CS451 – Software Analysis

Lecture 20

LLVM IR Generation

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

LLVM IR

- The LLVM project has designed an intermediate representation form that can be manipulated by several tools
- From the LLVM IR you can deliver machine code for different architectures
- At the LLVM IR you can perform several optimizations
- You can create tools to generate LLVM IR

Example

- How can we programmatically generate a function in LLVM IR?

```
; ModuleID = 'mul_add.bc'
source_filename = "mul_add"

define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

Headers and set up

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/PassManager.h"
#include "llvm/IR/CallingConv.h"
#include "llvm/IR/Verifier.h"
#include "llvm/IR/IRPrintingPasses.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Bitcode/BitcodeWriter.h"
#include <stdio.h>

using namespace llvm;

Module *makeLLVMModule(LLVMContext &Context);
```

Main functionality

```
int main(int argc, char **argv) {

  LLVMContext Context;
  Module *Mod = makeLLVMModule(Context);

  raw_fd_ostream r(fileno(stdout), false);
  verifyModule(*Mod, &r);

  // Print the produced IR

  // Write IR to a bitcode file

  delete Mod;

  return 0;
}
```

Print the produced IR

```
ModulePass *m = createPrintModulePass(outs(),
                                      "Module IR printer");

legacy::PassManager PM;
PM.add(m);
PM.run(*Mod);
```

Write IR to a file

```
FILE* mul_add_file = fopen("mul_add.bc", "w+");

raw_fd_ostream bitcodeWriter(fileno(mul_add_file), true);

WriteBitcodeToFile(*Mod, bitcodeWriter);
```

How to build the IR

- All the IR is constructed in `makeLLVMModule()`
- Recall that LLVM IR is composed by modules
  - Modules contain functions
  - Functions contain basic blocks
  - Basic blocks contain instructions

makeLLVMModule()

```
Module *makeLLVMModule(LLVMContext &Context) {
    Module *mod = new Module("mul_add", Context);

    /* Build IR  */

     return mod;
}
```

Create the function

- getOrInsertFunction() needs the type of the return value and the types of the three input arguments

```
FunctionCallee mul_add_fun =
    mod->getOrInsertFunction("mul_add",
        Type::getInt32Ty(Context),
        Type::getInt32Ty(Context),
        Type::getInt32Ty(Context),
        Type::getInt32Ty(Context));

Function *mul_add = cast<Function>(mul_add_fun.getCallee());
```

Calling conventions and arguments

- We can also give specific names in the arguments of the function

```
mul_add->setCallingConv(CallingConv::C);
Function::arg_iterator args = mul_add->arg_begin();

Value *x = args++;
x->setName("x");

Value *y = args++;
y->setName("y");

Value *z = args++;
z->setName("z");
```

Basic block of the function

```
BasicBlock *block =
    BasicBlock::Create(Context, "entry", mul_add);

IRBuilder<> builder(block);

Value *tmp = builder.CreateBinOp(Instruction::Mul,
                                 x, y, "tmp");
Value *tmp2 = builder.CreateBinOp(Instruction::Add,
                                  tmp, z, "tmp2");

builder.CreateRet(tmp2);
```

Compile and run

```
$ clang++ funcGenerator.cpp `../../bin/llvm-config
    --cxxflags --ldflags
    --libs core BitWriter
    --system-libs` -o funcGenerator
$ ./funcGenerator
Module IR printer
; ModuleID = 'mul_add'
source_filename = "mul_add"


define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```