+      fprintf(stderr, "Error generating code for top level expr");
     }
   } else {
     // Skip token for error recovery.
@@ -1439,11 +1459,11 @@ int main() {
   // target lays out data structures.
   TheModule->setDataLayout(TheExecutionEngine->getDataLayout());
   OurFPM.add(new DataLayoutPass());
+#if 0
   OurFPM.add(createBasicAliasAnalysisPass());
   // Promote allocas to registers.
   OurFPM.add(createPromoteMemoryToRegisterPass());
@@ -1218,7 +1210,7 @@ int main() {
   OurFPM.add(createGVNPass());
   // Simplify the control flow graph (deleting unreachable blocks, etc).
   OurFPM.add(createCFGSimplificationPass());

+  #endif
```

```
  OurFPM.doInitialization();

  // Set the global so the code gen can use this.
```

这些相对较小的更改使我们可以通过以下命令行将Kaleidoscope语言片段编译为可执行程序：

```
Kaleidoscope-Ch9 < fib.ks | & clang -x ir -
```

它在当前工作目录中给出a.out / a.exe。

编译单元

DWARF中一段代码的顶级容器是编译单元。它包含单个翻译单元的类型和功能数据（阅读：源代码一个文件）。因此，我们要做的第一件事是为fib.ks文件构造一个编译单元。

DWARF Emisson 设置

与 IRBuilder 该类相似，我们有一个DIBuilder类，该类有助于为LLVM IR文件构造调试元数据。IRBuider 与LLVM IR类似，它对应1：1，但名称更好。使用它确实需要您比使用DWARF术语更熟悉DAMR术语 IRBuilder 和 Instruction 名称，但是如果你通读了有关元数据格式的常规文档，则应该更加清。我们将使用此类构造所有的IR级别描述。它的构建需要一个模块，因此我们需要在构建模块后不久其进行构建。我们将其保留为全局静态变量，以使其易于使用。

接下来，我们将创建一个小容器来缓存一些常用数据。第一个是我们的编译单元，但是我们也将为我的一种类型编写一些代码，因为我们不必担心多种类型的表达式：

```
static DIBuilder *DBuilder;

struct DebugInfo {
  DICompileUnit *TheCU;
  DIType *DblTy;

  DIType *getDoubleTy();
} KSDbgInfo;

DIType *DebugInfo::getDoubleTy() {
  if (DblTy)
    return DblTy;

  DblTy = DBuilder->createBasicType("double", 64, dwarf::DW_ATE_float);
  return DblTy;
}
```

然后在构建模块 main 时：

```
DBuilder = new DIBuilder(*TheModule);

KSDbgInfo.TheCU = DBuilder->createCompileUnit(
    dwarf::DW_LANG_C, DBuilder->createFile("fib.ks", "."),
    "Kaleidoscope Compiler", 0, "", 0);
```

这里有几件事要注意。首先，当我们为称为Kaleidoscope的语言生成编译单元时，我们将语言常量用C。这是因为调试器不一定会理解其无法识别的语言的调用约定或默认ABI，因此我们遵循LLVM代码

成中的C ABI，因此它是最接近准确的东西。这确保了我们实际上可以从调试器中调用函数并使它们行。其次，您会在的调用中看到“ fib.ks” createCompileUnit。这是默认的硬编码值，因为我们使外壳重定向将源代码放入Kaleidoscope编译器中。在通常的前端中，通常有一个输入文件名，它将输该文件名。

通过DIBuilder发出调试信息的最后一件事是，我们需要“完成”调试信息。原因是DIBuilder的基础AI的一部分，但请确保在main末尾附近，在转储模块之前执行此操作：

```
DBuilder->finalize();
```

函数

现在我们有了 CompileUnit 和我们的源位置，可以将函数定义添加到调试信息中。因此，我们在 ProotypeAST::codegen() 中添加了几行代码来描述子程序的上下文（在本例中为“文件”）以及函数本的实际定义。

所以上下文：

```
DIFile *Unit = DBuilder->createFile(KSDbgInfo.TheCU.getFilename(),
                                    KSDbgInfo.TheCU.getDirectory());
```

给我们一个DIFile并询问我们上面创建 CompileUnit 的当前目录和文件名。然后，现在，我们使用一源位置0（因为AST当前没有源位置信息）并构造函数定义：

```
DIScope *FContext = Unit;
unsigned LineNo = 0;
unsigned ScopeLine = 0;
DISubprogram *SP = DBuilder->createFunction(
    FContext, P.getName(), StringRef(), Unit, LineNo,
    CreateFunctionType(TheFunction->arg_size(), Unit),
    false /* internal linkage */, true /* definition */, ScopeLine,
    DINode::FlagPrototyped, false);
TheFunction->setSubprogram(SP);
```

现在我们有了一个DISubprogram，其中包含对该函数所有元数据的引用。

源代码位置

调试信息最重要的是准确的源代码位置-这使我们可以将源代码映射回去。但是，我们有一个问题，Kaeidoscope确实在词法分析器或解析器中没有任何源位置信息，因此我们需要添加它。

