Debug Info Tutorial

Eric Christopher (echristo@gmail.com), David Blaikie (dblaikie@gmail.com)

What is debug info?

Mapping of source to binary

Program structure, lines, columns, variables

Debuggers, Profiling, Editors

Kaleidoscope Background

What is it?

http://llvm.org/docs/tutorial/

How does it work?

Jitting and top level statements

Kaleidoscope Background

Example program:

```
 1) def ret1()
 2)   1
 3)
 4) def fib(x)
 5)   if x < 3 then
 6)     ret1()
 7)   else
 8)     fib(x-1)+fib(x-2);
 9)
10) fib(10)
```

Kaleidoscope Background - Changes

JIT disabled

Optimization passes disabled

Remove helpful prompts and status

```
examples/Kaleidoscope-Ch8 < fib.ks | & clang -x ir - -o fib -g
```

What is DWARF then?

Standard for how to encode program structure, lines, columns, variables

Permissive standard with vendor extensions

Probably a bit too permissive

DWARF Vaguaries

DWARF consumers not generalized because they've only seen output from one tool (GDB works great with GCC's DWARF output)

is_stmt - a line table feature "indicating that the current instruction is a recommended breakpoint location" - omitting this caused GDB to do… strange things.

DWARF Structure

debug_info: source construct description (functions, types, namespaces, etc)

debug_line: instruction -> source line mapping

Use `llvm-dwarfdump` and similar tools to examine this data in object and executable files.

Bunch of other implementation details probably not worth discussing on a first pass: debug_loc, debug_ranges, debug_string, debug_abbrev, etc.

DWARF Structure - Info

Hierarchical tag+attribute format, imagine something like a binary XML.

```
0x0000004c: TAG_subprogram [3] *
              AT_name( "fib" )
              AT_decl_file( "/tmp/fib.ks" )
              AT_decl_line( 4 )
              AT_type( {0x00000097} ( double ) )
              AT_external( 0x01 )

0x0000006a:   TAG_formal_parameter [4]
                AT_location( fbreg -8 )
                AT_name( "x" )
                AT_decl_file( "/Users/echristo/tmp/fib.ks" )
                AT_decl_line( 4 )
                AT_type( {0x00000097} ( double ) )
```

DWARF Structure - Line Table

State machine & all that, but if you're just reading it it's fairly simple

```
                Dir  File Name
                ---- ---------
file_names[1]     0 fib.ks

Addr Line File Flags
---- ---- ---- -------------
0x00    1    1
0x07    2    1 prologue_end
0x0e    3    1
0x18    3    1 end_sequence
```

DIBuilder Background

The debug info equivalent of IRBuilder - a helper API for building LLVM IR metadata for debug info

```
DIBuilder DBuilder(MyModule);
DICompileUnit CU = DBuilder.createCompileUnit(“foo.ks”, …);
DISubprogram SP = DBuilder.createFunction(“func”, …);
…
```

Module level setup

Add a module flag with the debug info version!

```
TheModule->addModuleFlag(Module::Warning, "Debug Info Version”,
                         DEBUG_METADATA_VERSION)
```

Not all targets support all dwarf!

```
// Darwin only supports dwarf2.
if (Triple(sys::getProcessTriple()).isOSDarwin())
  TheModule->addModuleFlag(llvm::Module::Warning,
                           "Dwarf Version", 2)
```

Kaleidoscope Dwarf Additions

Language Name - DW_LANG_KS

Couple of simple places to put it in LLVM.

No vendor range for this so use with caution.

Creating the Compile Unit - Code

```
static DIBuilder *DBuilder;

struct DebugInfo {
  DICompileUnit TheCU;
} KSDbgInfo;

DBuilder = new DIBuilder(*TheModule);
KSDbgInfo.TheCU = DBuilder->createCompileUnit(dwarf::DW_LANG_KS,
"fib.ks", ".", "Kaleidoscope Compiler", 0, "", 0);
```

Creating the Compile Unit - Metadata

```
!2 = metadata !{metadata
    !"0x11\0032770\00Kaleidoscope Compiler\000\00\000\00\001",
  metadata !3, metadata !4, metadata !4, metadata !5,
  metadata !4, metadata !4} ;
    [ DW_TAG_compile_unit ] [./fib.ks] [DW_LANG_KS]
```

