CS615: Formal Specification and Verification of Programs 2019

Lecture 7: LLVM infrastructure

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2019-08-28

LLVM : compiler backend

LLVM is a compiler backend.

- Input: IR bytecode
- Actions: mostly compiler optimizations
- Outputs: machine code, transformed IR, etc

Supported by a large consortium of research groups, developers, and companies.

LLVM is not a frontend

By definition, LLVM does not include compiler frontend

- Input parsing
- AST generation
- ..etc..

Clang is the frontend written on top of LLVM to parse C++ programs.

In theory, you can have Jlang to parse Java programs.

Installing llvm

Get the following packages

- `cmake`
- `llvm-8-dev`
- `clang-8`
- `clang-format-8`
- `clang-tidy-8`
- `clang-tools-8`
- `libclang-8-dev`
- `xdot`

```
sudo apt-get install cmake llvm-8-dev clang-8 clang-format-8 clang-tidy-8 clang-tools-8 libclang-8-dev xdot
```

Where is llvm installed?

`llvm-config-8` is a utility that tells us where is what

- `llvm-config-8 --includedir`
  - location of header files
- `llvm-config-8 --libdir`
  - location of object files
- `llvm-config-8 --libs`
  - object files to link

Cmake and LLVM

Cmake is a build system.

LLVM provided scripts for Cmake for importing the library.

Cmake script to create Makefile

```
find_package(LLVM 8 REQUIRED CONFIG)
find_package(Clang REQUIRED CONFIG
    HINTS "/usr/lib/llvm-8/lib/cmake/clang")

include_directories(\${LLVM_INCLUDE_DIRS})
include_directories(\${CLANG_INCLUDE_DIRS})

add_executable(llvm-demo llvm-demo.cpp)
target_link_libraries(llvm-demo
        clangDaemon clangFrontendTool)
```

Helper Makefile to call Cmake!!

```
BUILDDIR = $(PWD)/build
SRCDIR = $(PWD)/

all :  $(BUILDDIR)/Makefile
       +make -C $(BUILDDIR)
       cp -f $(BUILDDIR)/llvm-demo llvm-demo

$(BUILDDIR)/Makefile:
       mkdir -p $(BUILDDIR)
       cd $(BUILDDIR)/; cmake \$(SRCDIR)
```

Directory structure so far

```
llvm-demo/Makefile              # helper make file
llvm-demo/CMakeLists.txt        # Makefile generator
llvm-demo/llvm-demo.cpp         # source code
llvm-demo/example/              # input programs
```

Inside `llvm-demo.cpp` : loading C++ programs

```
// included llvm/clang files
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
..
..
#include "clang/Basic/DiagnosticOptions.h"
#include "clang/Basic/TargetInfo.h"

std::unique_ptr<llvm::Module>
c2ir( string in_cpp ) {

  // list of options for the compiler
  std ::vector<const char *> args;
  args.push_back( "-emit-llvm" );
  args.push_back( ..... ); // many other options
  args.push_back( "-O1" );
  args.push_back( in_cppt.c_str() );
```

Loading C++ programs II

```
//allocate compiler context
clang::CompilerInstance Clang;
Clang.createDiagnostics();

//allocate compiler invocation context
std::shared_ptr<clang::CompilerInvocation>
      CI(new clang::CompilerInvocation());

// process compiler options
clang::CompilerInvocation::CreateFromArgs(*CI.get(),
                    &args[0], &args[0] + args.size(),
                    Clang.getDiagnostics());

//tell compiler about the invocation plan
Clang.setInvocation(CI);
```

Loading C++ programs III

```
  // setup llvm context
  llvm::LLVMContext llvm_ctx;

  // what kind of compilation we want
  clang::CodeGenAction *Act =
       new clang::EmitLLVMOnlyAction(&llvm_ctx);

  // invoke compiler to obtain LLVM IR
  if (!Clang.ExecuteAction(*Act)) return nullptr;

  // get module out of action
  std::unique_ptr<llvm::Module> m =Act->takeModule();

  return std::move(m);
}
```

Printing objects

Every object in LLVM can be printed.

For example

```
m->print( llvm::outs(), nullptr );
```

Uncomment the above instruction in file `llvm-demo.cpp`

If you have compiled LLVM in debug mode you can also call

```
m->dump();
```

Compile

Give command `make` to compile the code.

Example: running

Input `example/t1.cpp`

```
int add( int x, int y ) {
  x = x + y;
  return x;
}
```

Run

```
./llvm-demo example/t1.cpp
```

We will look at the output in a minute!

Passes

In LLVM, any action on the modules are done using passes

We can apply standard passes on the modules or apply our own passes.

Let us look an application of passes.

```
void prepare( std::unique_ptr<llvm::Module>& m ) {

 llvm::legacy::PassManager pm;
 pm.add( llvm::createPromoteMemoryToRegisterPass());
 pm.add( llvm::createCFGPrinterLegacyPassPass() );
 pm.run( *m.get() );

}
```

Example: printed module

```
; ModuleID = './example/t1.cpp'
source_filename = "./example/t1.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:6
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @_Z3addii(i32 %x, i32 %y) #0 !dbg !7 {

  call void @llvm.dbg.value(metadata i32 %x,
                 metadata !13,
                 metadata !DIExpression()), !dbg !15

  call void @llvm.dbg.value(metadata i32 %y,
                 metadata !14,
                 metadata !DIExpression()), !dbg !16
```

"!" is a reference

- preamble of module, declaration of function `@_Z3addii`, and debug info for the parameters as function calls.