```
struct SourceLocation {
  int Line;
  int Col;
};
static SourceLocation CurLoc;
static SourceLocation LexLoc = {1, 0};

static int advance() {
  int LastChar = getchar();

  if (LastChar == '\n' || LastChar == '\r') {
    LexLoc.Line++;
```

```
    LexLoc.Col = 0;
  } else
    LexLoc.Col++;
  return LastChar;
}
```

在这组代码中，我们添加了一些有关如何跟踪“源文件”的行和列的功能。在对每个标记进行词法分时，我们将当前当前的“词法位置”设置为标记词开头的各种行和列。为此 getchar()，我们使用新的dvance()跟踪信息的方法来覆盖以前的所有调用，然后将所有位置添加到所有AST类中：

```
class ExprAST {
  SourceLocation Loc;

  public:
    ExprAST(SourceLocation Loc = CurLoc) : Loc(Loc) {}
    virtual ~ExprAST() {}
    virtual Value* codegen() = 0;
    int getLine() const { return Loc.Line; }
    int getCol() const { return Loc.Col; }
    virtual raw_ostream &dump(raw_ostream &out, int ind) {
      return out << ':' << getLine() << ':' << getCol() << '\n';
    }
```

当我们创建一个新的表达式时，我们会忽略：

```
LHS = std::make_unique<BinaryExprAST>(BinLoc, BinOp, std::move(LHS),
                                      std::move(RHS));
```

给我们每个表达式和变量的位置。

为了确保每条指令都能获得正确的源位置信息，我们必须告诉 Builder 我们何时位于新的源位置。我为此使用一个小的辅助函数：

```
void DebugInfo::emitLocation(ExprAST *AST) {
  DIScope *Scope;
  if (LexicalBlocks.empty())
    Scope = TheCU;
  else
    Scope = LexicalBlocks.back();
  Builder.SetCurrentDebugLocation(
      DebugLoc::get(AST->getLine(), AST->getCol(), Scope));
}
```

这既可以告诉IRBuilder我们主要的位置，也可以告诉我们所处的作用域。该作用域可以在编译单元级，也可以是与当前函数类似的最接近的词法块。为了表示这一点，我们创建了一个范围堆栈：

```
std::vector<DIScope *> LexicalBlocks;
```

当我们开始为每个函数生成代码时，将作用域（函数）推到栈顶：

```
KSDbgInfo.LexicalBlocks.push_back(SP);
```

同样，我们可能不会忘记在函数的代码生成结束时将范围弹出弹出范围堆栈：

```
// Pop off the lexical block for the function since we added it
```

```
// unconditionally.
KSDbgInfo.LexicalBlocks.pop_back();
```

然后，确保每次开始为新的AST对象生成代码时都发出该位置：

```
KSDbgInfo.emitLocation(this);
```

变量

现在我们有了函数，我们需要能够打印出范围内的变量。我们首先设置函数参数，以便获得不错的回并查看如何调用函数。实现它不需要很多代码，并且通常只需要在创建中的allocas参数时处理 FunctinAST::codegen。

```
// Record the function arguments in the NamedValues map.
NamedValues.clear();
unsigned ArgIdx = 0;
for (auto &Arg : TheFunction->args()) {
  // Create an alloca for this variable.
  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());

  // Create a debug descriptor for the variable.
  DILocalVariable *D = DBuilder->createParameterVariable(
      SP, Arg.getName(), ++ArgIdx, Unit, LineNo, KSDbgInfo.getDoubleTy(),
      true);

  DBuilder->insertDeclare(Alloca, D, DBuilder->createExpression(),
                          DebugLoc::get(LineNo, 0, SP),
                          Builder.GetInsertBlock());

  // Store the initial value into the alloca.
  Builder.CreateStore(&Arg, Alloca);

  // Add arguments to variable symbol table.
  NamedValues[Arg.getName()] = Alloca;
}
```

在这里，我们首先创建变量，并为其指定范围（SP），名称，源位置，类型，并且由于它是一个自变，所以也提供了自变量索引。接下来，我们创建一个lvm.dbg.declare调用以在IR级别指示我们已在allca中获得了一个变量（并为变量提供了起始位置），并为声明的作用域起点设置了源位置。

此时要注意的一件事是，各种调试器都基于过去为它们生成代码和调试信息的方式进行了假设。在这情况下，我们需要做一些改动，以避免为函数序言生成行信息，以便调试器知道在设置断点时跳过那指令。因此，在我们在FunctionAST::CodeGen 中添加了更多行：

```
// Unset the location for the prologue emission (leading instructions with no
// location in a function are considered part of the prologue and the debugger
// will run past them when breaking on a function)
KSDbgInfo.emitLocation(nullptr);
```

然后当我们实际开始为函数主体生成代码时生成一个新位置：

```
KSDbgInfo.emitLocation(Body.get());
```

这样，我们就有足够的调试信息来设置函数中的断点，打印出参数变量和调用函数。只需几行简单的

码就可以调试的不错了！

完整代码清单

这是我们正在运行的示例的完整代码清单，其中增加了调试信息。想要运行并演示此示例，请使用以命令：

```
# Compile
clang++ -g chapter9-Adding-Debug-Information.cpp `llvm-config --cxxflags --ldflags --syst
m-libs --libs all mcjit native` -O3 -o toy
# Run
./toy
```

以下完整代码清单：

chapter9-Adding-Debug-Information.cpp

参考: Kaleidoscope: Adding Debug Information