Creating the Compile Unit - DWARF

```
0x0000000b: TAG_compile_unit [1] *
              AT_producer( "Kaleidoscope Compiler" )
              AT_language( DW_LANG_KS )
              AT_name( "fib.ks" )
              AT_stmt_list( 0x00000000 )
              AT_comp_dir( "." )
```

Types - Code

```
struct DebugInfo {
  DICompileUnit TheCU;
  DIType DblTy;
  DIType getDoubleTy();
} KSDbgInfo;


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

Types - Metadata

`!3 = {…} ; [ DW_TAG_base_type ] [double]` [line 0, size 64, align 64, offset 0, enc DW_ATE_float]

Types - DWARF

```
0x00000097: TAG_base_type [5]
              AT_name( "double" )
              AT_encoding( DW_ATE_float )
              AT_byte_size( 0x08 )
```

Functions - Code

```
static DICompositeType CreateFunctionType(unsigned NumArgs, DIFile Unit) {
  SmallVector<Value *, 8> EltTys;
  DIType DblTy = KSDbgInfo.getDoubleTy();

  // Add the result type.
  EltTys.push_back(DblTy);

  for (unsigned i = 0, e = NumArgs; i != e; ++i)
    EltTys.push_back(DblTy);

  DITypeArray EltTypeArray = DBuilder->getOrCreateTypeArray(EltTys);
  return DBuilder->createSubroutineType(Unit, EltTypeArray);
}
```

Functions - Code

```
DIFile Unit = DBuilder->createFile(KSDbgInfo.TheCU.getFilename(),
                                   KSDbgInfo.TheCU.getDirectory());
DIDescriptor FContext(Unit);
unsigned LineNo = 0;
unsigned ScopeLine = 0;
DISubprogram SP = DBuilder->createFunction(
      FContext, Name, StringRef(), Unit, LineNo,
      CreateFunctionType(Args.size(), Unit), false /* internal linkage */,
      true /* definition */, ScopeLine, DIDescriptor::FlagPrototyped, false, F);
```

Functions - Metadata

```
; [ DW_TAG_subprogram ]
!0 = {“…fib…”, …, double (double)* @fib, …, !1}
!1 = {…, !2, …} ; [ DW_TAG_subroutine_type ]
!2 = {!3, !3} ; ret type, parameter type
!3 = {…} ; [ DW_TAG_base_type ] [double]
```

Functions - DWARF

```
0x0000004c: TAG_subprogram [3] *
              AT_name( "fib" )
              AT_decl_file( "/tmp/fib.ks" )
              AT_decl_line( 4 )
              AT_type( {0x00000097} ( double ) )
              AT_external( 0x01 )
```

Line Information - A Digression

Build source location information into your front end!

Build AST and Lexer dumping mechanisms!

Line Information - Code

```
struct DebugInfo {
  DICompileUnit TheCU;
  DIType DblTy;
  std::vector<DIScope *> LexicalBlocks;
  std::map<const PrototypeAST *, DIScope> FnScopeMap;

  void emitLocation(ExprAST *AST);
  DIType getDoubleTy();
} KSDbgInfo;
```

Line Information - Metadata

```
%res = call double @ret1(), !dbg !4
…
!4 = {i32 6, i32 5, !0, null} ; in function !0 at line 6, col 5
```

Optimizations know how to propagate this information when transforming instructions.

Line Information - DWARF

```
 0x0000004c: TAG_subprogram [3] *
>              AT_low_pc( 0x0000000100000ec0 )
>              AT_high_pc( 0x0000000100000f52 )
>              AT_frame_base( rbp )
               AT_name( "fib" )
               AT_decl_file( "/tmp/fib.ks" )
               AT_decl_line( 4 )
               AT_type( {0x00000097} ( double ) )
               AT_external( 0x01 )
```

Line Information - DWARF