Example: printed module

```
  % add = add nsw i32 %x, %y, !dbg !17

  call void @llvm.dbg.value(metadata i32 %add,
                metadata !13,
                metadata !DIExpression()), !dbg !15

  ret i32 %add, !dbg !18
}
```

- addition instruction and
- the debug information for the value %add

Example: printed module

Dummy debug info function calls are declared

```
; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata, metadata, metadata) #1

attributes #0 = { nounwind
  "correctly-rounded-divide-sqrt-fp-math"="false"
  "disable-tail-calls"="false" "less-precise-fpmad"="false"
  "no-frame-pointer-elim"="true"
  "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false"
  "no-jump-tables"="false" "no-nans-fp-math"="false"
  "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
  "stack-protector-buffer-size"="8"
  "target-features"="+mmx,+sse,+sse2,+x87"
  "unsafe-fp-math"="false" "use-soft-float"="false" }

attributes #1 = { nounwind readnone speculatable }
```

Example: printed module (compilation unit info)

```
!llvm.dbg.cu = !{!0}
!llvm.module.flags = !{!3, !4, !5}
!llvm.ident = !{!6}

!0 = distinct !DICompileUnit(language: DW_LANG_C_plus_plus,
      file: !1,
      producer: "clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/
      isOptimized: true, runtimeVersion: 0,
      emissionKind: FullDebug, enums: !2)
!1 = !DIFile(filename: "./example/<stdin>",
       directory: "<path-to-demo-foldor>/llvm-demo")
!2 = !{}

!3 = !{i32 2, !"Dwarf Version", i32 2}
!4 = !{i32 2, !"Debug Info Version", i32 3}
!5 = !{i32 1, !"wchar_size", i32 4}
!6 = !{!"clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"}
```

Example: printed module (debug information function declaration)

```
!7 = distinct !DISubprogram(name: "add",linkageName:"_Z3addii",
      scope: !8, file: !8, line: 1, type: !9, isLocal: false,
      isDefinition: true, scopeLine: 1,flags: DIFlagPrototyped,
      isOptimized: true, unit: !0, variables: !12)

!8 = !DIFile(filename: "./example/t1.cpp",
       directory: "<path-to-folder>/llvm-demo")

!9 = !DISubroutineType(types: !10)
!10 = !{!11, !11, !11}
!11 = !DIBasicType(name: "int", size:32,encoding:DW_ATE_signed)

!12 = !{!13, !14}
!13 = !DILocalVariable(name: "x", arg: 1, scope: !7, file: !8,
                       line: 1, type: !11)
!14 = !DILocalVariable(name: "y", arg: 2, scope: !7, file: !8,
                       line: 1, type: !11)
```

Example: printed module (location information)

Every instruction in annotated with the following debug information.

```
!15 = !DILocation(line: 1, column: 14, scope: !7)
!16 = !DILocation(line: 1, column: 21, scope: !7)
!17 = !DILocation(line: 2, column: 9, scope: !7)
!18 = !DILocation(line: 3, column: 3, scope: !7)
```

LLVM data structure

- `llvm::Module` is top level object
- `llvm::Module` has list of `llvm::Function`s and global objects
- `llvm::Function` has list of `llvm::BasicBlock`s
- `llvm::BasicBlock` contains `llvm::Instruction`s

Besides Module, everyone else is inherited from `llvm::Value`

Writing our own pass

```
class my_first_pass : public llvm::FunctionPass {
  public:
  static char ID;
  my_first_pass() : llvm::FunctionPass(ID) {}

  bool runOnFunction(llvm::Function &f) {
    std::string name = f.getName();
    std::cout << name << "\n";
  }
  void getAnalysisUsage(llvm::AnalysisUsage &au) const {
    au.setPreservesAll();
  }
  llvm::StringRef getPassName() const {
    return "my first pass!";
  }
};
char my_first_pass::ID = 0;
```

Running our own pass

```
void run_my_pass( std::unique_ptr<llvm::Module>& m ) {
  llvm::legacy::PassManager pm;
  pm.add( new my_first_pass() );
  pm.run( *m.get() );
}
```

Doing something interesting in the pass

```
bool runOnFunction(llvm::Function &f) {
  for( const llvm::BasicBlock& b : f ) {
    for( const llvm::Instruction& Iobj : b.getInstList() ) {
      const llvm::Instruction* I = &(Iobj);
      if(auto bop = llvm::dyn_cast<llvm::BinaryOperator>(I) ) {
        auto op0 = bop->getOperand( 0 );
        auto op1 = bop->getOperand( 1 );
        if( bop->getOpcode() == llvm::Instruction::Add ) {
          std::cout << "Add instruction found: \n";
          bop->print( llvm::outs() );
          std::cout << "\ninput 0 : \n";
          op0->print( llvm::outs() );
          std::cout << "\ninput 1 : \n";
          op1->print( llvm::outs() );
          std::cout << "\n";
        }
      }
    }
  }
  return false;
}
```

Example: running input with if

Input example/t2.cpp

```
int add( int x, int y) {
  if( x > 0 ){
    x = x + y;
  }else{
    x = x + x;
  }
  return x + 3;
}
```

Run

```
./llvm-demo example/t2.cpp
```

Understanding blocks

Let us look at the return block for `example/t2.cpp`.

Block id

reference to previous blocks

```
; <label>:3:                                     ; preds = %2, %1
  % x.addr.0 = phi i32 [ %add, %1 ], [ %add1, %2 ]

  call void @llvm.dbg.value(metadata i32 %x.addr.0,
                metadata !13,
                metadata !DIExpression()), !dbg !15

  % add2 = add nsw i32 %x.addr.0, 3, !dbg !25

  ret i32 %add2, !dbg !26
```

Blocks have three parts

- Preamble with phi nodes
- block content
- one of termination instructions

Exercise 7.1

Why there is no intrinsic debug call for value `% add2`?

End of Lecture 7