```
 Address            Line   Column File   ISA Discriminator Flags
 ------------------ ------ ------ ------ --- ------------- -------------
[0x0000000100000eb0      1      0      1   0             0  is_stmt
|0x0000000100000ebc      2      3      1   0             0  is_stmt prologue_end
)0x0000000100000ebe      2      3      1   0             0  is_stmt end_sequence
[0x0000000100000ec0      4      0      1   0             0  is_stmt
|0x0000000100000ecb      5      3      1   0             0  is_stmt prologue_end
|0x0000000100000ed8      5     10      1   0             0  is_stmt
|0x0000000100000ef3      6      5      1   0             0  is_stmt
|0x0000000100000f0a      8      9      1   0             0  is_stmt
|0x0000000100000f0f      8     11      1   0             0  is_stmt
|0x0000000100000f23      8     18      1   0             0  is_stmt
|0x0000000100000f28      8     20      1   0             0  is_stmt
)0x0000000100000f52      8     20      1   0             0  is_stmt end_sequence
[0x0000000100000f60     10      0      1   0             0  is_stmt
|0x0000000100000f6c     10      5      1   0             0  is_stmt prologue_end
)0x0000000100000f73     10      5      1   0             0  is_stmt end_sequence
ret1, fib, main
```

Variables - Code

```
DIScope *Scope = KSDbgInfo.LexicalBlocks.back();
DIFile Unit = DBuilder->createFile(KSDbgInfo.TheCU.getFilename(),
                                   KSDbgInfo.TheCU.getDirectory());
DIVariable D = DBuilder->createLocalVariable(dwarf::DW_TAG_arg_variable,
                                             *Scope, Args[Idx], Unit, Line,
                                             KSDbgInfo.getDoubleTy(), Idx);

Instruction *Call = DBuilder->insertDeclare(
    Alloca, D, DBuilder->createExpression(), Builder.GetInsertBlock());
Call->setDebugLoc(DebugLoc::get(Line, 0, *Scope));
```

Variables - A Bit More Code

```
// 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);
```

```
+#if 0
   // Provide basic AliasAnalysis support for GVN.
   OurFPM.add(createBasicAliasAnalysisPass());
   // Promote allocas to registers.
   OurFPM.add(createPromoteMemoryToRegisterPass());
```

Variables - Metadata

Only works well with allocas, some limited support when in reg.

```
%x.addr = alloca double
store double %d, double* %x.adr
call void @llvm.dbg.declare(metadata !{double* %x.addr},
    metadata !5, …), !dbg !14
…
!5 = {“…x…”, !0, …, !3} ; [ DW_TAG_arg_variable ] [x]
```

Variables - DWARF

```
0x0000004c: TAG_subprogram [3] *
              AT_low_pc( 0x0000000100000ec0 )
              AT_high_pc( 0x0000000100000f52 )
              AT_frame_base( rbp )
              AT_name( "fib" )
              AT_decl_file( "/tmp/fib.ks" )
              AT_decl_line( 4 )
              AT_type( {0x00000097} ( double ) )
              AT_external( 0x01 )

0x0000006a:   TAG_formal_parameter [4]
                AT_location( fbreg -8 )
                AT_name( "x" )
                AT_decl_file( "/Users/echristo/tmp/fib.ks" )
                AT_decl_line( 4 )
                AT_type( {0x00000097} ( double ) )
```

Inlining

you are investigating issues in inlining debug info quality

you want to simulate inlining in your front end's codegen

you're implementing a custom inlining pass in LLVM

```
store double 1.0e0, double* %x.adr, !dbg !6
…
!6 = {i32 8, i32 3, !7, !4} ; in function !7 at line 2, col 3
                            ; inlined at !4
!7 = {"…ret1…", …}
```

Inlining DWARF

Allows debuggers to simulate distinct function calls, even after they've been inlined.

```
0x00000042: TAG_subprogram [3]
              AT_name( "ret1" )
              ...
0x00000052: TAG_subprogram [4] *
              AT_name( "fib" )
              ...
0x00000062:   TAG_inlined_subroutine [5]
                AT_abstract_origin( {0x00000042} ( "ret1" ) )
                AT_low_pc( 0x000000010000005f )
                AT_high_pc( 0x0000000100000064 )
                AT_call_file( "/tmp/fib.ks" )
                AT_call_line( 6 )
```

Debugger Support

DW_LANG_KS isn’t supported by any debugger…

… but DW_LANG_C is!

Questions?